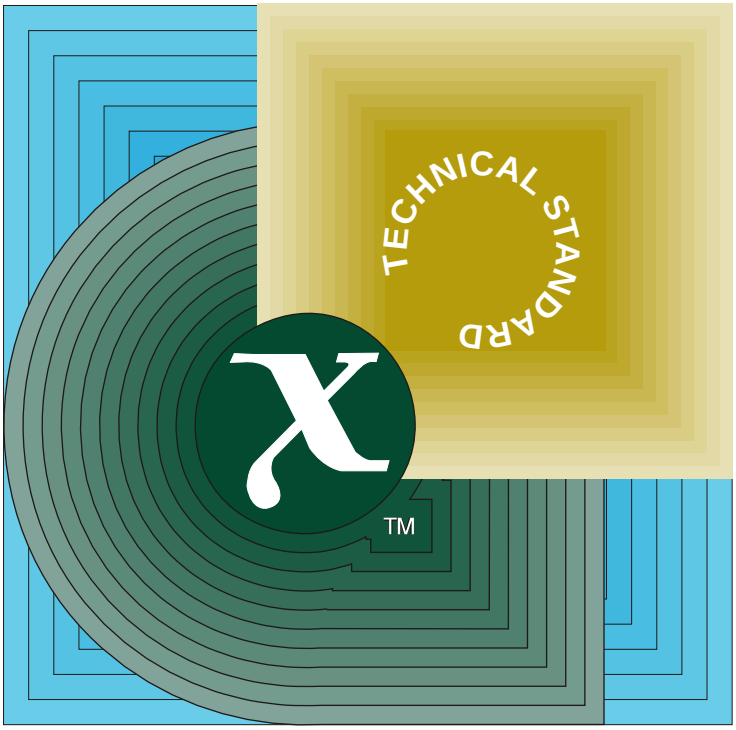
Technical Standard

DCE 1.1:
Authentication and Security Services



THE Open GROUP

[This page intentionally left blank]

CAE Specification

DCE 1.1: Authentication and Security Services

The Open Group

CAE Specification

DCE 1.1: Authentication and Security Services

Document Number: C311

Published by The Open Group, U.K.

Any comments relating to the material contained in this document may be submitted to The Open Group at:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

List of Figures

List of Tables

Preface

The Open Group

The Open Group is the leading vendor-neutral, international consortium for buyers and suppliers of technology. Its mission is to cause the development of a viable global information infrastructure that is ubiquitous, trusted, reliable, and as easy-to-use as the telephone. The essential functionality embedded in this infrastructure is what we term the *IT DialTone*. The Open Group creates an environment where all elements involved in technology development can cooperate to deliver less costly and more flexible IT solutions.

Formed in 1996 by the merger of the X/Open Company Ltd. (founded in 1984) and the Open Software Foundation (founded in 1988), The Open Group is supported by most of the world's largest user organizations, information systems vendors, and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to meet its new mission, as well as to assist user organizations, vendors, and suppliers in the development and implementation of products supporting the adoption and proliferation of systems which conform to standard specifications.

With more than 200 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritizing, and communicating customer requirements to vendors
- conducting research and development with industry, academia, and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating, and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the Open Brand, represented by the ''X'' mark, that designates vendor products which conform to Open Group Product Standards
- promoting the benefits of the IT DialTone to customers, vendors, and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development, and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trademark on behalf of the industry.

The Development of Product Standards

This process includes the identification of requirements for open systems and, now, the IT DialTone, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product. There are currently two forms of Product Standard, namely the Profile Definition and the Component Definition, although these will eventually be merged into one.

The ''X'' mark is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the Open Brand they guarantee, through the X/Open Trade Mark Licence Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical documentation, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys, and business titles.

There are several types of specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our Product Standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. Where appropriate, they can demonstrate product compliance through the Open Brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *Preliminary Specifications*

  Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organizations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif, and CDE.

  Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation—programmer's guides, user manuals, and so on—relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development, or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Program. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

This Document

This document is a Preliminary Specification (see above). It specifies the DCE security model, services, interfaces and protocols. Its purpose is to provide a portability guide for security application programs and a conformance specification for DCE security implementations.

This document is organized as follows:

- Part 1 presents an introduction to issues in security and describes how general concepts in security are supported by DCE. Included new for DCE 1.1 are such topics as delegation, extended registry attributes, and extended login and password management.
- Part 2 defines in detail the security specifications, formats, protocols and RPC interfaces supported by DCE. These include the following:
  - — infrastructure protocols and services, including checksum and encryption/decryption mechanisms, key distribution services and privilege services
  - — access control lists (ACLs) and ACL managers
  - — delegation
  - — replication and propagation
  - — protected communication services
  - — higher-level facilities, including ACL editors, registration service, ID map facility, key management facility, login facility and credentials facility.
- Part 3 contains reference manual pages describing the following security APIs supported by DCE:
  - — access control list
  - — registry

    This API has significant additions for DCE 1.1.
  - — ID map
  - — key management
  - — login

    This API has additions for delegation.
  - — credentials.
- Part 4 contains a symbol mapping table, list of error codes and a glossary. The error codes are new for DCE 1.1.

Intended Audience

This document is written for security application programmers and developers of DCE security implementations.

Typographic Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used for system elements that must be used literally, such as interface names and defined constants.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote function names and variable values such as interface arguments.
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- The notation [EABCD] is used to identify an error value EABCD.
- Syntax, code examples and user input in interactive examples are shown in `fixed width` font.
- Variables within syntax statements are shown in *`italic fixed width font`*.

Trade Marks

Open Software Foundation™, OSF™, the OSF logo, OSF/1™, OSF/Motif™ and Motif™ are trade marks of The Open Software Foundation, Inc.

Network Computing System® is a registered trade mark of Hewlett-Packard Company.

DECnet® and VAX® are registered trade marks of Digital Equipment Corporation.

Microsoft®, NetBIOS® and NetBEUI® are registered trade marks of Microsoft Corporation.

NetWare® is a registered trade mark of Novell, Inc.

System/370® and IBM® are registered trade marks of International Business Machines Corporation.

Cray® is a registered trade mark of Cray Research, Inc.

Postscript® is a registered trade mark of Adobe Systems Incorporated.

Statemate® is a registered trade mark of i-Logix Incorporated.

UNIX® is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

This list represents, as far as possible, those products that are trademarked. The Open Group acknowledges that there may be other products that might be covered by trademark protection and advises the reader to verify them independently.

X/Open® is a registered trademark, and the ''X'' device is a trademark, of X/Open Company Limited.

Referenced Documents

The following documents are referenced in this specification:

ANSI X3.92
: American National Standards Institute, Inc. (ANSI): X3.92–1981, American National Standard Data Encryption Algorithm

ANSI X3.106
: ANSI X3.106–1983, American National Standard for Information Systems — Data Encryption Algorithm — Modes of Operation

CCITT V.42
: CCITT (now ITU-T) Recommendation V.42–1988.

CCITT X.208
: CCITT (now ITU-T) Recommendation X.208-1988.

CCITT X.209
: Recommendation X.209-1988 (Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)).

CCITT X.509
: Recommendation X.509-1988.

  It is cited in Section 2.2 on page 136. ISO/IEC 3309:1993(E) is equivalent for the purposes of that section.

DCE Directory
: X/Open CAE Specification, December 1994, X/Open DCE: Directory Services (ISBN: 1-85912-078-4, C312).

DCE RPC
: X/Open CAE Specification, August 1994, X/Open DCE: Remote Procedure Call (ISBN: 1-85912-041-5, C309).

DCE Time
: X/Open CAE Specification, January 1994, X/Open DCE: Time Services (ISBN: 1-85912-067-9, C310).

ISO 8859-1
: ISO 8859-1: 1987, Information Processing — 8-bit Single-byte Coded Graphic Character Sets — Part 1: Latin Alphabet No. 1.

RFC 1321
: The Internet document RFC 1321, by R. Rivest, dated April 1992.

RFC 1510
: The Internet document RFC 1510, by J. Kohl and C. Neuman, dated September 1993.

CAE Specification

Part 1
Introduction

The Open Group

Chapter 1

Introduction to Security Services

This chapter provides an overall introduction to the security services supported.

Section 1.1 supplies some *general* background information on security, in an analytic top-down fashion. Section 1.2 on page 12 supplies an encompassing *security model* for DCE. The remaining sections of this chapter supply information on the *specific* security features supported by DCE, in a synthetic bottom-up fashion. Thus, this chapter as a whole gives the reader an overall understanding of how his / her intuitive concepts of security are supported by DCE.

The remaining chapters discuss the detailed descriptions, specifications and interfaces (both RPC interoperability interfaces and API portability interfaces) of all the supported DCE security features.

1.1 Generalities on Security — The Architecture of Trust

The goal of this section is to introduce terminology, within the context of explicating the concepts that computer users (''should'') have in mind when they think about ''(distributed) computer security from first principles''. Although a technical orientation is taken, this explication is an intuitive and informal discussion of the *philosophy* and *psychology* of security — as distinguished from a formal or rigorous model of the *logic* and *mathematics* of security. Thus, to this extent at least, this section is largely *informative*, and not *normative*. Nevertheless, the inclusion of a section on generalities like this is considered important (even necessary) here, for several reasons:

- Computer security as an academic discipline has not yet penetrated the engineering curriculum to the point where it can be assumed as common background for the audience of DCE (producers and consumers of products based upon it).
- The current security literature is too voluminous and complex to admit an adequate appraisal of all of its technical aspects here — and yet there does not seem to exist a convenient summary meeting the requirements that can be cited. (The interested reader is encouraged to consult the literature; the reader already familiar with this material may skim this section, but should not skip it altogether because of the terminology introduced.)
- The various models and terminology sets that exist in the current literature differ from one another to various extents, and despite some general agreement on certain overall principles, many details have not yet reached final form. It would thus be inappropriate for this specification to be expressed in a way that favours a single one of those models. An intuitive approach as presented in this section, admitting mappings to various such models, best satisfies the needs of this specification.
- At the deepest levels, some scepticism continues to be expressed about the very efficacy and viability of the more abstract attempts to ''finalise'' security theory, especially formal models and the ability to verify implementations of such models. For example, no completely satisfactory model of even so basic a notion as ''identity'' has yet been given. Indeed, it has been seriously suggested that the shortcomings of attempted abstractions indicate there is no *single* ''Platonic form'' of security (even theoretically), but rather *several* concepts of security that bear only a ''family resemblance'' to one another.
- Ultimately, it is imperative to come to a ''human'' understanding (for example, an intuitively appealing ''programmer's model'') of the meaning of security, precisely because the

problems it attempts to solve are explicitly human ones (as opposed to technological ones, see below), and are consequently not completely self-evident.

It can be concluded that (at least currently) some intuition is not captured by, and the intuition is therefore more reliable than, ''rigorous'' definitions of security (if such exist) — certainly so for the casual user — and this justifies the limited goal of this section: merely to explicate this intuition. As stated above, since an appropriate treatment of the basic intuitive content of computer security along these lines does not seem to be readily available, it is therefore necessary to provide one here.

As the discussion thus far shows, in the present state of information technology *definitive* understanding of many of the problems of computer security is lacking, much less their solutions (though as this specification shows, there are some *adequate* solutions for some of the problems that are understood). Security is unique in this respect, in that other areas of computer technology are circumscribed by relatively limited *technological* parameters, whereas security must take into account the unlimited ingenuity of skillful and determined *human* attackers having the full range of computer (and other) tools at their disposal. This dichotomy is justified and rationalised by the *costs* required to reduce to acceptable levels the risks to correct system behaviour from various *threats*: the risks from (non-human) *benign glitches* (for example, bugs, equipment failure, natural disaster) can often be held to very low levels by comparatively simple and cheap engineering practices, but the risks from (human) *malicious attacks* can usually be held to acceptably low levels only by instituting more complex and expensive countermeasures.

Therefore the security services supported by this specification must be viewed as indicative of only one possible state of the (security) art. In particular, the fitness of the facilities described here for the security requirements of a given installed environment relies on specialised *evaluation processes* beyond the scope of this specification, especially the way that these facilities interact with security facilities provided by other system components (such as hardware, OSs, user guidelines and administrative policies).

Finally, it is to be noted that nothing currently specified in this revision of this specification is intended to preclude future enhancements as they become socially acceptable, technically viable and commercially available.

1.1.1 Security Attributes: Authenticity, Integrity, Confidentiality

The overall *goal* of security is to *prevent* the (human) *misuse* (either illicit use altogether, or licit-but-irresponsible use) of resources — or, failing prevention, to at least *detect* misuse and *recover* from it. This definition is to be interpreted broadly (for example, it includes such ''misuses'' as *illicit repudiation*). *Proper* usage is to be allowed, of course, but it is the essentially negative nature of the ''prevent misuse'' clause that signals why security is so difficult to achieve in practice: namely, anticipating and defending against all possible misuses is an essentially open-ended challenge — it's hard to prove that ''bad things didn't happen''.

Translating this goal into the language of information technology, it can be said that computer security attempts to *protect* the *security attributes* (to be specified below) of computer *resources*, especially information (data) — that is, to preserve these attributes as *invariants*. The word ''protect'' is used here in a primitive, undefined (but intuitively appealing) sense. Resources whose security attributes are ''adequately'' protected are said to be *secure*; otherwise, they are said to be *(potentially) compromised* or *insecure*.

In DCE, the resources to be protected further exist in a *distributed* environment; that is, one in which the notion of *communication* is an explicit model primitive — in particular ''data-in-communication'' (''on-the-wire'') is equally as important as static data (in memory, or in storage media) itself.

The specific ‘‘security attributes’’ (in an intuitive sense, as always throughout this section) to be protected must support the stated goal of preventing misuse of resources. To this end, the ones currently supported in the DCE security model are the following:

- *Authenticity*

  The state of genuinely representing reality, in an extrinsic sense; that is, of being *correct* (actually representing that which is alleged to be represented), especially of data originating at a definitive, authoritative source.

- *Integrity*

  The state of being in an unimpaired condition, in an intrinsic sense; that is, of being sound and whole, especially of being unmodified (either in their place of residence, or in transit).

- *Confidentiality*

  The state of controlled accessibility; that is, of being accessible to only a designated few; of being *known* to only a limited few (*privacy* or *secrecy*).

To a first order of approximation, and focusing only on communications traffic, one can think of preserving ‘‘authenticity’’ to mean being certain of the *identity* of the peer communicating entity; preserving ‘‘integrity’’ to mean protection against (undetectable) *writes* (either in-place or *active wiretapping*); and preserving ‘‘confidentiality (privacy)’’ to mean protection against *reads* (either in-place or *passive wiretapping*).

Security attributes other than those listed above are sometimes contemplated — for example, *assured service* (the state of being available and obtainable for use when needed; its opposite is known as *denial of service*) — but these are currently only peripherally (that is, other than access control via ACLs, see below) within the scope of DCE (though they are typically supported by certain implementation and administrative practices associated with DCE, such as partitioning, replication, backup and restore). On the other hand, some other attributes (for example, qualitative or intangible attributes, such as accuracy, timeliness and reliability) are commonly considered to be ineligible as security attributes, on the basis that they cannot be protected from misuse directly (and are better protected indirectly by other security attributes such as those above), or are beyond the reach of current computer security technology.

1.1.2 Policy versus Service versus Mechanism

The notion of *security policy* refers to a set of high-level requirements or rules an ‘‘organisation’’ places on the security attributes of its assets (often independently of the use of computers). *Security services* refer to the tools (computer and otherwise) available to the organisation for enforcing such policies, and *security mechanisms* refer to the lowest-level technology used to implement security services. A *security domain* (or *realm*) is the scope of a particular security policy. Since organisations often exist in hierarchical or other relationships (in the degenerate case they consist of just a single individual), the condition of overlapping security domains is potentially an important one (for example, site security officer may require every department’s policies to be subservient to the site’s, in some sense).

As an example of policies, security policies that allow or require individual members of an organisation to protect the data they ‘‘own’’ are said to be *discretionary*; policies that mandate organisational, not individual users’, control are said to be *mandatory*. As an example of services, services for controlling access of ‘‘subjects’’ to ‘‘objects’’ (see Section 1.1.3 on page 6) can be *identity-based* (that is, ‘‘*who*’’-criteria such as individual identity and group membership), or *rule-based* (for example, ‘‘*what*’’-criteria such as *clearance* of subject and *sensitivity* of object, ‘‘*when*’’-criteria such as time-of-day restrictions, and ‘‘*where*’’-criteria, such as the hosts on which programs reside). As an example of mechanisms: cryptographic algorithms (see below) can be

based on *symmetric* or *asymmetric* key technology. Various types of security services and mechanisms, or combinations of them, may (or may not) be suitable for protecting an organisation's resources, depending on the security policy the organisation has adopted (for example, discretionary policies are often supported by identity-based services, and mandatory policies are often supported by rule-based services).

This specification supports computer security *services* (and the *mechanisms* necessary to implement them) that can be used by organisations in a variety of ways, to support in whole or in part many different security *policies*. But it is always the responsibility of the organisation to evaluate the adequacy of this specification's (or any other) security services and mechanisms for supporting its specific policies. Such an evaluation will normally be based on a *threat analysis*, but a discussion of that topic is beyond the scope of this specification.

1.1.3 Subjects and Objects, Privilege and Authorisation

In the security sense, the term *object* is used to refer to ''the *passive* aspect of'' entities (or resources, for example, data) whose security attributes are to be protected, and *subject* or *principal* refers to ''the *active* aspect of'' entities (for example, people or computer processes) that interact with objects. Subjects are also considered to be objects insofar as they have security attributes that need to be protected. Subjects are sometimes further classified into *initiators* (those subjects that initiate interaction with objects, and are accountable for the interaction) and *intermediaries* or *delegates* (intermediates that merely assist a principal in an interaction but are not responsible for initiating it). This distinction is typically only important in a distributed environment, because the initiator/delegate relationship is typically established by subjects communicating with one another. (Note that DCE supports delegation in this revision.)

In the security sense, the term *access* refers to the interaction (mentioned above) of a subject with an object, the possible types of access in a system being classified into types or classes called *access modes* (for example, *read* or *write* a file, *signal* or *communicate* with a process, *execute* on a processor). The specific instances of access modes that a specific subject has been *granted* (as opposed to *denied*) to a specific object are called the subject's *access permissions* (or *rights*) to the object. A subject that has been granted privilege to access an object in a specified mode is said to be *authorised* to access the object in that mode.

The abstract matrix (which is primarily conceptual, not actually realised by a single monolithic data structure in implementations) whose rows are parameterised by subjects and whose columns are parameterised by objects, and whose entries consist of the permissions that the subjects have to the objects, is called the *access matrix* of the security system. A *row* of the access matrix (that is, the ''subject-side'' or ''client-side access information that a single subject has to all objects) is called the *privilege information* (or *vector*) or *capability list* associated with the subject. A *column* of the access matrix (that is, the ''object-side'' or ''server-side'' access information that all subjects have to a single object) is called the *control information* (or *vector*) or *authorisation list* or *access control list* (*ACL*) associated with the object.

Note that there *do* exist entities *other* than subjects and objects in computer systems; for example, hardware and OSs. (No special name is reserved for these, other than simply ''(computing) entities''.) Such entities do not figure into the access matrix (alternatively, such entities as hardware and OSs may be inserted into the access matrix with ''infinite privileges''), but they obviously have a part to play in the security of systems. Much of what is said here about subjects and objects applies with appropriate modification of detail (in the usual intuitive sense appealed to throughout this section) to these entities — and this understanding is assumed throughout this specification unless explicitly stated otherwise.

1.1.4 Knowledge versus Belief; Trust

Subjects cannot always have absolute *knowledge* (in the sense of logical *provability*) of the true status of security attributes of objects. Instead, they must often rely on relative *belief* (based on a variety of considerations, evoking various levels of *confidence* or *assurance*) about their status — and they are then entitled and expected to act ''as if'' their belief were knowledge. Belief is often a *subjective* matter and can be based on such qualitative criteria as ''climate of opinion'' (for example, ''everybody 'knows' DES is 'pretty safe' '') or ''assessment of personal character (of individual humans)'', but there does exist at least one *objective* criterion that is universally accepted as an appropriate standard on which to base belief, and that is *mathematical probability* (and its concomitant, *computational complexity*).

This is best illustrated by an example. Users are usually justified in believing that their password (which is a piece of protectable data, and is therefore an object in the security sense), if well-chosen, is secure (in particular, its confidentiality attribute is protected), because the probability of someone's guessing it is acceptably low. But if the user discovers their password written on a scrap of paper in a wastebasket, they are justified in believing (and acting ''as if'') it has been compromised — that is, even if they do not know with certainty that a miscreant intends to illicitly access their account, the probability of that event is now unacceptably high. Of course, the actual values of ''low'' and ''high'' probability, and the *recovery* procedures to be taken upon suspicion of compromise, are matters of security policy.

A subject is said to *trust* an object (or a subject, or other entity such as hardware or OS) if it believes the object is secure. That is, subject *A* trusts object *B* if *A* believes *B*'s security attributes such as authenticity, integrity and confidentiality have not been compromised. If case *B* is a subject (or other active entity, such as hardware or OS), this is often paraphrased by saying ''*A* trusts *B* if *A* believes *B* behaves correctly (that is, the way *B* is specified to behave)'', though this paraphrase focuses only on the attribute of authenticity. As conceived of here, ''trust'' cannot be *created*; it can only be *posited* (*a priori*) of an entity, and then *transferred* to other entities (''chains'' of trust) — see Section 1.1.5.

1.1.5 Untrusted Environments: A Priori Trust and Trust Chains

It is the classical position of computer security to take a conservative (or ''fail-safe'') stance; that is, to council a subject to believe its environment to be untrustworthy unless it can be convinced to believe otherwise. The accepted technique for this is to ''bootstrap'' the trusted environment: introduce a minimal number of ''small'' (that is, easily protectable) *a priori* trusted entities to serve as the foundation upon which a layer of more and ''larger'' (that is, harder to protect) trusted objects can be established (by trusted means, especially by logically sound arguments), and continue in this way to build up *(trust) chains* of trusted entities (also called *indirect* or *transitive trust*), resulting in an overarching *trusted environment* (= set of trusted objects).

These *a priori* trusted entities can take many forms, and go by many names, for example:

- *Self*

  Every subject is assumed to trust itself (self-deception is beyond the scope of most security models).

- *Physical Security*

  A secure entity to which no attacker can gain access can be trusted to remain secure from external attack. (This does not address the question of internal security, however; for example, malicious code.)

- *Trusted Computing Base* (TCB)

  The fundamental core set of hardware and software that must be trusted *a priori* (for example, local machine hardware and OS). (As usual in this section, this definition does not purport to be rigorous, but its intuitive import is clear enough: ''All the computer stuff that is relevant to supporting security policy''.)

- *Reference Monitor*

  A trusted subject (or entity) that mediates all access to a protected object. Indeed, the reference monitor is considered to *embody* the protection of the object.

- *Trusted Algorithm or Protocol*

  An algorithm or protocol whose efficacy is trusted — in particular, cryptographic algorithms (see below), and, in a distributed environment, cryptographic protocols (a ''protocol'' is simply a distributed algorithm; that is, one in which communication is an explicit primitive operation).

- *Secrets*

  The ''smallest'' (in the sense of ''easiest to protect'') objects, whose security is considered tantamount to the security of ''larger'' (in the sense of ''harder to protect'') objects, by means of trust chains (see discussion below on key-based security).

- *Authority* (*Trusted Third Party*)

  An entity which is trusted to know the secrets of objects other than itself.

A typical responsibility for an authority is to *certify* objects; that is, to vouch for their security. For example, consider a *credential*, which is a data element (an object) containing security information (say, privilege information) about a subject, say *A*. Suppose *A* presents its (purported) credential to another subject, *B*, which acts as a reference monitor for the object to which *A* desires access. In order for *B* to make an informed access decision, it needs to be convinced of the credential's security (otherwise, it should make the default ''fail-safe'' decision to deny access). But if *A* is trusted by an authority, say *X*, which *B* trusts, and if *X* has *certified* the credential (thereby turning it into a *certificate* (or *token*, or *ticket*), and so on), then *B* is justified in granting or denying access on the basis of the (now trusted) credential.

1.1.6 Distributed Security: Secrets and Cryptology

A certain amount of physical security is a *necessary* element of *a priori* trust in all environments, and may even (depending on security policy) be sufficient for all the security of some suitable environments (for example, of standalone machines, or of machines and the network itself in small local area networks). But physical security by itself is usually an incomplete solution even within a single physical security domain (because there usually exist threats from sources other than ones related to physical access), and is certainly inadequate in an environment of geographically dispersed distributed systems (such as those contemplated by DCE) that span multiple physical security domains. So, physical security is almost always supplemented with *logical security* (security based on non-material entities).

Indeed, the single most important tool for building trust chains, especially in a distributed environment, is an entity of logical security, namely the concept of *secrets*. The idea is that if a subject *A* can demonstrate (via a trusted protocol in the distributed environment) to another subject *B* that it knows a (secure) secret, then *B* is justified in believing *A* itself is secure. This ''link'' in the ''chain'' of trusted subjects, from a ''small'' object like a secret to a ''large'' object like a subject (and the objects it acts as reference monitor for), effects the ''bootstrapping'' mentioned above. In this way, the seemingly intractable problem of the security of a complex

system as a whole is reduced to the more tractable problem of the security of a small subset of the system: its secrets (''Kerckhoffs' Doctrine'' — see below).

The science of using secrets to implement security mechanisms is called *cryptography*, and the art of analysing cryptographic mechanisms for the purpose of (potentially) compromising systems based on them is called *cryptanalysis*; the two together go under the combined name of *cryptology*. Of course, these ideas predate the use of computers; using secrets for security purposes was implemented already for military purposes in prehistoric antiquity, where messages relayed by courier figured in early ''distributed systems''.

1.1.7 Encoding/Decoding and Encryption/Decryption of Messages

Secrets are particularly effective in protecting the security of *messages*; that is, ''data-in-communication''. A primitive way to do this is by the mechanism of *encoding/decoding*. By prearrangement, two subjects agree that a specified message is to be *semantically* represented (*encoded*) by a specified utterance — the actual mapping between message and utterance (the so-called *codebook*) is kept secret between the two subjects. For example, ''The moon is full'' might be mapped to ''Drop the bomb at midnight''. While this mechanism in its purest form is rather secure, it is rigid (it's hard to communicate concepts not in the prearranged codebook), and it's difficult to implement securely (the codebook is difficult to protect).

By making use of *syntax* instead of *semantics* (via alphabetic writing), a mechanism flexible enough to effectively support the security of arbitrary messages, called *encryption/decryption* or *encipherment/decipherment*, becomes available. The message to be communicated is first (non-secretly) represented in a well-known alphabetic representation (for example, first express the message in English, then spell it using the symbols for, say, lower-case letters, digits, space and period — an alphabet of 38 characters), then the resulting representation is ''scrambled'' (*encrypted* or *enciphered*) by some secret (or secret-based) technique prearranged between the communicating subjects. For example, the scrambling algorithm might consist of a specified combination of *substitutions* (whereby each symbol is replaced by another predetermined one, possibly from another alphabet; for example, ''*a*'' is replaced by ''α'', ''*b*'' by ''β'', and so on), and *transpositions* (*permutations*) (whereby each symbol of the message is interchanged with another symbol of the message; for example, each symbol is interchanged in pairs with its neighbour). The receiving subject *decrypts* (or *deciphers*) the received *cryptotext* (*ciphertext*) by an inverse technique, thereby recovering the original *plaintext* (*cleartext*) message.

With the advent of digital computers, the principles of encryption/decryption not only remain valid (the plaintext alphabet consisting now of binary bit-representations, say ASCII), but have been raised to new levels of sophistication because of the raw power of computers for both cryptographic and cryptanalytic purposes. A new ''golden age'' of cryptology has, in fact, arisen precisely because of the ''digital (computer) age'' — and due to this, the field is changing rapidly at the present time.

1.1.8 Key-based Security: Kerckhoffs' Doctrine

As discussed above, the security of (secret-based) communications can be reduced to the security of the secrets driving the encryption/decryption mechanisms. A further refinement of this idea is to require to be kept secret, not the entire *algorithm* used for encryption/decryption, but only a small (that is, even easier to protect) part of it, namely, a *parameter* to the algorithm. Such a secret parameter is called a *cryptovariable* or *key* (in analogy with ''locks and keys'', not ''database query keys'' — the terminology predates computers). This idea is called *Kerckhoffs' Doctrine* (in honour of the pioneering 19th century cryptologist Auguste Kerckhoffs, who first articulated it), and is usually paraphrased as:

SECURITY RESIDES SOLELY IN THE KEYS

and not, for example, in such qualities as attackers' ignorance of the cryptoalgorithms themselves (the latter is humourously known as ''security by obscurity''). By this means, the security of the encryption / decryption mechanism is decomposed into two components:

1. the *strength* of the underlying (non-secret) algorithm (that is, its resistance to cryptanalysis that may compromise the contents of encrypted messages to an attacker not initially knowing the key)
2. the secrecy of the key itself.

Extended to *distributed* security, Kerckhoffs' Doctrine continues to assume that distributed security must reside solely in the keys, even if the attacker knows the cryptographic protocols in use, and has completely ''compromised the network''; that is, has the unlimited ability to intercept and modify *all* data-in-communication. In this manner, the infrastructure of secrecy-based secure communication mechanisms is reduced to just three elements (which must be scrutinised both individually and in combination for their adequacy in supporting the chosen security policy):

- strong cryptographic algorithms
- *key management* — creation, storage, distribution, use and destruction of the keys themselves
- secure protocols (the new element introduced by *distribution*).

DCE security, in common with all practical contemporary distributed computer-based security, is based on these elements (see the discussion of *tickets*, in Section 1.2 on page 12).

1.1.9 Outline of the Remainder of this Chapter, and of this Specification

The presentation of this chapter is now shifted from generalities (''general security theory independent of DCE'') to specifics (''specific services and mechanisms supported by DCE''). To that end, the remaining sections of this chapter proceed as follows:

- Next to be presented is an overall DCE Security Model.

  This marshalls all the DCE security services and mechanisms into a unified encompassing mental construct (or architecture, or framework).

- The most basic infrastructural elements of DCE security are then presented, revolving around cryptographic algorithms and cryptographic protocols as discussed above:

  — Checksum Mechanisms (MD4, MD5)

  — Encryption / Decryption Mechanisms (DES)

  — Key Distribution Services (Kerberos AS+TGS Authentication System)

  — Privilege Services (PS Authorisation System)

  — Cells, and the Cross-cell Authentication / Authorisation Model.

- The DCE mechanisms for general access control (authorisation) are presented next:

  — Access Control Lists (ACLs)

  — ACL Managers, Permissions and Access Determination Algorithms(s)

- The above-listed services and facilities reside at a conceptual layer ''below'' Protected RPC. At this point, the central fact of integration of security with communications, which determines the DCE programming model, is presented: Protected Communication Services (RPC Application-level Authentication, Authorisation, Integrity and Confidentiality).

- The remaining services and facilities reside ‘‘above’’ Protected RPC. The first to be discussed is the management of ACLs, identities and keys:
  - — ACL Editors
  - — Registration Service (RS, including the ACL manager types it supports)
  - — ID Map Facility
  - — Key Management Facility
  - — Login Facility
  - — Extended Registry Attribute Facility
  - — Extended Privilege Attribute Facility
  - — Password Management Facility.
- Next, discussion of the integration of security services with other services supported by DCE is given:
  - — integration with Time Services
  - — integration with RPC Services
  - — integration with Naming Services.
- Finally, these additional security features are discussed:
  - — delegation, introduced for DCE 1.1 and newer versions, and which extends the DCE security facilities to encompass this capability.

Note: It is anticipated that future versions of DCE will support additional security features — for example, auditing, alternate cryptographic algorithms (especially, asymmetric key technology), alternate authentication and privilege services, and so on.

1.2 DCE Security Model

This section forms a bridge between the generalities of Section 1.1 on page 3, and the specifics of the following sections. It introduces the basic DCE *security model*; that is, the architecture or framework into which the various DCE security services and facilities may fit. This section just gives a ''once over lightly'' treatment — all terminology and details not properly introduced in this section are discussed fully in the following sections and chapters.

The security model is depicted in Figure 1-1, which shows as its focal element an application *Client* (source of an RPC) interacting with an application *Server* (target of an RPC) in a DCE environment.

From the point of view of *communications* (RPC), the client and server are communicating entities, with the server acting as the *Resource Manager* for the *resources* (*objects*) under its control — that is, the client invokes a remote procedure call, which is executed by the server, acting on its resources. From the point of view of *security*, the client and server are *subjects*, and the server acts as a *Reference Monitor* for its (protected) *objects*; that is, the ultimate arbiter for access by clients to the objects.

The application server is responsible for associating an ACL to every object it wishes to protect. The exact definition of what is an ''object'' (or ''resource'') is entirely at the discretion of the server; that is, it is application-dependent. As examples, an object could be an item of stored data (such as a file), or could be a purely computational operation (such as matrix inversion). Said another way, by its *choice* of the things it attaches ACLs to, the server *defines* what its (protected) objects are. Note that even if the ACLs are actually physically stored separately from the objects they protect (that is, in an ''ACL server''), DCE recognises them as being within the same security domain as the objects and the object server — intuitively stated, ''the ACLs cannot be more secure than the objects or the object server (reference monitor)''. *ACL Editors* are programs that directly manipulate servers' ACLs (without actually accessing the objects protected by those ACLs). (ACL Editors that support a *user interface*, enabling end-users such as the ''owners'' of objects to interactively manage the ACLs on objects, are called *ACL Editing Commands* — these are not specified in this specification.)

The security environment in which such client/server access happens has three *services* — implemented as RPC servers — at its core:

- *Registration Service* (RS), or *Registry Service*, or simply *The Registry* (Rgy)
- *Key Distribution Service* (KDS), or *Authentication Service/Ticket-granting Service* (AS+TGS)
- *Privilege Service* (PS), or *Privilege-ticket-granting Service* (PTGS).

These three services are *trusted third parties* in the DCE security model, and they form part of the DCE *Trusted Computing Base* (TCB) — also called the *network TCB*, for short. These three services are thus entrusted to know the secrets of subjects and other security information, and to implement the mechanisms for enforcing security policies. Their security must therefore not be compromised, and in an installed site they must run on secure computers (consistent with the organisation's security policy; for example, physically secure machines running secure OSs under the administration of a security officer).

The RS, KDS and PS are actually distributed, partitioned (and potentially replicated) services, with the unit of partition being the *cell* (that is, for security purposes, an *instance* of the RS/KDS/PS triple). The cell in whose RS datastore the security information for a given principal is held is called the *home cell* of the principal. In a multiple-cell environment, the various RS, KDS and PS services participate in an *inter-cell* (or *cross-cell*) coordination, to provide logically unitary services (that is, to create the effect of a *multi-cell DCE TCB*). In this inter-cell coordination, the per-cell RS, KDS and PS servers do not need to communicate directly with

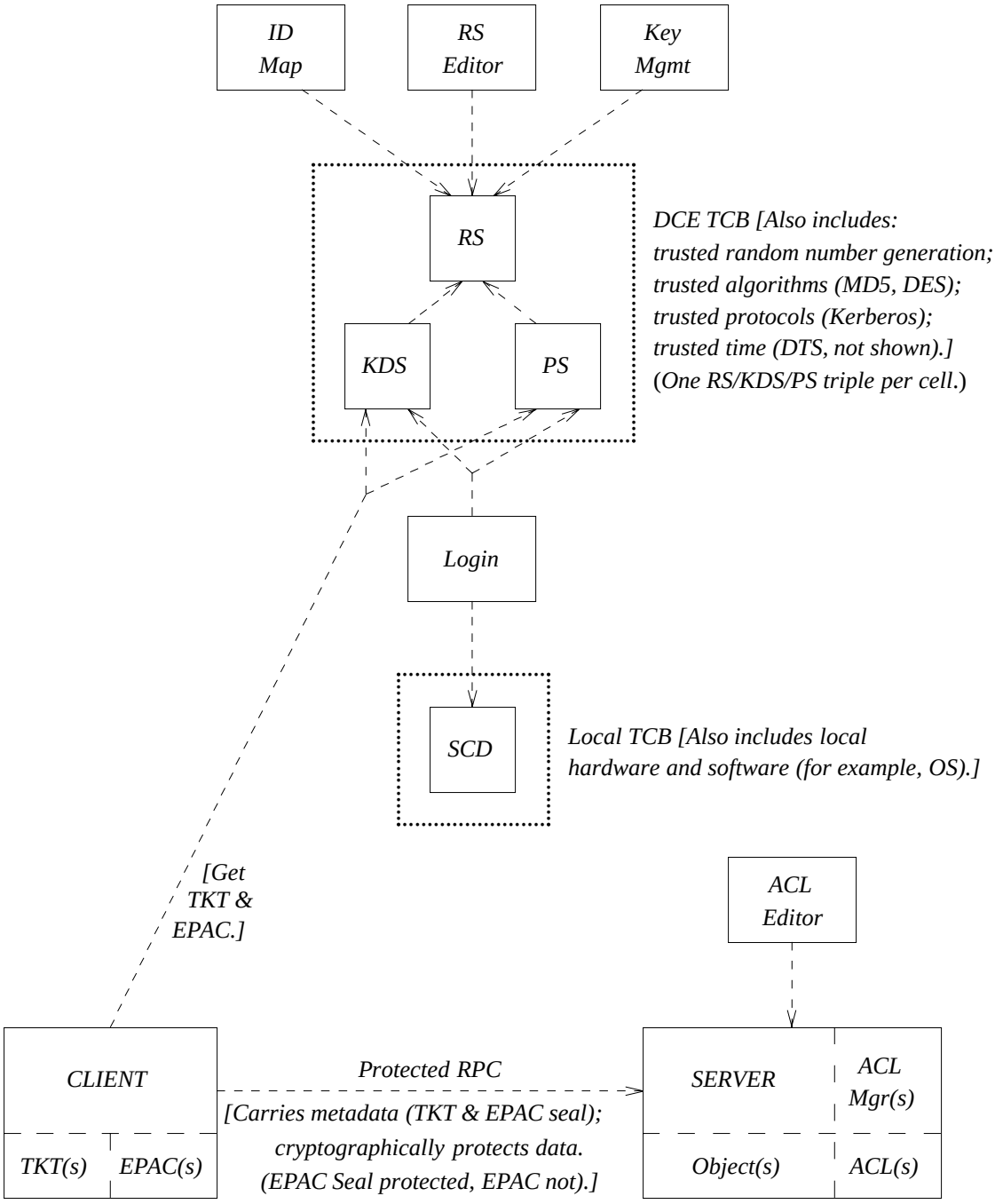


Figure 1-1 DCE Security Model

their foreign-cell counterparts in the performance of their services (they may do so incidentally, however, for such purposes as parsing stringnames into their component pieces, or for cross-cell key management). *Intra-cell*, the RS, KDS and PS do communicate amongst themselves, but these communications are *not* specified in the current version of this specification. Thus, for example, implementations are not prevented from implementing a cell's RS, KDS and PS within a single process (potentially replicated) — which is then typically known by a comprehensive

name, such as a *security server* or *security daemon*.

Before a principal (either client or server) can participate in the DCE security environment, it must have a *(principal) identity registered* with the RS. (This registration must *initially* be ''out of band'' of the protocols specified by DCE, in order to guarantee its security. The intuitive reason for this is that a user's password must be initially agreed to ''by word of mouth'' (that is, ''out of band'') between the user and the administrator before the system can use the protocols specified herein to authenticate the user. Furthermore, the initial administrative principal must also be installed by another ''out of band'' process, such as by pre-installing it before initial system startup.) These identities are represented by both user-friendly *cell* and *principal (string)names* and by their *UUIDs* (the *ID Map Facility* provides correspondences between these). The RS maintains a *datastore* of the identities of all subjects, and long-term secrets (cryptographic keys for the *DES* cryptoalgorithm) associated with them. *RS Editors* (or *Registry Editors*) are programs that directly manipulate RS datastores — typically, RS Editors support an *administrative interface*, enabling security administrators to interactively manage the RS datastore (this is not specified in this document).

Having previously registered their identities with the RS, before a client and a server can successfully participate in a client/server *session* within the DCE security environment, they must ''establish their identities'', which can be accomplished only by ''knowing their own secret'' (that is, knowing the secret (long-term key) associated with their identity in the RS datastore). Clients are typically endowed (by process-hierarchy inheritance) with the identity of the end-user invoking them, and these end-users establish their identities by means of the *Login Facility* (which is *password-driven*, for the convenience of *interactive* human users). Servers, on the other hand, are typically endowed with an identity independent of any end-user (for example, a system administrator) invoking them, and they establish their identities by means of the *Key Management Facility* (which is *key-driven*, for the convenience of *non-interactive* servers). In order for the local TCB to evaluate its trust of the DCE TCB (for such purposes as, for example, storing its ''standalone-machine'' user data in the ''network'' RS datastore), the local TCB must *itself* be a principal (the *host principal*, or *machine principal*) — this is the role fulfilled by the *Security Client Daemon* (SCD). The SCD (''host principal'' or ''machine principal'') can be viewed in some ways as the security analog of the ''host address'' communications concept.

At this point, it is convenient to introduce the notions of *ticket* and *Privilege Attribute Certificate* (PAC). Tickets are (protected) credential certificates, representing the ''authenticated identity'' of a client, trusted by a specified server to which they are *targeted* (that is, encrypted in the server's long-term key), and containing a short-term *session key*, which actually represents the authentication between the client and the server. Either this session key, or another one securely negotiated between the client and server, can function as a *conversation key* (also known as a *subsession key* or *true session key*); that is, actually used to cryptographically protect client/server communications.

Note: It is a common practice to use the terms ''session key'' and ''conversation key'' synonymously; indeed, the same key can function as both, but it is preferable to distinguish between these notions.

PACs are (protected) certificates, specifying the attributes of the client that the server uses to determine (''grant'' or ''deny'') access to its protected objects. Tickets that have PACs associated with them are called *privilege-tickets*). Non-privilege-tickets are managed by the KDS, and PACs are managed by the PS; privilege-tickets are managed by the KDS and PS working together.

All of this scaffolding has its culmination in the *Protected RPC* facility. When a client wishes to initiate a session with a server, it obtains a privilege-ticket targeted to the specified server, and then RPC service requests and responses between the client and server are *protected* (for *authenticity*, *integrity* and/or *confidentiality*, as agreed upon by the client and server) with the

session key contained in the privilege-ticket or a negotiated conversation (''true session'') key. When the client's initial RPC service request containing the privilege-ticket arrives at the server, the server's *ACL Manager* module uses the PAC associated with the privilege-ticket and the ACL attached to the protected object specified by the client, to make an access control decision; that is, whether to grant or deny access (for the specific operation specified by the client) to the object in question. Subsequent RPC service requests need not carry a privilege-ticket — once the session/conversation key and PAC have been securely established with the initial service request, they can be used with confidence to protect subsequent requests until an agreed-upon *time-out* date, typically on the order of a couple of hours, is reached (and then they can be re-established if necessary).

Note: The preceding description has been couched in terms of RPC, because that is the *communications* technology specified by DCE. However, the *security* technology specified in this specification is clearly applicable, with appropriate modification of detail, to arbitrary communications mechanisms. Clients and servers can still participate in a secure environment (clients protect their PACs, servers protect their objects with ACLs, and so on), provided a well-defined means is specified for communicating the necessary messages. However, the only communications mechanism currently specified in DCE is RPC.

The authentication protocols employed ensure *two-way* (*mutual*, *bilateral*) authentication between client and server. That is, they not only ''authenticate the client to the server'' (preventing a *masquerading* attack, whereby the client can gain access to a server posing as an identity for which it does not know the secret), but also ''authenticate the server to the client'' (preventing a *spoofing* attack, whereby a counterfeit server — other than the one designated by the client — can trick the client into believing it is communicating with the designated server).

Note: There is a potential ''vicious circle'' co-dependency of security on (RPC) communications and *vice versa*. This is because the protected RPC architecture requires that certain metadata (called *RPC verifiers*) are present in RPC PDUs (*protocol data units*), but that metadata is not available until *after* the client has communicated with the KDS and PS servers. This potential problem is averted by the KDS and PS services being offered over ''unprotected'' RPC. The security information required by protected RPC is thereby conveyed as *data* (cryptographically protected, to be sure), not *metadata*. The resulting dependency graph is thus simplified; protected RPC depends on security services, which in turn depend (only) on unprotected RPC (which of course does not depend on security services).

All the servers specified in DCE (RS, KDS, PS, SCD) communicate via RPC. The KDS and PS use unprotected RPC (their data is protected by direct cryptographic means, not by protected RPC). The RS and SCD use protected RPC, with authentication service **rpc_c_authn_dce_secret**, of authorisation type **rpc_c_authz_dce**, and of protection level **rpc_c_protect_level_pkt_integ**. (See the referenced X/Open DCE RPC Specification, augmented by this specification, for explanations of this terminology.)

1.3 Message Digests 4 and 5 (MD4, MD5)

Message Digest 4 (MD4) and *Message Digest 5* (MD5) are ''*non-invertible*'' (''*one-way*'') *functions*: given any message as input, MD4/5 produces a 128-bit *message digest* (or *hash*, *checksum* or *fingerprint*) as output. The critical cryptographic property claimed by MD4/5 is that they are *collision-resistant* (or *collision-*''*proof*''); it is very ''difficult'' (that is, computationally infeasible) to exhibit distinct messages having the same MD4/5 checksum.

Note that the MD4/5 algorithms are *not* encryption/decryption mechanisms (which are *invertible* functions), and they do *not* depend on a cryptographic key. In DCE, MD4/5 checksums are encrypted with DES to produce keyed cryptographic checksums for purposes of integrity-protection. A keyed cryptographic checksum of a message is called a *signature* or *message integrity code* (MIC) or *token* for the message.

DCE protects the authenticity and integrity (but not the confidentiality) attributes of a message by DES-MD4/5 crypto-checksumming the message; that is, DES-encrypting its MD4/5 checksum using a DES conversation key known only to the originator and recipient of the message (and, possibly, other third parties they trust). (Confidentiality protection is supported by DES-encrypting the whole message, not just its MD4/5 checksum.)

No interfaces to raw MD4/5 routines are directly supported in DCE. Instead, MD4/5 are embedded in various DCE protocols as discussed below.

1.4 Data Encryption Standard (DES)

The only encryption/decryption algorithm currently supported by DCE is the *Data Encryption Standard* (DES), in *Cipher Block Chaining* (CBC) Mode. This algorithm has been in steady use since the late 1970s, and in that time has received intense scrutiny without revealing debilitating weaknesses. Hence, it is generally considered to be ''secure'' by many commercial users. (Though as always it is the responsibility of each organisation to determine if DES is secure enough to satisfy its security policy.)

DES is based on 64-bit keys, of which only 56 bits are ''active''; that is, the key is treated as a 64-bit data item, though only 56 bits actually participate in the cryptographic characteristics of the algorithm. This means that an *exhaustive key search attack* would require $O(2^{56})$ operations, which is currently considered to be ''computationally infeasible'' for most commercial (non-military) applications. Note, however, that the key space may *effectively* (depending on the actual implementation) be much smaller in systems that use human-memorisable secondary keys such as *passwords* that are subsequently mapped into DES keys — such keys may therefore be more susceptible to *dictionary attacks* (exhaustive ''guessing'' of all passwords). Furthermore, the mere size of key spaces is no guarantee that more efficient (non-exhaustive) attacks don't exist.

The core DES algorithm acts on 64-bit blocks of plaintext and ciphertext. It is the CBC Mode that specifies how to encrypt/decrypt messages of length other than 64 bits.

DCE protects the authenticity, integrity and confidentiality attributes of a message by DES-encrypting it (or, if confidentiality is not required, encrypting only the message's MD4/5 message digest) in a DES conversation key known only to the originator and recipient of the message (and, possibly, third parties they trust).

No API to raw DES routines is directly supported in DCE. Instead, DES is embedded in various DCE protocols as discussed below.

Note: There may be restrictions on the use and/or import/export of DES by some national governments. Future versions of DCE may support other cryptographic algorithms to support the needs of these and other organisations.

1.5 Kerberos Key Distribution (Authentication) Service (KDS)

The function of the *Kerberos Key Distribution Service* (KDS) is to distribute *session keys* and *tickets* (certificates, protected credentials). Session keys are *short-term* keys *generated on the fly* and used to authenticate a client and a server to one another. Either the session key or a subsequently negotiated *conversation* (''*true session*'') *key* is used to protect most communications (especially, application-level communications) in the DCE environment — these are to be distinguished from the *long-term* keys associated with principals, which are held in the RS datastore and used only minimally, namely to protect internal protocols (''metadata'', as opposed to application-level data), ''bootstrapping'' the rest of the protected DCE environment (see the outline below). Stringnames in tickets represent the ''authentication identity'' of clients (as distinguished from their ''authorisation identity'', which is represented by UUIDs carried in privilege-tickets — see Section 1.6 on page 25). Tickets are *trusted* by a specific server to which they are ''targeted'', by *cryptographically protecting* them in the server's long-term key (or another key the server trusts).

A ticket that is *targeted to a KDS server principal* (KDS principals are also called *cell principals*, because of their central position in the architecture) is called a *ticket-granting-ticket* (TGT); its only use is as a kind of ''metaticket'', to be used for authenticating the client to the KDS server, so that the client can avail itself of the KDS's services (as seen in the next paragraph, the KDS supports only the ''metaservice'' of issuing tickets). A ticket that is targeted to a non-KDS server (that is, a non-ticket-granting-ticket) is sometimes called a *service-ticket* when it is necessary to emphasise its role in obtaining ''useful'' services (as opposed to the ''metaservice'' of merely obtaining tickets from the KDS). However, *pointedly making a distinction between ticket-granting-tickets and service-tickets is avoided in this specification unless it is absolutely necessary*.

The KDS offers (exactly) two kinds of services (and because of these two services, the KDS is also known as the ''*AS+TGS*''):

- *Authentication Service* (AS)

  Issues *initial tickets* (either ticket-granting-tickets or service-tickets), on the basis of *unauthenticated* requests (that is, an authenticating ticket is *not* required by the AS).

- *Ticket-granting Service* (TGS)

  Issues *subsequent* (*non-initial*) *tickets* (either ticket-granting-tickets or service-tickets), on the basis of *authenticated* requests (that is, an authenticating ticket *is* required by the TGS).

By abuse of language, one speaks of these two services as if they were autonomous entities. Thus, for example, ''*A* sends a message to the AS'' really means ''*A* sends a message to the KDS, requesting its AS service''.

Tickets contain the following information, appropriately protected (the ''names'' mentioned here are fully discussed elsewhere in this specification):

- *Named Client*

  Stringname of the client principal, represented by its cell name, and by its per-cell principal name (held in its cell's RS datastore).

- *Targeted Server*

  Stringname of the server principal, again represented by cell name and per-cell RS datastore name.

- *Session Key*

  For protecting communications (or negotiating a subsequent conversation — ''true session'' — key to do the actual protection of communications) between the named client and targeted server (or indeed, between any two principals that the client and/or server share the session

key with).

- *Lifetime Timestamps*

  The interval of time for which the ticket — and also the session key it carries, and the conversation keys derived from it — is to be honoured. This is represented by a set of timestamps, primarily consisting of a *start time* and an *expiration time*, but also including an additional *authentication time* and *absolute expiration time* for technical reasons.

- *Transit Path*

  The ordered sequence of authentication authorities (KDS servers) that vouch for this ticket, treated as an unordered set.

- Various other data discussed in detail in Chapter 4.

Given these concepts, the *basic (intra-cell) Kerberos authentication protocol* for authenticating a client *A* and a server *B* to one another is outlined below. This outline, while not an entirely minimal one, is intended to give only a ''working knowledge'' of the protocol, and does not delve into its many intricacies (full details are covered in Chapter 4). To this end, the scope of the outlines in this and the following two sections is to discuss the roles of:

- Primarily: *identities* (represented by stringnames in this section, and/or by UUIDs in Section 1.6 on page 25); *session keys* (representing the concrete manifestation of the abstract notion of ''authentication''); and *tickets* (certificates). Readers encountering Kerberos for the first time are advised to focus solely on these primary items.
- Secondarily: (*forward and reverse*) *authenticators* (for client-to-server and server-to-client authentication, respectively); *lifetime timestamps* (consisting of a *start timestamp* earlier than which the ticket is not to be honoured for service requests, *expiration timestamp* later than which the ticket is not to be honoured for service requests, and an *absolute expiration timestamp* later than which the ticket is not to be renewed by the KDS); *authentication timestamp* (determining ''freshness'' of communications); *nonces* (for matching requests and responses); *checksums* (for supporting integrity); *transit paths*; *conversation keys* (which are the ''true session'' keys used for actual protection of application-level data, as opposed to the metadata of the Kerberos protocol itself); and *authorisation data* (which is more properly introduced in Section 1.6 on page 25).
- Tertiary paraphernalia of the Kerberos protocol, such as *client addresses*, *options*, *flags*, *sequence numbers*, and so on, are not discussed in this chapter at all (see Chapter 4).
- Privilege-tickets and the inter-cell authentication protocol are discussed in Section 1.6 on page 25 and Section 1.7 on page 32.
- Access control lists (ACLs) and access determination algorithms are covered in Section 1.8 on page 40 and Section 1.9 on page 46.
- Finally, integrating security with RPC is mentioned in Section 1.10 on page 54 (but is not analysed at the protocol level until Chapter 9).

Note: The aspect of *lifetimes* of tickets (and short-term session and conversation keys) enables the capability of (secure) *caching*, which has a profound impact on implementations. Caching is currently considered to be implementation-dependent and therefore beyond the scope of this specification, but typical implementations exploit caching heavily because of the benefits in performance efficiency that it confers. In particular, a client need obtain a ticket-granting-ticket to a given cell only when it first needs to authenticate to a server in that cell (for example, at ''login time'' in its home cell), and again whenever it expires or ''times out'', and then the client can use it to obtain many other tickets. Similarly, a client need obtain a service-ticket

to a given server only the first time the server is contacted (and again whenever it times out), and then the client and server can use the corresponding session key (or an associated conversation key) many times. And similarly again, a client's privilege attributes (PAC) need be obtained by the server only once (and again whenever it times out), and cached there to be used many times to determine access rights for many service requests.

Throughout the outlines below, the following (standard) notations are used:

- $A \rightarrow B$: M
- $A \leftarrow B$: M'

  ''Message'' (that is, data) M communicated from A to B (typically via RPC invocation), followed by message M' communicated from B to A (typically via RPC return).

  Strictly speaking, the notation ''$A \rightarrow B$: M'' actually means: ''it is the *design intent* of the protocol that the message M be sent by A and received by B.'' And in a sequence of messages, ''it is the *design intent* that the order of messages be that specified''. Since the communications environment of cryptographic protocols is one in which messages may be rerouted, corrupted, maliciously modified, duplicated, resequenced, delayed, lost, and so on, assurance of any of these qualities cannot be guaranteed unless it is *provided by the protocol itself*.

- M_1, M_2

  Message consisting of two ''parts'' M_1 and M_2. (Similarly for messages consisting of more than two parts.)

  As befits the high-level intent of this chapter, this notation is *not* to be interpreted as carrying low-level formatting connotations, such as ''ordering'' or ''concatenation'' of the parts M_1 and M_2. (However, such formatting issues are covered carefully in the detailed protocol specifications of later chapters.)

- $\{M\}K$

  Message M encrypted with a key K, via some specified encryption mechanism.

  Again in accordance with the high-level intent of this chapter, this notation is *not* to be interpreted as carrying low-level information. In particular, the ''specified encryption mechanism'' may encompass more than a mere ''raw'' encryption algorithm; that is, higher-level information such as ''confounder'' and ''built-in-integrity'' information (see Section 4.3.5.1 on page 188).

Figure 1-2 on page 21, then, is the basic Kerberos authentication protocol, in the environment of a single cell X.

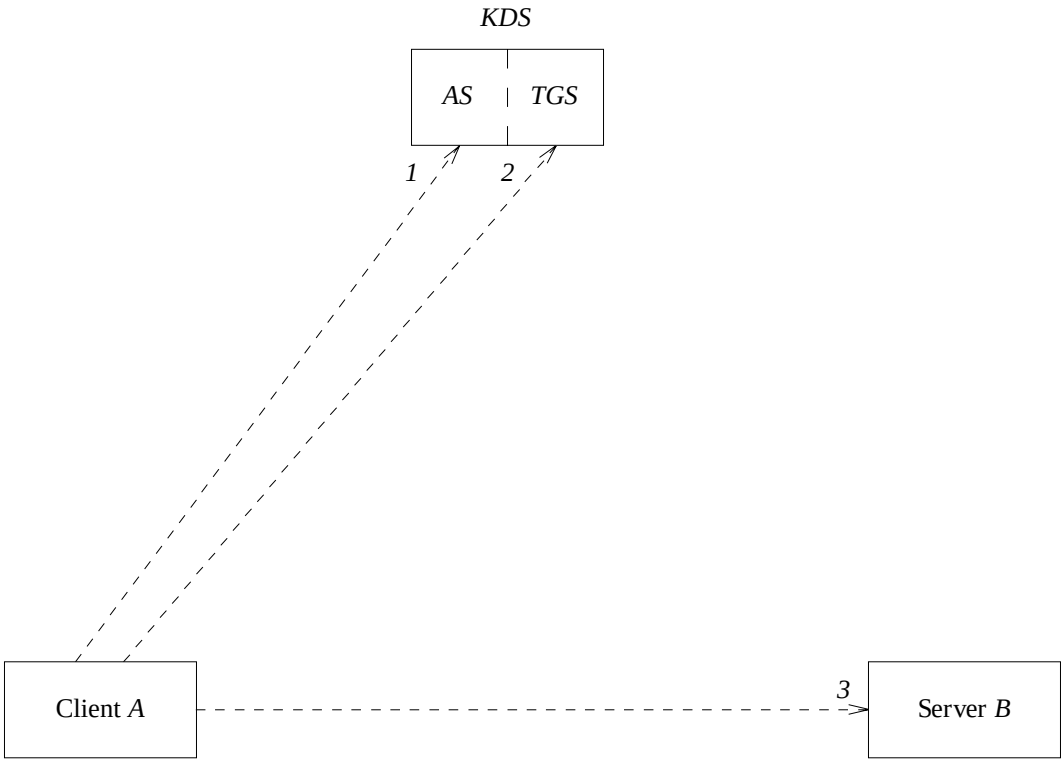


Figure 1-2 Basic KDS (AS+TGS) Protocol

- $A \rightarrow$ AS: *A*, KDS, $L_{A,KDS}$, $N_{A,AS}$

  1 (*AS Request: Ask for ticket to KDS*.)

  A, as a communicating entity, sends an AS Request message to the KDS server (in *X*, *A*'s home cell), communicating its (*A*'s) claimed principal stringname (not UUID), thereby requesting a (ticket-granting-)ticket (TGT), denoted $Tkt_{A,KDS}$, targeted to the (same) KDS server. (The principal stringnames are here denoted simply ''*A*'' and ''KDS'', though in reality they are something like **/.../cellX/clientA** and **/.../cellX/krbtgt/cellX**. See Section 1.7 on page 32 and Section 1.18 on page 84). Included in this request is the desired lifetime $L_{A,KDS}$ of the ticket $Tkt_{A,KDS}$ and of the session key $K_{A,KDS}$ it carries (no ticket is to be honoured whose lifetime has *expired*), and a nonce $N_{A,AS}$ that *A* will use to associate this request message with its corresponding response message. Prior to DCE 1.1, this communication is *unprotected* (''in the clear'', ''unauthenticated'') — in particular, this figure currently specifies no ''pre-authentication'' (whereby *A* tries to prove to the KDS that it ''really is *A*'', in the sense of knowing the long-term key, K_A, of the principal it claims to be). Refer to Section 1.23.3 on page 114 for the DCE 1.1 (and newer versions) pre-authentication protocol.

- $A \leftarrow$ AS: *A*, $\{KDS, K_{A,KDS}, L_{A,KDS}, N_{A,AS}\}K_A$, $Tkt_{A,KDS}$

  1½ (*AS Response: Receive ticket to KDS*.)

  Upon receiving the AS Request message (allegedly) from *A*, the KDS server first generates a ''high-quality'' (cryptographically random) secure session key, $K_{A,KDS}$, to be used in protecting communications between *A* and the KDS. Then the KDS constructs the ticket, $Tkt_{A,KDS}$, naming the claimed principal *A* and containing the session key $K_{A,KDS}$, protecting $Tkt_{A,KDS}$ with the long-term key K_{KDS} of the KDS (that is, the ticket is targeted to the KDS server itself). Also included in this ticket are the identity of the KDS server to which it is

targeted, and the lifetime $L_{A,KDS}$ (which may be different from the lifetime requested by *A*, depending on policy, though this isn't indicated notationally), and its transit path P_X (which, in this intra-cell case, is merely a trivial path indicating this cell *X* only):

$$Tkt_{A,KDS} = KDS, \{A, K_{A,KDS}, L_{A,KDS}, P_X\}K_{KDS}$$

The KDS then communicates all this information back to *A*, including also the nonce $N_{A,AS}$ which associates this response with the request that triggered it. This AS Response message is protected with the long-term key K_A of the principal *A* claimed in the request (the KDS learns this key by retrieving it from the RS).

- $A \rightarrow$ TGS: $B, L_{A,B}, N_{A,TGS}, [\{EPAC\}K^{\lceil\rceil}_{A,KDS},] Tkt_{A,KDS}, \{A, C_{A,TGS}, [K^{\wedge}_{A,KDS},] T_{A,TGS}\} K_{A,KDS}$

  2 (*TGS Request: Ask for ticket to server*.)

  When *A* receives the AS Response message from the KDS, *A* can correctly interpret it (that is, decrypt it, and thereby learn the session key $K_{A,KDS}$) only if it ''really is *A*'' (in the sense of knowing the named client's long-term key K_A). For the same reason, *A* is convinced that the AS Response message really did come from the genuine KDS server (that is, no ''reverse authenticator'' is necessary here — the message itself is ''self-mutually-authenticating''). *A*'s next step is to send a TGS Request message to the KDS, containing the principal stringname (not UUID) of the desired target (non-KDS) server *B*, and $Tkt_{A,KDS}$, thereby requesting a ticket to *B*, denoted $Tkt_{A,B}$. As with AS Requests, a desired lifetime $L_{A,B}$ and a nonce $N_{A,TGS}$ are included in the TGS Request; and *A* can also optionally include some *authorisation data*, denoted here ''EPAC'' (evoking ''extended privilege attribute certificate''), to be more properly introduced in Section 1.6 on page 25 (this option is rarely used in TGS Requests — see below in this paragraph for the meaning of the notation $K^{\lceil\rceil}_{A,KDS}$). But unlike an AS Request, the TGS Request is *protected* (or ''authenticated''), by including a (*forward*) *authenticator* in it. This authenticator contains the identity of *A* and a timestamp $T_{A,TGS}$ which indicates the current time-of-day on *A*'s system (and also on KDS' system, modulo an allowable *clock skew*, assuming ''loosely (on the order of a few minutes) synchronised clocks'') — it is primarily these two items that ''authenticate *A* to KDS'', in the sense that they prove to the KDS that ''*A* really does know the session key $K_{A,KDS}$, *now*''. Also included in the authenticator are a checksum $C_{A,TGS}$ binding the otherwise-unprotected (unencrypted) data in the message ($B, L_{A,B}, N_{A,TGS}$) to the authenticator, and optionally a conversation key $K^{\wedge}_{A,KDS}$ generated by *A*. If *A* includes the optional key $K^{\wedge}_{A,KDS}$ in the TGS Request, that key is used to protect the authorisation data (in the EPAC, with an EPAC seal), otherwise the key $K_{A,KDS}$ is used (that's the meaning of the notation $K^{\lceil\rceil}_{A,KDS}$ above). The authenticator is encrypted with the key $K_{A,KDS}$.

- $A \leftarrow$ TGS: $A, \{B, K_{A,B}, L_{A,B}, N_{A,TGS}\} K^{\lceil\rceil}_{A,KDS}, Tkt_{A,B}$

  2½ (*TGS Response: Receive ticket to server*.)

  The KDS now generates a session key, $K_{A,B}$, which is the actual ''physical manifestation'' of the logical notion of ''authentication'' between *A* and *B*; it will be used to protect the initial communications between *A* and *B*, and optionally to support negotiation of subsequent conversation (''true session'') key(s) ($K^{\wedge}_{A,B}$ and/or $K^{\wedge\wedge}_{A,B}$) between them. Then the KDS constructs the ticket $Tkt_{A,B}$, naming *A* and containing $K_{A,B}$, as well as the lifetime $L_{A,B}$, the authorisation data PAC if it had been included in the TGS Request, and the transit path P_X. $Tkt_{A,B}$ is protected with the long-term key K_B of *B* (that is, the KDS targets this ticket to *B*):

  $$Tkt_{A,B} = Tkt_{A,X,B} = B, \{A, K_{A,B}, L_{A,B}, [EPAC,] P_X\} K_B$$

  The KDS communicates all this information to *A* in the TGS Response message. As with the AS Response message, the TGS Response message requires no ''reverse authenticator''. The KDS uses the key $K^{\lceil\rceil}_{A,KDS}$ to protect the TGS Response message.

- $A \rightarrow B$: $Tkt_{A,B}$, {*A*, [$C_{A,B}$,] [$K^{\wedge}_{A,B}$,] $T_{A,B}$} $K_{A,B}$
- $A \leftarrow B$: {[$K^{\wedge\wedge}_{A,B}$,] $T_{A,B}$} $K_{A,B}$
- $A \rightarrow B$: {*Service Request(s) / Application-level Data*} $K^{[\wedge[\wedge]]}_{A,B}$
- $A \leftarrow B$: {*Service Response(s) / Application-level Data*} $K^{[\wedge[\wedge]]}_{A,B}$

3 (*Service Request/Response: Get service from server*.)

Finally, *A* sends a message to *B*, containing the ticket $Tkt_{A,B}$ and an authenticator (including in it an optional conversation key $K^{\wedge}_{A,B}$ if it so desires), protecting the authenticator with $K_{A,B}$. *B* can correctly interpret (decrypt) $Tkt_{A,B}$, thereby learning the named client *A* and the session key $K_{A,B}$ (and the other information in $Tkt_{A,B}$), only if it ''really is *B*'' (in the sense of knowing the targeted server's long-term key K_B). Using the session key $K_{A,B}$, *B* then decrypts the authenticator, which completes the authentication of *A* to *B*. The (mutual) authentication of *B* to *A* is finally completed when *B* returns to *A* the *reverse authenticator*, containing the same timestamp that *A* had sent to *B*, $T_{A,B}$, together with an optional conversation key $K^{\wedge\wedge}_{A,B}$ if *B* so desires. From this point on, *A* and *B* can now engage in protected communications (that is, service request/responses) using either the session key $K_{A,B}$ or one of the ''negotiated'' conversation keys $K^{\wedge}_{A,B}$ or $K^{\wedge\wedge}_{A,B}$ (exactly which of these keys is used is an application-dependent determination) — that's the meaning of the notation $K^{[\wedge[\wedge]]}_{A,B}$. (The conversation key could be further negotiated at application-level; for example ''exclusive-OR of $K^{\wedge}_{A,B}$ and $K^{\wedge\wedge}_{A,B}$'', but that is beyond the scope of this specification.) There is no need for *A* and *B* to make further exchanges of tickets, authenticators, and so on, until the conversation key $K^{[\wedge[\wedge]]}_{A,B}$ times out (as indicated by the lifetime $L_{A,B}$). (As an optimisation, an initial service request could have been piggy-backed with the $Tkt_{A,B}$ and authenticator message, and its corresponding service response piggy-backed with the reverse authenticator message — but such piggy-backing is application-dependent, not an integral feature of the security architecture.)

Note that the *programming model* supported by DCE hides these complications behind API and RPC interfaces, and does not expose them to application developers. Applications invoke the KDS only indirectly through the RPC facility (not through a direct API), by ''annotating the binding handle with **rpc_c_authn_dce_secret**'' — see Section 1.10 on page 54 and Section 1.17 on page 82, and the referenced X/Open DCE RPC Specification.

The KDS has the principal name **krbtgt/***cell-name* (within its cell), and it supports the **krb5rpc** RPC interface, which supports the following operation:

- *kds_request*( ) — send a message (via RPC, as always) to the KDS, requesting a ticket (or privilege-ticket — see Section 1.6 on page 25). The supported messages are those exhibited in the basic Kerberos protocol just described, namely:
  - — AS messages (together comprising the *AS Exchange*):
    - — *AS Request* — unauthenticated request to the AS, requesting an initial ticket (usually a ticket-granting-ticket, but occasionally a service-ticket).
    - — *AS Response* — Response to AS Request, returning the requested initial ticket.
  - — TGS messages (together comprising the *TGS Exchange*):
    - — *TGS Request* — Authenticated request to the TGS, requesting a subsequent (that is, non-initial) ticket (usually a service-ticket, but occasionally a ticket-granting-ticket).
    - — *TGS Response* — Response to TGS Request, returning the requested subsequent ticket.
  - — *KDS Error* — KDS error message, generated in response to either a failed AS Request or a failed TGS Request.

Note: In the current version of DCE, only an indirect RPC interface to the KDS is supported, but no true direct ''APIs'' (just the **rpc_c_authn_dce_secret** RPC ''annotation'' constant). Such APIs may be added in a future version.

1.6 Privilege (Authorisation) Service (PS)

The DCE *Privilege* (or *Authorisation*) *Service* (PS) (or *Privilege-ticket-granting Service* (PTGS)) manages the privilege attributes associated to principals, and issues credentials witnessing these privileges. The credentials issued by the PS are called *Privilege Attribute Certificates* (PACs) prior to DCE 1.1. As of DCE 1.1, the PAC has been extended to include additional attributes stored in the RS, and is called an EPAC. PACs are carried in (and protected by) *privilege-tickets*. EPACS are not carried in *privilege-tickets*. Instead, a cryptographic checksum of the EPAC is generated by the Privilege Server when a *privilege-ticket* is created. This checksum is called the *seal* of the EPAC, and this *seal* is what is carried in the privilege-ticket for DCE 1.1 and newer versions. (*PACs may still be carried in the privilege-ticket for legacy reasons.*) In either case, privilege-tickets represent the ''authorisation identity'' of clients, represented by UUIDs (as opposed to their ''authentication identity'', represented as a stringname, in non-privilege-tickets). So in this sense (that is, the sense of identifying the client by UUIDs instead of by stringname, except optionally), privilege-tickets are sometimes said, by abuse of language, to be ''anonymous''.

Privilege-tickets are instances of tickets, and as such participate all the same general concepts; in particular, a privilege-ticket that is targeted to a KDS server is called a *privilege-ticket-granting-tickets* (PTGT). But there are three significant features about privilege-tickets that distinguish them from non-privilege-tickets:

- The most obvious difference is that privilege-tickets contain a PAC (prior to DCE 1.1) or an EPAC *seal* (for DCE 1.1 and newer versions), but non-privilege-tickets don't.
- The most counter-intuitive difference is that, although privilege-tickets contain a ''named client'' field (a stringname, not a UUID) and either a PAC containing ''client privileges'' (UUIDs, not stringnames) or an EPAC seal containing ''extended client privileges'', a different ''client'' is referred to in each use of this word: *in a privilege-ticket, the ''named client'' is the PS in the server's cell, while the ''nominated client'' to which the PAC (or EPAC) refers (via its privilege attribute UUIDs) is the actual initiating client (or a delegate acting on its behalf) which is requesting access to a targeted server's protected resources*. It is by this means (that is, by checking that the privilege-ticket names the PS) that the targeted server can trust that the PS actually vouches (that is, is responsible) for the PAC (or EPAC). The terminology *nominated client* is therefore introduced to denote the client to which the PAC (or EPAC) refers (via its privilege attribute UUIDs), and the PAC (or EPAC) is said to **nominate** *A*. (The nominated client is able to use the privilege-ticket because of protocol guarantees that it knows the session key carried in the privilege-ticket, so it can, in this sense, ''legitimately impersonate'' the named client, the PS in the server's cell.)
- The final difference between non-privilege-tickets and privilege-tickets is that non-privilege-tickets carry a *transit path* field (that is, a record of the trust chain involved in an authentication) whose trust the target server is responsible for evaluating, but privilege-tickets do not carry such a transit path (or rather, they don't carry a ''meaningful'' transit path: the transit path is present, but it always indicates the trivial path consisting only of the target server's cell). The responsibility for evaluating the trust path falls, not to the target server itself, but to the PS in the target server's cell (which ''consumes'' the transit path in doing so). That is to say, in this sense, privilege-tickets are trusted by the targeted server by virtue of the trust the target server has in the PS in its cell (in particular, the PS vouches for the identity of the nominated client and its cell, which is still projected to the target server in the privilege-ticket's PAC (or EPAC seal)).

PACs contain *privilege attributes* (that is, client-side access control information — that portion of the client's credentials to be used in server-based access control decisions), consisting of:

- *Authentication Flag*

Boolean flag, communicated to target servers, indicating whether or not this PAC has been authenticated by the TCB (and therefore whether or not the server can trust the PAC, depending on security policy). Clients have the ability to send unauthenticated (said to be ''merely *asserted*'') PACs to servers, but these must be viewed suspiciously by the servers. Typically, server security policy will require that unauthenticated requests are rejected outright, although a special ''UNAUTHENTICATED ACL Entry'' is supported to deal with unauthenticated requests if server security policy allows this — see Section 1.8.1 on page 40.

- *(Local) Cell UUID*

  UUID of the cell to which the client ''belongs'', and hence the cell whose PS has initially vouched for the contents of this PAC. (In a cross-cell operation, PSs in multiple cells vouch for the PAC's contents.) The Cell UUID uniquely determines this cell, to which the PAC is said to *refer*.

- *Principal UUID*

  UUID of the principal for which this PAC contains privilege attributes. The ordered pair <(Local) Cell UUID, Principal UUID> uniquely determines this principal, to which the Principal UUID and the PAC are said to *refer*.

- *Primary Group UUID*

  UUID identifying the ''primary'' or ''main'' group of which the principal is a member. The ordered pair <(Local) Cell UUID, Primary Group UUID> uniquely determines this group, to which the Primary Group UUID and the PAC are said to *refer*.

- *Local Secondary Group UUIDs*

  List (unordered set of distinct elements, possibly empty) of UUIDs of the ''local'' ''secondary'' groups of which the principal is a member. Each ordered pair <(Local) Cell UUID, Local Secondary Group UUID>, uniquely determines such a group, to which the Local Secondary Group UUID is said to *refer*, and to all of which the PAC is said to *refer*.

- *Foreign Secondary Group IDs*

  List of ordered pairs of two UUIDs, <*Foreign Cell UUID*, *Foreign Secondary Group UUID*>, of the ''foreign'' ''secondary'' groups of which the principal is a member. (''Foreign'' in the sense of PACs has the commonsense meaning that the Foreign Cell UUID is not equal to the (Local) Cell UUID. However, it is not forbidden for a PAC to contain a Foreign Secondary Group ID whose Foreign Cell UUID component is equal to the (Local) Cell UUID — it is just inefficient to do so.) Each such ordered pair uniquely determines such a group, to which the Foreign Secondary Group ID is said to *refer*, and to all of which the PAC is said to *refer*.

In addition to the identity and group membershop information present in a DCE 1.0 format PAC, the extended PAC (EPAC) contains optional delegation controls, optional and required restrictions (both of which are discussed under the topic, ''Delegation'', in this chapter), and extended attributes. EPACS also require the following:

- *Global Group Name*

  Name consisting of the UUID identifying the cell to which the client, *acting as a delegate*, belongs, and hence the cell whose PS has initially vouched for the contents of this EPAC. Also the UUID identifying the ''primary'' or ''main'' group of which the principal is a member.

- *Global Principal Name*

  Name consisting of the UUID identifying the cell to which the client, *acting as a delegate*, belongs. Also the UUID identifying the principal.

For more information on Global Group Names and Global Principal Names refer to Chapter 12 on page 489 and Chapter 17 on page 705.

(The distinction between primary groups and secondary groups is made for administrative purposes only, and is largely historical. In particular, the two kinds of groups are not discriminated between in the Common Access Determination Algorithm (see Section 1.9.1 on page 48).)

Note: As is seen from the structure of PACs and EPACs above (see also Section 5.2.5 on page 280, as well as the Common Access Determination Algorithm in Section 8.2 on page 321), authorisation identities in DCE are represented not by a *single* UUID, but by a *pair* of UUIDs: <Cell UUID, Subject UUID> (where, for the purposes of this discussion, ''subject'' means principal or group). At first glance this may seem odd: since UUIDs are ''unique in space and time'', one wonders why two UUIDs are needed to identify one security subject. The answer has to do not with identification *per se*, but in the *trust* to be invested in the uniqueness of UUIDs (especially in an environment where untrusted other parties may be generating some of the UUIDs), and especially in the containment of damage in the event of UUID non-uniqueness. For, assuming a single-UUID scheme, consider the situation where a cell *X* were compromised, but that no other cell were aware of this compromise. In that situation, the compromiser of cell *X* could then assign arbitrary (bogus) UUIDs to principals and groups in *X* — these could even be the (otherwise genuine) UUIDs assigned to principals and groups in (any) other cells. The (bogus) clients from cell *X* would then be able to authenticate to servers in *X* and in those other cells that ''trust *X*'' (through (potentially chains of) cross-registrations), and would then be able to access *all* protected objects in those servers (because the server would be unable to distinguish the bogus UUIDs from the genuine ones). That is to say, under a single-UUID scheme, the compromise of a single cell would compromise the security of *all* protected objects in all cells that ''trust *X*''. This is unacceptable. In the double-UUID scheme, this doesn't happen: the compromise of cell *X* compromises the security, not of *all* protected objects in cells that ''trust *X*'', but only of those protected objects (in cells that ''trust *X*'') that themselves ''trust *X*'', in the sense of only those protected objects whose very ACLs grant access to subjects from *X*. This is a more acceptable containment of damage.

The manner in which privilege-tickets figure into the DCE security environment is that the basic (intra-cell) Kerberos authentication protocol is extended to include the PS, as outlined below. Consider a client *A* that desires to obtain service from a server *B* (see Figure 1-3 on page 28).

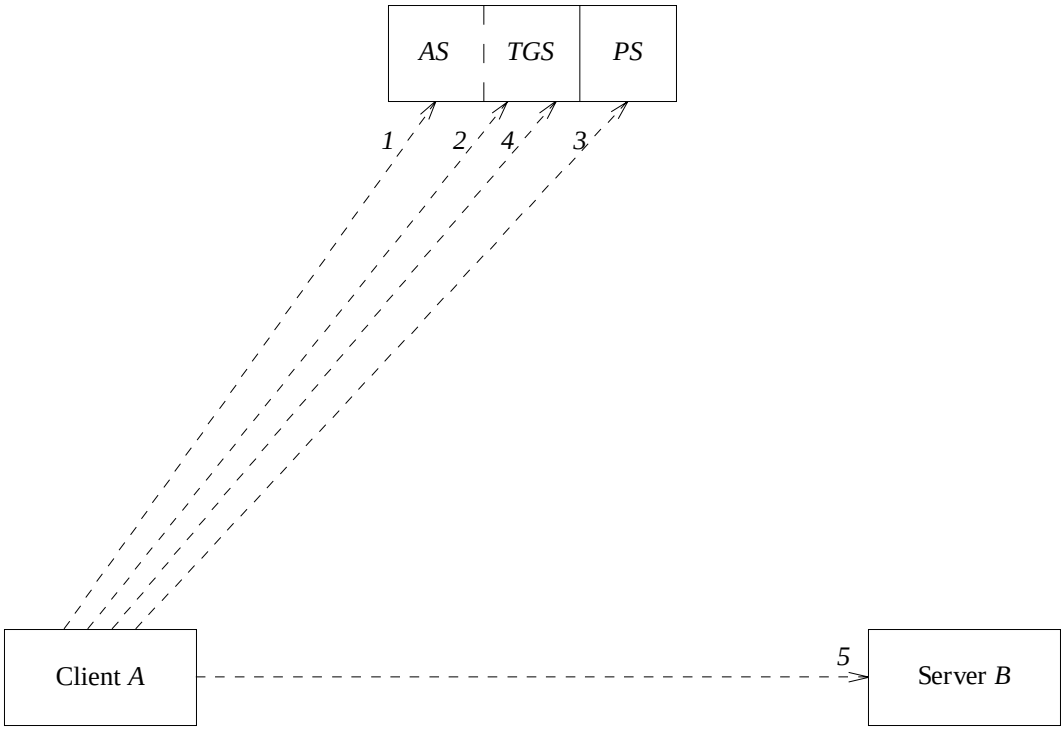


Figure 1-3 KDS+PS Protocol

- $A \rightarrow$ AS: *A*, KDS, $L_{A,KDS}$, $N_{A,AS}$
- $A \leftarrow$ AS: *A*, {KDS, $K_{A,KDS}$, $L_{A,KDS}$, $N_{A,AS}$} K_A, $Tkt_{A,KDS}$

  1 (*AS Request/Response: Get ticket to KDS*.)

  First, *A* obtains a ticket, $Tkt_{A,KDS}$, from the AS (naming *A*, targeted to the KDS, and containing a session key $K_{A,KDS}$), exactly as in the basic authentication protocol (above).

- $A \rightarrow$ TGS: PS, $L_{A,PS}$, $N_{A,TGS}$, $Tkt_{A,KDS}$, {*A*, $C_{A,TGS}$, [$K^{\wedge}_{A,KDS}$,] $T_{A,TGS}$} $K_{A,KDS}$
- $A \leftarrow$ TGS: *A*, {PS, $K_{A,PS}$, $L_{A,PS}$, $N_{A,TGS}$} $K^{[\wedge]}_{A,KDS}$, $Tkt_{A,PS}$

  2 (*TGS Request/Response: Get ticket to PS*.)

  Next, *A* sends its $Tkt_{A,KDS}$ and the principal stringname of the PS to the KDS, requesting a ticket, $Tkt_{A,PS}$, to the PS. Still following the basic authentication protocol, the KDS obliges by returning the requested $Tkt_{A,PS}$ (naming *A*, targeted to the PS, and containing a session key $K_{A,PS}$) to *A*.

- $A \rightarrow$ PS: KDS, $L^{\sim}_{A,KDS}$, $N_{A,PS}$, $Tkt_{A,PS}$, {*A*, $C_{A,PS}$, [$K^{\wedge}_{A,PS}$,] $T_{A,PS}$} $K_{A,PS}$

  3 (*PS Request: Ask for privilege-ticket to KDS*.)

  Now, *A* sends $Tkt_{A,PS}$ to the PS, thereby requesting a privilege-(ticket-granting-)ticket (PTGT), denoted $PTkt_{A,KDS}$, targeted to the KDS. The PS decrypts $Tkt_{A,PS}$, learning *A*'s stringname and the session key $K_{A,PS}$. (The ''~''-notation just distinguishes a new occurrence of a data item — in this case, the lifetime of a key to be shared between *A* and the KDS — from a similar one that has occurred previously in this run of the protocol.)

- $A \leftarrow$ PS: A, {PS, $K^{\sim}_{A,KDS}$, $L^{\sim}_{A,KDS}$, $N_{A,PS}$} $K^{[\hat{}]}_{A,PS}$, $PTkt_{A,KDS}$

  3½ (*PS Response: Receive privilege-ticket to KDS*.)

  The PS constructs PAC_A from A's privilege information (which it retrieves from the RS), and then constructs the privilege-ticket, $PTkt_{A,KDS}$. This $PTkt_{A,KDS}$ *names* the PS itself (not A), contains PAC_A (that is, *nominates* A), contains a new session key $K^{\sim}_{A,KDS}$ (generated by the KDS), and is protected with the long-term key, K_{KDS}, of the KDS:

  $PTkt_{A,KDS}$ = KDS, {PS, $K^{\sim}_{A,KDS}$, $L^{\sim}_{A,KDS}$, PAC_A, P_X} K_{KDS}

  (The mechanics of how this $PTkt_{A,KDS}$ gets generated — in particular, how its encrypted part gets encrypted in the key K_{KDS} — is implementation-dependent, and not specified in this document. One method is that the encryption key K_{KDS} can be shared between the KDS and PS (the manner is unspecified here, but it must be done in some secure manner — for example, the PS and KDS could be co-located on the same host, or even in the same process). Another method (usable in implementations where the KDS and PS are not co-located) is that the PS sends a TGS Request including PAC_A to the KDS (PS $\rightarrow$ TGS: KDS, $L_{PS,KDS}$, $N_{PS,TGS}$, $\{PAC_A\}K^{[\hat{}]}_{PS,KDS}$, $Tkt_{PS,KDS}$, {PS, $C_{PS,TGS}$, $[K^{\hat{}}_{PS,KDS},]$ $T_{PS,TGS}$} $K_{PS,KDS}$), receiving in the corresponding TGS Response the ticket $Tkt^{\sim}_{PS,KDS}$ = KDS, {PS, $K_{PS,KDS}$, $L_{PS,KDS}$, PAC_A, P_X}K_{KDS} — this ticket then serves as the required $PTkt_{A,KDS}$.) In particular, note that the PAC_A in $PTkt_{A,KDS}$ contains UUIDs describing A's privilege attributes for authorisation purposes, but does not necessarily contain the principal stringname of A (though it may do so optionally) for authentication purposes. The PS returns $K^{\sim}_{A,KDS}$ and $PTkt_{A,KDS}$ to A (protecting this message with $K^{[\hat{}]}_{A,PS}$). As with AS and TGS Responses, the PS Response requires no ''reverse authenticator''.

- $A \rightarrow$ TGS: B, $L_{A,B}$, $N^{\sim}_{A,TGS}$, $PTkt_{A,KDS}$, {PS, $C^{\sim}_{A,TGS}$, $[K^{\sim\hat{}}_{A,KDS},]$ $T^{\sim}_{A,TGS}$} $K^{\sim}_{A,KDS}$

  4 (*TGS Request: Ask for privilege-ticket to server*.)

  At this point, A proceeds along the same lines as the basic authentication protocol, but now using the privilege-ticket $PTkt_{A,KDS}$ instead of the non-privilege-ticket $Tkt_{A,KDS}$ (it is A's knowledge of $K^{\sim}_{A,KDS}$ that enables it to use $PTkt_{A,KDS}$, by ''legitimately posing'' as PS). Thus, A sends a message to the KDS, containing the principal stringname of B and the $PTkt_{A,KDS}$, requesting a privilege-ticket, $PTkt_{A,B}$ targeted to B. (Note incidentally that the identity in the authenticator is that of the PS, not A, because the named client in the privilege-ticket $PTkt_{A,KDS}$ is PS, not A.) This message is protected with the session key $K^{\sim}_{A,KDS}$.

- $A \leftarrow$ TGS: PS, {B, $K_{A,B}$, $L_{A,B}$, $N^{\sim}_{A,TGS}$} $K^{\sim[\hat{}]}_{A,KDS}$, $PTkt_{A,B}$

  4½ (*TGS Response: Receive privilege-ticket to server*.)

  The KDS treats this request just as it would an ordinary request (that is, one using a non-privilege-ticket instead of a privilege-ticket), with the additional function of ''blindly copying'' (that is, without interpreting) the PAC_A received in $PTkt_{A,KDS}$ into the privilege-ticket, $PTkt_{A,B}$. That is, the KDS constructs a $PTkt_{A,B}$, with PS as its named client, containing a session key $K_{A,B}$ and PAC_A, and protected with the long-term key, K_B, of B:

  $PTkt_{A,B}$ = B, {PS, $K_{A,B}$, $L_{A,B}$, PAC_A, P_X} K_B

  The KDS returns $K_{A,B}$ and $PTkt_{A,B}$ to A, in a message protected by $K^{\sim[\hat{}]}_{A,KDS}$.

- $A \rightarrow B$: $PTkt_{A,B}$, {PS, $[C_{A,B},]$ $[K^{\hat{}}_{A,B},]$ $T_{A,B}$} $K_{A,B}$

- $A \leftarrow B$: {$[K^{\hat{}\hat{}}_{A,B},]$ $T_{A,B}$} $K_{A,B}$

- $A \rightarrow B$: {*Service Request(s) / Application-level Data*} $K^{[\hat{}[\hat{}]]}_{A,B}$

- $A \leftarrow B$: {*Service Response(s) / Application-level Data*} $K_{A,B}^{[\hat{}[\hat{}]]}$

  5 (*Service Request/Response: Get service from server*.)

  Finally, *A* sends its privilege-ticket, $PTkt_{A,B}$, to *B* (protecting this message with $K_{A,B}$). *B* decrypts $PTkt_{A,B}$, learning its named client (which must be PS, so *B* knows it can trust PAC_A), PAC_A nominating *A* (though not the principal stringname of *A*, except optionally), and the session key $K_{A,B}$. *A* and *B* can now proceed with their interactions exactly as in the basic authentication protocol, with the additional functionality that *B* can make its authorisation decisions based on the trusted PAC_A information contained in $PTkt_{A,B}$.

As with the KDS, no APIs to the PS are directly supported in DCE. Instead, the use of the PS is signaled in RPC applications by the use of the **rpc_c_authz_dce** identifier (see Section 1.10 on page 54 and Section 1.17 on page 82, and the referenced X/Open DCE RPC Specification).

The PS has the principal name **dce-ptgt** (within its cell), and it supports the **rpriv** RPC interface, which supports the following operations:

- *ps_request_ptgt*( ) — request a privilege-ticket-granting-ticket (as in the protocol just described).
- *ps_request_eptgt*( ) — request an extended privilege certificate to the ticket-granting service (PS).
- *ps_request_become_delegate*( ) — request a privilege-ticket-granting-ticket for an intermediary caller (server) so the intermediary can become a delegate for the caller.
- *ps_request_become_impersonator*( ) — request a privilege-ticket-granting-ticket for an intermediary caller (server) so the intermediary can become an impersonator for the caller.

1.6.1 Name-based versus PAC-based Authorisation

Note: Prior to DCE 1.1, name-based authorisation was included in DCE primarily for support of legacy applications only; its use for any other purpose was discouraged. However, in DCE 1.1 and newer versions, since delegation is being supported, name-based authorisation is being used to ensure the integrity of the arguments across the network due to the introduction of delegated identities.

In addition to the *PAC-based* authorisation service described above, DCE also supports another authorisation service, said to be (*string*)*name-based*. It is signaled in RPC applications by the use of the identifier **rpc_c_authz_name**. Name-based authorisation is a very primitive service compared to the sophisticated privilege service described above, in several senses (some terminology is used here that won't be introduced until later sections):

- It is based solely on KDS-related protocols, not PS-related ones (see also the cross-cell protocol in Section 1.7 on page 32); that is, the PS is *not visited* in the course of a name-based authorisation. In technological terms: no privilege-tickets are involved, only non-privilege-tickets; thus the ticket's ''nominated client'' is not that referred to by a PAC (because there is no PAC), but is instead the ''named client'', identified by principal stringname.
- It is not flexible or extensible, because the only ''privilege attribute'' (or ''authorisation information'') projected from the client to the server is the client's principal stringname, not a UUID profile (as in a PAC). In particular, there is no support for ''name-based groups''.
- There is no DCE support for ''name-based ACLs'', nor for ''name-based permissions'' or ''name-based access control managers'', ''name-based access control editors'' or ''name-based common access determination algorithm''.

- It doesn't afford good cross-cell security, because the PS isn't visited. The point here is that while the KDS specifications have it blindly copying authorisation information and evaluating trust chains at the level of immediate cross-cell links, the PS specifications have it ''vetting'' (or ''modulating'', or ''tempering'') authorisation information and evaluating the global shape of trust chains. This is discussed in Section 1.7 on page 32.

1.7 Cells — Cross-cell Authentication and Authorisation

A *cell* (sometimes called *realm* or *domain*, when the focus is solely on security) is the basic unit of configuration and administration in a DCE environment. Each cell contains one RS/KDS/PS triple (potentially replicated). In the preceding sections, only the *per-cell* nature of the DCE security features has been discussed. In this section, the manner in which these features are extended *across cells* is explained. This creates the effect of a multi-cell DCE TCB (though different levels of trust may be invested in different cells). As will be seen, principals in distinct cells can establish trust chains to one another. But such trust chains are inherently less trustworthy than trust relationships within a single cell. This is due simply to general security principles (and is not specific to DCE security), namely cross-cell trust chains are longer than intra-cell ones, and trust chains are in general (by the ''fail-safe'' principle) no more trustworthy than their weakest link (and the longer the chain the higher the likelihood of some link being compromised). This simply reinforces the general security principle that entities ''near'' to ''self'' are more trustworthy (easier to protect) than entities ''farther away''.

Consider, therefore, a client *A* in cell *X*, and a server *B* in cell *Y*. Denote each cell's security services with subscripts (for example, KDS_X, $KDS_{X,Y}$ — but when they appear in subscripts themselves they will be upgraded to avoid embedded subscripts; for example, K_{KDSX}, K_{KDSXY}). The problem is to extend the single-cell security model to this multi-cell case. In terms of *trust chains*, *A* trusts TCB_X, and *B* trusts TCB_Y, so what remains is to establish a trust link between TCB_X and TCB_Y. In terms of *keys*, in order for *A* and *B* to communicate securely, they need to share a session key they both trust, and it is this key distribution problem that is at the crux of the cross-cell security model. In terms of *tickets*, *A* must present to *B* a privilege-ticket protected with *B*'s long-term key, but the normal distributor of tickets to *A*, KDS_X, does not know *B*'s long-term key (nor should it — only KDS_Y should know *B*'s key).

The solution involves *cross-registering* the cell principals KDS_X and KDS_Y in one another's cells, using (two copies of) a ''surrogate'' cell principal, as follows (see Figure 1-4 on page 33). In order for clients in cell *X* to be able to (mutually) authenticate to servers in cell *Y*, KDS_Y is endowed with an additional **surrogate** principal identity, denoted $KDS_{X,Y}$, and registered in RS_Y with a new long-term key K_{KDSXY} (distinct from the long-term key, K_{KDSY}, of KDS_Y itself — which is also denoted $KDS_{Y,Y}$ in this context); and in addition, this *same* surrogate principal $KDS_{X,Y}$ is also registered with the *same* key K_{KDSXY} in RS_X, as another copy of the surrogate of the $KDS_{X,Y}$ in RS_Y. That is, it is the *cross-cell surrogate (double) principal* $KDS_{X,Y}$ which mediates the $KDS_X \rightarrow KDS_Y$ *trust link* (as detailed below, the point is that KDS_Y knows the long-term key of the principal $KDS_{X,Y}$ in the *foreign* cell *X*). As a *communicating* entity (that is, as an RPC server), $KDS_{X,Y}$ is typically implemented to be the *same* as KDS_Y (in cell *Y*) or as KDS_X (in cell *X*), respectively.

Note: The use of arrow notation to denote a trust link, ''$KDS_X \rightarrow KDS_Y$'', is distinct from, and not to be confused with, the use of arrow notation to denote communications messages.)

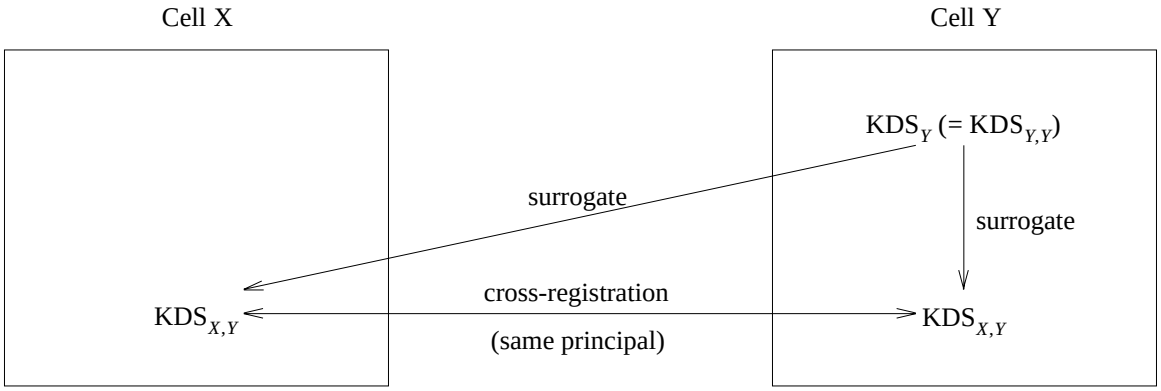


Figure 1-4 Cross-registration Mediating Cross-cell Trust Link

The above discussion concerned only the ''unilateral'' mediation of trust; that is, from clients in *X* to servers in *Y*. Conversely, for clients in *Y* to be able to authenticate to servers in *X*, KDS$_X$ is endowed with a similar cross-cell surrogate double principal identity, denoted KDS$_{Y,X}$, registered in both RS$_X$ and in RS$_Y$, with a new long-term key K$_{KDSYX}$. That is, KDS$_{Y,X}$ mediates the KDS$_Y$ → KDS$_X$ trust link. In general (that is, for a given pair of cells *X* and *Y*), zero, one or two of the principals KDS$_{X,Y}$ and KDS$_{Y,X}$ may exist. And in the case where both exist, they may be equal (KDS$_{X,Y}$ = KDS$_{Y,X}$; that is, K$_{KDSXY}$ = K$_{KDSYX}$) or they may be *distinct* (KDS$_{X,Y}$ ≠ KDS$_{Y,X}$; that is, K$_{KDSXY}$ ≠ K$_{KDSYX}$). These are called the *symmetric* (or *one-key*) and *asymmetric* (or *two-key*) cases of *mutual trust peers*, respectively. Cross-registration is an *explicit* expression of trust; cells that do not trust one another do not cross-register with one another.

The *naming* model for the principal stringnames of the cross-cell surrogates involved in cross-registration is specified as follows. Suppose the cell name of *X* is (in full DCE syntax) **/.../cellX** and the cell name of *Y* is **/.../cellY**, so that the per-cell principal name of KDS$_X$ (= KDS$_{X,X}$) is **krbtgt/cellX** (within RS$_X$) and the per-cell principal name of KDS$_Y$ (= KDS$_{Y,Y}$) is **krbtgt/cellY** (within RS$_Y$) (that is, in full DCE syntax, **/.../cellX/krbtgt/cellX** and **/.../cellY/krbtgt/cellY**, respectively). Then within RS$_X$, *both* KDS$_{X,Y}$ and KDS$_{Y,X}$ are identified by the single per-cell principal name **krbtgt/cellY** (that is, in full DCE syntax, **/.../cellX/krbtgt/cellY**). And within RS$_Y$ both KDS$_{X,Y}$ and KDS$_{Y,X}$ are named **krbtgt/cellX** (that is, **/.../cellY/krbtgt/cellX**).

Note: (The following comments are worded in terms of RS$_X$, though the same comments apply to RS$_Y$, of course.) According to the above naming model, the (potentially distinct) cross-cell principals KDS$_{X,Y}$ and KDS$_{Y,X}$ in RS$_X$ have the *same stringname*, namely **/.../cellX/krbtgt/cellY**. Thus, those principals (and the keys associated to them) cannot be distinguished within RS$_X$ by their stringnames. In the symmetric case this causes no confusion, because there is only one long-term key associated to the name **/.../cellX/krbtgt/cellY**. In the asymmetric case (if it is supported by a given implementation), some administrative means of distinguishing between the (distinct) ''outgoing'' key (K$_{KDSXY}$) and ''incoming'' key (K$_{KDSYX}$) must be provided by RS$_X$. In practice, the policy of most cells is to support only the symmetric case (and indeed, most implementations of DCE do not provide the required administrative means to support the asymmetric case). That is, while the protocols as specified in this specification support both symmetric and asymmetric mutual trust peers, most cells (and implementations) support only the symmetric case. This has implications for interoperability between any pair of cells supporting different policies (namely, if one of the cells requires that its trust relationship with the other cell be symmetric, while that other cell requires that the relationship be asymmetric, then the two cells cannot be mutual trust peers — full, unrestricted bidirectional authentication between

clients and servers in the two cells cannot exist).

The consequence of cross-registration is to enable the establishment of trust relationships between clients and servers in different cells, as the following steps outline. The discussion in this outline is briefer than the previous ones, since so much of it has already been explained in the previous outlines (see Figure 1-5).

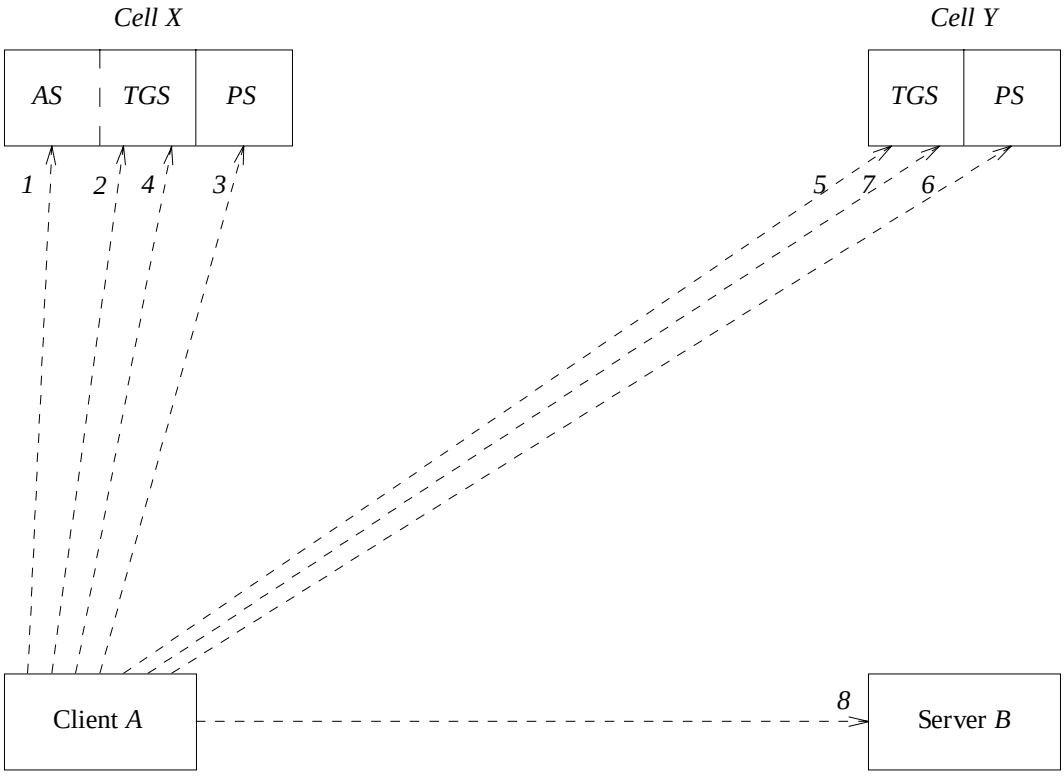


Figure 1-5 Cross-cell Protocol (Single-hop)

- $A \rightarrow \mathrm{AS}_X$: A, KDS_X, $\mathrm{L}_{A,\mathrm{KDS}X}$, $\mathrm{N}_{A,\mathrm{AS}X}$
- $A \leftarrow \mathrm{AS}_X$: A, $\{\mathrm{KDS}_X, \mathrm{K}_{A,\mathrm{KDS}X}, \mathrm{L}_{A,\mathrm{KDS}X}, \mathrm{N}_{A,\mathrm{AS}X}\}\,\mathrm{K}_A$, $\mathrm{Tkt}_{A,\mathrm{KDS}X}$
- $A \rightarrow \mathrm{TGS}_X$: PS_X, $\mathrm{L}_{A,\mathrm{PS}X}$, $\mathrm{N}_{A,\mathrm{TGS}X}$, $\mathrm{Tkt}_{A,\mathrm{KDS}X}$, $\{A, \mathrm{C}_{A,\mathrm{TGS}X}, [\hat{\mathrm{K}}_{A,\mathrm{KDS}X},]\,\mathrm{T}_{A,\mathrm{TGS}X}\}\,\mathrm{K}_{A,\mathrm{KDS}X}$
- $A \leftarrow \mathrm{TGS}_X$: A, $\{\mathrm{PS}_X, \mathrm{K}_{A,\mathrm{PS}X}, \mathrm{L}_{A,\mathrm{PS}X}, \mathrm{N}_{A,\mathrm{TGS}X}\}\,\mathrm{K}^{[\hat{}]}_{A,\mathrm{KDS}X}$, $\mathrm{Tkt}_{A,\mathrm{PS}X}$
- $A \rightarrow \mathrm{PS}_X$: KDS_X, $\tilde{\mathrm{L}}_{A,\mathrm{PS}X}$, $\mathrm{N}_{A,\mathrm{PS}X}$, $\mathrm{Tkt}_{A,\mathrm{PS}X}$, $\{A, \mathrm{C}_{A,\mathrm{PS}X}, [\hat{\mathrm{K}}_{A,\mathrm{PS}X},]\,\mathrm{T}_{A,\mathrm{PS}X}\}\,\mathrm{K}_{A,\mathrm{PS}X}$
- $A \leftarrow \mathrm{PS}_X$: A, $\{\mathrm{PS}_X, \tilde{\mathrm{K}}_{A,\mathrm{KDS}X}, \tilde{\mathrm{L}}_{A,\mathrm{KDS}X}, \mathrm{N}_{A,\mathrm{PS}X}\}\,\mathrm{K}^{[\hat{}]}_{A,\mathrm{PS}X}$, $\mathrm{PTkt}_{A,\mathrm{KDS}X}$

  1; 2; 3 (*Get ticket to local KDS; Get ticket to local PS; Get privilege-ticket to local KDS*.)

  Client A in cell X desires a service from server B in a foreign cell Y. A begins, as usual, by acquiring its ticket to KDS_X, $\mathrm{Tkt}_{A,\mathrm{KDS}X}$, from AS_X, and its privilege-ticket, $\mathrm{PTkt}_{A,\mathrm{KDS}X}$, from PS_X, for services within cell X.

- $A \rightarrow \mathrm{TGS}_X$: B, $\mathrm{L}_{A,B}$, $\tilde{\mathrm{N}}_{A,\mathrm{TGS}X}$, $\mathrm{PTkt}_{A,\mathrm{KDS}X}$, $\{\mathrm{PS}_X, \tilde{\mathrm{C}}_{A,\mathrm{TGS}X}, [\tilde{\hat{\mathrm{K}}}_{A,\mathrm{KDS}X},]\,\tilde{\mathrm{T}}_{A,\mathrm{TGS}X}\}\,\tilde{\mathrm{K}}_{A,\mathrm{KDS}X}$
- $A \leftarrow \mathrm{TGS}_X$: PS_X, $\{\mathrm{KDS}_{X,Y}, \mathrm{K}_{A,\mathrm{KDS}XY}, \mathrm{L}_{A,\mathrm{KDS}XY}, \tilde{\mathrm{N}}_{A,\mathrm{TGS}X}\}\,\tilde{\mathrm{K}}^{[\hat{}]}_{A,\mathrm{KDS}X}$, $\mathrm{Tkt}_{A,X,\mathrm{KDS}XY}$

  4 (*Get ticket to foreign KDS*.)

Following the usual protocol, *A* now requests KDS_X for a privilege-ticket to *B*. *But instead of that*, KDS_X returns to *A* a **cross-cell referral (ticket-granting-)ticket**, $Tkt_{A,X,KDSXY}$, which names PS_X, nominates *A*, and is targeted to the *surrogate* principal $KDS_{X,Y}$ in cell *X*:

$$Tkt_{A,X,KDSXY} = KDS_{X,Y}, \{PS_X, K_{A,KDSXY}, L_{A,KDSXY}, PAC_{A,X}, P_{X,Y}\} K_{KDSXY}$$

As for any request for any ticket or privilege-ticket, KDS_X blindly copies $PAC_{A,X}$ from $PTkt_{A,KDSX}$ into $Tkt_{A,X,KDSXY}$, and returns it to *A* with a session key $K_{A,KDSXY}$ between *A* and $KDS_{X,Y}$. Note that $Tkt_{A,X,KDSXY}$ is not yet considered to be a ''privilege-ticket'' at this point, because it names PS_X instead of PS_Y. This indicates that PS_Y has not yet had an opportunity to ''vet'' (''modulate'', ''temper'') $PAC_{A,X}$ and the transit path $P_{X,Y}$ (indicating the trust link $TCB_X \rightarrow TCB_Y$).

- $A \rightarrow TGS_Y$: $PS_Y, L_{A,PSY}, N_{A,TGSY}, Tkt_{A,X,KDSXY}, \{PS_X, C_{A,TGSY}, [\hat{K}_{A,KDSXY},] T_{A,TGSY}\} K_{A,KDSXY}$
- $A \leftarrow TGS_Y$: $PS_X, \{PS_Y, K_{A,PSY}, L_{A,PSY}, N_{A,TGSY}\} K^{[\hat{}]}_{A,KDSXY}, Tkt_{A,X,Y,PSY}$

5 (*Get ticket to foreign PS*.)

Now, *A* presents this $Tkt_{A,X,KDSXY}$ to (not the surrogate $KDS_{X,Y}$ in cell *X*, but to) the surrogate $KDS_{X,Y}$ in cell *Y* (which recognises $Tkt_{A,X,KDSXY}$ as a well-formed ticket because it is protected with its long-term key, K_{KDSXY}), requesting a ticket, $Tkt_{A,X,Y,PSY}$, naming PS_X, nominating *A*, and targeted to PS_Y:

$$Tkt_{A,X,Y,PSY} = PS_Y, \{PS_X, K_{A,PSY}, L_{A,PSY}, PAC_{A,X}, P_{X,Y}\} K_{PSY}$$

KDS_Y ($KDS_{X,Y}$) issues this $Tkt_{A,X,Y,PSY}$ (blindly copying $PAC_{A,X}$ from $Tkt_{A,X,KDSXY}$ to $Tkt_{A,X,Y,PSY}$ in the usual manner) and returns it and its session key $K_{A,PSY}$ to *A*. Again, $Tkt_{A,X,Y,PSY}$ is not yet considered to be a ''privilege-ticket''.

- $A \rightarrow PS_Y$: $KDS_Y, \tilde{L}_{A,KDSY}, N_{A,PSY}, Tkt_{A,X,Y,PSY}, \{PS_X, C_{A,PSY}, [\hat{K}_{A,PSY},] T_{A,PSY}\} K_{A,PSY}$
- $A \leftarrow PS_Y$: $PS_X, \{PS_Y, \tilde{K}_{A,KDSY}, \tilde{L}_{A,KDSY}, N_{A,PSY}\} K^{[\hat{}]}_{A,PSY}, PTkt_{A,KDSY}$

6 (*Get privilege-ticket to foreign KDS*.)

Next, *A* presents this $Tkt_{A,X,Y,PSY}$ to PS_Y, requesting a privilege-ticket, $PTkt_{A,KDSY}$ (naming PS_Y, nominating *A*, and targeted to KDS_Y), in the usual manner:

$$PTkt_{A,KDSY} = KDS_Y, \{PS_Y, \tilde{K}_{A,KDSY}, \tilde{L}_{A,KDSY}, PAC_{A,Y}, P_Y\} K_{KDSY}$$

PS_Y honours this request, but since RS_Y does not hold *A*'s privilege attributes, PS_Y instead retrieves $PAC_{A,X}$ from $Tkt_{A,X,Y,PSY}$, and inspects it, in the process *vetting* (*modulating*, *tempering*) it for use in *Y*, thereby turning it into a ''$PAC_{A,Y}$''. (This vetting could potentially, depending on policy, involve such activities as discarding some privilege attributes disallowed in cell *Y*, or translating ''*X*-privileges'' into ''*Y*-privileges'', and so on.) This $PAC_{A,Y}$ is placed into $PTkt_{A,KDSY}$, becoming the authorisation information nominating *A* to servers in cell *Y*. Moreover, PS_Y ''consumes'' the transit path $P_{X,Y}$ (that is, vets it, and replaces it with the trivial transit path P_Y). PS_Y also returns to *A* a new session key $\tilde{K}_{A,KDSY}$ between *A* and KDS_Y, as usual.

- $A \rightarrow TGS_Y$: *B*, $L_{A,B}, \tilde{N}_{A,TGSY}, PTkt_{A,KDSY}, \{PS_Y, \tilde{C}_{A,TGSY}, [\tilde{\hat{K}}_{A,KDSY},] \tilde{T}_{A,TGSY}\} \tilde{K}_{A,KDSY}$
- $A \leftarrow TGS_Y$: $PS_Y, \{B, K_{A,B}, L_{A,B}, \tilde{N}_{A,TGSY}\} \tilde{K}^{[\hat{}]}_{A,KDSY}, PTkt_{A,B}$
- $A \rightarrow B$: $PTkt_{A,B}, \{PS_Y, [C_{A,B},] [\hat{K}_{A,B},] T_{A,B}\} K_{A,B}$
- $A \leftarrow B$: $\{[\tilde{\hat{K}}_{A,B},] T_{A,B}\} K_{A,B}$
- $A \rightarrow B$: {*Service Request*(s) / *Application-level Data*} $K^{[\hat{}[\hat{}]]}_{A,B}$

- $A \leftarrow B$: {*Service Request*(s) / *Application-level Data*} $K_{A,B}^{[\hat{}[\hat{}]]}$

  7; 8 (*Get privilege-ticket to server; Get service from server*.)

  Armed with a $PTkt_{A,KDSY}$ (containing $PAC_{A,Y}$) usable in *Y*, *A* can now proceed according to the usual Kerberos protocol to request services from servers (such as *B*) in cell *Y*: send the stringname of *B* and $PTkt_{A,KDSY}$ to KDS_Y requesting a privilege-ticket, $PTkt_{A,B}$, to *B*, and so on. Since this is exactly the same as the single-cell authentication protocol (which has already been explained), it is not repeated here. Note in particular that the privilege-ticket that KDS_Y receives from the foreign principal *A*, $PTkt_{A,KDSY}$, *looks exactly the same as* the privilege-tickets that KDS_Y receives from the local principals within its own cell (KDS_Y *cannot distinguish* between the two kinds of privilege-tickets). The same is true of the privilege-ticket that *B* receives from *A*:

  $$PTkt_{A,B} = B, \{PS_Y, K_{A,B}, L_{A,B}, PAC_{A,Y}, P_Y\} K_B$$

  That is to say, *B* cannot tell from this privilege-ticket's *format* that *A* is a foreign principal (it can only tell so by looking into *A*'s identity information carried in $PAC_{A,Y}$).

1.7.1 The Complete Cross-cell Scenario

The preceding outline envisions only the case where a *direct* trust link has been established between cells *X* and *Y*. The general case of *indirect* trust chains (that is, whose number of links is greater than 1, with intermediate cells intervening between *X* and *Y*) is an inductive generalisation of that one, whereby *A* engages in a succession of cross-cell referrals from one cell's KDS server to another's (intermediate PS servers are *not* visited), bringing it ''ever closer to'' (and eventually arriving at) *B*'s cell, at each stage engaging in the protocol outlined above. (The various cells' KDS servers only ''indirectly refer'' to one another, never ''directly chain'' to one another, in the sense that during the performance of their mainline services they only ever ''communicate'' cross-cell with one another via the intermediary of *A*, never communicating directly to one another — they only do so for such incidental purposes as parsing stringnames into their component pieces, or for cross-cell key management. See Section 1.13 on page 67 and Section 1.14 on page 69.)

The successive intermediate cross-cell referral tickets contain at each stage a record (transit path) of the trust chain of cells to that point, say:

$$P_{X,Z',Z'',\cdots,Z''',Z''''} = TCB_X \rightarrow TCB_{Z'} \rightarrow TCB_{Z''} \rightarrow \cdots \rightarrow TCB_{Z'''} \rightarrow TCB_{Z''''}$$

and so the corresponding cross-cell referral ticket may be denoted:

$$Tkt_{A,X,Z',Z'',\cdots,Z''',KDSZ'''Z''''} = KDS_{Z''',Z''''}, \{PS_X, K_{A,KDSZ'''Z''''}, L_{A,KDSZ'''Z''''}, PAC_{A,X}, P_{X,Z',Z'',\cdots,Z''',Z''''}\} K_{KDSZ'''Z''''}$$

When the target server's cell, *Y*, is eventually reached, the corresponding cross-cell referral ticket, $Tkt_{A,X,Z',Z'',\cdots,Z'''',KDSZ''''Y}$, is used to obtain a (service) ticket to the PS server in cell *Y*:

$$Tkt_{A,X,Z',Z'',\cdots,Z''',Z'''',Y,PSY} = PS_Y, \{PS_X, K_{A,PSY}, L_{A,PSY}, PAC_{A,X}, P_{X,Z',Z'',\cdots,Z''',Z'''',Y}\} K_{PSY}$$

This $Tkt_{A,X,Z',Z'',\cdots,Z''',Z'''',Y,PSY}$ is then presented to PS_Y, which vets the $PAC_{A,X}$ (turning it into a ''$PAC_{A,Y}$'') and the transit path $P_{X,Z',Z'',\cdots,Z''',Z'''',Y}$ (''consuming'' the latter), and returns a privilege-ticket that *A* can use in cell *Y*:

$$PTkt_{A,KDSY} = KDS_Y, \{PS_Y, \tilde{K}_{A,KDSY}, \tilde{L}_{A,KDSY}, PAC_{A,Y}, P_Y\} K_{KDSY}$$

Finally, this $PTkt_{A,KDSY}$ is presented to KDS_Y, which uses it to generate a (service-)privilege-ticket, $PTkt_{A,B}$:

$$PTkt_{A,B} = B, \{PS_Y, K_{A,B}, L_{A,B}, PAC_{A,Y}, P_Y\} K_B$$

In summary, the sequence of communications (RPCs) involved in a complete protected (authenticated/authorised) RPC from a client *A* in cell *X* to a server *B* in cell *Y*, transiting through cells $_{Z'}$, $_{Z''}$, ···, $_{Z'''}$, $_{Z''''}$ can be depicted as shown in Figure 1-6.

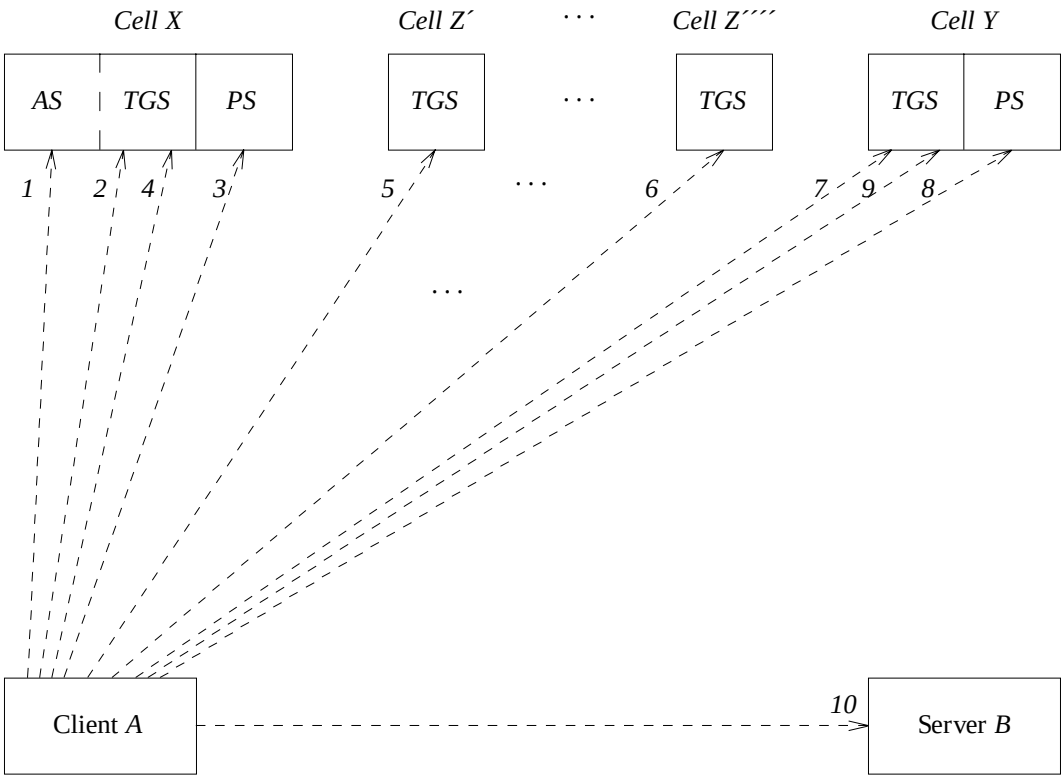


Figure 1-6 Cross-cell Protocol (Multi-hop)

Following are very brief descriptions of each step (for details see above, or the chapters to follow). In each step after the first one, *A* presents to its target the ticket (or privilege-ticket) it received from the previous step.

- 1 (*Get ticket to local KDS*.)

  A presents its identity to KDS_X, requesting AS_X for an initial ticket, $Tkt_{A,KDSX}$ to KDS_X, and receives it.

- 2 (*Get ticket to local PS*.)

  A presents $Tkt_{A,KDSX}$ to KDS_X, requesting TGS_X for a ticket, $Tkt_{A,PSX}$ (= $Tkt_{A,X,PSX}$), to PS_X, and receives it.

- 3 (*Get privilege-ticket to local KDS*.)

  A presents $Tkt_{A,PSX}$ to PS_X, requesting PS_X for a privilege-ticket, $PTkt_{A,KDSX}$, to KDS_X, and receives it. (Note that steps 1–3 typically happen at *A*'s ''login time'', while the remaining steps occur ''on demand'' when *A* desires service from an end-service *B*.)

- 4 (*Get ticket to foreign KDS*.)

  A presents $PTkt_{A,KDSX}$ to KDS_X, requesting TGS_X for a privilege-ticket to *B*, but instead receives a cross-cell referral ticket, $Tkt_{A,X,KDSXZ'}$, to $KDS_{X,Z'}$. (The cross-cell tickets issued in this sequence of steps are not yet privilege-tickets, even though they carry authorisation

information (a PAC) for the nominated client, *A*. This is explained in Section 1.7 on page 32.)

- 5 (*Get ticket to foreign KDS*.)

  A presents $\text{Tkt}_{A,X,\text{KDS}XZ'}$ to $\text{KDS}_{Z'}$, requesting $\text{TGS}_{Z'}$ for a ticket to PS_Y, but instead receives a cross-cell referral ticket, $\text{Tkt}_{A,X,\,Z',\,\text{KDS}Z'Z''}$, to $\text{KDS}_{Z',Z''}$.

- ··· (*Get tickets to foreign KDSs*.) ···

- 6 (*Get ticket to foreign KDS*.)

  A presents $\text{Tkt}_{A,X,Z',\,Z'',\cdots,\,Z''',\,\text{KDS}Z'''Z''''}$ to $\text{KDS}_{Z''''}$, requesting $\text{TGS}_{Z''''}$ for a ticket to PS_Y, but instead receives a cross-cell referral ticket, $\text{Tkt}_{A,X,Z',\,Z'',\cdots,\,Z''',\,Z'''',\,\text{KDS}Z''''Y}$, to $\text{KDS}_{Z'''',Y}$.

- 7 (*Get ticket to foreign PS*.)

  A presents $\text{Tkt}_{A,X,Z',\,Z'',\cdots,\,Z''',\,Z'''',\,\text{KDS}Z''''Y}$ to KDS_Y, requesting TGS_Y for a ticket, $\text{Tkt}_{A,X,Z',\,Z'',\cdots,\,Z''',\,Z'''',Y,\text{PS}Y}$, to PS_Y, and receives it. (Again, this $\text{Tkt}_{A,X,Z',\,Z'',\cdots,\,Z''',Z'''',\,Y,\text{PS}Y}$, is not yet a privilege-ticket.)

- 8 (*Get privilege-ticket to foreign KDS*.)

  A presents $\text{Tkt}_{A,X,Z',\,Z'',\cdots,\,Z''',\,Z'''',Y,\text{PS}Y}$, to PS_Y, requesting PS_Y for a privilege-ticket, $\text{PTkt}_{A,\text{KDS}Y}$, to KDS_Y, and receives it.

- 9 (*Get privilege-ticket to server*.)

  A presents $\text{PTkt}_{A,\text{KDS}Y}$ to KDS_Y, requesting TGS_Y for a privilege-ticket, $\text{PTkt}_{A,B}$, to *B*, and receives it.

- 10 (*Get service from server*.)

  A presents $\text{PTkt}_{A,B}$ to *B*, requesting *B* for service, and receives it (provided that *A*'s $\text{PAC}_{A,Y}$ grants it access per *B*'s access determination algorithm and the ACL protecting the accessed object).

Of course, the DCE RPC programming model hides all these complexities from the application programmer.

1.7.2 Multi-hop Trust Chains

In the descriptions given above, no solution has yet been given to the following closely related twin problems, which together are referred to as the *multi-hop trust chain problem*:

- How does each cell's KDS server ''know'' which foreign KDS servers to refer clients to?

  That is, how does each KDS_Z, which nominally knows only about the foreign KDSs it is *directly* cross-registered with, understand the overall global graph of *other* KDS_W's direct cross-registrations, so that the cross-cell referral ticket, $\text{Tkt}_{A,X,Z,\,\text{KDS}ZW}$, that $\text{KDS}_{Z'}$ returns to a client *A* actually does point *A* ''closer to'' its targeted end-server *B*, guaranteeing that any sequence of cross-cell referrals $\text{KDS}_X \rightarrow \text{KDS}_{Z'} \rightarrow \text{KDS}_{Z''} \rightarrow \cdots \rightarrow \text{KDS}_{Z'''} \rightarrow \text{KDS}_{Z''''} \rightarrow \cdots$ ultimately terminates (at KDS_Y)?

- How does each cell's PS server ''know'' whether or not to trust a transit path?

  That is, how does each PS_Y know whether or not the transit path $\text{P}_{X,Z',\,Z'',\cdots,\,Z''',\,Z'''',Y}$, presented to it for vetting, conforms to a ''trusted shape''?

This revision of this specification does not, in fact, specify a solution to this multi-hop trust chain problem. Thus, the only solution to the cross-cell trust chain problem that is fully specified in this revision of DCE is the single-hop (direct) case. Nevertheless, it is anticipated that a solution to the multi-hop trust chain problem will be specified in a future revision of DCE, and that the

specified solution will be the so-called *hierarchical* (or *up-over-down*) *trust chain* solution, which is depicted in Figure 1-7. In that figure, the arrowed lines indicate trust links, while the non-arrowed lines indicate namespace parent/child relationships (solid ones indicate direct links; dotted ones indicate a span of several generations; dashed ones indicate the step from a cell's TCB to a principal within that cell). Very briefly, the idea is that the *hierarchical structure of cell names* (indicated in Figure 1-7 by sequences of atomic names in angle brackets) is exploited to navigate an ''up-over-down'' path from client to server, seeking the ''least common trust peer link'' between the client's cell ancestry and the server's cell ancestry. Only a direct relationship between these cell ancestries is deemed acceptable (not a chain through intermediate ancestries, $_{Z}$´´, as indicated).

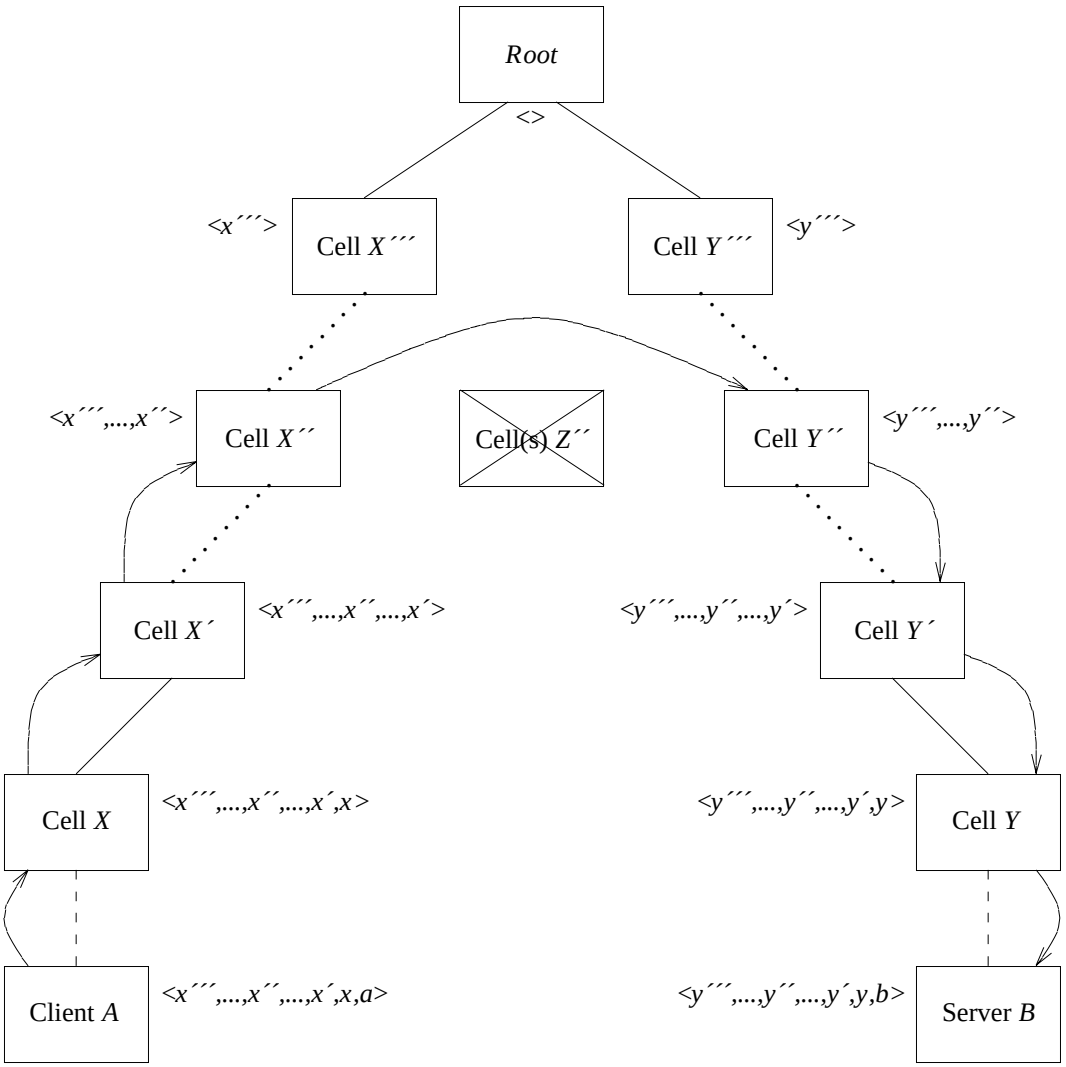


Figure 1-7 Hierarchical Trust Chains

1.8 Access Control Lists (ACLs)

Note: Much of the ACL Facility specified in DCE is ''modelled after'' the ACL Specification in POSIX P1003.6 Draft 12 (with allowable extensions). However, no claim of ''conformance to'' POSIX ACLs is made here, because of the preliminary (*draft*) nature of the POSIX ACL model. Harmonisation of the ACL Facility specified here with the *final* POSIX ACL Standard is for future study. (Note, for example, the disappearance and reappearance of MASK_OBJ in various POSIX drafts.)

The *Access Control List* (*ACL*) *Facility* manages server-side access control information, thereby enabling a server to control clients' access to the (*protected*) *objects* (that is, objects to which ACLs are attached) managed by the server. The ACL Facility comprises *ACLs* and *ACL Managers*, which are discussed in this section and Section 1.9 on page 46.

An ACL consists of:

- *ACL Manager Type UUID*

  UUID identifying the *semantics* of the ACL; that is, the kind or type of ACL Manager that can interpret the ACL (see below), especially with respect to the permissions granted or denied in the ACL Entries (ACLEs). By this means, protected objects are partitioned into classes, according to the types of ACL managers that can interpret their ACLs. An object protected by ACLs can have an arbitrary number of ACLs associated with it, but at most one of any given ACL Manager Type (see Section 1.9.2 on page 52).

- *Default Cell UUID*

  UUID identifying the cell to which the ACLEs of ''local'' (non-''foreign'') type apply (see below), and said to be the cell to which the ACL *refers*.

  Note: The ACL's default cell UUID does not necessarily identify the cell the protected object *itself* belongs to (that is, the cell in which the server protecting the object is registered as a principal), though this will typically be the case.

- *ACL Entries (ACLEs)*

  Specifies the access rights of subjects to the protected object, according to the access determination algorithm (see below).

1.8.1 ACL Entries and their Types

An ACLE consists of:

- *ACLE Type*

  This indicates how the ACLE is to be interpreted by the access determination algorithm (see below).

- *Tag UUID Field(s)*

  UUID(s) (0, 1 or 2 of them), of some combination of cells, principals and groups, used to ''qualify'' certain ACLE types, as explained below.

- *Permissions Field*

  Specifies access rights (up to 32 of them) afforded by this ACLE. The semantics of these access rights are interpretable only by ACL Managers of the ACL Manager type specified by the ACL (see below).

The *ACLE Types* are organised into the following taxonomy:

- *Local* ACLE types

  Refer to subjects in the ACL's *default cell*:

  — *USER_OBJ* or *UO* (no Tags)

  Establishes permissions for the object's *owning user* (in the POSIX sense), identified by the ordered pair <Default Cell UUID, *Owning Principal UUID*>, to which the USER_OBJ ACLE is said to *refer*. The Owning Principal UUID is a UUID determined in an application-specific way (not specified in DCE).

  Note: The USER_OBJ ACLE type is principally supported for purposes of POSIX compliance (especially, for POSIX-compliant filesystems). Its utility for most other applications is minimal — indeed, some authorities believe its use should in general be deprecated.

  — *USER* or *U* (with a Principal Tag UUID)

  Establishes permissions for the principal identified by the ordered pair <Default Cell UUID, Principal Tag UUID>, to which the USER ACLE is said to *refer*.

  — *GROUP_OBJ* or *GO* (no Tags)

  Establishes permissions for the object's *owning group* (in the POSIX sense), identified by the ordered pair <Default Cell UUID, *Owning Group UUID*>, to which the GROUP_OBJ ACLE is said to *refer*. The Owning Group UUID is a UUID determined in an application-specific way (not specified in DCE).

  Note: The GROUP_OBJ ACLE type is principally supported for purposes of POSIX compliance (especially, for POSIX-compliant filesystems). Its utility for most other applications is minimal — indeed, some authorities believe its use should in general be deprecated.

  — *GROUP* or *G* (with a Group Tag UUID)

  Establishes permissions for the group identified by the ordered pair <Default Cell UUID, Group Tag UUID>, to which the GROUP ACLE is said to *refer*.

  — *OTHER_OBJ* or *O* (no Tags)

  Establishes permissions for all principals and groups in the cell identified by the ACL's Default Cell UUID, to which the OTHER_OBJ ACLE is said to *refer*.

- *Foreign* ACLE types

  Refer to subjects in *foreign cells* (''foreign'' in the sense of ACLs has the commonsense meaning of referring to cells *other* than the ACL's default cell; that is, that the Cell Tag UUIDs of foreign ACLE types are distinct from the ACL's Default Cell UUID. However, it is not forbidden for an ACLE to contain a Cell Tag UUID which is equal to the ACL's Default Cell UUID — it is just inefficient to do so):

  — *FOREIGN_USER* or *FU* (with a Cell Tag UUID and a Principal Tag UUID)

  Establishes permissions for the principal identified by the ordered pair <Cell Tag UUID, Principal Tag UUID>, to which the FOREIGN_USER ACLE is said to *refer*.

  — *FOREIGN_GROUP* or *FG* (with a Cell Tag UUID and a Group Tag UUID)

  Establishes permissions for the group identified by the ordered pair <Cell Tag UUID, Group Tag UUID>, to which the FOREIGN_GROUP ACLE is said to *refer*.

— *FOREIGN_OTHER* or *FO* (with a Cell Tag UUID)

Establishes permissions for all principals and groups identified by the Cell Tag UUID, to which the FOREIGN_OTHER ACLE is said to *refer*.

- *Universal* ACLE type

Refers to subjects in the ACL's default cell or in foreign cells:

— *ANY_OTHER* or *AO* (no Tags)

Establishes permissions for all principals and groups (in the ACL's default or in foreign cells), to which the ANY_OTHER ACLE is said to *refer*.

- *Mask* ACLE types

Used as *masks* (subtracting rights at most, not adding them) in the access determination algorithm:

— *MASK_OBJ* or *M* (no Tags)

Masks all ACLEs except USER_OBJ and OTHER_OBJ.

Note: The MASK_OBJ ACLE type is principally supported for purposes of POSIX compliance (especially, for POSIX-compliant filesystems). Its utility for most other applications is minimal — indeed, some authorities believe its use should in general be deprecated.

— *UNAUTHENTICATED* or *UN* (no Tags)

Masks all ACLEs in an ''unauthenticated'' access request (that is, one whose PAC has a FALSE authentication flag).

Note: The UNAUTHENTICATED ACLE type is principally supported for purposes of ''system level'' applications (for example, ''bootstrapping'' the system). Its utility for most other applications is minimal — indeed, some authorities believe its use should in general be deprecated.

- *Extended* ACLE type

Used as a compatibility aid, and does not appear directly in the access determination algorithm:

— *EXTENDED* or *E* (no Tags)

A special type for ensuring extensibility/compatibility of the ACL mechanism. The intent is that if a new version of a server supports ACLE types that were not supported in earlier versions, then the new server is supposed to convert said new ACLE types to the EXTENDED type before passing ACLs back to old clients (via *rdacl_lookup*( ) — see Section 1.11 on page 55).

- Delegation *Local* ACLE types

Refer to subjects in the ACL's *default cell*:

— *USER_OBJ_DELEG* or *UOD* (no Tags)

Establishes permissions for the object's *owning user* (in the POSIX sense) *acting as a delegate*, identified by the ordered pair <Default Cell UUID, *Owning Principal UUID*>, to which the USER_OBJ_DELEG ACLE is said to *refer*. The Owning Principal UUID is a UUID determined in an application-specific way (not specified in DCE).

Note: The USER_OBJ_DELEG ACLE type is principally supported for purposes of POSIX compliance (especially, for POSIX-compliant filesystems). Its utility for most other applications is minimal — indeed, some authorities believe its use should in general be deprecated.

— *USER_DELEG* or *UD* (with a Principal Tag UUID)

Establishes permissions for the principal *acting as a delegate* identified by the ordered pair <Default Cell UUID, Principal Tag UUID>, to which the USER_DELEG ACLE is said to *refer*.

— *GROUP_OBJ_DELEG* or *GOD* (no Tags)

Establishes permissions for the object's *owning group* (in the POSIX sense) *acting as a delegate*, identified by the ordered pair <Default Cell UUID, *Owning Group UUID*>, to which the GROUP_OBJ_DELEG ACLE is said to *refer*. The Owning Group UUID is a UUID determined in an application-specific way (not specified in DCE).

Note: The GROUP_OBJ_DELEG ACLE type is principally supported for purposes of POSIX compliance (especially, for POSIX-compliant filesystems). Its utility for most other applications is minimal — indeed, some authorities believe its use should in general be deprecated.

— *GROUP_DELEG* or *GD* (with a Group Tag UUID)

Establishes permissions for the group *acting as a delegate* identified by the ordered pair <Default Cell UUID, Group Tag UUID>, to which the GROUP_DELEG ACLE is said to *refer*.

— *OTHER_OBJ_DELEG* or *OD* (no Tags)

Establishes permissions for all principals *acting as a delegate* in the cell identified by the ACL's Default Cell UUID, to which the OTHER_OBJ_DELEG ACLE is said to *refer*.

- Delegation *Foreign* ACLE types

Refer to subjects in *foreign cells acting as a delegate* (''foreign'' in the sense of ACLs has the commonsense meaning of referring to cells *other* than the ACL's default cell; that is, that the Cell Tag UUIDs of foreign ACLE types are distinct from the ACL's Default Cell UUID. However, it is not forbidden for an ACLE to contain a Cell Tag UUID which is equal to the ACL's Default Cell UUID — it is just inefficient to do so):

— *FOREIGN_USER_DELEG* or *FUD* (with a Global Principal Tag UUID)

Establishes permissions for the principal *acting as a delegate* identified by the <Global Principal Tag UUID>, to which the FOREIGN_USER_DELEG ACLE is said to *refer*.

— *FOREIGN_GROUP_DELEG* or *FGD* (with a Global Group Tag UUID)

Establishes permissions for the group *acting as a delegate* identified by the <Global Group Tag UUID>, to which the FOREIGN_GROUP_DELEG ACLE is said to *refer*.

— *FOREIGN_OTHER_DELEG* or *FOD* (with a Cell Tag UUID)

Establishes permissions for all principals *acting as a delegate* identified by the Cell Tag UUID, to which the FOREIGN_OTHER_DELEG ACLE is said to *refer*.

- Delegation *Universal* ACLE type

Refers to subjects in the ACL's default cell or in foreign cells:

— *ANY_OTHER_DELEG* or *AOD* (no Tags)

Establishes permissions for all principals *acting as a delegate* (in the ACL's default or in foreign cells), to which the ANY_OTHER_DELEG ACLE is said to *refer*.

There are a few conditions that an ACL must satisfy in order to be considered (*absolutely*) *well-formed*. For example, a well-formed ACL must not contain more than one USER ACLE that refers to a given principal. These ''formation rules'' are given in detail in Chapter 7.

1.8.2 Object Types, ACL Types and ACL Inheritance

Resource managers in general (and, by extension, reference monitors) distinguish between two types of objects: *container* objects and *simple* objects. (In general the unqualified word ''object'' refers to either container or simple objects, but when the context requires just one of these types, plain ''object'' means a simple object.) Container objects ''contain'' (in some application-defined sense, often reflected in a hierarchical name structure) other objects (simple or container). Simple objects do not contain other objects. The words *parent* and *child* are used to express the relationship between containers and the objects contained in them, respectively. Examples of container objects might include a filesystem directory or a database table; examples of simple objects might include a file or a database entry. Not all resource managers need support container objects.

To protect both simple and container objects, and to enable newly created objects to *automatically inherit default ACLs from their parent container objects* (a usability criterion), the ACL facility supports two *ACL types* (this is to be distinguished from the ACL's *manager type*, defined in Section 1.8 on page 40):

- *Protection* (or *Object*) *ACLs* are associated with either simple or container objects, and control access to them (that is, figure into the access determination algorithm exercised by ACL Managers (see below)).

  Note: By abuse of language, when one speaks of ''the'' ACLs associated with protected objects, it is always the Protection ACLs that are meant, unless explicitly indicated otherwise.

- *Initial* (or *Default Creation*) *ACLs* are associated with container objects only. Their function is not to control access to the container, but rather to supply default values for the ACLs inherited by child objects (both simple objects and containers) when they are initially created in the container. There are two kinds of Initial ACLs:

  — *Initial Object* (''*IO*'') *ACLs*

  Supplies default values for Protection ACL of simple child objects, and for the IO ACL of container child objects.

  — *Initial Container* (''*IC*'') *ACLs*

  Supplies default values for the Protection ACL and the IC ACL of container child objects.

Thus, in this inheritance model, simple objects have only one ACL associated with them (a Protection ACL), while container objects have three ACLs associated with them (Protection, IO and IC ACLs). And the creation of (simple and container) child objects obeys the following *ACL Inheritance Rules*:

- When a simple child object is created in a parent container, the child inherits, by default, the parent's IO ACL as the child's Protection ACL. (But if an ACL is specified, it overrides this default.)

- When a child container is created in a parent container, the child inherits, as defaults, the parent's IC ACL as the child's Protection ACL, and the parent's IO and IC as the child's IO and IC ACLs, respectively. (But if any of these ACLs are specified, they override the corresponding default.)

Other than the distinctions described above, there are no differences between the Protection ACL and Initial ACL types — therefore, the information about ACLs in the rest of this section does not differentiate between ACL types.

1.9 ACL Managers, Permissions, Access Determination Algorithms

ACL Managers are the modules within RPC servers that interpret (that is, lend semantics to) ACLs. Namely, every ACL Manager is identified within a server by a UUID called the ACL Manager's *type UUID*, and a given ACL Manager can interpret a given ACL if and only if the ACL Manager's type UUID is the same as the ACL's ACL Manager Type UUID (the latter is defined in Section 1.8 on page 40).

Note: Different ACL Managers — that is, those having different type UUIDs — can share the same executable code.

This notion of interpreting an ACL manifests itself in the following areas:

- *ACLs* and *ACLEs*

  An ACL Manager need not support all possible absolutely well-formed ACLs (where ''absolute'' refers to the general ACL Facility), but may define its own notion of *relatively well-formed* ACLs (where ''relative'' refers to the specific ACL Manager itself) — but the ACLs that it does support must be well defined. As an example, an ACL Manager need not support all the ACLE types supported by the ACL Facility itself (for example, an ACL Manager that never grants access to unauthenticated requests need not support the UNAUTHENTICATED ACLE). As another example, an ACL Manager might require that a certain ''minimal'' configuration of ACLEs is present in every ACL it supports (for example, a USER_OBJ/GROUP_OBJ/OTHER_OBJ triple, or at least one ACLE that grants Control (''Write-ACL'') permission (see below)).

  (Also, not all *ACL Types* (Protection, IO, IC) and their *Inheritance Rules* as specified in Section 1.8.2 on page 44 need be supported — but that is a property of the server as a whole, not a property of a specific ACL Manager.)

- *Permissions*

  The meanings of the permissions to protected objects vary according to the ACL Manager type associated with the ACL. Permissions themselves are considered to be ''primitives'', and access requests are determined (granted or denied) on the basis of *combinations* ((unordered) sets) of these primitives. In DCE, permissions are implemented as *bits* (represented as integers 2^i for $0 \le i \le 31$), and combinations of them as *bit-vectors* (represented as integers in the range $[0, 2^{32}–1]$). The number of permissions supported by an ACL Manager can be any number between 1 and 32 inclusive. (DCE does not support ACL Managers having 0 permissions; objects protected by more than 32 permissions can be supported by protecting them with multiple ACLs, each having a distinct ACL Manager Type — see Section 1.9.2 on page 52.)

- *Printstrings* and *Helpstrings*

  To each permission it supports, an ACL Manager associates, for human consumption, a printstring identifying the permission, as well as a helpstring that further explains the semantics of the permission. These strings are relayed to users by ACL Editors, as discussed above.

- *Access Determination Algorithm* (or *Authorisation Decision Computation*)

  The algorithm that takes as input a principal's PAC, an object's ACL, and an access request, and returns as output a judgement whether the server should *grant* or *deny* the access, is implemented by the ACL Manager.

In principle, there can be many distinct classes of ACL Managers, implementing the above areas in different application-specific ways (consistent with the security requirements of application servers). DCE currently defines (below) one distinguished class of ACL Managers, namely, the

so-called *Common ACL Managers*. However, *no* strict rules of the form ''DCE-conformant ACL Managers must be Common ACL Managers'' (or for that matter, ACL managers of any other type) are specified in DCE. Thus, the notion of Common ACL Manager may be considered merely to be a suggestive *example*. Furthermore, other classes of ACL Managers (other than Common ACL Managers) may be defined in future revisions of DCE. (On the other hand, applications that *can* support their security requirements by implementing Common ACL Managers may be well-advised to do so — it is a desirable ''user-friendliness'' issue for users to find such a level of consistency amongst all the ACL managers in all the applications throughout their environment.)

Common ACL Managers are defined as follows:

- *ACLs* and *ACLEs*

  Common ACL Managers support at least the USER, GROUP, OTHER_OBJ, FOREIGN_USER, FOREIGN_GROUP, FOREIGN_OTHER and ANY_OTHER ACLEs. (Thus, Common ACL Managers need not, but may, support the USER_OBJ, GROUP_OBJ, MASK_OBJ, UNAUTHENTICATED or EXTENDED ACLEs.)

  Concerning *ACL Types* and their *Inheritance Rules* (which are within the scope of the server as a whole, not of a specific ACL Manager): *if* a Common ACL Manager supports both simple objects and containers, then it supports all the ACL Types (Protection, IO, IC), with their Inheritance Rules as specified in Section 1.8.2 on page 44.

- *Permissions* and *printstrings*

  There are 7 permission bits that are distinguished by the ACL Facility, with respect to their (*C-language*) *names*, *values* (*bit representations*) and *printstrings* (see Section 8.1.1 on page 319 and Section 8.1.2.1 on page 320; for recommended *helpstrings*, see Section 8.1.2.2 on page 320). However, the generic ACL Facility itself does not specify the *access semantics* of those permissions — that is the responsibility of individual ACL managers themselves. In the case of Common ACL Managers, those semantics are described in the following list, wherein the 7 distinguished permissions are identified by their ''colloquial'' names (in English). (Thus, Common ACL Managers that support permissions having semantics as described in the following list use the bit representation and printstring associated with them (as specified in Section 8.1.1 on page 319 and Section 8.1.2.1 on page 320), and those bit representations and printstrings are not used for any other permission semantics; however, Common ACL Managers need not support all seven of these permission semantics, nor are they limited to supporting only these seven permission semantics.)

  — *Read*

    Disclose information associated with the protected object.

  — *Write*

    Modify information associated with the protected object.

  — *Execute*

    Cause a processing element to deliver service associated with the protected object.

  — *Control* (or *Change* or *Write-ACL*)

    Modify the ACL itself (as for protected object), thereby ''controlling'' access to the object protected by the ACL. Typically, ''owners'' (in some site- or application-specific sense) of protected objects are given control access to the object's ACL. (See *rdacl_replace*( ).)

    Note: DCE does not specify how Common ACL Managers manage *disclosure* of ACLs (*Read-ACL* access), so that aspect of Common ACL managers remains

implementation-specific. One possible solution is to support it with a permission, similar to the Control permission. Another solution (the most *typical* one, and the one *suggested* by DCE) is to grant Read-ACL access to any principal to which the ACL grants *any* access whatsoever (according to the Common Access Determination Algorithm described below). (This type of ''any access'' is not represented by a ''permission'' supported by the ACL Manager, so is an *exception* to the ''rule'' that Common ACL Managers mediate all access to protected objects (such as the ACL itself) via the Common Access Determination Algorithm (below).) The POSIX solution (which contemplates only filesystems, and only in a non-distributed environment) is to grant Read-ACL access to any principal that is granted ''POSIX search'' access to the hierarchy of (filesystem) directories containing the object (file) in question (but not necessarily the object itself). (See *rdacl_lookup*( ) and other operations in Section 1.11 on page 55.)

— *Insert*

Insert objects into the protected container object.

— *Delete*

Delete objects from the protected container object.

— *Test*

Compare a presented value to a value associated with the protected object, without disclosing its actual value. For example, compare a presented password with an actual password (though this isn't the way DCE supports password-based authentication).

- *Access Determination Algorithm*

  Common ACL managers support the Common Access Determination Algorithm specified in the next section.

Note: DCE does not currently specify an ACL Manager API (although it is envisioned that one will be supported in a future revision). Typically, implementations will support (implementation-dependent) ACL Manager APIs with routines that support the server's **rdacl** RPC interface (see Section 1.11 on page 55).

1.9.1 The Common Access Determination Algorithm for Delegation

As of DCE 1.1, Common ACL Managers support the *Common Access Determination Algorithm*, as described in this section (and specified carefully in Chapter 8). This is the algorithm that is executed by a Common ACL Manager upon an access request. The *output* of the algorithm consists of a judgement whether the server should ''grant'' or ''deny'' access to the protected object. Prior to DCE 1.1, Common ACL Managers needed only to ensure secure operations between two principals, typically described as a client and a server. As of DCE 1.1, clients and servers are able to invoke secure operations through one or more intermediaries. The *Common Access Determination Algorithm* has been modified as of DCE 1.1 in order to verify that the privilege attributes for each principal involved in the operation have the necessary rights in order to support delegation. This is because, for delegation, there is a delegation chain consisting of the intermediaries involved in servicing the requested operation. The order of intermediaries is not considered significant (for access determination — order is important in terms of *certification*, for instance. See Section 1.15.2 on page 77 for further discussion on this subject). The *input* to the algorithm consists of:

- An EPAC (presented from the client to the server, via a protected RPC). This EPAC contains the identity and group membership information present in a DCE 1.0 format PAC. This

EPAC also contains the privilege attributes of each participant (principal) in the chain.

Note: For compatibility with DCE 1.0, a PAC may be generated from an EPAC (by the Privilege Server, PS_Z, and be transmitted in the A_D field of a ticket (PTGT). Thus, existing *legacy* authorization models will continue to work.

- The principal involved in the operation, for each principal involved in the EPAC.

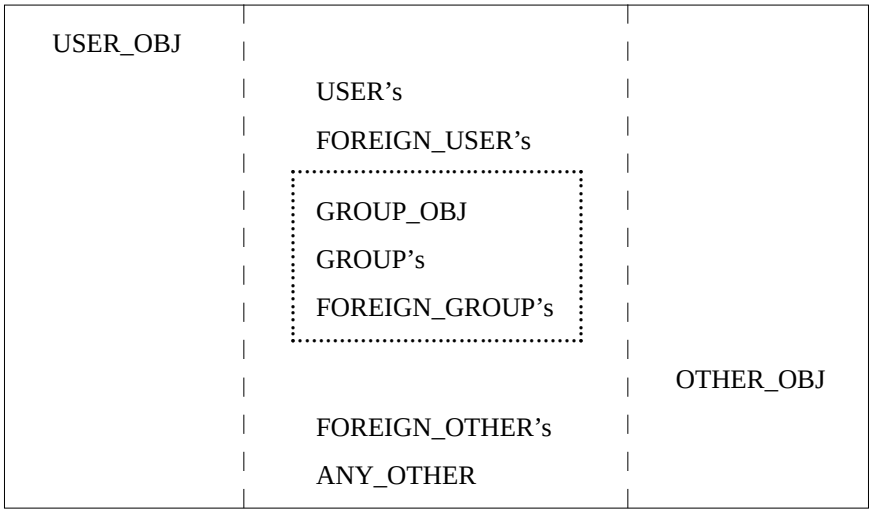
  Note: The order of intermediaries is not considered significant. Thus, the order of the checking of principals is not specified, and has no effect upon the standard access algorithm.

- Privilege Attribute Set for each principal involved in the operation. This Privilege Attribute Set is contained within the EPAC.
- An ACL, having the ACL manager type UUID of the invoked Common ACL Manager, and containing a list of ACLEs with types and tags as described above. (Note that a single ACL can represent at most 32 distinct permissions, because a single ACL can be associated with only a single ACL manager type UUID. The case of more than 32 permissions is dealt with in Section 1.9.2 on page 52.)
- An *access request* (of ‘‘required permissions’’), which is considered to be a *non-empty subset* of the permissions supported by the ACL Manager. (The case of an empty access request is not specified by DCE; it is implementation-specific.)

1.9.1.1 Common ACL Manager Algorithm

Given this input, the Common ACL Manager grants or denies access according to Figure 1-8 (of the ACLEs of the ACL in question), which affords a memorisable mental image of the common access determination algorithm.

Match PAC against ‘‘access ACLEs’’:



Mask acquired permissions against ‘‘mask ACLEs’’:

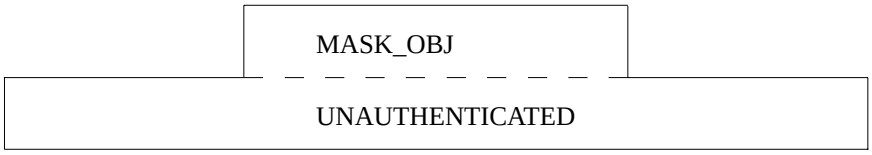


Figure 1-8 Common Access Determination Algorithm

In words, Figure 1-8 is to be interpreted as follows (this description is a loose *paraphrase* of the common access determination algorithm which is specified in detailed pseudocode in Chapter 8):

- *Match* (in the sense defined in the pseudocode in Chapter 8) the incoming PAC against the ACL’s *access ACLEs* (in the top-to-bottom order shown, namely: UO, U, FU, GO/G/FG, O, FO, AO), stopping at the *first* such match (except that *all* matches are considered ‘‘simultaneously’’ in the case of the indicated *group-like* ACLEs), and note the permissions granted by the matched ACLE (or, in the case of the group-like ACLEs, the *union* of the permissions granted by all the matched ACLEs).
- *Mask* (that is, *intersect*) the acquired permissions against the permissions in the ACL’s *mask ACLEs*, as necessary (namely, mask with MASK_OBJ permissions if the match occurred in the center column, and/or mask with UNAUTHENTICATED permissions if the PAC is unauthenticated). (If the ACL Manager doesn’t support these two mask ACLEs, this step is a null operation.)

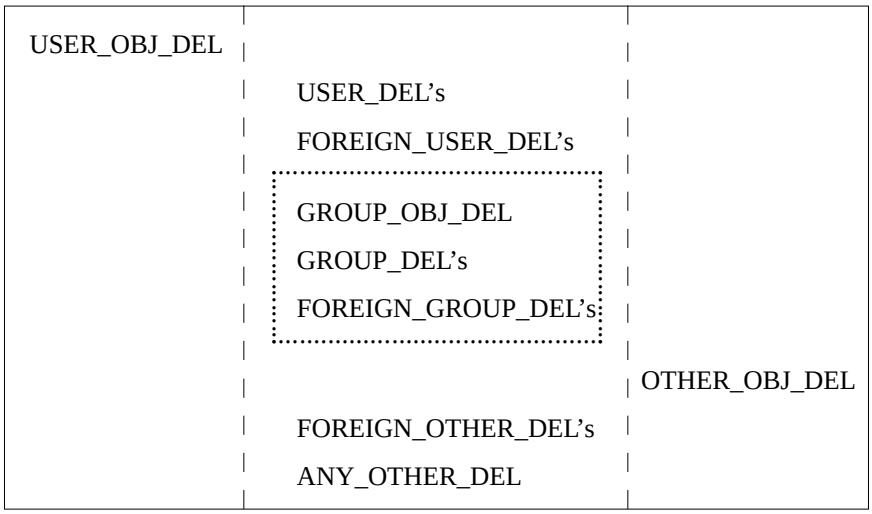
Note: While this mental image shows incoming PACs, it is applicable to EPACs as well — in particular, for the initiator of a request. Intermediaries use the algorithm whose mental image is given in Section 1.9.1.2 on page 51.

1.9.1.2 Delegation Common ACL Manager Algorithm

Given the input specified in Section 1.9.1 on page 48, the Common ACL Manager grants or denies access according to Figure 1-9 (of the ACLEs of the ACL in question), which affords a memorisable mental image of the delegation common access determination algorithm. This algorithm for delegation operates upon a set of extended entries in the ACL that apply only to principals acting as intermediaries. These extended entries permit intermediaries to be listed on the ACL without granting those intermediaries the ability to operate on the target object directly. Without these extensions, an intermediary would be otherwise be granted the ability to perform the operation requested of their own initiative.

Note: In this figure, the standard ACL entries are extended with entries identifying the abilities of the intermediary. The entries are traditionally extended with the **_DELEGATE** specification. However, in this mental image, they have been shortened to **_DEL**.

Match EPAC against ‘‘access ACLEs’’:



Mask acquired permissions against ‘‘mask ACLEs’’:

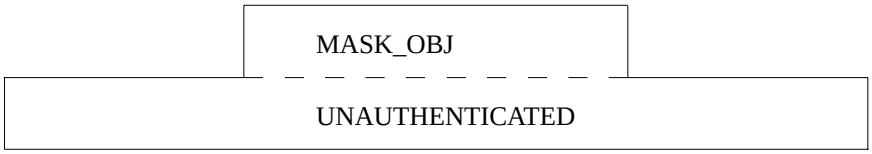


Figure 1-9 Delegation Common Access Determination Algorithm

In words, Figure 1-9 is to be interpreted as follows (this description is a loose *paraphrase* of the common access determination algorithm which is specified in detailed pseudocode in Chapter 8):

- *Match* (in the sense defined in the pseudocode in Chapter 8) the incoming EPAC against the ACL’s *access ACLEs* (in the top-to-bottom order shown, namely: UOD, UD, FUD, GOD / GD / FGD, OD, FOD, AOD), stopping at the *first* such match (except that *all* matches are considered ‘‘simultaneously’’ in the case of the indicated *group-like* ACLEs), and note the permissions granted by the matched ACLE (or, in the case of the group-like ACLEs, the *union* of the permissions granted by all the matched ACLEs).

- *Mask* (that is, *intersect*) the acquired permissions against the permissions in the ACL's *mask ACLEs*, as necessary (namely, mask with MASK_OBJ permissions if the match occurred in the center column, and/or mask with UNAUTHENTICATED permissions if the EPAC is unauthenticated). (If the ACL Manager doesn't support these two mask ACLEs, this step is a null operation.)

1.9.1.3 Notes on Common ACL Manager ACLs

Notes:

1. In the case of Common ACL Managers that support only ''relatively well-formed'' ACLs (such as, for example, ACL Managers that do not support MASK_OBJ), some simplifications of the two figures above and (or) the pseudocode description in Chapter 8 may be possible (for example, by omitting some ACLEs from the diagram, or rearranging some clauses of the algorithm). But such simplifications could lead to confusion, and should be avoided, as the diagram and pseudocode as presented are fully general enough to encompass all Common ACL Managers.
2. The access model supported by DCE is ''object-based'', as opposed to ''name-based'', in the sense that it depends only on the ACL of the target object, not those of intermediate naming nodes used to specify the object. This is in contradistinction to certain other systems, notably POSIX, whose access semantics support a notion of ''pathname resolution'', whereby a ''search'' (''traverse'') permission is required of intermediate naming nodes in addition to the access permissions of the ultimate target (leaf) object.

1.9.2 Multiple ACLs and ACL Managers

It is a typical scenario for a server to support only a single (protection) ACL per protected object, and a single ACL Manager to interpret the ACLs on all protected objects. But it is entirely conceivable that a server may support more than one ACL and/or ACL Manager. The following are some of the scenarios where this is useful or necessary:

- *Multiple types of protected objects*

  If a server supports multiple types of protected objects, it should (or rather, ''must'', since this is virtually the *definition* of ''type of protected object'') support a different ACL Manager type for each type of protected object. An example of this is given by the RS server itself (which supports five types of protected objects) — see Section 1.12.1 on page 61.

- *More than 32 permissions*

  If a server supports more than 32 permissions, then since each ACL and each ACL Manager can support at most 32 permissions, the server must support multiple ACLs and ACL managers. Conceptually, all such ACL Managers would support exactly the same access determination algorithm (perhaps even sharing the same code), but the interpretation of the permission bits would differ among the ACL Managers. Colloquially speaking, ''bit *i* (that is, 2^i, for $0 \le i \le 31$) means something different for ACL Manager Type $UUID_1$ than it does for ACL Manager Type $UUID_2$.'' For example, a given server might protect its objects with ACLs having a (hypothetical) ''Search'' permission mapped to bit 27 of ACL Manager Type $UUID_1$ and an ''Append'' permission mapped to bit 27 of ACL Manager Type $UUID_2$. An access request that required only, say, Search permission would then have to query only one ACL, while a request requiring both Search and Append permissions would have to query two ACLs (exactly the same ACL Manager code could potentially be used for both checks, but that's an implementation choice). At the extreme, a server could even support only a single

permission bit per ACL Manager (leaving the other 31 bits unused), thereby effecting a one-to-one mapping between its protected object's permission bits and UUIDs (the latter coming from the ACL Manager's Type UUID).

- *Exotic combinations*

  Some servers may need to query multiple quite different ACLs, some application-specific combination, to determine access; the potential combinations are unbounded in number. A simple example is afforded by the RS server itself, whose deletion operation (*rs_pgo_delete*( ) — see Section 11.5.4 on page 384) checks both the ACL of the object to be deleted as well as the ACL of the parent container of that object.

Variations on the ACL and ACL Manager architecture along these lines will be taken for granted in the remainder of this specification, and will not be mentioned explicitly again.

1.10 Protected RPC

As discussed above, the KDS and PS identify subjects to one another (including communicating privilege information), and perform session key management. Once a client and server share a session (or conversation) key, they can communicate ''application-level'' data securely. The *protection level* that an application may set using RPC determines the level of security of network messages.

Note: Part of the rationale for supporting a range of protection options is that, as a rule, higher security has an inversely proportional relationship to performance (because of the cost of code paths that go through security code, especially encryption/decryption and cryptographic checksum checking). The RPC facility provides several levels of protection so that applications can control this tradeoff between security and performance.

Following are the supported protection levels, arranged in order of ''increasing'' protection. The exact interpretation of these descriptions is dependent on underlying RPC and transport protocols (as specified in the referenced X/Open DCE RPC Specification), and is given in Chapter 9.

- **rpc_c_protect_level_none**

  No security guarantees. This amounts to the client and server receiving mere ''assertions'' (of low trustworthiness) about security attributes, as opposed to more convincing evidence of their trustworthiness (namely, TCB cryptographic protection). ''Protected RPC'' of this protection level is also called *unprotected RPC*.

- **rpc_c_protect_level_connect**

  Mutual authentication established when client and server initially establish a session.

- **rpc_c_protect_level_call**

  Mutual authentication established at the initiation of every RPC call between client and server.

- **rpc_c_protect_level_pkt**

  Mutual authentication established for every *packet* (unit of communication, as interpreted by underlying RPC and transport protocols) between client and server.

- **rpc_c_protect_level_pkt_integ**

  rpc_c_protect_level_pkt protection plus integrity protection of every packet (in the **rpc_c_authn_dce_secret** regime, this integrity protection is achieved with DES and MD5).

- **rpc_c_protect_level_pkt_privacy**

  rpc_c_protect_level_pkt_integ protection plus confidentiality protection of every packet (in the **rpc_c_authn_dce_secret** regime, this confidentiality protection is achieved with DES).

1.11 ACL Editors

Clients that manipulate a server's ACLs *as data* (as distinguished from clients that only use ACLs as *metadata*; that is, clients that only access the *underlying objects* that the ACLs actually protect), are called *ACL Editors*. The API that ACL Editors call is the **sec_acl** API, and the RPC interface that RPC servers export to support the **sec_acl** API is the **rdacl** RPC interface.

For the purposes of the **rdacl** interface, ACLs are identified by a combination of 4 items:

- The identity of the protected object itself. This is identified by:
  1. the identity of the server managing the protected object, and
  2. a server-supported stringname that ''further identifies'' the protected object within the server.
- The identity of the specific ACL associated with the protected object. This is given by:
  3. its ACL manager UUID, and
  4. its ACL type.

The server-supported stringname, in particular, enables the possibility of *extending the naming model*, so that protected objects ''appear as named objects in the (combined CDS-supported and server-supported) namespace'' — and this is how the **sec_acl** API views protect objects. (Concerning CDS, see the referenced X/Open DCE Directory Services Specification.) This is especially useful for servers that support large numbers of protected objects (though its use is ''optional,'' in the sense that servers can register each protected object solely via a server name in the CDS and an empty server-supported stringname, if they so wish). In the extended naming model, a server registers its own RPC binding name and an RPC object UUID in the usual way (with the CDS and with the local Endpoint Map) — the RPC object UUID (only one such should be registered in a given CDS namespace server entry) represents the ''root'' of the (server-supported) namespace (which may be hierarchical or not, and may have a different syntax from the CDS namespace). An ACL Editor can then address protected objects (and their ACLs) by uttering a *pair* of names: the server's CDS-registered name and the server-supported name of the protected object. It is even possible to view this pair of names as a *single* (syntactically concatenated) name, provided that the naming service in which the server is registered supports a *resolution-with-residual* support operation (that is, the namespace server resolves the concatenated name up to the server's registration point, returning to the client the *resolved* (parsed) and *residual* (unparsed) parts of the name). This enables the client to query the namespace at the resolved name for the server's registered binding information, and then to bind to it and present to it the server-supported part of the name. (The CDS name servers do support such a resolution-with-residual operation — see *rpc_ns_entry_inq_resolution*( ); potentially, the two steps of resolution-with-residual and binding-query could be combined into a single operation.) Note that this describes a general mechanism, called the *namespace junction* (or *federated naming*) *model*, whose utility is not limited to ACL editing. This is illustrated in Figure 1-10 (where hierarchical namespaces are illustrated using abstract ordered tuples of nodes, but for simplicity their concrete syntaxes are not shown). (For an example of junctions, see Section 1.12 on page 60, which discusses the RS's own use of this naming model.)

The **rdacl** RPC interface consists of the following operations:

- *rdacl_get_manager_types*( )

  Obtain a list of ACL manager types protecting an object.

- *rdacl_get_mgr_types_semantics*( )

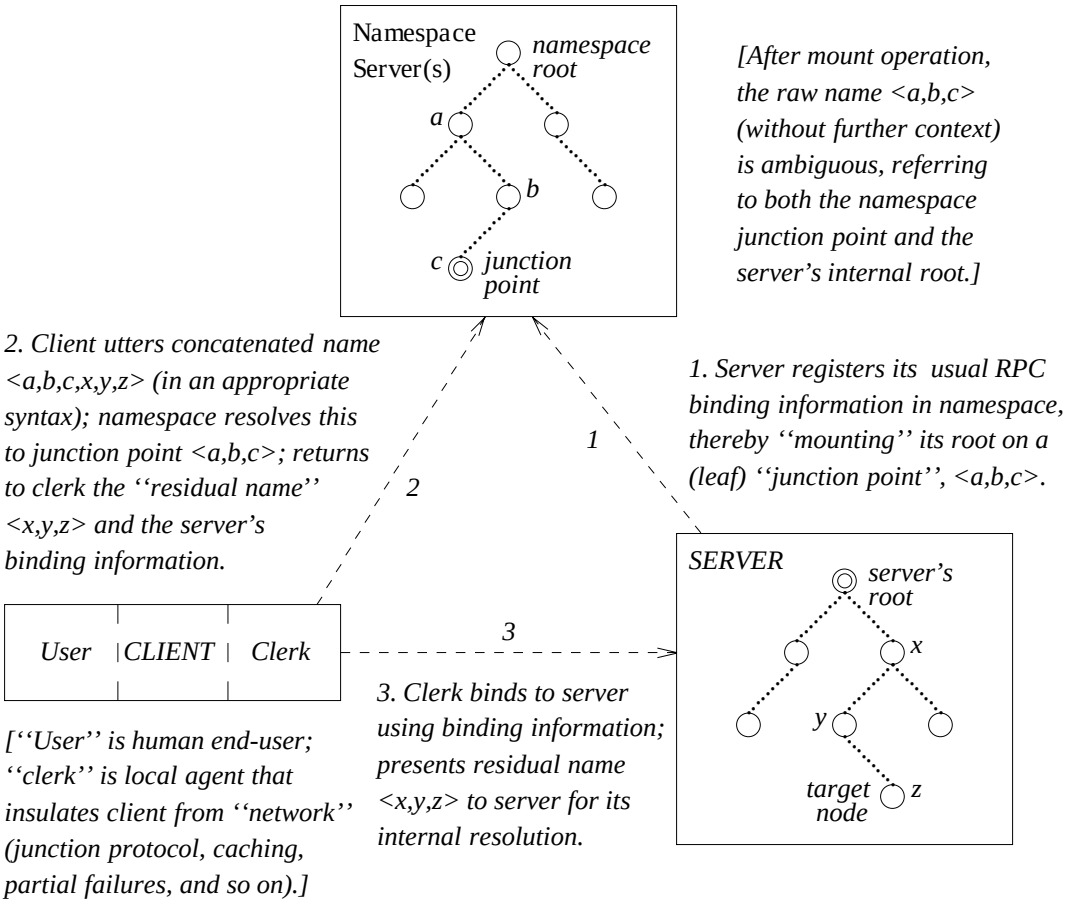


Figure 1-10 Namespace Junction (Federated Naming) Model

Obtain a list of ACL manager types protecting an object, together with information about the semantics they support.

- *rdacl_get_printstring*( )

  Obtain human-readable representations of permissions supported by an ACL manager.

- *rdacl_lookup*( )

  Retrieve (that is, ''read'') ACLs on a protected object, creating a copy locally on the client. ACL modification is done by manipulating the local copy, then applying this to the remote protected object with *rdacl_replace*( ).

- *rdacl_replace*( )

  Apply (that is, ''write'') ACLs to a protected object. This replaces the currently existing ACL on the protected object with a new one.

  Note: The ''currently existing ACL'' that gets replaced by an invocation of *rdacl_replace*( ) might not be the same as the ''old ACL'' that was previously retrieved by *rdacl_lookup*( ), because of a race condition: an intervening *rdacl_replace*( ) from another client may have already replaced that old ACL on the protected object by a different ACL (that is, no locking/transactional semantics are supported to prevent this from happening). In fact, an *rdacl_replace*( ) may

even fail if an intervening *rdacl_replace*( ) installs an ACL that denies the necessary control permission. This potential for interleaved *rdacl_lookup*( )/*rdacl_replace*( )s may be a minor inconvenience for some applications, but it is not typically a major inconvenience because it is rare for multiple clients to be managing the same ACL at the same time. In any case, this race condition does not compromise security (because every client's authority to manage ACLs is always checked, via the ACL's control permission), nor does it compromise consistency (because *rdacl_replace*( ) is *atomic*, writing whole ACLs at a time).

- *rdacl_get_access*( )

  Obtain calling principal's permissions to a protected object.

- *rdacl_test_access*( )

  Determine whether calling principal has the specified permission to access a protected object.

- *rdacl_test_access_on_behalf*( )

  Determine whether a specified principal (not necessarily the calling principal) has the specified permission to access a protected object. This can be used to support a primitive form of *delegation*.

- *rdacl_get_referral*( )

  Obtain a referral to an ACL ''update site'' (that is, a server instance or replica that manages a writable copy of the ACL, as opposed to a read-only copy). This operation can be used to support server replication (of a simple type).

The **sec_acl** API consists of the following routines:

- *sec_acl_bind*( )

  Obtain a ''protected object handle'' (that is, a handle referring to a protected object, represented by the **sec_acl_handle_t** data type), identifying the protected object by full name (that is, by CDS namespace entry concatenated with a server-supported namespace name). ACLs themselves are identified by a combination of this handle, the manager type UUID of the ACL, and the type of the ACL (protection, object creation or container creation).

- *sec_acl_bind_to_addr*( )

  Identical to *sec_acl_bind*( ), except that the protected object in question is identified by a combination of the address of the server and a server-supported namespace name.

- *sec_acl_release_handle*( )

  Release protected object handle previously created by *sec_acl_bind*( ) or *sec_acl_bind_to_addr*( ).

- *sec_acl_get_manager_types*( )

  Obtain a list of ACL manager types (UUIDs) protecting an object. (Layered over *rdacl_get_manager_types*( ).)

- *sec_acl_get_mgr_types_semantics*( )

  Obtain a list of ACL manager types protecting an object, together with information about the semantics they support. (Layered over *rdacl_get_mgr_types_semantics*( ).)

- *sec_acl_get_printstring*( )

  Obtain human-readable representations of permissions supported by an ACL manager. (Layered over *rdacl_get_printstring*( ).)

- *sec_acl_lookup*( )

  Retrieve (''read'') ACLs on a protected object. (Layered over *rdacl_lookup*( ).)

- *sec_acl_replace*( )

  Apply (''write'') ACLs to a protected object. (Layered over *rdacl_replace*( ).)

- *sec_acl_release*( )

  Free (local copy of) ACLs previously created by *sec_acl_lookup*( ).

- *sec_acl_get_access*( )

  Obtain calling principal's permissions to a protected object. (Layered over *rdacl_get_access*( ).)

- *sec_acl_test_access*( )

  Determine whether calling principal has the specified permission to access a protected object. (Layered over *rdacl_test_access*( ).)

- *sec_acl_test_access_on_behalf*( )

  Determine whether another principal has the specified permission to access a protected object. (Layered over *rdacl_test_access_on_behalf*( ).)

- *sec_acl_get_error_info*( )

  Obtain DCE runtime support routine error information related to the **sec_acl** API.

- *sec_acl_calc_mask*( )

  Calculate a new MASK_OBJ ACLE for an ACL (or list of ACLs), whose permissions are the union of the permissions of all ACLEs of types USER, FOREIGN_USER, GROUP_OBJ, GROUP, FOREIGN_GROUP, FOREIGN_OTHER, ANY_OTHER.

  Notes:

  1. The *sec_acl_calc_mask*( ) routine is supported for POSIX-compliant applications; that is, those that support MASK_OBJ with its POSIX semantics. The set of ACLEs listed, of which the newly calculated MASK_OBJ is the union, corresponds approximately to what POSIX calls the ''File Group Class ACLEs'', though that designation is inappropriate in the context of DCE.

  2. The **sec_acl** API is designed to be a general programming interface for managing all ACLs in such a way that the client is unaware of the principal identity of the server that controls the objects protected by the ACLs. As such, the server's principal name does not occur as a parameter to the **sec_acl** API (see, for example, *sec_acl_bind*( )). This implies, in particular, that the **sec_acl** API supports only *one-way* (client-to-server) authentication, not mutual (server-to-client) authentication. Applications that require mutual authentication should use the ''raw'' **rdacl** RPC protocol, not the **sec_acl** API. (Mutual authentication may be added to the **sec_acl** API in a future revision of DCE.)

  3. No local ''high-level ACL manipulation'' API routines are currently supported (for example, ''add or delete such-and-such an ACLE from this ACL'', or ''deny all privileges for such-and-such a principal''). Such operations are typically supported by interactive command-level ACL editing user interfaces, but those are beyond the scope of this revision of

DCE.

1.12 Registration Service (RS) and RS Editors

The *Registration* (or *Registry*) *Service* (RS) is the TCB's repository for security-relevant data (in particular, identities, long-term cryptographic keys and privilege information) in DCE. The RS maintains a datastore, whose clients are the KDS and PS, and RPC interfaces for the RS Editor, ID Map and Key Management facilities (see Figure 1-1 on page 12). The stored elements of the RS datastore are here called *items* — not *objects*, to distinguish terminologically between ''the entities themselves'' (''objects'') and the datastore information (''items'' with ''attributes'' attached to them) stored about the objects. Certain collections of items are called *domains*. The items in the RS datastore are organised into the following five categories:

- (*RS) Policy Item*

  Contains *cell-wide* information applicable to *all* the entities in the cell (not just, for example, principals in a particular organisation or group within the cell — not to be confused with ''organisation policies'', below). The Policy item has two categories of attributes, respectively called (*Registry*) *policies* and (*Registry) properties*.

- *Principal* (*P*) *Domain*

  Contains *principal items*; that is, information about principals (or ''principal objects''). Each principal is normally associated with an *account* (see below).

- *Group* (*G*) *Domain*

  Contains *group items*; that is, information about groups (or ''group objects''). Each group is associated with an (unordered) set of principals (not, however, of other groups), called the *members* of the group.

- *Organisation* (*O*) *Domain*

  Contains *organisation items*; that is, information about organisations (or ''organisation objects''). Each organisation is associated with an (unordered) set of principals (not, however, of groups or other organisations), called the *members* of the organisation. Organisations have as attributes *organisation policies*.

- *Account Domain*

  Contains *account items*; that is, information about accounts (or ''account objects''). Accounts identify the targets of DCE login rituals (see the Login Facility, below). They consist of a triplet, consisting of principal, group, and organisation items.

Principals (and other entities, such as groups and organisations) whose security data is held in a cell's RS are said to *belong to* that cell (in the security sense at least, and usually in the administrative sense also); similarly, the protected objects that server principals act as reference monitors for are said to *belong to* the server principal's cell. The RS itself has the principal name *dce-rgy* (within its cell).

Programs that manipulate (for administrative purposes) the RS datastore are called *RS Editors* (or *Rgy Editors*). To support RS Editors, the RS supports the **rs_policy, rs_pgo** and **rs_acct** RPC interfaces, described below. (The RS also supports *additional* RPC interfaces for other uses, and these are introduced in context in other sections.)

Note: In the current revision of DCE, the only RS Editor APIs that are supported are those related to binding to RS servers — that is, the RS Editor RPC interfaces supported by DCE are not currently supported by corresponding APIs. It is anticipated that such APIs will be added in a future revision.

1.12.1 ACL Manager Types Supported by the RS

The RS is the only service in the DCE TCB that supports a datastore of persistent items, and is the only one (not counting DTS) that protects its objects with ACLs (the KDS and PS protect the objects they manage by direct cryptographic means, not with ACLs). This section briefly discusses RS ACL management (an extended discussion occurs in Section 11.1 on page 358).

The RS acts as the reference monitor to five kinds of protected objects (items in its datastore), supported by five Common ACL Manager types. Each of the RS's protected objects is of exactly one of the following kinds (that is, no item can (currently) be ''polymorphic'', in the sense of being of more than one of the following kinds):

- *Policy ACL Manager Type*

  Manages ACLs on the RS's Policy item.

- *Directory ACL Manager Type*

  Manages ACLs on directories. These directories are containers internal to the RS itself, not one of the items managed by the RS datastore.

- *Principal ACL Manager Type*

  Manages ACLs on principal items.

- *Group ACL Manager Type*

  Manages ACLs on group items.

- *Organisation ACL Manager Type*

  Manages ACLs on organisation items.

The *ACL Manager Type UUIDs*, the *supported permissions* and *ACLE Types* of these ACL Manager types, and the *printstrings*, *bit representations* and *semantics* associated with their supported permissions, are all specified in Chapter 11.

1.12.2 RS Binding; rs_bind Interface and sec_rgy_bind API

This section discusses the RS binding and replication model, and its corresponding **rs_bind** RPC interface and **sec_rgy_bind** API. These are used to manage RPC bindings (or ''contexts'') between clients and (potentially replicated) RS servers, thereby enabling communications sessions between them.

A basic concept supported by DCE is that of *replication* of the RS service. For the purposes of this replication model, the terms *server*, *instance*, *replica* and *site* are considered to be *synonymous*. Specifically, the RS replication model supports a model of replication of the RS datastore consisting of two kinds of replicas: (one or more) *writable* (*update*) servers, and (zero or more) *readable* (*query*) servers, where every update server is a query server as well. This model is satisfied by, for example, any of the following:

1. an unreplicated RS server
2. replicated ''master/slave'' RS servers (one update replica and multiple query replicas)
3. ''mirrored'' RS servers (multiple RS servers which support all RS services and whose datastores are maintained synchronously).

Note: The RPC-level protocols to actually support this replication model are not specified in this revision of DCE; it is anticipated that they will be supported in a future revision.

All RS replicas (query or update) export the **rs_bind** RPC interface, which supports the following operation:

- *rs_bind_get_update_site*( )

  Get binding to update server (in the same cell).

This *rs_bind_get_update_site*( ) operation supports the RS Binding and replication models discussed above. Namely, clients can target query operations to arbitrary replicas; and they can target update operations to update replicas whose binding is determined by an invocation of *rs_bind_get_update_site*( ) (the latter can be targeted to an arbitrary (query or update) replica).

At RPC level, a context with an RS site is represented by an RPC binding handle, of data type **handle_t**. At API level (in the **sec_rgy_bind** API, and other RS APIs to be supported in future revisions of DCE), a context is represented by an *RS handle*, of data type **sec_rgy_handle_t**. The **sec_rgy_bind** API supports the following routines:

- *sec_rgy_cell_bind*( )

  Bind to (that is, establish a context with) some (unspecified) RS site in a specified cell.

- *sec_rgy_site_bind*( )

  Bind to an RS site.

- *sec_rgy_site_bind_update*( )

  Bind to an RS update site.

- *sec_rgy_site_open*( )

  Bind to an RS site, with default security characteristics.

- *sec_rgy_site_open_update*( )

  Bind to an RS update site, with default security characteristics.

- *sec_rgy_site_close*( )

  Free an RS handle, and unbind RS site.

- *sec_rgy_site_is_readonly*( )

  Determine whether an RS site is a query site or an update site.

- *sec_rgy_site_binding_get_info*( )

  Retrieve information about RS site.

1.12.3 Policy Item, Policies and Properties; rs_policy RPC Interface

The (*RS*) *Policy Item* holds *Policy* and *Property* information that affects all accounts in a cell. (This is not to be confused with *Organisation Policies*, which affect only the accounts of principals that are members of a particular organisation.)

- (*Registry*) *Properties* include such information as:

  — The version number of the RS software used to create and read the RS datastore.

  — The name and UUID of the cell associated with the RS, and whether an RS site is valid for update (‘‘writable’’) or only for query (‘‘read-only’’).

  — Minimum and default lifetimes for tickets issued to principals.

— Bounds on the local ID numbers used for principals, and whether the UUIDs of principals also contain embedded local ID numbers.

- (*Registry*) *Policies* and *Organisation Policies* control the accounts of principals that are members of the cell or of an organisation within the cell. This data controls the lifetime and length of passwords, as well as the character set from which passwords may be composed. It also controls the default lifespan of accounts. The policy in effect for any principal or organisation (said to be its *effective policy*) is the most restrictive combination of the cell policy and the policy for the principal or organisation.
- (*Registry*) *Authentication Policies* and *Account Authentication Policies* control the maximum lifetime of tickets, both upon first issue and upon renewal. The policy in effect for any account (said to be its *effective policy*) is the most restrictive combination of the cell policy and the policy for the account.

The RS supports the **rs_policy** RPC interface for operating on property and policy information, which supports the following operations:

- *rs_properties_get_info*( )

  Get cell property information.

- *rs_properties_set_info*( )

  Set cell property information.

- *rs_policy_get_info*( )

  Get cell or organisation policy information.

- *rs_policy_get_effective*( )

  Get cell or organisation effective policy information.

- *rs_policy_set_info*( )

  Set cell or organisation policy information.

- *rs_auth_policy_get_info*( )

  Get cell or account authentication policy information.

- *rs_auth_policy_get_effective*( )

  Get account's effective authentication policy.

- *rs_auth_policy_set_info*( )

  Set cell or account authentication policy information.

1.12.4 PGO Items; rs_pgo RPC Interface

The Principal, Group and Organisation items in the RS datastore are known collectively as *PGO items*. PGO items primarily contain:

- A ''human-friendly'' (*string*)*name*, which serves as the normal *datastore query* (*lookup*) *key* (not to be confused with cryptographic keys) to PGO items (within a cell). There are separate namespaces for Principals, for Groups and for Organisations, and each is organised hierarchically. (For more on *naming*, see Section 1.13 on page 67 and Section 1.18 on page 84.)
- A ''computer-friendly'' (because of its small fixed size and its ability to be automatically generated) UUID which serves as the *definitive* identifier of the PGO item (within the context of its cell). (''Definitive'' here means that each principal has a *unique* UUID, though it may

have multiple (alias) stringnames.)

The Principal domain (though not the Group or Organisation domains) supports *aliases*; that is, multiple names pointing to the same datastore entry.

The following principal names are *reserved* (within each cell).

Note: The corresponding principal UUIDs are *not* reserved, and in fact they're expected to be different in each cell — see Section 1.6 on page 25 concerning DCE's ''double-UUID identification scheme''.

- **krb5tgt/***cell-name*

  KDS (or surrogate) in (local or foreign) cell **/.../***cell-name*. Thus, for example, if the cell name of cell *X* is **/.../cellX**, then the principal name of KDS_X within the RS_X principal namespace is **krb5tgt/cellX**. Similarly, for a foreign cell *Y* whose cell name is **/.../cellY**, the principal name of KDS_Y within the RS_X principal namespace is **krb5tgt/cellY**.

- *dce-ptgt*

  PS in the cell.

- *dce-rgy*

  RS in the cell.

- *host-name***/self**

  SCD on the specified host (*host-name*) in the cell. This is also called the *host principal* or *machine principal* of the host *host-name*.

The following group name is *reserved* (within each cell):

- *none*

  Names a default group, intended to be used as the primary group associated with an account if no other group is specified (also used by some reserved accounts).

The following organisation name is *reserved* (within each cell):

- *none*

  Names a default organisation, intended to be used as the organisation associated with an account if no other organisation is specified (also used by some reserved accounts).

The RS supports the **rs_pgo** RPC interface for operating on RS PGO datastore items, which supports the following operations:

- *rs_pgo_add*( )

  Add a PGO item.

- *rs_pgo_delete*( )

  Delete a PGO item (also delete any *account* (see below) depending on the deleted PGO item).

- *rs_pgo_rename*( )

  Change the name of a PGO item.

- *rs_pgo_replace*( )

  Replace the information associated with a specified PGO item.

- *rs_pgo_add_member*( )

  Add a member principal to a group or organisation.

- *rs_pgo_delete_member*( )

  Delete a member principal from a group or organisation.

- *rs_pgo_is_member*( )

  Test whether a principal is a member of a group or organisation.

- *rs_pgo_get_members*( )

  Return a list of member principals of a group or organisation.

- *rs_pgo_get*( )

  Retrieve a PGO item (identified by any of several query key formats) from the RS datastore.

- *rs_pgo_key_transfer*( )

  Convert one query key format to another.

1.12.5 Accounts; rs_acct RPC interface

Accounts are the targets of login rituals. Account items consist of, minimally, a triplet of a principal, a *primary* group, and an organisation. The account may also contain a *list* of *secondary* groups known as a *concurrent group set*, which together with the primary group specifies the principal's *project list*; that is, all the groups to which the principal corresponding to the account belongs. The principal and group information in an account is typically constructed for a user (in a PAC, in a privilege-ticket) at login time, for use by the user throughout its login session.

Accounts are *named* by their principal domain component name. Thus, when a user responds to a ''login prompt'' with their principal name, they are identifying their account name as well. But even though a principal *name* can be associated with only one account, the principal itself (identified within its cell by its definitive principal UUID identifier) can belong to multiple accounts, because of the *alias* feature of the Principal domain (each alias of a principal can identify a different account).

The following accounts are *reserved* within each cell (expressed as a <P, G, O> triple):

- <**krb5tgt/***cell-name*, *none*, *none*>

  KDS account in the (local) cell (where *cell-name* is this cell's name).

- <*dce-ptgt*, *none*, *none*>

  PS account in the cell.

- <*dce-rgy*, *none*, *none*>

  RS account in the cell.

- <*host-name*/**self**, *none*, *none*>

  SCD account on the specified host (*host-name*) in the cell.

The RS supports the **rs_acct** RPC interface for operating on RS account items, which supports the following operations:

- *rs_acct_add*( )

  Add an account.

- *rs_acct_delete*( )

  Delete an account.

- *rs_acct_get_projlist*( )

  Return the project list for an account.

- *rs_acct_lookup*( )

  Return data for an account.

- *rs_acct_rename*( )

  Rename an account (associate account information to another principal name).

- *rs_acct_replace*( )

  Replace an account's data.

1.12.6 Miscellaneous; rs_misc RPC Interface

The RS supports the **rs_misc** RPC interface for miscellaneous operations:

- *rs_login_get_info*( )

  Retrieve local system login information from the RS.

1.13 ID Map Facility

The *ID Map Facility* maps (global) PGO (principal, group or organisation) names into their cell-name and cell-relative components, and to the UUIDs corresponding to them — and *vice versa*. It supports the **secidmap** RPC interface and the *ID Map* (or **sec_id**) API.

As has been mentioned already (at least for principals and groups, see Section 1.6 on page 25), PGOs in the DCE security environment are definitively identified by a double-UUID scheme; that is, by an ordered pair of (''computer-friendly'') UUIDs, consisting of a cell UUID and a per-cell PGO UUID — *pgo-ID* = <*cell-UUID*, *pgo-UUID*>, or:

- *principal-ID* = <*cell-UUID*, *principal-UUID*>
- *group-ID* = <*cell-UUID*, *group-UUID*>
- *organization-ID* = <*cell-UUID*, *organization-UUID*>

Note: Under normal circumstances, all *pgo-UUID*s in all cells will be distinct from one another. However, the security of the services specified in DCE require this property to hold only within cells, not necessarily across cells. This is discussed in detail in Section 1.6 on page 25.

Additionally, PGOs can be addressed by ''human-friendly'' (string)names, namely they are identified by a concatenated cell name and a per-cell PGO name — *pgo-name* = **/.../***cell-name*/*cell-relative-pgo-name* or:

- *principal-name* = **/.../***cell-name*/*cell-relative-principal-name*
- *group-name* = **/.../***cell-name*/*cell-relative-group-name*
- *organization-name* = **/.../***cell-name*/*cell-relative-organization-name*

Thus, what the ID Map Facility does is give a bidirectional mapping — *pgo-name* ↔ *pgo-ID* or:

/.../*cell-name*/*cell-relative-pgo-name* ↔ <*cell-UUID*, *pgo-UUID*>

(For more information concerning naming syntax, see Section 1.18 on page 84.)

Note that the ID Map Facility depends on *dynamic* (or *semantic*) information, because there is no *static* (or *syntactic*) way to decompose global PGO names into their cell-name and cell-relative name components. For example, a PGO name such as **/.../foo/bar/zot** is ''syntactically ambiguous'' (disregarding the fact that the actual syntactic string **foo** is not supported by any currently DCE-supported global naming service), in the sense that it is impossible to determine syntactically whether its components are:

cell-name = **foo** and *cell-relative-PGO-name* = **bar/zot**

or:

cell-name = **foo/bar** and *cell-relative-PGO-name* = **zot**

The DCE security facilities guarantee, however, that names are not ''semantically ambiguous''; that is, that only one (at most) principal (or group, or organisation) has the name **/.../foo/bar/zot**. Note, however, that there may be a principal *and* a group *and* an organisation all having the name **/.../foo/bar/zot**. (Some more discussion of the relationship between security and naming is given in Section 1.18 on page 84.)

The RS exports the **secidmap** RPC interface, to support the **sec_id** API. It supports the following operations:

- *rsec_id_parse_name*( )

  Decompose (or ‘‘parse’’) a global PGO name into its cell-name and cell-relative name parts, and their UUIDs.

- *rsec_id_gen_name*( )

  Generate (or ‘‘translate’’) a global PGO name, and its corresponding cell name and cell-relative name parts, from a cell UUID and a cell-relative UUID.

- *rsec_id_parse_name_cache*( ) and *rsec_id_gen_name_cache*( )

  These are the same as the above, but additionally support client-side cache management.

The **sec_id** API supports the following routines:

- *sec_id_parse_name*( )

  Decompose a global principal name into its cell-name and cell-relative principal name, and their UUIDs.

- *sec_id_parse_group*( )

  Decompose a global group name into its cell-name and cell-relative group name, and their UUIDs.

- *sec_id_gen_name*( )

  Generate (or ‘‘translate’’) a global principal name, and its corresponding cell name and cell-relative name parts, from a cell UUID and a cell-relative UUID.

- *sec_id_gen_group*( )

  Generate (or ‘‘translate’’) a global group name, and its corresponding cell name and cell-relative name parts, from a cell UUID and a cell-relative UUID.

1.14 Key Management Facility

The *Key Management Facility* provides the means by which ‘‘non-interactive’’ subjects (that is, ones that do not respond to an interactive ‘‘login’’ prompt, such as server programs) manage their long-term cryptographic keys (stored in the RS datastore, as well as in local key store). It supports the *Key Management* (or **sec_key_mgmt**) API.

Every account — interactive or non-interactive — in DCE has an entry in its cell’s RS datastore that specifies a long-term secret key. In the case of an interactive principal (in particular, an end-user), this long-term key is derived from the principal’s password (see Section 1.15 on page 71), and such principals need to keep their password secure by memorising it (rather than, say, writing it down). Similarly, a non-interactive principal (in particular, an RPC server) also needs to be able to store and retrieve its long-term key in a secure manner. This has both advantages and disadvantages: it is an advantage that a non-interactive principal’s long-term key need not be memorisable — that is, need not be derived from a password (because randomly generated keys are more secure than keys derived from passwords, since they are drawn from a larger key space and therefore cannot be ‘‘guessed’’ as easily as passwords (password-based keys)); but it is a disadvantage that a non-interactive principal does not have a secure storage area (such as a ‘‘brain’’) for memorising its long-term key. The Key Management Facility provides such a secure key management facility for non-interactive principals.

The RS datastore holds a (single) long-term key (together with its key version number, see below) for each account, which is referred to as the ‘‘current’’ long-term key. Since cryptographic keys become inherently less secure as more and more data is protected with them, it is good security policy to change long-term keys occasionally (this is analogous to interactive users changing their passwords). When the current long-term key is changed, however, there may continue to exist outstanding tickets protected with previous key(s) (in fact, this is the usual case). This necessitates a mechanism for keeping track of ‘‘usable’’ keys (that is, those for which outstanding usable tickets protected with these keys continue to exist). That problem is solved in DCE by providing for *key version numbers* and a *local key store*, whereby all usable (current and previous) long-term keys are tagged with a version number; and this information is maintained ‘‘locally’’ by each server (that is, securely with respect to security policy, and in an implementation-defined sense). It is the **sec_key_mgmt** API that realises this key maintenance facility. Implementations are required to retain keys of all version numbers for which there may exist outstanding tickets, unless such tickets need to be explicitly *revoked* because their keys are suspected of being compromised (see *sec_key_mgmt_delete_key*( ) and *sec_key_mgmt_delete_key_type*( ), below).

While these key management routines themselves are designed to be secure, the ultimate security of the long-term keys they manage — in particular the implementation-dependent local key storage of the keys themselves — depends also upon the security of local hardware and OS, and that involves questions of integration of DCE security and local security beyond the scope of this specification. Typical implementations will locally store keys in files (called *key table files*) managed by the local OS (and therefore protected by whatever means the local OS protects it files), although the Key Management Facility accommodates other storage means as well.

Finally, the Key Management Facility provides a means to *delete* keys (thereby *revoking* tickets protected with those keys) — non-interactive principals use this when old keys expire, or when a key is suspected of being compromised.

Some of the routines below take a *key type* as a parameter, which identifies the cryptoalgorithm in use (for example, DES).

The Key Management (or **sec_key_mgmt**) API consists of the following routines (except for those specifically noted to communicate with the RS server, they are all ‘‘local’’; that is, no protocol is specified for them, and the implementation must guarantee their security):

- *sec_key_mgmt_change_key*( )

  Change a principal's key to a specified value, both locally (in local key storage) and remotely (in the RS datastore).

- *sec_key_mgmt_set_key*( )

  Change a principal's key to a specified value, locally but not remotely.

- *sec_key_mgmt_gen_rand_key*( )

  Generate a random key. (Does not change principal's key, either locally or remotely.)

- *sec_key_mgmt_manage_key*( )

  Change a principal's key to a random value, locally and remotely, on a periodic schedule.

- *sec_key_mgmt_get_key*( )

  Retrieve principal's key, of a specified key version number, from local key storage.

- *sec_key_mgmt_free_key*( )

  Free memory allocated to a key.

- *sec_key_mgmt_get_next_kvno*( )

  Determine next key version number from RS.

- *sec_key_mgmt_initialize_cursor*( )

  Initialise a cursor to local key store.

- *sec_key_mgmt_release_cursor*( )

  Dispose of cursor to local key store.

- *sec_key_mgmt_get_next_key*( )

  Retrieve key pointed to by cursor to local key store.

- *sec_key_mgmt_delete_key*( )

  Delete keys of specified version number and all types from local key store (thereby ''revoking'' tickets protected with those keys).

- *sec_key_mgmt_delete_key_type*( )

  Delete key of specified version number and type from local key store (thereby revoking tickets protected with that key).

- *sec_key_mgmt_garbage_collect*( )

  Delete unusable keys (that is, those for which no usable ticket can exist) from local key store.

1.15 Login Facility and Security Client Daemon (SCD)

The *Login Facility* provides the means by which subjects (RPC clients) manage their DCE login context(s). It supports the *Login* (or **sec_login**) API.

A (''*network*'') *login context* contains the information necessary for a subject to become a client in the DCE security environment; that is, to invoke protected RPCs. Namely, a client ''annotates'' an RPC binding handle with the security information present in a login context, via *rpc_binding_set_auth_info*( ) (as described in the referenced X/Open DCE RPC Specification).

Login contexts are represented to the application programmer as an ''opaque pointer'' (that is, a pointer to a data structure whose internal structure is implementation-dependent and not further specified), but conceptually the information held in login contexts normally includes such items as:

- Identity information concerning the subject's account (stored in the RS datastore), such as principal stringname and principal/group UUIDs, contained in a ticket and privilege-ticket issued by the KDS and PS of the subject's cell. The protected parts of this information (including the session keys between the subject and the KDS and PS) may be encrypted (if the login context is *unvalidated*) or decrypted (if it is *validated*).
- (Registry) policy information, such as the maximum lifetime of tickets.
- State information, including the validation state of the login context (that is, whether or not it has been ''validated'' (decrypted)), and its certification state (that is, whether or not is has been ''certified'' (described below)).
- The authority for authentication information (it may originate from the DCE TCB, or from the local TCB if the network is unavailable for some reason).

Note: A ''login context'' need not be established by an actual interactive end-user login ritual, but may be established via the **sec_login** API. Furthermore, multiple login contexts may be associated with a single ''process context'', and *vice versa*. Another name for ''login context'' is (client-side) *security context*.

A login context is said to be *validated* if the information contained in it is trusted by the principal/account associated with the login context (provided, as always, that the principal/account trusts the security of its own long-term key). Said in more intuitive terms, an *unvalidated* (or *prevalidated*) login context is one which is still protected (that is, partially encrypted) in the long-term key of the principal/account with which it is associated; a *validated* login context is one which has been (correctly) decrypted (and hence can be trusted to the extent that the key used to decrypt it is trusted). The practical meaning of ''validation'' is that the login context is in a format (namely, decrypted) that can be used for protected RPCs (that is, to ''annotate'' RPC handles), and that the client trusts it for this purpose.

A more specialised notion than that of validation is certification. A validated login context is said to be *certified if* — and to the extent *that* — it is trusted by *parties other than* the principal/account associated with the login context. The prototypical example of such an ''other party'' is the local TCB of the client's host, especially its ''login program'' (or other ''entrance portal''): the local TCB typically requires a high level of trust in a client's login context, because it typically uses the client's network login context to establish its local login context, issuing the client its local security credentials on the basis of its global (''network'') credentials (by means of a host-specific ''UUID-to-host-ID'' mapping), thereby enabling the client to access local OS resources (modulo local access controls). However, an ''other party'' could just as well be a non-TCB party (that is, another principal) instead. This certification of a (validated) login context, by another party suspicious of it, can be accomplished if the other party can use the login context to successfully ''impersonate'' the principal/account associated with the login

context (note that the other party has access to the required identity information to attempt this, since it is present in the login context — in particular, the client must trust the other party to the extent that it exposes this information to it): for, if the login context can be used to successfully execute a protected RPC to a trusted party (say, to the local TCB itself), then the information in the login context must have been genuine; that is, it is trustable. (There is, however, a subtle point regarding ''trusted communications'' between the other party and its local TCB — this is discussed in Section 1.15.2 on page 77.) This model requires that the local TCB is instantiated as a (server) principal with respect to the DCE TCB — and it is the function of the *Security Client Daemon* (SCD) to perform that role: the SCD ''is the DCE principal of the local machine''. It has principal name *host-name*/**self** (within its cell). The SCD supports the **scd** RPC interface for the purpose of supporting certification.

Note: There is no *necessary* relationship between the DCE host name, *host-name*, and any other (non-DCE) name the machine may be endowed with (for example, last component of Internet host name). By *convention*, however, *host-name* typically has the form **hosts/***short-host-name* (within its cell), where *short-host-name* is ''the same'' short name that the machine is known by for other purposes (for example, the last component of an Internet host name).

Of all the login contexts supported by a given process, there is (at most) one that is distinguished among them, called the *process' current login context*. This is a process-wide notion (and not, for example, a thread-specific notion). A process' current login context is, by definition, the (only) login context that is (automatically) inherited by child processes; it is very common for a process to possess only a single login context, and for this to be designated its current login context. The current login context is *also* the *process' default login context*, which is by definition the login context that is considered to be in effect in situations where a login context is required but none is otherwise specified (see, in particular, *rpc_binding_set_auth_info*( ) in the referenced X/Open DCE RPC Specification). Care should be taken not to confuse the (process-wide) notions of current and default login context with the notion of *the host's login context*, which is by definition the login context associated with the host's principal/account itself (''*self*'').

A process at its start-up time may or may not have a current login context, depending on its parent's actions (namely, whether or not the parent had designated a current login context and enabled or disabled current login context inheritance). This consideration is especially interesting in the case of a ''machine boot'' scenario, where for typical implementations it is desirable for an operating system's ''initial process'' (sometimes called *init*) to ''run as the host principal'' (that is, to have its current login context be the login context of the host principal/account), and for that login context to be inherited by the other boot or ''daemon'' processes (but not by ordinary ''user'' processes). Special routines are included in the **sec_login** API (below) to cater to this important special case.

An end-user command for ''logging in'' to the DCE environment is not specified in this document, though implementations will typically provide a host-specific ''*login program*'' (or ''login command''), and give it the following functionality (all relative to the user's home cell):

- Request the user's login name, and obtain a ticket for that user from the KDS. (That is, begin to set up a login context for the user; see *sec_login_setup_identity*( ) below.)
- Request the user's password, map it to a cryptographic key, and verify that it is the same as the user's long-term key protecting the ticket (by verifying that it correctly decrypts the ticket). (That is, validate and complete the set-up of the login context; see *sec_login_validate_identity*( ) below.) Then, send the ticket to the KDS, requesting a ticket to the PS; and send this ticket to the PS to obtain a privilege-ticket for the user, and decrypt that.
- Map the user's ''network identity information'' (stored as a stringname in the ticket, and as UUIDs in the PAC (or obtainable through the EPAC set *seal*) in the privilege-ticket) to ''local

host identity information'', after first checking that the information contained in the ticket and privilege-ticket is trustworthy. (That is, certify the login context; see *sec_login_certify_identity*( ) below.) (Actually, for the important special case of a (locally) privileged process — that is, a trusted process in the host's TCB, such as its login program — a special routine is supported, *sec_login_valid_and_cert_ident*( ), which not only combines validation and certification, but is ''more infallible'' than *sec_login_certify_identity*( ) — see Section 1.15.2 on page 77.)

- Arrange for the user's login context to be inherited by the login process hierarchy, so that it is available for the user's entire login session (typically, the user's ''login shell'' and its child processes (recursively)). (That is, set the current context; see *sec_login_set_context*( ) below.)

To accomplish these activities, the login program uses (parts of) the **sec_login** API, which consists of the following routines. These routines are designed to be called both by interactive host login programs (as described above), and by non-interactive programs that need to act as RPC clients (under potentially multiple principal/account identities). A few of these routines are primarily designed to be called by specific programs, such as a host's login program, or the local host's first (or ''initial'') process (''*init*'') after a system boot.

- *sec_login_init_first*( )

  Initialise the calling process' current login context inheritance mechanism, thereby making the calling process' current login context (potentially) accessible to other processes on the local host. In typical usage, this routine is called only by the host's initial process at boot time (''*init*'') to initialise the host's login context for inheritance by the host's hierarchy of daemon processes which are spawned as child (and sub-child) processes of the initial process. (The calling process' current login context is actually ''set up'' by *sec_login_setup_first*( ), below.)

- *sec_login_setup_first*( )

  Similar to *sec_login_setup_identity*( ) (below), except that the only login context it can set up is the local host's (unvalidated) login context (that is, the login context associated with the host's principal/account (''*self*'')). In typical usage, this routine is called only by the host's initial process (or, if this functionality has not been integrated into the host's initial process, by SCD).

- *sec_login_validate_first*( )

  Similar to *sec_login_validate_identity*( ) (below), except that the only login context it can validate is the local host's login context.

- *sec_login_setup_identity*( )

  Set up a login context; that is, create an (unvalidated and uncertified) login context for a specified principal/account (recall that accounts are uniquely identified by their principal name component), in the address space of the calling client. Such a login context is protected (that is, parts of it are encrypted), and it is not usable until it has been validated (that is, decrypted, by *sec_login_validate_identity*( )).

- *sec_login_validate_identity*( )

  Validate a login context; that is, make it usable for making protected RPCs (in the sense of making it usable by *rpc_binding_set_auth_info*( )), and in the process demonstrate its trustworthiness (for use in protected RPCs) to the principal/account to which it is associated (assuming the security of the long-term key of the principal/account associated with the login context). This is typically accomplished by decrypting the encrypted part of the login context (and verifying that the decryption is correct), using the long-term key of the

principal/account — hence, this information must have been encrypted by an entity knowing the principal/account's long-term key, which must have been a trusted entity (by hypothesis).

- *sec_login_certify_identity*( )

  Certify a (validated) login context; that is, demonstrate its trustworthiness (for the purpose of basing access decisions on information carried in it) to parties other than the principal/account to which it is associated. This is typically accomplished by impersonating (as a client) the principal/account to which the login context is associated, by verifying that the login context can be used to execute a protected RPC to the local host's SCD. (See Section 1.15.2 on page 77.)

- *sec_login_valid_and_cert_ident*( )

  Simultaneously validate and certify a login context (that is, combine the functionality of *sec_login_validate_identity*( ) and *sec_login_certify_identity*( ) into a single routine), in a manner appropriate for use by *privileged* processes. This is typically accomplished by impersonating the local host's SCD, which may be thought of as ''the local TCB invoking a protected RPC to itself'', and is ''infallible'' (that is, completely secure, modulo the security of the local TCB). (See Section 1.15.2 on page 77.)

- *sec_login_set_context*( )

  Set a login context; that is, *register* it in the sense of making it (potentially) accessible to other processes on the local host, and moreover make it the calling process' current login context.

- *sec_login_get_current_context*( )

  Retrieve the calling process' current login context.

- *sec_login_release_context*( )

  Release a login context; that is, free the memory allocated to it (this does not affect the accessibility of other processes to the login context).

- *sec_login_purge_context*( )

  Purge a login context; that is, *unregister* it in the sense of making it inaccessible to the calling process and to other processes on the local host (typically, this involves eradicating all memory and disk storage of the login context).

- *sec_login_export_context*( )

  Export a login context; that is, create an *advertisement* for it. Such an advertisement consists of information that can be communicated to other processes and enables them to (potentially) share the login context (such sharing is restricted to processes on the local host).

- *sec_login_import_context*( )

  Import a login context; that is, create a login context from its advertisement.

- *sec_login_newgroups*( )

  Restrict the list of groups associated with a login context. (This is designed for supporting ''least privilege'' security policies that require clients to act with the minimal set of privileges necessary for successful completion of their tasks.)

- *sec_login_get_groups*( )

  Retrieve local host group membership information from a login context.

- *sec_login_get_expiration*( )

  Retrieve the *expiration date* of a login context, which is the date beyond which RPC binding handles *annotated* with the login context (in the sense of *rpc_binding_set_auth_info*( ) in the referenced X/Open DCE RPC Specification) will fail.

- *sec_login_refresh_identity*( )

  Refresh a login context; that is, increase its expiration date to the maximum allowable; the refreshed login context must be revalidated.

- *sec_login_inquire_net_info*( )

  Retrieve certain ''network information'' (PAC data and expiration data) from a login context.

- *sec_login_free_net_info*( )

  Free memory allocated for ''network information'' structure.

- *sec_login_get_pwent*( )

  Retrieve local host information associated with a login context.

In addition, support for delegation introduced into DCE in DCE 1.1 provides the additional functionality listed in the next section.

1.15.1 Delegation Related Functions

The following functions are used to establish delegation chains and perform delegation related functions. The provide the following functionality:

- The ability to enable delegation.
- Also, the ability to select the type or form of delegation desired. This permits the following capabilities:
  - The ability of an intermediary service to *impersonate* the initiator of the service. In this instance, the identity of the (impersonator) participant in the call chain is not preserved in a call chain of participants associated with performing the requested service.
  - The ability of an intermediary service to act as a *delegate* on behalf of the initiator of the service. In this instance, the identity of the (delegated) participant is preserved in a call chain of participants associated with performing the requested service.

To accomplish this, the implementation's host-specific ''*login program*'' (or ''login command'') uses (parts of) the **sec_login** API consisting of the following routines. As previously mentioned, these routines are designed to be called by both interactive host login programs and by non-interactive programs that need to act as RPC clients. The routines are:

- *sec_login_become_delegate*( )

  This function is used by intermediate servers to become a delegate for their caller.

- *sec_login_become_impersonator*( )

  This function is used by intermediate servers to become an impersonator for their caller.

- *sec_login_become_initiator*( )

  This function constructs a new login context that enables the selected delegation type. The login context is unvalidated and uncertified. Unless this function is invoked, delegation is not permitted on behalf of the the originator of a request for service.

The delegation types that are permitted are *impersonation* and *(traced) delegation*.

- *sec_login_cred_get_delegate* ( )

  This function is used to iterate through and extract the privilege attributes of the delegates listed in a specified login context.

- *sec_login_cred_get_initiator* ( )

  This function is used to extract the initiator's privilege attributes from a specified login context.

- *sec_login_cred_init_cursor* ( )

  This function is used to intiialise a **sec_cred_curson_t** to be used as a cursor to iterate through the list of delegates for a service. It is used in calls to the iterative routine **sec_login_cred_get_delegate**.

- *sec_login_disable_delegation* ( )

  This function returns a login context without *delegation* or *impersonation* enabled, from one that has one of the two delegation types enabled.

- *sec_login_purge_context_exp* ( )

  This function destroys expired network credentials associated with a specified (AS) ticket.

- *sec_login_set_extended_attrs* ( )

  This function constructs a new login context that contains the requested attributes.

  Note: Attributes cannot be added to a delegation chain in this manner. Thus, if a login context referring to a delegation chain is passed to this routine, an error indication an invalid context will be returned.

- *sec_login_tkt_request_options* ( )

  This function is used by a client to request specific AS ticket options. It is an optional function. It is designed to be called after *sec_login_setup_identity* ( ) or *sec_login_refresh_identity* ( ) and before *sec_login_validate_identity* ( ) or *sec_login_valid_and_cert_ident* ( ).

  The requeste options will override the defaults when the ticket is requested at validation time.

The SCD exports the **scd** RPC interface, which supports the following operation:

- *scd_protected_noop* ( )

  This operation is nothing more than an ''protected no-op'', which serves the purpose of ''certifying the login context'' that was involved in the client's invocation of this RPC call (see *sec_login_certify_identity* ( ) above).

1.15.2 Further Discussion of Certification

The notion of certification (as introduced in Section 1.15 on page 71) is a subtle one, in respect both of:

1. the need for such a notion
2. the requirements it makes upon implementations.

Therefore, in order to clarify this notion, this section discusses it in more depth, with special emphasis on these two points. The synopsis of this section is as follows:

> There exists a certain (very small) risk that a login context can be returned by *sec_login_setup_identity*( ) and validated by *sec_login_validate_identity*( ), yet still be counterfeit. The scenario for this to happen is a rather arcane attack, called a *multi-prong* attack (described below). Applications that are concerned about the risk presented by this threat must defend themselves against this multi-prong attack. Simple validation is insufficient for this defense; the more subtle notion of certification is required. *It is the thwarting of this multi-prong attack that is precisely the purpose of ''certification'' — no more, no less.* On the other hand, applications that are unconcerned about the risk presented by this threat of a multi-prong attack need not certify their login contexts (and it is to support this latter set of applications that *sec_login_certify_identity*( ) is provided as a *separate* routine).

To simplify the exposition, it is convenient to begin the discussion in this section with a definite example. To this end, the focus will begin on the concrete case of a host's *login program* (which resides in a host's local TCB, and is *privileged*) — the general case of arbitrary (not necessarily privileged) ''other parties'' as envisioned in Section 1.15 on page 71 will then be a corollary of this discussion of the login program.

Consider, then, a user *U* attempting to log in to a host *H*, claiming (a string name that identifies) a principal identity *A*, and presenting a password *P* which maps to a long-term key K_P by means of a well-known algorithm (see Section 4.3.6.1 on page 190). The login program *L* begins by calling *sec_login_setup_identity*( ) to set up a login context, and *sec_login_validate_identity*( ) to validate it. In terms of the Kerberos authentication protocol (see Section 1.5 on page 18 and Section 4.12 on page 220), *sec_login_setup_identity*( ) sends an AS Request message (intended to be received by the genuine KDS (in *A*'s home cell)) and receives an AS Response message (intended to be received from the genuine KDS), and the successful invocation of *sec_login_validate_identity*( ) convinces *L* that the AS Response message was correctly protected with the key K_P. If *L* could somehow be certain that $K_P = K_A$ (where, as usual, K_A denotes the long-term key of *A*, held in the genuine RS datastore (in *A*'s home cell)), then *L* could be confident that *U* ''really is'' *A*, and that the (suspicious) ticket, $Tkt_{U,KDS}$ (let's call it), received in the AS Response (and contained in the login context) really is a genuine $Tkt_{A,KDS}$ naming *A*. This $Tkt_{A,KDS}$ could then be used to obtain a ticket, $Tkt_{A,PS}$, to the genuine PS (in *A*'s home cell) and a privilege-ticket, $PTkt_{A,KDS}$ containing a trustable *seal* of an $EPAC_A$ set (or PAC_A), and *L* could then with confidence retrieve this $EPAC_A$ set *seal* (or PAC_A) from the login context (using *sec_login_inquire_net_info*( )), and map the information in the $EPAC_A$ set *seal* (or PAC_A) to trustable local OS credentials. Note here that the EPAC set contains one EPAC for the initiator and one EPAC for each delegate involved in the operation when traced delegation is in use. In order to be sure that the privilege-ticket, $PTkt_{A,KDS}$ is genuine in this scenario, the *seal* the PS (in *A*'s home cell) places into $PTkt_{A,KDS}$ must be for the ordered list of EPAC *seal*s in the EPAC set. In turn, in order for *L* to be certain that $K_P = K_A$, it is sufficient that *L* be certain that the AS Request/Response message exchange had actually been conducted with the *genuine* KDS.

However, *L* needs to be *convinced* of this — it cannot blindly *assume* that the AS Request/Response message exchange had been conducted with the genuine KDS (because the AS service is unauthenticated). Namely, there is a threat that *L* is the target of a *multi-prong*

attack, whereby a malicious user *U* collaborates with *bogus security servers* (that is, malicious RPC servers that masquerade as the genuine KDS, PS, RS and SCD servers), with the goal of the attack being to trick *L* into believing that a counterfeit login context is genuine, and thereby granting *U* an unauthorised login session on the targeted host. In such an attack, the principal *A* (which properly exists in the genuine RS datastore, with its genuine long-term key K_A stored there) would have a fraudulent entry in the bogus RS datastore, with the fraudulent long-term key K_P stored there. If an unwary login program *L* were to accept mere validation as proof of the authenticity of login contexts, it would be vulnerable to a multi-prong attack such as this. What is required is a way for *L* to *certify* login contexts; that is, to defend against such a multi-prong attack.

What, then, is involved in certification? That is, what does it take to convince *L* that $K_P = K_A$? By tracing backwards along the trust chain involved in the following sequence of tickets (where the subscript ''*U*'' indicates that *L* is suspicious of these tickets until it can be convinced of their genuineness):

$$Tkt_{U,KDS} \rightarrow Tkt_{U,PS} \rightarrow PTkt_{U,KDS} \rightarrow PTkt_{U,SCD}$$

it can be deduced that: if $PTkt_{U,SCD}$ is correctly protected with the genuine long-term key K_{SCD} of the local host's SCD, and if *L* can be convinced of this fact, then *L* is justified in believing that all these suspicious tickets are in fact genuine, and in particular that the long-term key protecting $Tkt_{U,KDS}$ is K_A — that is, that $K_P = K_A$. (All this assumes that the DCE network TCB and the local host's TCB are uncompromised, of course.)

At this point, recall that *L* is part of the local host's TCB; that is, is *privileged*. Therefore, it is possible (and is not a breach of security) for *L* to (legitimately) adopt the identity (as a server) of the local host's SCD, in the sense of *L*'s knowing and using the long-term key K_{SCD} (which *L* can retrieve by using *sec_key_mgmt_get_key*( ) or an implementation-specific equivalent). If *L* can successfully use K_{SCD} to directly verify that $PTkt_{U,SCD}$ is correctly protected with the genuine long-term key K_{SCD}, then the certification problem is thus solved for *L*. This adoption of the SCD's identity by *L* can be thought of as ''the local TCB invoking a protected RPC to itself'', and it is ''infallible'' (in the sense of being completely secure, modulo the security of the network and local TCBs). And this is typically how *sec_login_valid_and_cert_ident*( ) is implemented (furthermore, this direct access to K_{SCD} shows ''why'' *sec_login_valid_and_cert_ident*( ) is a *privileged* routine).

While this adoption of the SCD's identity solves the certification problem for privileged processes, it is not a solution that is available to *non-privileged* processes (because it is a breach of security for non-privileged processes to have knowledge of the local TCB's long-term key K_{SCD}; that is, to have K_{SCD} in their address spaces). This is why *sec_login_certify_identity*( ) is supported. The design criterion for *sec_login_certify_identity*( ) is that it is intended to provide the *strongest guarantee of certification that can be provided to a non-privileged process, modulo the exigencies of a given implementation*. The strength of this guarantee is, in general, *implementation-specific* — in particular, *sec_login_certify_identity*( ) may be ''not quite as infallible'' as *sec_login_valid_and_cert_ident*( ) (in which case, this must be specified in the implementation's documentation of *sec_login_certify_identity*( )).

Note: In typical implementations, *sec_login_certify_identity*( ) is implemented as a protected RPC intended to be handled by the local host's SCD server, at a protection level other than **rpc_c_protect_level_none** (see Section 1.10 on page 54), and annotated (in the sense of *rpc_binding_set_auth_info*( )) using the suspicious login context in question (that is, using $PTkt_{U,SCD}$ to authenticate the invoker of *sec_login_certify_identity*( ) to the SCD server). If this RPC returns successfully (that is, the SCD's *scd_protected_noop*( ) operation succeeds as a protected operation), then, similarly to the case of *sec_login_valid_and_cert_ident*( ) above, the calling application is justified

in concluding that $PTkt_{U,SCD}$ is correctly protected with the genuine long-term key K_{SCD}, and the certification problem is thus solved in the non-privileged case.

There is a caveat in this scenario, however; the implementation of the RPC (called inside *sec_login_certify_identity*( )) must somehow guarantee (in a manner that *does not depend* on the suspicious $PTkt_{U,SCD}$) that this RPC *really is handled by the local host's genuine SCD server*. To the extent that the implementation can make this guarantee, *sec_login_certify_identity*( ) can be trusted (that is, approaches the ''infallibility'' of *sec_login_valid_and_cert_ident*( )). An example of criteria that implementations may support that are sufficient to make this guarantee (and thereby make certification via *sec_login_certify_identity*( ) *just as infallible* as that via *sec_login_valid_and_cert_ident*( )) is the following:

The client's RPC to the SCD is:

1. done over a trusted local host-only transport (such as a ''local loop-back transport'', or ''UNIX-domain sockets'', or ''local shared memory'', and so on)
2. addressed to a trusted transport endpoint on which the SCD is listening (for example, the SCD could have written this to a world-readable file, protected by local host security, at its boot time, from which *sec_login_certify_identity*( ) could read it in a trusted fashion).

Not all platforms are able to meet the above criteria, and even those that do may not do so in a portable manner. Another example of implementation criteria, which are more widely supported and portable but which, however, give a guarantee *not* as ''infallible'' as the preceding, are the following:

The client's RPC to the SCD is:

1. done over an untrusted transport implemented in such a way that messages addressed to transport endpoints on the local host do not go off-host (this property, which is quite common in existing implementations, guarantees that the conversation key K^^^ in criterion (3) below is not exposed to network eavesdropping)
2. addressed to a trusted transport endpoint on which the SCD is listening (see above)
3. the conversation key K^^^ involved in the RPC is specified *by the client* and is used by the SCD (to protect its returned *error_status_ok* status value). (Note that criterion 3. is satisfied by the CL RPC protocol (see Section 9.2.1.2 on page 332), but not by the CO RPC protocol (see Section 9.3.1.3 on page 340).) (The threat of off-host bogus servers sending messages to the client at exactly the right moments still exists, but the danger of doing so successfully is lessened by the presence of the client-chosen conversation key.)

If an implementation of *sec_login_certify_identity*( ) does not support the same strong guarantee of ''infallible'' certification that *sec_login_valid_and_cert_ident*( ) does, this fact (as well as information about the strength of the guarantee that actually is supported) must be noted in the implementation's documentation of *sec_login_certify_identity*( ).

1.16 Integration with Time Services

It is a fundamental characteristic of cryptographic algorithms and protocols based on *computational complexity* (as opposed to those based on ‘‘theoretically perfect security’’, such as so-called *one-time pads* (for example, single-use codebooks)) that their security depends in an essential way on *time.* The reason is that their security can be undermined if the cryptanalyst has access to sufficient computational power to overcome the barriers imposed by computational complexity, and availability of such computational power can be guaranteed (at least theoretically, if not practically) if enough *time* is available. For example, given enough time, a cryptanalyst can ‘‘crack’’ (that is, cryptanalyse) a key-based cryptosystem by simply ‘‘exhausting the keyspace’’; attempting to decrypt encrypted messages using all possible keys until one ‘‘correctly’’ decrypts the message (provided the ‘‘correct’’ decryption can be recognised, for example, by recognising some ‘‘verifiable plaintext’’). Further, the longer a key has been in use, the more traffic has likely been protected with it, and therefore the more likely it will become the target of a compromise attempt. Thus, among other considerations, the length of time a key is valid needs to be limited.

As a countermeasure intended to thwart such attacks, the Kerberos authentication protocols use *timestamps,* in several ways:

- *Tickets*

  Four timestamps in each ticket are used as *timeouts* to bracket the validity of the ticket; that is, to delimit the times at which the clients should present and servers should honour the ticket (that is, use the session key in the ticket or a negotiated conversation key, with good expectations of security). These timestamps are generated by the KDS itself, and interpreted by the target server. (With most current technology, a day is usually considered a relatively safe lifetime for a DES key.)

- *Authenticators*

  One timestamp (broken up into two fields) is used in each *authenticator,* as a *freshness constraint* to counter *replay attacks* (see below). This timestamp is generated by the originating client and interpreted by the target server, to prove to the server that the client ‘‘currently’’ knows the session key (where a *clock skew* on the order of a few minutes is built into the interpretation of ‘‘currently’’, to compensate for network or other processing delays — this clock skew is taken into account in all other authentication protocol timestamps, too).

  (‘‘Replay attack’’ in general refers to a wiretapper intercepting a message and later (perhaps after disabling the initiator’s system, for example, for the purpose of using its network/transport address) resending it to the intended target claiming to be the original sender. In the present case, the replay attack threatens only to provide opportunities for bogus authentications, not to expose keys — thus only the security attribute of authenticity is at risk, not integrity or confidentiality.)

- *Other*

  Timestamps are also used in various other places in the authentication protocol (generated and interpreted by various entities, sometimes protected and sometimes not protected), and elsewhere (for example, as timeouts for long-term keys in the RS). Details are discussed at appropriate places in later chapters.

Some compromises of timestamp security might be tolerated if they only lead to denial of service, but in order to generally guarantee the security of the authentication protocols, timestamps must be not only protected by the protocol itself, but they must also be secure when initiators *generate* them, and when receivers *interpret* them. That is, all these entities must have a *secure source of (accurate) time* available to them.

This secure source of time is provided in the DCE environment by the *Distributed Time Service* (DTS) — which therefore needs to be considered as a component of the DCE TCB. (This does not preclude other sources of trusted time being used.) In the DTS architecture, communications among DTS entities are protected, leaving the only exposed transmission of time information being the original importation of time from an external source into the DCE environment. Securing that external trust link, together with resolving the issues involved in bootstrapping and configuration of the DCE environment (for example, to break the potential ''vicious circle'' of the symbiotic relationship of DTS depending on Security and *vice versa*), while security-relevant, are administrative (not architectural) concerns, and as such are beyond the scope of this specification. That is, this aspect of the security environment is implementation-dependent.

1.17 Integration with RPC Services

The *programming model* supported by DCE endeavours to present security *services* to ''main-line'' RPC programmers in a way that largely insulates them from the intricacies of the underlying security *mechanisms*, and from the security APIs and RPC interfaces specified in this specification, thereby enhancing assurances that security-sensitive applications can be written correctly. The security RPC APIs (specified in the referenced X/Open DCE RPC Specification) relevant to this purpose are the following:

Notes:

1. Prior to DCE 1.1, servers would call *rpc_binding_inq_auth_client*( ) to obtain a handle to a client's credentials (**rpc_authz_handle_t**). This handle was not opaque and was dereferenced with a cast to the data type **sec_id_pact_t** *.

   Because that datatype lacks abstraction, it cannot be used to obtain DCE 1.1 (or beyond) credentials, which may be a chain of EPACs. Instead, *rpc_binding_inq_auth_caller*( ) has been added for use by all servers for DCE 1.1 and newer versions. It replaces the *rpc_binding_inq_auth_client*( ) which is still available for use with existing (pre-DCE 1.1) application servers.

2. The security-relevant parameters to these interfaces relating to authentication, protection and authorisation services, **rpc_c_authn_dce_secret** and **rpc_c_authz_dce**, are introduced elsewhere in this chapter.

- *rpc_server_register_auth_info*( )

  Set server's security information, by registering it with the RPC runtime.

- *rpc_binding_set_auth_info*( )

  Set client's security information, by ''annotating'' an RPC binding handle to a server.

- *rpc_binding_inq_auth_caller*( )

  Server retrieval of an authenticated client's security information from a binding handle.

- *rpc_binding_inq_auth_client*( )

  Server retrieval of client's security information from a binding handle.

  Note: This call is provided for compatibility with pre_DCE 1.1 applications only. It should not be used for DCE 1.1 and newer version applications.

- *rpc_binding_inq_auth_info*( )

  Client retrieval of server's security information from a binding handle.

- *rpc_mgmt_inq_server_princ_name*( )

  Return server's principal name to client.

- *rpc_mgmt_set_authorization_fcn*( )

  Establish server's authorisation function, for its RPC management routines.

Finally, it is to be noted that security metadata (such as tickets and authenticators), apart from its implicit appearance within protected RPC protocols (and therefore *invisible* from the point of view of the programming model), appears as explicit application-level data in the **krb5rpc** and **rpriv** RPC interfaces. Due to the specification of the transfer syntax of this data (in ASN.1/BER, not IDL/NDR), these RPC interfaces are specified in terms of the IDL **byte** data type (so-called ''opaque RPC transport'', not making use of the marshalling features of IDL). These RPC interfaces are invoked at protection level **rpc_c_protect_level_none** (because the data they carry

is already protected by direct cryptographic means, not via ‘‘protected RPC’’).

Note: Some implementations may wish to offer KDS services over ‘‘raw’’ UDP (port 88) in order to be compatible with other implementations of the Kerberos protocols, but that is an implementation choice and this document makes no specifications along those lines.

1.18 Integration with Naming Services

As discussed in Section 1.12 on page 60, the RS in every cell maintains a (RS) Policy item and three datastore ''domains'' of PGO items. For *datastore query/lookup purposes*, the RS addresses these items by names managed by the RS server itself, called *RS-names* or *security-relative names* (the relationship of RS-names to other kinds of names is specified in this section). The RS-names of the Policy item and of the PGO items have the forms:

- **policy**

  (RS) Policy item.

- **principal/***P-name*

  Principal items.

- **group/***G-name*

  Group items.

- **org/***O-name*

  Organisation items.

P-names, G-names and O-names are collectively known as *PGO-names*. (For *reserved* PGO-names, see Section 1.12.4 on page 63. See also Section 11.5 on page 379, where the three PGO ''domains'' are identified by explicit parameters instead of by initial namestring components — a naming technique equivalent to the namestring syntax discussed here, but more convenient for ''computer-oriented'' use, as opposed to ''human-oriented'' use.)

Now, the RS protects its (Policy and PGO) items with ACLs, hence in accordance with Section 1.11 on page 55, these protected items must be *named*. In order to name these protected RS items, DCE specifies that the RS in every cell must be registered, not only as an RPC service in CDS as a simple RPC server entry (via its **rs_bind** interface), *but also as an RPC server group entry*, called the *security junction RPC group*, referred to as ''*sec-junction*'' (concerning the notion of junctions, see Section 1.11 on page 55). Within each cell, the actual name of *sec-junction* is not itself directly specified, but it is indirectly identified in the **/.:/cell-profile** RPC profile entry (which *is* a well-known name) as having the name associated with a certain UUID (namely, that of the **rs_bind** interface). (Conventionally, *sec-junction* is usually simply called *sec*.) For more information on this topic, see Section 1.18.1 on page 86.

That is, the (RS) Policy and PGO items held in the RS in each cell are identified, *for the purposes of ACL editing* (''as for protected objects in the RS datastore itself''), by names appropriate to the **sec_acl** API interface, and these have the form:

- **/.../***cell-name*/*sec-junction*/**policy**
- **/.../***cell-name*/*sec-junction*/**principal/***P-name*
- **/.../***cell-name*/*sec-junction*/**group/***G-name*
- **/.../***cell-name*/*sec-junction*/**org/***O-name*

Note that these names, because they include the /*sec-junction/domain* substrings (where *domain* varies over *principal*, *group* and *org*), are slightly different from the global principal and group names appropriate to the ID Map Facility. (Note the ID Map Facility has no notion of the Policy item at all; and while the **secidmap** RPC interface supports ''organisation names'', the **sec_id** API doesn't.) Namely, global principal (resp., group) names consist only of the cell-name and the cell-relative principal (resp., group) name, where the latter is defined to be ''the same as'' the corresponding RS datastore item's P-name (resp., G-name) — the *sec-junction/domain* substring is not included. Thus, principals (resp., groups) have ''the same'' global names as their

corresponding RS datastore items, except that the substring *sec-junction*/**principal/** (resp., *sec-junction*/**group/**) is *omitted*.

Thus, for example, the ''partially qualified'' terminal string **foo/bar/zot** taken out of context is ambiguous, because it can name several different things depending on its *context*, as follows:

- ''The principal **foo/bar/zot** in a cell'' (in a context where a principal name is appropriate) has the fully qualified name (in the sense of the ID Map Facility):

  /.../*cell-name*/**foo/bar/zot**

- ''The RS datastore item in a cell holding information for principal **foo/bar/zot**'' (that is, having this P-name) has the fully qualified name (in an ACL management context):

  /.../*cell-name*/*sec-junction*/**principal/foo/bar/zot**

- ''The group **foo/bar/zot** in a cell'' (in a context where a group name is appropriate) has the fully qualified name (in the sense of the ID Map Facility):

  /.../*cell-name*/**foo/bar/zot**

- ''The RS datastore item in a cell holding information for group **foo/bar/zot**'' (that is, having this G-name) has the fully qualified name (in an ACL management context):

  /.../*cell-name*/*sec-junction*/**group/foo/bar/zot**

- ''The RS datastore item in a cell holding information for organisation **foo/bar/zot**'' (that is, having this O-name) has the fully qualified name (in an ACL management context):

  /.../*cell-name*/*sec-junction*/**org/foo/bar/zot**

A final question arises about the relationship between client and server principal names and their CDS-registered service names (holding binding information) as RPC applications. That there is no *necessary* relationship is easily seen from the fact that both clients and servers must have principal names to participate in the DCE security environment, yet for RPC binding purposes only servers (not clients) need be registered with CDS (and in fact, even servers need not be registered if ''string bindings'' are used). Similarly, there is no *necessary* relationship between the principal name of an RPC server and its CDS service name. The simplest *convention* is for servers to have ''the same'' name in both roles, for example:

- ''The principal **foo/bar/zot** in a cell'' (in a context where a principal name is appropriate):

  /.../*cell-name*/**foo/bar/zot**

- ''The service **foo/bar/zot** in a cell'' (that is, the CDS entry where its RPC binding information is kept):

  /.../*cell-name*/**foo/bar/zot**

However, this is merely a *convention*, and is not required by DCE. All that is really required is that the means by which the principal name of a server (or any other subject) is obtained must be *secure*. It is best to consider the principal name to be part of the service's very *definition* (or ''service contract'') — just as much a part of its definition as, say, its service (binding) name or its RPC interface UUID — and that an incorrect specification of any part of the service's definition will lead to incorrect results.

1.18.1 RPC Binding Models

This section gives more detailed information on the *RPC binding models* supported by DCE; that is, the manner in which RPC clients use the CDS directory services to locate and establish an RPC communications context with the RPC servers supported by DCE — both ''TCB servers'' (servers belonging to the DCE TCB, namely, RS, KDS, PS, and local hosts' SCD's), as well as ''ACL servers'' (RPC servers, possibly, but not necessarily, belonging to the TCB, that support protected objects (via ACLs), by exporting the **rdacl** RPC interface). This section requires familiarity with the referenced X/Open DCE RPC Specification and referenced X/Open DCE Directory Services Specification, and it has a closer affinity with those specification than it does with the security-specific material of the present specification.

1.18.1.1 Binding to TCB Servers

By specification, in every cell there exists in the cell's CDS namespace, a well-known CDS node known as **/.:/cell-profile**, which is an RPC profile node. Also in every cell, there exists by specification in the cell's CDS namespace a CDS node known as **/.:/***sec-junction*, which is an RPC group node; the actual name of **/.:/***sec-junction* is not ''well-known'' (it can vary per cell), but *by convention* it is named ''**/.:/sec**'', and it will be denoted as such in the remainder of this section. (As will be seen, it would even be technically possible to split the name *sec* into multiple different names, say *sec1*, *sec2*, and so on, according to its usage by different servers, say RS, KDS, and so on — however, that would lead to unnecessary complication, and it is not supported by this specification.)

By administrative action at cell-creation time, the following 3 interface specifications (interface UUID and version number) are added as elements to the **/.:/cell-profile** node, all of them pointing to the **/.:/sec** group node:

- The interface UUID and version number of the **rs_bind** interface, as specified in Section 11.3.2 on page 364. This interface specification is to be used by clients when they bind to the RS server.
- The interface UUID and version number of the **krb5rpc** interface, as specified in Section 4.1.1 on page 161. This interface specification is to be used by clients when they bind to the KDS server.
- The interface UUID and version number of the **rpriv** interface, as specified in Section 5.1.1 on page 263. This interface specification is to be used by clients when they bind to the PS server.

During their initialisations, the RS, KDS and PS servers in a cell create CDS RPC server nodes for themselves (the names of these are unspecified in DCE, though in typical implementations the RS, KDS and PS servers all reside in a single RPC server, and they have names like **/.:/subsys/dce/sec/master** for the master instance and **/.:/subsys/dce/sec/rep_***n* for replicas), in which they register their (per-cell) RPC object UUIDs (all replicas share this UUID), their interface handles (in the notation of the ''**rpc data types**'' reference page of the referenced X/Open DCE RPC Specification, these interface handles are **rs_bind_v2_0_s_ifspec**, **krb5rpc_v1_0_s_ifspec** and **rpriv_v1_0_s_ifspec**, respectively), and their RPC binding information. Then, they add themselves to the **/.:/sec** CDS RPC group node, using the CDS RPC name of the server node they just created and registered themselves in.

To bind to security services (RS, KDS or PS), a client uses the *rpc_ns_binding_import_**( ) routines to query the **/.:/cell-profile** node, using the appropriate interface handle (**rs_bind_v2_0_c_ifspec**, **krb5rpc_v1_0_c_ifspec** or **rpriv_v1_0_c_ifspec**, respectively), and a null object UUID.

Concerning local hosts' SCDs, see Section 1.15.2 on page 77. As discussed there, in order to support a secure *sec_login_certify_identity*( ), the SCD's RPC binding information (as well as the transport underlying the RPC) must be part of the local host's TCB.

1.18.1.2 Binding to ACL Servers

The model (‘‘junctions’’) that clients use for binding to ACL servers (servers exporting the **rdacl** RPC interface) has already been discussed in the context of ACL Editors (clients of the **rdacl** interface), in Section 1.11 on page 55. That discussion is best treated in the context of ACL Editors, and need not be repeated here.

In the particular case where the ACL server in question is the RS itself, the ACL binding model of Section 1.11 on page 55 together with the RS binding model of Section 1.18.1.1 on page 86, leads to the naming conventions (of the form **/.../***cell-name***/sec/principal/***P-name*, and so on) mentioned in Section 1.18 on page 84. (Note that it would be technically possible to use a junction name that is different from the group name **/.:/sec** (say) — however, that would lead to unnecessary complication, and it is not supported by this specification.)

1.19 DCE Delegation Model

In a simple distributed environment, the DCE 1.0 facilities were sufficient to permit secure operations between two principals, typically described as a client and a server — the initiator and the target of the operation. In this simple environment, the target of the operation can reasonably make authorisation decisions based upon the identity of the initiator. This model, while adequate for simple client/server distributed computing, does not address the requirements of more complex environments, for example, those based upon distributed objects, where the target (server) needs to perform operations on other components on behalf of the initiator. These operations on other components must be invoked in a secure manner by intermediaries that may not be known to the initiating principal (cleint).

The distributed environment has the characteristic that intermediate objects, which are used to hide the details of complex system operations, pervient the target (service as distinguished from object) from securely determining the identity of the initiator of the operation on the object. All requests arriving at the target service appear to be the result or action of the intermediary rather than the true initiator of the operation.

The inability to determine the true initiator of a request presents distributed systems with unsatisfactory choices, such as:

- *implementing the service as a local one running with the identity of the initiator.*

  In this instance, the service loses the benefits of distribution.

- *running as a privileged principal that has full access to all services it abstracts (via distributed objects).*

  In this instance, the service may retain distribution, but then it would be required to implement an access control function for each of the services (represented by distributed objects).

- *requiring use of an alternate set of target service interfaces that permit the authorised principal to specify the principal on whose behalf the operation is (really) being performed.*

  This solution comes at the expense of redundant interfaces that expose the details of privilege attributes to the application protocol (thus providing opportunities for malicious attacks).

- *implementing the service in a manner that impersonates the (true) initiator, but where the initiator transmits its credentials (tickets and keys) to the intermediary (impersonator) so as to permit the intermediary to be indistinguishable from the initiator.*

  Unfortunately, this approach, while distributed (and in DCE, having a high degree of location transparency), exposes the client to a very great risk of attack — for instance, being compromised by a Trojan horse.

The solution to this problem involves the concept of *delegation*. Delegation permits an intermediary to operate upon other (non-target) objects on behalf of the initiator in a manner that both reflects the initiator of the operation and is distinguishable from the (true) initiator of the request. In its simplest form, *delegation* permits a subject (as described in Section 1.1.3 on page 6, which herein is called principal *A* to invoke an oepration upon (an object associated with) principal *C* through (an object associated with) principal *B* (herein called an intermediary) in such a manner that reflects the true initiator of the operation (*A*) but which can also be distinguished from the same operation invoked directly by A (or *B*, for that matter,) on *C*.

1.19.1 Overview of Delegation Model

The DCE 1.1 delegation model being described here consists of three components:

- *An intermediary is permitted to operate upon objects in a manner that reflects the initiator's identity as well as its own.*

  A target (server) receiving such a chained request would see the privilege attributes of each participant in the chain (without their being exposed to the application).

- *The authorisation model is extended so as to permit the target (server or servers) to make use of the distinction between initiators and intermediaries.*

  Thus, owners of a rewource may grant rights to principals acting as intermediaries on behalf of authorised initiators without granting rights for these principals to act (upon the objects representing these or other services) on their own behalf.

- *Clients (initiators) performing operations (on objects) are permitted to place restrictions upon the use of their identity in a chain of requests for services (upon abstract objects representing the totality of the services necessary to perform the original client reqest) upon a target (service).*

  A client may choose to entirely disallow delegation or to limit which principals (intermediaries) may use the client's identity in a delegated manner.

  - *Additionally, the nature of a caller's identity (whether initiator, intermediary or target) is extended to include* **arbitrary** *(in the sense of being optional) attributes stored in the registry services (RS) of the user.*

    These attributes require extensions to the PAC, and may be used by applications when making authorisation decisions. Constrained (delegate) restrictions and also restrictions upon the target may be imposed by the client, and also by intermediaries, with these attributes.

In essence, the DCE 1.1 Delegation Model consists of extensions to the following DCE 1.0 items:

- The DCE 1.0 Privilege Attribute Certificate (PAC). (See Section 5.2.13 on page 283 for details.)
- The Privilege Server (PS) interface for generating PACs.

  (A new interface, *sec_cred*, is provided for abstracting the contents of the EPAC. Details can be found in Chapter 20.)

- Privilege Server evaluation of inter-cell extended attributes.
- ACL entry types (ACLE's). See Section 7.1.2 on page 312 and Section 7.1.5 on page 313 for details.

  and finally,

- The ACL evaluation algorithm, more commonly termed the Access Determination Algorithm, described initially in Section 1.9.1 on page 48 and also in Chapter 8, in greater detail.

It also consists of a new set of user interfaces to support this delegation model. These are primarily in support of the new ACLE's which also require a set of new library interfaces as well. All this will be discussed in greater detail in the following sections.

1.20 Components of Delegation Model

This section discusses the major components of the DCE 1.1 Delegation Model introduced in the preceeding section, in greater detail. Extensions to the *sec_login* interface (listed in Section 1.15 on page 71) provide the ability for a client to enable and disable delegation and also provide a function for servers to become delegates. A new interface, *sec_cred*, is provided as an abstraction of a set of accessor functions for access to privilege attributes.

In addition, the PAC has been extended to encompass the privilege attributes neccessary to accomplish delegation, and the DCE access control mechanism has been amended to check access of the composed principals (intermediaries) involved, as well. Part of the delegation extensions to the PAC include delegation controls as well as delegate and target restrictions. Additional application-specified required and optional restrictions may be present.

Finally, a new set of remote interfaces is provided to aid in authentication and authorisation. These interfaces are invoked (indirectly) in support of delegation, and are not intended for direct use by the application.

1.20.1 The Extended PAC (EPAC)

The trust mechanisms for authentication employed in DCE 1.0 by the Privilege Server when constructing a PAC are built upon for delegation by nesting delegated privilege attributes in an extended PAC (EPAC).

The extended PAC contains the identity and group membership information present in a DCE 1.0 format PAC. For compatibility with DCE 1.0.x, a PAC may be imbedded in the authorisation data (A_D) field of (the version 5, which is used in DCE 1.1) Kerberos tickets. This will be shown in Figure 1-12 on page 92 and is dependent upon whether compatibility is required by the DCE application.

In addition to the information contained in the PAC, the EPAC contains the following:

- optional delegation controls

  These controls provide a mechanism for specifying the form of delegation — permitting one to select between simple delegation and traced delegation (simple delegation is also known as impersonation). These concepts are explained elsewhere in this document.

  In addition, DCE 1.1 delegation provides the ability for an initiator or intermediary to impose restrictions on the identities of delegates in a chain and the set of targets to which an identity can be presented.

- optional and required restrictions

  These are provided as classes of extended registry attributes for use by applications that have specific authorisation requirements. They may be set by initiators and intermediaries.

- extended attributes

  These include the privilege attributes of the initiator (of the chain of operations) as well as the privilege attributes for each intermediary involved in the chain. Thus, the notion of identity is extended to include chained identities. Adding extended attributes to the EPAC permits DCE to use existing *legacy* authorisation models.

Note that by including attributes within an EPAC, the attributes may be transmitted securely (and automatically) along with a principal's identity information. This content change also requires a change in the manner in which extended PACs are handled in DCE.

Figure 1-11 on page 91 shows both an EPAC, and part of the Authorisation Data field of a PTGT. Note that since the extensions in an EPAC (as compared to a PAC) can contain an arbitrary

amount of application-selected data (which may be limited by the practicalities imposed by RPC authorisation information size restrictions). For this reason (and also other reasons, such as performance) EPACs will not be placed in PTGTs. Instead, the authorisation data field will contain a *seal* (cryptographic checksum) of the EPAC.

The A_D field may optionally contain a DCE 1.0 format PAC (as shown in Figure 1-12) if compatibility with DCE 1.0 is required. When delegation is enabled, the initiator may specify this parameter.

Note: The cryptographic checksum (*seal* of the EPAC) is performed by the Privilege Server, PS_Z, and placed within the EPAC. This provides the same integrity guarantees previously provided for DCE 1.0 PACs, where the entire PAC was included in the A_D field, which is encrypted by the privilege server.

The following figure depicts the *seal* of the EPAC.

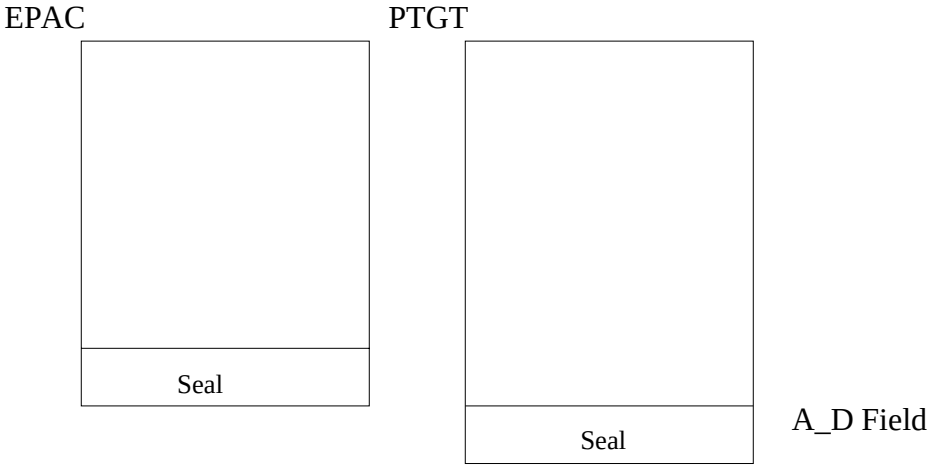


Figure 1-11 EPAC Seal within EPAC and A_D Field of PTGT

The following figure shows the optional Version 1.0 PAC within the PTGT.

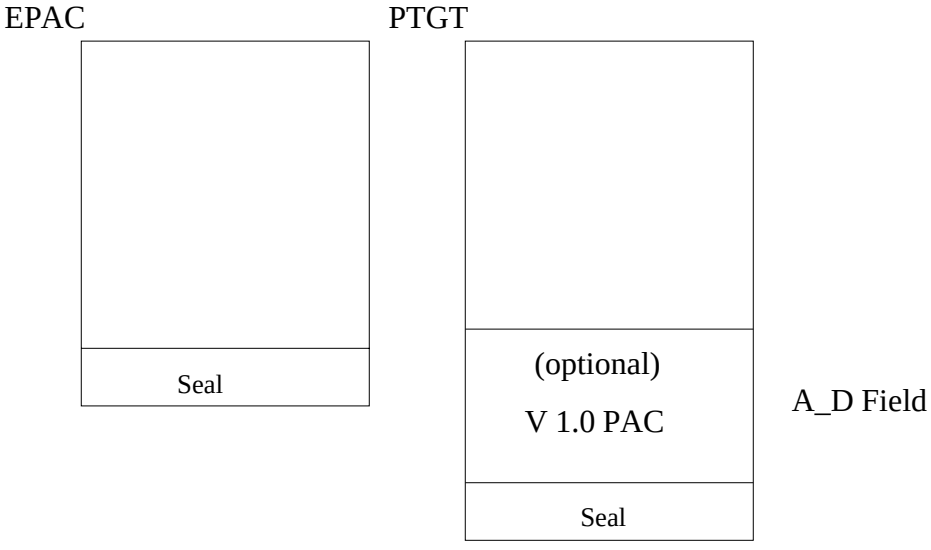


Figure 1-12 EPAC Seal (and Optional Version 1.0 PAC) within A_D Field of PTGT

Information on the EPAC seal can be found in Chapter 5, in the section ‘‘Data Types’’. In particular, see Section 5.2.13 on page 283 and Section 5.2.13.8 on page 285.

1.20.1.1 Linking EPAC Sets to Tickets

When traced delegation is (enabled and) in use, there may be a set of EPACs to transmit. This set of EPACs contains one EPAC for the initiator and one EPAC for each delegate involved in an operation. In this case, a single seal in the PTGT is no longer sufficient to guarantee the validity of an EPAC. The individual integrity of each EPAC as well as the specific order of the EPACs must be guaranteed. To obtain this integrity, the A_D portion of a PTGT carries the seal of the ordered list of EPAC seals.

1.20.2 Transmitting and Receiving EPACs

EPACs are transmitted during the authentication phase of a request. During this phase, A PTGT is sent from the client to the server. In DCE 1.0, the Authorization Data field of the PTGT contained a Version 1.0 PAC. As of DCE 1.1, the Authorization Data field of the PTGT contains the *seal* of the EPAC, rather than the EPAC, for performance reasons (since it can contain arbitrary amounts of data). So, for DCE 1.1, the KRB_AP_REQ that is passed between the client and server will have the EPAC (or EPAC chain) appended directly to the Kerberos KRB_AP_REQ, whereas for DCE 1.0, the KRB_AP_REQ did not have this appendage. To summarize, for DCE 1.0, the security message that passes between the client and server consisted solely of a KRB_AP_REQ, which contained a PAC. As of DCE 1.1, the security message consists of a KRB_AP_REQ which contains a seal of an EPAC, followed by an EPAC chain. This is shown by the following figure:

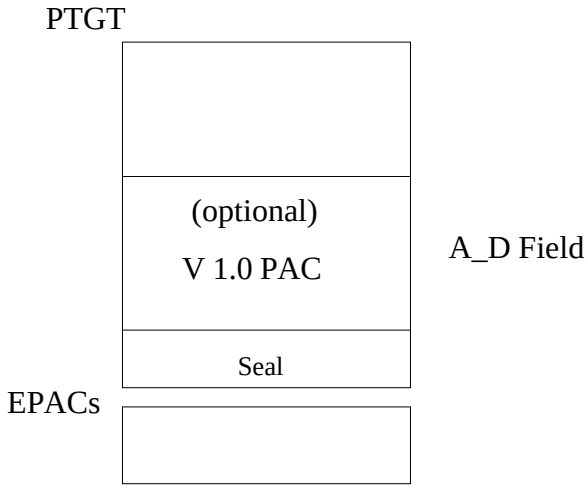


Figure 1-13 Transmitting EPACs with Service Tickets

1.20.3 Extended Privilege Attribute Facility

The *Extended Privilege Attribute Facility* permits clients and servers to invoke secure operations by way of one or more intermediate servers. Prior to DCE 1.1, simple client/server operations involved two principals:

- the initiator of the operation
- the target of the operation.

  The target (server) in this scenario makes decisions based upon the identity of the initiator.

However, in distributed object oriented environments, frequently server principals need to perform operations on behalf of a client principal. In these instances, authorisation decisions based simply on the identity of the initiator, since the initiator of the operation may not be tie principal that requests the operation. This is particularly true in the case of delegation.

To handle these situations, the Extended Privilege Attribute Facility allows principals to operate on objects on behalf of (*as delegates of*) an initiating principal. The collection of the delegation initiator and the intermediaries is referred to as a *delegation chain*. Using this facility (including related **sec_login** calls), an application may be written that allows client principal *A* to invoke an operation on server principal *C* by way of server principal *B*. The Authorisation Service (AS) will know the true initiator of the operation (principal *A*) and can distinguish the delegated operation from the same operation invoked directly by principal *A*.

The Extended Privilege Attribute Facility consists of:

- Login functions of the form **sec_login** that are used to establish delegation chains and other related delegation functions, listed in Section 1.15.1 on page 75.
- Security credential functions of the form **sec_cred** listed in the next section.

The **sec_cred** and Login **sec_login_cred** functions extract privilege attribute information associated with an opaque handle to an authenticated identity. The **sec-cred** functions are used by servers that have been called by a client with atuenticate credentials. They provide servers with the ability to retrieve information from the caller's EPAC, since the EPAC is not available to the application as PACs were prior in DCE 1.0. The **sec_login_cred** functions are used by clients that are part of a delegation chain, and also provide the ability to return the privilege attributes of delegates in or initiators of, a delegation chain.

1.20.4 EPAC Accessor Function API

The EPAC Accessor Functions consist of an API for retrieval of Privilege Attribute information from Extended PACs. (They all start with the identifier, *sec_cred*.)

This API is provided as an abstraction of the contents of an EPAC. It insulates applications from the format of a PAC or EPAC, thereby relieving the necessity of dealing directly with PAC or EPAC formats.

There are 18 functions which comprise this interface. Their names and brief descriptions of their functions follow. See Section 5.2.14 on page 288 for further information:

- *sec_cred_is_authenticated*( )

  Returns true if the caller privilege attrbutes are authenticated; false otherwise.

- *sec_cred_get_client_princ_name*( )

  This function is used to extract the principal name of a server's RPC client, if the authorisation service can provide it. If not, a **sec_cred_s_authz_cannot_comply** status is returned.

- *sec_cred_get_initiator*( )

  This function is used to extract the initiator's privilege attributes from the RPC runtime.

- *sec_cred_get_delegate*( )

  This function is used to iterate through and extract the privilege attributes of the delegates involved in this operation from the RPC runtime.

- *sec_cred_get_authz_session_info*( )

  This function is used to retrieve session-specific information that represents the authenticated client's credentials. The session information is meant to aid applicaation servers in the construction of identity-based caches.

- *sec_cred_get_v1_pac*( )

  This function is used to extract a version 1 DCE PAC from a privilege attribute handle.

- *sec_cred_get_pa_data*( )

  This function is used to extract identity information from a privilege attribute handle.

- *sec_cred_get_extended_attrs*( )

  This function is used to extract extended attributes from a privilege attribute handle.

- *sec_cred_initialize_attr_cursor*( )

  This function is used to initialise a **sec_cred_attr_cursor_t** for use in calls to the iterative routine *sec_cred_get_extended_attrs*( ).

- *sec_cred_initialize_cursor*( )

  This function is used to initialise a **sec_cred_cursor_t** for use in calls to the iterative routine *sec_cred_get_delegate*( ).

- *sec_cred_get_delegation_type*( )

  This function is used to extract the allowed delegation type from the privilege attribute handle.

- *sec_cred_get_tgt_restrictions*( )

  This function is used to extract target restrictions from a privilege attribute handle.

- *sec_cred_get_deleg_restrictions*( )

  This function is used to extract delegate restrictions from a privilege attribute handle.

- *sec_cred_get_opt_restrictions*( )

  This function is used to extract optional restrictions from a privilege attribute handle.

- *sec_cred_get_req_restrictions*( )

  This function is used to extract required restrictions from a privilege attribute handle.

- *sec_cred_free_cursor*( )

  Free the local resources assocated with a delegate cursor.

- *sec_cred_free_attr_cursor*( )

  Free the local resources assocated with an attribute cursor.

- *sec_cred_free_pa_handle*( )

  Free the local resources assocated with a privilege attribute handle returned by *sec_cred_get_initiator*( ) or *sec_cred_get_delegate*( ).

1.20.5 RPC Authorisation Extension

Due to the need for abstraction in handling DCE 1.1 credentials (the preceeding section listed the functions available for this purpose), a new function, *rpc_binding_inq_auth_caller*( ) has been added for DCE 1.1. It replaces the *rpc_binding_inq_auth_client*( ) which is still available for use with existing (pre-DCE 1.1) application servers. This is also discussed in Section 1.17 on page 82, ''Integration with RPC Services''. This new function is used in conjunction with the *sec_cred_**( ) functions — in particular, for obtaining the client's credentials.

1.20.6 Enabling and Disabling Delegation

A set of extensions to the *sec_login_**( ) functions provides the ability to enable delegation and also to select the type of delegation desired. They also provide the capability for servers to become intermediaries in support of an operation. These functions are listed in Section 1.15 on page 71 and described in Chapter 19, ''Login API''.

In addition, the extensions to *sec_login_**( ) provide facilities for controlling the use of delelgation, such as the ability to restrict the allowable set of delegate and target principals. These facilities are described in the next section, Section 1.20.7.

The *sec_login_disable_delegation*( ) function disables delegation for a specified login context. Use of this function for that context prevents any further delegation.

1.20.7 Delegation Controls

The DCE 1.0 model of privelege attributes permits the data in a PAC to be extended. With the extensions to the PAC, the following mechanisms for controlling the form of delegation are provided. They permit an application to implement its own variety of models and policy for delegation.

Note: Delegation only occurs if a client has chosen to enable the projection of its identity to another entity in the distributed system in a manner that permits that entity to operate on behalf of the initiator.

A client process may enable delegation by annoting a login context with the allowable set of delegates and target principals. Once so annotated, operations using the context will transmit the appropriate data to the server servicing the operation. In addition, a client desiring to delegate access (to an object) obtains a delegation token from the Privilege Server. The delegation token is ''signed'' by the Privilege Server, PS_Z, to guarantee that the privilege attributes are not modified by either the client or the server involved in the delegation. The client identifies the desired delegates and the eventual targets of the delegation when making the call to the Privilege Server. The Privilege Server, in turn, seals that information along with the privilege attributes of the client, which are transmitted in the request (from the PAC or EPAC), into the delegation token. Delegation Tokens are discussed in greater detail in Section 1.20.7.2 on page 97. The data type is described in Section 5.2.15 on page 289.

The following summarizes the delegation controls provided in DCE 1.1.

- Impersonation or Delegation

  The application can control the form of delegation by selecting between simple (called impersonation) or traced delegation. This form is enabled by the client application calling

sec_login_become_initiator( ) with the *delegation_type_permitted* argument specifying either delegation or impersonation. (See Section 5.2.13.6 on page 285 for specific encodings.)

In addition, the form of delegation can be selected by an intermediary when calling either *sec_login_become_delegate*( ) or *sec_login_become_impersonator*( ).

While two forms (delegation or impersonation) of delegation may be specified, an intermediary is not permitted to use a form of delegation that was not enabled by the initiator. Impersonation permits an intermediate service to ''*impersonate*'' the initiator — where the initiator transmits to the intermediary service the credentials (tickets and keys) necessary to be indistinguishable from the initiator. It is often useful for implementing remote system login utilities, for instance, and while this increases the risk to the client of being compromised, by a Trojan horse application, its usefulness precludes its not being provided. If an intermediate service attempts to select a delegation type that is not enabled by the client, an error will be returned. (Appendix B lists the errors that may be returned in DCE 1.1).

- Delegate and Target Restrictions

  Delegate and Target Restrictions are placed by the client and enforced by the Privilege Server (for Delegate Restrictions) when producing delegation credentials, or by the Authentication Services (for Target Restrictions) when determining if the target server is permitted to see the client's identity. The data types for Delegate and Target Restrictions are specified in Section 5.2.13.2 on page 284. They are specified by initiators and intermediaries as an argument to one of the functions *sec_login_become_initiator*( ), *sec_login_become_delegate*( ) or *sec_login_become_impersonator*( ).

- Required and Optional Restrictions

  Required restrictions are added to (inserted into) the EPAC by the Privilege Server to restrict the activities that a server can perform. An example can be time-of-day restrictions, or interface selections, or target object restrictions. They are enforced by the Authentication Services for the target server. Required restrictions *must be* understood by the receiving application server; otherwise they *must* deny access.

  Optional restrictions are processed like required restrictions except that applications that are unable to decode a given optional restriction are free to ignore them. Thus, their effect may also be limited to a given (sub)set of applications.

  The data type defining Optional and Required restrictions is found in Section 5.2.13.1 on page 283.

  Required and Optional Restrictions are specified by initiators and intermediaries as an argument to one of the functions *sec_login_become_initiator*( ), *sec_login_become_delegate*( ) or *sec_login_become_impersonator*( ).

1.20.7.1 Anonymous Identity

When one of the Delegate Restrictions or Target Restrictions placed by a client prevents an identity from being delegated by an intermediary or seen by a target server, that identity will be protected by being replaced in the delegation chain by an identity containing well known *anonymous* privilege attributes — in other words, by an EPAC containing id's belonging to a well known anonymous principal and group.

The data type representing the Anonymous Identity can be found in Section 5.2.14.1 on page 288.

Notes:

1. All implementations must implement these id's as specified in the just referenced section to ensure interoperability.
2. Currently (in DCE 1.1 and newer versions) the Authorisation Service only has access to the server's principal name. It does not have access to any other privilege attributes. Thus, Target Restrictions are limited in that if *ANY* are specified in a given EPAC, all target servers will only see the Anonymous Identity (consisting of the Anonymous Group UUID and the Anonymous Principal UUID) for that EPAC.
3. For Delegate Restrictions, if the intermediary does not satisfy the restrictions set by a particular identity, the Privilege Server will replace that identity's EPAC with one indicating that an anonymous participant (Anonymous Identity - consisting of the Anonymous Group UUID and the Anonymous Principal UUID) was involved.

1.20.7.2 Delegation Tokens

A client that desires to delegate access to a server obtains a delegation token from the Privilege Server, PS_Z, which is ''signed'' by the PS to quarantee that privilege attributes are not modified by either the client or any server involved in the delegation.

When a new PTGT is generated as a result of calling either *ps_request_ptgt*( ), *ps_request_become_delegate*( ) or *ps_request_become_impersonator*( ) (described in Chapter 5), the Privilege Server inserts a deletation token into the Authorisation Data field of the PTGT. (These functions are described in Chapter 5.

This delegation token will be passed along with the Authorisation Data, in any authenticated RPC calls made using the new login context. Servers receiving those calls will need to use the delegation token in a *ps_request_**( ) to become a delegate or impersonator. Upon receiving such a request, the PS will verify that the delegation token matches the delegation chain (or single identity passed in) to ensure that the proper steps were taken to enable delegation and that none of the data has been tampered with (since that time).

The delegation token is an MD5 checksum over the EPAC, encrypted in the key of the Privilege Server — the seal being of the type **sec_id_seal_type_md5_des**. Only the PS can generate such a seal (it's signature), so any tampering of the data sealed by a delegation token would break that seal, and the Privilege Server would then reject any requests to use that data.

Note: The delegation token is passed in the Authorization Data field along with the standard seal over the EPAC, which has not been previously encrypted. The seal of the EPAC is used to verify the integrity of the EPAC data for authenticated RPC calls.

1.20.8 Remote Interfaces

A set of new interfaces, described in Section 5.1.1 on page 263 is also provided to aid in authorisation. They permit a PTGT to be obtained from the Privilege Server, PS_Z, by intermediary callers after having the appropriate verifications made (depending upon the circumstances). The data types associated with these functions are also described in Chapter 5, in the subsections under ''Data Types''.

1.20.9 Extensions to ACLs

DCE 1.0 access control lists contain entries that identify the access rights to be granted to principals bearing certain privilege attributes. As of DCE 1.1, clients and servers are able to invoke secure operations through one or more intermediaries. When delegation is activated, a target server will receive an extended PAC that contains the privilege attributes for the initiator of the chain of operations as well as the privilege attributes for each intermediary involved in the chain. The Common Access Determination Algorithm has been modified to verify that the privilege attributes for each principal involved in the operation, as specified in the EPAC, have been granted the necessary rights to perform the operation.

More specifically, the standard ACL entries are extended with a set of entries that only apply to principals acting as intermediaries. These extended entries are called *delegate* entries and permit intermediaries to be listed on the ACL without granting those intermediaries the ability to operate on the target object directly. The new *delegate* ACL entry types used to provide access to an identity acting as an intermediary only are:

user_obj_deleg	Identity that owns object
user_deleg	Specific principal, identified by cell-relative principal name
for_user_deleg	Specific principal, identified by global principal name
group_obj_deleg	Identity of group that is listed as owner of object
group_deleg	Specific group, identified by cell-relative group name
for_group_deleg	Specific group, identified by global group name
other_obj_deleg	Any principal in the local cell
for_other_deleg	Any principal in the specified cell
any_other_deleg	Any principal in any cell

For specific information on these types, the **sec_acl_entry_type_t** data type describing the delegate ACLEs can be found in Section 7.1.2 on page 312.

The extended authorisation algorithm is as follows:

```
 (I) Check Initiator:
     Apply standard algorithm
     IF access mode is denied THEN
   Deny Access & Terminate Algorithm
 ENDIF

(II) Check Each Intermediary:
 FOR EACH Extended PAC DO
   Apply standard algorithm (allow delegate entries)
   IF access mode is denied THEN
     Deny Access & Terminate Algorithm
   ENDIF
 END

(III) Grant Access
```

Figure 1-14 Extended Delegation Access Control Algorithm

The preceeding figure presents a very high-level overview of the algorithm. Section 1.8 on page 40 and Section 1.9.1 on page 48 provide additional information on the extensions to the ACLEs and the authorisation steps. The details of the extended delegation access control algorithm can be found in Chapter 8, in Section 8.2 on page 321.

1.20.10 User Interfaces

The user interfaces to support this delegation model are limited to a new set of DCE ACL entry types that are in addition to the ones existing for DCE 1.0. The addition of these types does not cause any change in behavior for existing ACL Editors (unless they explicitely need to use one or more of the new entry types to provide authorisation control for delegation purposes). For the new entries, the key and permission types are defined exactly as they would for other existing DCE ACLEs. (As noted in the previous section, the new extensions to the ACL entry types can be seen in Section 1.8 on page 40. The authorisation steps can be seen in Section 1.9.1 on page 48 which lists the steps for both existing (legacy) ACL management, and also those for delegation.)

No wire protocol changes are necessary to support these new ACL entry types, since they are simply additional values of the existing **sec_acl_entry_type_t** data type, which can be seen in Section 7.1.2 on page 312.

1.21 Extended Registry Attribute Facility

The DCE 1.0 user registry facility uses a static schema. The *Extended Registry Attribute Facility* is an attribute facility that employs a dynamic schema.

The Registry (for DCE 1.0) is a repository for principal, group, organization and account data. It stores the network privilege attributes used by the DCE as well as account data used by local operating systems. The local account atributes are appropriate only for UNIX operating systems.

The Extended Registry Attribute (ERA) Facility provides a mechanism for extending the Registry schema to include data (attributes) for operating systems other than UNIX. This data includes attribute schema and attribute instances. These are stored in the Registry and propagated from the master security server to replicas, just like the existing Registry data. The attribute schema has a cell-wide influence, but not an inter-cell influence. The Extended Registry Attributes (ERAs) are manipulated by a set of operations for creating and maintaining the attribute schema and attribute instances.

These operations provide the ability to define attribute types and attach attribute instances to registry objects. Registry objects are nodes in the Registry (database) to which access is controlled by an ACL Manager type. The Registry Objects are:

- Principal
- Group
- Organisation
- Policy
- Directory
- Replist
- Attr_schema

These registry objects and their accompanying ACL Manager type are explained elsewhere in this document. The ACL Manager types can be seen in Section 11.1 on page 358.

The Extended Registry Attribute (ERA) Facility also provides a trigger interface that servers use to integrate their attribute services with the ERA services.

1.21.1 Attribute Schema

An attribute schema is a catalog of attribute types known to a system (in the sense of ‘‘operating system’’). Each entry in the schema defines the format and function of an attribute type. There is an attribute schema identified by the architectural name ‘‘xattrschema’’ under the security junction point (usually ‘‘/.:/sec’’) in the CDS namespace. The schema may be dynamcially updated to create or destroy schema entries. Access to the (attribute) schema is controlled by an ACL on the schema object. (As previously mentioned, the schema is propagated from the master security server to replicas like other Registry data.) Since the schema is local to a cell, it defines the types that can be used within the cell, but not outside the cell (unless the type is also defined in another cell). See Section 1.21.2 on page 101 for information on the ACL object permissions.

Note: When an attribute is used in an authorisation decision, it is likely that the value of that attribute for a particular principal must not be the same as the value of that attribute for any other principal. Therefore, when attributes are used for authorisation decision, they should be unique. The ERA schema entries have a ‘‘unique’’ boolean as one of the characteristics of the attribute type. In order for an

attribute to be unique, this boolean must be set to the value TRUE (1).

1.21.2 Access Control for the xattrschema Object

The Registry ACL Manager has the **xattrschema acl_mgr_type** identified by the UUID *755cd9ce-ded3-11cc-8d0a-08009353559* which supports the following set of permission bits:

Permission Bit	Name	Description
d	*delete*	Delete an entry from the schema
i	*insert*	Create a new entry in the schema
m	*management*	Modify a schema entry
r	*read*	View the contents of the schema
c	*control*	Modify the ACL on the schema

Table 1-1 Extended Attribute Schema ACL Manager Permission Bits

1.21.3 Schema Entries

A schema entry defines the characteristics of an attribute type known to the system. Once a schema entry has been created for an attribute type, instances of that attribute type can be created on objects that are dominated by the schema.

The primary identifier of an attribute type is its unique identifier (UUID). The schema entry also contains a unique string name that can be used as a secondary key. Although schema entries may be created and deleted dynamically, only certain fields in the schema entry may be modified after creation. The prohibition of certain fields from being modified after creation avoids inconsistent Registry data and access control. The **sec_attr_schema_entry_t** data type defines a schema entry. It is shown in Section 11.9.1.6 on page 431.

The following fields in a schema entry (sometimes also called an attribute type) can be modified:

- attr_name
- acl_mgr_set
- reserved
- intercell_action
- trig_binding
- comment

Additional ACL managers may be added to the *acl_mgr_set* field after creation of the entry, but none may be deleted or changed. If a dramatic change, such as a different *attr_encoding* or **acl_mgr_type** (in the *acl_mgr_set*), must be made to an schema entry, the schema entry must first be deleted (which will force deletion of all attributes of that type), then re-created.

The *acl_mgr_set* lists the ACL Manager types that support the object types on which attributes of a type can be created. For instance, if an attribute is to be attached to principal objects, the **principal acl_mgr_type** must be specified. Similarly, if the attribute is additionally intended for use on the policy object, the **policy acl_mgr_type** must be specified. For each **acl_mgr_type**, the permissions required for attribute operations on the corresponding object type are also specified. The data type structure for this is the **sec_attr_acl_mgr_info_t**, which is defined in Section 11.9.1.1 on page 428.

1.21.3.1 Attribute Type Flags

Some of the fields in a schema entry are known as flags. Their settings have special meanings to the entry. If for instance, the **reserved** flag is set, the entry cannot be deleted. If a principal with the required permissions changes this flag (field), for instance, the cell administrator, to FALSE (0), then authorized principals can delete the schema entry.

If the **use_defaults** flag is TRUE (1), then when searching for a default attribute value, the DCE 1.1 RS ERA function first examines the organisation (specified in the principal's account, if any) for an attribute instance of the requested type. It secondly inspects the policy object for an attribute instance of the same type (When a *rs_attr_get_effective*( ) query is made on a group or organisation object, only the policy object is inspected for an attribute instance to use as a default value).

The flags are defined in the **sec_attr_sch_entry_flags_t** data type in Section 11.9.1.2 on page 428.

1.21.4 The use_defaults Algorithm

The algorithm used by *rs_attr_get_effective*( ) (which is not useful for queries on the policy object itself) for a single-valued attribute type is:

- Query the named object for an attribute instance of the requested type. If an instance exists, return it. If an instance is not found, proceed.
- If the object is one of the following,
  - a principal without an account,
  - a group, or
  - an organization

  proceed. Otherwise (Else), if the object is a principal with an account, query the organisation specified in the principal's account for an attribute instance of the requested type. If an instance exists, return it. If an instance is not found, proceed.
- Query the policy object for an attribute of the requested type. If an instance exists, return it. Otherwise, return ''attribute_not_found''.

The search algorithm for a *default* attribute differs slightly for a **multi_valued** (if this flag is TRUE (1)) attribute type. When an attribute type is defined with both the **use_defaults** and the **multi_valued** flags set, then the same progression from principal to organisation to policy is made; however, all attribute instances are collected and returned after the check on the policy object. Thus, if an attribute instance of the requested type exists on both the principal object and the policy object, then *rs_attr_get_effective*( ) will return two attribute instances for that query.

The **use_defaults** behavior depends upon support for the given attribute type on the organisation and policy object types. For instance, if the schema entry for a given attribute type does not include the organisation **acl_mgr_type** in its *acl_mgr_set*, then the query on organisation will be skipped. Likewise, if the attribute type is not supported on the policy object, the query on the policy object will be skipped.

1.21.5 The intercell_action Algorithm

The *intercell_action* field of the schema entry specifies the action that should be taken by the Privilege Server when reading attributes from a foreign cell. This field can contain one of three values:

sec_attr_intercell_act_accept Accept the foreign attribute instance.

sec_attr_intercell_act_reject Reject the foreign attribute instance.

sec_attr_intercell_act_evaluate Call a remote trigger server to determine how the attribute instance should be handled.

When the Privilege Server is generating a PTGT for a foreign principal, it:

- retrieves the list of attributes from the foreign principal's EPAC,

  Note: The attribute instances may be attached to the principal object itself or attached to the group or organisation object associated with the principal object.

- passes the list to the Privilege Server through the * *prv_attr_check_intercell_attrs*( ) call which retains, discards, or maps the attributes in the list, producing an output list of attributes, and
- includes the output list of acceptable attributes in the EPAC it generates for the object, for the PTGT for the foreign principal.

The Privilege Server then checks the local attribute schema for the attribute types with UUIDs that match the UUIDs of the attribute instances from the foreign cell that are contained in the foreign principal's EPAC. At this point, the Privilege Server does one of two things, as follows:

1. If the Privilege Server cannot find a matching attribute type in the local attribute schema, it checks the *unknown_intercell_action* attribute on the policy object. If this attribute is set to:

   - **sec_attr_intercell_act_accept**:

     The foreign attribute instance is retained and included in the EPAC generated for the object by the Privilege Server.

   - **sec_attr_intercell_act_reject**:

     The foreign attribute is discarded.

   Note: The **unknown_intercell_action** attribute must be created by the system administrator and attaached to the policy object. The attribute type, which takes the same values as the **intercell_action** flag (field), is defined in Section 1.21.12.1 on page 108.

2. If the Privilege Server finds a matching attribute type in the local attribute schema, it retrieves the attribute. The action it now takes depends upon the setting of the attribute type's intercell action field and unique flag, as follows:

   - If the intercell action field (*intercell_action*) is set to the value **sec_attr_intercell_act_accept** and:

     - The *unique* flag (field) is set to FALSE (0), the Privilege Server includes the foreign attribute instance in the principal's EPAC
     - The *unique* flag (field) is set to TRUE (1), the Privilege Server includes the foreign attribute instance in the principal's EPAC *only if* the attribute instance value is unique among all instances of the attribute type within the local cell.

       Note: If the *unique* attribute type flag is set to TRUE (1) and a query trigger exists for a given attribute type, the intercell action field cannot be set to

the value **sec_attr_intercell_act_accept** because, in this case, only the query trigger server can reasonably perform a uniqueness check.

- If the intercell action (*intercell_action*) field is set to the value **sec_attr_intercell_act_reject**, the Privilege Server unconditionally discards the foreign attribute instance.
- If the intercell action (*intercell_action*) field is set to the value **sec_attr_intercell_act_evaluate**, the Privilege Server makes a remote *sec_attr_trig_intercell_avail*( ) call to an attribute trigger using the binding information in the local attribute type schema entry. The remote attribute trigger determines whether to retain, discard, or map the attribute instance to another value or values. The Privilege Server includes the values returned by the attribute trigger in the ( ) sec_attr_trig_query call output array in the principal's EPAC. Section 11.9.1.4 on page 429 defines the types and values for schema entry types.

1.21.6 Attribute Scope

The scope field (defined in Section 11.9.1.6 on page 431) controls the objects to which an attribute type can be attached. If scope is defined, the attribute can be attached *only* to objects defined by the scope. If the scope for a given attribute is defined as some directory name, instances of that attribute type can be attached only to objects in that directory. If the scope is narrowed by fully specifying an object in that directory, for instance, /directory_name/another_directory_name, then the attribute can only be attached to the another_directory_name principal.

1.21.7 Attribute Encodings

Attribute encoding defines the legal encoding for instances of the attribute type. The encoding controls the format of the attribute instance values, such as whether the attribute value is an integer, string, a UUID, or vector of UUIDs that define an attribute set.

Attribute encodings are defined by the **sec_attr_encoding_t** data type which can be found in Section 11.8.1.16 on page 418.

1.21.8 Attribute Triggers

Some extended registry attributes require the support of an outside server either to verify input attribute values before storing them (in the Registry Store) or to supply output values when the data are stored in an external database. Such a server could be the connection to a legacy registry system or could be part of a new security application. The attribute trigger facility privides for automatic calls (triggers) to outside DCE servers for certain attribute operations. More specifically, triggers will automatically be invoked by the **sec_rgy_attr_client** agent whenever an *rs_attr_**( ) call indicates that a trigger is required for the operation to complete. The attribute trigger facility is discussed in the next section.

1.21.8.1 Attribute Trigger Facility

The attribute trigger facility consists of three components, specified as follows:

- The attribute schema trigger fields (*trig_types* and *trig_binding*), defined in Section 11.9.1.6 on page 431, which enable the association of a trigger and binding information with an attribute type. These fields are part of a ''standard'' schema entry that defines an attribute type.
- The **sec_attr_trig** interface, which defines the query and update trigger operations. The interface functions are defined in the *sec_rgy_attr_**( ) functions in Chapter 16. (For instance, for the query operation, the *sec_rgy_attr_**( ) function specifies the **sec_attr_t attr_value** field

is used to pass in optional information required by the attribute trigger query.)

- The trigger servers (implemented as DCE servers), independent of the DCE security server, that implement the trigger operations for the attribute types configured with their bindings.

The first two components are described in this specificaion, and are provided as part of the DCE 1.1 extended registry attribute support. Trigger servers are written by security application developers.

A trigger may be configured for any attribute type of any encoding type by filling in the *trig_types* and *trig_binding* fields of the schema entry.

1.21.8.2 Trigger Binding

When an attribute is created with the *sec_rgy_attr_update*( ) call (defined in *sec_rgy_attr_update*( ) on page 609), the association between the attribute type and an attribute trigger is defined by specifying the following:

- **Trigger Type** - (Schema entry *trig_types*)

  Defines the trigger as a query server, for query operations; or an update server, for update operations.

  Notes:

  1. If one of these flags (query or update) is set, then schema field *trig_binding* must also be set.
  2. For DCE 1.1, the ‘‘query’’ and ‘‘update’’ flags are mutually exclusive.

- **Trigger Binding** - (Schema entry *trig_binding*)

  Defines the server binding handle for the attribute trigger. The details of the trigger binding are defined by a number of data types. The *trig_binding* field contains an array of bindings, each of which may be a server directory entry name, a string binding, or an RPC protocol tower set. It also contains optional authentication and authorisation information for making authenticated RPC calls.

  It is recommended that the *trig_binding* specify the directory entry name for the trigger server and that actual address information be stored in the directory service (Prototype applications may want to specify a string binding or protocol tower for convenience.).

  When a server name is retrieved from the *trig_binding* field, the *rpc_ns_binding_import_begin*( ) function may be called specifying the server name, the *rpc_c_ns_syntax_dce*( ) entry name syntax, and the **sec_attr_trig** interface handle to establish a context for importing RPC binding handles from the name service database.

  When a string binding is retrieved, the *rpc_ns_binding_import_begin*( ) function may be used to generate an RPC binding handle. Once a binding handle is obtained, the *rpc_binding_set_auth_info*( ) function may be called with the binding handle and the authentication information from the *trig_binding* field to set authentication and authorisation information for this handle to the trigger server. See Section 11.8.1.3 on page 413 for more information on authentication information to be used.

Refer to Section 11.9.1.6 on page 431, the definition of the **sec_attr_schema_entry_t** data type, for more information on these two schema fields. Only if both of the above fields are specified will the association between the attribute type and attribute trigger be created. An association can be defined to any attribute type encoding except an attribute set. Attribute sets are described in Section 1.21.9 on page 106.

1.21.8.3 Query Triggers

If an attribute type is configured with a query trigger and a *sec_rgy_attr_lookup_**( ) of an attribute of that type is performed, the client side attribute lookup code will:

- bind to the triger server (using a binding from the attribute type's schema entry),
- make the remote *sec_attr_trig_query*( ) call, passing in the attribute keys (there can be optional information in the *attr_value* field) provided by the caller in the lookup, and
- if successful, return the output attribute(s) to the caller.

If a *sec_rgy_attr_update*( ) function is called for an attribute type with a query trigger, the input attrubute value is ignored and a ''stub'' attribute instance is created on the named object. This serves to mark the existence of this attribute on the object (nothing else is done). Modifications to the real attribute value must occur at the trigger server.

1.21.8.4 Update Triggers

If an attribut type is configured with an update trigger and a *sec_rgy_attr_update*( ) function is called, the server-side update code will:

- bind to the triger server (using a binding from the attribute type's schema entry),
- make the remote *sec_attr_trig_query*( ) call, passing in the attributes specified by the caller in the write, and
- if successful, store the output attribute(s) in the (ERA) database and return the output attribute(s) to the caller.

1.21.9 Attribute Sets

Attribute sets provide a flexible method of grouping related attributes on an object (for easier search and retrieval of related attributes — For instance, a query on an attribute set expands to a query on its members). The members of a set are defined in an instance of the attribute set, whose value is a vector of attribute type UUIDs. Different sets of members may be defined in each attribute set instance in order to tailor a set for the object to which it is attached.

Note: Attribute sets may not be nested. A member UUID of an attribute set may not itself identify an attribute set. This limitation may be removed in a future version of DCE (newer than DCD 1.1).

The attribute type UUIDss referenced in an attribute set instance must correspond to existing attribute schema entries. It is possible to create an attribute set instance on an object, however, before creating member attribute instances on that object (which might be advisable).

Attribute sets are defined by the **sec_attr_enc_attr_set_t** data type in Section 11.8.1.15 on page 418.

1.21.10 Access Control for Attribute Types

The definition of an attribute type in the schema enables the creation of attribute instances of the same type with a consistent format. In general, access to an attribute instance is controlled by the ACL on the object to which the attribute is attached. ACLs are not attached to attributes themselves. An attribute is ''just another data field'' on the Registry object to which it is attached. Access to that data field is controlled by permission bits on the Registry object's ACL. In other words, access to an attribute instance is controlled by the ACL on the object to which the attribute instance is attached, whereas access to a schema entry is controlled by the ACL on the **xattrschema** object.

For instance, consider an attribute, *A*. Suppose it is attached to a Principal object, *P*. Then, access to the *A* attribute on the *P* Principal object is controlled by the ACL on the *P* object.

Access control for a given attribute type is specified in the *acl_mgr_set* (**acl_mgr_type** and **permset** fields) of its schema entry. If a given ACL Manager type is not specified in the *acl_mgr_set*, then it is not possible to attach the attribute to Registry objects that use that ACL Manager type. (The ACL Manager types and the permissions they support are shown in Section 11.1 on page 358.)

1.21.10.1 Additional Attribute Permission Bits

The RS ACL Managers have been enhanced to support additional (generic) attribute type permissions that administrators may assign for access control for attribute types of their choice. The set of new permission bits {O, P, Q, ..., X, Y, Z} are supported by each ACL Manager for each Registry object type that supports ERAs. Thus, the list of valid permissions for each ACL Manager shown in Section 11.1 on page 358 has been extended with the ''O'' through ''Z'' permission bits. All uses of these additional (O-Z) attribute permission bits

- in the Access Permsets fields of schema entries,
- on ACLS, and
- in policies regarding their use,

will be controlled by the cell administrator. The DCE security services will not interpret or assign meaning to these bits other than what is implied by their inclusion in a schema entry.

1.21.11 Access Control on Attributes with Triggers

Access to information maintained by a trigger server is controlled entirely by that trigger server. The trigger server may choose to implement any authorisation mechanism (including none). A query operation on an attribute associated with a query trigger will undergo the following authorisation checks (the process for an update trigger is very similar):

- The *sec_rgy_attr_lookup_by_id*( ) or other *sec_rgy_attr_**( ) query function performs the *rs_attr_**( ) query remote call to the security server.
- The *rs_attr_lookup_by_id*( ) (or other *rs_attr_**( ) query function) will check the ACL of the named object to see if the client has the required permissions (specified in the **query_permset** field of the (*acl_mgr_set*) field of the schema entry). If access is granted, the operation returns the trigger binding information required to perform the *sec_attr_trig_query*( ). If access is not granted, the operation fails (see the specific function in Chapter 16) for specific information about the failure status codes returned.
  - If *sec_attr_trig_query*( ) *rs_attr_**( ) query succeeds, the *sec_attr_query**( ) function creates an authenticated (with the client's login context) binding handle to the trigger server and performs the *sec_attr_trig_query*( ).

    Upon receiving the *sec_attr_trig_query*( ) call, the trigger server may inquire of the RPC the client's identity, perform name-based authentication, perform ACL checks if it implements an ACL manager, or execute any other access control mechanism it has in place for the information being accessed.

    The trigger server may choose to query the Registry attribute schema for the **query_permset** configured for this attribute type, and use that information in an ACL check (it is however, under no obligation to so do).

    Essentially, the implementation of access controls on attribute information stored outside of the Registry database is left to the designers of those applications. This specification

does not describe, nor does it recommend the inforcement of, an authorisation mechanism for (use by) attribute trigger servers.

1.21.12 Well-Known Attribute Types

Certain definitions, known as ‘‘well-known’’ attribute types, are required for DCE 1.1 security services. These are definitions of architectural attribute types that will be used by the DCE security service to make security policy decisions. Well-known attributes ahve documented, well-known attribute type UUIDs (The type UUID is the authoritative key by which such attributes are known).

Although a standard DCE 1.1 installation may create the schema entries for well-known attributes as well as instances of those attributes, nothing can prevent an administrator from deleting them. Security policy will be affected by the absence of any given well-known attribute as described by the specification for that attribute. A cell administrator can easily recreate a schema entry for a well-known attribute if the type UUID and other characteristics for it are known.

The DCE 1.1 Extended Registry Attribute facility requires the well-known ‘‘unknown intercell action attribute’’ for determining policy that is defined in the next section. Refer to Section 1.21.5 on page 103 for implications of its absence upon the Privilege Server (PS), PS_Z, when reading attributes from a foreign cell.

1.21.12.1 Unknown Intercell Action Attribute

The DCE 1.1 Extended Registry Attributes facility requires the creation of the following well-known attribute type for the policy object:

- **attr_name**: unknown_intercell_action
- **attr_id**: 71e0ef2c-d12e-11cc-bb7b-080009353559

  (This is the ACL Manager type for the attr_schema Registry Object.)
- **attr_encoding**: sec_attr_encoding_integer
- **acl_mgr_type**: policy_acl_mgr
- **unique**: FALSE
- **multi-valued**: FALSE
- **reserved**: TRUE
- **comment**: Flag indicating whether to accept or reject foreign attributes for which no schema entry exists

Section 1.21.5 on page 103 provides information on the security policy implications of this attribute.

1.22 Extended Login and Password Management Overview

For DCE 1.1, login and password management have been improved to enhance several areas; pre-authentication, login denial, and password management.

The improvements to login and password management for DCE 1.1 and newer versions include the following:

- Pre-authentication
- Login denial through environmental parameters
- Login denial based upon registry attributes
- Detection and limitation of invalid login attempts
- Password strength control

1.22.1 Pre-authentication

Passive attacks on user passwords occur when the initial request for validation sends the login name of the user in the request and this verifiable plaintext data is returned from the Key Distribution Service (KDS) encrypted in the user's key in addition to a TGT. An attacker is able to make an initial request for validation using the user's name and then proceed to attack the response by using additional resources to derive the key and obtain the TGT by trying all combinations until the decryption yields the verifiable plaintext data sent in the original request.

Pre-authentication, in the generic sense, is a method of authenticating login attempts that attempts to thwart passive attacks on the KDS. It requires access to the derived key of a user to generate the initial request and does not send verifiable plaintext in the request. If the server is unable to decrypt the request, the login attempt fails and no credentials are returned.

Prior to DCE 1.1, authentication is particularly vulnerable to off-line password guessing attacks when users have ''weak passwords'' (derived from words). For DCE 1.1 (and newer versions), there are three protocols used by DCE Security clients and servers to perform this first part (pre-authentication) of the user-authentication process. They are as follows:

- The *third-party* protocol, which provides the highest level of security.
- The *timestamps* protocol, which is less secure (but still more secure than the DCE 1.0 protocol).
- The DCE 1.0 protocol, which is the least secure, and is provided solely to enable DCE 1.1 Security Servers to process requests from pre-DCE 1.1 clients.

1.22.1.1 Login Denial

This consists of two aspects — the Client and the Server.

1.22.2 Server

With the addition of pre-authentication, checks which were previously provided by the client can now be enforced by the server. Before pre-authentication, the only way a login was known to be successful was if the client could use the user's derived key to decrypt the TGT returned from the KDS. Utilising Extended Registry Attributes (new for DCE 1.1), administrators can now limit the number of failed login attempts for a principal and also lock an account if that number is exceeded.

Prior to DCE 1.1, checks for expired passwords were optional. They were enforced by the client. As of DCE 1.1, the TGT returned from the KDS is marked as *expired* if the password expiration

time has been reached or will be reached for the time the ticket is valid and the rejection of ‘‘expired TGT’s" will be enforced by the KDS.

1.22.2.1 Client

Prior to DCE 1.1, the inability of applications to specify additional criteria for controlling authentication of users was a weakness in the authentication mechanism. The presence of ERAs in DCE 1.1 (and newer versions) permits the specification of additional *application specific* checks for authentication. An application may define a ‘‘server’’ function and specify the binding to this function in an ERA which will be invoked AFTER authentication but BEFORE returning to the client. Section 1.21.8.2 on page 105 specifies details on this topic.

1.22.3 Password Management

User passwords are often cited as the weakest links in any security system. Control of passwords has been limited prior to DCE 1.1 to checks performed by the password program on the host machine before the calls to update the Registry. The only check enforced then was minimum password length.

Using DCE 1.1 ERAs, additional password controls may be specified and enforced in the server. In particular, ‘‘well-known’’ ERAs have been defined to permit application specific server functions for password strength checking and password generation. Administrators must specify the binding for these functions in the ERA and the presence of these ERAs will be checked by the server on password updates. In addition, ERAs are used to specify control parameters for the enforcement of limited password reuse. These ‘‘well-known’’ ERAs are defined in Section 1.23.6 on page 117.

1.23 Pre-authentication and Obtaining a TGT

The protocol used by the Security client when it makes a login request to the AS is determined as follows:

- Pre-DCE 1.1 clients always use the DCE 1.0 protocol.
- DCE 1.1 clients use the **third-party** protocol, unless the host machine's session key (which the client uses to construct the request) is unavailable. It then uses the **timestamps** protocol. These protocols will be explained in the descriptions that follow.

The protocol used by the AS to respond to the client is determined by the following:

- The protocol used by the client making the login request.
- The value of a *pre_auth_req* Extended Registry Attribute (ERA) attached to the requesting principal.

This can be summarised as follows:

The Authentication Service alsaays attempts to reply using the same protocol used by the client making the request, unless the value of the ERA ''forbids'' it to do so.

1.23.1 The Timestamps (AS + TGS) Protocol

Note: This pre-authentication protocol is also known as PADATA-ENC-TIMESTAMPS.

The *timestamps* protocol is identical to the DCE 1.0 protocol in Figure 1-2 on page 21 with the following additions:

- In step 1 (*AS Reguest: Ask for ticket to KDS*), Client *A* sends to the AS, in addition to the user's name (UUID) (as one of its arguments (to *sec_login_setup_identity*( )), a timestamp encrypted in the user's derived (secret) key.
- In step 1½ (*AS Response: Receive ticket to KDS*), the AS, before preparing the user's TGT, verifies the user as follows:
  1. It decrypts the timestamp using the copy of the user's key it obtained from the Registry.
  2. If the decryption succeeeds, and the timestamp is within 5 minutes of the current time, the user is verified, and the AS proceeds to prepare the TGT. If the decryption fails, or if the timestamp is not within 5 minutes of the current time, the Authentication Service rejects the login request.

  With this protocol the AS can verify that the client login request is timely and that the requesting client knows the user's password. It is thus aware of, and can manage, persistent login failures for a given user, eliminating passive password-guessing attacks. Details about this will be given later in this discussion.

  Note, however, that with this protocol, since the timestamp is identifiable text and the key for encryption is the user's derived key, it is still vulnerable to off-line attacks by processes monitoring network requests (although it eliminates the passive attacks made by sending bogus initial requests for validation).

1.23.2 The Third-Party (AS + TGS) Protocol

Note: This pre-authentication protocol is also known as PADATA-ENC-THIRD-PARTY.

The **third-party** protocol addresses both types of attacks mentioned previously — passive attacks made by sending bogus initial requests for validation (iteratively), and off-line attacks by processes (monitoring network requests).

This protocol uses a strong ''third party'' key to encrypt the *padata* in the request. (Thus, since it is a pre-authentication mechanism and it uses a ''strong session key'' (instead of the user's secret key) for encryption, it addresses the previously mentioned attacks.) The strong key is provided each machine running as a DCE client. They thus have access to a strong 56-bit DES key which is shared with the KDS.

1.23.2.1 Client Side

When a client principal is about to initiate a KRB_AS_REQ (initial authentication exchange with the KDS), it must first obtain the information it needs to compose the request. This information is as follows:

1. First, the client (*A*) must obtain the TGT for the machine principal on which it is executing. Client *A*'s process must have special privileges (usually termed, a ''privileged'' process) in order to obtain the machine principal TGT. ''Privileged'' in the sense meant here refers to, say, a program in a standard UNIX operating system that runs as a setuid to the root process. ''Privileged'' in terms of nonIX systems is beyond the scope of this document.

   Note: If the login application itself is ''privileged'', it has access to the machine principal TGT and construct this part of the request itself. If not, it obtains the following from ''sec_clientd'':

   1. The TGT
   2. A random key encrypted in the Machine Session Key (a strong key) otherwise known as the user's derived key.
   3. A random key to be shared between itself and the KDS.

   If sec_clientd is not available, which is the case when initially configuring a cell, then a third-party request cannot be generated and the protocol defaluts to the *timestamps* protocol.

2. Second, the user's derived key must be constructed using the user's supplied password and a ''salt''. The password is known by Client *A* and the ''salt'' is, in most cases a *default* value.

   Note: To reduce the number of RPC's necessary in the usual case, Client *A* will attempt to generate the user's derived key using the default salt. On a login failure, the server (KDS) will check to see if Client *A* uses a non-default salt, and if so, will return this ''non-default'' salt with the error. Client *A* will detect the presence of this salt and retry the login using the ''non-default'' salt.

3. Third, the client (Client *A*) generates a random key to use as a shared key between itself and the KDS for the reply (called the ''reply random key'').

Now that the required information is obtained, Client *A* must compose the *padata* block. To do this, Client *A* takes the current time and the ''reply random key'' and encrypts this data using the user's derived key. It then takes the random key and encrypts the data a second time.

The *padata* now consists of the machine TGT, the random key encrypted in the Machine Session Key and an encrypted data block. The machine TGT identifies the third party sponsor for the

exchange and the reply random key established a shared key between the login application (referred to herin as Client *A*) and the KDS.

1.23.2.2 Signature of padata Field

The padata field constructed as described in the preceeding section, has the signature below. For items denoted as ‘‘X_y’’ the meaning is to be interpereted as ‘‘X’’ is encrypted using ‘‘$_y$’’ as a key. Also, ‘‘msk’’ is shorthand for ‘‘machine session key’’, and ‘‘uk’’ is shorthand for ‘‘user’s derived key’’. Lastly, ‘‘rk’’ is shorthand for ‘‘random key’’.

$$[[\text{machine-TGT}][(\text{random-key})_{msk}][(([\text{timestamp}][\text{reply-random-key}])_{uk})_{rk}]]$$

Figure 1-15 Signature of the KDS *padata* Field

1.23.2.3 Server Side

On the KDS, the principal database is extended with an Extended Registry Attribute *pre_auth_req* that determines if that principal must use pre-authentication. This is a ‘‘well-known’’ ERA, defined in Section 1.23.6 on page 117,
whose allowable values are (relating to if the principal must use pre-authentication):

- **0** — NONE — (Denotes no pre-authentication)
- **1** — PADATA-ENC-TIMESTAMPS — (Denotes *timestamps* protocol)
- **1** — PADATA-ENC-THIRD-PARTY — (Denotes *third-party* protocol)

When processing a request, the KDS server examines the *pre_auth_req* ERA for the selected principal. If the request satisfies the restriction, then the server will provide the corresponding reply. There is an implicit hierarchy in the ERA values for pre-aauthentication.

The KDS will only reject requests if the ERA contains a value more restrictive than the actual request.

Thus, if Client *A* initiates a KRB_AS_REQ in the third-party protocol form and the the client principal has an ERA with a value of PADATA-ENC-THIRD-PARTY, the restriction is satisfied and the login is successful (all other things being equal), so the reply will be a TGT encrypted in the random-reply-key.

If however, a DCE 1.1 KDS servr receives a DCE 1.0 (no pre-authentication) KRF_AS_REQ from Client *A* whose principal has an ERA with the value of PADATA-ENC-THIRD-PARTY, the restriction is not satisfied and the request will be denied.

If the ERA for a principal contains the value NONE, but the principal logs in from a DCE 1.1 client, this client (Client *A* for reference purposes), Client *A* will generate a pre-authentication request (KRB_AS_REQ) specifying PADATA-ENC-THIRD-PARTY, and send it to the KDS. The KDS will accept this request because PADATA-ENC-THIRD-PARTY is a stronger authentication mechanism than specified in the ERA (*pre_auth_req*).

On the other hand, if the scenario were reversed and the ERA value was PADATA-ENC-THIRD-PARTY, and the request came from a DCE 1.0 client with no pre-authentication support, the login attempt would fail since the authentication mechanism is less secure than that specified in the ERA.

Note that if an initial KRB_AS_REQ is denied, it may be because a salt other than the default was used to generate the user’s derived key. The KDS (server) can determine if the default salt was used and if a non-default salt was used and the decryption of the request fails, the KDS will issue

a KRB_ERROR reply containing the correct salt retrieved from the Registry. Client *A* can then repeat the construction of the *padata* and retry the KRB_AS_REQ. (The KRB_ERROR has the status KDC_PREAUTH_FAILED.)

Thus, there are two reasons Client *A*'s initial KRB_AS_REQ can fail. They are as follows:

- The salt used to derive the user's key is a value other than the default.
- The ERA on the client principal has a protocol more restrictive than that used in the request.

1.23.3 Third-party Pre-authentication Protocol

Figure 1-16 on page 115 illustrates the steps in the *sec_login_**( ) functions that support pre-authentication using the **third-party** protocol. (Section 1.23.1 on page 111 describes the *timestamps* protocol.)

In the figure, step 1 is shown simply to note that while previously, *sec_login_setup_identity* ( ) initiated the KRB_AS_REQ to the (AS function of the) KDS, this functionality is now (in DCE 1.1 and newer versions) being accomplished by the two functions *sec_login_validate_identity* ( ) and *sec_login_valid_and_cert_ident* ( ).

Step 2 is also initiated by a DCE 1.1 Client, Client *A*, in terms of the terminology of Figure 1-2 on page 21 (Basic KDS (AS+TGS) Protocol).

In step 3, the AS portion of the KDS will check for the existence of a *pre_auth_req* ERA and return the value of the ERA along with the ''salt'' (used for encryption of the user's password).

In step 4, Client *A* uses the returned information in order to perform step 5. This (step 5) step corresponds to the bullet item $A \rightarrow$ AS: *A*, KDS, $L_{A,KDS}$, $N_{A,AS}$ in Figure 1-2 on page 21 where step 1 (*AS Request: Ask for ticket to KDS*) is performed.

In step 6 (which corresponds to bullet item $A \leftarrow$ AS: *A*, {KDS, $K_{A,KDS}$, $L_{A,KDS}$, $N_{A,AS}$}K_A, $Tkt_{A,KDS}$ of Figure 1-2 where in step 1½ (*AS Response: Receive ticket to KDS*) a response is sent to Client *A* for the KRB_AS_REQ it sent to the KDS), the KDS attemptss to decrypt the request. If successful, the KRB_AS_REP returned to Client *A* will contain a TGT. If the decryption fails, a KRB_ERROR (KDC_PREAUTH_FAILED) is returned and the login attempt is terminated.

If the decryption is successful, the login sequence contines (at step 2 (*TGS Request: Ask for ticket to server*) in Figure 1-2 on page 21).

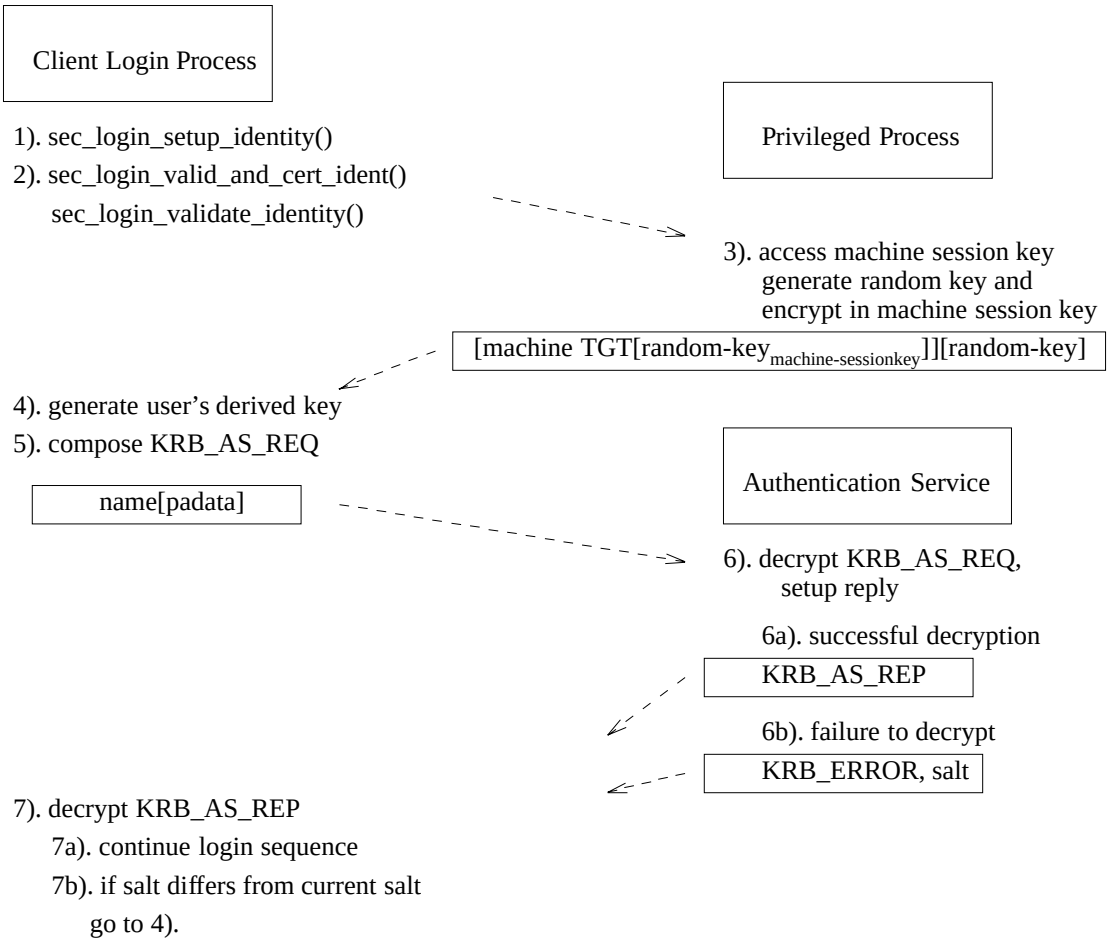


Figure 1-16 Pre-authentication Protocol for KDS

1.23.4 Environmental Parameters and Registry Attributes

Authentication checks performed by application servers may require additional information about the application user to determine whether or not the user is authorised to use the application. Examples of such information might include items such as the following:

- physical location
- user's timezone
- ID of user's machine (where logged on)
- type of connection from user to Client machine

New features for DCE 1.1, delegation and Extended Registry Attributes, provide an underlying mechanism to provide for additional application specific information being securely attached to a client principal's credentials. Along with this, changes made in the *sec_login_**() functions support pre-authentication. These include changes to *sec_login_validate_identity*() and *sec_login_valid_and_cert_identity*(). These changes are discussed in Section 1.23.3 on page 114. The mentioned functions are in Chapter 19.

Several ‘‘well-known’’ Extended Registry Attribute sets called:

- *environment_set*,
- *login_set* and
- *policy_check_set*

are provided in DCE 1.1 to assist in pre-authentication. In addition, when it has been determined that an invalid login attempt has occurred (has been detected by the Security Server), two ‘‘well-known’’ ERAs are defined to limit the impact of password attacks. They permit an administrator to control two aspects of the user’s authentication by:

- setting a maximum number of consecutive bad attempts allowed before an account is locked (the *max_invalid_attempts* ERA). This is an integer that reflects the number (minus one) of login attempts allowed before a principal is disabled from login attempts. A single successful login to an account resets the number of bad attempts.
- specifying a time to disable an account once the maximum number of attempts (at login) has been reached. This ERA is known as the *disable_time_interval* ERA. The time is specified in seconds. It can also be specified as FOREVER if manual administrative action is the desired policy.

These ‘‘well known’’ ERAs are defined in Section 1.23.6 on page 117.

1.23.5 Password Management

Prior to DCE 1.1, the following password strength policies were supported:

- Minimum password length (default 0)
- Whether a password can be all spaces (default YES)
- Whether a password can be all alphanumeric (default YES)

As of DCE 1.1, known deficiencies with user-defined passwords such as common names, reuse, and so forth, have been restricted by password management options that include the following:

- Non-trivial password strength checking

  The *pwd_val_type* ERA has been provided to assist in this. It contains the following values:

  - NONE

    No password checking is necessary.

  - USER_SELECT

    A user must supply a password.

  - USER_CAN_SELECT

    A user can supply a password or an ‘‘*’’ to indicate they would like the system to generate a password for them.

  - GENERATION_REQUIRED

    User input will be ignored. See next item.

  (For USER_SELECT and USER_CAN_SELECT values, individual sites can specify a binding for the server exporting the interface as described in Section 1.21.8.2 on page 105, ‘‘Trigger Binding’’.)

- Password generation

If the value of the *pwd_val_type* ERA is GENERATION_REQUIRED, a strong password will be generated. As with the USER_* values, individual sites can replace the supplied password generation server with one of their own.

- Password reuse checking

  To assist in this, two ''well-known'' ERAs are defined — *minimum_password_cycle_time* (the time in minutes before passwords can be reused) and *passwords_per_cycle* (the limit on the number of previous passwords).

  Users are not permitted to change their passwords more often than the *passwords_per_cycle* times during *minimum_password_cycle_time*.

1.23.5.1 Password Expiration

Previously (prior to DCE 1.1), whenever a password expired, the Client determined whether or not to allow logins. With the advent of DCE 1.1, the KDS will terminate a login attempt if the password is expired.

The ''well-known'' *passwd_override* ERA has been defined in DCE 1.1 to permit a principal assigned the ERA to login even if their password has expired. This ERA should be configured by default on the administrator principal so that an administrator will not be locked out from getting tickets.

It is recommended by this specification that user principal accounts use the default action which enforces the password expiration.

1.23.6 Schemas for Well-known Attributes

The following prototype schema entries illustrate the ''well-known'' ERAs mentioned in this document. They are all of the *sec_attr_schema_entry_t* data type defined in Section 11.9.1.6 on page 431.

These ''well-known'' ERAs share certain characteristics. The **axl_mgr_set**, for instance, identifies *princ*, *org* and *policy*. It is intended that ''well-known'' ERAs are set on different objects in order to permit either fine or rough granularity of enforcement. For instance, regular users might have a *disable_time_interval* ERA of 1 day, since it is conceivable they could have a bad day and exceed their 2max_invalid_attempts allocation. Thus, instances of the *disable_time_interval* ERA with a value of ''1'' might be created on the *org* and *policy* object. But, if an attack is made on the **admin** principal, installations would want to know about it. This can be handled by creating a f2disable_time_interval value of ''forever'' on this principal (forcing a manual intervention if *max_invalid_attempts* is exceeded).

To permit this granularity, the *apply_defaults* field (of the ERA) must be set to ''TRUE''. It is important to specify how access is controlled to attribute instances, and this is done by the ACL on the object to which the attribute is attached. Access control is specified in the *acl_mgr_set*, which consists of an **acl_mgr_type** (say principal) and **permset** (say m) of a schema entry. The intent is that ''principals'' should not be able to change or delete these ''well-known'' ERAs. Only those that possess the ''m'' permission should be permitted to access the ERAs.

The syntax of the *acl_mgr_set* schema (entry) is as follows:

acl_mgr_type:Query / Update / Test / Delete

1.23.6.1 disable_time_interval ERA

attr_name: ‘‘disable_time_interval’’
attr_uuid: 63005af0-dd2d-11cc-946f-080009353559
attr_encoding: INTEGER
acl_mgr_set:
    princ:m/m/m/m
    org:m/m/m/m
    policy:m/m/m/m
unique:FALSE
multi-valued: FALSE
reserved:TRUE
apply_defaults: TRUE
intercell_action:
trig_types:
trig_binding:
comment: "amount of time in minutes to disable account"

1.23.6.2 max_invalid_attempts ERA

attr_name: ‘‘max_invalid_attempts’’
attr_uuid: 657eb68c-dd2d-11cc-8990-080009353559
attr_encoding: INTEGER
acl_mgr_set:
    princ:m/m/m/m
    org:m/m/m/m
    policy:m/m/m/m
unique:FALSE
multi-valued: FALSE
reserved:TRUE
apply_defaults: TRUE
intercell_action:
trig_types:
trig_binding:
comment: "number of invalid attempts allowed before locking account"

1.23.6.3 minimum_password_cycle_time ERA

attr_name: ‘‘minimum_password_cycle_time’’
attr_uuid: 66513166-dd2d-11cc-9db5-080009353559
attr_encoding: INTEGER
acl_mgr_set:
princ:m/m/m/m
org:m/m/m/m
policy:m/m/m/m
unique:FALSE
multi-valued: FALSE
reserved:TRUE
apply_defaults: TRUE
intercell_action:
trig_types:
trig_binding:
comment: "minutes before password can be reused"

1.23.6.4 passwords_per_cycle ERA

attr_name: ‘‘passwords_per_cycle’’
attr_uuid: 67090868-dd2d-11cc-a84d-080009353559
attr_encoding: INTEGER
acl_mgr_set:
princ:m/m/m/m
org:m/m/m/m
policy:m/m/m/m
unique:FALSE
multi-valued: FALSE
reserved:TRUE
apply_defaults: TRUE
intercell_action:
trig_types:
trig_binding:
comment: "limit on number of previous passwords within minimum password cycle time"

1.23.6.5 pwd_val_type ERA

attr_name: ‘‘pwd_val_type’’
attr_uuid: 689843ce-dd2d-11cc-a3e1-080009353559
attr_encoding: INTEGER
acl_mgr_set:
princ:m/m/m/m
org:m/m/m/m
policy:m/m/m/m
unique:FALSE
multi-valued: FALSE
reserved:TRUE
apply_defaults: TRUE
intercell_action:
trig_types:
trig_binding:
comment: "{0=NONE, 1=USER_SELECT, 2=USER_CAN_SELECT, 3=GENERATION_REQUIRED}"

1.23.6.6 password_generation ERA

attr_name: ‘‘password_generation’’
attr_uuid: 69b421a6-dd2d-11cc-bac5-080009353559
attr_encoding: BINDING
acl_mgr_set:
princ:m/m/m/m
org:m/m/m/m
policy:m/m/m/m
unique:FALSE
multi-valued: FALSE
reserved:TRUE
apply_defaults: TRUE
intercell_action:
trig_types:
trig_binding:
comment: "binding to server exporting the sec_login_password_generate interface"

1.23.6.7 pwd_mgmt_binding ERA

attr_name: ‘‘pwd_mgmt_binding’’
attr_uuid: 6a93b8f2-dd2d-11cc-9be7-080009353559
attr_encoding: BINDING
acl_mgr_set:
    princ:m/m/m/m
    org:m/m/m/m
    policy:m/m/m/m
unique:FALSE
multi-valued: FALSE
reserved:TRUE
apply_defaults: TRUE
intercell_action:
trig_types:
trig_binding:
comment: "binding to server exporting the sec_pwd_mgmt_binding interface"

1.23.6.8 pre_auth_req ERA

attr_name: ‘‘pre_auth_req’’
attr_uuid: 6c9d0ec8-dd2d-11cc-abdd-080009353559
attr_encoding: INTEGER
acl_mgr_set:
    princ:m/m/m/m
    org:m/m/m/m
    policy:m/m/m/m
unique:FALSE
multi-valued: FALSE
reserved:TRUE
apply_defaults: TRUE
intercell_action:
trig_types:
trig_binding:
comment: "{0=NONE, 1=PADATA-ENC-TIMESTAMPS, 2=PADATA-ENC-THIRD-PARTY}"

1.23.6.9 passwd_override ERA

attr_name: ‘‘passwd_override’’
attr_uuid: bc51691e-dd2d-11cc-9866-080009353559
attr_encoding: INTEGER
acl_mgr_set:
princ:m/m/m/m
org:m/m/m/m
policy:m/m/m/m
unique:FALSE
multi-valued: FALSE
reserved:TRUE
apply_defaults: TRUE
intercell_action:
trig_types:
trig_binding:
comment: "{0=NONE, 1=OVERRIDE}"

1.23.6.10 login_set ERA

attr_name: ‘‘login_set’’
attr_uuid: 6d8d97bc-dd2d-11cc-b1cc-080009353559
attr_encoding: attr_set
acl_mgr_set:
princ:m/m/m/m
org:m/m/m/m
policy:m/m/m/m
unique:FALSE
multi-valued: FALSE
reserved:TRUE
apply_defaults: TRUE
intercell_action:
trig_types:
trig_binding:
comment: "standard set of attributes to be returned on login"

1.23.6.11 environment_set ERA

attr_name: ‘‘environment_set’’
attr_uuid: ba4a5824-dd2d-11cc-a9f3-080009353559
attr_encoding: attr_set
acl_mgr_set:
princ:m/m/m/m
org:m/m/m/m
policy:m/m/m/m
unique:FALSE
multi-valued: FALSE
reserved:TRUE
apply_defaults: TRUE
intercell_action:
trig_types:
trig_binding:
comment: "attached to machine principals; standard set of environment attributes"

CAE Specification

Part 2
Security Services and Protocols

The Open Group

Chapter 2

Checksum Mechanisms

Section 2.1 is devoted to general terminology, notation and conventions that are used throughout this specification.

Subsequent sections of this chapter specify the (cryptographic and non-cryptographic) checksum mechanisms supported by DCE. The following list specifies all currently supported checksum mechanisms, and this chapter is therefore restricted to these checksums only:

- *Non-cryptographic checksums*

  The *CRC-32 Cyclic Redundancy Checksum*.

- *Cryptographic checksums*

  The *MD4* and *MD5 Message Digest Algorithms*, of RSA Data Security, Inc.

2.1 Terminology, Notation and Conventions

This section introduces terminology, notation and conventions, including low-level *formatting details*, used throughout this specification.

2.1.1 Use of Pseudocode

Pseudocode is employed in this and the following chapters. *It has the expository purpose of explaining observable external behaviour only, and does not impose internal requirements on conforming implementations.* The pseudocode notation is mostly ''C-like'', augmented by some other elements of standard scientific exposition (for example, *''f(x) = y'' to define the value of a function f* at an argument *x* to be *y*). Anything expressed in pseudocode is intended to be readily understandable by the intended audience of this specification, but if confusion can possibly result the pseudocode is also expressed in words.

2.1.2 Sequences

A (*finite*) *sequence* (or *string* or *vector* or *n-tuple*) V of *length* $\lambda(V) = n \geq 0$ will be denoted in order of increasing ''address'' (or ''name'' or ''subscript'') notation; that is, in the form $V = \langle v_0, \cdots, v_{n-1} \rangle$; if $n = 0$, this notation degenerates to $V = \langle\rangle$, the *empty* string. The n-tuple V is said to *begin*, at the *left*, with *initial* element v_0, and to *end*, at the *right*, with *final* element v_{n-1}. If $0 \leq n' < n'' \leq n-1$, then $v_{n'}$ is said to *precede* or *occur before* (or *earlier than* or *to the left of*) $v_{n''}$ in V, and $v_{n''}$ is said to *follow* or *occur after* (or *later than* or *to the right of*) $v_{n'}$. The *natural identification* of a 1-tuple is always made with ''the thing itself'': $\langle V \rangle = V$. Similarly, the *concatenation* of an m-tuple and an n-tuple are also identified with an $(m+n)$-tuple via $\langle U, V \rangle = \langle\langle u_0, \cdots, u_{m-1} \rangle, \langle v_0, \cdots, v_{n-1} \rangle\rangle = \langle u_0, \cdots, u_{m-1}, v_0, \cdots, v_{n-1} \rangle$. In particular, $\langle V, \langle\rangle\rangle = \langle\langle V \rangle\rangle = \langle V \rangle = V$ (via natural identifications). The terminology *prepend to V* refers to concatenating another vector to the beginning of *V*; *append to V* refers to concatenation to the end of *V*.

2.1.3 Bits, Bytes, Words, etc.

A *bit* is a boolean (binary) quantity that can take either of the values T (*true*) or F (*false*). A *byte* (or *octet*) is a sequence of 8 bits. A *shortword* is a sequence of 2 bytes, a *word* is a sequence of 4 bytes, a *longword* is a sequence of 8 bytes, and a *quadword* is a sequence of 16 bytes. By concatenation, a sequence of bits of length congruent to 0 modulo 8, 16, 32, 64 or 128, respectively, can be identified with a sequence of bytes, shortwords, words, longwords or quadwords (of the appropriate shorter length). Similarly, a sequence of bytes of length divisible by 4 can be identified with a sequence of words, and so forth.

Notes:

1. Some other terms exist in the literature that won't be used in this specification; for example, ''nibble'' for a sequence of 4 bits. Also, the definitions of some of the terms above vary throughout the literature; for example, in some quarters (''16-bit architectures''), ''word'' means a sequence of 2 bytes, with corresponding redefinitions of ''longword'' and ''quadword''.
2. Bits can be naturally identified with the elements of the finite field of 2 elements (integers modulo 2), $\mathbf{F}_2 = \{\mathbf{0}, \mathbf{1}\}$ (called the *bit field* in this context), via the mapping (or correspondence) defined by: F $\leftrightarrow$ **0**, T $\leftrightarrow$ **1**. And then, *n*-tuples of bits can be viewed as elements of the *vector space* $\mathbf{F}_2^n$ of dimension *n* over $\mathbf{F}_2$. This point of view is standard in mathematical treatments of cryptography, but it is not insisted upon here.

2.1.4 Integer Representations (Endianness)

The notion of ''endianness'' arises when one maps bit- and/or byte-sequences to (*non-negative*) integers. Namely, the question is whether the *left* (~ ''big'') or *right* (~ ''little'') end of the sequence is mapped to the higher value (that is, is ''more significant'').

First recall that any non-negative integer *i* has a natural *2-adic expansion* (which is *unique* if it is the *minimal* 2-adic expansion; that is, if one does not allow superfluous ''leading zeroes''): $i = 2^{k-1}i_{k-1} + \cdots + 2^0 i_0$ with each coefficient i_j ($0 \le j \le k-1$, $k \ge 0$) equal to 0 or 1 (and $2^{k-1} = 1$). The question of endianness arises when one imposes an *order* on the collection of coefficients $i_0, \cdots, i_{k-1}$; that is, when one realises this collection of coefficients as a *sequence*. Given a mapping between coefficients and sequences, bits, bytes, shortwords, words, longwords and quadwords (or, for that matter, bit- or byte-sequences of any length) can be interpreted as non-negative (''unsigned'') integers, in the ranges $[0, 2^k-1]$ where k = 1, 8, 16, 32, 64 or 128, respectively, as described below.

See Figure 2-1 on page 129 for an illustration of the endianness concepts defined below. As shown there, ''addresses'' (names or subscripts of sequence elements) are always visualised as increasing (with respect to a fixed ordering on addresses themselves) from left to right. The meaning of big- (respectively, little-) endianness is then determined by whether sequence elements are mapped to integer power-of-2 values (''significance'') in a decreasing (respectively, increasing) manner from left to right, respectively.

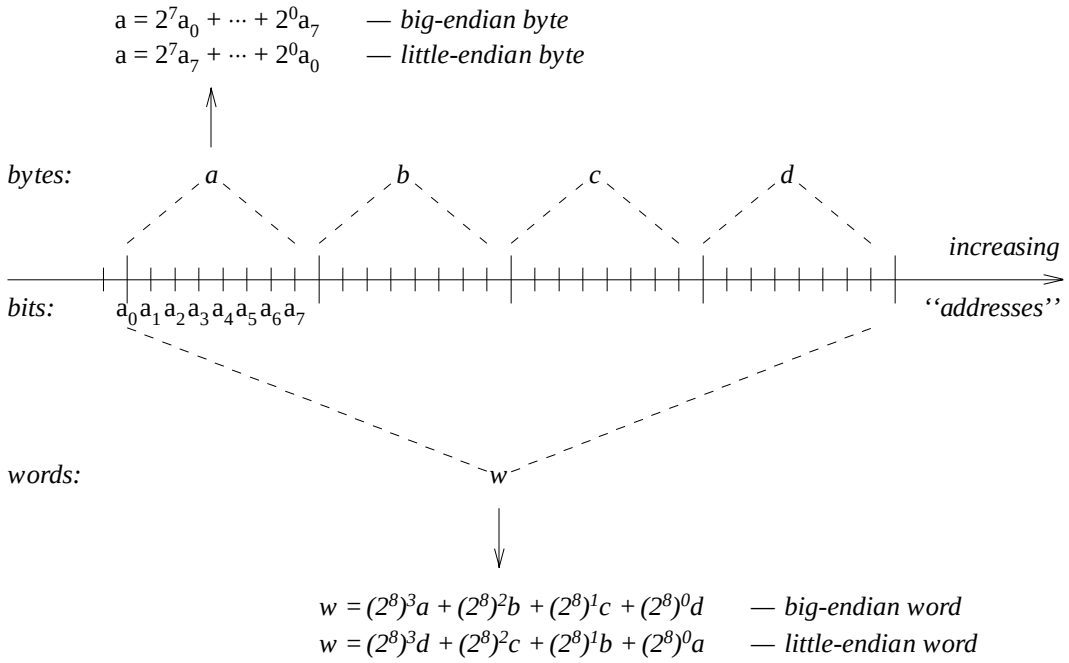


Figure 2-1 Endianness

2.1.4.1 Mapping Bit Sequences to Integers

In all mappings of bit-sequences to integers that are considered, *single* bits are mapped to integers via the mapping F ↔ 0 and T ↔ 1 (and these mappings are always treated as *identifications*, even notationally); that is, for *bit-sequences of length 1* the mapping/identification are always made between bit-sequences and integers:

<0> ↔ 0 — that is, ''<0> = 0''

<1> ↔ 1 — that is, ''<1> = 1''

Bytes (and more generally, any *bit*-sequences) can be interpreted as integers via either the *big-endian* (or *most-significant-first*) mapping, or the *little-endian* (or *least-significant-first*) mapping, which are now defined. (Other mappings of bit-sequences to integers are also possible, but not of interest.)

In big-endian mapping, the bits of the byte are considered to be ordered in decreasing significance, from the *high-order* or *most significant bit* (MSb) (on the left) to the *low-order* or *least significant bit* (LSb) (on the right). That is, for bytes (for example, bit-sequences of other lengths are similar):

$\langle a_0, a_1, \cdots, a_7 \rangle \leftrightarrow 2^7 a_0 + 2^6 a_1 + \cdots + 2^0 a_7$

And in little-endian mapping, the ordering goes the opposite way:

$\langle a_0, a_1, \cdots, a_7 \rangle \leftrightarrow 2^7 a_7 + 2^6 a_6 + \cdots + 2^0 a_0$

When one of these mappings has been chosen and fixed in a given context, it will usually be considered to be an ''identification'' instead of a ''mapping'', and the symbol ''='' instead of ''↔'' will be used (by a common abuse of notation).

2.1.4.2 Mapping Byte-sequences to Integers

Similarly, bytes, shortwords, words, longwords and quadwords (and more generally, any bit-sequence of length a multiple of 8 *that is considered as a byte-sequence*) can be interpreted, *once a mapping of bytes to integers has been chosen and fixed*, as integers via either big-endian or little-endian mapping (for single bytes, the two mappings coincide of course), which are now defined.

In big-endian mapping, the bytes of the byte-sequence are considered to be ordered in decreasing significance, from the high-order or *most significant byte* (MSB) to the low-order or *least significant byte* (LSB). That is, for words (for example, byte-sequences of other lengths are similar):

$$<a, b, c, d> \leftrightarrow (2^8)^3a + (2^8)^2b + (2^8)^1c + (2^8)^0d$$

And in little-endian mapping, the ordering goes the opposite way:

$$<a, b, c, d> \leftrightarrow (2^8)^3d + (2^8)^2c + (2^8)^1b + (2^8)^0a$$

2.1.4.3 Mapping Mixed Bit/Byte-sequences to Integers

Since bit-sequences of length a multiple of 8 can also be viewed as byte-sequences, for such sequences the above mappings can be mixed to arrive at the following *taxonomy of endianness* for ''byte / bit''-sequences.

Let W be, say, a (''word-sized'') integer in the range $[0, 2^{32}–1]$ (similar remarks hold for integers of other sizes), with 2-adic expansion:

$$W = 2^{31}w_{31} + \cdots + 2^0w_0 =$$
$$(2^8)^3(2^7w_{31} + \cdots + 2^0w_{24}) + (2^8)^2(2^7w_{23} + \cdots + 2^0w_{16}) +$$
$$(2^8)^1(2^7w_{15} + \cdots + 2^0w_8) + (2^8)^0(2^7w_7 + \cdots + 2^0w_0)$$

There are then the following four bit-representations of W. Here, the terminology ''X / Y-endian'' (for X, $Y \in$ {''big'', ''little''}) means: ''X-endian with respect to bytes, the bytes being Y-endian with respect to bits''. (See also Figure 2-1 on page 129.)

- *Big/big-endian* mapping:

  $$W \leftrightarrow <<w_{31}, \cdots, w_{24}>, <w_{23}, \cdots, w_{16}>, <w_{15}, \cdots, w_8>, <w_7, \cdots, w_0>>$$

- *Little/big-endian* mapping:

  $$W \leftrightarrow <<w_7, \cdots, w_0>, <w_{15}, \cdots, w_8>, <w_{23}, \cdots, w_{16}>, <w_{31}, \cdots, w_{24}>>$$

- *Little/little-endian* mapping:

  $$W \leftrightarrow <<w_0, \cdots, w_7>, <w_8, \cdots, w_{15}>, <w_{16}, \cdots, w_{23}>, <w_{24}, \cdots, w_{31}>>$$

- *Big/little-endian* mapping:

  $$W \leftrightarrow <<w_{24}, \cdots, w_{31}>, <w_{16}, \cdots, w_{23}>, <w_8, \cdots, w_{15}>, <w_0, \cdots, w_7>>$$

Of the four mappings listed above, the first two are the most important in this specification, because bytes in computer memories are invariably considered to represent integers (in the range $[0, 2^8–1]$) via the big-endian mapping. (On the other hand, bytes are variously transmitted over different serial networks in big-endian or little-endian order: the most or least significant bit may be transmitted first.) Clearly, one could extend the above ideas about endianness to include such higher-level constructs as shortword-sequences, word-sequences, and so on — the ideas are straightforward, and there is no need to elaborate on them here.

2.1.5 Modular Arithmetic

The usual convention is adopted that *modular arithmetic* on integers is denoted by ''(mod *n*)'' where $n > 0$ is the *modulus*, and the modular equivalence class of any integer N is *identified* with its representative in the interval $[0, n-1]$. That is, for any integer N (positive, negative or zero):

$$0 \le N \pmod{n} \le n-1$$

2.1.6 Bitwise Operations and Rotations

The following notations for four *bitwise operations* will be used, all operating on bit-sequences (usually words) U, V, W of the same length:

- $\tilde{}\, U$

  Bitwise boolean NOT (or *binary complement*). On bits, it is defined by:

  $\tilde{}\, 0 = 1$

  $\tilde{}\, 1 = 0$

- $U \mid V$

  Bitwise boolean OR. On bits, it is defined by:

  $0 \mid 0 = 0$

  $0 \mid 1 = 1$

  $1 \mid 0 = 1$

  $1 \mid 1 = 1$

- $U \,\&\, V$

  Bitwise boolean AND. On bits, it is defined by:

  $0 \,\&\, 0 = 0$

  $0 \,\&\, 1 = 0$

  $1 \,\&\, 0 = 0$

  $1 \,\&\, 1 = 1$

- $U \mathbin{\hat{}} V$

  Bitwise boolean XOR (''exclusive or''; that is, bitwise binary (mod 2) addition of integers). On bits, it is defined by:

  $0 \mathbin{\hat{}} 0 = 0$

  $0 \mathbin{\hat{}} 1 = 1$

  $1 \mathbin{\hat{}} 0 = 1$

  $1 \mathbin{\hat{}} 1 = 0$

Also, the following notation for two kinds of *rotations* will be used (the first is used in this chapter, the second is used in Chapter 3):

- $i \lll s$

  Left numerical rotation (or *left numerical circular shift*) by s, operating on the *value* of an integer i (not any of its *bit-representations*), is defined as follows. If $i = 2^{k-1}i_{k-1} + \cdots + 2^0 i_0$ is a 2-adic expansion of i in powers of 2 (not necessarily the minimal 2-adic expansion of i; that is,

leading zeroes are permitted), then for any integer *s*:

$$i <<< s = 2^{k-1} i_{(k-1+s) (\mathrm{mod}\ k)} + \cdots + 2^0 i_{(0+s) (\mathrm{mod}\ k)}$$

(Even though this definition is valid for all *s*, it will only be used for $0 \le s \le k-1$.)

- $V <<<< s$

  Left bitwise rotation (or *left bitwise circular shift*) by *s*, operating on the bit-vector *V* (not its *value* with respect to either endianness), is defined as follows. If $V = \langle v_0, \cdots, v_{n-1} \rangle$, then for any integer *s*:

  $$V <<<< s = \langle v_{(0+s) (\mathrm{mod}\ n)}, \cdots, v_{(n-1+s) (\mathrm{mod}\ n)} \rangle$$

  (Even though this definition is valid for all *s*, it will only be used for $0 \le s \le n-1$.)

Notes:

1. Note that the two kinds of rotations agree for, say, words ($k = n = 32$) if and only if words are identified with integers via the *big/big-endian* mapping — but we're using the *little/big-endian* mapping in the remainder of this chapter.
2. The notations ''<<<'' and ''<<<<'' for the two left circular shift operators defined above have been adopted in analogy with the C-language ''<<'' operator. (The C ''<<'' operator is a left non-circular numerical shift operator, despite the fact that it is usually spoken of as a ''bitwise'' operator.)

2.1.7 (IDL/NDR) Pickles

By a *pickle* is meant an encoded/marshalled (or ''linearised'' or ''flattened'') bit-vector representation of a (value of a) data type specified in some computer language, suitable for application-level storage purposes in the absence of a communications context (hence the use of the word ''pickle'', meaning ''preserved substance''). Pickles appear in several places in this specification.

In the case of the present revision of DCE, the only language currently supported for pickles is IDL, and the only encoding/marshalling currently supported for IDL pickles is NDR (for the specifications of the IDL language and its NDR marshalling/encoding, see the referenced X/Open DCE RPC Specification). Therefore, for the purposes of this revision of DCE, ''pickle'' always means *IDL/NDR pickle* (specified in detail below). In essence, then, a pickle for the purposes of this specification is an in-memory representation of RPC input/output data that ''normally'' exists only ''on-the-wire'' (and hence is normally only interpreted by the RPC runtime), in a form that can be interpreted at application level.

Pickles as specified in this specification are bit-vectors (actually, *byte*-vectors — see below) having a common structure, which will be denoted:

$$PKL = \langle H, B \rangle$$

where:

- *H* is the pickle's *header*. It is metadata, describing the actual data carried by the (body of the) pickle. It is always non-empty.
- *B* is the pickle's *body*. It embodies the actual data carried by the pickle. It consists of IDL-defined NDR-marshalled data. (Actually, as will be seen below, *B* itself also contains some second-level metadata.) It is always non-empty.

Each of *H* and *B* is actually a *byte*-sequence (that is, has *bit*-length a non-negative integral multiple of 8); they will henceforth always be regarded as byte-sequences, not bit-sequences. In particular, the *length* of such a (byte-)vector *M* henceforth in this section will always mean its

length *in bytes*.

PKL's header *H* is specified as an *IDL data type* (but whose encoding is *not* NDR) as follows (for the IDL definition of the data types not defined here, see Appendix N, IDL Data Type Declarations, of the referenced X/Open DCE RPC Specification):

```
typedef struct {
    uuid_t                          stx_id;

    unsigned32                      stx_version;
}                                       rpc_syntax_id_t;


typedef struct {
    unsigned8                       pkl_version;
    unsigned8                       pkl_length_hi;
    unsigned16                      pkl_length_low;
    rpc_syntax_id_t                 pkl_syntax;

    uuid_t                          pkl_type;
}                                       idl_pkl_header_t;
```

The fields of these data types are the following:

- **stx_id**

  RPC (''transfer'') syntax identifier (16-byte IDL-defined UUID (**uuid_t**), encoded in big/big-endian format — see below). Since this field represents a UUID, it can also be specified as a string (see Appendix A, Universal Unique Identifier, of the referenced X/Open DCE RPC Specification). Its registered values are specified in Appendix I, Protocol Identifiers, of the referenced X/Open DCE RPC Specification (the only currently supported syntax is NDR).

- **stx_version**

  RPC syntax version number (4-byte integer, big/big-endian encoded). This field specifies the version number of the syntax identified in the **stx_id** field. Its registered values are specified in Appendix I, Protocol Identifiers, of the referenced X/Open DCE RPC Specification (the only currently supported version number of NDR is version 1).

- **pkl_version**

  Pickle header version number (1-byte integer, big-endian encoded). The only currently supported value is 0.

- **pkl_length_hi** and **pkl_length_low**

  The length (3-byte integer, big/big-endian encoded), in bytes, of the pickle body *B*. Its value is in the range [0, 2^{24}–1].

- **pkl_syntax**

  RPC syntax (20-byte **rpc_syntax_id_t**) of the encoded/marshalled data in the pickle body *B*. In the case of NDR version 1 (the only currently supported value), the pickle body *B* includes some metadata in addition to its actual encoded/marshalled data (this is specified in detail below).

- **pkl_type**

  Pickle type identifier (16-byte IDL-defined UUID (**uuid_t**), encoded in big/big endian format — see below). Since this field represents a UUID, it can also be specified as a string (see

Appendix A, Universal Unique Identifier, of the referenced X/Open DCE RPC Specification). This field specifies the semantics of the data in the pickle body *B*.

Here, ''encoding a UUID in big/big-endian format'' means that the UUID's string representation (see Appendix A, Universal Unique Identifier, of the referenced X/Open DCE RPC Specification) is to have its hyphens removed, and the resulting 32-character string interpreted as a hexadecimal integer, which is then encoded in big/big-endian format. Thus, for example, the UUID (in string representation) 01234567-89ab-cdef-0123-456789abcdef is mapped to the byte vector <0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef, 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef> (where ''0x'' is the usual C-language hexadecimal prefix notation, and where each byte is bit-encoded in big-endian format). (This definition can also be formulated equivalently in terms of big/big-endian encodings of the individual fields of a **uuid_t** — see Appendix N, IDL Data Type Declarations, of the referenced X/Open DCE RPC Specification — but this is not done here.)

Note that the encoding/marshalling of the fields of **rpc_syntax_id_t** and of **idl_pkl_header_t** is included in their specifications above (namely, it is always big/big-endian, *not* NDR). It is further specified that the encoded *order* of the fields of **rpc_syntax_id_t** and of **idl_pkl_header_t** is the same order as the syntactic listing of the fields in their IDL structure definitions, above; and there are no ''padding'' bits. Thus, according to this encoding specification, the encoded/marshalled length of **idl_pkl_header_t** is 40 bytes, formatted as follows (using C-like ''dot notation'' for structure fields, and using subscripts for sizes in bytes):

$$\begin{aligned}&\texttt{<pkl\_version}_1\texttt{,}\\&\texttt{<pkl\_length\_hi}_1\texttt{, pkl\_length\_low}_2\texttt{>,}\\&\texttt{<pkl\_syntax.stx\_id}_{16}\texttt{, pkl\_syntax.stx\_version}_4\texttt{>,}\\&\texttt{pkl\_type}_{16}\texttt{>}\end{aligned}$$

PKL's body *B* depends on *PKL*'s syntax (that is, *PKL*'s syntax field, *H***.pkl_syntax**). The pickle bodies for the currently supported pickle syntaxes are specified as follows:

- *NDR version 1* (the only currently supported syntax)

  B has the form:

  $$B = \langle L, 0_4, D \rangle$$

  where (note that *L* and 0_4 are second-level metadata — only *D* is really application-level data):

  — *L* is a 4-byte *NDR format label* (IDL-defined and NDR-marshalled value of the **ndr_format_t** data type, as specified in Appendix N, IDL Data Type Declarations, of the referenced X/Open DCE RPC Specification). It is initialised to the appropriate system-specific values at pickle generation time.

  Note: The NDR format label (**ndr_format_t** data type, as defined in Appendix N, IDL Data Type Declarations, of the referenced X/Open DCE RPC Specification) is not to be confused with the actual (''on-the-wire'') NDR encoding information that flows in RPC calls (as specified in Section 14.1, Data Representations Format Label, of the referenced X/Open DCE RPC Specification). Both have the same semantics, but they have different syntaxes (format or ''physical'' layout). In particular, the endianness and character format fields of on-the-wire NDR encoding information occupy only 4 bits each, while in **ndr_format_t** they occupy 8 bits each.

  — 0_4 is a 4-byte zero vector (padding).

— *D* is an NDR-marshalled *datastream* (byte-vector), as specified in Chapter 14, Transfer Syntax NDR, of the referenced X/Open DCE RPC Specification, formatted according to the NDR format label *L*. The application-level semantics of this field are indicated by the pickle's type (*H*.**pkl_type**). It is normally an NDR-marshalled value of an IDL data type (though potentially it could be directly defined in terms of NDR, bypassing an IDL definition).

Note: Fully-fledged support for pickling (or ''encoding services''), as opposed to the ''hand-rolled'' pickles described in this section, is anticipated in a future revision of DCE.

2.2 CRC-32

This section specifies the ''non-cryptographic'' checksum mechanisms employed in this specification. Only one such type of mechanism, CRC-32, is currently supported, so this whole section is devoted to that.

Notes:

1. This section is based on, and (unless stated otherwise) is technically aligned with, CCITT V.42. However, for editorial reasons, this section stands independently, and no familiarity with that document is required. (Thus, the part of this chapter that duplicates information in the cited document is intended to be technically equivalent to that document, rewritten for the expository purposes of this document, and any technical discrepancies between the two are inadvertent and to be reconciled.)
2. CRC-32 is used in this specification primarily in Chapter 4 on page 159 (Kerberos, see Section 4.3.5.1 on page 188), and in Chapter 9 on page 3291 (Protected RPC). It also makes a minor appearance in Section 11.6.1.21 on page 400.

2.2.1 Cyclic Redundancy Checksums

Cyclic redundancy checksums (CRCs) are defined abstractly in terms of polynomials with modulus-2 coefficients (that is, in terms of the *polynomial ring* $\mathbf{F}_2[X]$ in the indeterminate X), as follows. (Note that in this abstract definition there is no need to specify any conventions regarding identifications (endianness) of bit-sequences with integers.)

For purposes of this definition, identify arbitrary non-empty bit-vectors V of length $k > 0$ with polynomials $V(X)$ regarded as having **highest power** k–1 via the following *big-endian* correspondence:

$$V = \langle v_0, \cdots, v_{k-1} \rangle \leftrightarrow V(X) = v_0 X^{k-1} + \cdots + v_{k-1} X^0$$

Here, $V(X)$ is regarded as having ''highest power'' k–1, though not necessarily ''degree'' k–1, since some of its leading coefficients $v_0, v_1, \cdots$, may be 0.

Now fix an integer $N > 0$ (for the purposes of DCE, N will always be 32), and fix a polynomial $G(X) \in \mathbf{F}_2[X]$ of degree exactly N (that is, the coefficient of X^N in G is non-zero). Then, for any $S(X) = s_0 X^{N-1} + \cdots + s_{N-1} X^0$ regarded as having highest power N–1, and any $M(X) = m_0 X^{K-1} + \cdots + m_{K-1} X^0$ regarded as having highest power K–1 ($K > 0$), there exist unique polynomials $Q(X)$ and $R(X)$ (depending on K, $S(X)$, $G(X)$ and $M(X)$), such that:

- $X^K S(X) + M(X) = G(X)Q(X) + R(X)$
- $\text{degree}(R(X)) < N$

This follows from the elementary technique of polynomial long division (using modular arithmetic with modulus 2 on the coefficients); that is, $Q(X)$ is the *quotient* and $R(X)$ is the *remainder* of the division of $X^K S(X) + M(X)$ by $G(X)$. Note that under the (big-endian) identification of bit-vectors with polynomials, the correspondence is as follows:

$$X^K S(X) + M(X) = s_0 X^{K+N-1} + \cdots + s_{N-1} X^K + m_0 X^{K-1} \cdots + m_{K-1} X^0 \leftrightarrow$$
$$\langle s_0, \cdots, s_{N-1}, m_0, \cdots, m_{K-1} \rangle$$

(That is, in terms of bit-vectors, the polynomial $X^K S(X) + M(X)$ corresponds to prepending (the bits of) $S(X)$ to (the bits of) $M(X)$.)

Now by definition, the N-**bit CRC** of an arbitrary non-empty ''bit-message'' M, with respect to the *generator* G and the *seed* (or *initialisation vector*) S (identifying bit-vectors and polynomials as

above, of course), is the N-bit vector $<r_0, \cdots, r_{N-1}>$ identified with the remainder polynomial $R(X)$ as described above. The notation for this CRC is:

$\mathrm{CRC}_G(S, M)$

In the common case of the seed S being 0 (that is, a 0-vector or 0-polynomial), the notation $\mathrm{CRC}_G(0, M)$ is sometimes simplified to:

$\mathrm{CRC}_G(M)$

With this notation, the following *CRC composition law* (or *chaining property*) is immediately seen to hold:

$\mathrm{CRC}_G(S, <M, M'>) = \mathrm{CRC}_G(\mathrm{CRC}_G(S, M), M')$

Now suppose further that the generator $G(X)$ is *irreducible* (that is, $G(X)$ cannot be expressed as a product of two polynomials of positive degree). Then N-bit CRCs based on $G(X)$ have the *error-detecting property* that any two distinct messages differing in at most a subsequence of N bits have distinct CRCs (it also detects ''most'' other errors, in a sense not described here). That property, while important for the data communications heritage of CRCs, is not significant for this chapter. Instead, for the purposes of this specification, the significant property met for CRCs based on irreducible generators is the ''probabilistic collision-resistance'' property mentioned previously. (Note that any given message can easily be modified — by appending a single 32-bit word to it — to yield any given CRC_G-checksum value, so that CRCs are not ''cryptographically collision-resistant''.)

Finally, a ''twisted'' version of CRCs is defined as follows. Let ''†'' denote the *bit-reflection* (or *bit-reversal*) operation, which is defined on arbitrary bit-vectors $V = <v_0, \cdots, v_{k-1}>$ by:

$V^\dagger =$
$<v_0, \cdots, v_{k-1}>^\dagger =$
$<v_{k-1}, \cdots, v_0>$

Likewise, let ''‡'' denote the *per-byte bit-reflection* operation, which is defined on bit-vectors M as follows. First, if M is not a *byte*-vector; that is, if the bit-length of M is not a positive multiple of 8, append the minimal number of zero-bits to it such that the resulting bit-vector (still denoted M, by abuse of notation) does have bit-length a positive multiple of 8. Then, define the per-byte bit-reflection of the resulting byte-vector M by:

$M^\ddagger =$
$<<m_0 \cdots, m_7>, \cdots, <m_{8l+0}, \cdots, m_{8l+7}>>^\ddagger =$
$<<m_0, \cdots, m_7>^\dagger, \cdots, <m_{8l+0}, \cdots, m_{8l+7}>^\dagger> =$
$<<m_7, \cdots, m_0>, \cdots, <m_{8l+7}, \cdots, m_{8l+0}>>$

Then with these notations, the *twisted CRC* corresponding to CRC_G, denoted $\mathrm{CRC}^\S_G$, is defined for bit-vectors M as follows:

$\mathrm{CRC}^\S_G(S, M) = (\mathrm{CRC}_G(S^\dagger, M^\ddagger))^\dagger$

If the seed S is 0, the notation $\mathrm{CRC}^\S_G(0, M)$ is sometimes simplified to:

$\mathrm{CRC}^\S_G(M)$

Note that twisted CRCs still satisfy the CRC chaining property (for byte-vectors M, M'):

$\mathrm{CRC}^\S_G(S, <M, M'>) = \mathrm{CRC}^\S_G(\mathrm{CRC}^\S_G(S, M), M')$

Note: The twisting convention is related to the data communications heritage of CRCs: bytes in computer memories are often communicated across serial data lines least-significant-bit-first. That is, while ''in-core'' bytes are invariably interpreted as big-endian, they are often (but not always) twisted to become little-endian bytes ''on-

the-wire''.

The specific irreducible generating polynomials, $G(X)$, currently registered in DCE are collected in Section 2.2.1.1.

2.2.1.1 Registered CRCs

The only CRC currently used in DCE arises by taking $G(X)$ to be the following specific choice of irreducible generating polynomial (of degree $N = 32$), said to be the *CCITT-32* (or *ITU-T*, *OSI/IEC*, *Autodin-II*, *Ethernet*, *FDDI*, *PKZip*, and so on) *polynomial*:

$$G_{CCITT\text{-}32}(X) =$$

$$X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X^1 + X^0$$

If the X^{32}-term is ignored (or rather, considered to be implicitly present, since the X^{32}-term of a 32-degree generating polynomial must always be present), this polynomial corresponds, according to the (big-endian) identification of polynomials with bit-vectors, to the bit-vector:

$$G_{CCITT\text{-}32}(X) =$$

<0,0,0,0, 0,1,0,0, 1,1,0,0, 0,0,0,1, 0,0,0,1, 1,1,0,1, 1,0,1,1, 0,1,1,1>

Note: This bit-vector in turn corresponds to various integers under the various identifications of bit-vectors with integers. For example, under the big/big-endian identification it corresponds to the integer 0x04c11db7 = 79764919, and under the little/little-endian identification it corresponds to 0xedb88320 = 3988292384. These integer representations are of no interest for the purposes of this specification, though they may be of some use for particular implementations.

This CCITT-32 CRC, being the *only* CRC used in DCE, will henceforth be denoted simply, without fear of confusion: (''*the*'') *CRC-32* (this is perhaps a slight abuse of terminology, though a commonly accepted one, because ''CRC-32'' is sometimes also used as a generic term meaning ''an N-bit CRC where $N = 32$''). Accordingly, this CCITT-32 CRC and its twisted version (which is the version actually used in Chapter 9) will henceforth be denoted simply:

$$CRC_{32}(S, M)$$

$$CRC^{\S}_{32}(S, M)$$

If $S = 0$, the notation is sometimes simplified to:

$$CRC_{32}(M)$$

$$CRC^{\S}_{32}(M)$$

2.3 MD4

This section specifies MD4, one of the ''cryptographic'' checksum mechanisms employed in this specification.

Note: This section is based on, and (unless stated otherwise) is technically aligned with, the Internet document RFC 1320, by R. Rivest, dated April 1992. However, for editorial reasons, this section stands independently, and no familiarity with that document is required. (Thus, the part of this section that duplicates information in the cited document is intended to be technically equivalent to that document, rewritten for the expository purposes of this document, and any technical discrepancies between the two are inadvertent and to be reconciled.)

MD4 is described by an algorithm that takes as input a bit-message *M* of arbitrary length and produces as output a 128-bit *message digest* (or *hash*, *checksum*, *checksumtext*, *fingerprint*) of *M*. MD4 is a *non-invertible* (''*one-way*'') *function*, and it is claimed that it has the cryptographic property of being *collision-resistant* (or *collision-*''*proof*''): it is computationally infeasible to exhibit (that is, to produce, generate or construct) two distinct messages having the same message digest as one another, or to exhibit a single message having a given prespecified message digest. More precisely, it is claimed that the ''difficulty'' (computational complexity) of exhibiting two distinct messages having the same message digest has average order $O(\frac{1}{2}\cdot 2^{64})$, and that the difficulty of exhibiting a single message having a given message digest has average order $O(\frac{1}{2}\cdot 2^{128})$.

Let $M = <m_0, \cdots, m_{n-1}>$ be an arbitrarily given bit-message (sequence) of length *n* bits. Here *n* is an arbitrary non-negative integer (it may be 0, it need not be congruent to 0 (mod 8), and it may be arbitrarily large). Then the algorithm below is performed to compute the message digest of the message *M*, which is denoted by MD4(*M*).

For the remainder of this section, the little/big-endian identification of integers and 32-bit vectors is used (as defined in Section 2.1.4.3 on page 130).

2.3.1 Some Special Functions

The following four special functions are defined for use in the MD4 algorithm. Let *U*, *V*, *W* be 32-bit vectors (or, what amounts to the same given the fixing of the little/big endian correspondence in the remainder of this section, integers in the range $[0, 2^{32}-1]$).

- $F(U, V, W) = (U \,\&\, V) \mid ((\sim U) \,\&\, W)$
- $G(U, V, W) = (U \,\&\, V) \mid (V \,\&\, W) \mid (W \,\&\, U)$
- $H(U, V, W) = U \,\hat{}\, V \,\hat{}\, W$

Note: It is interesting (from a cryptographic design perspective) to note that various observations can be made about these functions (even though these observations are quite unnecessary from a DCE conformance point of view). For example, it may be noted that *F* acts as the conditional: ''if *U* then *V* else *W*'' (or in C-language notation, ''*U*?*V*:*W*''). The function *F* could have been defined using ''addition (mod 2^{32})'' instead of ''|'', since ''*U* & *V*'' and ''(~*U*) & *W*'' will never have 1s in the same bit position. In each bit position, *G* acts as the *majority function*: if at least two of (the bits of) *U*, *V*, and *W* are set, then the corresponding bit of *G*(*U*, *V*, *W*) is set, otherwise it is reset. If the bits of *U*, *V* and *W* are independent and unbiased in the statistical sense, then each bit of *F*(*U*, *V*, *W*) will be independent and unbiased. The functions *G* and *H* are similar to the function *F*, in that they act in ''bitwise parallel'' to produce their output from the bits of *U*, *V* and *W*, in such a manner that if the corresponding bits of *U*, *V* and *W* are independent and unbiased, then each bit of *G*(*U*, *V*, *W*) and

$H(U, V, W)$ will be independent and unbiased. The function H is just the *parity function* of its inputs (that is, each bit position of its output is the bit sum (mod 2) of its inputs).

2.3.2 Append Padding Bits

The message M is *padded* (appended to), to produce a new bit-message of bit-length $n' = \lambda(M') > n$ and $n' \equiv 448 \pmod{512}$ (the meaning of the ''magic number'' 448 is merely that 448 + 64 = 512 — see Section 2.3.3 on the significance of the number 64):

```
M' = <M, 1, <0, ···, 0>>;
```

Padding is always performed, even if the length of the original message M is already congruent to 448 (mod 512). It is performed as follows: a single ''1'' bit is appended to the message (on the right), and then a *0-vector* (one consisting entirely of (zero or more) ''0'' bits) is appended, so that the length in bits of the padded message becomes congruent to 448 (mod 512). In all, at least 1 bit and at most 512 bits are appended.

2.3.3 Append Length

The padded message M' is appended with the 64-bit representation of $n \pmod{2^{64}}$, to produce the new bit-message of bit-length $n'' = \lambda(M'') = n' + 64$:

```
M'' = <M', n(mod 2^64)>;
```

Namely, write $n \pmod{2^{64}}$ as an integer, as described above, and then append the 8 bytes of this longword (in little/big-endian order) to the end of the padded message M' from the previous step.

The result is now $n'' \equiv 0 \pmod{512}$. As described above, M'' can also be viewed as a sequence of *words*: let $M''[0], \cdots, M''[N-1]$ denote these words, where $N = n''/32 \equiv 0 \pmod{16}$.

Note: This step has the theoretical limitation that it does not distinguish between messages that differ in length by multiples of 2^{64}. In practice this is not a significant limitation (in particular, it does not pose a significant security threat), for two reasons:

1. ''For all practical purposes'', it may be assumed that *all* messages have length less than 2^{64}. This is because the length of time it would take to merely transmit (over RPC or any other medium) a message of length 2^{64} bits (much less compute its MD4 checksum) is impractically large. Indeed, at a transmission rate of one gigabit (10^9 bits) per second, it would take ~584½ years just to transmit such a message. Adding to this transmission time any actual local processing time on the data (such as ''generating'' it on the sending side, or ''interpreting'' it on the receiving side) reduces to insignificance the practical uses of such very large messages (in the present era of computing).
2. In the actual usage of MD4 as specified in DCE (namely, in Protected RPC), all messages actually protected with MD4 have length $< 2^{64}$ (see Chapter 9, and the referenced X/Open DCE RPC Specification).

2.3.4 Initialise State Buffer and Trigonometric Vector

A quadword (= 128 bits) (*external*) *state buffer*, denoted <*A´*, *B´*, *C´*, *D´*>, is used to describe the pseudocode computation below. Here, each of *A´*, *B´*, *C´*, *D´* is a word-sized variable. These variables are initialised to the following *initialisation values* or *seeds* (expressed as integers in C-like hexadecimal notation, with the corresponding little/big-endian byte-sequences in comments):

```
A' = 0x67452301; /* = <0x01,0x23,0x45,0x67> */
B' = 0xefcdab89; /* = <0x89,0xab,0xcd,0xef> */
C' = 0x98badcfe; /* = <0xfe,0xdc,0xba,0x98> */
D' = 0x10325476; /* = <0x76,0x54,0x32,0x10> */
```

The following array of 2 ''quadratic'' (unsigned) words, *Q*[2] and *Q*[3] is further initialised:

```
Q[2] = 0x5a827999;
Q[3] = 0x6ed9eba1;
```

Note: The origin of the name ''quadratic'' comes from the fact that $Q[i] = [2^{32} \cdot \mathrm{sqrt}(i)]$, where on the right hand side ''[···]'' denotes the integral part of a real number, and ''sqrt'' is the usual square root function. The values *Q*[2] and *Q*[3] are chosen for use in the MD4 algorithm because the bits of these 2 values are statistically well-distributed for this application.

2.3.5 Compress Message in 16-word Chunks

Perform the following algorithm (expressed in pseudocode).

Its *outer loop* processes the padded, appended message *M´´* in chunks of 16 words (= 512 bits) as follows:

```
for (i = 0; i <= N/16 - 1; i += 1) {
        <A',B',C',D'> =
            COMPRESS(A',B',C',D',M''[16*i+0],···,M''[16*i+15]);
}
```

Its inner *compression function*, denoted COMPRESS(*A´*, *B´*, *C´*, *D´*, *R*[0], ···, *R*[15]), takes as input a 128-bit *state vector* <*A´*, *B´*, *C´*, *D´*> and a 512-bit message chunk <*R*[0], ···, *R*[15]>, and produces as output a 128-bit state vector, say <*A´´*, *B´´*, *C´´*, *D´´*>. It is defined by the following algorithm (consisting of 3 *rounds* of 16 compression operations each), where ''+'' denotes (''unsigned'') addition (mod 2^{32}), and where *F*, *G* and *H* are the special functions defined in Section 2.3.1 on page 139:

```
/* Initialise internal state buffer <A,B,C,D> = <A',B',C',D'>. */
A = A'; B = B'; C = C'; D = D';

/* Round 1 -- Let "FF{abcd,r,s}" denote the F-compression statement
           a = (a + F(b,c,d) + R[r] <<< s
   and invoke it 16 times as follows: */
FF{ABCD, 0, 3}; FF{DABC, 1, 7}; FF{CDAB, 2,11}; FF{BCDA, 3,19};
FF{ABCD, 4, 3}; FF{DABC, 5, 7}; FF{CDAB, 6,11}; FF{BCDA, 7,19};
FF{ABCD, 8, 3}; FF{DABC, 9, 7}; FF{CDAB,10,11}; FF{BCDA,11,19};
FF{ABCD,12, 3}; FF{DABC,13, 7}; FF{CDAB,14,11}; FF{BCDA,15,19};

/* Round 2 -- Let "GG{abcd,r,s}" denote the G-compression statement
           a = (a + G(b,c,d) + R[r] + Q[2]) <<< s
   and invoke it 16 times as follows: */
GG{ABCD, 0, 3}; GG{DABC, 4, 5}; GG{CDAB, 8, 9}; GG{BCDA,12,13};
GG{ABCD, 1, 3}; GG{DABC, 5, 5}; GG{CDAB, 9, 9}; GG{BCDA,13,13};
GG{ABCD, 2, 3}; GG{DABC, 6, 5}; GG{CDAB,10, 9}; GG{BCDA,14,13};
GG{ABCD, 3, 3}; GG{DABC, 7, 5}; GG{CDAB,11, 9}; GG{BCDA,15,13};

/* Round 3 -- Let "HH{abcd,r,s,t}" denote the H-compression statement
           a = (a + H(b,c,d) + R[r] + Q[3]) <<< s
   and invoke it 16 times as follows: */
HH{ABCD, 0, 3}; HH{DABC, 8, 9}; HH{CDAB, 4,11}; HH{BCDA,12,15};
HH{ABCD, 2, 3}; HH{DABC,10, 9}; HH{CDAB, 6,11}; HH{BCDA,14,15};
HH{ABCD, 1, 3}; HH{DABC, 9, 9}; HH{CDAB, 5,11}; HH{BCDA,13,15};
HH{ABCD, 3, 3}; HH{DABC,11, 9}; HH{CDAB, 7,11}; HH{BCDA,15,15};

/* Output state <A'',B'',C'',D''> = <A',B',C',D'> + <A,B,C,D>. */
A'' = A'+A; B'' = B'+B; C'' = C'+C; D'' = D'+D;
```

2.3.6 Output

Finally, the message digest, MD4(*M*), is defined to be the final quadword state resulting from the above algorithm, after the outer loop completes (it begins with the low-order byte of *A´*, and ends with the high-order byte of *D´*):

```
MD4(M) = <A', B', C', D'>;
```

This completes the description of MD4.

2.4 MD5

This section specifies MD5, one of the ''cryptographic'' checksum mechanisms employed in this specification.

Note: This section is based on, and (unless stated otherwise) is technically aligned with, the Internet document RFC 1321, by R. Rivest, dated April 1992. However, for editorial reasons, this section stands independently, and no familiarity with that document is required. (Thus, the part of this section that duplicates information in the cited document is intended to be technically equivalent to that document, rewritten for the expository purposes of this document, and any technical discrepancies between the two are inadvertent and to be reconciled.)

MD5 is described by an algorithm that takes as input a bit-message M of arbitrary length and produces as output a 128-bit *message digest* (or *hash*, *checksum*, *checksumtext*, *fingerprint*) of M. MD5 is a *non-invertible* (''*one-way*'') *function*, and it is claimed that it has the cryptographic property of being *collision-resistant* (or *collision-*''*proof*''): it is computationally infeasible to exhibit (that is, to produce, generate or construct) two distinct messages having the same message digest as one another, or to exhibit a single message having a given prespecified message digest. More precisely, it is claimed that the ''difficulty'' (computational complexity) of exhibiting two distinct messages having the same message digest has average order $O(\frac{1}{2}\cdot 2^{64})$, and that the difficulty of exhibiting a single message having a given message digest has average order $O(\frac{1}{2}\cdot 2^{128})$.

Let $M = <m_0, \cdots, m_{n-1}>$ be an arbitrarily given bit-message (sequence) of length n bits. Here n is an arbitrary non-negative integer (it may be 0, it need not be congruent to 0 (mod 8), and it may be arbitrarily large). Then the algorithm below is performed to compute the message digest of the message M, which is denoted by MD5(M).

For the remainder of this section, the little/big-endian identification of integers and 32-bit vectors is used (as defined in Section 2.1.4.3 on page 130).

2.4.1 Some Special Functions

The following four special functions are defined for use in the MD5 algorithm. Let U, V, W be 32-bit vectors (or, what amounts to the same given the fixing of the little/big endian correspondence in the remainder of this section, integers in the range $[0, 2^{32}-1]$).

- $F(U, V, W)$: as defined in Section 2.3.1 on page 139.
- $G(U, V, W) = (U \,\&\, W) \mid (V \,\&\, (\tilde{}\, W))$
- $H(U, V, W)$: as defined in Section 2.3.1 on page 139.
- $I(U, V, W) = V \,\hat{}\, (U \mid (\tilde{}\, W))$

Note: As with the functions E and H (see Section 2.3.1 on page 139), the functions G and I act in ''bitwise parallel''.

2.4.2 Append Padding Bits

The message M is *padded* (appended to), to produce a new bit-message of bit-length $n' = \lambda(M') > n$ and $n' \equiv 448 \pmod{512}$. This is performed as described in Section 2.3.2 on page 140.

2.4.3 Append Length

The padded message M' is appended with the 64-bit representation of $n \pmod{2^{64}}$, to produce the new bit-message of bit-length $n'' = \lambda(M'') = n' + 64$. This is performed as described in Section 2.3.3 on page 140.

Note: In the actual usage of MD5 as specified in DCE (namely, in Protected RPC), all messages actually protected with MD5 have length $< 2^{64}$ (see Chapter 9, and the referenced X/Open DCE RPC Specification).

2.4.4 Initialise State Buffer and Trigonometric Vector

A quadword (= 128 bits) (*external*) *state buffer*, denoted $<A', B', C', D'>$, is used to describe the pseudocode computation below. Here, each of A', B', C', D' is a word-sized variable. These variables are initialised to the same *initialisation values* or *seeds* as described in Section 2.3.4 on page 141.

The following array of 64 ''trigonometric'' (unsigned) words, $T[0], \cdots, T[63]$ is further initialised:

```
T[] = {0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee,
       0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501,
       0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be,
       0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821,
       0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa,
       0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8,
       0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed,
       0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a,
       0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c,
       0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70,
       0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05,
       0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665,
       0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039,
       0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1,
       0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1,
       0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391};
```

Note: The origin of the name ''trigonometric'' comes from the fact that $T[i] = [2^{32} \mid \sin(i+1) \mid]$, where on the right hand side ''$[\cdots]$'' denotes the integral part of a real number, ''$\mid \cdots \mid$'' denotes absolute value, and ''sin'' is the usual trigonometric sine function, whose arguments (''angles'') $i+1$ are measured in radians. The values $T[0], \cdots, T[63]$ are chosen for use in the MD5 algorithm because the bits of these 64 values are statistically well-distributed for this application.

2.4.5 Compress Message in 16-word Chunks

Perform the following algorithm (expressed in pseudocode).

Its *outer loop* processes the padded, appended message *M´´* in chunks of 16 words (= 512 bits) as follows:

```
for (i = 0; i <= N/16 - 1; i += 1) {
        <A',B',C',D'> =
            COMPRESS(A',B',C',D',M''[16*i+0],…,M''[16*i+15]);
}
```

Its inner *compression function*, denoted COMPRESS(*A´*, *B´*, *C´*, *D´*, *R*[0], ···, *R*[15]), takes as input a 128-bit *state vector* <*A´*, *B´*, *C´*, *D´*> and a 512-bit message chunk <*R*[0], ···, *R*[15]>, and produces as output a 128-bit state vector, say <*A´´*, *B´´*, *C´´*, *D´´*>. It is defined by the following algorithm (consisting of 4 *rounds* of 16 compression operations each), where ''+'' denotes (''unsigned'') addition (mod 2^{32}), and where *F*, *G*, *H* and *I* are the special functions defined in Section 2.4.1 on page 143:

```
/* Initialise internal state buffer <A,B,C,D> = <A',B',C',D'>. */
A = A'; B = B'; C = C'; D = D';

/* Round 1 -- Let "FF{abcd,r,s,t}" denote the F-compression statement
           a = b + ((a + F(b,c,d) + R[r] + T[t]) <<< s)
   and invoke it 16 times as follows: */
FF{ABCD, 0, 7, 0}; FF{DABC, 1,12, 1}; FF{CDAB, 2,17, 2}; FF{BCDA, 3,22, 3};
FF{ABCD, 4, 7, 4}; FF{DABC, 5,12, 5}; FF{CDAB, 6,17, 6}; FF{BCDA, 7,22, 7};
FF{ABCD, 8, 7, 8}; FF{DABC, 9,12, 9}; FF{CDAB,10,17,10}; FF{BCDA,11,22,11};
FF{ABCD,12, 7,12}; FF{DABC,13,12,13}; FF{CDAB,14,17,14}; FF{BCDA,15,22,15};

/* Round 2 -- Let "GG{abcd,r,s,t}" denote the G-compression statement
           a = b + ((a + G(b,c,d) + R[r] + T[t]) <<< s)
   and invoke it 16 times as follows: */
GG{ABCD, 1, 5,16}; GG{DABC, 6, 9,17}; GG{CDAB,11,14,18}; GG{BCDA, 0,20,19};
GG{ABCD, 5, 5,20}; GG{DABC,10, 9,21}; GG{CDAB,15,14,22}; GG{BCDA, 4,20,23};
GG{ABCD, 9, 5,24}; GG{DABC,14, 9,25}; GG{CDAB, 3,14,26}; GG{BCDA, 8,20,27};
GG{ABCD,13, 5,28}; GG{DABC, 2, 9,29}; GG{CDAB, 7,14,30}; GG{BCDA,12,20,31};

/* Round 3 -- Let "HH{abcd,r,s,t}" denote the H-compression statement
           a = b + ((a + H(b,c,d) + R[r] + T[t]) <<< s)
   and invoke it 16 times as follows: */
HH{ABCD, 5, 4,32}; HH{DABC, 8,11,33}; HH{CDAB,11,16,34}; HH{BCDA,14,23,35};
HH{ABCD, 1, 4,36}; HH{DABC, 4,11,37}; HH{CDAB, 7,16,38}; HH{BCDA,10,23,39};
HH{ABCD,13, 4,40}; HH{DABC, 0,11,41}; HH{CDAB, 3,16,42}; HH{BCDA, 6,23,43};
HH{ABCD, 9, 4,44}; HH{DABC,12,11,45}; HH{CDAB,15,16,46}; HH{BCDA, 2,23,47};

/* Round 4 -- Let "II{abcd,r,s,t}" denote the I-compression statement
           a = b + ((a + I(b,c,d) + R[r] + T[t]) <<< s)
   and invoke it 16 times as follows: */
II{ABCD, 0, 6,48}; II{DABC, 7,10,49}; II{CDAB,14,15,50}; II{BCDA, 5,21,51};
II{ABCD,12, 6,52}; II{DABC, 3,10,53}; II{CDAB,10,15,54}; II{BCDA, 1,21,55};
II{ABCD, 8, 6,56}; II{DABC,15,10,57}; II{CDAB, 6,15,58}; II{BCDA,13,21,59};
II{ABCD, 4, 6,60}; II{DABC,11,10,61}; II{CDAB, 2,15,62}; II{BCDA, 9,21,63};

/* Output state <A'',B'',C'',D''> = <A',B',C',D'> + <A,B,C,D>. */
A'' = A'+A; B'' = B'+B; C'' = C'+C; D'' = D'+D;
```

2.4.6 Output

Finally, the message digest, MD5(*M*), is defined to be the final quadword state resulting from the above algorithm, after the outer loop completes (it begins with the low-order byte of *A´*, and ends with the high-order byte of *D´*):

```
MD5(M) = <A', B', C', D'>;
```

This completes the description of MD5.

Chapter 3

Encryption/Decryption Mechanisms

This chapter specifies the (cryptographic) encryption/decryption mechanisms supported by DCE. Currently, only one such mechanism is supported, namely the *Data Encryption Standard* (DES) (or *Data Encryption Algorithm* (DEA)), and its *Cipher Block Chaining* (*CBC*) *Mode of Operation*, so this whole chapter is devoted to that.

Note: This chapter is based on, and (unless stated otherwise) is technically aligned with, ANSI X3.92 and ANSI X3.106. However, for editorial reasons, this chapter stands independently, and no familiarity with those documents is required. (Thus, the part of this chapter that duplicates information in the cited documents is intended to be technically equivalent to those documents, rewritten for the expository purposes of DCE, and any technical discrepancies between the two are inadvertent and to be reconciled.)

3.1 Basic DES

In the context of DES, a bit-vector of length 64 to be encrypted or decrypted is called a (*DES*) *block*. For terminology, notation and conventions regarding sequences of bits, see Chapter 2. In particular, when *bit*-sequences (especially, bytes) are interpreted as integers in this chapter, the *big-endian* mapping is always used. (This chapter does not interpret *byte*-sequences as integers, so their endianness should be considered.)

The ''basic DES'' algorithm (specified in detail in Section 3.5 on page 154) is an *encryption* mechanism, parameterised by a 64-bit vector, *K*, taking single-block plaintext inputs, *P*, and producing single-block ciphertext outputs, *Q*, for which the following notation is adopted:

$Q = \mathrm{DES}(K, P)$

The corresponding (inverse) *decryption* mechanism is denoted:

$P = \mathrm{DES}^{-1}(K, Q)$

The bit vector $K = \langle k_0, \cdots, k_{63} \rangle$, while nominally 64 bits long, has only 56 *active* (that is, ''cryptographically significant'') bits, in the sense that while the algorithm defining DES makes sense for an arbitrary 64-bit vector *K*, the algorithm ignores *K*'s 8 *passive* bits k_j, $j \equiv 7 \pmod 8$, which are the *low-order* bits (not the high-order bits, notably) of the 8 bytes of *K*; similarly for the algorithm defining DES^{-1}. In other words, the DES algorithm makes active use of only 56 of *K*'s bits, namely the 56 bits k_j for $j \not\equiv 7 \pmod 8$. By definition, a 64-bit vector *K* is said to be a *DES key* if it has *odd parity*; that is, if every byte of *K*, $\langle k_{8j+0}, \cdots, k_{8j+7} \rangle$ $(0 \le k \le 7)$, has odd bit sum (mod 2) — $k_{8j+0} + \cdots + k_{8j+7} \equiv 1 \pmod 2$. Obviously, the 8 passive bits of an arbitrary 64-bit vector *K* can always be uniquely chosen such that the resulting 64-bit vector, *K´* say, has odd parity (in this context, *K*'s passive bits are also called its *parity bits*); of course, as observed above, DES(*K*, *P*) = DES(*K´*, *P*) for all plaintexts *P*. *K´* is then called the (unique) (*odd-parity*) *normal form* of *K*. When explicit DES keys are written down in DCE, they are always expressed as 8-byte vectors (never as integers) in their odd-parity normal form.

Note: As a practical matter, note that even though the DES algorithm itself defining DES(*K*, *P*) makes sense for an arbitrary 64-bit vector *K*, some implementations check for parity, rejecting any key that is not a DES key (that is, does not have odd parity).

3.2 CBC Mode

The CBC mode of DES (specified in detail in Section 3.6 on page 158) is an extension of the basic DES algorithm, for the purpose of encrypting and decrypting bit-sequences of length a (strictly) positive multiple of 64 (that is, a positive number of blocks). It is parameterised not only by the key *K*, but also by a 64-bit *initialisation vector* (or *seed*), *IV* (all of whose bits are ''active''). DES CBC *encryption* is denoted by:

Q = DES-CBC(K, IV, P)

and its corresponding *decryption* by:

P = DES-CBC^{-1}(K, IV, Q)

The length of *Q* is the same as that of *P*.

Note: As seen in Section 3.6 on page 158, the role of the initialisation vector is to ''initialise'' the CBC algorithm, by using it to ''scramble'' the first block of plaintext. There are two common ways to use initialisation vectors, both of which are actually used in this specification:

1. The first common usage is as *confounders*. In this usage, initialisation vectors are chosen randomly, so that knowledge of common patterns that are present in the first blocks of many plaintexts (that is, ''known plaintexts'', such as occur in standard headers of protocol data units) cannot potentially be used in a cryptanalytic attack. Thus, in this usage, the initialisation vector improves the CBC algorithm's ''ergodicity'' (but does not increase the size of the DES key space). For examples of the usage of initialisation vectors as confounders, see Section 4.3.4.1 on page 185 and Section 4.3.5.1 on page 188.
2. The second common usage is as *links of chains*, whereby complex messages can be encrypted in pieces, yet the end result is the same as if the whole message had been encrypted monolithically, as a consequence of the chaining property stated in Section 3.3.1 on page 150. For examples of the usage of initialisation vectors as links in chains, see Section 9.2.2.3 on page 336 and Section 9.3.2.3 on page 342.

X3.106 itself does not specify any standard way to encrypt/decrypt plaintext/ciphertext of length other than a positive multiple of 64 (bits), merely stating that ''the final partial data block should be encrypted in a manner specified for the application''. Accordingly, DCE adopts the following convention. The meaning of (*DES*) *padding* is a bit-vector *R* of minimal length appended to a given bit vector *M* in order to bring the total bit-length of $<M, R>$ up to a positive multiple of 64. Note that the bit-length of *R*, $\lambda(R)$, is either 64 (in the case $\lambda(M) = 0$) or $(-\lambda(M))(\text{mod } 64)$ (in the case $\lambda(M) > 0$). In DCE, the DES padding required in a given situation will usually be explicitly specified. Nevertheless, it is convenient to also specify a default padding vector. *Therefore, unless stated otherwise*, DCE takes the (DES) padding vector *R* be a 0-vector of the appropriate length. Notationally, if *P* is a plaintext (or *Q* is a ciphertext) of length other than a positive multiple of 64, the following is defined:

- DES-CBC(K, IV, P) = DES-CBC(K, IV, $<P, <0, \cdots, 0>>$)
- DES-CBC^{-1}(K, IV, Q) = DES-CBC^{-1}(K, IV, $<Q, <0, \cdots, 0>>$)

When the initialisation vector *IV* is the (64-bit) *0-vector* $<0, \cdots, 0>$ (consisting entirely of ''0'' bits), the term *IV* is sometimes omitted from the above notations:

- DES-CBC(K, P) = DES-CBC(K, $<0, \cdots, 0>$, P)

- DES-CBC^{-1}(K, Q) = DES-CBC^{-1}(K, <0, ···, 0>, Q)

3.3 DES-CBC Checksum

The *DES-CBC checksum* of a plaintext *P*, with respect to a key *K* and an initialisation vector *IV*, is by definition the *final block* of DES-CBC(*K*, *IV*, *P*). It is denoted:

DES-CBC-CKSUM(*K*, *IV*, *P*)

It differs from MD4 and MD5(*P*) in being key- (and initialisation vector-)based, and in being only 8 bytes long instead of 16. When *IV* is a 0-vector, it is sometimes omitted from the notation:

DES-CBC-CKSUM(*K*, *P*)

3.3.1 Composition Laws (Chaining Properties)

The following *DES composition laws* or *chaining properties* are immediately seen to follow from the detailed description of the CBC mode algorithm given in Section 3.6 on page 158. For vectors of blocks *P* and *P´* of positive length:

- DES-CBC(*K*, *IV*, <*P*, *P´*>) =
  <DES-CBC(*K*, *IV*, *P*), DES-CBC(*K*, DES-CBC-CKSUM(*K*, *IV*, *P*), *P´*)>
- DES-CBC-CKSUM(*K*, *IV*, <*P*, *P´*>) =
  DES-CBC-CKSUM(*K*, DES-CBC-CKSUM(*K*, *IV*, *P*), *P´*)

3.4 Keys to be Avoided

There are certain DES keys (64 of them, in normal form) that ''should be'' avoided in any use of DES encryption, because they have poor cryptographic characteristics. Some of these are called *weak* keys, some are called *semi-weak* keys, and there are some others with no generally accepted moniker but which are called here *possibly weak* keys (rigorous definitions are given below). The complete list of such keys are listed below.

Note: To say that the keys in question ''should be'' avoided means, for the purposes of DCE, that it is *recommended* (though not required) that implementations *not generate* these keys as session, conversation or long-term keys; in particular, implementations should reject passwords that map to these keys (via the password-to-key mappings specified in Section 4.3.6.1 on page 190). However, all implementations are required to *accept all* keys generated by other implementations (for interoperability with implementations that *do* generate the keys that should be avoided).

There is no mention of these keys in ANSI X3.92 (though they have been noted elsewhere in the literature).

The rigorous definition of key weakness is as follows. As seen in the remainder of this chapter, the cryptographic strength of DES depends on the ''complexity'' (measured by the involvement of the S-boxes — see Section 3.5.3 on page 155) of the algorithm defining it. That algorithm involves, among other things, manipulating the key, K, via the *key schedule* subalgorithm, to generate 16 48-bit *subkeys*, $K_0, \cdots, K_{15}$. Define the *strength* of K, denoted $\sigma(K)$, to be the cardinality (number of elements) of this set of subkeys. If $\sigma(K) \geq \sigma(K')$, then the DES algorithm for K is more complex than that for K', so K is said to be *stronger* than K'. Obviously $1 \leq \sigma(K) \leq 16$, and the *strongest* keys are those for which $\sigma(K) = 16$; that is, those for which the subkeys K_i are all *distinct*, because that maximises the overall complexity of the DES algorithm. Using this measure of strength, the rigorous definitions of weak, semi-weak and possibly weak keys are that $\sigma(K) = 1$, 2 and 4, respectively.

Note that the weak and semi-weak keys have the following characteristics. The weak keys K are those for which encryption is the same as decryption: $\text{DES}(K, B) = \text{DES}^{-1}(K, B)$ for every block B. The semi-weak keys K are those for which there exists another key $K' \neq K$ which gives identical encryption and decryption: $\text{DES}(K, B) = \text{DES}(K', B)$ and $\text{DES}^{-1}(K, B) = \text{DES}^{-1}(K', B)$ for all blocks B — hence, K' can decrypt any message encrypted by K, and *vice versa*.

3.4.1 Weak Keys

The following are the 4 (normal form) weak keys:

```
<0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x01>
<0x1f,0x1f,0x1f,0x1f,0x0e,0x0e,0x0e,0x0e>
<0xe0,0xe0,0xe0,0xe0,0xf1,0xf1,0xf1,0xf1>
<0xfe,0xfe,0xfe,0xfe,0xfe,0xfe,0xfe,0xfe>
```

3.4.2 Semi-weak Keys

The following are the 12 (normal form) semi-weak keys:

```
<0x01,0x1f,0x01,0x1f,0x01,0x0e,0x01,0x0e>
<0x01,0xe0,0x01,0xe0,0x01,0xf1,0x01,0xf1>
<0x01,0xfe,0x01,0xfe,0x01,0xfe,0x01,0xfe>
<0x1f,0x01,0x1f,0x01,0x0e,0x01,0x0e,0x01>
<0x1f,0xe0,0x1f,0xe0,0x0e,0xf1,0x0e,0xf1>
<0x1f,0xfe,0x1f,0xfe,0x0e,0xfe,0x0e,0xfe>
<0xe0,0x01,0xe0,0x01,0xf1,0x01,0xf1,0x01>
<0xe0,0x1f,0xe0,0x1f,0xf1,0x0e,0xf1,0x0e>
<0xe0,0xfe,0xe0,0xfe,0xf1,0xfe,0xf1,0xfe>
<0xfe,0x01,0xfe,0x01,0xfe,0x01,0xfe,0x01>
<0xfe,0x1f,0xfe,0x1f,0xfe,0x0e,0xfe,0x0e>
<0xfe,0xe0,0xfe,0xe0,0xfe,0xf1,0xfe,0xf1>
```

3.4.3 Possibly Weak Keys

The following are the 48 (normal form) possibly weak keys:

```
<0x01,0x01,0x1f,0x1f,0x01,0x01,0x0e,0x0e>
<0x01,0x01,0xe0,0xe0,0x01,0x01,0xf1,0xf1>
<0x01,0x01,0xfe,0xfe,0x01,0x01,0xfe,0xfe>
<0x01,0x1f,0x1f,0x01,0x01,0x0e,0x0e,0x01>
<0x01,0x1f,0xe0,0xfe,0x01,0x0e,0xf1,0xfe>
<0x01,0x1f,0xfe,0xe0,0x01,0x0e,0xfe,0xf1>
<0x01,0xe0,0x1f,0xfe,0x01,0xf1,0x0e,0xfe>
<0x01,0xe0,0xe0,0x01,0x01,0xf1,0xf1,0x01>
<0x01,0xe0,0xfe,0x1f,0x01,0xf1,0xfe,0x0e>
<0x01,0xfe,0x1f,0xe0,0x01,0xfe,0x0e,0xf1>
<0x01,0xfe,0xe0,0x1f,0x01,0xfe,0xf1,0x0e>
<0x01,0xfe,0xfe,0x01,0x01,0xfe,0xfe,0x01>
<0x1f,0x01,0x01,0x1f,0x0e,0x01,0x01,0x0e>
<0x1f,0x01,0xe0,0xfe,0x0e,0x01,0xf1,0xfe>
<0x1f,0x01,0xfe,0xe0,0x0e,0x01,0xfe,0xf1>
<0x1f,0x1f,0x01,0x01,0x0e,0x0e,0x01,0x01>
<0x1f,0x1f,0xe0,0xe0,0x0e,0x0e,0xf1,0xf1>
<0x1f,0x1f,0xfe,0xfe,0x0e,0x0e,0xfe,0xfe>
<0x1f,0xe0,0x01,0xfe,0x0e,0xf1,0x01,0xfe>
<0x1f,0xe0,0xe0,0x1f,0x0e,0xf1,0xf1,0x0e>
<0x1f,0xe0,0xfe,0x01,0x0e,0xf1,0xfe,0x01>
<0x1f,0xfe,0x01,0xe0,0x0e,0xfe,0x01,0xf1>
<0x1f,0xfe,0xe0,0x01,0x0e,0xfe,0xf1,0x01>
<0x1f,0xfe,0xfe,0x1f,0x0e,0xfe,0xfe,0x0e>
<0xe0,0x01,0x01,0xe0,0xf1,0x01,0x01,0xf1>
<0xe0,0x01,0x1f,0xfe,0xf1,0x01,0x0e,0xfe>
<0xe0,0x01,0xfe,0x1f,0xf1,0x01,0xfe,0x0e>
<0xe0,0x1f,0x01,0xfe,0xf1,0x0e,0x01,0xfe>
<0xe0,0x1f,0x1f,0xe0,0xf1,0x0e,0x0e,0xf1>
<0xe0,0x1f,0xfe,0x01,0xf1,0x0e,0xfe,0x01>
<0xe0,0xe0,0x01,0x01,0xf1,0xf1,0x01,0x01>
<0xe0,0xe0,0x1f,0x1f,0xf1,0xf1,0x0e,0x0e>
<0xe0,0xe0,0xfe,0xfe,0xf1,0xf1,0xfe,0xfe>
```

```
<0xe0,0xfe,0x01,0x1f,0xf1,0xfe,0x01,0x0e>
<0xe0,0xfe,0x1f,0x01,0xf1,0xfe,0x0e,0x01>
<0xe0,0xfe,0xfe,0xe0,0xf1,0xfe,0xfe,0xf1>
<0xfe,0x01,0x01,0xfe,0xfe,0x01,0x01,0xfe>
<0xfe,0x01,0x1f,0xe0,0xfe,0x01,0x0e,0xf1>
<0xfe,0x01,0xe0,0x1f,0xfe,0x01,0xf1,0x0e>
<0xfe,0x1f,0x01,0xe0,0xfe,0x0e,0x01,0xf1>
<0xfe,0x1f,0x1f,0xfe,0xfe,0x0e,0x0e,0xfe>
<0xfe,0x1f,0xe0,0x01,0xfe,0x0e,0xf1,0x01>
<0xfe,0xe0,0x01,0x1f,0xfe,0xf1,0x01,0x0e>
<0xfe,0xe0,0x1f,0x01,0xfe,0xf1,0x0e,0x01>
<0xfe,0xe0,0xe0,0xfe,0xfe,0xf1,0xf1,0xfe>
<0xfe,0xfe,0x01,0x01,0xfe,0xfe,0x01,0x01>
<0xfe,0xfe,0x1f,0x1f,0xfe,0xfe,0x0e,0x0e>
<0xfe,0xfe,0xe0,0xe0,0xfe,0xfe,0xf1,0xf1>
```

3.5 Details of Basic DES Algorithm

Let $K = \langle k_0, \cdots, k_{63} \rangle$ be a DES key, and let $P = \langle p_0, \cdots, p_{63} \rangle$ be a plaintext DES block. The value of DES(K, P) will now be described as a functional composition of 33 (invertible) transformations from blocks to blocks, as follows:

$$\mathrm{DES}(K, P) = (\mathrm{FP} \circ \mathrm{T}_{K,15} \circ \theta_{14} \circ \mathrm{T}_{K,14} \circ \theta_{13} \circ \cdots \circ \mathrm{T}_{K,1} \circ \theta_0 \circ \mathrm{T}_{K,0} \circ \mathrm{IP})(P)$$

Of these 33 transformations, 15 of them are equal to the simple *cyclic permutation* (or *transposition*), $\theta_i = \theta$ $(0 \le i \le 14)$, which *interchanges* the *left* and *right halfblocks* of a block: if $B = \langle \mathrm{L}_B, \mathrm{R}_B \rangle$ where L_B and R_B are 32-bit vectors, then:

$$\theta(B) = \langle \mathrm{R}_B, \mathrm{L}_B \rangle$$

That is:

$$\theta(\langle b_0, \cdots, b_{63} \rangle) = \langle b_{32}, \cdots, b_{63}, b_0, \cdots, b_{31} \rangle$$

3.5.1 Initial Permutation (IP) and Final Permutation (FP)

The *initial permutation*, IP, mapping blocks to blocks, is defined as follows:

$$\begin{aligned}\mathrm{IP}(\langle p_0, \cdots, p_{63} \rangle) = \langle & p_{57}, p_{49}, p_{41}, p_{33}, p_{25}, p_{17}, p_9, p_1, p_{59}, p_{51}, p_{43}, p_{35}, p_{27}, p_{19}, p_{11}, p_3, \\ & p_{61}, p_{53}, p_{45}, p_{37}, p_{29}, p_{21}, p_{13}, p_5, p_{63}, p_{55}, p_{47}, p_{39}, p_{31}, p_{23}, p_{15}, p_7, \\ & p_{56}, p_{48}, p_{40}, p_{32}, p_{24}, p_{16}, p_8, p_0, p_{58}, p_{50}, p_{42}, p_{34}, p_{26}, p_{18}, p_{10}, p_2, \\ & p_{60}, p_{52}, p_{44}, p_{36}, p_{28}, p_{20}, p_{12}, p_4, p_{62}, p_{54}, p_{46}, p_{38}, p_{30}, p_{22}, p_{14}, p_6 \rangle\end{aligned}$$

The *final permutation* (or *inverse initial permutation*), FP, is defined to be the inverse of IP, $\mathrm{FP} = \mathrm{IP}^{-1}$, and is therefore given by:

$$\begin{aligned}\mathrm{FP}(\langle q_0, \cdots, q_{63} \rangle) = \langle & q_{39}, q_7, q_{47}, q_{15}, q_{55}, q_{23}, q_{63}, q_{31}, q_{38}, q_6, q_{46}, q_{14}, q_{54}, q_{22}, q_{62}, q_{30}, \\ & q_{37}, q_5, q_{45}, q_{13}, q_{53}, q_{21}, q_{61}, q_{29}, q_{36}, q_4, q_{44}, q_{12}, q_{52}, q_{20}, q_{60}, q_{28}, \\ & q_{35}, q_3, q_{43}, q_{11}, q_{51}, q_{19}, q_{59}, q_{27}, q_{34}, q_2, q_{42}, q_{10}, q_{50}, q_{18}, q_{58}, q_{26}, \\ & q_{33}, q_1, q_{41}, q_9, q_{49}, q_{17}, q_{57}, q_{25}, q_{32}, q_0, q_{40}, q_8, q_{48}, q_{16}, q_{56}, q_{24} \rangle\end{aligned}$$

Note: As seen in Section 3.5.4 on page 157, IP and FP are the only two transformations among DES's 33 component transformations that are *not self-inverse*. This fact highlights the functional significance of IP and FP, but it does not explain the *cryptographic* significance of IP and FP — and indeed they appear to have no known cryptographic significance.

3.5.2 Key Schedule (KS): Permuted Choices (PC1, PC2) and Left Shift (LS)

The remaining transformations $\mathrm{T}_{K,i}$ depend on a *key schedule* of 16 48-bit *subkeys* $K_i = \mathrm{K}_{K,i} = \mathrm{KS}(K, i)$ $(0 \le i \le 15)$, which are defined as follows.

First, two 28-bit *auxiliary vectors*, $\mathrm{C}_{K,-1}$ and $\mathrm{D}_{K,-1}$, are defined by:

$$\langle \mathrm{C}_{K,-1}, \mathrm{D}_{K,-1} \rangle = \mathrm{PC1}(K)$$

where PC1 is the *first permuted choice* mapping from 64-bit (key) vectors to 56-bit (auxiliary) vectors, defined as follows:

$$PC1(<k_0, \cdots, k_{63}>) =$$
$$<k_{56}, k_{48}, k_{40}, k_{32}, k_{24}, k_{16}, k_8, k_0, k_{57}, k_{49}, k_{41}, k_{33}, k_{25}, k_{17},$$
$$k_9, k_1, k_{58}, k_{50}, k_{42}, k_{34}, k_{26}, k_{18}, k_{10}, k_2, k_{59}, k_{51}, k_{43}, k_{35},$$
$$k_{62}, k_{54}, k_{46}, k_{38}, k_{30}, k_{22}, k_{14}, k_6, k_{61}, k_{53}, k_{45}, k_{37}, k_{29}, k_{21},$$
$$k_{13}, k_5, k_{60}, k_{52}, k_{44}, k_{36}, k_{28}, k_{20}, k_{12}, k_4, k_{27}, k_{19}, k_{11}, k_3>$$

Note, in particular, that PC1 ''destroys'' the passive bits of K (and this shows why they do not figure into the cryptographic properties of DES).

Now, the remaining 28-bit auxiliary vectors, $C_{K,i}$ and $D_{K,i}$, and the subkeys $K_i = K_{K,i} = KS(K, i)$, are defined by:

- $C_{K,i} = LS_i(C_{K,i-1})$
- $D_{K,i} = LS_i(D_{K,i-1})$
- $K_i = K_{K,i} = KS(K, i) = PC2(<C_{K,i}, D_{K,i}>)$

where LS_i is the (*circular bitwise*) *left shift* (or *bitwise left rotation*) mapping 28-bit vectors to 28-bit vectors:

$$LS_i(<e_0, \cdots, e_{27}>) = <e_0, \cdots, e_{27}> <<<< s_i$$

according to the *shift schedule*:

$$<s_0, \cdots, s_{15}> = <1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1>$$

and where PC2 is the *second permuted choice* mapping from 56-bit (auxiliary) vectors to 48-bit (subkey) vectors, defined as follows:

$$PC2(<l_0, \cdots, l_{55}>) =$$
$$<l_{13}, l_{16}, l_{10}, l_{23}, l_0, l_4, l_2, l_{27}, l_{14}, l_5, l_{20}, l_9,$$
$$l_{22}, l_{18}, l_{11}, l_3, l_{25}, l_7, l_{15}, l_6, l_{26}, l_{19}, l_{12}, l_1,$$
$$l_{40}, l_{51}, l_{30}, l_{36}, l_{46}, l_{54}, l_{29}, l_{39}, l_{50}, l_{44}, l_{32}, l_{47},$$
$$l_{43}, l_{48}, l_{38}, l_{55}, l_{33}, l_{52}, l_{45}, l_{41}, l_{49}, l_{35}, l_{28}, l_{31}>$$

3.5.3 Rounds (T): Cipher Function (F), Expansion (E), Permutation (P) and Selection/Substitution (S)

With the subkeys K_i in hand, the *rounds* $T_{K,i}$, which map blocks to blocks, can now be defined as follows. Let B be a block, then by definition for $0 \le i \le 15$ (''^'' denoting *bitwise boolean XOR* of bit-vectors):

$$T_{K,i}(B) = <L_B \hat{} F_{K,i}(R_B), R_B>$$

where the *cipher functions* $F_{K,i}$ ($0 \le i \le 15$) map 32-bit (halfblock) vectors to 32-bit (halfblock) vectors, and are defined by the formula (note this is the only place the subkeys K_i are used):

$$F_{K,i}(R) = P(S(K_i \hat{} E(R)))$$

with E, P and S defined immediately below.

E is the *expansion* mapping from 32-bit (halfblock) vectors to 48-bit (subkey) vectors given by:

$$E(<r_0, \cdots, r_{31}>) =$$
$$<r_{31}, r_0, r_1, r_2, r_3, r_4, r_3, r_4, r_5, r_6, r_7, r_8,$$
$$r_7, r_8, r_9, r_{10}, r_{11}, r_{12}, r_{11}, r_{12}, r_{13}, r_{14}, r_{15}, r_{16},$$
$$r_{15}, r_{16}, r_{17}, r_{18}, r_{19}, r_{20}, r_{19}, r_{20}, r_{21}, r_{22}, r_{23}, r_{24},$$
$$r_{23}, r_{24}, r_{25}, r_{26}, r_{27}, r_{28}, r_{27}, r_{28}, r_{29}, r_{30}, r_{31}, r_0>$$

P is the *permutation* mapping from 32-bit (halfblock) vectors to 32-bit (halfblock) vectors given by:

$$P(\langle t_0, \cdots, t_{31}\rangle) = \langle t_{15}, t_6, t_{19}, t_{20}, t_{28}, t_{11}, t_{27}, t_{16}, t_0, t_{14}, t_{22}, t_{25}, t_4, t_{17}, t_{30}, t_9, t_1, t_7, t_{23}, t_{13}, t_{31}, t_{26}, t_2, t_8, t_{18}, t_{12}, t_{29}, t_5, t_{21}, t_{10}, t_3, t_{24}\rangle$$

S is the *selection/substitution* mapping from 48-bit (subkey) vectors to 32-bit (halfblock) vectors defined as follows:

$$S(\langle z_0, \cdots, z_{47}\rangle) = \langle S_0(\langle z_0, \cdots, z_5\rangle), S_1(\langle z_6, \cdots, z_{11}\rangle), S_2(\langle z_{12}, \cdots, z_{17}\rangle), S_3(\langle z_{18}, \cdots, z_{23}\rangle), S_4(\langle z_{24}, \cdots, z_{29}\rangle), S_5(\langle z_{30}, \cdots, z_{35}\rangle), S_6(\langle z_{36}, \cdots, z_{41}\rangle), S_7(\langle z_{42}, \cdots, z_{47}\rangle)\rangle$$

In this expression, each *submapping* S_h ($0 \le h \le 7$) is a mapping of 6-bit vectors to 4-bit vectors, defined in terms of a 4×16 **matrix**, S*h* (whose entry in its f^{th} row and g^{th} column, S*h*[*f*,*g*] ($0 \le f \le 3$, $0 \le g \le 15$), is a 4-bit vector (or, equivalently, an integer in the range [0, 15])), given by the *rule*:

$$S_h(\langle y_0, \cdots, y_5\rangle) = Sh[\langle y_0, y_5\rangle, \langle y_1, y_2, y_3, y_4\rangle]$$

In this rule, the *identification* of 2-bit and of 4-bit vectors with integers is made via the *big-endian mappings* (namely, $f = \langle u_0, u_1\rangle = 2^1u_0 + 2^0u_1$ and $g = \langle v_0, v_1, v_2, v_3\rangle = 2^3v_0 + 2^2v_1 + 2^1v_2 + 2^0v_3$).

The matrices S*h* ($0 \le h \le 7$) are called *S-boxes*. They lie at the very heart of DES, in the sense of being the major source of its complexity — besides greatly adding to the egodicity of DES, they comprise its only component of *non-linearity* (with respect to *block space*, the 64-dimensional vector space $\mathbf{F}_2^{64}$ over the bit field $\mathbf{F}_2$). The S-boxes have the following values (in pseudocode):

```
S0 = {{14, 4,13, 1, 2,15,11, 8, 3,10, 6,12, 5, 9, 0, 7},
      { 0,15, 7, 4,14, 2,13, 1,10, 6,12,11, 9, 5, 3, 8},
      { 4, 1,14, 8,13, 6, 2,11,15,12, 9, 7, 3,10, 5, 0},
      {15,12, 8, 2, 4, 9, 1, 7, 5,11, 3,14,10, 0, 6,13}};

S1 = {{15, 1, 8,14, 6,11, 3, 4, 9, 7, 2,13,12, 0, 5,10},
      { 3,13, 4, 7,15, 2, 8,14,12, 0, 1,10, 6, 9,11, 5},
      { 0,14, 7,11,10, 4,13, 1, 5, 8,12, 6, 9, 3, 2,15},
      {13, 8,10, 1, 3,15, 4, 2,11, 6, 7,12, 0, 5,14, 9}};

S2 = {{10, 0, 9,14, 6, 3,15, 5, 1,13,12, 7,11, 4, 2, 8},
      {13, 7, 0, 9, 3, 4, 6,10, 2, 8, 5,14,12,11,15, 1},
      {13, 6, 4, 9, 8,15, 3, 0,11, 1, 2,12, 5,10,14, 7},
      { 1,10,13, 0, 6, 9, 8, 7, 4,15,14, 3,11, 5, 2,12}};

S3 = {{ 7,13,14, 3, 0, 6, 9,10, 1, 2, 8, 5,11,12, 4,15},
      {13, 8,11, 5, 6,15, 0, 3, 4, 7, 2,12, 1,10,14, 9},
      {10, 6, 9, 0,12,11, 7,13,15, 1, 3,14, 5, 2, 8, 4},
      { 3,15, 0, 6,10, 1,13, 8, 9, 4, 5,11,12, 7, 2,14}};

S4 = {{ 2,12, 4, 1, 7,10,11, 6, 8, 5, 3,15,13, 0,14, 9},
      {14,11, 2,12, 4, 7,13, 1, 5, 0,15,10, 3, 9, 8, 6},
      { 4, 2, 1,11,10,13, 7, 8,15, 9,12, 5, 6, 3, 0,14},
      {11, 8,12, 7, 1,14, 2,13, 6,15, 0, 9,10, 4, 5, 3}};

S5 = {{12, 1,10,15, 9, 2, 6, 8, 0,13, 3, 4,14, 7, 5,11},
```

```
      {10,15, 4, 2, 7,12, 9, 5, 6, 1,13,14, 0,11, 3, 8},
      { 9,14,15, 5, 2, 8,12, 3, 7, 0, 4,10, 1,13,11, 6},
      { 4, 3, 2,12, 9, 5,15,10,11,14, 1, 7, 6, 0, 8,13}};

S6 = {{ 4,11, 2,14,15, 0, 8,13, 3,12, 9, 7, 5,10, 6, 1},
      {13, 0,11, 7, 4, 9, 1,10,14, 3, 5,12, 2,15, 8, 6},
      { 1, 4,11,13,12, 3, 7,14,10,15, 6, 8, 0, 5, 9, 2},
      { 6,11,13, 8, 1, 4,10, 7, 9, 5, 0,15,14, 2, 3,12}};

S7 = {{13, 2, 8, 4, 6,15,11, 1,10, 9, 3,14, 5, 0,12, 7},
      { 1,15,13, 8,10, 3, 7, 4,12, 5, 6,11, 0,14, 9, 2},
      { 7,11, 4, 1, 9,12,14, 2, 0, 6,10,13,15, 3, 5, 8},
      { 2, 1,14, 7, 4,10, 8,13,15,12, 9, 0, 3, 5, 6,11}};
```

3.5.4 DES Decryption

It is true, but not quite obvious, that the DES transformation described above is *invertible* — nor is it obvious, assuming that it is invertible, how to compute its inverse. It will be proved that:

$$DES^{-1}(K, Q) = (FP \circ T_{K,0} \circ \theta \circ T_{K,1} \circ \theta \circ \cdots \circ T_{K,14} \circ \theta \circ T_{K,15} \circ IP)(Q)$$

Comparing this formula for DES^{-1} to the formula defining DES, the situation can be paraphrased by saying: ''DES^{-1} is the 'same' as DES, except that the schedule of subkeys is visited in reverse order''.

To prove the formula for DES^{-1}, one notes first that:

$$DES^{-1}(K, Q) =$$

$$(FP \circ T_{K,15} \circ \theta \circ T_{K,14} \circ \theta \circ \cdots \circ T_{K,1} \circ \theta \circ T_{K,0} \circ IP)^{-1}(Q) =$$

$$(IP^{-1} \circ T_{K,0}^{-1} \circ \theta^{-1} \circ T_{K,1}^{-1} \circ \theta^{-1} \circ \cdots \circ T_{K,14}^{-1} \circ \theta^{-1} \circ T_{K,15}^{-1} \circ FP^{-1})(Q)$$

Therefore, to prove the claimed formula, it suffices to observe the following *inversion* equalities:

- $IP^{-1} = FP$
- $FP^{-1} = IP$
- $\theta^{-1} = \theta$
- $T_{K,i}^{-1} = T_{K,i}$

The first three of these equalities are straightforward, and the fourth holds because for any block *B*:

$$T_{K,i}(T_{K,i}(B)) =$$

$$T_{K,i}(<L_B \hat{} F_{K,i}(R_B), R_B>) =$$

$$<(L_B \hat{} F_{K,i}(R_B)) \hat{} F_{K,i}(R_B), R_B> =$$

$$<L_B, R_B> =$$

$$B$$

assuming that $U \hat{} U$ is a 0-vector for any bit-vector U, and $U \hat{} V = U$ for any 0-vector V.

3.6 Details of CBC Mode Algorithm

Let $P = <P_0, \cdots, P_{n-1}>$, $n \geq 1$, be a plaintext which has bit-length a positive multiple of 64, where each P_i is a block. Then the CBC mode encryption of P, Q = DES-CBC(K, IV, P), is defined as follows:

$$Q = <Q_0, \cdots, Q_{n-1}>$$

where:

- $Q_0 = \text{DES}(K, IV \hat{} P_0)$
- $Q_i = \text{DES}(K, Q_{i-1} \hat{} P_i)$ for $1 \leq i \leq n-1$

The decryption, $P = \text{DES-CBC}^{-1}(K, IV, Q)$ is given by:

- $P_0 = IV \hat{} \text{DES}^{-1}(K, Q_0)$
- $P_i = Q_{i-1} \hat{} \text{DES}^{-1}(K, Q_i)$ for $1 \leq i \leq n-1$

These transformations are easily seen to be inverses of one another.

Chapter 4

Key Distribution (Authentication) Services

This chapter specifies the key distribution, or authentication, services supported by DCE, together with the protocols associated with them. Currently, only one such service is supported, namely the *Kerberos Key Distribution Service* (KDS), so this whole chapter is devoted to that. The KDS is comprised of two specialised (sub-)services, the *Authentication* (*Sub-*)*Service* (AS) (not to be confused with the generic terminology ''authentication service'') and the *Ticket-granting* (*Sub-*)*Service* (TGS) — and for that reason the KDS is sometimes also referred to as the *AS/TGS*.

For an overview of this chapter, see Section 1.5 on page 18 through Section 1.7 on page 32 — which are considered a *prerequisite* for this whole chapter.

Notes:

1. This chapter is based on, and (unless stated otherwise) is technically aligned with, (a subset of) RFC 1510. However, for editorial reasons, this chapter stands independently, and no familiarity with RFC 1510 is required. (Thus, the part of this chapter that duplicates information in RFC 1510 is intended to be technically equivalent to that document, rewritten for the expository purposes of this document, and any technical discrepancies between the two are inadvertent and to be reconciled.) Differences between the two documents are minor and are readily justified, but the reader should note in particular the following changes:

   a. What is called ''cell'' in DCE is called ''realm'' in RFC 1510.

   b. The service called Key Distribution Service (KDS) here is called *Key Distribution Center* (KDC) there (the difference in terminology merely indicating their preferred communications medium, RPC or raw UDP).

   c. The data type identifiers in the two documents are conservatively different, in an attempt to improve clarity and consistency (instead of, for example, using a mixture of ASN.1, C and other identifiers, such as ''**AP-REQ**'', ''**KRB_AP_REQ**'' and ''authentication header'' all referring to the same object in RFC 1510, Section 5.5.1), without affecting interoperability or placing conformance requirements on implementations.

   For the convenience of the reader, cross-references between this document and RFC 1510 are explicitly indicated generously throughout this chapter, using notation of the form ''[RFC 1510: *x.y.z*]'' as a reference to RFC 1510, Section *x.y.z*.

2. Extensive use is made in this chapter of natural-language algorithmic descriptions. In them, it is the mainline ''success'' (non-error) case that is emphasised — in particular, this document permits different implementations to encounter errors (exceptions) in different orders, and so report different errors under the same external conditions. This is indicated by marking error conditions in algorithms by ''{**errStatusCode-***NAME-OF-ERROR*}'' (perhaps with some explanatory material), leaving to implementations the decision at what point to abort a failed algorithm, and which error to report.

3. [RFC 1510: 5.1]

   This chapter employs the ''ASN.1/BER/DER'' standards for data description/representation (IDL is used only in Section 4.1.1 on page 161),

which are defined in three CCITT (now ITU-T) Recommendations. The *data description language* used is *Abstract Syntax Notation One* (ASN.1), which is defined in CCITT X.208. The *data representation* (*encoding*) used for data described by ASN.1 is the *Basic Encoding Rules* (BER), which are defined in CCITT X.209. Familiarity with those documents, including the data types defined in them, is required for this chapter.

Additionally, the following *Distinguished Encoding Restrictions* (DER) to BER, as specified in CCITT X.509, Section 8.7 — (only) the ones actually used in this chapter are repeated here, in order that this chapter can stand independently of CCITT X.509 — are in effect:

a. The definite form of length encoding shall be used, encoded in the minimum number of octets.

b. For string types, the constructed form of encoding shall not be used.

c. Each unused bit in the final octet of the encoding of a **BIT STRING** value, if there are any, shall be reset (to 0).

Furthermore, implementations that transmit any currently unspecified bits of **BIT STRING** must reset them, for reasons of future extensibility and compatibility. (Note that some existing *implementations* emit full BER, not just the DER subset — new implementations that want to interoperate with those old implementations should therefore *accept* full BER, but in order to be conformant to this document they must *generate* DER.)

Finally, when ASN.1 descriptions are referenced during the course of pseudocode expositions, ASN.1 is *augmented* with the familiar (but non-ASN.1-standard) pseudocode *dot notation* for field elements. For **SEQUENCE**s, this takes the form illustrated by the example: if *seq* is a value of type **SeqType ::= SEQUENCE {int [0] INTEGER, oct [1] OCTET}**, then the values of the fields of *seq* are denoted *seq***.int** and *seq***.oct**, respectively. For **BIT STRING**s, it takes a similar form: for example, if *bits* is a value of type **Bits ::= BIT STRING {bit0 (0), bit1 (1), bit2 (2)}**, then the values of the bits of *bits* are denoted *bits***.bit0**, *bits***.bit1** and *bits***.bit2**, respectively. If the value in question (for example, *seq* or *bits*) is implicitly understood from context, it (and the dot immediately following it) may even be omitted if no confusion will result. For **SEQUENCE OF**s, ASN.1 is further augmented with the familiar pseudocode *bracket notation* for arrays: if *seqof* is a value of type, say, **SeqOfType ::= SEQUENCE OF {SeqType}**, having 2 entries, then its entries are denoted *seqof***[0]** and *seqof***[1]**.

4.1 Fundamental Concepts

[RFC 1510: 1]

This chapter deals with the authentication of (RPC) clients and servers, in the context of cells. The following general notational conventions will be used for these concepts. Recall that the *home cell* of a principal is the cell whose RS datastore holds the security information for the principal. (More precisely, ''a'' home cell of a principal should be spoken of, since some principals may be registered in multiple cells — though this is not to be recommended in general. DCE specifies just one, very specific, kind of multiply registered principal, namely cross-cell surrogate KDS server principals (see below), and in that case care should be taken to indicate which cell is being considered its home cell in a given situation. For non-KDS principals, continue to speak of ''the'' home cell, leaving to the reader the (easy) translation to the case of multiple-registration.)

- *X*, *Y*, *Z*, *W*, *X´*, ···, are reserved for cells.
- *A*, *A´*, ···, are reserved for clients, with home cells *X*, *X´*, ···, respectively.
- *B*, *B´*, ···, are reserved for servers, with home cells *Y*, *Y´*, ···, respectively.
- *C*, *C´*, ···, denote clients *or* servers, in arbitrary cells.

These (especially cells) often appear as subscripts (for example, KDS_Z, $KDS_{X,Y}$), but in order to improve legibility of embedded subscripts they will upgraded when they themselves appear in subscripts (for example, K_{KDSZ}, K_{KDSXY} instead of K_{KDS_Z}, $K_{KDS_{X,Y}}$).

The KDS is a distributed, partitioned RPC service, instantiated by a (conceptually unitary, but potentially replicated) RPC server in each cell *Z*, denoted KDS_Z. If the name of cell *Z* is, say, **/.../cellZ**, then the *RPC service name* of KDS_Z (used for RPC binding purposes) is determined from **/.../cellZ/cell-profile** via the **krb5rpc** interface UUID and version number (specified in Section 4.1.1) — typically, the name associated with this profile element will be **/.../cellZ/sec**, which will be an RPC server group pointing to the individual (replicated) KDS_Z server(s). (See Section 1.18.1 on page 86 for more details on binding models.) (The *principal names* of KDS servers (used for security purposes) — as opposed to their RPC server names (used for communications purposes) — are introduced in Section 4.2.6 on page 172 and Section 4.2.7 on page 174.)

4.1.1 The krb5rpc RPC Interface

[RFC 1510: 8.2]

Each KDS server, KDS_Z, supports the following RPC interface (data types not defined in this specification are defined in the referenced X/Open DCE RPC Specification):

```
[uuid(8f73de50-768c-11ca-bffc-08001e039431), version(1.0)]
interface krb5rpc
{ /* begin running listing of krb5rpc interface */
    [idempotent] void
    kds_request (
        [in] handle_t                       rpc_handle,
        [in] unsigned32                     request_count,
        [in, size_is(request_count)] byte   request[],
        [in] unsigned32                     response_count_max,
        [out] unsigned32                    *response_count,
        [out, size_is(response_count_max), length_is(*response_count)]
            byte                            response[],
        [out] error_status_t                *status );
} /* end running listing of krb5rpc interface */
```

The semantics of *kds_request*( ) are that a client, *C*, invokes *kds_request*( ) to ‘‘send a *KDS Request message*’’ to a KDS server, KDS_Z; *C* receives a *KDS Response message* from KDS_Z when *kds_request*( ) returns. Its parameters are the following:

- *rpc_handle*

  RPC binding handle, bound by the client *C* to a KDS server KDS_Z.

- *request_count*

  Length, in bytes, of KDS Request.

- *request*

  (Array) value (that is, data bytes, or ‘‘contents octets’’) of KDS Request.

- *response_count_max*

  Length of buffer supplied (*response*[]) to receive KDS Response.

- *response_count*

  (Pointer to) length, in bytes, of KDS Response.

- *response*

  (Array) value of KDS Response.

- *status*

  (Pointer to) status code. The currently registered values for *status* (returnable by *kds_request*( )) are the following:

  — *error_status_ok*

  Success in the *communications* sense; that is, a (purported) KDS has successfully received, processed and responded to the request — hence, there is a well-formed response in the *response*[] output parameter. Whether or not the request was successful in the *security* sense must be determined by the examination of *response*[], as specified in Section 4.1.2 on page 163 (and the remainder of this chapter).

  — Status codes other than *error_status_ok* may indicate some transient or permanent failure of the KDS, such as inability to allocate memory; an application should retry the remote call with a different replica if one is available.

The contents, formats and semantics of *request*[] and *response*[] (including their lengths, *request_count* and *response_count*) are defined below (beginning in Section 4.1.2 on page 163, but

their full elaboration consumes this entire chapter).

Note: RFC 1510 specifies *security* protocols that can be supported (with the same security guarantees) over various *communications* mechanisms. Typical non-DCE implementations use UDP (port 88) as the communications mechanism (in conformance with RFC 1510). The **krb5rpc** interface as specified in this document uses RPC as the communications mechanism.

4.1.2 AS and TGS Services

[RFC 1510: 1]

There is no *single* service known as ''*the* KDS service'' supported by KDS servers. Instead, KDS servers support *two* services, each of which is associated with a specific kind of corresponding pair of request/response messages, as follows (for definitions of ''initial'' and ''subsequent'' tickets, see Section 4.1.3):

- *Authentication Service* (AS)

  AS Request/*AS Response* message pair (that is, *request*[] is a value of data type **ASRequest**, and *response*[] is a value of data type **ASResponse**). This is the service by which clients acquire new initial tickets.

- *Ticket-granting Service* (TGS)

  TGS Request/*TGS Response* message pair (that is, *request*[] is a value of data type **TGSRequest**, and *response*[] is a value of data type **TGSResponse**). This is the service by which clients either acquire new subsequent tickets, or manipulate old (initial or subsequent) tickets.

Thus, a KDS Request message is either an AS Request or TGS Request message, and a KDS Response message is either an AS Response or TGS Response message. The KDS supports one additional kind of response message, for reporting errors:

- *KDS Error* message (that is, *response*[] is a value of data type **KDSError**). This is used, in lieu of a KDS Response, to return to the client status information from a failed KDS Request.

4.1.3 Tickets, Keys and Cross-registration

[RFC 1510: 1, 1.1, 2.1]

(*Kerberos*) *tickets* are the (trusted) information objects that the KDS manages (and which are returned by *kds_request*( ), as will be seen in this chapter). Tickets have three kinds of identities associated with them:

- *Issuing cell TCBs*, or what amounts to the same thing, *issuing KDS server*(s), say $KDS_{Z'}$, ···, $KDS_{Z''}$

  The KDS server(s) (which are trusted third parties) that (cooperatively) issued this ticket, also said to be the *issuing authorities* for the ticket.

- *Named client*, say *A* (in cell *X*)

  The client that this ticket authenticates to the targeted server (called *B* in next item). The ticket is also said to be *issued in the name of A*.

- *Targeted server*, say *B* (in cell *Y*)

  The server that this ticket authenticates to the named client (called *A* in previous item).

As will be seen below, there are relationships amongst *A*, *B*, *X*, *Y* and *Z´*, ···, *Z´´* that must be satisfied for a ticket to be valid, namely a *trust chain* must be established:

$$A \rightarrow \mathrm{TCB}_X \rightarrow \mathrm{TCB}_{Z'} \rightarrow \cdots \rightarrow \mathrm{TCB}_{Z''} \rightarrow \mathrm{TCB}_Y \rightarrow B$$

Here, the arrows denote links in the trust chain, *not* communicated messages; more precisely (as discussed below), the above trust chain notation is actually shorthand for the trust chain *of principals*:

$$A \rightarrow \mathrm{KDS}_{X,X} \rightarrow \mathrm{KDS}_{X,Z'} \rightarrow \cdots \rightarrow \mathrm{KDS}_{Z'',Y} \rightarrow \mathrm{KDS}_{Y,Y} \rightarrow B$$

The following notation can be used:

$\mathrm{Tkt}_{A,X,Z',\cdots,Z'',Y,B}$ or $\mathrm{Tkt}_{A,Z',\cdots,Z'',B}$

for a ticket as described above. The home cells *X* and *Y* can be omitted from the notation because they are implicitly known from knowledge of *A* and *B*, but it is convenient to include them for information. All the issuing authorities can even be omitted from the notation if they are implicitly understood or are uninteresting in a given context:

$\mathrm{Tkt}_{A,\cdots,B}$ or $\mathrm{Tkt}_{A,B}$

The simple special case of intra-cell tickets can be abbreviated:

$\mathrm{Tkt}_{A,X,B}$ (instead of $\mathrm{Tkt}_{A,X,X,B}$)

As described in Section 1.1.8 on page 9, *encryption keys* are the *a priori* trusted objects upon which the DCE trusted environment in general is leveraged (in accordance with Kerckhoffs' Doctrine), and in particular are the means by which tickets are protected. These keys come in 2 ''flavours'' (actually, as discussed below, encryption keys in general depend also on a selected *encryption type* and optionally a *key version number*, but these can be mostly omitted from the discussion and notation):

- *Long-term* (or *initial*) *key* of a principal *C*, denoted K_C

  It is stored in the RS_Z datastore of *C*'s home cell *Z*, and it is considered to be secure if and only if it is known only by *C* and TCB_Z (or other TCB components; for example, on the local host). Long-term keys are primarily used to protect internal protocol meta-data (tickets).

- *Short-term* (*subsequent*, *session*, *conversation*, *''true session''*; see Chapter 1, especially the description of the TGS Response message in Section 1.5 on page 18) *key* shared by a client *A* and server *B*, denoted $\mathrm{K}_{A,B}$

  It is not stored in any RS datastore, but rather is generated dynamically for transmittal by a ticket or an authentication header or reverse-authentication header (these are defined later). It is considered to be secure if and only if it is known only by *A* and *B*, and by third parties they (implicitly or explicitly) trust: TCB_X, $\mathrm{TCB}_{Z'}$, ···, $\mathrm{TCB}_{Z''}$, TCB_Y (or other TCB components). Short-term keys are primarily used to protect application-level data (though they are also used in internal protocols).

Tickets carry a short-term key in them, called a *session key*, and are protected by the long-term key of their targeted server. *It is this session key that is the concrete manifestation of the abstract notion of ''authentication'' between a client and server.* As will be seen, a client *C* can successfully use a ticket to authenticate to the targeted server only if *C* knows the ticket's session key (though *C* need not be the ticket's named client). That is, ''stolen'' tickets (obtained, say, by ''wiretapping'') are useless (assuming the keys involved are not compromised). Application data communicated between client and server is protected by a session key (transmitted by a ticket) or by a negotiated conversation key (transmitted by an authentication header or reverse-authentication header).

Tickets are further classified into two broad kinds of category:

- *Ticket-granting-ticket* (TGT) versus *service-ticket*

  Targeted to (that is, protected with the long-term key of) a KDS server or to a non-KDS server, respectively. (In general, *the distinction between ticket-granting-tickets and service-tickets is to be avoided, unless it is absolutely necessary*.)

- *Initial ticket* versus *subsequent ticket*

  Issued on the basis of, respectively, an unauthenticated (or merely preauthenticated) request, or an authenticated request. That is, the issuing authority(ies) are, respectively, uncertain or certain (in the sense of ''certainty'' afforded by a ticket-granting-ticket naming the client) that the identity of the *requesting* client (a communications concept), *C*, is the same as that of the ticket's *named* client or ''claimed identity'' (a security concept). Equivalently, initial tickets are those issued on the basis of the AS Request/Response message exchange, while subsequent tickets are those issued on the basis of the TGS Request/Response exchange. (See the definition of the **tkt-Initial** flag in Section 4.4.2 on page 198.)

This categorisation divides tickets into 4 classes (initial ticket-granting-tickets, subsequent ticket-granting-tickets, initial service-tickets and subsequent service-tickets), all of which can and do actually occur in practice. However, the category of initial service-tickets is rather rare. (An example would be a ''password-changing'' program that insisted on an initial ticket to guarantee that the user changing his/her password has ''recent knowledge'' of the old password (as opposed to an intruder requesting a password change via an unattended seat or hijacked session); if the password-changing program is running under an identity other than the KDS, this initial ticket will be an initial service ticket.)

In order for the definition of ticket to make sense, KDS servers must in fact ''be principals'', in the sense of being registered in the RS datastores in their cells, and this is always assumed to be the case. The principal corresponding to the KDS server in cell X is denoted by the same notation, KDS_X, if no confusion will result, or by the expanded notation $KDS_{X,X}$ if emphasis is needed. (Strictly speaking, the notation KDS_X should be reserved for the KDS server ''as for TCB component (communicating entity)'', and $KDS_{X,X}$ should be reserved to denote the KDS server ''as for principal''.)

More generally, it is possible for a KDS server, KDS_X, to issue a ticket targeted to a KDS server, KDS_Y, other than itself. In order for this to happen, KDS_Y must know the key of a principal registered in RS_X — that principal is denoted by $KDS_{X,Y}$. This arrangement is called a *cross-cell registration* of KDS servers; $KDS_{X,Y}$ is called a *surrogate (principal)* of KDS_Y in cell X, and its long-term key stored in RS_X, K_{KDSXY}, is called a *cross-cell key*. At the same time, this same principal $KDS_{X,Y}$ is also registered in RS_Y, with a different principal stringname (because it's in a different cell) but with the *same* cross-cell key, K_{KDSXY} — and principal (in RS_Y) is also called a *surrogate* (*principal*) of KDS_Y. Thus, in this terminology, ''surrogate'' refers to the principal $KDS_{X,Y}$ registered in both RS_X and in RS_Y (with the *same* key K_{KDSXY}), the only distinction between the two being their principal names (which reflects the cells they are being considered in; that is, which RS datastore their principal information is held in) — and for that reason, the combined terminology *cross-cell surrogate* (*double principal*) is sometimes used; and this principal is also said to *mediate* the $X \rightarrow Y$ trust link. Finally, cross-cell registration also always entails the symmetrically defined surrogate registration of KDS_X in RS_Y: thus, $KDS_{Y,X}$ is the surrogate of KDS_X in cell Y (and in cell X), both with the same long-term key K_{KDSYX}, mediating the $Y \rightarrow X$ trust link. (See Section 1.7 on page 32 for more discussion.)

4.2 Some Basic Data Types

A number of common, non-security-specific data types are defined in this section. Note that the descriptions of the semantics of the data types in this section through Section 4.10 on page 215 are supported by descriptions of the KDS services in Section 4.12 on page 220 through Section 4.15 on page 258, giving the rationale by which applications can evaluate the *trust* to be placed in the KDS's support of these semantics.

4.2.1 Protocol Version Numbers

[RFC 1510: 8.3]

Protocol version numbers are represented by the **ProtocolVersionNumber** data type, which is defined as follows:

```
ProtocolVersionNumber ::= INTEGER
```

Its semantics are that it identifies the KDS protocol version in use. Values for protocol version numbers are centrally registered. Currently registered values are collected in Section 4.2.1.1.

4.2.1.1 Registered Protocol Version Numbers

[RFC 1510: 8.3]

The currently registered values for **ProtocolVersionNumber** are the following:

- **protoVersNum-KRB5** = 5 — Kerberos version 5.

4.2.2 Protocol Message Types

[RFC 1510: 8.3]

Protocol message types are represented by the **ProtocolMessageType** data type, which is defined as follows:

```
ProtocolMessageType ::= INTEGER
```

Its semantics are that it identifies KDS protocol messages. Values for protocol message types are centrally registered. Currently registered values are collected in Section 4.2.2.1.

4.2.2.1 Registered Protocol Message Types

[RFC 1510: 8.3]

The currently registered values for **ProtocolMessageType** are the following (the format and semantics of these messages are defined in Section 4.6 through Section 4.10 on page 215).

- **protoMsgType-AS-REQUEST** = 10 — AS Request.
- **protoMsgType-AS-RESPONSE** = 11 — AS Response.
- **protoMsgType-TGS-REQUEST** = 12 — TGS Request.
- **protoMsgType-TGS-RESPONSE** = 13 — TGS Response.
- **protoMsgType-AUTHN-HEADER** = 14 — Authentication Header.
- **protoMsgType-REVAUTHN-HEADER** = 15 — Reverse-authentication Header.
- **protoMsgType-KDS-ERROR** = 30 — KDS Error.

(Note that the other protocol message types defined in RFC 1510 are not defined or used in DCE.)

4.2.3 Timestamps, Microseconds and Clock Skew

[RFC 1510: 5.2, 5.3.2]

Timestamps are represented by the **TimeStamp** data type, which is defined as follows:

```
TimeStamp ::= GeneralizedTime -- X.208 32.3b
```

Its semantics are that it indicates the generating system clock's UTC time (see the referenced X/Open DCE Time Services Specification), in the syntax of CCITT X.208, Section 32.3b, measured to the granularity of seconds. This syntax is a string of 15 characters, the first 14 of which are the first 14 decimal digits of a DTS string format timestamp, and the 15th of which is the character ''**Z**''. Thus, if this syntax is denoted by ''*CCYYMMDDhhmmss*Z'', then *CCYY* indicates the (century and) year, *MM* the month, *DD* the day, *hh* the hour, *mm* the minute, and *ss* the second. For example, the *TimeStamp* ''**19950825163947Z**'' indicates the UTC time: ''AD August 25, 1995, at 4:39:47 PM''.

Note: The appearance of the character ''Z'' is historical. It is a reference to the ''Z'' (usually verbalised as ''Zulu'') time zone. See the referenced X/Open DCE Time Services Specification for more information.

Note that although the **TimeStamp** data type is a string type on which no *ordering* is defined *a priori*, there is an obvious sense (of ''earlier'' and ''later'') in which timestamps can be compared (see also the referenced X/Open DCE Time Services Specification), and this ordering of **TimeStamp**s will be assumed throughout this chapter. Similarly, there is an obvious sense in which arithmetic operations can be applied to **TimeStamp**s. Namely, the ordering and arithmetic operations are those resulting from regarding the 14 digits of a **TimeStamp** as representing an *integer*, instead of merely a character string.

One particular timestamp is singled out for special attention (see Section 4.8.1 on page 208); this is the ''end-of-time'' timestamp, which is defined as follows (midnight, January 1, 1970):

```
endOfTimeStamp TimeStamp ::= "197001010000Z"
```

Furthermore, **endOfTimeStamp** is considered to occur ''later'' than any other timestamp, in the order mentioned above.

Finally, some protocol elements require, in addition to a timestamp, a *microsecondstamp* (that is, ''microsecond-granularity timestamp''). This is represented by the **MicroSecond** data type, which is defined as follows:

```
MicroSecond ::= INTEGER -- 0..999999
```

The only allowable values of this data type are in the range [0, 999999]. Nominally, the semantic of this data type is to indicate the number of microseconds past an accompanying (second-granularity) timestamp. However, the real security semantic of this data type is to function as a *per-second nonce* (see Section 4.3.1 on page 183 below).

Note: Thus, implementations of the DCE security services on systems that do not have hardware clocks supporting microsecond granularity can finesse the **MicroSecond** data type by simply supporting its security semantic. This can be accomplished, for example, by a ''pseudo-microsecond register'', which merely increments by one (mod 1,000,000) after each request for a microsecondstamp (resetting the register to 0 every second is unnecessary). (In order to preserve the semantics of nonces, this assumes the hardware is incapable of servicing more than 1,000,000 such requests per second.)

4.2.3.1 Maximum Allowable Clock Skew

[RFC 1510: 1.2]

No two clocks in a distributed environment can be synchronised perfectly. In a DCE environment, the DTS service keeps clocks closely synchronised, and even maintains a measure of their *inaccuracy* (distance from UTC). The KDS's use of timestamps depends also on near-synchronisation with UTC, but further requires only, instead of the fine-grained DTS inaccuracies, a very coarse measure of *clock skew* (distance between the clock producing a timestamp and the clock interpreting it, independently of UTC). Namely, the KDS protocols require that clocks are synchronised with one another (and hence, because of DTS, with UTC) to within a *maximum allowable clock skew*, denoted **maxClockSkew** — which is not a single, fixed quantity, but is maintained separately for each clock (and can even be variable (for example, time-dependent or session-dependent) per clock, depending on local policy).

The **maxClockSkew** takes into consideration not only the non-perfect synchronisation of distinct clocks, but also the various expected (and, to some extent, the unexpected) delays due to communications transmission (''networking'') and other processing. Typically, the values of **maxClockSkew** are on the order of 5 minutes — a figure that is much less than the typical lifetime of tickets, and is chosen so as not to introduce unwarranted security risks into the protocols described below. (Five minutes is so much greater than the typical inaccuracies guaranteed by DTS clock synchronisation that the granularity of seconds in **TimeStamp**s (without including the DTS inaccuracies) is quite sufficient.)

All timestamp interpretations in this chapter are to be understood ''modulo **maxClockSkew***'' (where the word ''modulo'' is here used in its common English sense, not in its technical mathematical sense, for which the short form ''mod'' is reserved). For example, to say that two timestamps,* T_0 and T_1, are ''equal'' means that $|T_0 - T_1| \leq$ **maxClockSkew** *(where* **maxClockSkew** *is the maximum allowable clock skew appropriate to the interpreting clock). Similarly, the ''lifetime'' of a ticket (introduced below), which is nominally the (closed) time interval [***tkt-StartTime***,* **tkt-ExpireTime***], is in actuality to be interpreted as [***tkt-StartTime** – **maxClockSkew***,* **tkt-ExpireTime** + **maxClockSkew***] (where* **maxClockSkew** *is the one appropriate to the interpreting clock).*

Note: In accordance with the actual semantic of the **MicroSecond** data type as a nonce instead of its nominal semantic as a microsecond, **maxClockSkew** is applied only to timestamps *T*, not to pairs <*T, M*> of timestamps and microsecondstamps. See the discussion of server ''replay caches'' (which is the only place that the semantics **MicroSecond**s are actually used), in Section 4.5 on page 200.

4.2.4 Cell Names

[RFC 1510: 5.2]

Cell (or, equivalently, realm) names are represented by the **CellName** data type, which is defined as follows:

```
CellName ::= GeneralString -- X.208 31.2; ISO 2022, 2375
```

Its semantics are that it provides references to cells by means of a ''global naming/directory service'' (see the referenced X/Open DCE Directory Services Specification), which is ''external'' to security services (contrast this with RS names, below). Cell names themselves carry enough *syntactic* information to distinguish amongst different global naming services (hence a separate ''**-Type**'' field is not necessary for cell names, as it is for RS names). Acceptable syntaxes for cell names are centrally registered. Currently registered syntaxes are collected in Section 4.2.4.1 on page 169.

Note that global naming services are typically hierarchically organised, and interpret certain syntactic *metacharacters* (such as ''**/**'') as separators of hierarchical naming components. However, for the purposes of the security services provided by the KDS, cell names are *uninterpreted* (''opaque'') (that is, not manipulated or processed, such as decomposing them into hierarchical components), except for testing for equality.

4.2.4.1 Registered Syntaxes for Cell Names

[RFC 1510: 7.1]

The currently registered syntaxes for **CellName**s are the following. Note that these syntaxes do not begin with an initial ''**/.../**''; when these syntaxes are considered in conjunction with the fully qualified DCE syntax, an initial ''**/.../**'' is to be prepended, as specified in the referenced X/Open DCE Directory Services Specification.

- *Internet DNS name type*

  The Internet Domain Naming System (DNS) syntax is specified in the referenced X/Open DCE Directory Services Specification. It provides a hierarchical (little-endian) namespace, with ''**.**'' as (non-escapable) component-separating metacharacter, and the characters ''**/**'', ''**:**'' are not permitted in names; for example, **abc.def.com** (or **/.../abc.def.com** in the fully qualified DCE syntax).

- *DCE X.500 name type*

  The DCE X.500 syntax is specified in the referenced X/Open DCE Directory Services Specification. It provides a hierarchical (big-endian) namespace, with ''**/**'' as (escapable) component-separating metacharacter (other metacharacters ''**=**'', ''**,**'', ''\'' are used for other purposes), and the character ''**:**'' cannot occur in any component before the ''**=**'' (meta)character; for example, **c=xy/o=fed/ou=cba** (or **/.../c=xy/o=fed/ou=cba** in the fully qualified DCE syntax).

- *Prefixed name type(s)*

  The prefixed name type has the syntax:

  *NAMETYPE***:***rest-of-name*

  where *NAMETYPE* is a centrally registered prefix that contains no ''**.**'', ''**/**'' or ''**:**'', and where *rest-of-name* is a string whose syntax is determined by the value of *NAMETYPE*. Currently, there are no registered *NAMETYPE* prefixes.

- *Other name type(s)*

  All other cell naming syntaxes are reserved for future extensibility.

4.2.5 Transit Paths

[RFC 1510: 5.3.1]

Transit paths are represented by the **TransitPath** data type, which is defined as follows:

```
TransitPathType ::= INTEGER
TransitPathValue ::= OCTET STRING

TransitPath ::= SEQUENCE {
        transitPath-Type  [0] TransitPathType,
        transitPath-Value [1] TransitPathValue
}
```

Its semantics are that it indicates the *trust chain* of KDS servers that have participated in a cross-cell authentication. Its fields are the following:

- **transitPath-Type**

  The kind of transit path that **transitPath-Value** represents, including its syntax.

- **transitPath-Value**

  The transited path information itself, to be interpreted according to the value of **transitPath-Type**.

Values of **TransitPathType** are reserved for centrally registered transit path data types. The currently registered values are collected in Section 4.2.5.1.

Note: Negative values of **TransitPathType** do not appear to be specified as unreserved (and therefore available for local assignment) by [RFC 1510: 5.3.1].

4.2.5.1 Registered Transit Path Types

[RFC 1510: 8.3, 3.3.3.1]

The currently registered values for **TransitPathType** are as follows:

- **transitPathType-DNS-X500** = 1

  This transit path type is used for transit paths through cells with DNS and X.500 cell names, with an (*optional*) *compression technique* (where ‘‘optional’’ means that implementations need not employ compression on output, though they must accept it on input). It is defined as follows.

The transit paths (values of type **transitPath-Value**) of this transit path type represent the (underlying **OCTET STRING**s of the) cell names (that is, **CellName GeneralString**s) of the successive transited cells, treated as an unordered set (in other words, not necessarily in the order in which they were visited), expressed in a syntax that supports *lexical compression* (to conserve communications bandwidth), as specified below. This transit path type depends on the Internet DNS and DCE X.500 syntaxes being the only cell name types in effect; transit paths of this type typically contain no upper-case letters (since the ‘‘canonicalised’’ forms of names in these syntaxes contain no upper-case letters), though implementations must accept both cases on input.

Note: It must be emphasised that the ‘‘compression’’ component of this syntax is of a *lexical* nature only. **TransitPath**s must always faithfully represent *every* cell that has actually participated in a cross-cell authentication, and never short-circuit transit paths (for example, by recording only ‘‘third legs of trust triangles’’). That is, transit paths are intended to give a complete secure record only of the sequence of *individual links* of cross-cell trust chains — the *global evaluation of transit path trust* (that is, of the ‘‘overall shape of the trust chain’’) is a higher-level function. In the DCE environment, this function is carried out by the PS, not the KDS (see Chapter 5).

In the **transitPathType-DNS-X500** syntax, the initial ‘‘**/.../**’’ of the DCE naming syntax is always omitted (that is, it is implicitly present). Also for this syntax, the following characters are *metacharacters*; that is, they have special properties discussed in this section (and are to be distinguished from the metacharacters for the DNS and X.500 syntaxes — for those, see the referenced X/Open DCE Directory Services Specification):

‘‘**,**’’ This metacharacter separates the (potentially compressed) representations of successive cell names of a transit path. These are called the *components* of the transit path, and they can be empty (that is, two ‘‘**,**’’1s can occur successively, and ‘‘**,**’’ can occur at the beginning and/or

at the end of a transit path).

''\'' An ''escape'' metacharacter. Any metacharacter (including ''\'' itself) can be escaped by (immediately) preceding it with a ''\'' (with the semantic that the escaped metacharacter is no longer considered to be a metacharacter; that is, it no longer has the special properties discussed here). Unless explicitly indicated otherwise, metacharacters appearing in this specification are assumed to be unescaped (that is, the character immediately preceding them, if any, is not a ''\'' character).

''/'' When it occurs at the beginning of a component (otherwise, it's a non-metacharacter).

''□'' (This notation is used in this section to denote a ''space'' character, for visual clarity.) When it occurs at the beginning of a component (otherwise, it's a non-metacharacter).

''.'' When it occurs at the end of a component (otherwise, it's a non-metacharcter).

In the **transitPathType-DNS-X500** syntax, no component can both begin with a ''/'' and end with a ''.''. A component that is empty, or begins with ''/'', or ends with ''.'' is said to be *compressed*; otherwise, it is said to be *expanded*. A transit path that contains one or more compressed components, or which begins or ends with a '','', is said to be *compressed*; otherwise, it is said to be *expanded*.

In any application of the **transitPathType-DNS-X500** syntax, two cells will be singled out for special treatment: a ''client'' and a ''server'' cell (relative to a posited ''client-server relationship'', which will always be clear from context). Conceptually, the client cell always occurs at the beginning of a transit path, and the server cell always occurs at the end; however, these occurrences of these cells are always implicit, and are never indicated explicitly in a transit path (either compressed or expanded).

An empty transit path (one having no components) is represented by a character string of length 0; it indicates either a trust chain of length 0 (that is, an intra-cell trust path), or of length 1 (that is, a cross-cell trust path with a direct cross-cell registration between the client's cell and the server's cell, with no intermediaries). A non-empty expanded component ''stands for itself'' (that is, directly represents a cell name, in the DNS or X.500 naming syntax). A non-empty compressed component must consist of a non-empty string of non-metacharacters, prepended with an initial ''/'' or ''□'', or appended with a terminal ''.'', but not both.

Compressed transit paths represent expanded transit paths, by expanding their compressed components one at a time, left to right, according to the following rules (examples are given below):

- A compressed component with an initial ''/'' represents the expanded component that results from appending the part of the compressed component following the initial ''/'' to the (expansion of the) preceding component, both of them representing cell names in the X.500 syntax.
- A compressed component with an initial ''□'' must actually begin with the two-character sequence ''□/'', with the meaning that ''□'' escapes ''/''.
- A compressed component with a terminal ''.'' represents the expanded component that results from prepending the part of the compressed component preceding the terminal ''.'' to the (expansion of the) preceding component, both of them representing cell names in the DNS syntax.
- An empty component indicates that all the cells ''between'' the (expansion of the) component preceding it and the (expansion of the) component following it are to be interpolated. In the case of a '','' occurring at the beginning or end of a transit path, this applies to the implicit initial (client) cell or final (server) cell, respectively. Here, ''between'' means ''up to the *least*

common ancestor with respect to (distinguished names of the) DNS and X.500 naming hierarchies''. The (virtual) ''global root'' (denoted ''**/...**'' in the DCE naming syntax) is considered to be the (virtual) least common ancestor of all first-level DNS and first-level X.500 nodes, though it doesn't actually occur in expanded transit paths, of course.

Here are some examples of some compressed transit paths, and their expansions (where, for clarity in the expansions, the client and server cells are shown explicitly, and the symbol ''→'' is used with them and also in place of the ''**,**'' metacharacter). (Recall that the initial ''**/.../**'' of the DCE naming syntax is only implicitly, not explicitly, present.)

- **com,def.,abc.,jkl.org,ghi.**

  Expands to:

  client's cell → **com** → **def.com** → **abc.def.com** → **jkl.org** → **ghi.jkl.org** → *server's cell*

- **c=xy,/o=fed,/ou=cba,c=zw/o=lkj,/ou=ihg**

  Expands to:

  client's cell → **c=xy** → **c=xy/o=fed** → **c=xy/o=fed/ou=cba** → **c=zw/o=lkj** → **c=zw/o=lkj/ou=ihg** → *server's cell*

- **c=uv/o=fed/ou=cba,,c=uv/o=lkj/ou=ihg**

  Expands to:

  client's cell → **c=uv/o=fed/ou=cba** → **c=uv/o=fed** → **c=uv** → **c=uv/o=lkj** → **c=uv/o=lkj/ou=ihg** → *server's cell*

- **c=uv,□/o=fed**

  Expands to (since it is not compressed to begin with):

  client's cell → **c=uv** → **/o=fed** → *server's cell*

- **,com,c=uv,**

  Expands to:

  client's cell (assumed to be **abc.def.com**) → **def.com** → **com** → **c=uv** → **c=uv/o=lkj** → *server's cell* (assumed to be **c=uv/o=lkj/ou=ihg**)

- **,**

  Expands to the same thing as in the preceding example, assuming the client and server cells are the same as in that example.

4.2.6 RS Names

[RFC 1510: 5.2]

Registry Service (RS) names are represented by the **RSName** data type, which is defined as follows:

```
RSNameType ::= INTEGER
RSNameValue ::= SEQUENCE OF GeneralString

RSName ::= SEQUENCE {
        rsName-Type  [0] RSNameType,
        rsName-Value [1] RSNameValue
}
```

Its semantics are that it indicates the per-cell hierarchical names supported by the RS datastores. Its fields are the following:

- **rsName-Type**

  The kind of name that **rsName-Value** represents, including its syntax.

- **rsName-Value**

  The name information itself, to be interpreted according to the value of **rsName-Type**. The entries **rsName-Value[0]**, **rsName-Value[1]**, ···, are called the *components* of the RS name.

Values of **RSNameType** are reserved for centrally registered principal names types.

Note: Negative values of **RSNameType** do not appear to be specified as unreserved (and therefore available for local assignment) by [RFC 1510: 5.2].

The currently registered values are collected in Section 4.2.6.1.

4.2.6.1 Registered RS Name Types

[RFC 1510: 7.2, 8.2.3]

The currently registered values for **RSNameType** are the following:

- **rsNameType-PRINCIPAL** = 1

  Identifies principals registered in the RS datastore.

- **rsNameType-SERVER-INSTANCE** = 2

  The first component (**rsName-Value[0]**) of the RS name identifies an abstract service, and the remaining components (**rsName-Value[***i***]**, $i \geq 1$) identify various concrete *instances*; that is, principal identities under which this service renders its services.

Note: The RS datastore contains various named items other than principals (for example, groups and organisations), however these are irrelevant to the Kerberos authentication architecture specified in this chapter, so they do not rate an **RSNameType**. In particular, there is no architectural relationship between server instances (as defined here) and RS groups.

As discussed in Chapter 11, the RS identifies principals via a hierarchical (‘‘/’’-separated) string-based namespace whose syntax is identical to the CDS naming syntax (which is specified in the referenced X/Open DCE Directory Services Specification). The sequence of components of those string-names map canonically to the sequence of (underlying **GeneralString**s of the) components of **rsName-Value** of type **rsNameType-PRINCIPAL**.

The main use of the RS name type **rsNameType-SERVER-INSTANCE** is for the KDS itself, especially in *cross-cell registrations*. In that application, **rsName-Value[0]** is **krbtgt** (a name which is *reserved* for this use), and (the underlying **GeneralString** of) **rsName-Value[1]** is identical to the string-name of the cross-registered cell’s KDS. This is illustrated by the following examples, assuming *X* and *Y* are cells with names **cellX** and **cellY**, respectively (actually, ‘‘**cellX**’’ and ‘‘**cellY**’’ denote cell names with an internal syntactic structure (for example, Internet DNS syntax or DCE X.500 syntax), but that is irrelevant for this example):

- **krbtgt/cellX**

  The RS name of $KDS_{X,X}$, which is the principal of the KDS server, KDS_X, for cell *X*. It is held as a datastore item in RS_X’s datastore.

- **krbtgt/cellY**

The RS name of $KDS_{X,Y}$, which is the surrogate principal of KDS_Y in cell *X* (held in RS_X's datastore); it is the same principal as the surrogate $KDS_{X,Y}$ in cell *Y* (held in RS_Y's datastore), in the sense that these two principals have the same long-term key, K_{KDSXY}.

Unless stated otherwise, all RS names in this chapter identifying KDSs (including their surrogates) are assumed to have type **rsNameType-SERVER-INSTANCE***, and all non-KDSs are assumed to have type* **rsNameType-PRINCIPAL**.

Note: There are no security mechanisms in place to enforce adherence to these conventions. In particular, the mere *occurrence* in an RS name of a type indicator, such as **rsNameType-SERVER-INSTANCE** (indicating, according to the convention just articulated, a KDS surrogate), does not in itself imply any degree of *trust* in the RS name: all parts of the KDS protocol treat RS names of all types exactly the same. See also the uniqueness requirement in Section 4.2.7. (This is sometimes expressed by saying that the RS name's type is ''merely a hint''.)

4.2.7 Principal Names

[RFC 1510: 5.2]

Principal names are represented by the **CellAndRSName** data type, which is defined as follows:

```
CellAndRSName ::= SEQUENCE {
        cellName [0] CellName,
        rsName   [1] RSName
}
```

Its semantics are that it represents a ''fully qualified'' (in the semantic sense, not the syntactic sense) principal name, consisting of a cell name and an RS name.

The relationship between the semantic **CellAndRSName** data type and the DCE syntactic *string forms of principal names* (or just *stringnames*, for short) is as follows: if **cellAndRSName** is a value of type **CellAndRSName**, whose underlying **cellName** is, say, **cellX**, and whose underlying **rsName-Value**s (that is, their underlying **GeneralString**s, disregarding their **rsName-Type**s) are, say, <**rsName**$_0$, ···, **rsName**$_{r-1}$>, then the corresponding string form of **cellAndRSName** is:

/.../cellX/rsName$_0$**/**···**/ rsName**$_{r-1}$

Implementations of the DCE security services must guarantee that these stringnames are *unique* or *unambiguous*, in the sense that every such stringname refers to at most one principal. Users, including administrators, trust the implementation to guarantee this. In particular, in cross-cell operations, a cell's security administrator must not establish a trust link (that is, exchange KDS registrations) with multiple foreign cells having the same cell name, or whose RS namespace is not trusted to guarantee this uniqueness.

Note that the **CellAndRSName** data type doesn't occur directly in the remaining data types and protocols defined in the remainder of this chapter: the *concept* of principal name is used throughout, though its two component fields (**cellName** and **rsName**) appear *separately*, not bound together in a **CellAndRSName** data type.

Note: It would make sense to use the identifier ''**PrincipalName**'' instead of ''**CellAndRSName**'' for the data type defined above. However, that would conflict with the terminology of RFC 1510, which uses the identifier ''**PrincipalName**'' to denote the data type called ''**RSName**'' in DCE. Unfortunately, RFC 1510's definition of ''**PrincipalName**'' conflicts with the commonsense notion of ''principal name'' (and with RFC 1510's own expository use of the notion of ''principal name''), which connotes *global uniqueness*, not the mere *per-cell-context local uniqueness* of **RSName**s. Hence, to avoid confusion, the identifier ''**PrincipalName**'' is avoided altogether in

DCE.

4.2.8 Host Addresses

[RFC 1510: 5.2]

Note: The notion of ‘‘(transport-level) host addresses’’ properly belongs to the realm of communications (see the referenced X/Open DCE RPC Specification), as opposed to that of security, and arguably should play no role in a security specification. Nevertheless, RFC 1510 does specify usages of host addresses for security purposes, namely that servers may (depending on policy) deny service to clients on the basis of the client’s host address, unless such clients exhibit the appropriate ‘‘proxied’’ or ‘‘forwarded’’ credentials. In order to support coexistence of RFC 1510 environments and DCE environments, host addresses are therefore supported by DCE. *However, KDS servers conformant to DCE must not issue initial tickets (to AS Requests) to clients that do not supply at least one host address, and non-KDS servers conformant to DCE must not deny service to clients on the basis of the client’s host address(es)* (these requirements could change in future revisions of DCE). Thus, host addresses do not currently figure into the DCE security model with nearly the significance they have in RFC 1510. See Section 4.4.1 on page 195.

Host addresses are represented by the **HostAddresses** data type, which is defined as follows:

```
HostAddressType ::= INTEGER
HostAddressValue ::= OCTET STRING

HostAddresses ::= SEQUENCE OF SEQUENCE {
        hostAddr-Type  [0] HostAddressType,
        hostAddr-Value [1] HostAddressValue
}
```

Its semantics are that it represents the host (that is, computer) address(es) (zero or more of them) at which a host is connected to a communications (network) substrate. Its fields are the following:

- **hostAddr-Type**

  The kind of address that **hostAddr-Value** represents, including its syntax.

- **hostAddr-Value**

  The address information itself, to be interpreted according to the value of **hostAddr-Type**.

Non-negative values of **HostAddressType** are reserved for centrally registered host address types; negative values are unreserved (and may, therefore, be assigned locally). The currently registered values are collected in Section 4.2.8.1.

4.2.8.1 Registered Host Address Types

[RFC 1510: 8.1]

Note: In order for this specification to be complete, it should supply references to definitive specifications for the host address types listed below. However, that is not done in RFC 1510 (and therefore, in order not to preempt RFC 1510 on this point, it is not done here). Furthermore, it is not necessary to do so in DCE, because this information is insignificant in DCE environments (as explained in the Note in Section 4.2.8; see also Section 4.4.1 on page 195). Therefore, such references are not given here, and so for the purposes of this specification the following list is supplied only

because it is ‘‘suggestive’’.

The currently registered values for **HostAddressType** are the following:

- **hostAddrType-INTERNET** = 2

  Internet address (corresponding **HostAddressValue** is 4 octets long).

- **hostAddrType-CHAOSNET** = 5

  CHAOSnet address (corresponding **HostAddressValue** is 2 octets long).

- **hostAddrType-XNS** = 6

  XNS address (corresponding **HostAddressValue** is 6 octets long).

- **hostAddrType-ISO** = 7

  ISO address (corresponding **HostAddressValue** has variable length).

- **hostAddrType-DECNET-IV** = 12

  DECnet Phase IV address (corresponding **HostAddressValue** is 2 octets long).

- **hostAddrType-DDP** = 16

  AppleTalk DDP address (corresponding **HostAddressValue** is 3 octets long).

4.2.9 Sequence Numbers

[RFC 1510: 3.2.2, 5.3.2]

Sequence numbers are represented by the **SequenceNumber** data type, which is defined as follows:

```
SequenceNumber ::= INTEGER
```

Its semantics are that it indicates the sequence number of a message in a multi-message sequence (for example, the (potentially) several fragments of a fragmented message). The detailed use of sequence numbers is application-specific. Typically, the initial sequence number is chosen randomly (see below for ‘‘random’’ numbers), and subsequent sequence numbers are unit increments from the initial one. For security purposes, the individual messages (fragments) and the sequence numbers themselves must be bound together and protected in an appropriate application-specific way. (In the case of DCE RPC applications, the use of sequence numbers is specified as part of the RPC protocol specifications — see Chapter 9.)

4.2.10 Last Requests

[RFC 1510: 5.2]

Last requests are represented by the **LastRequests** data type, which is defined as follows:

```
LastRequestType ::= INTEGER
LastRequestValue ::= TimeStamp

LastRequests ::= SEQUENCE OF SEQUENCE {
        lastReq-Type  [0] LastRequestType,
        lastReq-Value [1] LastRequestValue
}
```

Its semantics are that it indicates information maintained by a principal’s home KDS of the principal’s most recent KDS service requests. Its fields are the following:

- **lastReq-Type**

  The kind of last request that **lastReq-Value** represents.

- **lastReq-Value**

  The time of the request, to be interpreted according to the value of **lastReq-Type**.

All values of **LastRequestType** are reserved for centrally registered last request types. The currently registered values are collected in Section 4.2.10.1.

4.2.10.1 Registered Last Request Types

[RFC 1510: 5.2]

The currently registered values for **LastRequestType** are the following, together with the interpretation of corresponding values of **LastRequestValue**. Positive values pertain to the whole cell of the responding KDS server (which cell may contain multiple instances or replicas of KDS servers); negative values pertain only to the individual responding KDS server itself.

- **lastReqType-NONE** = 0

  No information is conveyed by **lastReq-Value**.

- **lastReqType-AS-TGT-ALL** = 1; **lastReqType-AS-TGT-ONE** = –1

  Time of most recent previous AS Request by this principal for an (initial) ticket-granting-ticket.

- **lastReqType-AS-REQ-ALL** = 2; **lastReqType-AS-REQ-ONE** = –2

  Time of most recent previous AS Request by this principal (for an initial ticket, whether a ticket-granting-ticket or a service-ticket).

- **lastReqType-TGT-PRESENTED-ALL** = 3; **lastReqType-TGT-PRESENTED-ONE** = –3

  Authentication time (see Section 4.4.1 on page 195) of the most recent ticket-granting-ticket presented by (in the sense of Section 4.14.1 on page 240) this principal in a previous successful TGS Request. (Note that a principal can have multiple outstanding valid ticket-granting-tickets issued to it.)

- **lastReqType-RENEWAL-ALL** = 4; **lastReqType-RENEWAL-ONE** = –4

  Time of most recent previous successful renewal by this principal.

- **lastReqType-KDS-REQ-ALL** = 5; **lastReqType-KDS-REQ-ONE** = –5

  Time of most recent previous KDS Request (AS Request or TGS Request) by this principal, of any type, successful or not.

4.2.11 Error Status Codes/Text/Data

[RFC 1510: 5.9.1]

Error status codes are represented by the **ErrorStatusCode** data type; error status text is represented by the **ErrorStatusText** data type; error status data is represented by the **ErrorStatusData** data type. These are defined as follows:

```
ErrorStatusCode ::= INTEGER
ErrorStatusText ::= GeneralString
ErrorStatusData ::= OCTET STRING
```

Their semantics are that they indicate error conditions of a failed KDS Request. Values of these three data types always occur as a triple consisting of an **ErrorStatusCode**, an **ErrorStatusText** (optionally) and an **ErrorStatusData** (optionally). The value of **ErrorStatusCode** identifies an error that occurred; **ErrorStatusText** is a character string accompanying **ErrorStatusCode** explaining the error in a human-readable fashion, and **ErrorStatusData** is supplementary information further explaining the error in machine-readable fashion. Note that since **ErrorStatusData** is merely an ''opaque'' **OCTET STRING**, its interpretation must be implicit from the corresponding **ErrorStatusCode**.

Values of error status codes (with associated error text and data) are centrally registered. The currently registered values are collected in Section 4.2.11.1.

Notes:

1. Negative values of **ErrorStatusCode** do not appear to be specified as unreserved (and therefore available for local assignment) by [RFC 1510: 5.9.1].
2. In addition to reporting error codes not specified here, implementations are also permitted to report errors at algorithmic execution points other than those specified in this chapter. For example, a server's failure to allocate memory for a data structure may be reported to the client.

4.2.11.1 Registered Error Status Codes/Text/Data

[RFC 1510: 8.3]

The currently registered values for error codes/text/data are the following. The notational convention is used where *NAME-OF-ERROR* denotes a string specific to each error condition:

- **errStatusCode-***NAME-OF-ERROR*

  Values of **ErrorStatusCode** data type.

- **errStatusText-***NAME-OF-ERROR*

  Values of **ErrorStatusText** data type.

- **errStatusData-***NAME-OF-ERROR*

  Values of **ErrorStatusData** data type.

Registered status codes without accompanying registered status text and/or status data indicates that the latter are not currently specified (but may be in a later revision of this document). (Some terminology is used in these descriptions that won't be formally introduced until later.)

- **errStatusCode-NONE** = 0

  No error (that is, success).

- **errStatusCode-CLIENT-ENTRY-EXPIRED** = 1

  Client entry in RS datastore has expired.

- **errStatusCode-SERVER-ENTRY-EXPIRED** = 2

  Server entry in RS datastore has expired.

- **errStatusCode-BAD-PROTO-VERS-NUM** = 3

  Protocol version number not supported.

- **errStatusCode-CLIENT-OLD-MASTER-KEY-VERS-NUM** = 4

  Client's key encrypted in an old (expired) master key, in the following sense. It is recommended that implementations of DCE protect all copies of the RS datastore other than those actually in use (in the address spaces of trusted programs) at any given moment (such as on-disk files, tape backups, and so on) by encrypting them (or at least the sensitive data contained in them, especially accounts' long-term keys), using some policy-dependent or implementation-dependent trusted encryption mechanism. An encryption key used for this purpose is known as a *master key*. A master key is said to be ''old'' if it is expired or unavailable (for whatever reason — it may just have been lost). In such a case, accounts' keys are unavailable; that is, accounts are ''locked out'' until a new key is established by the security administrator. (Typical implementations use different master keys for different datastore entries, disambiguating them with version numbers, so that the datastore can be incrementally upgraded from one master key to another.) Thus, the master key plays no direct part in the protocol, but surfaces only in this failure code.

- **errStatusCode-SERVER-OLD-MASTER-KEY-VERS-NUM** = 5

  Server's key encrypted in old master key. (For explanation, see preceding error code.)

- **errStatusCode-CLIENT-UNKNOWN** = 6

  Client entry not found in RS datastore.

- **errStatusCode-SERVER-UNKNOWN** = 7

  Server entry not found in RS datastore.

- **errStatusCode-PRINCIPAL-NOT-UNIQUE** = 8

  Multiple entries for principal found in RS datastore.

- **errStatusCode-NULL-KEY** = 9

  Principal has NULL long-term key in RS datastore.

- **errStatusCode-CANNOT-POSTDATE** = 10

  Ticket not eligible for postdating.

- **errStatusCode-NEVER-VALID** = 11

  Requested ticket lifetime is too short (for example, this is always the case if the requested start time is later than the requested expiration time).

- **errStatusCode-POLICY** = 12

  Request not supported by local cell policy (as held in RS).

- **errStatusCode-BAD-OPTION** = 13

  Cannot accommodate requested option.

- **errStatusCode-ENCRYPTION-TYPE-NOT-SUPPORTED** = 14

  Encryption type(s) not supported.

- **errStatusCode-CHECKSUM-TYPE-NOT-SUPPORTED** = 15

  Checksum type not supported.

- **errStatusCode-AUTHN-DATA-TYPE-NOT-SUPPORTED** = 16

  Authentication data type not supported.

- **errStatusCode-TRANSIT-PATH-TYPE-NOT-SUPPORTED** = 17

  Transit path type not supported.

- **errStatusCode-CLIENT-REVOKED** = 18

  Credentials for client have been revoked.

- **errStatusCode-SERVER-REVOKED** = 19

  Credentials for server have been revoked.

- **errStatusCode-TKT-REVOKED** = 20

  Ticket has been revoked.

- **errStatusCode-CLIENT-NOT-VALID** = 21

  Client not (yet) valid, perhaps due to a transitory condition (''try again later'').

- **errStatusCode-SERVER-NOT-VALID** = 22

  Server not (yet) valid, perhaps due to a transitory condition (''try again later'').

- **errStatusCode-BAD-DECRYPTION** = 31

  Unsuccessful decryption (detected by the ''built-in integrity'' feature of DCE encryption types — see Section 4.3.5 on page 187).

- **errStatusCode-TKT-EXPIRED** = 32

  Ticket expired (that is, server's system time is later than ticket's expiration time (modulo **maxClockSkew**)).

- **errStatusCode-TKT-INVALID** = 33

  Ticket is invalid (that is, its invalid option is selected).

- **errStatusCode-REPLAY** = 34

  Request is a repeat of an earlier one.

- **errStatusCode-NOT-US** = 35

  Request intended for another server, not this one.

- **errStatusCode-BAD-MATCH** = 36

  Ticket and authenticator don't match.

- **errStatusCode-CLOCK-SKEW** = 37

  Clock skew is (apparently) greater than **maxClockSkew**.

- **errStatusCode-BAD-ADDRESS** = 38

  Bad host address.

- **errStatusCode-PROTO-VERS-NUM-MISMATCH** = 39

  Protocol version number mismatch.

- **errStatusCode-BAD-PROTO-MSG-TYPE** = 40

  Invalid protocol message type.

- **errStatusCode-BAD-CHECKSUM** = 41

Bad checksum. (Unless the checksum was incorrectly generated, this means that message stream modification has been detected.)

- **errStatusCode-BAD-MSG-ORDER** = 42

  Message is out of order (that is, doesn't conform to the protocol).

- **errStatusCode-BAD-KEY-VERS-NUM** = 44

  Bad key version number.

- **errStatusCode-SERVER-NO-KEY** = 45

  Server's key not available.

- **errStatusCode-BAD-MUTUAL-AUTHN** = 46

  Mutual authentication failure.

- **errStatusCode-BAD-DIRECTION** = 47

  Bad message direction (client ↔ server direction reversed).

- **errStatusCode-BAD-AUTHN-METHOD** = 48

  Alternative authentication method is required.

  — **errStatusData-BAD-AUTHN-METHOD**

  It is (the underlying **OCTET STRING** of) a value of the data type **AuthnMethodData**, defined by:

```
AuthnMethodType ::= INTEGER
AuthnMethodValue ::= OCTET STRING

AuthnMethodData ::= {
        authnMethod-Type  [0] AuthnMethodType,
        authnMethod-Value [1] AuthnMethodValue OPTIONAL
}
```

  Its semantics are that it indicates the alternative authentication method required. Its fields are the following:

  — **authnMethod-Type**

  The kind of authentication method required.

  — **authnMethod-Value**

  Any additional information required, to be interpreted according to the value of **authnMethod-Type**.

  Non-negative values of **AuthnMethodType** are reserved for centrally registered authentication data types; negative values are unreserved (and may, therefore, be assigned locally). There are no currently registered values.

- **errStatusCode-BAD-SEQ-NUM** = 49

  Bad sequence number (that is, message fragment is out of order).

- **errStatusCode-BAD-CHECKSUM-TYPE** = 50

  Bad type of checksum being used (for example, one which has been found to be insecure).

- **errStatusCode-GENERIC** = 60

  Generic, catch-all error code.

- **errStatusCode-FIELD-TOO-LONG** = 61

  Field is too long for this implementation.

4.3 Cryptography- and Security-related Data Types

The data types defined in this section are specifically related to cryptography and security.

4.3.1 Nonces

[RFC 1510: 5.4.1, 5.8.1]

Nonces are represented by the **Nonce** data type, which is defined as follows:

```
Nonce ::= INTEGER
```

Its semantics are that it indicates a *per-context unique (or distinct, or ''one-time'') number*; that is, a number which is always distinguishable from any other when used in a given context (where an appropriate notion of ''context'' must be specified whenever nonces are used). In the particular application of nonces in this chapter (Section 4.12.3 on page 227 and Section 4.14.3 on page 254), they are used to distinguish a client's KDS Requests from one another (to help thwart ''replay attacks''). Therefore, the security semantics of nonces in this chapter are that client principals need to believe (to a degree compatible with the security policies in effect) that the nonces associated with distinct KDS Requests from the same client principal will always be distinct. Implementations must, for example, take into consideration that ''instances'' of the same client principal may be active in different login sessions, perhaps even simultaneously (but that is an implementation-specific consideration not discussed further here).

Notes:

1. Even though a nonce generator of ''cryptographically high quality'' is necessary for security, this notion is not further specified in DCE. In particular, no specific nonce algorithm is (currently) specified. (This lack of specification does not affect interoperability.)
2. In Section 9.2.1.1 on page 332, nonces are introduced that are *byte-vectors*, instead of integers. However, the same principles as described above apply with appropriate modification of detail to that case, and need not be elaborated here.

4.3.2 Random Numbers

[RFC 1510: 3.1.3, 3.2.6]

Random ''numbers'' (actually, byte vectors) are represented by the **Random** data type, which is defined as follows:

```
Random ::= OCTET STRING
```

Its semantics are that it indicates a number which is unpredictable. That is, it is required that random number generators used in any implementation of the DCE Security Services are of *cryptographic high-quality*; that is, it must be computationally infeasible to predict the next random number to be generated, even if the sequence of all previously generated random numbers are known. In essence, this means that every possible value has equal probability of being generated as every other value, at every invocation of the random number generator, but there may be occasions when slight modifications of this idea are warranted (for example, when keys are ''changed'', the ''new'' key should be guaranteed to be different than the ''old'' key).

Notes:

1. Even though a random number generator of ''cryptographically high quality'' is necessary for security, this notion is not further specified in DCE. In particular, no specific random algorithm is (currently) specified. (This lack of

specification does not affect interoperability.)

2. The BER encoding respects the (big-endian) identification that has been made between bit-sequences of length a multiple of 8 and byte sequences. Therefore, random ‘‘numbers’’ can be spoken of equivalently either as byte-vectors or as bit-vectors (but not as integers), without possibility of confusion.

4.3.3 Encryption Keys

[RFC 1510: 6.2]

Encryption keys are represented by the **EncryptionKey** data type, which is defined as follows:

```
EncryptionKeyType ::= INTEGER
EncryptionKeyValue ::= Random

EncryptionKey ::= SEQUENCE {
        encKey-Type  [0] EncryptionKeyType,
        encKey-Value [1] EncryptionKeyValue
}
```

Its semantics are that it indicates the key for (one or more) encryption mechanism(s) and/or checksum mechanisms (see below). Its fields are the following:

- **encKey-Type**

  The **encKey-Type**’s **key type**; that is, the kind of encryption key that **encKey-Value** represents.

- **encKey-Value**

  The **encKey-Type**’s encryption key information itself, to be interpreted according to the value of **encKey-Type**.

Non-negative values of **EncryptionKeyType** are reserved for centrally registered encryption key types; negative values are unreserved (and may, therefore, be assigned locally). The currently registered values are collected in Section 4.3.3.1.

4.3.3.1 Registered Encryption Key Types

[RFC 1510: 6.3.1, 8.3]

The currently registered values for **EncryptionKeyType** are the following:

- **encKeyType-TRIVIAL** = 0

  K has type **encKeyType-TRIVIAL** if and only if *K* is the **trivial** key; that is, it is the empty **OCTET STRING**, of length 0.

- **encKeyType-DES** = 1

  K has type **encKeyType-DES** if and only if *K* is a DES key (64 bits long, though with only 56 ‘‘active’’ bits, and of odd parity — see Chapter 3). It is furthermore required (compare Section 3.4 on page 151) that implementations *must* avoid DES keys *K* that are weak or semi-weak, or for which the **variant key** $K \hat{} K_{f0}$ is weak or semi-weak; and they *should* avoid generating keys for which *K* or $K \hat{} K_{f0}$ is possibly weak (though they should accept such keys from foreign sources). (The reason for the conditions concerning the variant key is because of the use of variant keys in the algorithm of Section 4.3.4.1 on page 185.) Here, $\mathbf{K_{f0}}$ denotes the 64-bit vector (of even parity):

```
<0xf0,0xf0,0xf0,0xf0,0xf0,0xf0,0xf0,0xf0>
```

4.3.4 Checksums

[RFC 1510: 6.4]

Checksums are represented by the **CheckSum** data type, which is defined as follows:

```
CheckSumType ::= INTEGER
CheckSumValue ::= OCTET STRING

CheckSum ::= SEQUENCE {
        cksum-Type  [0] CheckSumType,
        cksum-Value [1] CheckSumValue
}
```

Its semantics are that it indicates a (non-invertible) checksum (or message digest, or one-way function) of some plaintext (which must be specified in context). Its fields are the following:

- **cksum-Type**

  The **CheckSum**'s **checksum type**; that is, the kind of checksum mechanism used to generate the **cksum-Value**.

- **cksum-Value**

  The **CheckSum**'s checksumtext; that is, the actual cryptographic information conveyed by this data structure, to be interpreted according to the value of **cksum-Type**. It is the checksumtext, of checksum type **cksum-Type**, of (the underlying byte string of) an application-specific data value.

Non-negative values of **CheckSumType** are reserved for centrally registered checksum mechanism types; negative values are unreserved (and may, therefore, be assigned locally). The currently registered values are collected in Section 4.3.4.1.

4.3.4.1 Registered Checksum Types

[RFC 1510: 6.4.5, 8.3, 9.1]

The currently registered values for **CheckSumType** are the following.

Notes:

1. The DES-CBC checksum was defined in Section 3.3 on page 150, but it does not occur in the following list.)
2. DCE does not specify a trivial checksum, say ''**cksumType-TRIVIAL**'', paralleling the **encKeyType-TRIVIAL** of Section 4.3.3.1 on page 184. It would be possible to specify such a trivial checksum (having, say **cksumType-TRIVIAL** = 0, and whose corresponding checksumtext, **cksum-Value** is always the bit-vector of length 0), but it is unnecessary to do so. The reason is that the only place the **checkSum** data type occurs in DCE is as the **authnr-Cksum** field of authenticators (see Section 4.5 on page 200), which is an *optional* field. Therefore, any application that wants to ''transmit a checksum of type **cksumType-TRIVIAL**'' needs merely to *omit* this field.

   [RFC 1510: 6.3.1]

- **cksumType-MD4-DES** = 3

Let *K* be an encryption key of type **encKeyType-DES**, and let *P* be a plaintext bit-message. Then the corresponding **cksum-Value**, denoted:

MD4-DES(*K*, *P*)

is defined by the following pseudocode algorithm:

```
R64 = RANDOM64(1);
P' = <R64, P>;
C128 = MD4(P');
P'' = <R64, C128>;
K' = K ^ Kf0;
MD4-DES(K, P) = DES-CBC(K', P'');
```

In words: First generate a random 64-bit vector, R_{64} (called a *confounder* when used in this context — see Section 3.2 on page 148). Let P' denote P prepended with R_{64}, and let C_{128} denote the 128-bit checksum MD4(P'). Let P'' denote the concatenation of R_{64} and C_{128}, and let K' denote the variant key $K \hat{} K_{f0}$ (see Section 4.3.3.1 on page 184). Then the checksum MD4-DES(*K*, *P*) is equal to DES-CBC(K', P'') (which is 192 bits long).

- **cksumType-MD5-DES** = 8

  Let *K* be an encryption key of type **encKeyType-DES**, and let *P* be a plaintext bit-message. Then the corresponding **cksum-Value**, denoted:

  MD5-DES(*K*, *P*)

  is defined by the following pseudocode algorithm:

  ```
  R64 = RANDOM64(1);
  P' = <R64, P>;
  C128 = MD5(P');
  P'' = <R64, C128>;
  K' = K ^ Kf0;
  MD5-DES(K, P) = DES-CBC(K', P'');
  ```

  In words: First generate a random 64-bit vector, R_{64} (called a *confounder* when used in this context — see Section 3.2 on page 148). Let P' denote P prepended with R_{64}, and let C_{128} denote the 128-bit checksum MD5(P'). Let P'' denote the concatenation of R_{64} and C_{128}, and let K' denote the variant key $K \hat{} K_{f0}$ (see Section 4.3.3.1 on page 184). Then the checksum MD5-DES(*K*, *P*) is equal to DES-CBC(K', P'') (which is 192 bits long).

- **cksumType-CL-RPC** = –32767

  Checksum used in the CL-RPC Conversation Manager protocol — see Section 9.2.1.2 on page 332 for details.

  Note: This checksum type is, strictly speaking, ''unregistered'' according to the convention stated in Section 4.3.4 on page 185 concerning negative values of **CheckSumType**. Note also that the value –32767 may be represented as the hexadecimal value 0x8001 in a 16-bit two's complement signed data type (for example, C **short** on typical implementations).

- **cksumType-CO-RPC** = –32766

  Checksum used in the CO-RPC protocol — see Section 9.3.1.3 on page 340 for details.

  Note: This checksum type is, strictly speaking, ''unregistered'' according to the convention stated in Section 4.3.4 on page 185 concerning negative values of **CheckSumType**. Note also that the value –32766 may be represented as the

hexadecimal value 0x8002 in a 16-bit two's complement signed data type (for example, C **short** on typical implementations).

4.3.5 Encrypted Data

[RFC 1510: 6.1]

Note: This section and its subsection give the detailed definition of the notation ''{M}K'' introduced in Section 1.5.

Encrypted data is represented (after it has been encrypted) by the **EncryptedData** data type, which is defined as follows:

```
EncryptionType ::= INTEGER
EncKeyVersionNumber ::= INTEGER
CipherText ::= OCTET STRING

EncryptedData ::= SEQUENCE {
        encData-EncType    [0] EncryptionType,
        encData-KeyVersNum [1] EncKeyVersionNumber OPTIONAL,
        encData-CipherText [2] CipherText
}
```

Its semantics are that it indicates an (invertible) encryption of some plaintext (which must be specified in context). Its fields are the following:

- **encData-EncType**

  The **EncryptedData**'s **encryption type**; that is, the kind of encryption mechanism used to generate the **encData-CipherText**. Encryption mechanisms depend on encryption keys, so the **encData-EncType** implicitly declares an associated **EncryptionKey encKey-Type** (though a given **encKey-Type** may be associated with multiple **encData-EncType**s). Once it has been **negotiated** (that is, agreed upon, in an application-specific manner), it remains constant throughout a given client-server conversation, until it is renegotiated or the conversation ends. As used in DCE, the key version number is (using notation and terminology introduced later in this chapter):

  — Present, in the context of long-term keys K_A associated with principals *A*. (These are the keys stored in RS datastores.)

  — Absent, in the context of short-term session keys $K_{A,B}$ shared between principals *A* and *B*. (These are the keys transmitted in tickets; they are generated by KDS and PS servers.)

  — Optionally present (application-specific choice), in the context of short-term conversation keys $K^{\wedge}_{A,B}$ and $K^{\wedge\wedge}_{A,B}$ shared between principals *A* and *B*. (These are the negotiated ''conversation keys'' transmitted in authentication headers and reverse-authentication headers; they are generated by *A* and *B*, respectively.)

  The key version number is generally omitted from the notations of this chapter, and usually speaks of ''the'' encryption type in a given context (this will never cause confusion).

- **encData-KeyVersNum**

  The **EncryptedData**'s **encryption key version number**; that is, a number selecting the precise encryption key (value of type **EncryptionKey**, appropriate to encryption type **encData-EncType**) used to generate the **encData-CipherText**. It need only be present under conditions where multiple keys may be outstanding. It is generally omitted from the notations of this chapter, preferring to speak of ''the'' key (modulo its version number) in a given context (this will never cause confusion).

- **encData-CipherText**

  The **EncryptedData**'s ciphertext; that is, the actual cryptographic information (the encryption of some plaintext bit-string) conveyed by this data structure, to be interpreted according to the values of **encData-EncType** and **encData-KeyVersNum**. It is the ciphertext, of encryption type **encData-EncType** and key version number **encData-KeyVersNum** (if present), of (the underlying bit-string of) an application-specific data value.

Non-negative values of **EncryptionType** are reserved for centrally registered encryption types; negative values are unreserved (and may, therefore, be assigned locally). The currently registered values are collected in Section 4.3.5.1.

4.3.5.1 Registered Encryption Types

[RFC 1510: 6.1, 6.3.1, 6.3.2, 6.3.3, 6.3.4, 8.3, 9.1]

The currently registered values for **EncryptionType** are the following:

- **encType-TRIVIAL** = 0

  Let *K* be an encryption key of type **encKeyType-TRIVIAL**, and let *P* be a plaintext bit-message. Then the corresponding **encData-CipherText** is simply the plaintext *P* itself, 0-padded if necessary to have length a non-negative multiple of 8 bits (because it must be an **OCTET STRING**). *This encryption type does not afford an adequate basis for security in a potentially hostile (''open'') environment.*

- **encType-DES-CBC-CRC** = 1

  Let *K* be an encryption key of type **encKeyType-DES**, and let *P* be a plaintext bit-message. Then the corresponding **encData-CipherText**, denoted:

  DES-CBC-CRC(*K*, *P*)

  is defined by the following pseudocode algorithm:

  R_{64} = RANDOM$_{64}$(1);
  P' = <R_{64}, 0_{32}, P>;
  C_{32} = $CRC^{\S}_{32}$(0_{32}, P');
  P'' = <R_{64}, C_{32}, P>;
  DES-CBC-CRC(K, P) = DES-CBC(K, P'');

  In words: First generate a random 64-bit vector, R_{64} (called a *confounder* when used in this context — see Section 3.2 on page 148). Let P' denote P prepended with R_{64} and with a 32-bit 0-vector (0_{32}), and let C_{32} denote the 32-bit (twisted) checksum $CRC^{\S}_{32}(0, P')$ with seed 0 (for padding, see Section 2.2.1 on page 136). Let P'' denote P prepended with R_{64} and C_{32}. Then the ciphertext DES-CBC-CRC(*K*, *P*) is equal to DES-CBC(*K*, P'') (for initialisation vector and padding, see Section 3.2 on page 148).

- **encType-DES-CBC-MD4** = 2

  Let *K* be an encryption key of type **encKeyType-DES**, and let *P* be a plaintext bit-message. Then the corresponding **encData-CipherText**, denoted:

  DES-CBC-MD4(*K*, *P*)

  is defined by the following pseudocode algorithm:

```
R64 = RANDOM64(1);
P' = <R64, 0128, P>;
C128 = MD4(P');
P'' = <R64, C128, P>;
DES-CBC-MD4(K, P) = DES-CBC(K, P'');
```

In words: First generate a random 64-bit vector, R_{64} (called a *confounder* when used in this context — see Section 3.2 on page 148). Let P' denote P prepended with R_{64} and with a 128-bit 0-vector (0_{128}), and let C_{128} denote the 128-bit checksum MD4(P') (no padding is used here). Let P'' denote P prepended with R_{64} and C_{128}. Then the ciphertext DES-CBC-MD4(K, P) is equal to DES-CBC(K, P'') (for initialisation vector and padding, see Section 3.2 on page 148).

- **encType-DES-CBC-MD5** = 3

  Let K be an encryption key of type **encKeyType-DES**, and let P be a plaintext bit-message. Then the corresponding **encData-CipherText**, denoted:

  DES-CBC-MD5(K, P)

  is defined by the following pseudocode algorithm:

```
R64 = RANDOM64(1);
P' = <R64, 0128, P>;
C128 = MD5(P');
P'' = <R64, C128, P>;
DES-CBC-MD5(K, P) = DES-CBC(K, P'');
```

  In words: First generate a random 64-bit vector, R_{64} (called a *confounder* when used in this context — see Section 3.2 on page 148). Let P' denote P prepended with R_{64} and with a 128-bit 0-vector (0_{128}), and let C_{128} denote the 128-bit checksum MD5(P') (no padding is used here). Let P'' denote P prepended with R_{64} and C_{128}. Then the ciphertext DES-CBC-MD5(K, P) is equal to DES-CBC(K, P'') (for initialisation vector and padding, see Section 3.2 on page 148).

Note that, by including a checksum in the ciphertext itself, the DES-CBC-CRC DES-CBC-MD4 and DES-CBC-MD5 ciphertexts exhibit *built-in integrity*; that is, correct decryption can be determined solely by inspecting the internal consistency of the resulting plaintext itself, without relying on information external to it. Said another way: ''integrity (at this level) is the concern of the cryptography layer itself, not of the consumer of cryptographic services''. In particular, the *length* of the plaintext need not be conveyed explicitly (at ''application level''; that is, by the consumer of cryptographic services) for the purposes of integrity. (However, if the plaintext P is derived from an ''original'' application-level message M which has been padded to an integral number of DES blocks, then conveying the number of padding bytes is usually a convenient thing to do, and conveying the length of M is a common way to do that.)

Note: It is intended that all encryption mechanisms (except for *encType-TRIVIAL*) registered in this document shall incorporate built-in integrity similar to that of *encType-DES-CBC-CRC*, *encType-DES-CBC-MD4* and *encType-DES-CBC-MD5*.

Note that the degree of assurance of this built-in integrity depends upon the strength of the cryptographic algorithms involved. In particular, non-collision-proof checksums (such as CRC-32) may be susceptible to attacks (such as truncation attacks) that render some assertions invalid (such as the one in the preceding paragraph about not needing to explicitly convey the length of the plaintext).

4.3.6 Passwords

[RFC 1510: 1.2, 6]

Passwords are represented by the **PassWord** data type, which is defined here to be merely a byte-string.

Its semantics are that it indicates a (byte string representation of a) character string (typically a ''memorisable but unguessable'' one, relative to some natural language) associated with a principal, *C*, from which the long-term encryption key K_C for *C* (relative to a specified encryption type **encType**), held in the RS datastore, is derived. The notion of passwords exists because there is a need for humans to be able to securely store a secret, and the best way to do that is to memorise it (''store it in their brain''), which is within normal human capabilities for passwords but not for random (''strong'') encryption keys. DCE specifies one or more password-to-encryption-key mapping for each supported encryption key type, and these mappings are centrally registered. The currently registered ones are collected in Section 4.3.6.1.

Note: Passwords *per se* are irrelevant to the KDS (because they do not appear in *kds_request*( )); they are relevant only to the Login Facility. However, password-to-key mappings do have security significance and depend on cryptography, so it is appropriate to define them in this chapter. Note furthermore that implementations may further restrict (on an implementation-specific basis) the passwords they accept (for example, to avoid easily guessable passwords, or passwords that map into possibly weak keys, and so on — such things are presently beyond the scope of this specification).

4.3.6.1 Registered Password-to-key Mappings

[RFC 1510: 6.3.4]

The password-to-key mappings supported by DCE return as an *output parameter* an encryption key of an appropriate type, and take as *input parameters* the following two data items:

- *passWord*

  The password itself. It is a value of type **PassWord**; that is, a byte string. Typically it is derived from the character string password input parameter to *sec_login_validate_identity* ( ) or *sec_login_valid_and_cert_ident* ( ). The mapping of character string to byte string by those routines is specified as follows (see Section 4.3.6.2 on page 192): they take as input PCS character strings, and map them to byte strings by mapping each character to a byte value according to the US ASCII mapping (or equivalently, for PCS characters, ISO 8859-1). (In particular, of course, upper-case letters are considered to be distinct from lower-case letters.)

- *salt*

  An initialising string (also called *seed*, *pepper*, *mix-in string*, and so on). Its value is a byte string. If no salt is explicitly specified, it defaults to the *default salt*, which consists (in a manner specified below) of:

  — *cellName*

  The (**home**) **cell** of the principal whose password is to be mapped; that is, the name of the cell in whose RS datastore the principal is registered. It is a value of type **CellName**, which is a **GeneralString**.

  — *rsName*

  The RS name of the principal whose password is to be mapped. It is a value of type **RSName**, which consists of an **rsName-Type** and of an **rsName-Value** which is a

sequence of **GeneralString**s — call the components of this sequence $rsName_0$, ···, $rsName_{r-1}$.

Corresponding to the input parameters *cellName* and *rsName* (which are **GeneralString**s), the underlying BER string of ''contents octets'' of their **GeneralString**s is denoted (that is, the BER encoding stripped of its ''identifier octets'' and ''length octets'' — no ''end-of-contents octets'' are present because of the DER in force), which are strings of octets, by respectively:

- *CN*
- *RSN*, with components RSN_0, ···, RSN_{r-1}

The default salt, mentioned above but not yet specified explicitly, is now defined to be:

$DEFAULTSALT = <CN, RSN_0, \cdots, RSN_{r-1}>$

In words: *DEFAULTSALT* is the concatenation of (the underlying strings of octets of) *CN* and of the components of *RSN*.

With the above notations, the currently registered password-to-key mappings, corresponding to the currently registered encryption key types, are defined as follows.

- **encKeyType-TRIVIAL**

  All *passWord*s, *cellName*s and *rsName*s map to the unique (trivial) key of type **encKeyType-TRIVIAL**.

- **encKeyType-DES**

  The mapping of a *passWord* and *salt* (the latter, unless explicitly indicated otherwise, defaults to *DEFAULTSALT*) to a key *K* of type **encKeyType-DES** is defined by the following algorithm.

  First, pad the password and salt:

  $PWS = <passWord, salt, 0, \cdots, 0>$

  In words: *PWS* (''password-with-salt'') is the concatenation of (the byte strings) *passWord* and of *salt*, appended with 0-bits to a (positive) multiple of 64 bits. *PWS* is considered to be a (pseudocode) array of 64-bit blocks, denoted:

  $PWS = <PWS[0], \cdots, PWS[p-1]>$

  where each $PWS[j]$ has length 64 bits. The key *K* is then defined by the following pseudocode:

```
K = <0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00>;
for (j = 0; j <= p-1; j += 1) {
        if (j is even) {
                K ^= PWS[j];
        } else {
                K ^= REVERSE(PWS[j]);
        }
}

FIX-PARITY(K);
K' = DES-CBC-CKSUM(K, PWS);
FIX-PARITY(K');

if (K' is weak or semi-weak) {
        K' ^= <0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xf0>;
        FIX-PARITY(K');
}
```

In words: First, initialise a 64-bit vector *K* by ''fan-folding'' *PWS*, as specified in the pseudocode above, where *REVERSE*( ) is the transformation that maps any bit-vector $<b_0, \cdots, b_{k-1}>$ to its **reversal**, $<b_{k-1}, \cdots, b_0>$. This *K* may not be a DES key, because it may not have odd parity, so next adjust the parity bits of *K* so that it has odd parity (that's the definition of *FIX-PARITY*( )). Then, apply the indicated DES-CBC checksum to *K* and *PWS* (thereby ''uniformly distributing the password and salt over DES key space''), calling the result *K´*. Again, adjust the parity of *K´*. Finally, if *K´* is a weak or semi-weak DES key, XOR it with the indicated constant (and adjust the parity again). The final output of this algorithm (the desired result of the password-to-key mapping), is the resulting *K´*.

Note: Since this is a ''public algorithm'', it is not permissible to modify it to avoid the possibly-weak DES keys mentioned in Section 3.4.3 on page 152, or other DES keys that may in the future be discovered to hold weaknesses. However, at a higher level, implementations are permitted (even encouraged) to reject passwords that would result in all DES keys that are weak in all known senses.

4.3.6.2 Minimum Implementation Requirements

All implementations must support passwords and salts consisting of characters drawn from the DCE *Portable Character Set* (PCS) (see Appendix A, Universal Unique Identifier, of the referenced X/Open DCE Directory Services Specification).

Note: Some implementations may support passwords consisting of characters beyond the PCS. Users should be aware, however, that there are practical limitations on the makeup of passwords, associated with ''seat portability''. Namely, ''input methods'' for all **GeneralString**s do not necessarily exist at all ''seats'' from which the user may want to login. Therefore, users must restrict the choice of characters in their passwords to the characters they know will be supported at all the seats from which they will want to login. For *universal* seat portability, users must restrict their passwords to the DCE PCS (because of the PCS minimal implementation requirement for all implementations of DCE stated in this section).

4.3.7 Authentication Data

[RFC 1510: 5.4.1]

Authentication data is represented by the **AuthnData** data type, which is defined as follows:

```
AuthnDataType ::= INTEGER
AuthnDataValue ::= OCTET STRING

AuthnData ::= SEQUENCE {
        authnData-Type  [1] AuthnDataType,
        authnData-Value [2] AuthnDataValue
}
```

Its semantics are that it indicates information that is exchanged between clients and KDS servers (in KDS Request/Response exchanges), which may be required before KDS services (that is, ticket issuance) can be accessed. (Note that the KDS does not support ACLs for access control to its AS/TGS services — just the opposite: tickets are required before PACs can be obtained, on which ACLs are based.) In the case of AS Request/Responses (as opposed to TGS Request/Responses), this information is usually referred to as *pre-authentication* data, because AS Request/Responses are, in fact, ‘‘unauthenticated’’ (in the sense of not containing an ‘‘authentication header’’ as defined in Section 4.6 on page 202). Its fields are the following:

- **authnData-Type**

  The kind of authentication data that **AuthnData-Value** represents, including its format.

- **authnData-Value**

  The authentication data information itself, to be interpreted according to the value of **authnData-Type**.

Non-negative values of **AuthnDataType** are reserved for centrally registered authentication data types; negative values are unreserved (and may, therefore, be assigned locally). The currently registered values are collected in Section 4.3.7.1.

4.3.7.1 Registered Authentication Data Types

[RFC 1510: 8.3]

The currently registered values for **AuthnDataType** are the following:

- **authnDataType-TGS-REQ** = 1

  authnData-Value contains (the underlying **OCTET STRING** of) an *authentication header* (defined in Section 4.6 on page 202), for use in a TGS Request.

  Note: (Reverse-)Authentication data does not occur in a TGS Response — see Section 4.14.2 on page 245.

- **authnDataType-PW-SALT** = 3

  authnData-Value contains (encoded as an **OCTET STRING**) a ‘‘salt’’ (whose underlying contents octets are to be used in computing the principal’s key from its password — see Section 4.3.6.1 on page 190). (A zero-length salt is a valid value — in particular, it does not mean ‘‘use default salt’’.) See Section 4.12.3 on page 227.

4.3.8 Authorisation Data

[RFC 1510: 5.2]

Authorisation data is represented by the **AuthzData** data type, which is defined as follows:

```
AuthzDataType ::= INTEGER
AuthzDataValue ::= OCTET STRING

AuthzData ::= SEQUENCE OF SEQUENCE {
        authzData-Type  [0] AuthzDataType,
        authzData-Value [1] AuthzDataValue
}
```

Its semantics are that it indicates information that may be (depending on application-specific authorisation policy) needed by a server in order to determine a client's access to the server's services. Its fields are the following:

- **authzData-Type**

  The kind of authorisation data that **authzData-Value** represents, including its format.

- **authzData-Value**

  The authorisation data information itself, to be interpreted according to the value of **authzData-Type**.

Non-negative values of **AuthzDataType** are reserved for centrally registered authorisation data types; negative values are unreserved (and may, therefore, be assigned locally). The currently registered values are collected in Section 4.3.8.1.

Note: The usage semantics to be attached to authorisation data is application-specific, but typically a value **authzData** of data type **AuthzData** is used for access based on the so-called ''AND model''; that is, the elements of the array, *authzData*[0], ···, *authzData*[n–1], all ''further restrict'' one another. That is to say, it is typically used to determine access in the following manner:

```
if (authzData[0], ···, authzData[n-1] all grant access) {
        GRANT access;
} else {
        DENY access;
}
```

4.3.8.1 Registered Authorisation Data Types

[RFC 1510: 8.3]

The currently registered values for **AuthzDataType** are the following:

- **authzDataType-PAC** = 64

  Pickled PACs, as specified in Section 5.2.6 on page 281.

4.4 Tickets

[RFC 1510: 5.3.1]

Tickets are represented by the **Ticket** data type, which is defined as follows:

```
Ticket ::= [APPLICATION 1] SEQUENCE {
        tkt-ProtoVersNum  [0] ProtocolVersionNumber,
        tkt-ServerCell    [1] CellName,
        tkt-ServerName    [2] RSName,
        tkt-EncryptedPart [3] EncryptedData
}
```

Its semantics have been described in Section 4.1.3 on page 163, where the notation $\mathrm{Tkt}_{A,B}$ for a ticket with named client *A* in cell *X* and targeted server *B* in cell *Y* are introduced (eliding, here, any issuing authority(ies), $\mathrm{KDS}_{Z'}$, ···, $\mathrm{KDS}_{Z''}$ from the notation). In terms of this notation, its fields are the following:

- **tkt-ProtoVersNum**

  The protocol version number of the **Ticket** data type. Its value is **protoVersNum-KRB5**.

- **tkt-ServerCell**

  B's home cell, *Y*.

- **tkt-ServerName**

  B's RS name in RS_Y.

- **tkt-EncryptedPart**

  The encryption type (**encData-EncType**), key version number (**encData-KeyVersNum**) and ciphertext (**encData-CipherText**) encryption of (the underlying BER-encoded bit-string of) a value of type **TicketEncryptPart**, which is defined in Section 4.4.1. The ''pre-encrypted'' plaintext of this field (which is a value of the data type **TicketEncryptPart**) is denoted by **tkt-EncryptPart**.

4.4.1 Part of Ticket to be Encrypted

[RFC 1510: 5.3.1]

The encrypted data carried in a ticket is represented (before it is encrypted) by the **TicketEncryptPart** data type, which is defined as follows:

```
TicketEncryptPart ::= [APPLICATION 3] SEQUENCE {
        tkt-Flags         [ 0] TicketFlags,
        tkt-SessionKey    [ 1] EncryptionKey,
        tkt-ClientCell    [ 2] CellName,
        tkt-ClientName    [ 3] RSName,
        tkt-TransitPath   [ 4] TransitPath,
        tkt-AuthnTime     [ 5] TimeStamp,
        tkt-StartTime     [ 6] TimeStamp OPTIONAL,
        tkt-ExpireTime    [ 7] TimeStamp,
        tkt-MaxExpireTime [ 8] TimeStamp OPTIONAL,
        tkt-ClientAddrs   [ 9] HostAddresses OPTIONAL,
        tkt-AuthzData     [10] AuthzData OPTIONAL
}
```

Its fields are the following:

- **tkt-Flags**

  Options (represented by flag bits) selected by this ticket. The **TicketFlags** data type is defined in detail in Section 4.4.2 on page 198. (In Section 4.12.2 on page 222 and Section 4.14.2 on page 245, when a KDS server creates a ticket its options are considered to be deselected by default unless they are explicitly selected.)

- **tkt-SessionKey**

  The **session key**, $K_{A,B}$, associated with this ticket; that is, the actual data item that represents the ensuing client-server session. (Either it or a subsequently negotiated conversation (''true session'') key can be used to protect client-server communications.) Any principal that knows this session key is considered to be ''the same principal (relative to this ticket)'' as the principal *A*.

- **tkt-ClientCell**

  A's home cell, *X*.

- **tkt-ClientName**

  A's RS name in RS_X.

- **tkt-TransitPath**

  The transit path of this ticket.

- **tkt-AuthnTime**

  The **authentication time** of this ticket; that is, the time of *A*'s original (AS) authentication (the time at which *A*'s home KDS server, KDS_X, issued the original initial ticket-granting-ticket, $Tkt_{A,KDSX}$, on which $Tkt_{A,B}$ is ultimately based, by a sequence of TGS Requests).

- **tkt-StartTime**

  The **start time** of this ticket; that is, the time before which $Tkt_{A,B}$ is not to be *honoured* (accepted as evidence of authentication by *B*). Together with the expiration time of this ticket (**tkt-ExpireTime**), this field determines the *lifetime* of the ticket, namely, the time interval [**tkt-StartTime**, **tkt-ExpireTime**] (modulo **maxClockSkew**). This field must be present if $Tkt_{A,B}$ is postdated (**tkt-Postdated** flag is set), and it may be present at other (depending on local policy); in its absence, the start time of this ticket defaults to the ticket's authentication time (**tkt-AuthnTime**).

- **tkt-ExpireTime**

  The (*relative*) *expiration time* of this ticket; that is, the time after which $Tkt_{A,B}$ is not to be honoured by *B*. Thus, the (*relative*) *lifetime* of the ticket is the time interval [**tkt-StartTime**, **tkt-ExpireTime**] (modulo **maxClockSkew**).

  Note: Individual servers may decline to honour certain tickets which have not yet expired, depending on local policy.

- **tkt-MaxExpireTime**

  The *absolute* (or *maximum*) *expiration time* of this ticket; that is, the time beyond which KDS servers will not issue a renewed ticket based on $Tkt_{A,B}$. Thus, the *absolute* (or *maximum*) *lifetime* of the ticket is the time interval [**tkt-StartTime**, **tkt-MaxExpireTime**] (modulo **maxClockSkew**). This field is present if and only if $Tkt_{A,B}$'s **tkt-Renewable** flag is set, in which case it must indicate a time later than **tkt-ExpireTime**; in its absence, the absolute expiration time of this ticket defaults to the ticket's expiration time (**tkt-ExpireTime**).

- **tkt-ClientAddrs**

  Zero (if absent) or more (if present) client host addresses. In the zero-address case, the ticket can be used from any ''location'' (that is, transport address — this is discussed further shortly); in the non-zero-address case, these are the addresses from which the ticket is ''supposed'' to be used (depending on server policy). In the RFC 1510 environment (as opposed to the DCE environment), this means the following:

  — A server *B* must never deny service to a client *A* presenting a $Tkt_{A,B}$ on the basis of *A*'s ''location'' (that is, the transport address of the host from which *A* is communicating, as reported to *B* by its local host's operating system's implementation of the transport provider), provided that *A*'s location is among the addresses indicated by $Tkt_{A,B}$'s **tkt-ClientAddrs** (and, if applicable, $Tkt_{A,B}$ has been properly proxied or forwarded — see Section 4.4.2 on page 198).

  — On the other hand, if *A*'s location is *not* among the addresses indicated by $Tkt_{A,B}$'s **tkt-ClientAddrs**, then *B may* deny service, depending on *B*'s policy.

  Thus, in the RFC 1510 environment, the decision by the target server *B* to honour such address restrictions is an optional server policy decision; that is, *B* may, depending on policy, choose to enforce or ignore the **tkt-ClientAddrs** field (possibly even on a per-client per-call basis). *In the current DCE environment* (that is, in environments conformant to DCE), this optionality has been removed, and the policy decision has been taken that the **tkt-ClientAddrs** field is *never* used to deny service: all servers *B* (including all system servers such as KDS and PS, and all application servers conforming to DCE) must *always* honour tickets $Tkt_{A,B}$ having arbitrary host address fields **tkt-ClientAddrs** (provided they are otherwise acceptable), regardless of the location of the client *A* — with the sole exception that the KDS's AS service always requires clients to supply at least one host address in every AS Request (see the Note in Section 4.2.8 on page 175). (This policy decision *could* conceivably change in future revisions of DCE, though there is no current intent to do so. Therefore, in order to support an orderly transition to such a possible future environment, clients are encouraged to use the **tkt-ClientAddrs** field as specified herein with this in mind.)

  In particular, the **HostAddressType** field(s) of the **HostAddresses** data type (see Section 4.2.8 on page 175) are always ignored in the DCE environment. (This is the reason that references need not be supplied in DCE for the registered host address types listed in Section 4.2.8.1 on page 175.)

- **tkt-AuthzData**

  The authorisation data associated with this ticket. If it is not present, no authorisation data is associated with the ticket (and therefore a server relying on the presence of such data for its access decisions must deny access).

Note that tickets are not in general interpretable (in the sense of being decryptable) by their named clients (except in the case where the named client also happens to be the targeted server). Nevertheless, the information in them is otherwise (securely) available to the named client. Namely (as seen in Section 4.12.3 on page 227 and Section 4.14.3 on page 254), all the fields except **tkt-TransitPath** and **tkt-AuthzData** are available in the KDS Response that delivers the ticket to the client. The client knows **tkt-TransitPath** because it is itself involved in passing this ticket to the corresponding issuing authorities identified by the transit path. The **tkt-AuthzData** field is not dealt with in this chapter, but as seen in Chapter 5, this information is communicated to the client by the PS in a DCE environment.

4.4.2 Ticket Flags

[RFC 1510: 2, 5.2, 5.3.1]

The options that may be selected by a ticket, $Tkt_{A,B}$ (that is, requested by the named client *A*, specified by issuing authority(ies), or interpreted by the targeted server *B*), are represented by the **TicketFlags** data type, which is defined as follows:

```
TicketFlags ::= BIT STRING {
        tkt-Forwardable (1),
        tkt-Forwarded   (2),
        tkt-Proxiable   (3),
        tkt-Proxied     (4),
        tkt-Postdatable (5),
        tkt-Postdated   (6),
        tkt-Invalid     (7),
        tkt-Renewable   (8),
        tkt-Initial     (9)
}
```

The semantics of these bits are that if a value of type **TicketFlags** has the corresponding bit set (to 1) then the option is selected; if the bit is reset (to 0), the option is deselected. The bits indicate the following options (bits not currently specified are reserved for future usage):

- **tkt-Forwardable**

  This $Tkt_{A,B}$ is **forwardable**; that is, this flag grants KDS servers the right to issue a forwarded $Tkt^{*}_{A,B}$ (which may even be a ticket-granting-ticket) based on $Tkt_{A,B}$. By a **forwarded** $Tkt^{*}_{A,B}$ is meant a ticket that is used by the (same) client *A* from a (potentially) different ''location'' (host address, **tkt-ClientAddrs**) than *A*'s ''location'' when $Tkt_{A,B}$ was originally issued. (Here, the notion of ''same client *A*'' is relative to a ticket, and it means any client that knows the session key carried in the ticket.)

- **tkt-Forwarded**

  This $Tkt_{A,B}$ has either itself been forwarded (see just above), or has been issued based on a ticket that had previously been forwarded.

- **tkt-Proxiable**

  This $Tkt_{A,B}$ is *proxiable*; that is, this flag grants KDS servers the right to issue a proxied $Tkt^{*}_{A,B}$ (but *not* a ticket-granting-ticket — this is the only difference between forwarding and proxying from the point of view of the KDS, though applications may opt to distinguish between them for other purposes) based on $Tkt_{A,B}$. By a *proxied* $Tkt^{*}_{A,B}$ is meant a ticket that is used by the (same) client *A* from a (potentially) different host address (**tkt-ClientAddrs**) than that to which $Tkt_{A,B}$ was originally issued. (Here, the notion of ''same client *A*'' is relative to a ticket, and it means any client that knows the session key carried in the ticket.)

- **tkt-Proxied**

  This $Tkt_{A,B}$ has either itself been proxied (see just above), or has been issued based on a ticket that had previously been proxied.

- **tkt-Postdatable**

  This $Tkt_{A,B}$ is *postdatable*; that is, this flag grants KDS servers the right to issue a postdated $Tkt^{*}_{A,B}$ based on $Tkt_{A,B}$. By a *postdated* $Tkt^{*}_{A,B}$ is meant a ticket that has substantially the same information in it as $Tkt_{A,B}$ but with a new start time (**tkt-StartTime**).

- **tkt-Postdated**

  This $\text{Tkt}_{A,B}$ has been postdated.

- **tkt-Invalid**

  This $\text{Tkt}_{A,B}$ is *invalid*; that is, this flag grants KDS servers the right to issue a valid $\text{Tkt}^{*}_{A,B}$ based on $\text{Tkt}_{A,B}$. By a *valid* (or *validated*) $\text{Tkt}^{*}_{A,B}$ is meant a ticket that has substantially the same information in it as $\text{Tkt}_{A,B}$ but with its invalid option (**tkt-Invalid**) deselected, with the semantic that target (non-KDS) end-servers *B* are to honour only valid tickets. This flag is used in conjunction with the **tkt-Postdated** flag.

  Note: This flag is supported for security purposes — it does not introduce any new functionality that is not otherwise available. In order to be used effectively, it needs to be used with a ‘‘revocation’’ (or ‘‘hotlist’’) mechanism, which is not specified in this revision of DCE. Namely, use of this flag forces the KDS to be ‘‘visited’’ in order for a postdated ticket (which is always originally issued in an invalid state) to be rendered valid, thereby giving the KDS a timely opportunity to revoke tickets by checking them against the hotlist.

- **tkt-Renewable**

  This $\text{Tkt}_{A,B}$ is *renewable*; that is, this flag grants KDS servers the right to issue a renewed $\text{Tkt}^{*}_{A,B}$ based on $\text{Tkt}_{A,B}$. By a *renewed* $\text{Tkt}^{*}_{A,B}$ is meant a ticket that has substantially the same information in it as $\text{Tkt}_{A,B}$ but with a new (later) expiration time (**tkt-ExpireTime**).

  Note: This flag is supported for security purposes — it does not introduce any new functionality that is not otherwise available. In order to be used effectively, it needs to be used with a ‘‘revocation’’ (or ‘‘hotlist’’) mechanism, which is not specified in this revision of DCE. Namely, use of this flag forces the KDS to be ‘‘visited’’ before the absolute expiration time of tickets, thereby giving the KDS a timely opportunity to revoke tickets by checking them against the hotlist.

- **tkt-Initial**

  This $\text{Tkt}_{A,B}$ is an *initial* ticket; that is, it was issued in response to an AS Request to *A*’s home KDS_{X}. If **tkt-Initial** is reset, the ticket is said to be a *subsequent* ticket; that is, it was issued in response to a TGS Request to some KDS server.

Note that the above descriptions do not give complete details. For that, see that detailed descriptions of these flags, in Section 4.12 on page 220 and Section 4.14 on page 240.

4.5 Authenticators

[RFC 1510: 3.2.3, 5.3.2]

Authenticators are represented by the **Authenticator** data type, which is defined as follows:

```
Authenticator ::= [APPLICATION 2] SEQUENCE  {
        authnr-ProtoVersNum    [0] ProtocolVersionNumber,
        authnr-ClientCell      [1] CellName,
        authnr-ClientName      [2] RSName,
        authnr-Cksum           [3] CheckSum OPTIONAL,
        authnr-ClientMicroSec  [4] MicroSecond,
        authnr-ClientTime      [5] TimeStamp,
        authnr-ConversationKey [6] EncryptionKey OPTIONAL,
        authnr-SeqNum          [7] SequenceNumber OPTIONAL,
        authnr-AuthzData       [8] AuthzData OPTIONAL
}
```

Its semantics are that it accompanies $Tkt_{A,B}$ in a client-server service request, and proves to *B* that this request is ‘‘really from *A*’’, in the sense that it was sent from *A* ‘‘now’’ (modulo **maxClockSkew**). Its fields are the following:

- **authnr-ProtoVersNum**

  The protocol version number of the **Authenticator** data type. Its value is **protoVersNum-KRB5**.

- **authnr-ClientCell**

  A’s home cell.

- **authnr-ClientName**

  A’s RS name in its cell.

- **authnr-Cksum**

  The checksum of the (application-specific) message that this authenticator accompanies. (This is used internally in the KDS protocol itself in the KDS Request ‘‘application’’, as specified below.)

- **authnr-ClientMicroSec**

  A’s *microsecondstamp*, interpreted in conjunction with *A*’s timestamp (**authnr-ClientTime**, below). The timestamp and microsecond timestamp pair, <*T*, *M*> (= <**authnr-ClientTime**, **authnr-ClientMicroSec**>), is used (differently) by clients and servers as a *nonce*. It is used by clients as a nonce to match up request/reply (authentication header/reverse-authentication header) pairs (see, for example, Section 9.3.1.3 on page 340). It is the client’s responsibility to store the timestamp and microsecond pairs <*T*, *M*> of all outstanding requests for which it remains interested in replies. It is also used by servers as a nonce to ensure they do not accept duplicate authenticators from the same client, because to do so risks a faked authentication due to ‘‘replay attacks’’ (which, incidentally, threaten authenticity only, not integrity or confidentiality, because they do not involve the compromise of a key). It is the server’s responsibility to maintain a *replay cache* for this purpose, storing the timestamp and microsecondstamp pairs <*T*, *M*> from all authenticators it receives from all clients over the time interval $1T - S| \leq$ **maxClockSkew**, where *S* is the server’s system timestamp.

  Note: This timestamp-based technique for replay detection is not the only possible technique; for example, random-number-based ‘‘challenge/response’’ techniques for environments that don’t want to rely on a time service. Such techniques are

supported elsewhere in DCE.

- **authnr-ClientTime**

  A's timestamp when it generated this authenticator. This timestamp (modulo **maxClockSkew**) has the semantics of convincing the receiving server *B* that the communication it is having with client *A* (securely identified in the accompanying $Tkt_{A,B}$) is happening in real-time. In colloquial terms: ''*A* is 'online' and is communicating with *B* 'now'.''

- **authnr-ConversationKey**

  A's choice of a *conversation* (or *subsession* or ''true session'') *key*, $\hat{K}_{A,B}$, indicating that *A* prefers this key to be used to protect client-server communications. If absent, the session key $K_{A,B}$ in $Tkt_{A,B}$ (**tkt-SessionKey**) is to be used as the conversation key. This is part of *conversation key negotiation* between *A* and *B*.

- **authnr-SeqNum**

  A's choice of sequence number. If absent, this application is not using sequence numbers, or no sequence number is applicable to this message (for example, it might be a non-fragmented message).

- **authnr-AuthzData**

  Additional authorisation data included by *A*. This is authorisation data additional to that specified in the accompanying $Tkt_{A,B}$ (**tkt-AuthzData**). Note that its contents are completely up to *A*'s discretion, unlike the **tkt-AuthzData** which is sealed in the ticket by the KDS server. Therefore, even though its use depends on application-specific authorisation policy, it is typically used only to ''further restrict'' *A*'s access rights at *B*; that is, it is typically used to determine access in the manner paraphrased as: ''If both **tkt-AuthzData** and **authnr-AuthzData** grant access, then access is granted, otherwise it is denied''. Or in pseudocode:

```
if ((this application does not require tkt-AuthzData)
1|  (tkt-AuthzData is present and grants access)) {
        if ((this application does not require authnr-AuthzData)
        1|  (authnr-AuthzData is present and grants access)) {
                GRANT access;
        }
} else {
        DENY access;
}
```

This can be used to support ''least privilege'' access policies.

4.6 Authentication Headers

[RFC 1510: 5.5.1]

Authentication headers are represented by the **AuthnHeader** data type, which is defined as follows (where **APPLICATION 14** indicates **protoMsgType-AUTHN-HEADER** — see Section 4.2.2.1 on page 166):

```
AuthnHeader ::= [APPLICATION 14] SEQUENCE {
        authnHdr-ProtoVersNum    [0] ProtocolVersionNumber,
        authnHdr-ProtoMsgType    [1] ProtocolMessageType,
        authnHdr-Flags           [2] AuthnHeaderFlags,
        authnHdr-Tkt             [3] Ticket,
        authnHdr-EncryptedAuthnr [4] EncryptedData
}
```

Its semantics are that it supplies the *forward authentication information* that actually ''authenticates a client *A* to a server *B*'', by binding together an authenticator and a ticket, $Tkt_{A,B}$ (naming *A* and targeted to *B*), associated with one another. Its fields are the following:

- **authnHdr-ProtoVersNum**

  The protocol version number of the **AuthnHeader** data type. Its value is **protoVersNum-KRB5**.

- **authnHdr-ProtoMsgType**

  The kind of protocol message this **AuthnHeader** represents. Its value is **protoMsgType-AUTHN-HEADER**.

- **authnHdr-Flags**

  Selected authentication header options. The **AuthnHeaderFlags** data type is defined in Section 4.6.1 on page 203. In Section 4.13.1 on page 232, when a client creates an authentication header its options are considered to be deselected by default unless they are explicitly selected.

- **authnHdr-Tkt**

  The ticket, $Tkt_{A,B}$, associated with the client-server session that this authentication header is authenticating.

- **authnHdr-EncryptedAuthnr**

  The encryption type (**encData-EncType**), key version number (**encData-KeyVersNum**) and ciphertext (**encData-CipherText**) encryption of (the underlying BER-encoded bit-string of) the authenticator (of type **Authenticator**) associated with the client-server session that this authentication header is authenticating. The ''pre-encrypted'' plaintext of this field (which is a value of the data type **Authenticator**) is denoted by **authnHdr-EncryptAuthnr**.

Of course, it is the final two fields that are the most significant — and for that reason, an authentication header is sometimes referred to as a ''ticket and authenticator'' (both of which are cryptographically protected).

4.6.1 Authentication Header Flags

[RFC 1510: 5.2, 5.5.1]

The options that may be selected by an authentication header (that is, specified by the client *A* named by $Tkt_{A,B}$ (**authnHdr-Tkt**), or interpreted by the targeted server *B*) are represented by the **AuthnHeaderFlags** data type, which is defined as follows:

```
AuthnHeaderFlags ::= BIT STRING {
        authnHdr-UseSessionKey  (1),
        authnHdr-MutualRequired (2)
}
```

The semantics of these bits are that if a value of type **AuthnHeaderFlags** has the corresponding bit set then the option is selected; if the bit is reset, the option is deselected. The bits represent the following options (bits not currently specified are reserved for future usage):

- **authnHdr-UseSessionKey**

  Usually (that is, if this option is deselected), $Tkt_{A,B}$ is protected with *B*'s long-term key, K_B. But if this option is selected, then the $Tkt_{A,B}$ (**authnHdr-Tkt**) in this authentication header is protected with a session key, $K^{\bullet}$, carried in an auxiliary (ticket-granting-)ticket targeted to *B*, $Tkt^{\bullet}$. For more explanation, see Section 4.6.2 (however, the explanation there is slight, since the use-session-key option is not used in the internal KDS protocol).

- **authnHdr-MutualRequired**

  A expects *B* to return a reverse-authentication header (of data type **RevAuthnHeader**), thereby completing *mutual authentication* by ''authenticating the server to the client'' (see Section 4.7 on page 205).

4.6.2 The Use-session-key Option

[RFC 1510: 3.2.3]

The use-session-key option is not used anywhere in DCE. therefore a detailed explanation of it would lead too far afield and is not appropriate here. However, a rough explanation of how it might be used at application level is given in this section, as an indication of its potential. Such an application-level usage is said to be a ''user-to-user authentication protocol''. (In this section a shorthand notation is used; for example, omitting ticket fields that are unnecessary for the purposes here.)

Recall (see Section 1.5 on page 18) that the basic Kerberos protocol for a client *A* to authenticate to a server *B* begins by having the client *A* obtain a session key $K_{A,KDS}$ and a ticket, $Tkt_{A,KDS}$, from the AS, and then proceeds with the following series of exchanges (retaining here *only those terms* that are critical to the discussion at hand):

- $A \rightarrow$ TGS: $B, Tkt_{A,KDS}$
- $A \leftarrow$ TGS: $\{B, K_{A,B}\}\ K^{[\wedge]}$ $_{A,KDS}, Tkt_{A,B}$
- $A \rightarrow B$: $Tkt_{A,B}$
- ··· *and so on*, ···

where $Tkt_{A,B} = \{A, K_{A,B}\}K_B$. The main point to focus on for the purposes of this discussion is that $Tkt_{A,B}$ is protected with the *long-term* key K_B of *B*. This requires, in order for *B* to be able to decrypt $Tkt_{A,B}$, that *B* ''*knows*'' (*has access to*) *its long-term key*. But the requirement of having its long-term key readily accessible could be a risk for *B*, especially if the machine on which *B* runs is not physically secured (for example, a ''public workstation'').

The above protocol can be modified so that $Tkt_{A,B}$ is replaced by another ticket, $Tkt^{\bullet}_{A,B}$, which is protected with a *short-term* (session) key (whose compromise represents a much smaller risk), $K^{\bullet} = K_{B,KDS}$, as follows. Suppose that *A* has obtained a copy of *B*'s (ticket-granting-)ticket, $Tkt_{B,KDS}$, (= $\{B, K_{B,KDS}\}K_{KDS}$); note this is not a security risk to *A* or *B*, and the manner of *A*'s obtaining *B*'s ticket need not be secure — for example, *B* might have sent a copy of $Tkt_{B,KDS}$ to *A*, or *B* might have ''published'' its $Tkt_{B,KDS}$ in a public place (such as in a directory service datastore) and *A* retrieved a copy of it. Then the required protocol can be achieved by:

- $A \rightarrow$ TGS: $B, Tkt_{A,KDS}, Tkt_{B,KDS}$
- $A \leftarrow$ TGS: $\{K_{A,B}\}\ K^{[\wedge]}$ $_{A,KDS}, Tkt^{\bullet}_{A,B}$
- $A \rightarrow B$: $Tkt^{\bullet}_{A,B}$
- ··· *and so on,* ···

where $Tkt_{B,KDS}$ (also denoted ''$Tkt^{\bullet}$'', without subscripts, in this context) is an ''additional ticket'' (trusted by *B*) that conveys to the KDS (via the **use-session-key** option) the session key $K_{B,KDS}$ (also denoted ''$K^{\bullet}$'', without subscripts, in this context) that is used to protect $Tkt^{\bullet}_{A,B}$; that is, $Tkt^{\bullet}_{A,B} = \{A, K_{A,B}\}K^{\bullet}$.

As presented in this brief discussion, the advantage of using $Tkt^{\bullet}_{A,B}$ instead of $Tkt_{A,B}$ is by no means apparent. But it becomes a powerful tool when embedded in an overall architecture of so-called *secret-key certificates* (as opposed to *public-key certificates*), here implemented as tickets used in new ways, because it permits many of the advantages of *public-key protocols* to be implemented using *secret-key encryption*. The deeper exploration of this is, however, beyond the scope of this specification.

4.7 Reverse-authentication Headers

[RFC 1510: 5.5.2]

Reverse-authenticator headers are represented by the **RevAuthnHeader** data type, which is defined as follows (where **APPLICATION 15** indicates **protoMsgType-REVAUTHN-HEADER** — see Section 4.2.2.1 on page 166):

```
RevAuthnHeader ::= [APPLICATION 15] SEQUENCE {
        revAuthnHdr-ProtoVersNum  [0] ProtocolVersionNumber,
        revAuthnHdr-ProtoMsgType  [1] ProtocolMessageType,
        revAuthnHdr-EncryptedPart [2] EncryptedData
}
```

Its semantics are that it supplies the *reverse authentication information* that actually ''authenticates a server *B* to a client *A*'', by extracting certain information from a corresponding authentication header (that the client trusts is only accessible by the legitimate server) and returning it (securely) to the client. Its fields are the following:

- **revAuthnHdr-ProtoVersNum**

  The protocol version number of the **RevAuthnHeader** data type. Its value is **protoVersNum-KRB5**.

- **revAuthnHdr-ProtoMsgType**

  The kind of protocol message this **RevAuthnHeader** represents. Its value is **protoMsgType-REVAUTHN-HEADER**.

- **revAuthnHdr-EncryptedPart**

  The encryption type (**encData-EncType**), key version number (**encData-KeyVersNum**) and ciphertext (**encData-CipherText**) encryption of (the underlying BER-encoded bit-string of) a value of type **RevAuthnHeaderEncryptPart**, which is defined in Section 4.7.1. The ''pre-encrypted'' plaintext of this field (which is a value of the data type **RevAuthnHeaderEncryptPart**) is denoted by **revAuthnHdr-EncryptPart**.

4.7.1 Part of Reverse-authentication Header to be Encrypted

[RFC 1510: 5.5.2]

The part of a reverse-authentication header to be encrypted is represented (before it is encrypted) by the **RevAuthnHeaderEncryptPart** data type, which is defined as follows:

```
RevAuthnHeaderEncryptPart ::= [APPLICATION 27] SEQUENCE {
        revAuthnHdr-ClientTime      [0] TimeStamp,
        revAuthnHdr-ClientMicroSec  [1] MicroSecond,
        revAuthnHdr-ConversationKey [2] EncryptionKey OPTIONAL,
        revAuthnHdr-SeqNum          [3] SequenceNumber OPTIONAL
}
```

Its fields are the following:

- **revAuthnHdr-ClientTime**

  The corresponding authentication header's client timestamp (**authnr-ClientTime**).

- **revAuthnHdr-ClientMicroSec**

  The corresponding authentication header's client microsecondstamp (**authnr-ClientMicroSec**).

- **revAuthnHdr-ConversationKey**

  B's choice of a conversation key, $K\hat{\hat{}}_{A,B}$, indicating that *B* prefers this key to be used to protect client-server communications instead of either the session key $K_{A,B}$ in the corresponding authentication header's $Tkt_{A,B}$ (**tkt-SessionKey**), or the client's choice of conversation key $K\hat{}_{A,B}$ (**authnr-ConversationKey**) if present, as part of conversation key negotiation between *A* and *B*.

- **revAuthnHdr-SeqNum**

  B's indication of sequence number. It is equal to the **authnr-SeqNum** field of the corresponding authentication header (or is absent if that field was omitted).

4.8 KDS (AS and TGS) Requests

[RFC 1510: 5.4.1]

AS Requests are represented by the **ASRequest** data type, and TGS Requests are represented by the **TGSRequest** data type. These are defined in terms of a common underlying data type, **KDSRequest**, as follows (where **APPLICATION 10** indicates **protoMsgType-AS-REQUEST**, and **APPLICATION 12** indicates **protoMsgType-AS-RESPONSE** — see Section 4.2.2.1 on page 166):

```
ASRequest ::= [APPLICATION 10] KDSRequest
TGSRequest ::= [APPLICATION 12] KDSRequest

KDSRequest ::= SEQUENCE {
        req-ProtoVersNum [1] ProtocolVersionNumber,
        req-ProtoMsgType [2] ProtocolMessageType,
        req-AuthnData    [3] SEQUENCE OF AuthnData OPTIONAL,
        req-Body         [4] KDSRequestBody
}
```

Its semantics are to indicate a calling client *A*'s request to a KDS server KDS_Z to return a $Tkt_{A,B}$ targeted to a server *B* (where *B* may be another KDS server, KDS_Z.). KDS servers (such as KDS_Z) support requests for the following two distinct kinds of services:

- *Request for a ''(manipulated) old'' Tkt*

  (Re-)issue a ticket that has substantially previously existed, by manipulating a presented ticket (in **req-AuthnData**). (A rigorous definition is given in Section 4.14.1 on page 240.) The old ticket may be of almost any type (the exception being that ticket-granting-tickets cannot be proxied), and the manipulation may be any one of the following:

  — *Validation*

    Validate an (invalid) ticket.

  — *Renewal*

    Renew a (renewable) ticket.

  — *Forwarding*

    Forward a (forwardable) (ticket-granting- or service-)ticket.

  — *Proxying*

    Proxy a (proxiable) (non-ticket-granting-)service-ticket.

  Note: Some implementations may support the simultaneous combination of proxying and fowarding, because these do not really conflict with one another, but other combinations are more problematic (validation conflicts with forwarding/proxying, renewal conflicts with validation/fowarding/proxying), so an implementation that supported those combinations would have to specify their semantics (in an implementation-specific manner).

- *Request for a ''new'' ticket*

  Issue a ticket that hasn't substantially previously existed. (A rigorous definition is given in Section 4.14.1 on page 240.) The new ticket may of any type (initial or subsequent, ticket-granting-ticket or service-ticket).

The fields of **KDSRequest** are the following:

- **req-ProtoVersNum**

  The protocol version number of the **KDSRequest** data type. Its value is **protoVersNum-KRB5**.

- **req-ProtoMsgType**

  The kind of protocol message this **KDSRequest** represents. If this is an AS Request, its value is **protoMsgType-AS-REQUEST**; if a TGS Request, it is **protoMsgType-TGS-REQUEST**.

- **req-AuthnData**

  This KDS Request's *authentication data* — it has different semantics depending on exactly what kind of KDS Request this is:

  1. If this is an AS Request, this field is not present (currently — if it were present, it would be called ''pre-authentication'' data).
  2. If this is a TGS Request for a new ticket, this field authenticates the calling client *A* to the KDS server KDS_Z, by containing one element of authentication data of type (**authnData-Type**) **authnData-TGS-REQ**, whose value (**authnData-Value**) contains (therefore) a ticket and an authentication header (value of type **AuthnHeader**) whose checksum (**authnr-Cksum**) contains the checksum of **req-Body** (see Section 4.14.1 on page 240).
  3. If this is a TGS Request for an old ticket, this field contains the old ticket that is to be manipulated.

- **req-Body**

  The body of this **KDSRequest**, of type **KDSRequestBody**, which is defined in Section 4.8.1.

4.8.1 KDS Request Body

[RFC 1510: 5.4.1]

KDS Request message bodies are represented by the **KDSRequestBody** data type, which is defined as follows:

```
KDSRequestBody ::= SEQUENCE {
        req-Flags              [ 0] KDSRequestFlags,
        req-ClientName         [ 1] RSName OPTIONAL,
        req-ServerCell         [ 2] CellName,
        req-ServerName         [ 3] RSName,
        req-StartTime          [ 4] TimeStamp OPTIONAL,
        req-ExpireTime         [ 5] TimeStamp,
        req-MaxExpireTime      [ 6] TimeStamp OPTIONAL,
        req-Nonce              [ 7] Nonce,
        req-EncTypes           [ 8] SEQUENCE OF EncryptionType,
        req-ClientAddrs        [ 9] HostAddresses OPTIONAL,
        req-EncryptedAuthzData [10] EncryptedData OPTIONAL,
        req-AdditionalTkts     [11] SEQUENCE OF Ticket OPTIONAL
}
```

Its fields are the following:

- **req-Flags**

KDS options selected by this KDS Request (requested by the calling client *A*, or interpreted by the KDS server KDS_Z). The **KDSRequestFlags** data type is defined in Section 4.8.2 on page 210. (In Section 4.12.1 on page 220 and Section 4.14.1 on page 240, when a client creates a KDS request its options are considered to be deselected by default unless they are explicitly selected.)

- **req-ClientName**

  A's RS name in its cell.

  Note: There is no field in **KDSRequestBody** for *A*'s cell name — it is unnecessary according to the AS and TGS Request/Response protocols.)

- **req-ServerCell**

  B's cell.

- **req-ServerName**

  B's RS name in its cell.

- **req-StartTime**

  A's desired start time for a new postdated ticket.

- **req-ExpireTime**

  A's desired expiration time for a new ticket. The particular value **endOfTimeStamp** (see Section 4.2.3 on page 167) is interpreted specially by the KDS server, namely as a request for a ticket with the largest expiration time it will issue, according to its policy.

  Note: This interpretation of **endOfTimeStamp** is not in RFC 1510.)

- **req-MaxExpireTime**

  A's desired absolute expiration time for a new renewable ticket.

  Note: For this field, **endOfTimeStamp** does not have a special interpretation.)

- **req-Nonce**

  Nonce generated by *A* (to be returned in KDS Response, defined in Section 4.9 on page 212).

- **req-EncTypes**

  A list of *A*'s preferences of encryption types to be used to protect a new ticket.

- **req-ClientAddrs**

  A's desired client host addresses for a new (potentially proxiable and/or forwardable) ticket.

- **req-EncryptedAuthzData**

  The encryption type (**encData-EncType**), key version number (**encData-KeyVersNum**) and ciphertext (**encData-CipherText**) encryption of (the underlying BER-encoded bit-string of) a value of type **AuthzData**. It is used to indicate *A*'s request for authorisation data. If this is an AS Request, it is not present. (See Section 4.14.1 on page 240 and Section 5.4.1 on page 292 for its use with the PS.) The ''pre-encrypted'' plaintext of this field (a value of the data type **AuthzData**) is denoted by **req-EncryptAuthzData**.

- **req-AdditionalTkts**

  Additional tickets, to be used in conjunction with this KDS Request's selected KDS options (**req-Flags**) which require such additional tickets. Each option that requires an additional ticket requires exactly one such additional ticket, and the list **req-AdditionalTkts** is in one-

to-one correspondence with such selected option(s), in the same order as the ordinal number(s) of such selected option(s). Currently, there is only one option that requires an additional ticket: the use-session-key option (**req-UseSessionKey**), which requires a $Tkt^{\bullet}$ to carry the session key $K^{\bullet}$ indicated by the option (see Section 4.6.1 on page 203). (Additional tickets are not used in the internal KDS protocol.)

4.8.2 KDS Request Flags

[RFC 1510: 5.2, 5.4.1]

The options that may be requested in a KDS Request are represented by the **KDSRequestFlags** data type, which is defined as follows:

```
KDSRequestFlags ::= BIT STRING {
        req-Forwardable   ( 1),
        req-Forward       ( 2),
        req-Proxiable     ( 3),
        req-Proxy         ( 4),
        req-Postdatable   ( 5),
        req-Postdate      ( 6),
        req-Renewable     ( 8),
        req-RenewableOK   (27),
        req-UseSessionKey (28),
        req-Renew         (30),
        req-Validate      (31)
}
```

The semantics of these bits are that if a value of type **KDSRequestFlags** has the corresponding bit set then the option is selected; if the bit is reset, the option is deselected. The bits represent the following options (bits not currently specified are reserved for future usage):

- **req-Forwardable**

  New ticket is to be forwardable.

- **req-Forward**

  Old ticket is to be forwarded.

- **req-Proxiable**

  New ticket is to be proxiable.

- **req-Proxy**

  Old ticket is to be proxied.

- **req-Postdatable**

  New ticket is to be postdatable.

- **req-Postdate**

  New ticket is to be postdated.

- **req-Renewable**

  New ticket is to be renewable.

- **req-RenewableOK**

New ticket is preferred to be non-renewable and have *A*'s desired maximum lifetime, but if KDS_Z declines to issue such a ticket then *A* will accept a renewable ticket whose maximum lifetime is as close as possible to (but not exceeding) the desired maximum lifetime.

- **req-UseSessionKey**

  New ticket is to be protected with the session key (**tkt-SessionKey**) of a presented additional ticket (**req-AdditionalTkts**), instead of with *B*'s long-term key. In order to use this option, *A* must somehow acquire possession of an appropriate additional ticket. Internally (in the KDS protocols), this option is not used.

- **req-Renew**

  Old ticket is to be renewed.

- **req-Validate**

  Old ticket is to be validated.

4.9 KDS (AS and TGS) Responses

[RFC 1510: 5.4.2]

AS Responses are represented by the **ASResponse** data type, and TGS Responses are represented by the **TGSResponse** data type. These are defined in terms of a common underlying data type, **KDSResponse**, which is defined as follows (where **APPLICATION 11** indicates **protoMsgType-TGS-REQUEST**, and **APPLICATION 13** indicates **protoMsgType-TGS-RESPONSE** — see Section 4.2.2.1 on page 166):

```
ASResponse ::= [APPLICATION 11] KDSResponse
TGSResponse ::= [APPLICATION 13] KDSResponse

KDSResponse ::= SEQUENCE {
        resp-ProtoVersNum  [0] ProtocolVersionNumber,
        resp-ProtoMsgType  [1] ProtocolMessageType,
        resp-AuthnData     [2] AuthnData OPTIONAL,
        resp-ClientCell    [3] CellName,
        resp-ClientName    [4] RSName,
        resp-Tkt           [5] Ticket,
        resp-EncryptedPart [6] EncryptedData
}
```

Its semantics are to indicate the called KDS_Z's response to a client *A*'s request for a ticket. Its fields are the following:

- **resp-ProtoVersNum**

  The protocol version number of the **KDSResponse** data type. Its value is **protoVersNum-KRB5**.

- **resp-ProtoMsgType**

  The kind of protocol message this **KDSResponse** represents. If this is an AS Response, its value is **protoMsgType-AS-RESPONSE**; if a TGS Response, it is **protoMsgType-TGS-RESPONSE**.

- **resp-AuthnData**

  This KDS Response's **authentication data**. It is used to return information to *A*. (See Section 5.4.2 on page 293 for its use with the PS, where it is actually used to return *authorisation* data, not ''authentication data''.)

- **resp-ClientCell**

  A's cell.

- **resp-ClientName**

  A's RS name in its cell.

- **resp-Tkt**

  The issued ticket. If this is an AS Response, it is an initial ticket. If this is a TGS Response, it is a subsequent ticket, and it is a new or old ticket depending on the TGS Request options selected. A KDS server KDS_Z always returns a ticket whose named client is the requested one (however in the case of an AS Request, which is unauthenticated, the message itself could be modified in transit, so the message *A* sends may not be the message KDS_Z receives). It also usually returns a ticket whose targeted server *B* is the requested one, the only exception being in the case that *B*'s home cell is not *Z*, in which KDS_Z returns a special cross-

cell referral ticket (see Section 4.14.1 on page 240 and Section 4.14.2 on page 245).

- **resp-EncryptedPart**

  The encryption type (**encData-EncType**), key version number (**encData-KeyVersNum**) and ciphertext (**encData-CipherText**) encryption of (the underlying BER-encoded bit-string of) a value of type **KDSResponseEncryptPart**, which is defined in Section 4.9.1. The ''pre-encrypted'' plaintext of this field (a value of the data type **KDSResponseEncryptPart**) is denoted by **resp-EncryptPart**.

4.9.1 Part of KDS Response to be Encrypted

[RFC 1510: 5.4.2]

The encrypted data carried in a KDS Response is represented (before it is encrypted) by the **KDSResponseEncryptPart** data type, which is defined as follows:

```
ASResponseEncryptPart ::= [APPLICATION 25] KDSResponseEncryptPart
TGSResponseEncryptPart ::= [APPLICATION 26] KDSResponseEncryptPart

KDSResponseEncryptPart ::= SEQUENCE {
        resp-SessionKey    [ 0] EncryptionKey,
        resp-LastRequests  [ 1] LastRequests,
        resp-Nonce         [ 2] Nonce,
        resp-KeyExpireDate [ 3] TimeStamp OPTIONAL,
        resp-Flags         [ 4] TicketFlags,
        resp-AuthnTime     [ 5] TimeStamp,
        resp-StartTime     [ 6] TimeStamp OPTIONAL,
        resp-ExpireTime    [ 7] TimeStamp,
        resp-MaxExpireTime [ 8] TimeStamp OPTIONAL,
        resp-ServerCell    [ 9] CellName,
        resp-ServerName    [10] RSName,
        resp-ClientAddrs   [11] HostAddresses OPTIONAL
}
```

Its fields are the following. Note that most of this duplicates information that is present in the enclosed $\text{Tkt}_{A,B}$ (**resp-Tkt**), so that *A* can check its conformance to what it had requested in the corresponding KDS Request (*A* cannot actually decrypt $\text{Tkt}_{A,B}$ itself, unless *A* happens to know the long-term key of the targeted server *B*). (The only information in $\text{Tkt}_{A,B}$ that is not repeated in **KDSResponseEncryptPart** are its transit path and authorisation data.)

- **resp-SessionKey**

  Duplicate of **resp-Tkt**'s session key (**tkt-SessionKey**).

- **resp-LastRequests**

  Last request information that KDS_X (*A*'s home KDS server) has pertaining to client *A*. Typically only usefully present in an AS Response. It is legal in a TGS Response (in fact, it's not an optional field, though implementations typically support a policy of returning only an *empty* array of **resp-LastRequests** in TGS Responses), and some implementations may indeed return last request information in TGS Responses, but it's not normally useful there — the last request information is normally intended to be displayed to the user, for example, at login-time, but most service-level applications don't do that.

- **resp-Nonce**

Copy of the corresponding KDS Request's nonce (**req-Nonce**).

- **resp-KeyExpireDate**

  The time at which *A*'s long-term key K_A (of the selected encryption type, held in RS_X) is scheduled to **expire**; that is, later than which (modulo **maxClockSkew**) KDS_X (*A*'s home KDS server) will not use it for protecting AS Responses and tickets targeted to *A* (which are the only data KDS servers protect with long-term keys). This supports principal long-term key management (subject to local policy). Typically only present in AS Responses, not TGS Responses.

- **resp-Flags**

  Duplicate of **resp-Tkt**'s ticket flags (**tkt-Flags**).

- **resp-AuthnTime**

  Duplicate of **resp-Tkt**'s authentication time (**tkt-AuthnTime**).

- **resp-StartTime**

  Duplicate of **resp-Tkt**'s start time, if present (**tkt-StartTime**).

- **resp-ExpireTime**

  Duplicate of **resp-Tkt**'s expiration time (**tkt-ExpireTime**).

- **resp-MaxExpireTime**

  Duplicate of **resp-Tkt**'s absolute expiration time, if present (**tkt-MaxExpireTime**).

- **resp-ServerCell**

  B's cell.

- **resp-ServerName**

  B's RS name in its cell.

- **resp-ClientAddrs**

  Duplicate of **resp-Tkt**'s client host addresses, if present (**tkt-ClientAddrs**).

4.10 KDS Errors

[RFC 1510: 5.9.1]

KDS Errors are represented by the **KDSError** data type, which is defined as follows (where **APPLICATION 30** indicates **protoMsgType-KDS-ERROR** — see Section 4.2.2.1 on page 166):

```
KDSError ::= [APPLICATION 30] SEQUENCE {
        err-ProtoVersNum   [ 0] ProtocolVersionNumber,
        err-ProtoMsgType   [ 1] ProtocolMessageType,
        err-ClientTime     [ 2] TimeStamp OPTIONAL,
        err-ClientMicroSec [ 3] MicroSecond OPTIONAL,
        err-ServerTime     [ 4] TimeStamp,
        err-ServerMicroSec [ 5] MicroSecond,
        err-StatusCode     [ 6] ErrorStatusCode,
        err-ClientCell     [ 7] CellName OPTIONAL,
        err-ClientName     [ 8] RSName OPTIONAL,
        err-ServerCell     [ 9] CellName,
        err-ServerName     [10] RSName,
        err-StatusText     [11] ErrorStatusText OPTIONAL,
        err-StatusData     [12] ErrorStatusData OPTIONAL
}
```

Its semantics are that it indicates diagnostic information returned from a server *C* to a calling client *A* concerning a failed (in the security sense) service request. The primary usage is when *C* is a KDS server KDS_Z, but it can also be used by applications if they so desire. In any case, KDS Error messages are *unprotected*, therefore the information carried in them must be viewed with suspicion in a hostile environment. Its fields are the following:

- **err-ProtoVersNum**

  The protocol version number of this **KDSError** data type. Its value is **protoVersNum-KRB5**.

- **err-ProtoMsgType**

  The kind of protocol message this **KDSError** represents. Its value is **protoMsgType-KDS-ERROR**.

- **err-ClientTime**

  A's timestamp from accompanying authenticator (**authnHdr-EncryptAuthnr.authnr-ClientTime**), if present. Otherwise, it is absent.

- **err-ClientMicroSec**

  A's microsecondstamp from accompanying authenticator (**authnHdr-EncryptAuthnr.authnr-ClientMicroSec**), if present. Otherwise, it is absent.

- **err-ServerTime**

  C's system time at which the error occurred.

- **err-ServerMicroSec**

  C's system microsecond time at which the error occurred.

- **err-StatusCode**

  Status code identifying the kind of error that occurred.

- **err-ClientCell**

  A's cell from accompanying ticket (**tkt-EncryptPart.tkt-ClientCell**), if present (not from accompanying authenticator, **authnHdr-EncryptAuthnr.authnr-ClientCell**, if present). Otherwise, it is absent.

- **err-ClientName**

  A's RS name from accompanying ticket (**tkt-EncryptPart.tkt-ClientName**, if present (not from accompanying authenticator, **authnHdr-EncryptAuthnr.authnr-ClientName**, if present). Otherwise, it is absent.

- **err-ServerCell**

  C's cell.

- **err-ServerName**

  C's RS name in its cell.

- **err-StatusText**

  Status text associated to **err-StatusCode**.

- **err-StatusData**

  Status data associated to **err-StatusCode**.

4.11 RS Information

[RFC 1510: 4]

Every KDS_Z requires access to certain (non-volatile) information. Such information is held in the RS_Z datastore, not in the KDS_Z itself. RS_Z maintains local cell-wide **property** and **policy** information, as well as information pertaining to individual principals, relevant to KDS_Z's processing of KDS Requests (below). These information items, together with the KDS Error status code values associated with them, are the following:

- *Cell name*

  This cell's name.

- *KDS server's RS name*

  RS name of the KDS server in this cell. If the cell name of this cell *Z* is, say, **cellZ**, then the RS_Z name of KDS_Z is **krbtgt/cellZ**.

- *Supported protocol version numbers*

  The protocol version numbers supported by KDS_Z {**errStatusCode-BAD-PROTO-VERS-NUM**}.

- *Supported authentication methods*

  The authentication methods supported by KDS_Z {**errStatusCode-BAD-AUTHN-METHOD**}.

- *Supported authentication data types*

  The authentication data types supported by KDS_Z {**errStatusCode-AUTHN-DATA-TYPE-NOT-SUPPORTED**}.

- *Supported transit path types*

  The transit path types supported by KDS_Z {**errStatusCode-TRANSIT-PATH-TYPE-NOT-SUPPORTED**}.

- *Supported encryption key types*

  The encryption key types supported by KDS_Z.

- *Supported encryption types*

  The encryption types supported by KDS_Z.

- *Supported checksum types*

  The checksum types supported by KDS_Z {**errStatusCode-CHECKSUM-TYPE-NOT-SUPPORTED**, **errStatusCode-BAD-CHECKSUM-TYPE**}.

- *Per-end-principal RS names*

  Entries for end-principals (that is, non-KDS-principals) stored in RS_Z's datastore.

- *Per-foreign-KDS RS names*

  Entries for KDS principals $KDS_{Z,Z'}$ from foreign cells *Z´* cross-registered with cell *Z*. If the cell name of foreign cell *Z´* is, say, **cellZ´**, then the RS_Z name of $KDS_{Z,Z'}$ is **krbtgt/cellZ´** {**errStatusCode-NOT-US**}.

- *Per-principal long-term key(s), with version numbers*

  One (or more, as distinguished by their key version numbers) key(s) for each encryption type supported. ''The'' key (modulo its version number, and eliding the encryption type from the

notation) for principal *C* and is denoted K_C {**errStatusCode-CLIENT-OLD-MASTER-KEY-VERS-NUM**, **errStatusCode-SERVER-OLD-MASTER-KEY-VERS-NUM**, **errStatusCode-NULL-KEY**, **errStatusCode-SERVER-NO-KEY**, **errStatusCode-BAD-KEY-VERS-NUM**}.

- *Per-principal ''salt''*

  Information used by principals to use in combination with their passwords (which are not, in general, stored in the RS datastore), in order to derive their long-term keys (see Section 4.3.6.1 on page 190).

- *Conversation key negotiation*

  Boolean indicating whether KDS_Z will accept client-supplied (in authentication headers, **authnHdr-EncryptAuthnr.authnr-ConversationKey**) conversation keys be used to protect client-KDS sessions, or KDS_Z will insist on supplying them itself (in reverse-authentication headers, **revAuthnHdr-ConversationKey**).

- *Maximum clock skew*

  Value(s) (certainly a cell-value one, though potentially also per-principal values) of **maxClockSkew** (see Section 4.2.3 on page 167), such that the authentication protocols specified herein fail if the KDS (or PS) encounter a timestamp from a principal whose clock skew compared to the KDS's (or PS's) clock exceeds **maxClockSkew** {**errStatusCode-CLOCK-SKEW**}.

- *Per-principal long-term key(s) expiration dates(s)*

  The time later than which (modulo **maxClockSkew**) KDS_Z will not use a principal's long-term key for protecting AS Responses and tickets targeted to *C* (which are the only data KDS servers protect with long-term keys).

- *Postdatability permitted*

  Boolean indicating whether or not KDS_Z will issue new postdatable or postdated tickets {**errStatusCode-CANNOT-POSTDATE**}.

- *Renewability permitted*

  Boolean indicating whether or not KDS_Z will issue new renewable tickets.

- *Proxiability permitted*

  Boolean indicating whether or not KDS_Z will issue new proxiable tickets.

- *Forwardability permitted*

  Boolean indicating whether or not KDS_Z will issue new forwardable tickets.

- *Cell-wide minimum ticket lifetime*

  Minimum lifetime for which KDS_Z will issue a new ticket. (A ticket request that would result in a ticket with a lifetime less than this minimum will be rejected by the KDS.) {**errStatusCode-NEVER-VALID**}.

- *Cell-wide default ticket-granting-ticket lifetime*

  Default lifetime for which KDS_Z will issue a new ticket-granting-ticket.

- *Cell-wide maximum ticket lifetime*

  Maximum lifetime for which KDS_Z will issue a new ticket.

- *Per-principal maximum ticket lifetime*

Maximum lifetime for which KDS_Z will issue a new ticket naming or targeting a principal *C*.

- *Cell-wide postdate maximum*

  Furthest date in the future for which KDS_Z will issue a postdatable ticket naming or targeting a principal *C*.

- *Per-principal postdate maximum*

  Furthest date in the future for which KDS_Z will issue a postdatable ticket.

- *Cell-wide maximum renewable ticket lifetime*

  Maximum lifetime for which KDS_Z will issue a new renewable ticket (if it issues new renewable tickets at all).

- *Per-principal maximum renewable ticket lifetime*

  Maximum lifetime for which KDS_Z will issue a new renewable ticket naming or targeting a principal *C* (if it issues new renewable tickets at all).

- *Client addresses*

  Boolean indicating whether KDS_Z requires client addresses (**tkt-ClientAddrs**) to be present in all tickets.

- *Per-principal last request(s)*

  Last request(s) information maintained for *C*, per local policy.

- *Replay cache*

  Replay cache used by KDS_Z (see Section 4.5 on page 200) {**errStatusCode-REPLAY**}.

- *Hot list (revocation) information*

  Information about (potentially) compromised entities (clients, servers, tickets, and so on), which have therefore been *revoked* (no longer supported by the security services). These entities comprise RS_Z's *hot list*. For example, whenever a hot-listed (revoked) ticket is presented to KDS_Z in (the **req-AuthnData** field of) a TGS Request, KDS_Z will refuse to honour it. (Note that when a ticket's maximum expiration time has passed, KDS_Z will not honour it under any circumstances, so there is no need to keep such tickets on the hot list.) {**errStatusCode-CLIENT-REVOKED**, **errStatusCode-SERVER-REVOKED**, **errStatusCode-TKT-REVOKED**}.

- *Next hops*

  Information about what cell a client should visit next (among those cross-registered with *Z*) if the server requested by the client in a TGS Request is not registered in RS_Z. There are various (policy-dependent) strategies for determining next hops. For example, some links may be more trusted than others, hence more suitable for some purposes than others.

  A common next hop strategy is the following *up-over-down* algorithm. If the server requested by a client is not registered in *Z*, then the next hop is:

  1. the server's cell, if *Z* holds a direct cross-registration to the server's cell, otherwise
  2. the first ancestor cell (in the namespace sense) of the server's cell which is cross-registered with *Z*, if *Z* holds such a cross-registration, otherwise
  3. the parent cell of *Z*, if *Z* holds a cross-registration with it.

4.12 AS Request/Response Processing

[RFC 1510: 1, 3.1]

This section specifies in detail the processing that occurs during an AS Request/Response exchange; that is, this section specifies the issuing of new initial tickets. There are three steps involved:

1. A client prepares an AS Request and sends it to a KDS server.
2. A KDS server receives the AS Request from a client, processes it, prepares an AS Response (success case) or KDS Error (failure case), and returns that to the client.
3. A client receives an AS Response or KDS Error.

The details of the three steps of the success case are specified next. (For the failure case, see Section 4.15 on page 258.)

4.12.1 Client Sends AS Request to KDS

[RFC 1510: 3.1.1, A.1]

Consider a client *A* that wants to obtain an initial ticket, $Tkt_{A,B}$, from KDS_X, where *X* is *A*'s home cell. (Usually, though not necessarily, an initial ticket is a ticket-granting-ticket; that is, the target server *B* will usually be KDS_X itself. In any case, *B* must be in cell *X*, or the AS Request will fail.) Then *A* prepares an AS Request, *asReq* (a value of the data type **ASRequest**), and ''sends it'' (that is, calls *kds_request*( )) to KDS_X, according to the following algorithm. Note that it is *A*'s responsibility to know (or to securely determine) all the information necessary to correctly formulate the AS Request message — especially, KDS_X's cell name and RS name (that's why ''well-known principal names'' (involving the component **krbtgt**) are used for KDS servers), as well as the RS names of *A* itself and of *B*. Since the AS Request is *unauthenticated*, KDS_X cannot know with certainty the principal identity of this calling client, *A* — in particular, *A* may request (or its request may be modified in transit) that the initial ticket in question be issued ''in the name of'' (that is, name) a client *A´* other than *A* itself (though in that case the resulting $Tkt_{A´,KDSX}$ will be unusable by *A*, unless *A* knows *A´*'s long-term key — except perhaps for cryptanalysis; for example, for an ''offline dictionary attack'').

- *Protocol version number*

  The protocol version number (*asReq***.req-ProtoVersNum**) is set to **protoVersNum-KRB5**.

- *Protocol message type*

  The protocol message type (*asReq***.req-ProtoMsgType**) is set to **protoMsgType-AS-REQUEST**.

- *Client name*

  The client name (*asReq***.req-Body.req-ClientName**) is set to *A*'s RS name in RS_X.

- *Server cell*

  The server cell (*asReq***.req-Body.req-ServerCell**) is set to KDS_X's cell name; that is, to the name of the cell *X*, say **cellX**. This is also *A*'s cell name — a KDS server can issue initial tickets naming only clients in its own cell, and targeted only to servers in its own cell.

- *Server name*

  The server name (*asReq***.req-Body.req-ServerName**) is set to *B*'s RS name in its cell. (In the usual case of an AS Request for an initial ticket-granting-ticket; that is, *B* = KDS_X, this RS name will be **krbtgt/cellX**, assuming that the cell name is **cellX**).

- *Options*
  - — *Forwardable*

    If *A* desires that $Tkt_{A,B}$ be forwardable, the forwardable option (*asReq***.req-Body.req-Flags.req-Forwardable**) is selected.
  - — *Proxiable*

    If *A* desires that $Tkt_{A,B}$ be proxiable, the proxiable option (*asReq***.req-Body.req-Flags.req-Proxiable**) is selected.
  - — *Postdatable*

    If *A* desires that $Tkt_{A,B}$ be postdatable, the postdatable option (*asReq***.req-Body.req-Flags.req-Postdatable**) is selected.
  - — *Postdate*

    If *A* desires that $Tkt_{A,B}$ be postdated, the postdate option (*asReq***.req-Body.req-Flags.req-Postdate**) is selected.
  - — *Renewable*

    If *A* desires that $Tkt_{A,B}$ be renewable, the renewable option (*asReq***.req-Body.req-Flags.req-Renewable**) is selected.
  - — *Renewable-okay*

    If *A* does not desire that $Tkt_{A,B}$ be renewable, but *A* will nevertheless accept a renewable $Tkt_{A,B}$ with a shorter lifetime than desired in lieu of no $Tkt_{A,B}$ at all, then the renewable-okay option (*asReq***.req-Body.req-Flags.req-RenewableOK**) is selected.
  - — *Other; use-session-key, validate, renew, proxy, forward*

    All of *asReq*'s options that have not been selected by any of the above steps are deselected. Currently, these include the use-session-key (*asReq***.req-Body.req-Flags.req-UseSessionKey**), validate (*asReq***.req-Body.req-Flags.req-Validate**), renew (*asReq***.req-Body.req-Flags.req-Renew**), proxy (*asReq***.req-Body.req-Flags.req-Proxy**) and forward (*asReq***.req-Body.req-Flags.req-Forward**) options.
- *Start time*

  If the postdate option for $Tkt_{A,B}$ has been selected, then the start time (*asReq***.req-Body.req-StartTime**) is set to the desired starting time. Otherwise, the start time is omitted.
- *Expiration time*

  The expiration time (*asReq***.req-Body.req-ExpireTime**) is set to the desired expiration time for $Tkt_{A,B}$.
- *Maximum expiration time*

  If the renewable option has been selected, then the maximum expiration time (*asReq***.req-Body.req-MaxExpireTime**) is set to the desired maximum expiration time for $Tkt_{A,B}$. Otherwise, the maximum expiration time is omitted.
- *Additional tickets*

  The additional tickets field (*asReq***.req-Body.req-AdditionalTkts**) is omitted.

- *Nonce*

  The nonce field (*asReq***.req-Body.req-Nonce**) is set to a nonce value.

- *Encryption types*

  The encryption types field (*asReq***.req-Body.req-EncTypes**) is set to the list of encryption types acceptable to *A* for protecting $Tkt_{A,B}$. *A* arranges this list in priority order of desirability (from *A*'s point of view), beginning with the most desirable and ending with the least desirable. (For maximum interoperability, the client *A* should send a list consisting of a single entry, indicating encryption type **encType-DES-CBC-CRC**, since all KDS servers are required to support clients requesting that single encryption type — at least for the present revision of this document.)

- *Client addresses*

  If *A* desires that $Tkt_{A,B}$ contain client host addresses, then the client address field (*asReq***.req-Body.req-ClientAddrs**) is set to the desired addresses. Otherwise, the client address field is omitted. In the present revision of DCE, clients must supply at least one address, or else the KDS will reject the AS Request.

- *Authorisation data*

  The authorisation data field (*asReq***.req-Body.req-EncryptedAuthzData**) is omitted.

- *Authentication data*

  The (pre-)authentication data (*asReq***.req-AuthnData**) is (currently) omitted.

At this point, the *asReq* message is well-formed, and *A* sends it to KDS_X.

4.12.2 KDS Server Receives AS Request and Sends AS Response

[RFC 1510: 2.1, 3.1.2, 3.1.3, A.2]

Consider an AS Request, *asReq*, received by KDS_X. That is, *asReq* is a value of type **ASRequest**, with protocol version number (*asReq***.req-ProtoVersNum**) **protoVersNum-KRB5** and protocol message type (*asReq***.req-ProtoMsgType**) **protoMsgType-AS-REQUEST**. Denote by *A* the client requested (*asReq***.req-Body.req-ClientName**) to be named in the to-be-issued initial $Tkt_{A,B}$. Then KDS_X executes the algorithm below. If the algorithm executes successfully, KDS_X prepares an AS Response (*asResp*, of type **ASResponse**) containing the newly issued initial $Tkt_{A,B}$ (*asResp***.resp-Tkt**) and ''returns'' it (that is, returns from the *kds_request*( ) invocation) to the calling client (which may be different from the requested client, *A*). If unsuccessful, KDS_X prepares a KDS Error (*kdsErr*, of type **KDSError**) and returns it to the calling client.

The following algorithm first discusses how KDS_X constructs $Tkt_{A,B}$ — also denoted *asTkt* when the emphasis is on the details of AS Request processing — and then how it constructs the rest of the AS Response, *asResp*.

- *Authentication data processing*

  The (pre-)authentication data (*asReq***.req-AuthnData**) is processed according to local policy (typically, it is absent).

- *Protocol version number*

  The new initial ticket's protocol version number (*asTkt***.tkt-ProtoVersNum**) is set to **protoVersNum-KRB5**.

- *Server cell*

The requested server cell name (*asReq***.req-Body.req-ServerCell**) is checked to be *X*'s cell name. (This is because KDS_X can issue initial tickets naming only clients in its own cell, since it must have access to their long-term key, with which it will protect *asResp*.)

- *Server name*

  The requested server name (*asReq***.req-Body.req-ServerName**) is checked to be KDS_X's RS name. KDS_X copies this into the new initial ticket's server name (*asTkt***.tkt-ServerName**). KDS_X also checks that it itself has a datastore entry in RS_X {**errStatusCode-SERVER-UNKNOWN**}.

- *Client cell*

  The client cell (*asTkt***.tkt-EncryptPart.tkt-ClientCell**) is set to *X*'s cell name (which must be *A*'s cell name, too).

- *Client name*

  The requested client name (*asReq***.req-Body.req-ClientName**) (that is, *A*'s RS name) is checked to have a datastore entry in RS_X {**errStatusCode-CLIENT-UNKNOWN**}. KDS_X copies *A*'s RS name into the new initial ticket's client name (*asTkt***.tkt-EncryptPart.tkt-ClientName**). *A* thereby becomes the client *named* by the new initial ticket $Tkt_{A,B}$.

- *Encryption types*

  KDS_X selects from the list of requested encryption types (*asReq***.req-Body.req-EncTypes**) the earliest one on the list that it is willing to accommodate (according to local policy) — call this *encType*. (Recall that the list had been generated in priority order of desirability, from *A*'s point of view. For guaranteed interoperability, all KDS servers are required to support clients requesting the single encryption type **enctype-DES-CBC-CRC** — at least for the present revision of this document.) {**errStatusCode-ENCRYPTION-TYPE-NOT-SUPPORTED**}.

- *Long-term key retrieval*

  KDS_X retrieves *A*'s most recent long-term key K_A (and its key version number, for the selected encryption type *encType*) from RS_X.

- *Session key generation*

  KDS_X generates a new (random) session key (of the selected encryption type, *encType*), $K_{A,KDSX}$, and copies it into $Tkt_{A,B}$'s session key field (*asTkt***.tkt-EncryptPart.tkt-SessionKey**).

- *Authentication time*

  $Tkt_{A,B}$'s authentication time (*asTkt***.tkt-EncryptPart.tkt-AuthnTime**) is set to KDS_X's system time.

- *Start time*

  If the requested start time (*asReq***.req-Body.req-StartTime**) is absent, or if it indicates a time earlier than $Tkt_{A,B}$'s authentication time (*asTkt***.tkt-EncryptPart.tkt-AuthnTime**), then $Tkt_{A,B}$'s start time (*asTkt***.tkt-EncryptPart.tkt-StartTime**) is omitted (indicating a default to the authentication time). Otherwise (that is, a requested start time is present and indicates a time later than or equal to the authentication time), if the postdate option (*asReq***.req-Body.req-Flags.req-Postdate**) is selected, then $Tkt_{A,B}$'s start time is set to the requested start time; if the postdate option is deselected, $Tkt_{A,B}$'s start time is omitted (indicating a default to the authentication time) or is set to KDS_X's system time (see also the postdated and invalid options, below) {**errStatusCode-POLICY**}.

- *Expiration time*

  KDS_X sets $Tkt_{A,B}$'s expiration time (*asTkt***.tkt-EncryptPart.tkt-ExpireTime**) to the earliest of the following:

  — The requested expiration time (*asReq***.req-Body.req-ExpireTime**).

  — $Tkt_{A,B}$'s start time (or authentication time, if the start time is absent) plus the maximum ticket lifetime associated with the named client *A*.

  — $Tkt_{A,B}$'s start time (or authentication time, if the start time is absent) plus the maximum ticket lifetime associated with the target server KDS_X.

  — $Tkt_{A,B}$'s start time (or authentication time, if the start time is absent) plus the cell-wide maximum ticket lifetime associated with the issuing authority KDS_X.

  KDS_X checks that the resulting lifetime of $Tkt_{A,B}$ is greater than or equal to the cell-wide minimum ticket lifetime associated with the issuing authority KDS_X {**errStatusCode-NEVER-VALID**}.

- *Maximum expiration time*

  If either the renewable option (*asReq***.req-Body.req-Flags.req-Renewable**) has been selected, or if the renewable-okay option (*asReq***.req-Body.req-Flags.req-RenewableOK**) has been selected and $Tkt_{A,B}$'s expiration time is earlier than the requested expiration time, then $Tkt_{A,B}$'s maximum expiration time (*asTkt***.tkt-EncryptPart.tkt-MaxExpireTime**) is present (otherwise it is omitted) and is set to the earliest of:

  — The requested maximum expiration time (*asReq***.req-Body.req-MaxExpireTime**), if present; otherwise, the requested expiration time (*asReq***.req-Body.req-ExpireTime**).

  — $Tkt_{A,B}$'s start time (or authentication time, if the start time is absent) plus the maximum renewable ticket lifetime associated with the named client *A*.

  — $Tkt_{A,B}$'s start time (or authentication time, if the start time is absent) plus the maximum renewable ticket lifetime associated with the target server KDS_X.

  — $Tkt_{A,B}$'s start time (or authentication time, if the start time is absent) plus the cell-wide maximum renewable ticket lifetime associated with the issuing authority KDS_X.

- *Transit path*

  $Tkt_{A,B}$'s transit path (*asTkt***.tkt-EncryptPart.tkt-TransitPath**) is omitted.

- *Client addresses*

  $Tkt_{A,B}$'s client address field (*asTkt***.tkt-EncryptPart.tkt-ClientAddrs**) is set to the requested client addresses (*asReq***.req-Body.req-ClientAddrs**), if present. (In particular, no attempt is made to check that this AS Request originated from one of the requested client addresses, if present.) Otherwise, it is omitted. In the present revision of DCE, at least one client address must be supplied by the client, otherwise the KDS will fail the AS Request.

- *Authorisation data*

  KDS_X checks that the requested authorisation data *asReq***.req-Body.req-EncryptedAuthzData**) is omitted. (Since the AS Request is unauthenticated, KDS_X cannot vouch for such authorisation data.) $Tkt_{A,B}$'s authorisation data field (*asTkt***.tkt-EncryptPart.tkt-AuthzData**) is omitted.

- *Additional tickets*

KDS_X checks that no additional tickets (*asReq***.req-Body.req-AdditionalTkts**) are present.

- *Options*

  — *Forwardable*

  If the forwardable option (*asReq***.req-Body.req-Flags.req-Forwardable**) has been requested and if forwardability is permitted by KDS_X, then $Tkt_{A,B}$'s forwardable option (*asTkt***.tkt-EncryptPart.tkt-Flags.tkt-Forwardable**) is selected.

  — *Proxiable*

  If the Proxiable option (*asReq***.req-Body.req-Flags.req-Proxiable**) has been requested and if proxiability is permitted by KDS_X, and if $B \neq KDS_X$, then $Tkt_{A,B}$'s proxiable option (*asTkt***.tkt-EncryptPart.tkt-Flags.tkt-Proxiable**) is selected. (If $B = KDS_X$, then this AS request is denied, because ticket-granting-tickets are not proxiable.)

  — *Postdatable*

  If the postdatable option (*asReq***.req-Body.req-Flags.req-Postdatable**) has been requested and if postdatability is permitted by KDS_X, then $Tkt_{A,B}$'s postdatable option (*asTkt***.tkt-EncryptPart.tkt-Flags.tkt-Postdatable**) is selected.

  — *Postdated*

  If $Tkt_{A,B}$'s start time is present, then $Tkt_{A,B}$'s postdated option (*asTkt***.tkt-EncryptPart.tkt-Flags.tkt-Postdated**) is selected.

  — *Invalid*

  If $Tkt_{A,B}$'s postdated option is selected (above), then $Tkt_{A,B}$'s invalid option (*asTkt***.tkt-EncryptPart.tkt-Flags.tkt-Invalid**) is selected.

  — *Renewable, renewable-okay*

  If $Tkt_{A,B}$'s maximum expiration time is present and if renewability is permitted by KDS_X, then $Tkt_{A,B}$'s renewable option (*asTkt***.tkt-EncryptPart.tkt-Flags.tkt-Renewable**) is selected.

  — *Other; use-session-key, validate, renew, proxy, forward*

  KDS_X checks that no other KDS options have been requested. Currently, these are the use-session-key (*asReq***.req-Body.req-Flags.req-UseSessionKey**), validate (*asReq***.req-Body.req-Flags.req-Validate**), renew (*asReq***.req-Body.req-Flags.req-Renew**), proxy (*asReq***.req-Body.req-Flags.req-Proxy**) and forward (*asReq***.req-Body.req-Flags.req-Forward**) options {**errStatusCode-BAD-OPTION**}.

  — *Initial*

  $Tkt_{A,B}$'s initial option (*asTkt***.tkt-EncryptPart.tkt-Flags.tkt-Initial**) is selected. (This marks $Tkt_{A,B}$ as an *initial* ticket.)

  — *Other; proxied, forwarded*

  All of $Tkt_{A,B}$'s options that have not been selected by any of the above steps are deselected. Additionally, the forwarded (*asTkt***.tkt-EncryptPart.tkt-Flags.tkt-Forwarded**) and proxied (*asTkt***.tkt-EncryptPart.tkt-Flags.tkt-Proxied**) options are deselected.

- *Encryption*

KDS_X encrypts *asTkt***.tkt-EncryptPart** using KDS_X's long-term key K_{KDSX} (of the chosen encryption type *encType*, and key version number). This is the ciphertext portion of $Tkt_{A,B}$ (*asTkt***.tkt-EncryptedPart.encData-CipherText**). KDS_X also sets the encryption type

(*asTkt*.**tkt-EncryptedPart.encData-EncType**) to *encType* and the key version number (*asTkt*.**tkt-EncryptedPart.encData-KeyVersNum**) to K_{KDS_X}'s version number.

At this point, $Tkt_{A,B}$ is well-formed, and KDS_X turns its attention to completing the construction of the AS Response message, *asResp*.

- *Protocol version number*

  The protocol version number (*asResp*.**resp-ProtoVersNum**) is set to **protoVersNum-KRB5**.

- *Protocol message type*

  The protocol message type (*asResp*.**resp-ProtoMsgType**) is set to **protoMsgType-AS-RESPONSE**.

- *Authentication data*

  The authentication data (*asResp*.**resp-AuthnData**) is either omitted, or it consists of a single element (*asResp*.**resp-AuthnData[0]**); in the latter case, the type (*asResp*.**resp-AuthnData[0].authnData-Type**) is **authnDataType-PW-SALT** and the value (*asResp*.**resp-AuthnData[0].authnData-Value**) is the salt to be used by the client to derive its long-term key (via the algorithm of Section 4.3.6.1 on page 190).

- *Client cell*

  The client cell (*asResp*.**resp-ClientCell**) is set to $Tkt_{A,B}$'s client cell (*asTkt*.**tkt-EncryptPart.tkt-ClientCell**).

- *Client name*

  The client name (*asResp*.**resp-ClientName**) is set to $Tkt_{A,B}$'s client name (*asTkt*.**tkt-EncryptPart.tkt-ClientName**).

- *Ticket*

  The ticket (*asResp*.**resp-Tkt**) is set to the newly created $Tkt_{A,B}$ (*asTkt*).

- *Session key*

  The session key (*asResp*.**resp-EncryptPart.resp-SessionKey**) is set to $Tkt_{A,B}$'s session key, K_{A,KDS_X} (*asTkt*.**tkt-EncryptPart.tkt-SessionKey**).

- *Nonce*

  The nonce (*asResp*.**resp-EncryptPart.resp-Nonce**) is set to the nonce that the calling client sent (*asReq*.**req-Body.req-Nonce**).

- *Client addresses*

  The client address field (*asResp*.**resp-EncryptPart.resp-ClientAddrs**) is set to $Tkt_{A,B}$'s client address field (*asTkt*.**tkt-EncryptPart.tkt-ClientAddrs**), if present. Otherwise, it is omitted.

- *Server cell*

  The server cell (*asResp*.**resp-EncryptPart.resp-ServerCell**) is set to $Tkt_{A,B}$'s server cell (*asTkt*.**tkt-ServerCell**).

- *Server name*

  The server name (*asResp*.**resp-EncryptPart.resp-ServerName**) is set to $Tkt_{A,B}$'s server name (*asTkt*.**tkt-ServerName**).

- *Authentication time*

The authentication time (*asResp***.resp-EncryptPart.resp-AuthnTime**) is set to $Tkt_{A,B}$'s authentication time (*asTkt***.tkt-EncryptPart.tkt-AuthnTime**).

- *Start time*

  The start time (*asResp***.resp-EncryptPart.resp-StartTime**) is set to $Tkt_{A,B}$'s start time (*asTkt***.tkt-EncryptPart.tkt-StartTime**), if present. Otherwise, it is omitted.

- *Expiration time*

  The expiration time (*asResp***.resp-EncryptPart.resp-ExpireTime**) is set to $Tkt_{A,B}$'s expiration time (*asTkt***.tkt-EncryptPart.tkt-ExpireTime**).

- *Maximum expiration time*

  The maximum expiration time (*asResp***.resp-EncryptPart.resp-MaxExpireTime**) is set to $Tkt_{A,B}$'s maximum expiration time (*asTkt***.tkt-EncryptPart.tkt-MaxExpireTime**), if present. Otherwise, it is omitted.

- *Key expiration date*

  The key expiration date (*asResp***.resp-EncryptPart.resp-KeyExpireDate**) is set to the expiration date of *A*'s long-term key (of the selected encryption type and key version number), K_A.

- *Last requests*

  The last requests field (*asResp***.resp-EncryptPart.resp-LastRequests**) is set to *A*'s last requests information.

- *Options*

  The options (*asResp***.resp-EncryptPart.resp-Flags**) are set to $Tkt_{A,B}$'s options (*asTkt***.tkt-EncryptPart.tkt-Flags**).

- *Encryption*

  KDS_X encrypts *asResp***.resp-EncryptPart** using *A*'s long-term key K_A (using the chosen encryption type *encType*). This is the ciphertext portion of the AS Response (*asResp***.resp-EncryptedPart.encData-CipherText**). KDS_X also sets the encryption type (*asResp***.resp-EncryptedPart.encData-EncType**) to *encType* and the key version number (*asResp***.resp-EncryptedPart.encData-KeyVersNum**) to the version number of the client's long-term key.

At this point, the KDS Response is well-formed, and KDS_X returns it to the calling client.

4.12.3 Client Receives AS Response

[RFC 1510: 3.1.5, A.3, A.4]

Consider a client *A* that receives an AS Response, *asResp* (that is, *asResp* is a value of type **ASResponse**, with protocol version number (*asResp***.resp-ProtoVersNum**) **protoVersNum-KRB5** and protocol message type (*asResp***.resp-ProtoMsgType**) **protoMsgType-AS-RESPONSE**), in response to an AS Request, *asReq* (as the result of calling *kds_request*( )) to KDS_X. Then *A* processes *asResp* according to the following algorithm. In the case this algorithm completes successfully, *A* is justified in believing that the returned $Tkt_{A,B}$ (or *asTkt*; that is, *asResp***.resp-Tkt**) is correctly and securely targeted to KDS_X, and that it contains the values returned elsewhere in *asResp* (in particular, that *A* is the client named by $Tkt_{A,B}$), and using it (especially, its session key, $K_{A,KDSX}$) in subsequent TGS Requests and Authentication Headers it sends to KDS_X. In the case the algorithm fails, *A* takes (application-specific) recovery action.

Here, the notions of ''success'' or ''failure'' of this algorithm are taken to mean ''conforming to *A*'s request (*asReq*)'', where the criteria of ''conformance'' are application-specific. Typically, but not necessarily, *A* will be satisfied only if KDS_X formulates $Tkt_{A,B}$ exactly as *A* requested. For example, *A* may have requested a very long maximum expiration time but KDS_X issued only a somewhat shorter one — whether *A* views that as a success or failure is an application-specific determination. (Note that *A* cannot inspect $Tkt_{A,B}$ directly, because *A* cannot decrypt it — *A* has to rely on the other, unencrypted, fields of the AS Response message.)

- *Client cell*

  The named client's cell (*asResp***.resp-ClientCell**) is checked for conformance to *A*'s cell name (which *A* had implicitly sent (*asReq***.req-Body.req-ServerCell**), by sending its AS Request to KDS_X).

- *Client name*

  The client name (*asResp***.resp-ClientName**) is checked for conformance to what *A* requested (*asReq***.req-Body.req-ClientName**); that is, to *A*'s RS name.

- *Authentication data*

  The authentication data (*asResp***.resp-AuthnData**), if present, is scanned for its least element (that is, the minimal *i*) for which the type (*asResp***.resp-AuthnData[***i***].authnData-Type**) is **authnDataType-PW-SALT**, and then the client derives its long-term key, K_A, from its password and the included salt (*asResp***.resp-AuthnData[***i***].authnData-Value**) (see Section 4.3.6.1 on page 190). If the authentication data is absent, then the client derives its long-term key from its password and the default salt (see Section 4.3.6.1 on page 190).

- *Ticket*

  $Tkt_{A,B}$ (*asTkt*, *asResp***.resp-Tkt**) is not directly interpretable (in the sense of being decryptable) by *A*, but the information in it is largely available elsewhere in *asResp*.

- *Decryption*

  The encryption type, *encType* (*asResp***.resp-EncryptedPart.encData-EncType**), is checked for conformance to what *A* had requested (*asReq***.req-Body.req-EncTypes**). If it is acceptable, then *A* attempts to decrypt the ciphertext portion of the AS Response (*asResp***.resp-EncryptedPart.encData-CipherText**), using its long-term key K_A (which *A* must know or derive; for example, from its password and salt as described above), of encryption type *encType* and the indicated key version number (*asResp***.resp-EncryptedPart.encData-KeyVersNum**). A successful decryption is recognised by the built-in integrity afforded by the ciphertext itself. In this way, *A* learns the information carried in *asResp***.resp-EncryptPart**. If *A* encounters an unsuccessful decryption, it takes application-specific action — this presumably includes rejection of *asResp* as untrustworthy (the ability to successfully decrypt *asResp***.resp-EncryptedPart** proves to *A* that it was encrypted by the legitimate KDS_X (since *A* trusts its long-term key K_A to be secure), and that it is not being spoofed by a counterfeit KDS_X). In particular, if *A* is not the client requested in *asResp* (and named in *asTkt*), in the sense of not knowing the correct long-term key of that client, then *A* will not be able to successfully decrypt *asResp*, and consequently will not be able to gain access to the information in *asResp***.resp-EncryptPart** (in particular, its session key, $K_{A,KDSX}$ (*asResp***.resp-EncryptPart.resp-SessionKey**)).

  Note: This assumes (as stated) that *A* trusts its long-term key K_A. In the terminology of the Login Facility (see Section 1.15 on page 71), the successful decryption of *asResp***.resp-EncryptedPart.encData-CipherText** amounts to ''validation of $Tkt_{A,B}$''. In order for arbitrary other parties (other than *A*) to become convinced of the genuineness of $Tkt_{A,B}$, the subtler protocols involved in ''certification of

$Tkt_{A,B}$'' (see Section 1.15.2 on page 77) must be employed.

- *Nonce*

  The nonce (*asResp***.resp-EncryptPart.resp-Nonce**) is checked for equality with the requested nonce (*asReq***.req-Body.req-Nonce**). If it is not equal, then this *asResp* does not correspond to *asReq*, and a ''replay attack'' may be suspected, and *A* takes application-specific action.

- *Last requests*

  The last requests (*asResp***.resp-EncryptPart.resp-LastRequests**) are inspected. If they do not match *A*'s own knowledge of its previous requests, then a potential breach of security may be suspected, and *A* typically invokes recovery measures consistent with local policy.

- *Key expiration date*

  The key expiration date (*asResp***.resp-EncryptPart.resp-KeyExpireDate**) is inspected if present. If it indicates a date in the ''near'' future, then *A* should invoke key update procedures according to local policy (see Chapter 11). Typically, this will involve *A*'s ''changing its password'' — a comparatively ''cheap'' undertaking. Failure to do so risks *A*'s long-term key K_A, and hence its password, actually expiring, which would require *A* to re-register itself with RS_X (for encryption type *encType*) — a comparatively ''expensive'' undertaking in itself, in addition to the lost opportunities for service access while the long-term key is expired.

- *Server cell*

  The server cell (*asResp***.resp-EncryptPart.resp-ServerCell**) is checked for conformance to what *A* requested (*asReq***.req-Body.req-ServerCell**); that is, to KDS_X's cell name, or to *X*'s cell name.

- *Server name*

  The server name (*asResp***.resp-EncryptPart.resp-ServerName**) is checked for conformance to what *A* requested (*asReq***.req-Body.req-ServerName**); that is, to KDS_X's RS name.

- *Session key*

  The session key (which *A* trusts is secure), $K_{A,KDSX}$ (*asResp***.resp-EncryptPart.resp-SessionKey**), is saved for later use in protecting communications with KDS_X; that is, subsequent TGS Requests.

- *Authentication time*

  The authentication time (*asResp***.resp-EncryptPart.resp-AuthnTime**) is checked for conformance to what *A* expects (typically, it should be equal to *A*'s system time (modulo **maxClockSkew**)).

- *Start time*

  The start time (*asResp***.resp-EncryptPart.resp-StartTime**) is checked for conformance to what *A* requested. Namely, if *A* requested $Tkt_{A,B}$ to be postdated, then the start time is checked to be present and checked for conformance to the start time *A* requested (*asReq***.req-Body.req-StartTime**); otherwise, the start time should be absent.

- *Expiration time*

  The expiration time (*asResp***.resp-EncryptPart.resp-ExpireTime**) is checked for conformance to what *A* requested (*asReq***.req-Body.req-ExpireTime**).

- *Maximum expiration time*

The maximum expiration time (*asResp***.resp-EncryptPart.resp-MaxExpireTime**) is checked for conformance to what *A* requested. It should be only present (at most) if *A* had selected the renewable option (*asReq***.req-Body.req-Flags.req-Renewable**) and supplied a requested maximum expiration time (*asReq***.req-Body.req-MaxExpireTime**), or if *A* had selected the renewable-okay option (*asReq***.req-Body.req-Flags.req-RenewableOK**).

- *Client addresses*

  If *A* requested that $Tkt_{A,B}$ contain client host addresses, then the client address field (*asResp***.resp-EncryptPart.resp-ClientAddrs**) is checked to be present and checked for conformance to what *A* requested (*asReq***.req-Body.req-ClientAddrs**). Otherwise, the client addresses should be absent.

- *Options*

  The options field (*asResp***.resp-EncryptPart.resp-Flags**) is inspected for conformance to what *A* requested (*asReq***.req-Body.req-Flags**).

This completes the specification of the AS Request / Response exchange.

4.13 (Reverse-)Authentication Header Processing

[RFC 1510: 1, 3.2.1]

This section specifies in detail the processing that occurs during an authentication/reverse-authentication header exchange. There are three steps involved:

1. A client prepares an authentication header and sends it to a target server as part of an ''authenticated request for (RPC) service'' (for example, this could be a TGS Request, in which case the target server is a KDS server). Typically, this authentication header will be merely a part of the whole message sent from client to server, and the rest of the message will contain RPC protocol information and the input parameters for the RPC service request.
2. A server receives an authentication header from a client, processes it, prepares a reverse-authentication header (in the case of a successful client-to-server authentication, and the client has requested the mutual authentication option) or an error message (in the case of a failed client-to-server authentication), and returns that to the client (though some servers may not return errors depending on their policy). Typically, in the success case, the server will also proceed to perform the requested service (subject to authorisation constraints) and return the output RPC parameters to the client in addition to the reverse-authentication header.
3. A client receives a reverse-authentication header (success case, if it had requested mutual authentication in its authentication header) or an error (failure case). Typically, in the success case, it also receives the results of its RPC service request, which it will then decide to accept or reject (on the basis of the reverse-authentication header).

The details of the three steps of the success case are specified next.

Note that it is *A*'s responsibility to know (or to securely determine) all the information necessary to correctly formulate its message to *B* — especially, *B*'s cell name and RS name.

Finally, note that not every RPC request/response needs to carry an authentication/reverse-authentication header; only those that need to establish a session or conversation (either initial or re-established) key need do so. Once such a key has been established (and trusted by both client and server), it can be used to protect numerous subsequent RPCs — see Chapter 9 for details.

Notes:

1. It should be noted that the descriptions here are ''typical'' of authentication/reverse-authentication header processing. But since the interpreters (*A* and *B*) of the authentication/reverse-authentication headers are in general application-specific, this whole discussion should be understood to implicitly accommodate some such wording as ''··· *or other such processing as the application requires or allows* ···''. For example, an especially cautious server may refuse to accept proxied tickets, or an especially lenient one may provide a ''grace period'' during which it will accept tickets that expire during a session. Another example is that a password-changing program might demand an initial ticket, to guard against the possibility of a miscreant's hijacking a user session by simply sitting down at an unattended seat. See also Section 4.14 on page 240, which is the only place authentication/reverse-authentication headers are used internally in the KDS protocol ''application''.
2. The presentation of Section 4.13.1 on page 232 through Section 4.13.3 on page 238 is cast in terms of: ''client *A* using $\mathrm{Tkt}_{A,B}$ (with session key $\mathrm{K}_{A,B}$) to authenticate to server *B*''. It will be observed, though, that a third principal *A´*

could be injected into the discussion, and the whole presentation recast as: ''client *A* using $Tkt_{A',B}$ (with session key $K_{A',B}$) to authenticate to server *B*, provided that *A* knows the session key $K_{A',B}$''. This notation becomes burdensome to the exposition, so it won't be employed here. Far from being a minor consequence of the authentication architecture, systematic use of this idea is central to the privilege architecture (with *A´* = PS, the Privilege Server — see Chapter 1 and Chapter 5).

4.13.1 Client Sends Authentication Header

[RFC 1510: 3.2.2, A.9]

Consider a client *A* in cell *X* which has successfully executed a KDS Request (*kds_request*( )), that is, whose KDS Response it accepts (in the sense of Section 4.12.3 on page 227 and/or Section 4.14.3 on page 254). Thus, *A* is in possession of a $Tkt_{A,B}$ received from KDS_Y, *kdsTkt*, whose contents it knows, especially its session key $K_{A,B}$ (and its encryption type *encType*), and now *A* wants to use $Tkt_{A,B}$ to ''authenticate to'' (that is, engage in protected communications with) server *B* in cell *Y*. The following algorithm specifies how *A* prepares an authentication header, *authnHdr* (a value of the **AuthnHeader** data type) containing $Tkt_{A,B}$ (*authnHdr***.authnHdr-Tkt**), and a newly generated authenticator *authnr* (*authnHdr***.authnHdr-EncryptAuthnr**, of type **Authenticator**), and sends it to *B* for this purpose.

The following algorithm first discusses how *A* constructs the newly generated authenticator, *authnr*, and then how it constructs the rest of the authentication header, *authnHdr*.

- *Protocol version number*

  The protocol version number (*authnr***.authnr-ProtoVersNum**) is set to **protoVersNum-KRB5**.

- *Client cell*

  The client cell (*authnr***.authnr-ClientCell**) is set to *A*'s cell name; that is, to *X*'s cell name.

- *Client name*

  The client name (*authnr***.authnr-ClientName**) is set to *A*'s RS name.

- *Client timestamp*

  The client timestamp (*authnr***.authnr-ClientTime**) is set to *A*'s system time.

- *Client microsecondstamp*

  The client microsecond stamp (*authnr***.authnr-ClientMicroSec**) is set to *A*'s system microsecond time.

- *Conversation key*

  If *A* desires to use a conversation key of its own choosing, say $\hat{K}_{A,B}$ (of the same encryption type, *encType*), instead of the KDS_Y-generated session key $K_{A,B}$, to protect this client-server session, then it sets the conversation key (*authnr***.authnr-ConversationKey**) to $\hat{K}_{A,B}$. Otherwise this field is omitted. (The keys $K_{A,B}$, $\hat{K}_{A,B}$ and $\hat{\hat{K}}_{A,B}$ all have the same encryption type, *encType*.)

- *Checksum*

  If this application uses checksums, then *A* uses an application-specific checksum type (*authnr***.authnr-Cksum.cksum-Type**) to set the checksum value (*authnr***.authnr-Cksum.cksum-Value**) to the checksum of some application-specific plaintext (typically, this will be the checksum of the ''service'' portion of the message that this authenticator is

authenticating). Otherwise the checksum is omitted. (In the case of DCE RPC applications, the use of checksums is specified as part of the RPC protocol specifications — see Chapter 9.)

- *Sequence number*

  The sequence number (*authnr***.authnr-SeqNum**) is processed in an application-specific manner (perhaps omitting it).

- *Authorisation data*

  If this application uses additional authorisation data, then *A* sets the authorisation data field (*authnr***.authnr-AuthzData**) to application-specific additional authorisation data. Otherwise, this field is omitted.

- *Encryption*

  A encrypts *authnr*, using the encryption type *encType* and the session key $K_{A,B}$ (not the conversation key $\hat{K}_{A,B}$, even if present) in the accompanying $Tkt_{A,B}$. This is the ciphertext portion of the authentication header (*authnHdr***.authnHdr-EncryptedAuthnr.encData-CipherText**). *A* also sets the encryption type (*authnHdr***.authnHdr-EncryptedAuthnr.encData-EncType**) to *encType* and the key version number (*authnHdr***.authnHdr-EncryptedAuthnr.encData-KeyVersNum**) to an appropriate application-specific value, if any (usually it is omitted).

At this point, *authnr* is well-formed and encrypted, and *A* turns its attention to completing the construction of the authentication header, *authnHdr*.

- *Protocol version number*

  The protocol version number (*authnHdr***.authnHdr-ProtoVersNum**) is set to **protoVersNum-KRB5**.

- *Protocol message type*

  The protocol message type (*authnHdr***.authnHdr-ProtoMsgType**) is set to **protoMsgType-AUTHN-HEADER**.

- *Ticket*

  The ticket (*authnHdr***.authnHdr-Tkt**) is set to $Tkt_{A,B}$ (*kdsTkt*).

- *Authenticator*

  The encrypted authenticator (*authnHdr***.authnHdr-EncryptedAuthnr**) is set to the encryption of *authnr* as constructed above.

- *Options*

  — *Use-session-key*

  If the accompanying $Tkt_{A,B}$ (*kdsTkt*, *authnHdr***.authnHdr-Tkt**), is protected with a session key, $K^{\bullet}$ (carried in a (ticket-granting-)ticket targeted to *B*), instead of with *B*'s long-term key K_B, then the use-session-key option (*authnHdr***.authnHdr-Flags.authnHdr-UseSessionKey**) is selected (see Section 4.6.2 on page 203).

  — *Mutual authentication*

  If *A* desires that *B* return a reverse-authentication header with its response, the mutual authentication option (*authnHdr***.authnHdr-Flags.authnHdr-MutualRequired**) is selected.

  — *Other*

All of *authnHdr*'s options that have not been selected by any of the above steps are deselected (unless they are used in application-specific ways). Currently, there are no other options that haven't already been mentioned above.

At this point, *authnHdr* is well-formed, and *A* sends it to *B*.

4.13.2 Server Receives Authentication Header and Sends Reverse-authentication Header

[RFC 1510: 3.2.3, 3.2.4, A.10, A.11]

Consider an authentication header, *authnHdr*, received by a server, *B* in cell *Y*, containing a $Tkt_{A,B}$ (*ahTkt*, *authnHdr***.authnHdr-Tkt**) and an authenticator, *authnr* (*authnHdr***.authnHdr-EncryptAuthnr**). (For example, KDS servers receive such an authentication header in the first authentication data field of a TGS Request (*tgsReq***.req-AuthnData[0].authnData-Value**, with *tgsReq***.req-AuthnData[0].authnData-Type** = **authnDataType-TGS-REQ**) — see Section 4.14.2 on page 245.) Thus, *authnHdr* is a value of type **AuthnHeader**, with protocol version number (*authnHdr***.authnHdr-ProtoVersNum**) **protoVersNum-KRB5** and protocol message type (*authnHdr***.authnHdr-ProtoMsgType**) **protoMsgType-AUTHN-HEADER**. Then *B* executes the algorithm below. If the algorithm executes successfully, *B* is justified in believing that it can at this time engage in secure communications with the client *A* named in $Tkt_{A,B}$, protecting their communications with the session key $K_{A,B}$ carried in $Tkt_{A,B}$, or with the conversation key $\hat{K}_{A,B}$ carried in the authentication header itself (*authnHdr***.authnHdr-EncryptAuthnr.authnr-ConversationKey**) (or with another conversation key, $\hat{\hat{K}}_{A,B}$, of *B*'s own choosing — see below). That is, the authentication header ''authenticates the client *A* to the server *B*''.

The following algorithm first discusses how *B* processes *authnHdr*, and then, if the mutual authentication option (*authnHdr***.authnHdr-Flags.authnHdr-MutualRequired**) has been selected, how *B* constructs a reverse-authentication header, *revAuthnHdr* (of type **RevAuthnHeader**), to return to *A*.

- *Authentication header options*:

  — *Use-session-key*

  If the use-session-key option (*authnHdr***.authnHdr-Flags.authnHdr-UseSessionKey**) is deselected, *B* knows that *ahTkt* is protected with its long-term key K_B. If it is selected, *B* knows that *ahTkt* is protected with a session key, $K^{\bullet}$ (see Section 4.6.2 on page 203).

  — *Mutual authentication*

  If the mutual authentication option (*authnHdr***.authnHdr-Flags.authnHdr-MutualRequired**) is selected, *B* knows *A* expects a reverse-authentication header to be returned.

- *Ticket protocol version number*

  The protocol version number (*ahTkt***.tkt-ProtoVersNum**) is checked to be **protoVersNum-KRB5**.

- *Ticket server cell*

  The server cell (*ahTkt***.tkt-ServerCell**) is checked to be the name of *B*'s cell; that is, *Y*'s cell name.

- *Ticket server name*

  The server name (*ahTkt***.tkt-ServerName**) is checked to be *B*'s RS name.

- *Ticket decryption*

The encryption type protecting $Tkt_{A,B}$, *encType* (*ahTkt***.tkt-EncryptedPart.encData-EncType**), is checked for support by *B*, as is the key version number (*ahTkt***.tkt-EncryptedPart.encData-KeyVersNum**) if present. If the use-session-key option is deselected, *B* uses its long-term key K_B to attempt to decrypt $Tkt_{A,B}$'s ciphertext (*ahTkt***.tkt-EncryptedPart.encData-CipherText**); otherwise, *B* uses the session key, $K^{\bullet}$. A successful decryption is recognised by the built-in integrity afforded by the ciphertext itself. In this way, *B* learns the information carried in $Tkt_{A,B}$, in particular its session key $K_{A,B}$ (*ahTkt***.tkt-EncryptPart.tkt-SessionKey**).

- *Authenticator decryption*

  The encryption type protecting the authenticator (*authnHdr***.authnHdr-EncryptedAuthnr.encData-EncType**) is checked to be the same as that protecting $Tkt_{A,B}$, namely *encType*. *B* decrypts the authenticator's ciphertext (*authnHdr***.authnHdr-EncryptedAuthnr.encData-CipherText**) using the session key $K_{A,B}$ (not $K^{\bullet}$, even if the use-session-key option is selected). The key version number (*authnHdr***.authnHdr-EncryptedAuthnr.encData-KeyVersNum**), if present, is processed in an application-specific way (usually, it is omitted). A successful decryption is recognised by the built-in integrity afforded by the ciphertext itself — *this is what convinces B that A knows the session key* $K_{A,B}$; that is, this is what actually ''authenticates'' *A* to *B*. In this way, *B* learns the information carried in *authnr* (*authnHdr***.authnHdr-EncryptAuthnr**).

- *Authenticator protocol version number*

  The authenticator's protocol version number (*authnr***.authnr-ProtoVersNum**) is checked to be **protoVersNum-KRB5**.

- *Client cell*

  The client cells from *ahTkt* (*ahTkt***.tkt-EncryptPart.tkt-ClientCell**) and from *authnr* (*authnr***.authnr-ClientCell**) are checked to be the same (namely, to *A*'s cell name; that is, *X*'s cell name).

- *Client name*

  The client names from *ahTkt* (*ahTkt***.tkt-EncryptPart.tkt-ClientName**) and from *authnr* (*authnr***.authnr-ClientName**) are checked to be the same (namely, to *A*'s RS name).

- *Client addresses*

  If there are client addresses present in *ahTkt* (*ahTkt***.tkt-EncryptPart.tkt-ClientAddrs**) and if *B* requires that client addresses be used, then *B* checks that *A* is communicating from one of them (as reported by *B*'s operating system, according to the level of trust *B* places in that); that is, that *authnHdr* was received from one of the addresses on the list.

- *Client timestamp*

  A's timestamp (*authnr***.authnr-ClientTime**) is checked to be equal to *B*'s system time (modulo **maxClockSkew**). It is in this way that *B* becomes convinced that it is communicating with *A* in real-time.

- *Client microsecondstamp*

  A's microsecondstamp (*authnr***.authnr-ClientMicroSec**), together with its timestamp, are checked to not be present in *B*'s replay cache. They are then stored in the replay cache.

- *Authentication time*

  The authentication time (*ahTkt***.tkt-EncryptPart.tkt-AuthnTime**) is typically ignored by application-level servers (see Section 4.14 on page 240 for the case of KDS servers).

- *Start time*

  The start time (*ahTkt***.tkt-EncryptPart.tkt-StartTime**) is checked to be earlier than or equal to *B*'s system time (modulo **maxClockSkew**).

- *Expiration time*

  The expiration time (*ahTkt***.tkt-EncryptPart.tkt-ExpireTime**) is checked to be later than or equal to *B*'s system time (modulo **maxClockSkew**).

- *Maximum expiration time*'

  The maximum expiration time (*ahTkt***.tkt-EncryptPart.tkt-MaxExpireTime**) is typically ignored by application-level servers (see Section 4.14 on page 240 for the case of KDS servers).

- *Sequence number*

  The sequence number (*authnr***.authnr-SeqNum**) is processed in an application-specific manner.

- *Conversation key*

  If a conversation key $K^{\wedge}_{A,B}$ (*authnr***.authnr-ConversationKey**) is present, *B* decides (in an application-specific way) whether it will use it to protect this client-server session, or if it will use *ahTkt*'s session key $K_{A,B}$ or will generate its own conversation key $K^{\wedge\wedge}_{A,B}$ for this purpose (informing *A* of it in the accompanying *revAuthnHdr*).

- *Transit path*

  The transit path (*ahTkt***.tkt-EncryptPart.tkt-TransitPath**) is typically ignored by application-level servers (see Section 4.14 on page 240 for the case of KDS servers, and Section 5.4 on page 292 for the case of PS servers).

- *Checksum*

  If a checksum (*authnr***.authnr-Cksum**) is present, it is processed in an application-specific way.

- *Ticket options*:

  — *Invalid*

  The invalid option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Invalid**) is typically checked by application-level servers to be deselected (see Section 4.14 on page 240 for the case of KDS servers).

  — *Forwardable, forwarded, proxiable, proxied, postdatable, postdated, renewable, initial*

  All other options are ignored by application-level servers (see Section 4.14 on page 240 for the case of KDS servers). Currently, these include forwardable (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Forwardable**), forwarded (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Forwarded**), proxiable (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Proxiable**), proxied (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Proxied**), postdatable (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Postdatable**), postdated (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Postdated**), renewable (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Renewable**), and initial (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Initial**).

- *Ticket authorisation data*

  If ticket authorisation data (*ahTkt***.tkt-EncryptPart.tkt-AuthzData**) is present, *B* uses it (in an application-specific way) to make authorisation decisions.

- *Authenticator authorisation data*

  If additional authenticator authorisation data (*authnr***.authnr-AuthzData**) is present, *B* uses it (in an application-specific way) to make authorisation decisions.

At this point, *B* has completed its processing of *authnHdr*. If *A* has not requested mutual authentication (by the *authnHdr***.authnHdr-Flags.authnHdr-MutualRequired** option), *B* proceeds with the application-specific performance of its service (which might include checking authorisation controls, and sending return parameters (potentially protected with a session or conversation key) back to *A* without a reverse-authentication header, and so on). Otherwise, *B* turns its attention to constructing a reverse-authentication header, *revAuthnHdr*, to be sent back to *A*, as follows:

- *Protocol version number*

  The protocol version number (*revAuthnHdr***.revAuthnHdr-ProtoVersNum**) is set to **protoVersNum-KRB5**.

- *Protocol message type*

  The protocol message type (*revAuthnHdr***.revAuthnHdr-ProtoMsgType**) is set to **protoMsgType-REVAUTHN-HEADER**.

- *Client timestamp*

  The client timestamp (*revAuthnHdr***.revAuthnHdr-EncryptPart.revAuthnHdr-ClientTime**) is set to the timestamp *A* sent in its authentication header (*authnr***.authnr-ClientTime**).

- *Client microsecondstamp*

  The client microsecondstamp (*revAuthnHdr***.revAuthnHdr-EncryptPart.revAuthnHdr-ClientMicroSec**) is set to the microsecondstamp *A* sent in its authentication header (*authnr***.authnr-ClientMicroSec**).

- *Conversation key*

  If *B* wants to protect the current client-server session with a key of its own choosing, instead of either the KDS_Y-generated session key ($K_{A,B}$) or the *A*-generated conversation key ($\hat{K}_{A,B}$) if present, then *B* generates a conversation key $\hat{\hat{K}}_{A,B}$ (of the same encryption type, *encType*) and returns it in the conversation key field (*revAuthnHdr***.revAuthnHdr-EncryptPart.revAuthnHdr-ConversationKey**). Otherwise, it is omitted.

- *Sequence number*

  The sequence number (*revAuthnHdr***.revAuthnHdr-EncryptPart.revAuthnHdr-SeqNum**) is processed in an application-specific manner (perhaps omitting it).

- *Encryption*

  B encrypts *revAuthnHdr***.revAuthnHdr-EncryptPart**, using the encryption type *encType* and session key $K_{A,B}$ (not $\hat{K}_{A,B}$ or $\hat{\hat{K}}_{A,B}$, even if they exist) specified by the authentication header *authnHdr*. This is the ciphertext portion of the reverse-authentication header (*revAuthnHdr***.revAuthnHdr-EncryptedPart.encData-CipherText**). *B* also sets the encryption type (*revAuthnHdr***.revAuthnHdr-EncryptedPart.encData-EncType**) to *encType* and the key version number (*revAuthnHdr***.revAuthnHdr-EncryptedPart.encData-KeyVersNum**) to an appropriate application-specific value.

At this point, the *revAuthnHdr* is well-formed, and *B* returns it to the calling client.

4.13.3 Client Receives Reverse-authentication Header

[RFC 1510: 3.2.5, A.12]

Consider a reverse-authentication header, *revAuthnHdr*, received by a client *A*, in response to an authentication header, *authnHdr* (with the mutual authentication option selected), that *A* had earlier sent to a server *B*. (This is a purely application-level scenario, not a system-level scenario, as the KDS rejects TGS Requests that request mutual authentication — see Section 4.14.2 on page 245.) Thus, *revAuthnHdr* is a value of type **RevAuthnHeader**, with protocol version number (*revAuthnHdr***.revAuthnHdr-ProtoVersNum**) **protoVersNum-KRB5** and protocol message number (*revAuthnHdr***.revAuthnHdr-ProtoMsgType**) **protoMsgType-REVAUTHN-HEADER**. Then *A* executes the algorithm below. If the algorithm executes successfully, *A* is justified in believing that it can at this time participate in secure communications with the server *B* targeted by the $Tkt_{A,B}$ in the corresponding authentication header (*authnHdr***.authnHdr-Tkt.tkt-ServerCell** and *authnHdr***.authnHdr-Tkt.tkt-ServerName**), protecting their communications with the session key $K_{A,B}$ (*authnHdr***.authnHdr-Tkt.tkt-EncryptPart.tkt-SessionKey**), or with a negotiated conversation key $K\hat{}_{A,B}$ (*authnHdr***authnHdr-EncryptPart.authnr-ConversationKey**) or $K\hat{}\hat{}_{A,B}$ (*revAuthnHdr***.revAuthnHdr-EncryptPart.revAuthnHdr-ConversationKey**) if these exist (if multiples of these exist, the choice of which to use is dependent on application-specific negotiation-resolution policy). That is, the reverse-authentication header ''authenticates the server *B* to the client *A*''.

- *Decryption*

  The encryption type of the reverse-authentication header, *encType* (*revAuthnHdr***.revAuthnHdr-EncryptedPart.encData-EncType**), is checked for conformance for what *A* had requested (*authnHdr***.authnHdr-Tkt.tkt-EncryptedPart.encData-EncType**). If it is acceptable, *A* then attempts to decrypt the ciphertext portion of the reverse-authentication header (*revAuthnHdr***.revAuthnHdr-EncryptedPart.encData-CipherText**), using the session key $K_{A,B}$ (*authnHdr***.authnHdr-Tkt.tkt-EncryptPart.tkt-SessionKey**) of encryption type *encType* and key version number *authnHdr***.authnHdr-EncryptedPart.encData-KeyVersNum** if present (not $K\hat{}_{A,B}$ or $K\hat{}\hat{}_{A,B}$, even if these exist). A successful decryption is recognised by the built-in integrity afforded by the ciphertext itself. If *A* encounters an ''unsuccessful'' decryption, it takes application-specific action — this presumably includes rejection of *revAuthnHdr* as untrustworthy.

- *Client timestamp*

  The client timestamp (*revAuthnHdr***.revAuthnHdr-EncryptPart.revAuthnHdr-ClientTime**) is checked for conformance with the client timestamp that *A* had sent *B* (*authnHdr***.authnHdr-EncryptPart.authnr-ClientTime**).

- *Client microsecondstamp*

  The client microsecondstamp (*revAuthnHdr***.revAuthnHdr-EncryptPart.revAuthnHdr-ClientMicroSec**) is checked for conformance with the client microsecondestamp that *A* had sent to *B* (*authnHdr***.authnHdr-EncryptPart.authnr-ClientMicroSec**). It is this step and the previous one that convince *A* it is securely communicating with *B* (''now'').

- *Conversation key*

  If a conversation key $K\hat{}\hat{}_{A,B}$ (*revAuthnHdr***.revAuthnHdr-EncryptPart.revAuthnHdr-ConversationKey**) is present, *A* processes it in an application-specific way (typically, it is used to protect this client-server session).

- *Sequence number*

  The sequence number (*revAuthnHdr*.**revAuthnHdr-EncryptPart.revAuthnHdr-SeqNum**) is processed in an application-specific manner.

This completes the specification of the Authentication/Reverse-authentication Header exchange.

4.14 TGS Request/Response Processing

[RFC 1510: 1, 3.3]

This section specifies in detail the processing that occurs during a TGS Request/Response exchange. That is, this section specifies the manipulating of old tickets, as well as the issuing of new tickets (both service-tickets and ticket-granting-tickets which are used in cross-cell authentication). There are three steps involved:

1. A client prepares a TGS Request and sends it to a KDS server.
2. A KDS server receives the TGS Request from a client, processes it, prepares an TGS Response (success case) or KDS Error (failure case), and returns that to the client.
3. A client receives a TGS Response or KDS Error.

The details of the three steps of the success case are specified next.

Note: It has been argued by some that there is no compelling security-related reason that the TGS service needs to be *authenticated* (in the sense of Section 4.13 on page 231; that is, an Authentication Header accompanies the TGS Request). (As seen in Section 4.14.2 on page 245, the TGS service is *not mutually* authenticated; that is, no Reverse-authentication Header accompanies the TGS Response). Nevertheless, the TGS service as specified below is indeed authenticated.

4.14.1 Client Sends TGS Request

[RFC 1510: 3.3.1, A.5]

Consider a client *A* in cell *X* which has in its possession some ticket, say $\mathrm{Tkt}_{A,\cdots,B}$, naming *A* and targeted to some server *B* in some cell *Y* (possibly *X* = *Y* — this important special case is included in everything written in this section). When it is necessary in this section to write out in full the trust chain of this ticket, it will be written as:

$$\mathrm{Tkt}_{A,\cdots,B} = \mathrm{Tkt}_{A,X,\cdots,W,Y,B}$$

As always, $\mathrm{Tkt}_{A,\cdots,B}$ is one of two kinds of ticket, depending on the kind of server (*B*) it is targeted to:

- $\mathrm{Tkt}_{A,\cdots,B}$ may be a service-ticket, in which case *B* is a non-KDS server.
- $\mathrm{Tkt}_{A,\cdots,B}$ may be a ticket-granting-ticket, in which case *B* is the server KDS_Y in one of its guises, $\mathrm{KDS}_{W,Y}$ (possibly $\mathrm{KDS}_{Y,Y}$), as a principal in *Y*.

Since $\mathrm{Tkt}_{A,\cdots,B}$ names *A*, *A* knows the contents of $\mathrm{Tkt}_{A,\cdots,B}$, especially its session key, which is denoted $\mathrm{K}_{A,B}$ (which is of the same encryption type, *encType*, as is used to protect $\mathrm{Tkt}_{A,\cdots,B}$). Note that the key $\mathrm{K}_{A,B}$ can always be used as a session key *between A and* KDS_Y (even though it is nominally only a session key between *A* and *B*, with possibly $B \neq \mathrm{KDS}_Y$). This is because $\mathrm{Tkt}_{A,\cdots,B}$ is protected in the long-term key of *B*, which KDS_Y knows, so KDS_Y has access to $\mathrm{K}_{A,B}$ too.

What *A* wants to do is ''*present*'' $\mathrm{Tkt}_{A,\cdots,B}$ to KDS_Y, and receive in return another ticket, which is denoted:

$$\mathrm{Tkt}^*_{A,\cdots,B^*}$$

which is ''*based on* $\mathrm{Tkt}_{A,\cdots,B}$'', naming *A*, and targeted to another (perhaps the same) server *B** in *Y*. That is, *A* wants to send to KDS_Y a TGS Request *tgsReq* (a value of the data type **TGSRequest**) containing $\mathrm{Tkt}_{A,\cdots,B}$ (*ahTkt*, in its *tgsReq***.req-AuthnData** field, as the *authnHdr***.authnHdr-Tkt** field of an authentication header *authnHdr*), and receive in response a TGS Response *tgsResp* (a value of data type **TGSResponse**) containing $\mathrm{Tkt}^*_{A,\cdots,B^*}$ (*tgsTkt*, in its *tgsResp***.resp-Tkt** field). *A* prepares *tgsReq* according to the algorithm below and ''sends it'' (that is, calls *kds_request*( )) to KDS_Y.

There are two distinct cases to consider throughout, according to what kind of service *A* requests:

- *Request for a ''***manipulated old***'' ticket* (that is, targeted to the same server *B** as the server *B* targeted by the presented ticket) — *A* wants KDS_Y to ''manipulate'' the presented $Tkt_{A,\cdots,B}$ in some way, and return it as $Tkt^*_{A,\cdots,B}$ (currently, the manipulations supported are: validation, renewal, proxying and forwarding; thus, most of the information in $Tkt^*_{A,\cdots,B}$ was already ''substantially pre-existing'' in $Tkt_{A,\cdots,B}$). In this case, $Tkt_{A,\cdots,B}$ can be either a service-ticket or a ticket-granting-ticket.
- *Request for a ''***newly issued***'' ticket* (that is, targeted to a different server *B** than the server *B* targeted by the presented ticket) — *A* wants KDS_Y to issue a new $Tkt^*_{A,\cdots,B^*}$ ''based on'' the presented $Tkt_{A,\cdots,B}$. In this case, $Tkt_{A,\cdots,B}$ must be a ticket-granting-ticket targeted to $B = KDS_{W,Y}$, and the resulting $Tkt^*_{A,\cdots,B^*}$ will be targeted to a server *B** other than $KDS_{W,Y}$ (depending on conditions detailed in the algorithm below):

  — *Service-ticket* (targeted to a non-KDS server *B**)

  $Tkt^*_{A,\cdots,B^*}$ will be a ''new'' service-ticket targeted to a non-KDS server *B** in *Y*.

  — *Cross-cell referral (ticket-granting-)ticket* (targeted to a new KDS server $B^* \neq B$)

  $Tkt^*_{A,\cdots,B^*}$ will be a ''new'' ticket targeted to another KDS server $B^* = KDS_{Y,Z}$ ($Z \neq W$) which is (cross-)registered with *Y*.

Clients (such as *A*) do not typically *intentionally* request cross-cell referral tickets. Except for their initial ticket-granting-ticket they only intentionally request ultimate service-tickets. But if such a request cannot be fulfilled, a cross-cell referral ticket is returned as a by-product of the algorithm, as described below:

- *Protocol version number*

  The protocol version number (*tgsReq***.req-ProtoVersNum**) is set to **protoVersNum-KRB5**.

- *Protocol message type*

  The protocol message type (*tgsReq***.req-ProtoMsgType**) is set to **protoMsgType-TGS-REQUEST**.

- *Client name*

  The client name (*tgsReq***.req-Body.req-ClientName**) is set to *A*'s RS name in RS_X.

- *Server cell*

  The server cell (*tgsReq***.req-Body.req-ServerCell**) is set to the cell name of the ultimate end-server that *A* desires $Tkt^*_{A,\cdots,B^*}$ to be targeted to. (If this is a request for a manipulated old $Tkt^*_{A,\cdots,B^*}$, this ultimate server cell name is the same as $Tkt_{A,\cdots,B}$'s targeted server's cell name (*ahTkt***.tkt-ServerCell**); that is, *Y*'s cell name.)

- *Server name*

  The server name (*tgsReq***.req-Body.req-ServerName**) is set to the RS name of the ultimate server that *A* desires $Tkt^*_{A,\cdots,B^*}$ to be targeted to. (If this is a request for a manipulated old $Tkt^*_{A,\cdots,B^*}$, this ultimate server RS name is the same as $Tkt_{A,\cdots,B}$'s targeted server's RS name (*ahTkt***.tkt-ServerName**), which must exist in RS_Y.)

- *Options*

  — *Forwardable*

If it is desired that a newly issued ticket be forwardable, the forwardable option (*tgsReq***.req-Body.req-Flags.req-Forwardable**) is selected.

— *Proxiable*

If it is desired that a newly issued ticket (which must be a service-ticket, not a ticket-granting-ticket) be proxiable, the proxiable option (*tgsReq***.req-Body.req-Flags.req-Proxiable**) is selected.

— *Postdatable*

If it is desired that a newly issued ticket be postdatable, the postdatable option (*tgsReq***.req-Body.req-Flags.req-Postdatable**) is selected.

— *Postdate*

If it is desired that a newly issued ticket be postdated, the postdate option (*tgsReq***.req-Body.req-Flags.req-Postdate**) is selected.

— *Renewable*

If it is desired that a newly issued ticket be renewable, the renewable option (*tgsReq***.req-Body.req-Flags.req-Renewable**) is selected.

— *Renewable-okay*

If it is not desired that a newly issued ticket be renewable but *A* will nevertheless accept a renewable ticket with a shorter lifetime than desired in lieu of no ticket at all, then the renewable-okay option (*tgsReq***.req-Body.req-Flags.req-RenewableOK**) is selected.

— *Use-session-key*

The use-session-key option (*tgsReq***.req-Body.req-Flags.req-UseSessionKey**) is deselected.

— *Validate*

If it is desired that an old ticket be validated, the validate option (*tgsReq***.req-Body.req-Flags.req-Validate**) is selected.

— *Renew*

If it is desired that an old ticket be renewed, the renew option (*tgsReq***.req-Body.req-Flags.req-Renew**) is selected.

— *Proxy*

If it is desired that an old ticket (which must be a service-ticket, not a ticket-granting-ticket) be proxied, the proxy option (*tgsReq***.req-Body.req-Flags.req-Proxy**) is selected.

— *Forward*

If it is desired that an old ticket be forwarded, the forward option (*tgsReq***.req-Body.req-Flags.req-Forward**) is selected.

- *Start time*

If the postdate option has been selected, then the start time (*tgsReq***.req-Body.req-StartTime**) is set to the desired starting time. Otherwise, the start time is omitted.

- *Expiration time*

The expiration time (*tgsReq***.req-Body.req-ExpireTime**) is set to the desired expiration time. (In the case of a request to manipulate an old ticket, this can be used to ‘‘clip’’ the lifetime of

the manipulated ticket to a shorter time.)

- *Maximum expiration time*

  If the renewable option has been selected, then the maximum expiration time (*tgsReq***.req-Body.req-MaxExpireTime**) is set to the desired maximum expiration time. (In the case of a request to manipulate an old ticket, this can be used to ''clip'' the lifetime of the manipulated ticket to a shorter maximum time.) Otherwise, the maximum expiration time is omitted.

- *Additional tickets*

  The additional tickets field (*tgsReq***.req-Body.req-AdditionalTkts**) is omitted. (The only option that currently requires an additional ticket is the use-session-key option, and that option is deselected in TGS Requests.)

- *Nonce*

  The nonce field (*asReq***.req-Body.req-Nonce**) is set to a nonce value.

- *Encryption types*

  The encryption types field (*tgsReq***.req-Body.req-EncTypes**) is set to the list of encryption types acceptable to *A* for protecting a newly issued ticket. The list is arranged in priority order of desirability, beginning with most desirable and ending with least desirable. (For maximum interoperability, the client *A* should send a list consisting of a single entry, indicating the same encryption type, *encType*, as that used to protect the presented ticket $Tkt_{A,\cdots,B}$. It is to be presumed that if this TGS request is a request for a manipulated old ticket, the resulting manipulated ticket will be re-encrypted in the newly chosen encryption type if it is different from *encType* — however, that is not apparent from RFC 1510.)

- *Client addresses*

  If *A* desires that a newly issued ticket contain client host addresses, then the client address field (*tgsReq***.req-Body.req-ClientAddrs**) is set to the desired addresses. Otherwise, the client address field is omitted. (If this is a request to manipulate an old ticket, these addresses will only be used in the manipulated ticket if the old ticket is forwarded or proxied; otherwise, the client addresses in the ticket authenticating this request — see the bullet on authentication data, below — will be used.)

- *Authorisation data*

  If *A* desires that a newly issued ticket contain authorisation data supplied by *A*, such data (a value of type **AuthzData**) is encrypted using the encryption type *encType* and using the conversation key $\hat{K}_{A,B}$ (*authnr***.authnr-ConversationKey**, see below) if present, otherwise using the session key $K_{A,B}$, and the authorisation data field (*tgsReq***.req-Body.req-EncryptedAuthzData**) is then set to the resulting encrypted value (a value of type **EncryptedData**). Otherwise this field is omitted. (Typically, it is omitted, an exception being where *A* is a privilege server PS_X, requesting a privilege-(ticket-granting-)ticket from KDS_X (in the case where PS_X and KDS_X are not co-located, so that an message must be transmitted) — see Section 1.6 on page 25 and Chapter 5.)

- *Authentication data*

  The first entry (*tgsReq***.req-AuthnData[0]**) in the list (*tgsReq***.req-AuthnData**) of authentication data items is set to have authentication data type (*tgsReq***.req-AuthnData[0].authnData-Type**) **authnDataType-TGS-REQ**, and its authentication data value (*tgsReq***.req-AuthnData[0].authnData-Value**) is set to (the underlying **OCTET STRING** of) an authentication header, *authnHdr*, that *A* constructs, based on $Tkt_{A,\cdots,B}$ (*authnHdr***.authnHdr-Tkt**). The construction of *authnHdr* proceeds as in Section 4.13.1 on page 232, with the

following supplements (*authnr* denotes the authenticator *authnHdr*.**authnHdr-EncryptAuthnr**):

— *Conversation key*

The conversation key field (*authnr*.**authnr-ConversationKey**) is set to a newly generated conversation key, $\hat{K}_{A,B}$ (of the same encryption key type, *encType*), if *A* desires such a key to be used (instead of $K_{A,B}$) to protect this client-server session (between *A* and KDS_Y). (If authorisation data (*tgsReq*.**req-Body.req-EncryptedAuthzData**) and $\hat{K}_{A,B}$ are both present, note that $\hat{K}_{A,B}$ had to be generated earlier in order to be used to encrypt the authorisation data — see above.)

— *Checksum*

A sets the checksum type (*authnr*.**authnr-Cksum.cksum-Type**) to a checksum type that uses the same encryption key type as the encryption type *encType* (except that if *encType* = **encKeyType-TRIVIAL**, then *authnr*.**authnr-Cksum** is omitted altogether), and uses it to compute the checksum value (*authnr*.**authnr-Cksum.cksum-Value**), over the KDS Request body (*tgsReq*.**req-Body**, which is well-formed at this point). (For maximum interoperability, the client *A* should use the checksum type *cksumType-MD4-DES*, since all KDS servers are required to support clients using that checksum type — at least for the present revision of this document.)

— *Sequence number*

The sequence number (*authnr*.**authnr-SeqNum**) is omitted.

— *Authorisation data*

The authorisation data field (*authnr*.**authnr-AuthzData**) is omitted.

— *Encryption*

A encrypts *authnr*, using the encryption type *encType* and the session key $K_{A,B}$ (not $\hat{K}_{A,B}$, even if present). This is the ciphertext portion of the authentication header (*authnHdr*.**authnHdr-EncryptedAuthnr.encData-CipherText**). *A* also sets the encryption type (*authnHdr*.**authnHdr-EncryptedAuthnr.encData-EncType**) to *encType* and the key version number (*authnHdr*.**authnHdr-EncryptedAuthnr.encData-KeyVersNum**) to an appropriate value depending on local policy, if any (typically, it is omitted).

— *Options*

— *Use-session-key*

The use-session-key option (*tgsReq*.**req-Body.req-Flags.req-UseSessionKey**) is deselected.

— *Mutual authentication*

The mutual authentication option (*authnHdr*.**authnHdr-Flags.authnHdr-MutualRequired**) is deselected. (As seen in Section 4.14.2 on page 245, KDS_Y rejects any TGS Request that has this option selected.)

— *Other*

The other entries (*tgsReq*.**req-AuthnData[***i***]**, $i \geq 1$) in the list of authentication data are omitted.

At this point, the *tgsReq* message is well-formed, and *A* sends it to KDS_Y.

4.14.2 KDS Server Receives TGS Request and Sends TGS Response

[RFC 1510: 3.3.2, 3.3.3, A.6]

Consider a TGS Request, *tgsReq*, received by KDS_Y from *A*. Thus, *tgsReq* is a value of data type **TGSRequest**, with protocol version number (*tgsReq***.req-ProtoVersNum**) **protoVersNum-KRB5** and protocol message type (*tgsReq***.req-ProtoMsgType**) **protoMsgType-TGS-REQUEST**. Then KDS_Y executes the algorithm below. If the algorithm executes successfully, KDS_Y returns a TGS Response (*tgsResp*, of type **TGSResponse**), containing the ''manipulated old'' or ''newly issued'' ticket, $Tkt^*_{A,\cdots,B^*}$ (*tgsResp***.resp-Tkt**, also denoted *tgsTkt*), to *A*. If unsuccessful, KDS_Y returns a KDS Error (*kdsErr*, of type **KDSError**).

The following algorithm discusses (in an appropriate order) how KDS_Y handles the authentication header *authnHdr* (and therefore the authenticator *authnr* and presented ticket, *ahTkt* ($Tkt_{A,\cdots,B}$)) accompanying *tgsReq*, how it manipulates the old or issues the new ticket $Tkt^*_{A,\cdots,B^*}$, and how it constructs the remainder of the TGS Response, *tgsResp* (which does not include a reverse-authentication header). (The manner in which KDS_Y handles the authentication header is an extension of the ''mainline'' usage of the authentication header by not-necessarily-KDS servers as specified in Section 4.13.2 on page 234, but it is all repeated here, both because its processing is intertwined with the processing of other parts of the TGS Request, and to indicate error conditions specific to KDS servers.)

- *Authentication data*

  The first entry (*tgsReq***.req-AuthnData[0]**) in the authentication data list (*tgsReq***.req-AuthnData**) is checked to be present and to be of authentication data type (*tgsReq***.req-AuthnData[0].authnData-Type**) **authnDataType-TGS-REQ**. (Other entries (*tgsReq***.req-AuthnData[***i***]**, $i \geq 1$), if present, are ignored.) Thus, the authentication data value (*tgsReq***.req-AuthnData[0].authnData-Value**) is regarded as (the underlying **OCTET STRING** of) an authentication header, *authnHdr*, containing an authenticator *authnr* (*authnHdr***.authnHdr-EncryptAuthnr**) constructed by *A*, and an ''authenticating ticket'' $Tkt_{A,\cdots,B}$ (*ahTkt*, *authnHdr***.authnHdr-Tkt**) naming *A* and targeted to some server *B* in cell *Y* (which may be either a non-KDS server or KDS_Y in one of its guises $KDS_{W,Y}$) {**errStatusCode-AUTHN-DATA-TYPE-NOT-SUPPORTED**}.

- *Protocol version number*

  The protocol version number of $Tkt_{A,\cdots,B}$ (*ahTkt***.tkt-ProtoVersNum**) is checked to be **protoVersNum-KRB5**.

- *Server cell*

  The server cell in $Tkt_{A,\cdots,B}$ (*ahTkt***.tkt-ServerCell**) is checked to be KDS_Y's cell name; that is, *Y*'s cell name.

- *Server name*

  The server name in $Tkt_{A,\cdots,B}$ (*ahTkt***.tkt-ServerName**) is checked to indicate either a non-KDS server *B* in *Y*, or KDS_Y in one of its guises $KDS_{W,Y}$.

- *Authentication header options*:

  — *Use-session-key*

  If the use-session-key option (*authnHdr***.authnHdr-Flags.authnHdr-UseSessionKey**) is selected, then KDS_Y rejects this TGS Request, returning a KDS Error. Otherwise, KDS_Y knows that the ''authenticating'' ticket, *ahTkt* ($Tkt_{A,\cdots,B}$), is protected with the long-term key K_B of the server *B* it is targeted to (as indicated by *ahTkt***.tkt-ServerCell** and *ahTkt***.tkt-ServerName**) ($K_B = K_{KDSWY}$ if $B = KDS_{W,Y}$).

— *Mutual authentication*

If the mutual authentication option (*authnHdr***.authnHdr-Flags.authnHdr-MutualRequired**) is selected (so that *A* expects a reverse-authentication header, to be returned), then KDS_Y rejects this TGS Request, returning a KDS Error (see Section 4.15 on page 258).

- *Ticket decryption*

  KDS_Y uses the key protecting *ahTkt* ($K_B = K_{KDSWY}$, determined above), together with the indicated encryption type *encType* (*ahTkt***.tkt-EncryptedPart.encData-EncType**) and key version number (*ahTkt***.tkt-EncryptedPart.encData-KeyVersNum**) if relevant, to decrypt the ciphertext (*ahTkt***.tkt-EncryptedPart.encData-CipherText**) of the authenticating ticket *ahTkt* ($Tkt_{A,\cdots,B}$). A successful decryption is recognised by the built-in integrity afforded by the ciphertext itself. In this way, KDS_Y learns the information carried in *ahTkt* ($Tkt_{A,\cdots,B}$), especially its session key ($K_{A,B}$, or $K_{A,KDSWY}$).

- *Authenticator decryption*

  KDS_Y uses *ahTkt*'s session key ($K_{A,B}$, or $K_{A,KDSWY}$), together with the encryption type *encType* (which must be the same as (*authnHdr***.authnHdr-EncryptedAuthnr.encData-EncType**), to decrypt the authentication header's ciphertext (*authnHdr***.authnHdr-EncryptedAuthnr.encData-CipherText**). (The key version number (*authnHdr***.authnHdr-EncryptedAuthnr.encData-KeyVersNum**) should not be present.) A successful decryption is recognised by the built-in integrity afforded by the ciphertext itself — *this is what convinces* KDS_Y *that A knows ahTkt's session key*; that is, this is what actually ''authenticates'' *A* to KDS_Y (modulo timestamp considerations, below). In this way, KDS_Y learns the information carried in *authnr* (*authnHdr***.authnHdr-EncryptAuthnr**).

- *Authenticator protocol version number*

  The authenticator's protocol version number (*authnr***.authnr-ProtoVersNum**) is checked to be **protoVersNum-KRB5**.

- *Client cell*

  The client cells from *ahTkt* (*ahTkt***.tkt-EncryptPart.tkt-ClientCell**) and from *authnr* (*authnr***.authnr-ClientCell**) are checked to be the same (namely, to *A*'s cell name; that is, *X*'s cell name).

- *Client name*

  The client names from *ahTkt* (*ahTkt***.tkt-EncryptPart.tkt-ClientName**) and from *authnr* (*authnr***.authnr-ClientName**) are checked to be the same (namely, to *A*'s RS name).

- *Client addresses*

  If there are client addresses present in *ahTkt* (*ahTkt***.tkt-EncryptPart.tkt-ClientAddrs**), then KDS_Y checks that *A* is communicating from one of them (as reported by KDS_Y's operating system, according to the level of trust KDS_Y places in that); that is, that *authnHdr* was received from one of the addresses on the list.

- *Client timestamp*

  A's timestamp (*authnr***.authnr-ClientTime**) is checked to be equal to KDS_Y's system time (modulo **maxClockSkew**). It is in this way that KDS_Y becomes convinced that it is communicating with *A* in real-time (and this completes the ''authentication'' of *A* to KDS_Y).

- *Client microsecondstamp*

  A's microsecondstamp (*authnr***.authnr-ClientMicroSec**), together with its timestamp, are checked to not be present in KDS_Y's replay cache. They are then stored in the replay cache. (They can be purged from the replay cache later, as discussed in Section 4.5 on page 200.)

- *Start time*

  ahTkt's start time (*ahTkt***.tkt-EncryptPart.tkt-StartTime**) is checked to be earlier than or equal to KDS_Y's system time (modulo **maxClockSkew**).

- *Expiration time*

  ahTkt's expiration time (*ahTkt***.tkt-EncryptPart.tkt-ExpireTime**) is checked to be later than (or later-than-or-equal-to, on an implementation-dependent basis) KDS_Y's system time (modulo **maxClockSkew**). (In particular, an expired *ahTkt* ($Tkt_{A,\cdots,B}$) cannot be renewed by the renew option processing step, below.)

- *Maximum expiration time*

  ahTkt's maximum expiration time (*ahTkt***.tkt-EncryptPart.tkt-MaxExpireTime**) is dealt with below.

- *Sequence number*

  The sequence number (*authnr***.authnr-SeqNum**) is processed in an application-specific manner.

- *Conversation key*

  If a conversation key $\hat{K}_{A,B}$ (*authnr***.authnr-ConversationKey**), of encryption type *encType* is present, KDS_Y will use it (instead of *ahTkt*'s session key $K_{A,B}$ (or K_{A,KDS_WY})) to protect this client-server session (but it will not generate its own conversation key, $\hat{\hat{K}}_{A,B}$, for this purpose, because KDS_Y does not return a reverse-authentication header).

- *Transit path*

  ahTkt's transit path (*ahTkt***.tkt-EncryptPart.tkt-TransitPath**) is dealt with below.

- *Checksum*

  The checksum (*authnr***.authnr-Cksum**) is checked to be present (unless *encType* = **encType-TRIVIAL**, in which case it must be absent), its checksum type (*authnr***.authnr-Cksum.cksum-Type**) is checked to be supported by and acceptable to KDS_Y (in particular, this checksum type must use the same encryption key type as the encryption type *encType*, and the checksum value (*authnr***.authnr-Cksum.cksum-Value**) is checked to be the checksum of the KDS Request body (*tgsReq***.req-Body**). (For guaranteed interoperability, all KDS servers are required to support the checksum type *cksumType-MD4-DES* — at least for the present revision of this document.)

- *Ticket options*:

  — *Invalid*

  If *ahTkt*'s invalid option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Invalid**) is selected, then KDS_Y checks that this *tgsReq*'s validate option (*tgsReq***.req-Body.req-Flags.req-Validate**) is selected.

  — *Forwardable, forwarded, proxiable, proxied, postdatable, postdated, renewable, initial*

  All other *ahTkt* options are dealt with below. Currently, these include forwardable (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Forwardable**), forwarded (*ahTkt***.tkt-EncryptPart.tkt-**

Flags.tkt-Forwarded), proxiable (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Proxiable**), proxied (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Proxied**), postdatable (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Postdatable**), postdated (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Postdated**), renewable (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Renewable**), and initial (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Initial**).

At this point, KDS_Y has completed its preliminary processing of the authentication header *authnHdr* (including *authnr* and *ahTkt* ($Tkt_{A,\cdots,B}$)), and now turns its attention to constructing the manipulated old or newly-to-be-issued *tgsTkt* ($Tkt^*_{A,\cdots,B^*}$).

- *Protocol version number*

  tgsTkt's protocol version number (*tgsTkt***.tkt-ProtoVersNum**) is set to **protoVersNum-KRB5**.

- *Client cell*

  tgsTkt's client cell (*tgsTkt***.tkt-EncryptPart.tkt-ClientCell**) is set to *A*'s (authenticated) cell name (that is, to *X*'s cell name).

- *Client name*

  tgsTkt's client name (*tgsTkt***.tkt-EncryptPart.tkt-ClientName**) is set to *A*'s (authenticated) RS name in RS_X.

- *Server cell*

  If the requested server cell name (*tgsReq***.req-Body.req-ServerCell**) is not KDS_Y's cell name — that is, not *Y*'s cell name (this will be the case if *A* is requesting a service-ticket to an ultimate end-server in some cell other than *Y*) — then KDS_Y sets *tgsTkt*'s cell name (*tgsTkt***.tkt-ServerCell**) to that of a cell, *Z*, that *A* is supposed to use as the next hop towards the requested server cell, if possible — this indicates to *A* that *tgsTkt* is to be used as a new *cross-cell referral ticket* (see Section 1.7 on page 32), and it is the *only instance* in which a KDS server ever issues a ticket which is targeted to a server other than the one requested by the client *A* (and this is how *A* detects that it has received a cross-cell referral ticket). Otherwise, KDS_Y copies the requested server cell name (that is, *Y*'s cell name) into *tgsTkt*'s cell name.

- *Server name*

  If *tgsTkt* is a new cross-cell referral ticket (see preceding step), then KDS_Y sets *tgsTkt*'s server name (*tgsTkt***.tkt-ServerName**) to the RS name of the chosen cross-registered surrogate KDS server, $KDS_{Y,Z}$, in RS_Z. Otherwise, the requested server name (*tgsReq***.req-Body.req-ServerName**) is checked to have a datastore entry in RS_Y, and KDS_Y sets *tgsTkt*'s server name to the requested server name.

- *Encryption types*

  KDS_Y selects from the list of requested encryption types (*tgsReq***.req-Body.req-EncTypes**) the earliest one on the list that it can accommodate, depending on policy — call this *encType*. (Typically, *encType* = encType*. This will happen, for example, in the common case of a client *A* sending a list consisting of a single entry, indicating the same encryption type, *encType*, as that used to protect the presented ticket $Tkt_{A,\cdots,B}$.) {**errStatusCode-ENCRYPTION-TYPE-NOT-SUPPORTED**}.

- *Session key generation*

  KDS_Y generates a new (random) session key (of the selected encryption type, *encType**), K^*_{A,B^*}, and copies it into *tgsTkt*'s session key field (*tgsTkt***.tkt-EncryptPart.tkt-SessionKey**).

- *Authentication time*

tgsTkt's authentication time (*tgsTkt***.tkt-EncryptPart.tkt-AuthnTime**) is set to *ahTkt*'s authentication time (*ahTkt***.tkt-EncryptPart.tkt-AuthnTime**).

- *Start time*

  If the requested start time (*tgsReq***.req-Body.req-StartTime**) is absent, or if it is present and indicates a time earlier than *tgsTkt*'s (that is, *ahTkt*'s) authentication time or KDS_Y's system time, then *tgsTkt*'s start time (*tgsTkt***.tkt-EncryptPart.tkt-StartTime**) is set to *tgsTkt*'s authentication time or KDS_Y's system time, whichever is later. Otherwise (that is, a start time is present and indicates a time later than or equal to both *tgsTkt*'s authentication time and KDS_Y's system time), if the postdated option (*tgsReq***.req-Body.req-Flags.req-Postdate**) is selected, KDS_Y sets *tgsTkt*'s start time to the requested start time; otherwise, *tgsTkt*'s start time is omitted. (See also the postdated and invalid options, below.) {**errStatusCode-CANNOT-POSTDATE**}.

- *Expiration time*

  KDS_Y sets *tgsTkt*'s expiration time (*tgsTkt***.tkt-EncryptPart.tkt-ExpireTime**) to the minimum of the following:

  — *ahTkt*'s expiration time (*ahTkt***.tkt-EncryptPart.tkt-ExpireTime**).

  — The requested expiration time (*tgsReq***.req-Body.req-ExpireTime**).

  — *tgsTkt*'s start time (or authentication time, if the start time is absent) plus the maximum ticket lifetime associated with the named client *A* (or the maximum ticket lifetime associated with the cross-cell principal $KDS_{W,Y}$ in the case that $X \neq Y$, since then KDS_Y doesn't have access to *A*'s maximum ticket lifetime).

  — *tgsTkt*'s start time (or authentication time, if the start time is absent) plus the maximum ticket lifetime associated with the server targeted by *tgsTkt* (*tgsTkt***.tkt-ServerName**).

  — *tgsTkt*'s start time (or authentication time, if the start time is absent) plus the cell-wide maximum ticket lifetime associated with the issuing authority KDS_Y.

  KDS_Y checks that the resulting lifetime of *tgsTkt* is greater than or equal to the cell-wide minimum ticket lifetime associated with the issuing authority KDS_Y (see also the renew and renewable options, below) {**errStatusCode-NEVER-VALID**}.

- *Maximum expiration time*

  If *ahTkt*'s renewable option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Renewable**) is selected, and if either the renewable option (*tgsReq***.req-Body.req-Flags.req-Renewable**) has been selected, or if the renewable-okay option (*tgsReq***.req-Body.req-Flags.req-RenewableOK**) has been selected and *ahTkt*'s expiration time is earlier than the requested expiration time, then *tgsTkt*'s maximum expiration time (*tgsTkt***.tkt-EncryptPart.tkt-MaxExpireTime**) is present (otherwise it is omitted) and is set to the minimum of:

  — *ahTkt*'s maximum expiration time (*ahTkt***.tkt-EncryptedPart.tkt-MaxExpireTime**).

  — The requested maximum expiration time (*tgsReq***.req-Body.req-MaxExpireTime**), if present; otherwise, the requested expiration time (*tgsReq***.req-Body.req-ExpireTime**).

  — *tgsTkt*'s start time (or authentication time, if the start time is absent) plus the maximum renewable ticket lifetime associated with the named client *A* (or the maximum ticket lifetime associated with the cross-cell principal $KDS_{W,Y}$ in the case that $X \neq Y$, since then KDS_Y doesn't have access to *A*'s maximum ticket lifetime).

  — *tgsTkt*'s start time (or authentication time, if the start time is absent) plus the maximum renewable ticket lifetime associated with the server targeted by *tgsTkt* (*tgsTkt***.tkt-**

ServerName).

— *tgsTkt*'s start time (or authentication time, if the start time is absent) plus the cell-wide maximum renewable ticket lifetime associated with the issuing authority KDS_Y.

(See also the renewable option, below, which may recalculate this maximum expiration time.)

- *Transit path*

  If *ahTkt* is itself a cross-cell referral ticket, $Tkt_{A,X,\cdots,W,KDSWY}$ (which KDS_Y recognises by decrypting it and inspecting its transit path), then KDS_Y sets *tgsTkt*'s transit path (*tgsTkt***.tkt-EncryptPart.tkt-TransitPath**) to the compression (see Section 4.2.5.1 on page 170) of the concatenation (in this order) of *ahTkt*'s transit path with *W*'s (not *Y*'s) cell name. Otherwise (that is, if *ahTkt* is not a cross-cell referral ticket), *tgsTkt*'s transit path is set to *ahTkt*'s transit path {**errStatusCode-TRANSIT-PATH-TYPE-NOT-SUPPORTED**}.

- *Client addresses*

  tgsTkt's client address field (*tgsTkt***.tkt-EncryptPart.tkt-ClientAddrs**) is set to *ahTkt*'s client address field (*ahTkt***.tkt-EncryptPart.tkt-ClientAddrs**), if present. Otherwise, it is omitted. (See also the forward and proxy options, below, which can change this client address field.)

- *Authorisation data*

  tgsTkt's authorisation data field (*tgsTkt***.tkt-EncryptPart.tkt-AuthzData**) is set to the concatenation (in this order) of *ahTkt*'s authorisation data (*ahTkt***.tkt-EncryptPart.tkt-AuthzData**) if present, with the TGS Request's authorisation data (*tgsReq***.req-Body.req-EncryptAuthzData**, obtained by decrypting *tgsReq***.req-Body.req-EncryptedAuthzData** using the conversation key *authnr***.authnr-ConversationKey** $\hat{K}_{A,Y}$ if present, otherwise using the session key $K_{A,B}$, both of encryption type *encType*), if present. If neither is present, this field is omitted.

  Notes:

  1. The server targeted by *tgsTkt* is thereby guaranteed of the authenticity of *tgsTkt*'s authorisation field — see Section 1.6 on page 25 and Chapter 5 on page 263 for an explanation of how this mechanism is used by the PS.
  2. *authnr*'s additional authenticator authorisation data (*authnr***.authnr-AuthzData**) is ignored here. Its use is application-specific, and *A* intends it to be interpreted only by an ultimate non-KDS end-server *B*, not KDS_Y.
  3. The authorisation data is uninterpreted by KDS_Y — see Section 4.3.8 on page 194 for an indication of its interpretation by end-servers. Even though it is uninterpreted by KDS_Y, it must handled in a trusted fashion, for otherwise a client could potentially insert arbitrary authorisation data and defeat the integrity of PAC transmission via these authorisation data fields, and illicitly masquerade as another principal for authorisation purposes.

- *Additional tickets*

  Additional tickets (*tgsReq***.req-Body.req-AdditionalTkts**) are processed as required according to the options (below) selected that require additional tickets.

- *Options*

  — *Forwardable*

  If the forwardable option (*tgsReq***.req-Body.req-Flags.req-Forwardable**) is requested, and if *ahTkt*'s forwardable option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Forwardable**) is

selected, then *tgsTkt*'s forwardable option (*tgsTkt***.tkt-EncryptPart.tkt-Flags.tkt-Forwardable**) is selected.

— *Forward*

If the forward option (*tgsReq***.req-Body.req-Flags.req-Forward**) is requested, and if *ahTkt*'s forwardable option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Forwardable**) is selected, then *tgsTkt*'s client addresses (*tgsTkt***.tkt-EncryptPart.tkt-ClientAddrs**) are set to the requested client addresses (*tgsReq***.req-Body.req-ClientAddrs**).

— *Forwarded*

If the forward option (*tgsReq***.req-Body.req-Flags.req-Forward**) is requested, or if *ahTkt*'s forwarded option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Forwarded**) is selected, then *tgsTkt*'s forwarded option (*tgsTkt***.tkt-EncryptPart.tkt-Flags.tkt-Forwarded**) is selected.

— *Proxiable*

If the proxiable option (*tgsReq***.req-Body.req-Flags.req-Proxiable**) is requested, and if *ahTkt*'s proxiable option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Proxiable**) is selected, then *tgsTkt*'s proxiable option (*tgsTkt***.tkt-EncryptPart.tkt-Flags.tkt-Proxiable**) is selected.

— *Proxy*

If the proxy option (*tgsReq***.req-Body.req-Flags.req-Proxy**) is requested, and if *ahTkt*'s proxiable option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Proxiable**) is selected, then *tgsTkt*'s client addresses (*tgsTkt***.tkt-EncryptPart.tkt-ClientAddrs**) are set to the requested client addresses (*tgsReq***.req-Body.req-ClientAddrs**).

— *Proxied*

If the proxy option (*tgsReq***.req-Body.req-Flags.req-Proxy**) is requested, or if *ahTkt*'s proxied option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Proxied**) is selected, then *tgsTkt*'s proxied option (*tgsTkt***.tkt-EncryptPart.tkt-Flags.tkt-Proxied**) is selected.

— *Postdatable*

If the postdatable option (*tgsReq***.req-Body.req-Flags.req-Postdatable**) is requested, and if *ahTkt*'s postdatable option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Postdatable**) is selected, then *tgsTkt*'s postdatable option (*tgsTkt***.tkt-EncryptPart.tkt-Flags.tkt-Postdatable**) is selected.

— *Postdated*

If *tgsTkt*'s start time is present, and if *ahTkt*'s postdatable option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Postdatable**) is selected, then *tgsTkt*'s postdated option (*tgsTkt***.tkt-EncryptPart.tkt-Flags.tkt-Postdated**) is selected.

— *Invalid*

If *tgsTkt*'s postdated option is selected (above), then *tgsTkt*'s invalid option (*tgsTkt***.tkt-EncryptPart.tkt-Flags.tkt-Invalid**) is selected.

— *Validate*

If the validate option (*tgsReq***.req-Body.req-Flags.req-Validate**) is requested, and if *ahTkt*'s invalid option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Invalid**) is set, and if *ahTkt*'s start time (*ahTkt***.tkt-EncryptPart.tkt-StartTime**) is earlier than or equal to KDS_Y's system time (modulo **maxClockSkew**), then $Tkt^*_{A,\cdots,B^*}$ is set equal to $Tkt_{A,\cdots,B}$, except that *tgsTkt*'s invalid option (*tgsTkt***.tkt-EncryptPart.tkt-Flags.tkt-Invalid**) is deselected.

— *Renew*

If the renew option (*tgsReq***.req-Body.req-Flags.req-Renew**) is requested, and if *ahTkt*'s renewable option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Renewable**) is selected, and if *ahTkt*'s maximum expiration time (*ahTkt***.tkt-EncryptPart.tkt-MaxExpireTime**) is later than or equal to KDS_Y's system time (modulo **maxClockSkew**), then *tgsTkt* is set equal to *ahTkt*, except that *tgsTkt*'s start time (*tgsTkt***.tkt-EncryptPart.tkt-StartTime**) is set to KDS_Y's system time and *tgsTkt*'s expiration time (*tgsTkt***.tkt-EncryptPart.tkt-ExpireTime**) is set to the minimum of:

— *ahTkt*'s maximum expiration time (*ahTkt***.tkt-EncryptPart.tkt-MaxExpireTime**).

— *tgsTkt*'s start time (which is KDS_Y's system time at this point) plus the lifetime of *ahTkt* (*ahTkt***.tkt-EncryptPart.tkt-ExpireTime** – *ahTkt***.tkt-EncryptPart.tkt-StartTime**).

— *Renewable-okay*

If the renewable-okay option (*tgsReq***.req-Body.req-Flags.req-RenewableOK**) is requested, and if *ahTkt*'s renewable option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Renewable**) is selected, and if *tgsTkt*'s expiration time (*tgsTkt***.tkt-EncryptPart.tkt-ExpireTime**) is earlier than the requested expiration time (*tgsReq***.req-Body.req-ExpireTime**), then the renewable option (*tgsReq***.req-Body.req-Flags.req-Renewable**) is selected (so that the renewable step below is processed) and the requested maximum expiration time (*tgsReq***.req-Body.req-MaxExpireTime**) is set equal to the minimum of:

— *ahTkt*'s maximum expiration time (*ahTkt***.tkt-EncryptPart.tkt-MaxExpireTime**).

— The requested expiration time (*tgsReq***.req-Body.req-ExpireTime**).

— *Renewable*

If the renewable option (*tgsReq***.req-Body.req-Flags.req-Renewable**) option is requested, and if *ahTkt*'s renewable option (*ahTkt***.tkt-EncryptPart.tkt-Flags.tkt-Renewable**) is selected, then *tgsTkt*'s renewable option (*tgsTkt***.tkt-EncryptPart.tkt-Flags.tkt-Renewable**) is selected, and *tgsTkt*'s maximum expiration time (*tgsTkt***.tkt-EncryptPart.tkt-MaxExpireTime**) is (re)calculated as in the maximum expiration time step, above.

— *Use-session-key*

If the request's use-session-key option (*tgsReq***.req-Body.req-Flags.req-UseSessionKey**) has not been requested, then KDS_Y knows that *A* wants *tgsTkt* to be protected with the long-term key of its targeted server (see encryption step, below). If the request's use-session-key has been requested, then KDS_Y knows that *A* wants *tgsTkt* to be protected with its targeted server's ticket-granting-ticket's session key (see encryption step, below), accordingly it verifies that the accompanying additional $Tkt^{\bullet}$ (*tgsReq***.req-Body.req-AdditionalTkts**) is present and is a (valid) ticket-granting-ticket targeted to *B*; if the use-session-key has been requested, but $Tkt^{\bullet}$ is not present or if KDS_Y cannot decrypt it and verify it is a ticket-granting-ticket targeted to *B*, the KDS_Y rejects the request.

— *Initial*

tgsTkt's initial option (*tgsTkt***.tkt-EncryptPart.tkt-Flags.tkt-Initial**), is deselected. (This marks *tgsTkt* as a *subsequent* (that is, non-initial) ticket.)

- *Encryption*

If the use-session-key option (*tgsReq***.req-Body.req-Flags.req-UseSessionKey**) has not been requested, then the long-term key K_{B^*} (of encryption type *encType**) of the server *B** targeted by *tgsTkt* ($Tkt^*_{A,\cdots,B^*}$) is used to protect it; if the use-session-key option has been requested, then the session key $K^{\bullet} = K_{B,KDS_{WY}}$ (of encryption type *encType**) in the accompanying ticket-

granting-ticket $Tkt^{\bullet} = Tkt_{B,KDS_{WY}}$ is used to protect it (see Section 4.6.2 on page 203). KDS_Y uses the chosen key to encrypt the encrypted part of *tgsTkt* (*tgsTkt***.tkt-EncryptPart**). This is the ciphertext portion of *tgsTkt* (*tgsTkt***.tkt-EncryptedPart.encData-CipherText**). KDS_Y also sets the encryption type (*tgsTkt***.tkt-EncryptedPart.encData-EncType**) to *encType**, and the key version number (*tgsTkt***.tkt-EncryptedPart.encData-KeyVersNum**) is set to its appropriate value.

At this point, *tgsTkt* ($Tkt^*_{A,\cdots,B^*}$) is well-formed, and KDS_Y turns its attention to completing the construction of the TGS Response message, *tgsResp*.

- *Protocol version number*

  The protocol version number (*tgsResp***.resp-ProtoVersNum**) is set to **protoVersNum-KRB5**.

- *Protocol message type*

  The protocol message type (*tgsResp***.resp-ProtoMsgType**) is set to **protoMsgType-TGS-RESPONSE**.

- *Authentication data*

  The authentication data (*tgsResp***.resp-AuthnData**) is omitted. (In particular, no reverse-authentication header is transmitted back to the client.)

- *Client cell*

  The client cell (*tgsResp***.resp-ClientCell**) is set to *tgsTkt*'s client cell name (*tgsTkt***.tkt-EncryptPart.tkt-ClientCell**); that is, *A*'s cell name to *X*'s cell name.

- *Client name*

  The client name (*tgsResp***.resp-ClientName**) is set to *tgsTkt*'s client name (*tgsTkt***.tkt-EncryptPart.tkt-ClientName**); that is, *A*'s RS name.

- *Ticket*

  The ticket (*tgsResp***.resp-Tkt**) is set to *tgsTkt* ($Tkt^*_{A,\cdots,B^*}$).

- *Session key*

  The session key (*tgsResp***.resp-EncryptPart.resp-SessionKey**) is set to *tgsTkt*'s ($Tkt^*_{A,\cdots,B^*}$'s) session key, K^*_{A,B^*} (*tgsTkt***.tkt-EncryptPart.tkt-SessionKey**, of encryption type *encType**).

- *Nonce*

  The nonce (*tgsResp***.resp-EncryptPart.resp-Nonce**) is set to the nonce that *A* sent (*tgsReq***.req-Body.req-Nonce**).

- *Client addresses*

  The client address field (*tgsResp***.resp-EncryptPart.resp-ClientAddrs**) is set to *tgsTkt*'s client address field if *tgsTkt* has been newly forwarded or proxied on this TGS Request (that is, the forward option (*tgsReq***.req-Body.req-Flags.req-Forward**) is selected, above). Otherwise, it is omitted.

- *Server cell*

  The server cell (*tgsResp***.resp-EncryptPart.resp-ServerCell**) is set to *tgsTkt*'s server cell (*tgsTkt***.tkt-ServerCell**).

- *Server name*

  The server name (*tgsResp***.resp-EncryptPart.resp-ServerName**) is set to *tgsTkt*'s server name (*tgsTkt***.tkt-ServerName**).

- *Authentication time*

  The authentication time (*tgsResp***.resp-EncryptPart.resp-AuthnTime**) is set to *tgsTkt*'s authentication time (*tgsTkt***.tkt-EncryptPart.tkt-AuthnTime**).

- *Start time*

  The start time (*tgsResp***.resp-EncryptPart.resp-StartTime**) is set to *tgsTkt*'s start time (*tgsTkt***.tkt-EncryptPart.tkt-StartTime**), if present. Otherwise, it is omitted.

- *Expiration time*

  The expiration time (*tgsResp***.resp-EncryptPart.resp-ExpireTime**) is set to *tgsTkt*'s expiration time (*tgsTkt***.tkt-EncryptPart.tkt-ExpireTime**).

- *Maximum expiration time*

  The maximum expiration time (*tgsResp***.resp-EncryptPart.resp-MaxExpireTime**) is set to *tgsTkt*'s maximum expiration time (*tgsTkt***.tkt-EncryptPart.tkt-MaxExpireTime**), if present. Otherwise, it is omitted.

- *Key expiration date*

  The key expiration date (*tgsResp***.resp-EncryptPart.resp-KeyExpireDate**) is omitted.

- *Last requests*

  The last requests field (*tgsResp***.resp-EncryptPart.resp-LastRequests**) is set, if policy permits, to *A*'s last requests information (see Section 4.2.10 on page 176), if available. (If this is a cross-cell request, that information wouldn't be available. Even if the information is available, KDS_Y may or may not send it back to the client, according to its policy; normally it is included only in AS Responses, not TGS Responses — see Section 4.9.1 on page 213.)

- *Options*

  The options (*tgsResp***.resp-EncryptPart.resp-Flags**) are set to *tgsTkt*'s options (*tgsTkt***.tkt-EncryptPart.tkt-Flags**).

- *Encryption*

  KDS_Y encrypts *tgsResp***.resp-EncryptPart** using the encryption type *encType* and *ahTkt*'s ($Tkt_{A,\cdots,B}$'s) session key $K_{A,B}$, unless *A* has requested that a conversation key $\hat{K_{A,B}}$ (*authnr***.authnr-ConversationKey**) be used, in which case KDS_Y uses that key instead. This is the ciphertext portion of the TGS Response (*tgsResp***.resp-EncryptedPart.encData-CipherText**). KDS_Y also sets the encryption type (*tgsResp***.resp-EncryptedPart.encData-EncType**) to *encType*. The key version number (*tgsResp***.resp-EncryptedPart.encData-KeyVersNum**) is omitted.

At this point, the KDS Response is well-formed, and the KDS returns it to the calling client.

4.14.3 Client Receives TGS Response

[RFC 1510: 3.3.4, A.4, A.7]

Consider a client *A* that receives a TGS Response, *tgsResp* (that is, *tgsResp* is a value of data type **TGSResponse**, with protocol version number (*tgsResp***.resp-ProtoVersNum**) **protoVersNum-KRB5** and protocol message type (*tgsResp***.resp-ProtoMsgType**) **protoMsgType-TGS-RESPONSE**), in response to a TGS Request, *tgsReq* (as the result of calling *kds_request*( )) to KDS_Y. *A* processes *tgsResp* according to the algorithm below. In the case this algorithm completes successfully, *A* is justified in believing that the returned *tgsTkt* (or $Tkt^*_{A,\cdots,B^*}$; that is, *tgsResp***.resp-Tkt**) is correctly and securely targeted to the server *B** specified (*tgsResp***.resp-**

EncryptPart.resp-ServerCell and *tgsResp***.resp-EncryptPart.resp-ServerName**), and that it contains the values returned elsewhere in the *tgsResp* (in particular, that *A* is the client named by *tgsTkt*), and using it (especially, its session key, $K^{*}_{A,B*}$, of encryption type *encType**) in subsequent Authentication Headers it sends in service requests to the targeted server, or in subsequent TGS Requests. In the case the algorithm fails, *A* takes (application-specific) recovery action.

Here, the notions of ''success'' or ''failure'' of this algorithm are taken to mean ''conforming to *A*'s request (*tgsReq*)'', where the criteria of ''conformance'' are application-specific. Typically, but not necessarily, *A* will be satisfied only if KDS_Y formulates the TGS Response exactly as *A* requested. For example, *A* may have requested a very long maximum expiration time but KDS_Y issued only a somewhat shorter one — whether *A* views that as a success or failure is an application-specific determination.

- *Client cell*

  The named client's cell (*tgsResp***.resp-ClientCell**) is checked for conformance to what *A* requested in the authenticator to its TGS Request (*authnr***.authnr.ClientCell**, where *authnr* is carried in *tgsReq***.req-AuthnData** as described previously).

- *Client name*

  The client name (*tgsResp***.resp-ClientName**) is checked for conformance to what *A* requested (*tgsReq***.req-Body.req-ClientName**).

- *Authentication data*

  The authentication data (*tgsResp***.resp-AuthnData**) is ignored.

- *Ticket*

  tgsTkt (*tgsResp***.resp-Tkt**) is not directly interpretable (in the sense of being decryptable) by *A* (unless *A* happens to also be the server targeted by *tgsTkt*), but the information in it is largely available elsewhere in *tgsResp*.

- *Decryption*

  The encryption type, *encType* (*tgsResp***.resp-EncryptedPart.encData-EncType**), is checked to be the same as the encryption type protecting the *ahTkt* ($Tkt_{A,\cdots,B}$) *A* had presented to KDS_Y in the authentication header of its TGS Request (*authnHdr***.authnHdr-Tkt**). If it is acceptable, then *A* attempts to decrypt the ciphertext portion of the TGS Response (*tgsResp***.resp-EncryptedPart.encData-CipherText**), using the session key $K_{A,B}$ from *ahTkt*, unless *A* has requested that a conversation key $\hat{K}_{A,B}$ (*authnr***.authnr-ConversationKey**) be used, in which case *A* uses that key instead. (Note that this differs from the case of an AS Response, which is protected with *A*'s long-term key, not a session key.) A successful decryption is recognised by the built-in integrity afforded by the ciphertext itself. In this way, *A* learns the information carried in *tgsResp***.resp-EncryptPart**. If *A* encounters an ''unsuccessful'' decryption, it takes application-specific action — this presumably includes rejection of *tgsResp* as untrustworthy (the ability to successfully decrypt *tgsResp***.resp-EncryptedPart** proves to *A* that it was encrypted by the legitimate KDS_Y (since *A* trusts the key $K_{A,B}$ or $\hat{K}_{A,B}$ to be secure), and that it is not being spoofed by a counterfeit KDS_Y. In particular, if *A* is not the client requested in *tgsReq*, in the sense of not knowing the correct session key ($K_{A,B}$ or $\hat{K}_{A,B}$), then *A* will not be able to successfully decrypt *tgsResp*, and consequently will not be able to gain access to the information in *tgsResp***.resp-EncryptPart** (in particular, its session key, $K^{*}_{A,B*}$ (*tgsResp***.resp-EncryptPart.resp-SessionKey**, of encryption type *encType** — see below)).

- *Nonce*

  The nonce (*tgsResp***.resp-EncryptPart.resp-Nonce**) is checked for equality with the requested nonce (*tgsReq***.req-Body.req-Nonce**). If it is not equal, then this *tgsResp* does not correspond

to *tgsReq*, and a ''replay attack'' may be suspected, to which *A* takes application-specific action.

- *Last requests*

  The last requests (*tgsResp***.resp-EncryptPart.resp-LastRequests**) are typically ignored (it may be inspected if present, but it is typically not present — see Section 4.9.1 on page 213).

- *Key expiration date*

  The key expiration date (*tgsResp***.resp-EncryptPart.resp-KeyExpireDate**) is typically ignored (it may be inspected if present, but it is typically not present — see Section 4.9.1 on page 213).

- *Server cell*

  The server cell (*tgsResp***.resp-EncryptPart.resp-ServerCell**) is checked for conformance to what *A* requested (*tgsReq***.req-Body.req-ServerCell**). (See also next step.)

- *Server name*

  The server name (*tgsResp***.resp-EncryptPart.resp-ServerName**) is checked for conformance to what *A* requested (*tgsReq***.req-Body.req-ServerName**). (If the server cell (*tgsResp***.resp-EncryptPart.resp-ServerCell**) and server name (*tgsResp***.resp-EncryptPart.resp-ServerName**) do not identify the server that *A* requested, then *A* knows *tgsTkt* is a new cross-cell referral ticket, and *A* will normally send a TGS Request to the new cross-cell KDS server it names.)

- *Session key*

  The encryption type, *encType** (*tgsTkt***tkt-EncryptedPart.encData-Enctype**), of the session key (which *A* trusts is secure), K^*_{A,B^*} (*tgsResp***.resp-EncryptPart.resp-SessionKey**), is checked for conformance to what *A* had requested (*tgsReq***reg-Body.reg-EncTypes**), and K^*_{A,B^*} is saved for later use (for example, in protecting communications with the server *B**).

- *Authentication time*

  The authentication time (*tgsResp***.resp-EncryptPart.resp-AuthnTime**) is checked for conformance to what *A* expects (namely, *A*'s authentication time from *ahTkt* and from *A*'s original initial ticket-granting-ticket on which *tgsTkt* is ultimately based).

- *Start time*

  The start time (*tgsResp***.resp-EncryptPart.resp-StartTime**) is checked for conformance to what *A* requested. Namely, if *A* requested *tgsTkt* to be postdated, then the start time is checked to be present and checked for conformance to the start time *A* requested (*tgsReq***.req-Body.req-StartTime**); otherwise, the start time should be absent.

- *Expiration time*

  The expiration time (*tgsResp***.resp-EncryptPart.resp-ExpireTime**) is verified for conformance to what *A* requested (*tgsReq***.req-Body.req-ExpireTime**).

- *Maximum expiration time*

  The maximum expiration time (*tgsResp***.resp-EncryptPart.resp-MaxExpireTime**) is checked for conformance to what *A* requested. It should only be present (at most) if *A* had selected the renewable option (*tgsReq***.req-Body.req-Flags.req-Renewable**) and supplied a requested maximum expiration time (*tgsReq***.req-Body.req-MaxExpireTime**), or if *A* had selected the renewable-okay option (*tgsReq***.req-Body.req-Flags.req-RenewableOK**).

- *Client addresses*

If *A* requested that *tgsTkt* contain client host addresses (as part of a forward or proxy request), then the client address field (*tgsResp***.resp-EncryptPart.resp-ClientAddrs**) is verified to be present and checked for conformance to what *A* requested (*tgsReq***.req-Body.req-ClientAddrs**). Otherwise, the client addresses are checked for conformance to the client addresses in the ticket-granting-ticket accompanying the TGS Request (*ahTkt***.tkt-EncryptPart.tkt-ClientAddrs**), if present.

- *Options*

  The options field (*tgsResp***.resp-EncryptPart.resp-Flags**) is inspected for conformance to what *A* requested (*tgsReq***.req-Body.req-Flags**).

This completes the specification of the TGS Request / Response exchange.

4.15 KDS Error Processing

[RFC 1510: 3.1.6, A.20]

This section specifies in detail the processing that occurs when a KDS server encounters a failure during its processing of an AS Request or a TGS Request, and returns a KDS Error to the requesting client. (Actually, KDS Error messages may be of some use to arbitrary application servers, not just KDS servers. That scenario is not examined in this section, though the discussion given here can easily be generalised to that situation.)

Consider a KDS Request, *kdsReq*, received by KDS_Y from a client *A*. This KDS Request is either an AS Request or a TGS Request, and KDS_Y performs the algorithm specified above accordingly. If it encounters one or more algorithmic failures, it chooses one (normally, the first one it encounters dynamically) to return in a KDS Error message. KDS_Y then proceeds to execute the algorithm below. This results in KDS_Y returning a KDS Error *kdsErr* (of type **KDSError**) to *A*.

authnr is written for the authenticator accompanying a KDS Request (carried in *kdsReq***.req-AuthnData** as described previously), provided it is present and KDS_Y can decrypt the encrypted authenticator successfully. In that case, the description is *authnr* (and the information in it) ''is available''.

- *Protocol version number*

  The protocol version number (*kdsErr***.err-ProtoVersNum**) is set to **protoVersNum-KRB5**.

- *Protocol message type*

  The protocol message type (*kdsErr***.err-ProtoMsgType**) is set to **protoMsgType-KDS-ERROR**.

- *Client cell*

  The client cell (*kdsErr***.err-ClientCell**) is set to the request's client cell (*authnr***.authnr-ClientCell**), if available. Otherwise, it is omitted.

- *Client name*

  The client name (*kdsErr***.err-ClientName**) is set to the request's client name (*authnr***.authnr-ClientName**), if available. Otherwise, it is omitted.

- *Server cell*

  The server cell (*kdsErr***.err-ServerCell**) is set to the request's server cell name (*kdsReq***.req-ServerCell**), which is KDS_Y's cell name; that is, *Y*'s cell name.

- *Server name*

  The server name (*kdsErr***.err-ServerName**) is set to the request's server's RS name (*kdsReq***.req-ServerName**), which is KDS_Y's RS name.

- *Client timestamp*

  The client timestamp (*kdsErr***.err-ClientTime**) is set to the request's client timestamp (*authnr***.authnr-ClientTime**), if available. Otherwise, it is omitted.

- *Client microsecondstamp*

  The client microsecondstamp (*kdsErr***.err-ClientMicroSec**) is set to the request's client microsecondstamp (*authnr***.authnr-ClientMicroSec**), if available. Otherwise, it is omitted.

- *Server timestamp*

  The server timestamp (*kdsErr***.err-ServerTime**) is set to KDS_Y's system time.

- *Server microsecondstamp*

  The server microsecondstamp (*kdsErr***.err-ServerMicroSec**) is set to KDS_Y's microsecondstamp.

- *Status code*

  The status code (*kdsErr***.err-StatusCode**) is set to the status code of the error being reported by this KDS Error message.

- *Status text*

  The status text (*kdsErr***.err-StatusText**) is set to the status text associated with the status code being reported, if any. Otherwise it is omitted.

- *Status data*

  The status data (*kdsErr***.err-StatusData**) is set to the status data associated with the status code being reported, if any. Otherwise it is omitted.

This completes the specification of the KDS Error message processing.

4.16 Cross-cell Authentication

[RFC 1510: 1.1]

As seen in Section 4.12 on page 220 and Section 4.14 on page 240, the authentication of a client *A* in cell *X* to a server *B* in cell *Y* is quite straightforward if $X = Y$. But the case $X \neq Y$ requires a sequence of non-obvious *cross-cell referrals*. The low-level details have already been specified above (in Section 4.14 on page 240), but without a higher-level understanding of cross-cell referrals the whole scenario remains obscure and mysterious. This section presents this higher-level view (see also Section 1.7 on page 32).

Consider a client *A* in cell *X* that wants to authenticate to an ultimate non-KDS end-server *B* in cell *Y*, with $X \neq Y$. *A* begins by obtaining an (initial or subsequent) ticket, $\text{Tkt}_{A,\text{KDS}X}$, protected with KDS_X's long-term key $\text{K}_{\text{KDS}X}$.

A then sends a TGS Request to KDS_X, presenting $\text{Tkt}_{A,\text{KDS}X}$ (in **req-AuthnData**) to KDS_X, requesting a service-ticket targeted to *B* in *Y*. Note that *A* must know the principal name of *B* (comprising the cell name of *Y* and the RS name of *B* in *Y*), but *A* does not *a priori* know the intermediate cells in the trust chain between *X* and *Y* — that is, *A* knows the structure of the namespace, but only the network of KDS servers knows the structure of the trust graph (consisting of the cross-cell registrations of KDS servers with one another).

Since *B*'s home cell is *Y* ($\neq X$), KDS_X does not know *B*'s long-term key K_B, and so it cannot construct the requested service-ticket targeted to *B*. Instead, KDS_X chooses a cell *Z* which is cross-registered with *X* to be used as the next hop towards *Y*, and returns to *A* a TGS Response which contains (in **resp-Tkt**) a cross-cell referral ticket, $\text{Tkt}_{A,X,\text{KDS}XZ}$. This TGS Response and $\text{Tkt}_{A,X,\text{KDS}XZ}$ contain a newly generated session key $\text{K}_{A,\text{KDS}XZ}$ between *A* and $\text{KDS}_{X,Z}$, and $\text{Tkt}_{A,X,\text{KDS}XZ}$ is protected with the long-term key $\text{K}_{\text{KDS}XZ}$ shared between the two surrogate KDS principals $\text{KDS}_{X,Z}$ cross-registered in RS_X and in RS_Z.

When *A* receives this TGS Response from KDS_X, it recognises that it has received (**resp-Tkt**) a cross-cell referral ticket instead of the service-ticket it had requested, because the TGS Response contains information (in **resp-ServerCell** and **resp-ServerName**) telling *A* that the target of **resp-Tkt** is not the server *B* that *A* had requested (and a cross-cell referral ticket is the only instance in which a KDS server ever issues a ticket targeted to a server other than that requested by the client). Accordingly, *A* now formulates a new TGS Request, to KDS_Z ($\text{KDS}_{X,Z}$) this time, again requesting a service-ticket targeted to *B*. The ticket *A* present (in **req-AuthnData**) in this TGS Request to KDS_Z is the cross-cell referral ticket, $\text{Tkt}_{A,X,\text{KDS}XZ}$, *A* just received from KDS_X.

When KDS_Z receives this TGS Request, it decrypts $\text{Tkt}_{A,X,\text{KDS}XZ}$ with the long-term key $\text{K}_{\text{KDS}XZ}$, thereby learning its session key $\text{K}_{A,\text{KDS}Z}$ (this authenticates *A* to KDS_Z and secures communications between them, and establishes the $A \rightarrow \text{KDS}_X \rightarrow \text{KDS}_Z$ trust chain).

Now shift notation slightly (for purposes of an inductive argument) and write Z' instead of *Z*. If $Z' = Y$, then $\text{KDS}_{Z'} = \text{KDS}_Y$ can satisfy *A*'s TGS Request, and can return to *A* the $\text{Tkt}_{A,X,Y,B}$ *A* requested. Otherwise, the above procedure is iterated: $\text{KDS}_{Z'}$ returns (if possible) to *A* another new cross-cell referral ticket, $\text{Tkt}_{A,X,Z',\text{KDS}Z'Z''}$, to a next-hop cell Z'' which is even closer to the desired cell *Y*. This process continues through a sequence of cross-cell referrals, Z', Z'', ···, Z''', until it eventually terminates (if no errors are encountered) when the desired cell $Z'''' = Y$ is ultimately reached. For at that point, *A*'s TGS Request to KDS_Y ($\text{KDS}_{Z'''Y}$) for a service-ticket targeted to *B* (protected in *B*'s long-term key K_B) will finally be satisfied, and is returned to *A* in the TGS Response from KDS_Y. *A* recognises that it has finally received a service-ticket targeted to its desired end-server *B*, so can then proceed to use this $\text{Tkt}_{A,X,Z',Z'',\cdots,Z''',Y,B}$ to authenticate itself to, and protect its communications with, *B*.

In particular, note that the successive steps of this cross-cell authentication scenario are all *secure*, because the referral information (cell name and RS name of successive KDS servers to be

contacted, session keys, and so on) is securely determined by a chain of trusted KDS servers and securely transmitted to *A* at each stage.

Finally, it is to be noted that this chapter has not explained how *authorisation data* or *UUIDs* make their appearance in the protocol. That is the province of the Privilege Service, as specified in Chapter 5.

Chapter 5

Privilege (Authorisation) Services

This chapter specifies the privilege (or authorisation), services supported by DCE, together with the protocols associated with them. Currently, two such services are supported, namely *PAC-based Privilege Service* (PS) and *Name-based Privilege Service*. Of these two, PAC-based authorisation used to be the more important and better supported, and most of this chapter is devoted to it. With the advent of delegation, extensions have been made to the PAC in order to support delegation controls and extensible restrictions. These extend the notion of identity to include chained identities, and to allow delegation and extensible restrictions to be specified.

In addition, the extensions also include certain attributes, those dealing with the data repository or registry. Including them as extensions to the PAC, designated EPAC, permits them to be transmitted securely and automatically along with a principal's identity information.

In order to ensure the integrity of the data being transmitted across the network in a delegated environment, name-based authorization is used. This is because specific attributes must be requested and examined prior to transmission. Name-based authorisation is treated briefly at the end of the chapter.

Throughout this chapter the notation *ps_request_**( ), wherever it appears, is used in a generic sense to mean any one of *ps_request_ptgt*( ), *ps_request_eptgt*( ), *ps_request_become_delegate*( ), or *ps_request_become_impersonator*( ) as appropriate. In addition, where it is used elsewhere in this document, it is used in the same sense.

For an overview of this chapter, see Section 1.5 on page 18 through Section 1.7 on page 32 of this specification — which are considered a *prerequisite* for this whole chapter.

5.1 PAC-based Privilege Service (PS)

The PS is a distributed, partitioned RPC service, instantiated by a (conceptually unitary, but potentially replicated) RPC server in each cell *Z*, denoted PS_Z. If the name of cell *Z* is, say, **/.../cellZ**, then the *RPC service name* of PS_Z (used for RPC binding purposes) is determined from **/.../cellZ/cell-profile** via the **rpriv** interface UUID and version number (specified in Section 5.1.1) — typically, the name associated with this profile element will be **/.../cellZ/sec**, which will be an RPC server group pointing to the individual (replicated) PS_Z server(s). (The *principal names* of PS servers (used for security purposes) — as opposed to their RPC server names — are introduced later in this chapter.)

5.1.1 The rpriv RPC Interface

Each PS server, PS_Z, supports the following RPC interface:

```
[uuid(b1e338f8-9533-11c9-a34a-08001e019c1e), version(1.1),
 pointer_default(ptr)]
interface rpriv {
 /* begin running listing of rpriv interface */
```

5.1.1.1 ps_message_t

This is the definition of the encoding used for the data in the **PTGS Request/PTGS Response** message pair by which clients acquire privilege-ticket-granting-tickets. (See Section 5.1.6 on page 275.)

```
typedef struct {
    unsigned32                      count;
    [size_is(count)] byte           bytes[];
}                                       ps_message_t;
```

5.1.1.2 ps_attr_request_t

This is the definition of the structure to encapsulate information relevant to an optional auxiliary attributes request from the privilege server (PS).

```
typedef struct {
    unsigned32              count_auxiliary_attribute_keys;
    [ptr, size_is(count_auxiliary_attribute_keys)]
        sec_attr_t          *auxiliary_attribute_keys;
}                                       ps_attr_request_t;
```

5.1.1.3 ps_attr_result_t

This is the definition of the structure to encapsulate information relevant to an optional auxiliary attributes response from the privilege server (PS).

```
typedef struct {
    error_status_t      status;
    unsigned32          count_attrs;
    [ptr, size_is(count_attrs)]
        sec_attr_t      *attrs;
}                                       ps_attr_result_t;
```

5.1.1.4 ps_app_tkt_result_t

This is the definition of the structure to encapsulate information relevant to an optional application service ticket response used for delegation — for extended-privilege-ticket-granting-tickets, as well as for becoming a delegate or an impersonator on behalf of a client.

```
typedef struct {
    error_status_t              status;
    [ptr] ps_message_t          *application_ticket_result;
}                                        ps_app_tkt_result_t;
```

5.1.1.5 ps_request_ptgt

```
void
ps_request_ptgt (
    [in]  handle_t                      rpc_handle,
    [in]  unsigned32                    authn_service,
    [in]  unsigned32                    authz_service,
    [in]  ps_message_t                  *request,
    [out] ps_message_t                  **response,
    [out, ref] error_status_t           *status
);
```

The semantics of *ps_request_ptgt*( ) are that a client, *C*, invokes *ps_request_ptgt*( ) to ‘‘send a *PS Request message*’’ to a PS server, PS_Z; *C* receives a *PS Response message* from PS_Z when *ps_request_ptgt*( ) returns. Its parameters are the following:

- *rpc_handle*

  RPC binding handle, bound by the client *C* to a PS server PS_Z.

- *authn_service*

  The authentication (or key distribution) service for which this invocation of *ps_request_ptgt*( ) is requesting a PAC. The currently supported authentication services are collected in Section 5.1.2 on page 273.

- *authz_service*

  The authorisation service for which this invocation of *ps_request_ptgt*( ) is requesting a privilege attribute certificate. The currently registered authorisation services are collected in Section 5.1.3 on page 273.

- *request*

  (Pointer to) PS Request message. It has length **(****request***).count** and value **(****request***).bytes[]**. See Section 5.2.10 on page 282.

- *response*

  (Pointer to pointer to) PS Response message. It has length **(*****response***).count** and value **(*****response***).bytes[]**. See Section 5.2.11 on page 283 and Section 5.2.12 on page 283.

- *status*

  (Pointer to) status code. See Section 5.1.4 on page 273.

5.1.1.6 ps_request_become_delegate

```
void
ps_request_become_delegate (
    [in]  handle_t                      rpc_handle,
    [in]  unsigned32                    authn_service,
    [in]  unsigned32                    authz_service,
    [in] sec_id_delegation_type_t       delegation_type_permitted,
    [in] sec_id_restriction_set_t       *delegate_restrictions,
    [in] sec_id_restriction_set_t       *target_restrictions,
    [in] sec_id_opt_req_t               *optional_restrictions,
    [in] sec_id_opt_req_t               *required_restrictions,
    [in] sec_id_compatibility_mode_t    compatibility_mode,
    [in] sec_bytes_t                    *delegation_chain,
    [in] sec_bytes_t                    *delegate,
    [in] sec_encrypted_bytes_t          *delegation_token,
    [in, ref]  ps_message_t             *request,
    [out] ps_message_t                  **response,
    [out] sec_bytes_t                   *new_delegation_chain,
    [in, ptr] ps_attr_request_t         *auxliary_attribute_request,
    [out] ps_attr_result_t              *auxliary_attribute_result,
    [in, ptr, string] unsigned char     *application_ticket_request,
    [out] ps_app_tkt_result_t           *application_ticket_result,
    [out, ref] error_status_t           *status
 );
```

The semantics of *ps_request_become_delegate*( ) are that an intermediary caller (on behalf of a Client, C,) invokes *ps_request_become_delegate*( ) to ‘‘send a *PS Request message*’’ to a PS server, PS_Z for an entire delegation chain. The delegation chain includes the EPAC(s) and associated Delegation Token (DT) from the intermediary’s caller, along with the intermdiary’s EPAC and *PS Request message*. The PS Server, PS_Z, once it ensures that no tampering has taken place (by verifying that the Delegation Token for the delegation chain is correct), will create a new delegation chain from the existing one with the intermediary’s EPAC as a new delegate. If any EPAC(s) in the chain have a delegate restriction preventing this intermediary (server) from transmitting it’s identity, the PS server will replace that participant’s EPAC with an anonymous EPAC. (See Section 1.20.7.1 on page 96).

The intermediary caller receives a *PS Response message* from PS_Z when *ps_request_become_delegate*( ) returns. This response pertains to the delegation chain. The *PS Response message* will contain a seal in the A_D field encrypted in the key of the privilege server, PS_Z, as the new delegation token for this delegation chain. In addition, the A_D field will also contain the seal of the EPAC chain. This seal is an MD5 checksum for each EPAC in the chain, and an MD5 checksum of those checksums.

The parameters of *ps_request_become_delegate*( ) are the following:

- *rpc_handle*

  RPC binding handle, bound by the client *C* to a PS server PS_Z.

- *authn_service*

  The authentication (or key distribution) service for which this invocation of *ps_request_become_delegate*( ) is requesting an EPAC. The currently supported authentication services are collected in Section 5.1.2 on page 273.

- *authz_service*

  The authorisation service for which this invocation of *ps_request_become_delegate*( ) is requesting a privilege attribute certificate. The currently registered authorisation services are collected in Section 5.1.3 on page 273.

- *delegation_type_permitted*

  Determines the delegation type to be permitted. This is specified by the client when it permits delegation to be enabled. Only two types are permitted — *traced delegation* or *impersonation*. An intermediary (server) is not permitted to use a delegation type that was not enabled by the initiator. For the data type definitions, see Section 5.2.13.6 on page 285.

- *delegate_restrictions*

  (Pointer to) the list of delegates that are permitted. Delegate restrictions are set by a client and limit the servers that may act as an intermediary for the client. The restriction is imposed by the PS when constructing a new PTGT that permits the intermediary to operate as a delegate for the client. For more information see Section 5.2.13.3 on page 284. For definitions of the entries for the restrictions, see Section 5.2.13.2 on page 284.

- *target_restrictions*

  (Pointer to) the list of targets to whom this identity may be presented. The restrictions are placed by a given client to restrict the set of targets to whom the client's identity may be projected. These restrictions are imposed by the PS_Z of the cell Z (the target cell). For more information see Section 5.2.13.3 on page 284. For definitions of the entries for the restrictions, see Section 5.2.13.2 on page 284.

- *optional_restrictions*

  (Pointer to) the list of (application defined) optional restrictions denoting specific authorization requirements. These restrictions may be set by initiators and delegates and apply to the delegation context (they are interpreted and enforced by the target server application). A server is free to ignore any optional restriction that cannot be interpreted. See Section 5.2.13.1 on page 283 for the data type definition.

- *required_restrictions*

  (Pointer to) the list of (application defined) required restrictions denoting specific authorization requirements. These restrictions may be set by initiators and delegates and apply to the delegation context (they are interpreted and enforced by the target server application). A server must reject requests for which there is a required restriction that cannot be interpreted. See Section 5.2.13.1 on page 283 for the data type definition.

- *compatibility_mode*

  Specifies the compatibility mode desired when operating on DCE 1.0 servers. The extensions to the PAC required by delegation are not understood by DCE 1.0 servers. This parameter determines the contents of the of the start of the Authorization Data (A_D) field. See Section 5.2.13.5 on page 285 for a description of the values and Section 1.20.1 on page 90 for a description of the DCE 1.1 A_D field in the PTGT.

- *delegation_chain*

  (Pointer to) a set of chained EPACs. This set of (one or more) EPACs is encrypted in the same session key used to ensure the privacy of the arguments in the RPC call, if the authenticated RPC call is using protect level packet_privacy (more specifically, **rpc_c_protect_level_pkt_privacy**, listed in Section 1.10 on page 54).

Note that for DCE 1.1, the chain of EPACs must all reside in the same cell. Thus, if a delegation request traverses outside the cell - for instance, from cell A to cell B, no further delegation is permitted - that is, a server in cell B may perform the function requested, but may not delegate the request to any other server.

- *delegate*

  (Pointer to) the EPAC for the intermediary that is issuing the *ps_request_become_delegate*( ).

- *delegation_token*

  (Pointer to) he encrypted seal of the EPAC chain. It consists of an MD5 checksum over the checksums for each EPAC in the chain of EPACs (known as the seal of the EPACs), encrypted in the key of the Privilege Server (PS_Z). This encrypted checksum is inserted into the A_D field of the EPAC in order to be passed along with the Authorization Data, in any requests (authenticated RPC calls) made subsequent to this call to other servers.

- *request*

  (Pointer to) PS Request message. It has length **(****request**)*.count** and value **(****request**)*.bytes[]**. See Section 5.2.10 on page 282.

- *response*

  (Pointer to pointer to) PS Response message. It has length **(*****response***)*.count** and value **(*****response***)*.bytes[]**. See Section 5.2.11 on page 283 and Section 5.2.12 on page 283.

- *new_delegation_chain*

  (Pointer to) a chain constructed from the existing (input to this function) one with the intermediary's EPAC as a new delegate. As noted previously, the input delegation chain may be modified if any EPACs in the chain have a delegate restriction preventing this intermediary server (making this *ps_request_become_delegate*( ) request) from transmitting their identity. In such instances, each such EPAC will be replaced by an anonymous EPAC. See Section 1.20.7.1 on page 96 for more details.

- *auxiliary_attribute_request*

  (Pointer to) types of attributes. Auxiliary attributes only have meaning in local requests to the PS. This is an optional parameter which specifies instances of the types of attributes to be searched for on the node of the principal for whom *ps_request_become_delegate*( ) is granted.

  Note: This parameter is not implemented in this version of DCE (DCE 1.1).

- *auxiliary_attribute_result*

  (Pointer to) the results obtained from the search for instances of types of attributes specified in *auxiliary_attribute_request*. For this version of DCE (DCE 1.1), auxiliary attributes are not implemented. Upon return, this parameter is set to the value **sec_rgy_not_implemented** in DCE 1.1.

- *application_ticket_request*

  (Pointer to) a string of characters. This is an optional parameter. It specifies (a pointer to) an application service ticket request which consists solely of the principal's name on whose behalf this PS request is being made. (See Section 4.2.7 on page 174).

- *application_ticket_result*

  (Pointer to) information relevant to an optional application service ticket response. This information is in the form of a structure consisting of a status code and the ticket response. For this version of DCE (DCE 1.1), application ticket requests are not implemented. Upon

return, this parameter is set to the value **sec_rgy_not_implemented** in DCE 1.1. See Section 5.1.1.4 on page 264.

- *status*

  (Pointer to) status code. See Section 5.1.4 on page 273.

5.1.1.7 ps_request_become_impersonator

```
void
ps_request_become_impersonator (
    [in]  handle_t                      rpc_handle,
    [in]  unsigned32                    authn_service,
    [in]  unsigned32                    authz_service,
    [in] sec_id_delegation_type_t       delegation_type_permitted,
    [in] sec_id_restriction_set_t       *delegate_restrictions,
    [in] sec_id_restriction_set_t       *target_restrictions,
    [in] sec_id_opt_req_t               *optional_restrictions,
    [in] sec_id_opt_req_t               *required_restrictions,
    [in] sec_bytes_t                    *delegation_chain,
    [in] sec_bytes_t                    *impersonator,
    [in] sec_encrypted_bytes_t          *delegation_token,
    [in]  ps_message_t                  *request,
    [out] ps_message_t                  **response,
    [out] sec_bytes_t                   *new_delegation_chain,
    [in, ptr] ps_attr_request_t         *auxliary_attribute_request,
    [out] ps_attr_result_t              *auxliary_attribute_result,
    [in, ptr, string] unsigned char     *application_ticket_request,
    [out] ps_app_tkt_result_t           *application_ticket_result,
    [out, ref] error_status_t           *status
);
```

The semantics of *ps_request_become_impersonator*( ) are that an intermediary caller (on behalf of a Client, C,) invokes *ps_request_become_impersonator*( ) to ‘‘send a *PS Request message*’’ to a PS server, PS_Z for the initiator’s EPAC (The initiator’s EPAC includes the EPAC(s) and associated Delegation Token (DT) from the intermediary’s caller), along with the intermediary’s EPAC and *PS Request message*. The PS Server, PS_Z, once it ensures that no tampering has taken place (by verifying that the Delegation Token for the initiator’s EPAC is correct), will then verify that the intermediary is permitted to impersonate the initiator.

The intermediary caller receives a *PS Response message* from PS_Z when *ps_request_become_impersonator*( ) returns. This response pertains to the impersonator’s EPAC. The *PS Response message* will contain a seal of the initiator’s EPAC in the A_D field. This seal is an MD5 checksum for the initiator’s EPAC. In addition, the A_D field will also contain that same seal encrypted in the key of the privilege server, PS_Z, as the new delegation token for the impersonators’s EPAC.

The parameters of *ps_request_become_impersonator*( ) are the following:

- *rpc_handle*

  RPC binding handle, bound by the client *C* to a PS server PS_Z.

- *authn_service*

  The authentication (or key distribution) service for which this invocation of *ps_request_become_impersonator*( ) is requesting an EPAC. The currently supported

authentication services are collected in Section 5.1.2 on page 273.

- *authz_service*

  The authorisation service for which this invocation of *ps_request_become_impersonator*( ) is requesting a privilege attribute certificate. The currently registered authorisation services are collected in Section 5.1.3 on page 273.

- *delegation_type_permitted*

  Determines the delegation type to be permitted. This is specified by the client when it permits delegation to be enabled. Only two types are permitted — *traced delegation* or *impersonation*. An intermediary (server) is not permitted to use a delegation type that was not enabled by the initiator. For the data type definitions, see Section 5.2.13.6 on page 285.

- *delegate_restrictions*

  (Pointer to) the list of delegates that are permitted. Delegate restrictions are set by a client and limit the servers that may act as an intermediary for the client. The restriction is imposed by the PS when constructing a new PTGT that allows the intermediary to operate as a delegate for the client. See Section 5.2.13.3 on page 284.

- *target_restrictions*

  (Pointer to) the list of targets to whom this identity may be presented. The restrictions are placed by a given client to restrict the set of targets to whom the client's identity may be projected. These restrictions are imposed by the PS_Z of the cell *Z* (the target cell). See Section 5.2.13.3 on page 284.

- *optional_restrictions*

  (Pointer to) the list of (application defined) optional restrictions denoting specific authorization requirements. These restrictions may be set by initiators and delegates and apply to the delegation context (they are interpreted and enforced by the target server application). A server is free to ignore any optional restriction that cannot be interpreted.

- *required_restrictions*

  (Pointer to) the list of (application defined) required restrictions denoting specific authorization requirements. These restrictions may be set by initiators and delegates and apply to the delegation context (they are interpreted and enforced by the target server application). A server must reject requests for which there is a required restriction that cannot be interpreted.

- *delegation_chain*

  (Pointer to) a set of chained EPACs. This set of (one or more) EPACs is encrypted in the same session key used to ensure the privacy of the arguments in the RPC call, if the authenticated RPC call is using protect level packet_privacy (more specifically, **rpc_c_protect_level_pkt_privacy**, listed in Section 1.10 on page 54).

- *impersonator*

  (Pointer to) the EPAC for the intermediary that is issuing the *ps_request_become_impersonator*( ).

- *delegation_token*

  (Pointer to) the encrypted seal of the EPAC. It consists of an MD5 checksum over the initiator's EPAC (known as the seal of the EPAC), encrypted in the key of the Privilege Server (PS_Z). This encrypted checksum is inserted into the A_D field of the EPAC in order to be

passed along with the Authorization Data, in any requests (authenticated RPC calls) made subsequent to this call to other servers, as the new delegation token for this identity.

- *request*

  (Pointer to) PS Request message. It has length **(****request***).count** and value **(****request***).bytes[]**. See Section 5.2.10 on page 282.

- *response*

  (Pointer to pointer to) PS Response message. It has length **(*****response***).count** and value **(*****response***).bytes[]**. See Section 5.2.11 on page 283 and Section 5.2.12 on page 283.

- *new_delegation_chain*

  (Pointer to) a chain constructed from the existing (input to this function) one with the intermediary's EPAC as an impersonator of the initiator. As noted previously, the input delegation chain may be modified if any EPACs in the chain have a delegate restriction preventing this intermediary server (making this *ps_request_become_impersonator*( ) request) from transmitting their identity. In such instances, each such EPAC will be replaced by an anonymous EPAC. See Section 1.20.7.1 on page 96 for more details.

- *auxiliary_attribute_request*

  (Pointer to) types of attributes. Auxiliary attributes only have meaning in local requests to the PS. This is an optional parameter which specifies instances of the types of attributes to be searched for on the node of the principal for whom *ps_request_become_impersonator*( ) is granted.

  Note: This parameter is not implemented in this version of DCE (DCE 1.1).

- *auxiliary_attribute_result*

  (Pointer to) the results obtained from the search for instances of types of attributes specified in *auxiliary_attribute_request*. For this version of DCE (DCE 1.1), auxiliary attributes are not implemented. Upon return, this parameter is set to the value **sec_rgy_not_implemented** in DCE 1.1.

- *application_ticket_request*

  This is an optional parameter. It specifies (a pointer to) an application service ticket request which consists solely of the principal's name on whose behalf this PS request is being made.

- *application_ticket_result*

  (Pointer to) information relevant to an optional application service ticket response. This information is in the form of a structure consisting of a status code and the ticket response. For this version of DCE (DCE 1.1), application ticket requests are not implemented. Upon return, this parameter is set to the value **sec_rgy_not_implemented** in DCE 1.1.

- *status*

  (Pointer to) status code. See Section 5.1.4 on page 273.

5.1.1.8 ps_request_eptgt

```
void
ps_request_eptgt (
    [in]  handle_t                      rpc_handle,
    [in]  unsigned32                    authn_service,
    [in]  unsigned32                    authz_service,
    [in] sec_bytes_t                    *requested_privileges,
    [in]  ps_message_t                  *request,
    [out] ps_message_t                  **response,
    [out] sec_bytes_t                   *granted_privileges,
    [in, ptr] ps_attr_request_t         *auxliary_attribute_request,
    [out] ps_attr_result                *auxliary_attribute_result,
    [in, ptr, string] unsigned char     *application_ticket_request,
    [out] ps_app_tkt_result_t           *application_ticket_result,
    [out, ref] error_status_t           *status
);
```

The semantics of *ps_request_eptgt*( ) are that a client, *C*, invokes *ps_request_eptgt*( ) using name-based authorisation to ensure the integrity of the parameters across the network to ‘‘send a *PS Request message*’’ to a PS server, PS_Z; *C* receives a *PS Response message* containing an Extended PAC (EPAC) from PS_Z when *ps_request_ptgt*( ) returns. The *PS Request message* may optionally request specific attributes. The parameters of *ps_request_eptgt*( ) are the following:

- *rpc_handle*

  RPC binding handle, bound by the client *C* to a PS server PS_Z.

- *authn_service*

  The authentication (or key distribution) service for which this invocation of *ps_request_eptgt*( ) is requesting an EPAC. The currently supported authentication services are collected in Section 5.1.2 on page 273.

- *authz_service*

  The authorisation service for which this invocation of *ps_request_eptgt*( ) is requesting a privilege attribute certificate. The currently registered authorisation services are collected in Section 5.1.3 on page 273.

- *requested_privileges*

  The set of privileges being requested from the PS. These privileges are usually found in one or more (encoded) extended PACs (unless the privileges being requested are not valid), for local requests. If the request is an intercell request, the PS uses the EPAC seal from the authorization data contained in the extended PTGT (EPTGT) in examining the privileges.

- *request*

  (Pointer to) PS Request message. It has length **(****request***).count** and value **(****request***).bytes[]**. See Section 5.2.10 on page 282.

- *response*

  (Pointer to pointer to) PS Response message. It has length **(*****response***).count** and value **(*****response***).bytes[]**. See Section 5.2.11 on page 283 and Section 5.2.12 on page 283.

- *granted_privileges*

  An encoded EPAC set (of one or more EPACs) containing the granted privileges.

- *auxiliary_attribute_request*

  Auxiliary attributes only have meaning in local requests to the PS. This is an optional parameter which specifies instances of the types of attributes to be searched for on the node of the principal for whom *ps_request_eptgt*( ) is granted.

  Note: This parameter is not implemented in this version of DCE (DCE 1.1).

- *auxiliary_attribute_result*

  For this version of DCE (DCE 1.1), auxiliary attributes are not implemented. Upon return, this parameter is set to the value **sec_rgy_not_implemented** in DCE 1.1.

- *application_ticket_request*

  This is an optional parameter. It specifies (a pointer to) an application service ticket request which consists solely of the principal's name on whose behalf this PS request is being made.

- *application_ticket_result*

  (Pointer to) information relevant to an optional application service ticket response. This information is in the form of a structure consisting of a status code and the ticket response. For this version of DCE (DCE 1.1), application ticket requests are not implemented. Upon return, this parameter is set to the value **sec_rgy_not_implemented** in DCE 1.1.

- *status*

  (Pointer to) status code. See Section 5.1.4.

} /* end running listing of rpriv interface */

5.1.2 Registered Authentication Services

The currently registered values for *authn_service* are the following:

- **ps_c_authn_secret** = 1

  Kerberos authentication (key distribution) service (as defined in Chapter 4).

5.1.3 Registered Authorisation Services

The currently registered values for *authz_service* are the following:

- **ps_c_authz_dce** = 2

  PAC-based authorisation service (as defined in this chapter).

5.1.4 Status Codes

[RFC 24.2: 2.]

The following status codes (transmitted as values of the type **error_status_t**) are specified for the **rpriv** interface. Only their values are specified here — their use is specified elsewhere in this specification. See RFC 24.2 for details of how the status code values are determined. Within that context, the base value for the component security whose mnemonic is ''sec'', translates into the base value, in hex, of 17122. For any given message, the last three digits appended to this base are the actual index value into the message catalog (in hex) of that message.

```
const unsigned32     sec_priv_s_server_unavailable    = 0x1712205a;
const unsigned32     sec_priv_s_invalid_principal     = 0x1712205b;
const unsigned32     sec_priv_s_not_member_any_group  = 0x1712205c;
const unsigned32     sec_priv_s_invalid_authn_svc     = 0x1712205e;
const unsigned32     sec_priv_s_invalid_authz_svc     = 0x1712205f;
const unsigned32     sec_priv_s_invalid_trust_path    = 0x17122060;
const unsigned32     sec_priv_s_invalid_request       = 0x17122061;
const unsigned32     sec_priv_s_deleg_not_enabled     = 0x17122065;
const unsigned32     sec_priv_s_intercell_deleg_req   = 0x17122069;
const unsigned32     sec_rgy_not_implemented          = 0x17122072;
const unsigned32     sec_rgy_server_unavailable       = 0x1712207b;
```

5.1.5 Status Code Origination

The status codes in the preceedng section can possibily be set by the following functions. Note this is a probable source of the code, and as such does not restrict a specific code to that particular function in a future revision of DCE.

Status Code	Possible Origin
sec_priv_s_server_unavailable	ps_request_get_ptgt()
sec_priv_s_invalid_principal	ps_request_get_ptgt()
sec_priv_s_not_member_any_group	ps_request_ptgt()
sec_priv_s_invalid_authn_svc	ps_request_ptgt() ps_request_become_delegate() ps_request_become_impersonator() ps_request_eptgt()
sec_priv_s_invalid_authz_svc	ps_request_ptgt() ps_request_become_delegate() ps_request_become_impersonator() ps_request_eptgt()
sec_priv_s_invalid_trust_path	ps_request_ptgt() ps_request_eptgt()
sec_priv_s_invalid_request	ps_request_ptgt()
sec_priv_s_deleg_not_enabled	ps_request_become_delegate() ps_request_become_impersonator()
sec_priv_s_intercell_deleg_req	ps_request_become_delegate()
sec_rgy_not_implemented	ps_request_become_delegate() ps_request_become_impersonator() ps_request_eptgt()
sec_rgy_server_unavailable	ps_request_ptgt()

Table 5-1 Possible Source of rpriv RPC Interface Status Codes

5.1.6 PTGS Service

PS servers support just one service, which is associated with a pair of request/response messages (for definitions relating to privilege-ticket-granting-tickets, see Section 5.1.7 on page 276):

- **Privilege-ticket-granting Service (PTGS)**

  PTGS Request/**PTGS Response** message pair (that is, **(****request**)bytes[]** is a value of data type **PTGSRequest**, and **(*****response**)bytes[]** is a value of data type **PTGSResponse**). This is the service by which clients acquire privilege-ticket-granting-tickets.

Thus, a PS Request message is a PTGS Request message, and a PS Response message is a PTGS Response message.

5.1.7 Privilege-tickets

Privilege-tickets are the (trusted) information objects that the PS manages (their relationships with *ps_request_**( ) are given later in this chapter).

Privilege-tickets are the ‘‘same’’ as non-privilege-tickets (as specified in Chapter 4 on page 159), with the following three supplements (for details see Section 5.2.7 on page 281):

- The client ‘‘named’’ in a privilege-ticket is always a PS server, never a non-PS client principal.
- The authorisation data in a privilege-ticket carries PAC information (see Section 4.3.8.1 on page 194, Section 5.2.5 on page 280 and Section 5.2.6 on page 281) pertaining to some client. That client is said to be the client *nominated* by the privilege-ticket (unless the PAC information is empty, in which case the nominated client defaults to the named client; that is, to the PS server mentioned in the preceding item).
- The transit path carried by a privilege-ticket is always empty (or absent).

Since privilege-tickets are special kinds of tickets, the terminology and other specifications of Chapter 4 relating to tickets applies with appropriate modification of detail to the privilege-tickets of this chapter — *and this will be understood to be in force by default unless explicitly indicated otherwise*. In particular, the following notation can and will be used without more elaborate high-level explanation than that given here (the low-level details are given in the remainder of this chapter, of course):

- **Privilege-ticket**

  $PTkt_{A,B}$. This denotes a privilege-ticket nominating *A* in cell *X*, naming PS_Y (in cell *Y*; not naming *A*, as was the case with a non-privilege-ticket), and targeted to *B* in cell *Y*. It is either a privilege-ticket-granting-ticket (PTGT) or a privilege-service-ticket, according to whether its targeted server is or is not a KDS server. (Note that since privilege-tickets carry no transit path information, there is no notion of a ‘‘$PTkt_{A,X,Z',\cdots,Z'',Y,B}$’’.)

Note that the privilege-ticket acquired via *ps_request_eptgt*( ) to a PS server PS_Z is always a privilege-ticket-granting-ticket (targeted to the KDS server KDS_Z in the same cell *Z*), never a privilege-ticket targeted to a non-KDS server (those are obtained from KDS servers, not PS servers).

5.2 Data Types

The data type definitions of Chapter 4 remain in effect for this chapter, and in addition the following data types are defined. The data description languages and encodings used are ASN.1/BER/DER and IDL/NDR (see the referenced X/Open DCE RPC Specification for the latter), which can be distinguished from one another by their syntaxes. The representations from Section 5.2.1 through Section 5.2.6 on page 281 are identified by the following interface:

```
[
 uuid(47EAABA3-3000-000-0D00-01DC6C000000)
]
```

5.2.1 Authorisation Identities

Identities suitable for the DCE authorisation architecture are represented by the **sec_id_t** data type, which is defined as follows:

```
typedef struct {
    uuid_t                              uuid;
    [string, ptr] char                  *name;
}                                           sec_id_t;
```

Its semantics are that it indicates the identity of security entities — principals (including KDS principals; that is, ''cell identities''), groups, and so on — for the purposes of the authorisation subsystem. Its fields are the following:

- **uuid**

  The ''definitive'', ''computer-friendly'' identifier, represented as a UUID, of the identity in question. Concerning the *version number* of this UUID, see Section 5.2.1.1 on page 278.

- **name**

  An ''advisory'', ''human-friendly'' string form of the entity's identity. It is useful for some purposes (such as performance efficiency), but in general it is *optional*, in the sense that it is the **uuid** (not **name**) field that is the definitive identifier of the identity in question (for example, **name** may even be NULL, on an implementation-specific basis). (The definitive mapping between UUID identifiers and stringname identifiers is given by the ID Map Facility — see Section 1.13 on page 67 and Chapter 12.)

Note: *Authentication identities* must be carefully distinguished from *authorisation identities* in DCE. The former are represented by stringnames (client name and server name), and their semantics are defined in Chapter 4; the latter are represented by UUIDs, and their semantics are defined in this chapter and in Chapter 7 and Chapter 8. As will be seen in the remainder of this chapter, PAC-based authorisation decisions made by servers in the DCE environment depend *only* on authorisation identities, not authentication identities, because the client's authentication identity is not (securely) transmitted to the server (the ''client name'' authentication identity that *is* transmitted is that of the PS, not the accessing client principal) — hence, in this sense, DCE service requests are sometimes said, by abuse of language, to be ''anonymous'' (with respect to authentication identities). Name-based authorisation decisions, on the other hand, do depend on authentication identity stringnames, not on UUIDs — see Section 5.9 on page 299.

5.2.1.1 Security-version (Version 2) UUIDs

The UUIDs that appear in the **uuid** field of **sec_id_t** must be **security-version** (or ‘‘**version 2**’’) **UUIDs**, in all cases except :

- those identifying an identity containing well known *anonymous* privilege attributes (See Section 1.20.7.1 on page 96 and Section 5.2.14.1 on page 288) , or
- those identifying cells (that is, ‘‘cell principals’’ or KDS principals — principals whose RS name has initial component **krbtgt**.

These *must* always have non-security-version (‘‘version 1’’) UUIDs as specified in Appendix A, Universal Unique Identifier, of the referenced X/Open DCE RPC Specification).

These security-version UUIDs are specified exactly as in Appendix A, except that they have the following special properties and interpretations:

- The **version number** is 2.
- The **clock_seq_low** field (which represents an integer in the range [0, 2^8–1]) is interpreted as a *local domain* (as represented by **sec_rgy_domain_t**; see Section 11.5.1.1 on page 379); that is, an identifier domain meaningful to the local host. (Note that the data type **sec_rgy_domain_t** can potentially hold values outside the range [0, 2^8–1]; however, the only values currently registered are in the range [0, 2], so this type mismatch is not significant.) In the particular case of a POSIX host, the value **sec_rgy_domain_person** is to be interpreted as the ‘‘POSIX UID domain’’, and the value **sec_rgy_domain_group** is to be interpreted as the ‘‘POSIX GID domain’’.
- The **time_low** field (which represents an integer in the range [0, 2^{32}–1]) is interpreted as a **local-ID**; that is, an identifier (within the domain specified by **clock_seq_low**) meaningful to the local host. In the particular case of a POSIX host, when combined with a POSIX UID or POSIX GID domain in the **clock_seq_low** field (above), the **time_low** field represents a POSIX UID or POSIX GID, respectively.

By this embedding of local host IDs in (security-version) UUIDs, local host identity information (privilege attributes) can be derived from UUIDs in an especially straightforward and efficient manner (as opposed to, say, going through an auxiliary ID mapping table, maintained either on the local host or elsewhere). (The embedding of local host IDs is specified above in the particular case of POSIX hosts; the embedding in the case of non-POSIX systems is not currently specified in DCE.)

Concerning the *uniqueness* of security-version UUIDs, see the discussion in Section 1.6 on page 25 of the double-UUID identification scheme. The point of that discussion is that the security architecture of DCE depends upon the uniqueness of security-version UUIDs *only within the context of a cell*; that is, only within the context of the local RS’s (persistent) datastore, and that degree of uniqueness can be guaranteed by the RS itself (namely, the RS maintains state in its datastore, in the sense that it can always check that every UUID it maintains is different from all other UUIDs it maintains). In other words, while security-version UUIDs are (like all UUIDs) specified to be ‘‘globally unique in space and time’’, security is not compromised if they are merely ‘‘locally unique per cell’’. This statement does not relax the requirements on implementations of security-version UUIDs, it is only a comment about the trust structure of DCE, namely, entities (security subjects and objects) need not *trust* that foreign RSs maintain globally unique UUIDs, only that their own local RS maintains locally unique UUIDs.

Note that the manner in which security-version UUIDs are generated is implementation-dependent: no API routine is supported by DCE that generates security-version UUIDs (recall that *uuid_create*( ) specified in the referenced X/Open DCE RPC Specification generates only version 1 UUIDs).

5.2.2 Local and Foreign Authorisation Identities

Foreign identities are represented by the **sec_id_foreign_t**, which is defined as follows:

```
typedef struct {
    sec_id_t                            id;
    sec_id_t                            cell;
}                                           sec_id_foreign_t;
```

Its semantics are that it indicates a ''foreign'' (as opposed to ''local'') entity for authorisation purposes. Here, ''local'' (resp., ''foreign'') are relative (context-dependent) terms, indicating entities registered in the RS of the same (resp., different) cell as some other (specified) entity. Cells themselves, as well as other local identities, are represented by the **sec_id_t** data type; foreign identities are represented by **sec_id_foreign_t**. In any specific application of these notions, the entity relative to which the local/foreign distinction is being made must be unambiguously specified.

The fields of the **sec_id_foreign_t** data type are the following:

- **id**

  The cell-relative identity of the entity being identified.

- **cell**

  The (foreign) cell of the entity being identified.

5.2.3 Groups Associated With a Foreign Cell

Foreign group sets are represented by the **sec_id_foreign_groupset_t**, which is defined as follows:

```
typedef struct {
sec_id_t                            cell;
unsigned16                          count_local_groups;
[size_is(count_local_groups), ptr]
sec_id_t                            *local_groups;
}                                       sec_id_foreign_groupset_t;
```

Its semantics are that it indicates a set of groups that are all associated with the *same* ''foreign'' entity for authorisation purposes. In that sense, the groups are ''local'' to the entity, and are registered in the RS of the same cell. Thus, the groups within this entity are represented by the **sec_id_t** data type.

The fields of the **sec_id_foreign_t** data type are the following:

- **cell**

  The (foreign) cell of the entity being identified.

- **count_local_groups**

  The number of local groups (**local_groups**) associated with this groupset.

- **local_groups**

  The non-primary local group authorisation identities associated with this groupset.

5.2.4 PAC Formats

PAC formats are represented by the **sec_id_pac_format_t** data type, which is defined as follows:

```
typedef enum {
    sec_id_pac_format_v1 /* 0 */
}                                           sec_id_pac_format_t;
```

Its semantics are that it identifies the format of PACs (defined in Section 5.2.5) in use. The currently supported formats are:

- **sec_id_pac_format_v1** = 0

  PAC format version 1.

5.2.5 Privilege Attribute Certificates (PACs)

Privilege attribute certificates are represented by the **sec_id_pac_t** data type, which is defined as follows:

```
typedef struct {
    sec_id_pac_format_t                     pac_format;
    unsigned32                              authenticated;
    sec_id_t                                cell;
    sec_id_t                                principal;
    sec_id_t                                primary_group;
    unsigned16                              count_local_groups;
    unsigned16                              count_foreign_groups;
    [size_is(count_local_groups), ptr]
        sec_id_t                            *local_groups;
    [size_is(count_foreign_groups), ptr]
        sec_id_foreign_t                    *foreign_groups;
}                                               sec_id_pac_t;
```

Its semantics are that it indicates authorisation attributes (or characteristics) which are ‘‘projected’’ (sent in a message) from a client to a server during a service request. Its fields are the following:

- **pac_format**

  The format of this PAC. The only currently supported PAC format is version 1 (**sec_id_pac_format_v1**).

- **authenticated**

  Boolean value, indicating whether (**true**) or not (**false**) this PAC is secure, in the sense of having been authenticated by the DCE TCB (more specifically, the PS in the server’s cell); if not authenticated, the PAC is said to be *unauthenticated* or *asserted* (as in the phrase ‘‘this PAC’s authorisation data is *merely asserted* to be legitimate by the client, but not trustworthily so; that is, not certified to be legitimate by the (server’s) PS’’). Servers may grant unauthenticated accesses if they so desire, depending on their policy (see the UNAUTHENTICATED ACLE in Section 7.1.2 on page 312, and its use in the Common Access Determination Algorithm in Section 8.2 on page 321).

- **cell**

  Identifier of the cell relative to which the principal and group entries of this PAC (below) are ‘‘local’’ or ‘‘foreign’’.

- **principal**

  The principal authorisation identity associated with this PAC.

- **primary_group**

  The primary (local) group authorisation identity associated with this PAC.

- **count_local_groups**

  The number of local groups (**local_groups**) associated with this PAC.

- **count_foreign_groups**

  The number of foreign groups (**foreign_groups**) associated with this PAC.

- **local_groups**

  The non-primary local group authorisation identities associated with this PAC.

- **foreign_groups**

  The foreign group authorisation identities associated with this PAC.

The significance of the principal and group attributes (authorisation identities) in the PAC is that servers use them for access authorisation purposes (see Section 8.2 on page 321).

5.2.6 Pickled PACs

Pickled PACs are represented by the **sec_id_pickled_pac_t** data type, which is defined to be a **sec_id_pac_t** pickle. In the terminology and notation of Section 2.1.7 on page 132, this pickle's type UUID (*H***.pkl_type**) is d9f3bd98-567d-11ca-9ec6-08001e022936, and its body datastream is an NDR-marshalled **sec_id_pac_t**.

As mentioned in Section 4.3.8.1 on page 194, the authorisation data type **authzDataType-PAC** is represented by a **sec_id_pickled_pac_t**.

5.2.7 Privilege-tickets

Privilege-tickets are represented by the **PrivilegeTicket** data type, which is defined as follows:

```
PrivilegeTicket ::= Ticket
```

Its semantics are identical with that of **Ticket**'s, with the following three supplementary semantics:

- The named client in a privilege-ticket is always a privilege server (that is, **tkt-***EncryptPart.tkt-ClientName* is always **dce-ptgt**).
- The authorisation data field (**tkt-***EncryptPart.tkt-AuthzData*) in a privilege-ticket carries an array of authorisation data, say *authzData* (data type **AuthzData**, see Section 4.3.8 on page 194), which contains one and only one element, say *authzData[i]*, whose type (*authzData*[*i*]**.authzData-Type**) is equal to **authzDataType-PAC**; that element's value (*authzData*[*i*]**.authzData-Value**) must be (the underlying **OCTET STRING** of) a pickled PAC, as defined in Section 5.2.6. The client *nominated* by the privilege-ticket is the principal identified by the underlying PAC.

  Notes:

  1. The case of an ''empty PAC'' (that is, the **pickled_data[]** array of the pickled PAC has length zero) is not currently supported, but is reserved for potential future usage. (It may, for example, be useful in supporting ''anonymous clients''.)

2. As seen in Section 5.4.2 on page 293, if the PS in a cell ever issued to a principal $A \neq$ PS in its own cell a PTGS Response containing a PTGT having an empty authorisation data array, a clear breach of security would result. For, such a PTGT would be indistinguishable from a TGT identifying the PS itself as the named client. Therefore, *A* could then use this PTGT in a TGS Request, requesting that arbitrary authorisation data (PAC) of *A*'s choosing be included in the resulting privilege-service-ticket (see Section 4.14.2 on page 245). In this manner, *A* could illicitly impersonate any principal it desired.

- The transit path (**tkt-***EncryptPart.tkt-TransitPath*) is always empty (or absent). In particular, servers targeted by privilege-tickets cannot use the privilege-ticket to distinguish remote service requests (that is, from clients in remote cells) from local service requests (that is, from clients in the server's local cell).

5.2.8 Privilege Authentication Headers

Privilege authentication headers are represented by the **PAuthnHeader** data type, which is defined as follows:

```
PAuthnHeader ::= AuthnHeader
```

Its semantics are identical with that of **AuthnHeader**'s, with the following supplementary semantics:

- The associated ticket (**authnHdr-Tkt**) is a privilege-ticket, not a non-privilege-ticket. Hence, a privilege authentication header supplies forward *authorisation* information (a PAC) — not really *authentication* information (at least, not in the sense of a stringname) — that ''authorises a client *A* to a server *B*''. (Whether or not *B* actually *grants A* its request for service depends upon a conceptually separate *access determination* step.)

(Note that the client named in **authnHdr-***EncryptedAuthnr* (**authnHdr-***EncryptAuthnr.authnr-ClientCell* and **authnHdr-***EncryptAuthnr.authnr-ClientName*) identifies the the PS server named in the privilege-ticket, not the calling client *A*.)

5.2.9 Privilege Reverse-authentication Headers

Privilege reverse-authentication headers are represented by the **PRevAuthnHeader** data type, which is defined as follows:

```
PRevAuthnHeader ::= RevAuthnHeader
```

Its semantics are identical with that of **RevAuthnHeader**'s. In particular, it does (securely) identify the server to the client on the basis of its authentication identity (stringname).

5.2.10 PTGS Requests

PTGS Requests are represented by the **PTGSRequest** data type, which is defined as follows:

```
PTGSRequest ::= TGSRequest
```

Its semantics are that it indicates a request for a privilege-ticket-granting-ticket.

Note: As seen in Section 5.4 on page 292, the semantics of a PTGS Request/Response pair differ somewhat from that of a TGS Request/Response pair, particularly with respect to the interpretation (or lack thereof) of some of the data fields transmitted by the client in the request body (denoted *ptgsReq***.req-Body** in Section 5.4 on page 292), which are simply ignored by the target PS server (this is the case for the options field,

ptgsReq.**req-Body.req-Flags**, and the data fields associated with its flag bits). This situation could be expressed by saying that ''certain information is 'lost' when passing from the authentication environment to the authorisation environment''. An argument could therefore be made that the present PTGS Request format (**PTGSRequest** = **TGSRequest**) is ''overkill'', and that a simpler format, transmitting a smaller amount of data, would be sufficient to effect the required semantics of a PTGS Request (and a similar argument could be made for the format of PTkts). Nevertheless, while such an argument may be reasonable, that isn't what is actually done (partially for historical reasons), as specified in this chapter.

5.2.11 PTGS Responses

PTGS Responses are represented by the **PTGSResponse** data type, which is defined as follows:

```
PTGSResponse ::= TGSResponse
```

Its semantics are that it indicates a returned privilege-ticket-granting-ticket.

5.2.12 PS Errors

There is *no* special ''**PSError** data type'' (analogous to the **KDSError** data type). All errors encountered in *ps_request_*()* are reported via its *status* parameter.

5.2.13 Extended PAC (EPAC) Interface

This is the base set of definitions that extend the PAC in order to support delegation. This extended PAC (EPAC) contains the identity and group membership information present in a DCE 1.0 format PAC. In addition, it contains optional delegation controls, optional and required restrictions, and extended attributes which are certified by means of being sealed in an EPAC by the privilege server (PS).

The following EPAC interface defines the base type definitions of the data supported by each PS server, PS_Z, for DCE Version 1.1 and newer versions.

```
[
    uuid(6a7409ee-d6c0-11cc-8fe9-0800093569b9)
]
```

interface sec_id_epac_base { /* Begin sec_id_epac_base Interface */

5.2.13.1 Optional and Required Restrictions

Optional and required restrictions are represented by the **sec_id_opt_req_t**, which is defined as follows:

```
typedef struct {
    unsigned16                 restriction_len;
    [size_is(restriction_len), ptr]
        byte                   *restrictions;
} sec_id_opt_req_t;
```

5.2.13.2 Entry Types for Delegate and Target Restrictions

The entries for delegate and target restrictions are defined by the **sec_rstr_entry_type_t**, which follows:

```
typedef enum {
    sec_rstr_e_type_user,     /* POSIX 1003.6                    */
                              /* Entry contains a key identifying
                               * a user                          */

    sec_rstr_e_type_group,    /* POSIX 1003.6                    */
                              /* Entry contains a key identifying
                               * a group                         */
    sec_rstr_e_type_foreign_user,
                              /* Entry contains a key identifying
                               * a user and the foreign realm    */
    sec_rstr_e_type_foreign_group,
                              /* Entry contains a key identifying
                               * a group and the foreign realm   */
    sec_rstr_e_type_foreign_other,
                              /* Entry contains a key identifying
                               * a foreign realm.  Any user that
                               * can authenticate to the foreign
                               *  realm will be allowed access.  */
    sec_rstr_e_type_any_other,
                              /* Any user that can authenticate to
                               * any foreign realm will be allowed
                               * access.                         */
    sec_rstr_e_type_no_other
                             /* No other user is allowed access. */
}                         sec_rstr_entry_type_t;
```

5.2.13.3 Delegate and Target Restriction Types

Optional and required restrictions are defined by the **sec_id_restriction_t**, which follows:

```
typedef struct {
    union sec_id_entry_u
    switch (sec_rstr_entry_type_t entry_type) tagged_union {
    case sec_rstr_e_type_any_other:
    case sec_rstr_e_type_no_other:
     /* Just the tag field */;
    case sec_rstr_e_type_user:
    case sec_rstr_e_type_group:
    case sec_rstr_e_type_foreign_other:
      sec_id_t id;
    case sec_rstr_e_type_foreign_user:
    case sec_rstr_e_type_foreign_group:
      sec_id_foreign_t foreign_id;
    }                                        entry_info;
}                              sec_id_restriction_t;
```

5.2.13.4 Set of Delegation and Target Restrictions

The set of delegation or target restrictions is represented by the **sec_id_restriction_set_t** which is defined as follows:

```
typedef struct {
    unsigned16 num_restrictions;
    [ptr, size_is(num_restrictions)]
        sec_id_restriction_t *restrictions;
}                           sec_id_restriction_set_t;
```

5.2.13.5 Delegation Compatibility Modes

Delegation compatibility modes determine which EPAC from the delegation chain, if any, to convert and insert into the V1.0 PAC portion of the Authorization Data field. They are defined by the **sec_id_compatibility_mode_t**, which follows:

```
typedef unsigned16 sec_id_compatibility_mode_t;
const unsigned16 sec_id_compat_mode_none      = 0;
const unsigned16 sec_id_compat_mode_initiator = 1;
const unsigned16 sec_id_compat_mode_caller    = 2;
```

5.2.13.6 Supported Delegation Types

Delegation may take the forms permitted in the **sec_id_delegation_type_t**, which is defined by the selections that follow:

```
typedef unsigned16 sec_id_delegation_type_t;
const unsigned16 sec_id_deleg_type_none          = 0;
const unsigned16 sec_id_deleg_type_traced        = 1;
const unsigned16 sec_id_deleg_type_impersonation = 2;
```

5.2.13.7 Supported Seal Types

The seal for a DCE 1.1 (and newer versions) EPAC is defined by the **sec_id_seal_type_t**, which follows. For delegation tokens, the type is **sec_id_seal_type_md5_des**:

```
typedef unsigned16 sec_id_seal_type_t;
const unsigned16 sec_id_seal_type_none    = 0;
const unsigned16 sec_id_seal_type_md5_des = 1;
const unsigned16 sec_id_seal_type_md5     = 2;
```

5.2.13.8 EPAC Seal

The seal (cryptographic checksum) over the EPAC is defined by the **sec_id_seal_t**, and is performed by the privilege server (PS) when the EPAC is generated. The **sec_id_seal_t** definition follows:

```
typedef struct {
    sec_id_seal_type_t seal_type;
    unsigned16 seal_len;
    [size_is(seal_len),ptr]
        byte *seal_data;
} sec_id_seal_t;
```

5.2.13.9 Privilege Attributes for the EPAC

The **sec_id_pa_t** data type provides for inclusion of privilege attributes in an Extended PAC (EPAC). By including attributes within an EPAC, they may be transmitted securely and automatically along with a principal's identity information. The definition of the **sec_id_pa_t** follows:

```
typedef struct {
    sec_id_t realm;
    sec_id_t principal;
    sec_id_t group;
    unsigned16 num_groups;
    [size_is(num_groups), ptr]
        sec_id_t *groups;
    unsigned16 num_foreign_groupsets;
    [size_is(num_foreign_groupsets), ptr]
        sec_id_foreign_groupset_t *foreign_groupsets;
} sec_id_pa_t;
```

5.2.13.10 Handle for Privilege Attribute Data

The **sec_cred_pa_handle_t** definition which follows, provides a handle for the opaque privilege attribute data. Direct access to an EPAC is not supported. This handle provides an interface to permit applications to abstract the contents of an EPAC.

```
typedef void *sec_cred_pa_handle_t;
```

5.2.13.11 Cursor for Delegate Iteration

The *sec_cred_cursor_t* provides an input or output cursor that can be used to iterate through a set of delegates via the *sec_cred_get_delegate*( ) or *sec_login_cred_get_delegate*( ) calls. It is initialized with a call to *sec_cred_initialize_cursor*( ) or *sec_login_cred_initialize_cursor*( ).

```
typedef void *sec_cred_cursor_t;
```

5.2.13.12 Cursor for Extended Attributee Iteration

sec_cred_attr_cursor_t provides an input or output cursor used to iterate through a set of extended attributes using calls to *sec_cred_get_extended_attributes*( ). It is initialized with a call to *sec_cred_initialize_attr_cursor*( ).

```
typedef void *sec_cred_attr_cursor_t;
```

5.2.13.13 Extended PAC Data

The extended PAC (EPAC) data is defined by the **sec_id_epac_data_t**, which follows:

```
typedef struct {
    sec_id_pa_t                 pa;
    sec_id_compatibility_mode_t compat_mode;
    sec_id_delegation_type_t    deleg_type;
    sec_id_opt_req_t            opt_restrictions;
    sec_id_opt_req_t            req_restrictions;
    unsigned32                  num_attrs;
    [size_is(num_attrs), ptr]
        sec_attr_t              *attrs;
    sec_id_restriction_set_t    deleg_restrictions;
    sec_id_restriction_set_t    target_restrictions;
} sec_id_epac_data_t;
```

This data type extends the notion of identities to include chained identities, and also permits delegation and extensible restrictions to be specified. In addition, arbitrary attributes are included within the EPAC. This permits them to be transmitted securely and automatically along with a principal's identity information.

Arbitrary in this context means they are freeform and do not necessarily have a clear cell boundary.

5.2.13.14 List of seals

This is the definition for a list of seals. It is represented by the **sec_id_seal_set_t**, which follows:

```
typedef struct {
    unsigned32              num_seals;
    [size_is(num_seals), ptr]
        sec_id_seal_t       *seals;
}  sec_id_seal_set_t;

typedef [ptr] sec_id_seal_set_t *sec_id_seal_set_p_t;
```

When traced delegation is in use, there may be a set of Extended PACs (EPACs) to transmit. Since the EPAC set consists of one EPAC for the initiator and one EPAC for *each* delegate involved in the operation, a single seal in the PTGT is no longer sufficient to guarantee the individual integrity of each EPAC as well as the integrity of the specific order of the EPACs. To obtain this integrity, the A_D field portion of a PTGT carries the seal of the ordered list of EPAC seals.

5.2.13.15 Extended PAC (EPAC)

This is the definition of the Extended PAC (EPAC). It is represented by the **sec_id_epac_t**, which follows:

```
typedef struct {
    sec_bytes_t        pickled_epac_data;
    [ptr] sec_id_seal_set_t   *seals;
} sec_id_epac_t;
```

5.2.13.16 Set of Extended PACs (EPACs)

This is the definition of the set of Extended PACs (EPACs). It is represented by the **sec_id_epac_set_t**, which follows:

```
typedef struct {
    unsigned32              num_epacs;
    [size_is(num_epacs), ptr]
        sec_id_epac_t       *epacs;
} sec_id_epac_set_t;
```

```
}           /* End sec_id_epac_base Interface */
```

5.2.14 The sec_cred API for Abstracting EPAC Contents

This API is provided for retrieval of Privilege Attribute information from Extended PACs since direct access to EPACs is not supported in this specification.

```
[ local ]

interface sec_cred {            /* Start of sec_cred interface */
```

5.2.14.1 Anonymous Identity

The following self-defining constants represent the definitions for the Anonymous Cell UUID, Anonymous Principal UUID and Anonymous Group UUID.

```
const char * SEC_ANONYMOUS_PRINC = "fad18d52-ac83-11cc-b72d-0800092784e9";
const char * SEC_ANONYMOUS_GROUP = "fc6ed07a-ac83-11cc-97af-0800092784e9";
const char * SEC_ANONYMOUS_CELL  = "6761d66a-cff2-11cd-ab92-0800097086e0";
```

These UUIDs represent well known anonymous identities — cell, principal and group. These UUIDs ensure that any implementation using them will have the same representations for the anonymous cell, principal and anonymous group UUIDs. These are non-security-version UUIDs (‘‘version 1’’ UUIDs) as specified in Appendix A, Universal Unique Identifier, of the referenced X/Open DCE RPC Specification). They have the following properties:

- The **time_hi_and_version** field contains the version number (1) in the 4 most significant bits.

  The Anonymous Principal UUID’s octet number 6 (starting from 0), being X’11’ has most significant bits 1-3 being (0) and most significant bit 4 being (1), and likewise, the Anonymous Group UUID’s octet number 6, being X’11’ also has most significant bits 1-3 being (0) and most significant bit 4 being (1). The same is true for the Anonymous Cell.

- The variant field consists of a variable number of msbs (most significant bits) of the **clock_seq_hi_and_reserved** field. This field determines the layout of the UUID. For the DCE variant, most significant bits 1 and 2 are defined as being the values (1) and (0), respectively.

  Thus, the Anonymous Principal UUID’s octet number 8 (starting from 0), being X’b7’ has most significant bits 1 and 2 being (1) and (0), respectively; and likewise, the Anonymous Group UUID’s octet number 8, being X’97’ also has most significant bits 1 and 2 being (1) and (0), respectively. Similarly, the Anonymous Cell UUID’s octet number 8, being X’ab’ also has most significant bits 1 and 2 being (1) and (0), respectively.

Section 5.2.14 provides the descriptions for the rest of the interface functions defined in this **sec_cred** interface. Since these functions are provided at the API level, their descriptions will not be repeated here. The functions that comprise this interface are also listed in <REFERENCE

UNDEFINED>(EPAC-Accessor).

```
}                                   /* End of sec_cred interface  */
```

5.2.15 Delegation Token (Version 0) Format

A delegation token consists of an expiration date and a byte stream. See Section 5.2.16 on page 290 for its representation. The first byte of any delegation token byte stream is a version number. The contents of subsequent bytes depends upon the version number.

As of DCE1.1, version 0 is the only valid delegation token version. The content of a version 0 delegation token byte stream is illustrated in the following figure.

0	**1**	**2**	**3**	**4**	**12**	**16**	**20**	**36**	**44-52**
0x0	flags1	flags2	key vers	confounder	crc32 chksum	expiration	md5 digest	conf bytes DES key	conf bytes DES ivec
cleartext				**ciphertext**					
required								**optional**	

Figure 5-1 Version 0 Delegation Token Format

Its fields have the following properties:

- The first 4 fields (0 through 3, inclusive) are in cleartext.
- The remaining fields (4 through 52, inclusive) are in ciphertext.
- The first 8 fields are required; the remaining 2 are optional.
- Field 4 (key version number) is the current version of the Privilege Server key used to encrypt the ciphertext portion of the token bytes.
- The ciphertext portion of a version 0 token consists of a 4 byte expiration timestamp in big-endian byte order followed by a 16 byte MD5 message digest (or checksum).

  In addition, if the low-order bit of byte 1, **flags1**, is set (to 1) the MD5 message digest is followed by an 8 byte DES key and an 8 byte initialization vector that is used for encrypting confidential byes of ERAs in EPACs.

  The ciphertext portion is encrypted using **sec_etype_des_cbc_crc**.

5.2.15.1 Version 0 Token Flags

This is the definition of the confidential bytes for the Version 0 delegation token:

```
const unsigned8 sec_dlg_token_v0_conf_bytes = 0x1;
```

5.2.16 Delegation Token

This is the representation of the **sec_dlg_token_t** data type:

```
typedef struct {
    unsigned32 expiration;
    sec_bytes_t token_bytes;
} sec_dlg_token_t;
```

Its semantics are that it is inserted into the A_D field of a new PTGT by the Privilege Server, PS_Z, to ensure that the proper steps were taken to enable delegation, and that none of the data has been tampered with since that time. Its fields are:

- **expiration**

  A 4 byte expiration timestamp in big-endian byte order, in cleartext.

- **token_bytes**

  A byte stream. The first 4 bytes are in cleartext and the rest are in ciphertext. The cyphertext portion of a version 0 token consists of a 4 byte timestamp in big-endian byte order followed by a 16 byte MD5 digest or checksum. In addition, if hte low-order bit of byte1 (**glags1** field) is set (to 1), the MD5 message digest is followed by an 8 byte DES key and an 8 byte initialization vector that is used for encrypting confidential bytes of ERAs (extended registry attributes) in EPACs.

  The ciphertext portion is encrypted using **sec_etype_des_cbc_crc** defined in Section 11.6.1.18 on page 399.

  In order that delegation tokens not be valid forever, the expiration timestamp is part of the encrypted data (The expiration timestamp is also provided in the clear for use by clients for checking for expired tokens).

5.2.17 Delegation Token Set

This is the representation of the **sec_dlg_token_set_t** data type:

```
typedef struct {
    unsigned32 num_tokens;
    [size_is(num_tokens), ptr]
        sec_dlg_token_t *tokens;
} sec_dlg_token_set_t;
```

Its semantics are that it represents the set of tokens involved in a request. Its fields are:

- **num_tokens** The number of tokens in the delegation token set.
- **tokens** (Pointer to) the set of delegation tokens.

5.3 RS Information

Every PS_Z requires access to certain (non-volatile) information. Such information is held in the RS_Z datastore, not in the PS_Z itself. Some of this information is, with appropriate modification of detail, the same as that held in RS_Z for the use of KDS_Z. (But note that such information held in the RS datastore for the PS servers, while similar in kind to that held for KDS servers, may be of different values — for example, the KDS may allow renewable tickets, while the PS may not allow renewable privilege-tickets.) Additionally, the following information in the RS_Z datastore, which has no analog for the KDS_Z, is held solely for use by PS_Z:

- *Authorisation (PAC) attributes*

  The information about principals in *Z* (currently, their principal identity UUIDs and group UUIDs) that PS_Z puts in their PACs.

- *PAC vetting rules*

  Information that PS_Z uses to *vet* (or *modulate*, or *temper*) PACs received in the cross-cell authorisation scenario. This depends on cell policy, but typically involves such activities as discarding some privilege attributes disallowed by *Z*'s policies. See Section 5.8 on page 298.

- *Transit path vetting rules*

  Information about the shape(s) of transit paths that are trusted by PS_Z. (This information is used only by PS_Z, not KDS_Z: KDS_Z knows only about the cells it is directly cross-registered with, not about the trust to be placed in transit paths having multiple links in them.)

  The simplest (and most secure) trusted shape model is the *self* model, in which the only transit paths that are trusted are those of length 0; that is, no cross-cell links are trusted. The next most simple (and next most secure) is the *peer-to-peer* model, in which a transit path is trusted if and only if it has length (0 or) 1; that is, only a cell's directly cross-registered peers are trusted (in addition to itself). Another simple (but rather insecure) model is the *universal trust* model, in which every transit path is trusted. A rather sophisticated (and rather secure) model is the *up-over-down* model, which makes use of the natural hierarchical structure of cell names: the trusted transit paths are those which include ancestors (in the namespace sense) of the client's and server's cells, up to a common ancestor or (at most one) non-ancestral peer-to-peer link (see Section 1.7.2 on page 38).

5.4 PTGS Request/Response Processing

This section specifies in detail the processing that occurs during a PTGS Request/Response exchange. That is, this section specifies the issuing of privilege-ticket-granting-tickets. There are three steps involved:

1. A client prepares a PTGS Request and sends it to a PS server.
2. A PS server receives the PTGS Request from a client, processes it, prepares a PTGS Response (success case) or diagnostic information (failure case), and returns that to the client.
3. A client receives a PTGS Response (*status* = **error_status_ok**) or PS error (*status* ≠ **error_status_ok**).

The details of the three steps of the success case are specified next. (For the failure case, see Section 5.7 on page 298.) By specification, it is, with appropriate modification of detail, identical to Section 4.14 on page 240, with the supplements specified in the present section being made.

Throughout the description of PTGS request/response processing, there are two essentially different cases to be considered:

- The client is in the same cell as the PS server. In this case, the client sends the PS server a PAC containing the authorisation data it wants to be included in the privilege-ticket-granting-ticket, and the PS server checks this requested data against the authorisation information registered for the client in the RS datastore before it issues the client a privilege-ticket-granting-ticket with the requested authorisation data (minus that not registered for the client) in it (informing the client at the same time of the contents of the issued PAC).
- The client is in a different cell from the PS server. In this case, the client sends the PS server a PAC, and the PS server vets the transit path taken by the client's request (that is, evaluates the transit path for a ''trusted shape''), and vets (modulates, tempers) the PAC itself for use in its cell.

5.4.1 Client Sends PTGS Request

Consider a client *A* in cell *X* which has in its possession a non-privilege-ticket targeted to the PS in cell *Y* (possibly *X* = *Y*), $\text{Tkt}_{A,X,\cdots,Y,\text{PS}Y}$ (containing session key $\text{K}_{A,\text{PS}Y}$, of encryption type *encType*). (This $\text{Tkt}_{A,X,\cdots,Y,\text{PS}Y}$ must not be a $\text{PTkt}_{A,\text{PS}Y}$ — PS_Y must reject such a PTGS Request.) And suppose *A* wants to present this $\text{Tkt}_{A,X,\cdots,Y,\text{PS}Y}$ to PS_Y, and receive in return a privilege-ticket-granting-ticket ''based on'' $\text{Tkt}_{A,X,\cdots,Y,\text{PS}Y}$, $\text{PTkt}_{A,\text{KDS}Y}$, which nominates *A*, names PS_Y, and targets KDS_Y. That is, *A* wants to send to PS_Y a PTGS Request, *ptgsReq* (a value of the data type **PTGSRequest**), containing $\text{Tkt}_{A,X,\cdots,Y,\text{PS}Y}$ (*ahTkt*, in its *ptgsReq***.req-AuthnData** field as the *authnHdr***.authnHdr-Tkt** field of an authentication header *authnHdr*), and receive in response a PTGS Response, *ptgsResp* (a value of data type **PTGSResponse**) containing $\text{PTkt}_{A,\text{KDS}Y}$ (*ptgsTkt*, in its *ptgsResp***.resp-Tkt** field). *A* prepares *ptgsReq* in the same way as a **TGSRequest** message (see Section 4.14.1 on page 240) with the supplements indicated below, and ''sends it'' (that is, calls *ps_request_**( )) to PS_Y.

- *Server cell*

  The server cell (*ptgsReq***.req-Body.req-ServerCell**) is set to the cell name of the target server PS_Y; that is, it is set to *Y*'s cell name.

- *Server name*

  The server name (*ptgsReq***.req-Body.req-ServerName**) is set to the RS name of the target server PS_Y; that is, it is set to **dce-ptgt**.

Note: As discussed in Chapter 1, the DCE RPC programming model handles communications with the PS in runtime code (just as it deals with, for example, cross-cell referrals), so that the application programmer does not have to deal with it directly.

- *Authorisation data*

  If $X \neq Y$, the authorisation data field (*ptgsReq***.req-Body.req-***EncryptAuthzData*) is omitted. (If it is not omitted, it will not result in an error, it will simply be ignored by PS_Y, as seen in Section 5.4.2.)

  If $X = Y$, the authorisation data field is used to indicate the (maximum) rights that *A* wants $PTkt_{A,KDSY}$ to convey (provided *A* is registered in RS_X to have these rights, and that PS_Y allows them be used in *Y*). That is, this field is used to implement the concept of *least privilege* (that is, the idea that clients should be accorded only the least set of rights necessary for them to get their job done, thereby preventing the potential misuse of ''excessive'' rights). Namely, if *A* desires that $PTkt_{A,KDSY}$ contain the authorisation information indicated by a PAC *pac* (a value of data type **sec_id_pac_t**), *A* pickles *pac* to produce a pickled PAC *ppac* (a value of data type **sec_id_pickled_pac_t**), then creates *authzData* (a value of data type **AuthzData**, whose **authzData-Type** has value **authzDataType-PAC** and whose **authzData-Value** is (the underlying octet string of) *ppac*), then *authzData* is encrypted using the conversation key $\hat{K}_{A,PSY}$ (*authnr***.authnr-ConversationKey**, of encryption type *encType*, see below) if present, otherwise using the session key $K_{A,PSY}$ in $Tkt_{A,X,\cdots,Y,PSY}$, and the authorisation data field (*ptgsReq***.req-Body.req-***EncryptedAuthzData*) is then set to the resulting encrypted value.

  If the authorisation data field is omitted (in the case $X = Y$), then PS_Y rejects the request.

  Note: A client *A* that wants to receive a PAC entitling it to the *absolute maximum* rights it is entitled to, can do so by first querying RS_X to determine what its maximum rights are — see *rs_acct_get_projlist*( ), Section 11.6.8 on page 407.)

- *Options*

  All options (that is, all flag bits of *ptgsReq***.req-Body.req-Flags**) are unselected (and all data connected with them, namely *ptgsReq***.req-Body.req-StartTime**, *ptgsReq***.req-Body.req-Max***ExpireTime*, *ptgsReq***.req-Body.req-ClientAddrs** and *ptgsReq***.req-Body.req-AdditionalTkts**, are omitted). (Or, if options are selected and/or their data included, they will be ignored by PS_Y, as seen in Section 5.4.2.)

- *Expiration time*

  The expiration time (*ptgsReq***.req-Body.req-***ExpireTime*) is set to *ahTkt***.tkt-***EncryptPart.tkt-ExpireTime*. (If set to any other value, that value will be ignored by PS_Y, as seen in Section 5.4.2.)

5.4.2 PS Server Receives PTGS Request and Sends PTGS Response

Consider a PTGS Request, *ptgsReq*, received by PS_Y from *A*. Thus, *ptgsReq* is a value of data type **PTGSRequest**. Then PS_Y behaves the same way that KDS_Y behaves when it receives a **TGSRequest** message (see Section 4.14.2 on page 245), with the supplements indicated below. (Recall also the description in Section 1.6 on page 25 of the PS_Y's implementation-dependent use of KDS_Y's long-term key, K_{KDSY}: if PS_Y does not have knowledge of K_{KDSY}, it may need to send a TGS Request to KDS_Y in order to get $PTkt_{A,KDSY}$ protected by K_{KDSY}.) In the success case, PS_Y returns a **PTGSResponse** to *A*; in the failure case it returns an error diagnostic (*status* ≠ **error_status_ok**).

- *Client name*

  In the case $X \neq Y$, PS_Y checks that the client name from $Tkt_{A,X,\cdots,Y,PSY}$ is that of a PS server; that is, is **dce-ptgt**.

- *Authorisation data*

  PS_Y checks the authorisation data requested by *A* (*ptgsReq***.req-Body.req-***EncryptAuthzData*) against the maximum *A* is entitled to (if $X = Y$, this information comes from RS_X; if $X \neq Y$, it comes from $Tkt_{A,X,\cdots,Y,PSY}$'s authorisation data field, **tkt-***EncryptPart.tkt-AuthzData*). If $X = Y$, the maximum rights PS_Y will issue to *A* in $PTkt_{A,KDSY}$ (in PACs, in *ptgsTkt***.tkt-***EncryptPart.tkt-AuthzData* and in *ptgsResp***.resp-AuthnData**) is the intersection of these two sets of authorisation data; if $X \neq Y$, the maximum rights PS_Y will issue to *A* is simply $Tkt_{A,X,\cdots,Y,PSY}$'s **tkt-***EncryptPart.tkt-AuthzData* (the intersection mentioned in the case $X = Y$ could conceivably have been used in the case $X \neq Y$ as well, but this isn't currently done — this is for potential future study). However (in either case, $X = Y$ or $X \neq Y$), PS_Y may further restrict *A*'s rights, according to its local vetting (modulating, tempering) policy, thereby issuing to *A* less than the maximum set of rights it would otherwise be entitled to. In any case ($X = Y$ or $X \neq Y$), the minimum rights PS_Y will issue to a principal $A \neq PS_Y$ will be non-empty (that is, if *A* requests rights it is not entitled to, a failure results and PS_Y issues an error). PS_Y also decides, again according to local policy, whether or not it ''vouches'' for the security of this authorisation data, and indicates this by (respectively) setting or resetting the **authenticated** bit of the PAC — this allows servers to grant unauthenticated accesses if they so desire, depending on their policy. (For example, a PS will typically reset the **authenticated** bit if it does not trust the strength of the encryption type used, or if the transit path (below) does not conform to a trusted shape.) The authorisation data issued to *A* is passed back to *A* in two PACs, one in *ptgsTkt***.tkt-***EncryptPart.tkt-AuthzData* and one in *ptgsResp***.resp-AuthnData**, and these two PACs have identical contents but are formatted differently: the former is of data type **AuthzData**, with **authzData-Type** = **authzDataType-PAC** and **authzData-Value** (the underlying **OCTET STRING** of) a pickled PAC; the latter is of data type **AuthnData**, with **authnData-Type** = –2 (see Note below) and **authnData-Value** (the underlying **OCTET STRING** of) the *encryption* of a pickled PAC; that is, a value of data type **EncryptedData**, with **encData-***EncType* = *encType*, an appropriate **encData-***EncKeyVersNum*, and **encData-CipherText** (the underlying **OCTET STRING** of) the encryption of a pickled PAC using the conversation key $\hat{K}_{A,PSY}$ if present, otherwise using the session key $K_{A,PSY}$.

  Note: If the value **authnData-Type** = –2 were being used in a Kerberos authentication protocol, it would be ''unregisterable'' in the sense of Section 4.3.7 on page 193 (because it is negative). However, this value is being used here not in that way, but in an authorisation protocol.

- *Options*

  PS_Y ''ignores'' all options (as well as data connected with them if present); that is, it treats all flag bits of *ptgsReq***.req-Body.req-Flags** as if they were unselected. Thus, PS_Y unselects all flag bits of $PTkt_{A,KDSY}$'s option field (*ptgsTkt***.tkt-***EncryptPart.tkt-Flags*).

- *Expiration time*

  PS_Y sets $PTkt_{A,KDSY}$'s expiration time (*ptgsTkt***.tkt-***EncryptPart.tkt-ExpireTime*) to $Tkt_{A,X,\cdots,Y,PSY}$'s expiration time (*ahTkt***.tkt-***EncryptPart.tkt-ExpireTime*). (Note that PS_Y ignores *ptgsReq***.req-Body.req-***ExpireTime*.)

- *Client addresses*

  PS_Y sets $PTkt_{A,KDSY}$'s client addresses (*ptgsTkt*.**tkt-***EncryptPart.tkt-ClientAddrs*) to $Tkt_{A,X,\cdots,Y,PSY}$'s client addresses (*ahTkt*.**tkt-***EncryptPart.tkt-ClientAddrs*). (Note that PS_Y ignores *ptgsReq*.**req-Body.req-ClientAddrs**.)

- *Transit path*

  PS_Y examines $Tkt_{A,X,\cdots,Y,PSY}$'s transit path field (*ahTkt*.**tkt-***EncryptPart.tkt-TransitPath*), and checks that it is a ''trusted shape'' (which in general depends on PS_Y's policy; in the special case where $X = Y$ or X and Y are cross-registered with one another, the transit path is empty (or absent), and this is always considered to be a trusted shape) — this is also called PS_Y's ''vetting the transit path''. PS_Y sets $PTkt_{A,KDSY}$'s transit path (*ptgsTkt*.**tkt-***EncryptPart.tkt-TransitPath*) to the empty (or absent) path.

5.4.3 Client Receives PTGS Response

Consider a client *A* that receives a PTGS Response, *ptgsResp* (that is, *ptgsResp* is a value of data type **PTGSResponse**), in response to a PTGS Request, *ptgsReq* (as a result of calling *ps_request_**( )) to PS_Y. *A* processes *ptgsResp* in the same way as a TGS Response (see Section 4.14.3 on page 254), with the supplements indicated below. In the success case, *A* is justified in believing that the returned $PTkt_{A,KDSY}$ (or *ptgsTkt*; that is, *ptgsResp*.**resp-Tkt**) is correctly and securely targeted to KDS_Y, and that it contains the values returned elsewhere in the *ptgsResp* (in particular, the authorisation data attributed to *A* (*ptgsResp*.**resp-AuthnData**)), and using it (especially, its session key, which is denoted $\tilde{K}_{A,KDSY}$) in subsequent TGS Requests to KDS_Y for privilege-tickets. In the failure case, *A* takes (application-specific) recovery action.

- *Authentication Data*

  A learns the authorisation information (PAC) that has been issued to it from *ptgsResp*.**resp-AuthnData** (which is encrypted as specified in Section 5.4.2 on page 293).

5.5 Privilege (Reverse-)Authentication Header Processing

This section specifies in detail the processing that occurs during a privilege authentication/reverse-authentication header exchange. There are three steps involved:

1. A client prepares a privilege authentication header and sends it to a server as part of a ''privilege authentication request for (RPC) service'' (for example, this could be a TGS Request for a privilege-ticket, in which case the server is a KDS server). Typically, this privilege authentication header will be merely a part of the whole message sent from client to server, and the rest of the message will contain RPC protocol information and the input parameters for the RPC service request.
2. A server receives a privilege authentication header from a client, processes it, prepares a privilege reverse-authentication header (in the case of a successful client-to-server privilege authentication, and the client has requested the mutual authentication option) or an error message (in the case of a failed client-to-server privilege authentication), and returns that to the client (though some servers may not return errors depending on their policy). Typically, in the success case, the server will proceed to perform the requested service (subject to authorisation constraints that it decides on the basis of the PAC transmitted in the privilege authentication header) and return the output RPC parameters to the client.
3. A client receives a privilege reverse-authentication header (success case, if it had requested mutual authentication in its privilege authentication header) or an error (failure case). Typically, in the success case, it also receives the results of its RPC service request, which it will then decide to accept or reject (on the basis of the privilege reverse-authentication header).

The details of the three steps of the success case are specified next. By specification, it is, with appropriate modification of detail, identical to Section 4.13 on page 231, with the supplements specified in the present section being made.

Note: As noted in Section 4.13 on page 231, the descriptions here are ''typical'' of privilege authentication/reverse-authentication header processing, but since the interpreters (*A* and *B*) of the authentication/reverse-authentication headers are in general application-specific, this whole discussion should be understood to implicitly accommodate some such wording as ''*··· or other such processing as the application requires or allows ···*''.

5.5.1 Client Sends Privilege Authentication Header

Consider a client *A* in cell *X* which has successfully obtained a privilege-ticket to a server in cell *Y* (possibly *X* = *Y*). Thus, *A* is in possession of a privilege-ticket (possibly a privilege-ticket-granting-ticket), *pTkt*, targeted to a server *B* in cell *Y*, whose contents *A* knows, especially its session key $K_{A,B}$ (and its encryption type *encType*), and now *A* wants to use this privilege-ticket to ''privilege-authenticate to'' (that is, engage in protected communications with, and ''project'' (transmit) privilege attributes to) server *B* in cell *Y*. Then *A* prepares a privilege authentication header, *pAuthnHdr* (a value of the **PAuthnHeader** data type) containing *pTkt* (*pAuthnHdr***.authnHdr-Tkt**), and a newly generated authenticator *authnr* (*pAuthnHdr***.authnHdr-***EncryptAuthnr*, of type **Authenticator**), in a way similar to an authenticator (see Section 4.13.1 on page 232) and sends it to *B*, with the supplements indicated below.

- *Client name*

  The client name (*authnr***.authnr-ClientName**) is set to PS_X's RS name (that is, to **dce-ptgt**).

- *Authorisation data*

  The authorisation data field (*authnr***.authnr-AuthzData**) is omitted.

Note: Even though *A* sets *authnr*'s client cell and client name (*authnr***.authnr-ClientCell** and *authnr***.authnr-ClientName**) to *A*'s cell name (that is, *X*'s cell name) and to PS_X's RS name, respectively, these cannot be trusted by the recipient, *B*. The only identities *B* can trust are those carried in the ticket, *pAuthnHdr***.authnHdr-Tkt** (that is, *pTkt*), which are PS_Y's cell name and RS name (in *pTkt***.tkt-***EncryptPart.tkt-ClientCell* and *pTkt***.tkt-***EncryptPart.tkt-ClientName*) and *A*'s PAC (in *pTkt***.tkt-***EncryptPart.tkt-AuthzData*).

5.5.2 Server Receives Privilege Authentication Header and Sends Privilege Reverse-authentication Header

Consider a privilege authentication header, *pAuthnHdr* (a value of the data type **PAuthnHeader**), received by a server *B*, containing a privilege-ticket, *pAhTkt* (*pAuthnHdr***.authnHdr-Tkt**), and an authenticator, *authnr* (*pAuthnHdr***.authnHdr-***EncryptAuthnr*). (For example, KDS servers receive such a privilege authentication header in an entry of the authentication data field of a TGS Request (*tgsReq***.req-AuthnData[***i***].authnData-Value** for some $i \geq 0$) — see Section 4.14.2 on page 245.) Then *B* processes *pAuthnHdr* similarly to the processing of an authenticator header (see Section 4.13.2 on page 234), with the supplements indicated below. In the success case, the privilege authentication header ''authorises the client *A* to the server *B*'' (but it does not ''authenticate'' *A* to *B* in the sense of stringname authentication discussed in Chapter 4, because the stringname of *A* is not necessarily projected to *B*).

- *Client cell*

  The client cell from *pAhTkt* (*pAhTkt***.tkt-***EncryptPart.tkt-ClientCell*) is checked to be *B*'s cell name; that is, *Y*'s cell name.

- *Client name*

  The client name from *pAhTkt* (*pAhTkt***.tkt-***EncryptPart.tkt-ClientName*) is checked to be PS_Y's RS name; that is, **dce-ptgt**.

- *Ticket authorisation data*

  B uses the ticket authorisation data (*pAhTkt***.tkt-***EncryptPart.tkt-AuthzData*); that is, *A*'s PAC information, to make authorisation decisions. Typically — that is, if *B* protects its resources via a Common ACL Manager — this will be done using the DCE Common Access Determination Algorithm.

- *Authenticator authorisation data*

  B ignores the authenticator's authorisation data field (*authnr***.authnr-AuthzData**).

5.5.3 Client Receives Privilege Reverse-authentication Header

Consider a privilege reverse-authentication header, *pRevAuthnHdr* (a value of the data type **PRevAuthnHeader**), received by a client *A*, in response to a privilege authentication header, *pAuthnHdr* (with the mutual authentication option selected), that *A* had earlier sent to a server *B*. Then *A* processes *pRevAuthnHdr* exactly as the processing of a reverse-authentication header (see Section 4.13.3 on page 238), with no supplements at all. In the success case, the privilege reverse-authentication header ''authenticates the server *B* to the client *A*'', exactly as a (non-privilege) reverse-authentication header does (by stringname).

5.6 TGS Request/Response Processing (By KDS)

Section 4.14 on page 240 persists to be completely valid (that is, it doesn't change at all), in the case that a client presents a privilege-ticket-granting-ticket (as opposed to an ordinary ticket-granting-ticket) to the KDS server to which it is targeted, thereby requesting the KDS to issue a privilege-ticket. Namely, the TGS Subservice of the KDS is ''blissfully unaware'' of the existence of the PS. The KDS simply issues tickets exactly as described in Chapter 4 (especially, the ''blind copying'' of authorisation data), and those tickets then ''just happen'' to be privilege-tickets by virtue of the very definition of privilege-tickets (namely, their named client is a PS server, they carry the nominated client's PAC authorisation data, and their transit path is empty). Therefore, no supplements at all to TGS request/response processing need be specified here to support ''TGS request/response processing (by the KDS)''; that is, to support the issuing of privilege-tickets by the KDS.

5.7 PS Error Processing

This section specifies in detail the processing that occurs when a PS server encounters a failure during its processing of a PS Request, and returns a PS error to the requesting client.

Consider a PS Request, *psReq*, received by PS_Y from a client *A*. PS_Y performs the algorithm specified in Section 5.4 on page 292, and if it encounters one or more algorithmic failures, it chooses one to return in the *status* parameter of *ps_request_**( ) (recall, there is no special **PSError** data type, or ''PS Error message'').

5.8 Cross-cell Authorisation — Vetting the Privilege-ticket-granting-ticket

As seen in Section 5.4 on page 292, PS_Y's processing of the PAC of a client *A* in cell *X* is quite straightforward if $X = Y$. But the case $X \neq Y$ requires that PS_Y *vet* the privilege-ticket-granting-ticket in the following two senses:

- *Vetting (or evaluating) the transit path*

  PS_Y examines the transit path of the incoming service-ticket (in the *ps_request_**( )) to verify that it conforms to a ''trusted shape'' (depending on *Y*'s policies).

- *Vetting (or modulating, tempering) the PAC*

  PS_Y removes from the incoming PAC (in the incoming service-ticket) any authorisation attributes that are prohibited by *Y*'s policies.

These two activities have already been discussed in context above, and need have no more said about them here.

5.9 Name-based Authorisation

Note: Name-based authorisation is included in DCE for support of legacy applications only, and its use for any other purpose is discouraged.

By *name-based authorisation* is meant authorisation of a client *A* in cell *X* to a server *B* in cell *Y*, on the basis of a non-privilege-ticket $Tkt_{A,X,Z',\cdots,Z'',Y,B}$ instead of a privilege-ticket $PTkt_{A,B}$, by means of the KDS authentication protocols specified in Chapter 4. That is, PS servers are not visited during name-based authorisation. Thus, the only ''authorisation'' information that is projected from *A* to *B* is the transit path (*kdsTkt***.tkt-***EncryptPart.tkt-TransitPath*) and the ''authentication'' stringname of *A* (*kdsTkt***.tkt-***EncryptPart.tkt-ClientCell* and *kdsTkt***.tkt-***EncryptPart.tkt-ClientName* — see Chapter 4).

DCE does not specify a suite of supporting facilities for name-based authorisation as it does for PAC-based authorisation. Therefore, an application (that is, a client/server pair) that chooses to use name-based authorisation must take upon itself the responsibility of dealing with the following issues in an application-dependent way:

- *B* must vet the transit path (using criteria of its own devising).
- *B* must be prepared to grant or deny access solely on the basis of the individual identity of *A* (because that's all that is projected to *B* — no ''name-based group identities'' are projected or supported).
- *B* must support its own facilities providing the functionality of the ACLs defined elsewhere in DCE (such as ''name-based permissions'', ''name-based access control managers'', ''name-based access control editors'' and ''name-based common access determination algorithm'').

All-in-all, while name-based authorisation may be of some use to some legacy applications, it should be avoided in favor of PAC-based authorisation for new applications (and even legacy applications should be migrated to PAC-based authorisation, if that is at all feasible, in order that they may participate seamlessly in the DCE security environment).

Chapter 6

DCE Security Replication and Propagation

The information in this chapter assumes a knowledge of the DCE security model. Refer to Section 1.2 on page 12 for a description of this model. Chapter 11 on page 357, about the interfaces and datatypes used for propagation of changes to replicas, specifically in Section 11.10 on page 439 to (and including) Section 11.16 on page 459, and Section 11.19 on page 464 to and including Section 11.22 on page 481, and finally, Section 11.24 on page 487.

In a given DCE cell many security servers may run. Each one of the security servers manages its own copy of the registry database. A security server and its database are known as a replica. The servers collaborate to keep their copies of the database consistent. Only one of the replicas accepts changes, the master replica. The action of copying a change from one server to another is propagating that change. All of the changes that occur in the master registry database are propagated to all remaining replicas at the granularity of the change. That is, whenever a change is made to the master registry, such as when a new principal is added, that change is then propagated to each replica. The act of propagating changes from the master security server to all other replica security servers is considered replication.

Replication improves the system's reliability and availability. When clients bind to a replica they bind to either a read or update site. The master site is the only update site; all read only sites are called slave replicas. If a slave replica fails to respond to a query the client can then rebind to another replica.

No facility for supporting replication is specified in this document, though implementations would likely provide some sort of service functions for this purpose. Typically they would consist of support for administration and configuration functions for DCE installation.

6.1 Replication Overview

All security servers can answer queries from clients. The master server is the only server that accepts updates from clients. When a client binds to a server it must bind according to the type of access required. There are currently two methods of binding to a registry server. The client can bind to an arbitrary server using *sec_rgy_site_bind()*; this will bind to any available server. The client can also target the master registry for binding with *sec_rgy_site_bind_update()*; this will force binding to the master registry. The master propagates updates it receives from clients to the other security servers, called slaves. The replication scheme is highly available and weakly consistent.

Replication is done only within a cell. That is, within hierarchical cells or cells connected intercell a change within a cell does not force a change or a replica update in any other cell.

When an update arrives at the master, the master server applies the update to its copy of the database. It then adds the update to its propagation queue. The master then persistently tries to deliver the updates on its propagation queue to all replicas. When an update has been delivered to all replicas, the master then removes the update from its propagation queue.

Note: The master server will maintain the update on the propagation queue until each replica server has received the update. If a replica is taken out of service without being properly retired the propagation queue will grow indefinitely. This will not stop the propagations from proceeding on all other slave replicas, however, all entries on the queue will remain until the out of service replica is in service again. This would be potentially damaging to a master replica. As the propagation queue

grows without bounds there are memory considerations that must be taken into account.

6.2 The Master Replica

The master replica is responsible for maintaining and administering the several entities with regard to replication. During operation the master replica maintains the registry database, replica list and propagation queue. In addition, checkpoint entries are made in order to preserve updates, for both the registry and the replica list. No method is specified in this document for such preservation although a typical implementation might take the form of update logs.

6.2.1 Propagation Queue

To maintain the synchronicity between the replicas the master replica maintains a propagation queue. Each entry on the propagation queue needs to be sent to one or more replicas. The entries on the propagation queue remain until all replicas have received them. All entries on the queue are positioned on the queue as they occur, positioned in first-in, first-out order. That is, when an event occurs at the master registry it is put on the queue. During normal propagation events the following list shows the interfaces which perform normal propagation. Each entry made to the propagation queue typically would be checkpointed (or preserved) in some manner (as with the registry and replica list). The technique for doing so is not specified in this document.

The current per-modification propagation interfaces are described in Chapter 11. These interfaces are:

- **rs_prop_acct_*** for propagating registry account information,
- **rs_prop_acl_*** for propagating registry ACL information,
- **rs_prop_attr_*** for propagating registry attributes,
- **rs_prop_attr_schema_*** for propagating registry attribute schemas,
- **rs_prop_pgo_*** for propagating registry PGO items,
- **rs_prop[_*]_plcy_*** for propagating registry policy information,
- **rs_prop_replist_*** for propagating registry replica list.

For more information regarding the individual propagation function calls, please see the appropriate **rs_prop** sections in Chapter 11.

Each time a modification is made to the registry a corresponding entry is made on the propagation queue. In some instances entries are made to the propagation queue and are not propagated out to the replicas immediately. That is, when an entry is added for propagation the interface has a general argument that places the data on the queue with the specific intention of no propagation. A specific instance for the no propagation flag is during change of master. In this specific case, the sequence of events occurring during a change of master, may cause loss of data.

The propagation queue contains the following information.

- **Sequence Number**
  The master registry sequence number of the change. This is the coordinating entry within the update and propagation process. When the replicas are communicating their relative up-to-dateness this number is the determining factor. This number is originally generated on the master replica, it is generated when an update occurs. For each change within the master there is a sequence number generated and assigned to that change.

- **Timestamp**
  The time when the update happened to the master registry.
- **Data**
  The data that is specific to this update. The actual data that was modified in the master.

6.2.2 Replica List

The replica list contains an entry for each security replica in operation. Each entry within the list contains the replica's UUID, name and tower information. The master replica controls the replica list. That is, in order for a replica to be added or removed from the list, the master controls the process.

6.2.2.1 Replica List Entries

Each security server in the cell manages a replica list. Several entries are common to all replica lists. This basic information on all security servers' replica lists gives each replica an idea of the location and status of all the replicas. The entries for the replica list are outlined here; please see Section 11.20.1.1 on page 469 which contains a complete description of replica list data items. The following entries within the replica list are common to all replicas.

- **replica's name**
  The cell relative name of the replica. The replica's name (which is not ''well-known'') falls in the form */.../cell_name/subsys/dce/sec/replica_name*, where *replica_name* represents the name for the replica.
- **replica's UUID**
  The replica's instance UUID.
- **replica's network address(es)**
  The network address(es) for the replica. There may be multiple addresses. See the data types for **rs_replica_item_t** in Section 11.20.1.1 on page 469 and **rs_replica_twr_vec_t** in Section 11.3.1.2 on page 364 for more information.
- **master**
  This flag indicates whether the described replica is currently the master replica.
- **deleted**
  Flag indicating if the replica has been or is marked for deletion.

The server maintains special information to manage propagation of updates to a replica.

- **propagation type**
  Describes the current relative state of the replica. This will determine the method of updates passed from the master replica.
  - **marked for initialization**
    The replica is currently in the process of receiving a database from a replica.
  - **marked for deletion**
    The replica has been scheduled to be deleted.
  - **ready for updates**
    The replica is currently accepting updates.
- **Sequence Number**
  If the replica is ready for updates, this is the sequence number of the last update successfully delivered to the replica. (This implies, of course, that *all* previous sequence numbers have been successfully delivered.) See the **rs_replica_prop_info_t** data type definition in Section

11.20.1.4 on page 471 for information on this entry (and also the next, Time Stamp.)

- **Time Stamp**
  The timestamp of the last update represented by the sequence number.
- **Communications Status**
  The communication status of this particular replica. There are currently three levels of communication status. They are the nominal state *replica_comm_ok*, short term communication interruption *replica_comm_short_failure*, and long term communication interruption *replica_comm_long_failure*.

The data types **rs_replica_item_t** in Section 11.20.1.1 on page 469 , **rs_replica_prop_info_t** in Section 11.20.1.4 on page 471 and **rs_replica_comm_t** in Section 11.20.1.5 on page 471 give details of replica list entries and communication status values.

6.3 Replica Information

Each security replica has a flag that defines its current state. This state is either known or distributed to other replicas. During certain states the replica is incapable of accepting propagation information or providing database information to clients and other replicas. The complete set of replication information that each replica maintains about itself and about the system is:

- **State**
  As defined in the next section.
- **Replica UUID**
  The replica's instance UUID. This UUID may or may not be identical to that of the **Master UUID**. If it is, then this replica is the master replica (otherwise, it is a slave replica).

  Note: The **master** flag in the replica information **rs_replica_item_t** data type would also indicate **TRUE** if this replica is the master replica.
- **Name**
  Its cell-relative name.
- **Sequence Number**
  The sequence number of the last update provided. This is noted on the master and the client replicas to maintain a cross check of the updates. That is, the master (''thinks'' it) has sent a specific number of updates and this number on the client would confirm that number.
- **Timestamp**
  The timestamp of the last update represented by the sequence number.
- **Initialization UUID**
  The UUID of the replica that provided the initialization of the replica's registry.
- **Network Address(es)**
  The security replica's network address(es).
- **Cell's Security UUID**
  This UUID is generated at the initialization of the master registry database when the cell is created. This entry is the same as the cell UUID. It uniquely determines the cell.
- **Master UUID**
  The UUID of the current master replica.
- **Sequence Number**
  The number of the event when this master became the master. This information is only

relevant if this is the master replica. The master keeps the sequence number that was current at the time this replica became the master.

The **rs_replica_info_t** structure in Section 11.19.1.2 on page 464 provides more information.

6.3.1 Replica State

Because of the variety of changes and situations that a replica can be in, the necessity of maintaining state information is critical. When a replica is attempting to communicate with a peer it needs to understand what the current state of that peer is. The concept of replica state provides this. If a new replica is going to request a database from a peer it needs to know whether that particular replica is able to be a provider. The state generally defines a series of events in the life of a replica, from initialization, name changes, slave to master changes or database key changes. The replica state defines the current condition of the replica. The data type definition of the replica states can be found in Section 11.20.1.2 on page 469. There are 13 possible states. The following is the list of the various states.

- **unknown to master**
  The current state of the replica is unknown to the master.
- **uninitialized**
  The replica remains uninitialized while the database is being created. This is generally a temporary state during creation of a replica.
- **initializing**
  The replica is currently being initialized by another replica.
- **in service**
  The replica is currently in service. The replica may either provide information for clients or become the master.
- **renaming**
  This state is in effect during a renaming of the replica.
- **copying database**
  This state is active when the database is in the process of being copied to a new replica or a replica that has requested a new database.
- **in maintenance**
  The replica is in maintenance mode.
- **master key changing**
  The current master key is in the process of being changed.
- **becoming master**
  When a replica receives the request to become a master, this state is active (on the replica that is in state transition from slave to master).
- **becoming slave**
  During the time when a slave replica receives a request to become a master this state is active (on the master that is in state transition from master to slave).
- **duplicate master**
  A replica that thinks it is the master has been informed by a slave that the slave believes a different replica to be the legitimate master.
- **closed**
  A replica has closed its databases and is in the process of exiting.

- **deleted**
  The replica has been deleted from the replica list.

6.4 Slave Replica

The slave replicas maintain both the registry database and replica list in memory and on disk. Each entry is updated when the master replica propagates a change. Each change is applied to the in-memory copy and then pushed to the disk copy. For each update that is propagated from the master replica an entry is made to the update log as well.

The slave replica also maintains a list of all changes that have been made.

6.4.1 Creating a Replica

In DCE 1.1 a replica is created by configuring the host as a DCE client. In addition, a new (''empty'' or skeletal) (security) database is created. Once the new database is created several entries from the current master are required to be cataloged. The database is initialized with the following entries:

- **Cell Security ID**
  The Cell well known security UUID.
- **Replica ID**
  The instance UUID of the replica being created. This UUID is (dynamically) created during the process of creating the replica.
- **Network Towers**
  The binding towers for the replica being created. See the **rs_replica_twr_vec_p_t** data type definition in Section 11.3.1.2 on page 364 for more information.
- **Replica Name**
  The replica name. This name is of data type **rs_replica_item_p_t**. It is supplied by the administrator upon creation of the replica. See Section 11.20.1.1 on page 469 for a more complete description.
- **Sequence Number**
  The master sequence number at the time of creation. This sequence number is of data type **rs_update_seqno_t**.
- **Creator Id**
  The **sec_id_t** of the entity creating the replica. This id is either supplied by the administrator, or it is the UUID and string name of the local cell on which the replica is created if not given by the administrator. Typically (usually) the (registry) Creator Id is supplied by the administrator.
- **Cell Id**
  The **sec_id_t** of the (local) cell. This Cell Id is initially stored in the registry database when the cell is being created. It consists of the cell's UUID and a string name identifying it. The cell name (string name) is retrieved when necessary (for instance when creating a replica). The method of retrieval is not specified in this document.
- **Keyseed**
  The initial keyseed for the database.
- **Master Rep Information**
  The **rs_replica_item_t** of the master replica (see Section 11.20.1.1 on page 469 for more information).

The replica name is then created and verified with the CDS name service. This is done by creating the replica name of the form */.../cell_name/subsys/sec/replica_name*. The name service is then checked to verify that the name is acceptable for use with *rs_ns_entry_validate()*. (See *rs_ns_entry_validate*( ) on page 810 for information about this routine.) The process of creating a database notifies the master replica to add the new replica to the master's replica list. The master is notified via the *rs_replist_add_replica()* operation. The state of the replica is set to **rs_c_state_uninitialized**. The cell name (Cell Id), Replica Id (instance UUID of the replica) and binding information is then stored in the name space. In conjunction, the name of the master site is also set in the name space. The security service uses this information when contacting the master security server during initialization.

When the master site is notified of a new replica the master server guides the initialization of the replica. When the replica is added to the master's replica list it is marked for initialization using *rs_rep_admin_init_replica()*, which sets the replica state to **rs_c_replica_prop_init**. The master sends an initialization request to the replica using *rs_rep_mgr_init()*. This request includes a list of other replicas that the new replica can use to initialize from (see Section 11.21.4 on page 477). The new slave replica selects one of these specified replicas and sends it a request to copy its entire database using *rs_rep_mgr_copy_all()*. The slave replica supplying the database goes into a special copying database state, **rs_c_state_copying_dbase**, during which it will not accept propagations from the master replica. It copies its database to the new replica. When the copy is complete the new replica finishes its registration in the name service, goes into the state, **rs_c_state_in_service**, and notifies the master that it is now initialized, using *rs_rep_mgr_init_done()*. The master marks the new replica as ready for updates on the master replica list and records the sequence number of the last update the replica received.

6.4.2 Delete A Replica

A replica is deleted in DCE 1.1 under command of (initiated by) the Security Administrator. Typically, installations have a set of commands for security administration-however, they are beyond the scope of this document and are not specified here.

When the replica deletion command is given, a delete replica request is sent to the master server using *rs_replist_delete_replica()*. The master marks the replica for deletion by setting replica state to **rs_c_replica_prop_delete** and puts the delete replica update on its propagation queue. The master then propagates the delete replica update to all other replicas sites on its list. These replicas remove the entry for the deleted replica from their respective replica lists. The master then delivers the delete request to the replica being deleted. When the replica server receives the request, it destroys its database and stops running. Upon completion, the master server removes the deleted replica from it's replica list.

Ultimately there is no verification the replica deleted its database and stopped operating. But the other slave replicas and the master replica would refuse communications because it has been marked as deleted.

6.5 Master Change

By issuing the appropriate command (implementation specific items are beyond the scope of this document), the security administrator can change a slave to a master. This change is effected through *rs_rep_admin_change_master*( ) (see Section 11.19.10 on page 467 for a detailed description of this function). This function causes the then current master to change its replica state into **rs_c_state_becoming_slave**. This stops the propagation activity and starts the transfer of the outstanding updates on the propagation queue to the new master. The becoming slave master then sends via *rs_rep_mgr_become_master()* a request to the selected slave to become a master. The new master replica will then request the propagation queue from the old master using the *rs_rep_mgr_copy_propq()*. When the change master (*rs_rep_admin_change_master*( )) function call returns successfully the old master (the replica becoming the slave) writes the new master information to disk and sets its replica state to **rs_c_state_in_service**.

During the entire master change sequence, from initiation to completion, changes to the master registry are not accepted. If a client is attempting to change the master registry at this time and is not successful, the client attempts resubmitting the change a number of times. The number of attempts permitted is determined by the security administrator.

The slave replica selected to become the new master replica will, upon receiving the request to become the master, read the original master's replica list using *rs_replist_read_full*( ). Once the replica list is successfully transferred to the new master a request is made to the original master to send the propagation queue using *rs_rep_mgr_copy_propq*( ). Having successfully received the propagation queue, the new master uses the lowest number on the propagation queue as it's master sequence number, and commits to being the master by writing the new master information to disk, sending updates to clients and accepting updates from clients.

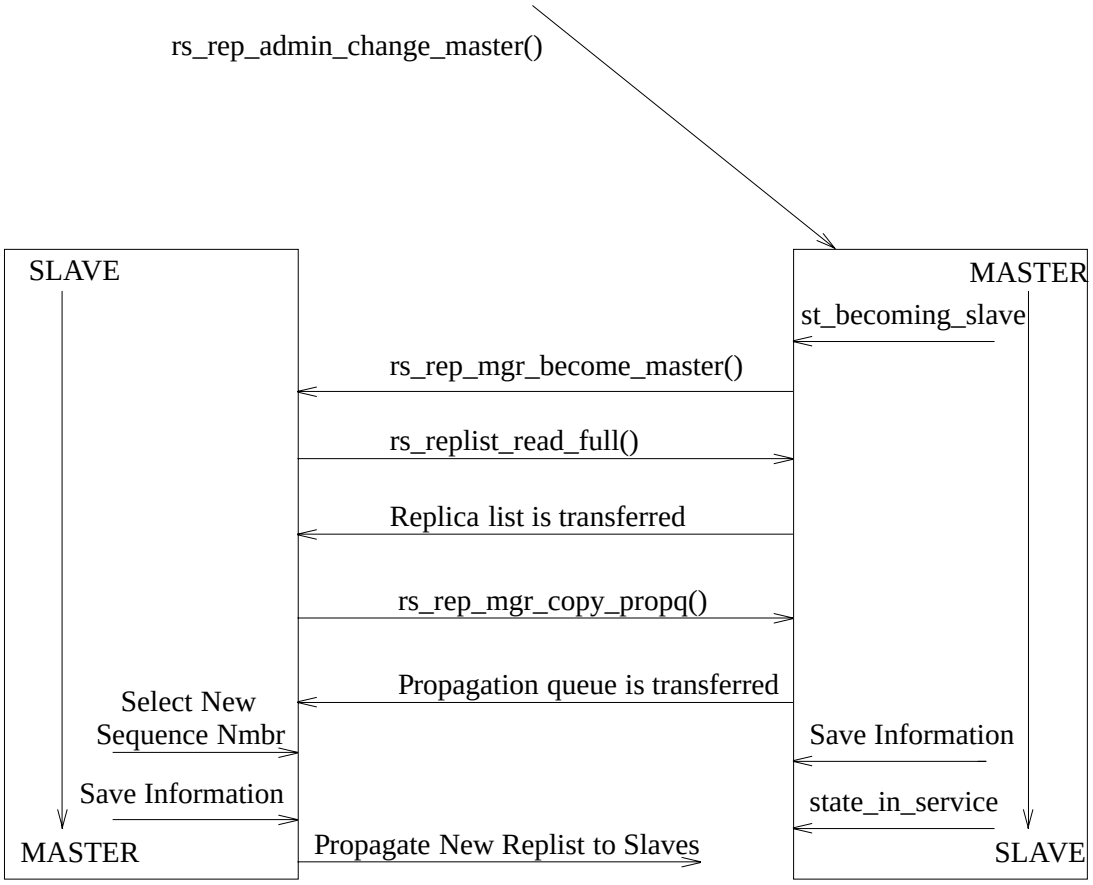


Figure 6-1 Master to Slave Conversion

6.6 Authentication between Replicas

Communication between replicas is secure. The master server authenticates to the slaves as the dce-rgy principal and the slaves authenticate to the master using the host principal of the machine on which they run. By default slaves need **i**, **m** and **I** ACL rights to the replica list (/.:sec/replist) — see Section 11.1 on page 358 for more information on ACL rights. The replica's information and credentials are acquired using the *rs_rep_mgr_get_info_and_creds*( ) function call (see Section 11.21.3 on page 476 for more information).

6.7 Name Service Registration

Each replica has a server entry name in CDS. The default is **/.:/***sec*. When binding to a security server, using this default will cause a binding to the cell's master replica. (An installation (cell) can change this default to any of the security server names registered in CDS. This is typically done via installation-supplied functions that set the default (and which are beyond the scope of this document) to a specific replica in **/.../cell_name/subsys/dce/sec**.)

The **/.../cell_name/subsys/dce/sec** node maps the replica's name to its location, its replica UUID (replica ID), and the cell's security object UUID (Cell Security ID), for any replicas that have been registered. When a replica server is first created it validates its server entry's information.

The security RPC group name is **/.../cell_name/sec**. This name is not ''well-known'', but by convention it is named ''**/.:/sec**'' (see Section 1.18.1.1 on page 86 for more detail on group names (and cell-profiles)). All initialized security server entry names appear in the security group. The cell profile, **/.../cell_name/cell-profile** (a well-known CDS node), maps a few security interface UUIDs to the security group name as follows: (For more information regarding RPC Profiles please reference the DCE RPC Specification. It's complete title can be found in the *Referenced Documents* preface section of this specification.)

```
         UUID                          Vers       Name            Priority  Interface
{{d46113d0-a848-11cb-b863-08001e046aa5 2.0}  /.../cell_name/sec      0     rs_bind}
{{0d7c1e50-113a-11ca-b71f-08001e01dc6c 1.0}  /.../cell_name/sec-v1   0     secidmap}
{{8f73de50-768c-11ca-bffc-08001e039431 1.0}  /.../cell_name/sec      0     krb5rpc}
{{b1e338f8-9533-11c9-a34a-08001e019c1e 1.0}  /.../cell_name/sec      0     ps_request}
{{b1e338f8-9533-11c9-a34a-08001e019c1e 1.1}  /.../cell_name/sec      0     ps_request}
```

In the preceding map, the interface *UUID* and version number (noted as *Vers*) pair together are known as the *Interface Identifier*, and identify the profile (they are the search key for the profile). The *Name* is short for the profile member name, and is the name of the server entry for the interface (specified by *Interface Identifier*). The *Priority* value of zero (0) indicates the highest priority. Also, the *Interface* is the annotation string that textually identifies the cell profile. Note that the **ps_request** annotation string is alternatively known as (the) **rpriv** (interface). (See Section 5.1.1 on page 263 for more information.)

6.7.1 Sample Cell Profile Entries

The CDS name **/.../cell_name/sec-v1** is an RPC Group designating the master security server. For more information regarding RPC Groups please reference the DCE RPC Specification.

6.8 Locate a Security Server

When a client needs to find a security server replica it does so by looking up a special security service interface UUID in the CDS cell profile **/.../cell_name/cell-profile** . This special interface UUID in the cell profile maps to the cell's security group name **/.../cell_name/sec**. The client binding code tries to bind to one of the servers in the security group.

Note: During initialization and configuration of a site, the client cannot locate a security server through the CDS name service as that information is not yet available. For these instances, installation-specific information is used to locate the servers. The handling of such information is not specified in this document.

6.9 Registry Database Encryption

Each replica maintains its own master key to encrypt the data it stores on disk. The key is initially generated via system administrator input. This key can be changed with the routine *rs_rep_admin_mkey*( ).

When a database is initially created by the administrator, as part of the creation process (for both slave and master replicas), the administrator command usually typically requires the specification of a keyseed in order to create the key for the database. In DCE 1.1, if a keyseed is not specified, the administrator is asked to input one as part of the creation process. This keyseed is a character string up to 1024 bytes in length that is then used to seed the random key generator in order to create the master key for the database being created (master or slave). This master key is used to encrypt account passwords. Note that each instance of a replica has its own master key.

Chapter 7

Access Control Lists (ACLs)

This chapter specifies the ACLs supported by DCE. It consists entirely of the static (data) properties of ACLs — the dynamic (programmatic) properties of ACLs are dealt with in Chapter 8. For generalities on ACLs, see Section 1.8 on page 40.

7.1 Data Types

This section defines (in IDL / NDR) the data types associated with ACLs.

7.1.1 Interface UUID for ACLs

The interface UUID for the ACL information specified in this chapter (and also in Chapter 8, is given by the following:

```
[ uuid(47AEE3EA-F000-0000-0D00-01DC6C000000) ]
interface sec_acl_base
```

7.1.2 ACLE Types

ACL entry (ACLE) types are represented by the **sec_acl_entry_type_t** data type, which is defined as follows (comments indicate the values and the ‘‘colloquial’’ names of each type, as used in Section 1.8 on page 40):

```
typedef enum {
    sec_acl_e_type_user_obj,        /*  0 -- USER_OBJ or UO */
    sec_acl_e_type_group_obj,       /*  1 -- GROUP_OBJ or GO */
    sec_acl_e_type_other_obj,       /*  2 -- OTHER_OBJ or O */
    sec_acl_e_type_user,            /*  3 -- USER or U */
    sec_acl_e_type_group,           /*  4 -- GROUP or G */
    sec_acl_e_type_mask_obj,        /*  5 -- MASK_OBJ or M */
    sec_acl_e_type_foreign_user,    /*  6 -- FOREIGN_USER or FU */
    sec_acl_e_type_foreign_group,   /*  7 -- FOREIGN_GROUP or FG */
    sec_acl_e_type_foreign_other,   /*  8 -- FOREIGN_OTHER or FO */
    sec_acl_e_type_unauthenticated, /*  9 -- UNAUTHENTICATED or UN */
    sec_acl_e_type_extended,        /* 10 -- EXTENDED or E */
    sec_acl_e_type_any_other,       /* 11 -- ANY_OTHER or AO */
    sec_acl_e_type_user_obj_deleg,  /* 12 -- USER_OBJ_DEL or UOD */
    sec_acl_e_type_user_deleg,      /* 13 -- USER_DEL or UD */
    sec_acl_e_type_for_user_deleg,  /* 14 -- FOREIGN_USER_DEL or FUD */
    sec_acl_e_type_group_obj_deleg, /* 15 -- GROUP_OBJ_DEL or GOD */
    sec_acl_e_type_group_deleg,     /* 16 -- GROUP_DEL or GD */
    sec_acl_e_type_for_group_deleg, /* 17 -- FOREIGN_GROUP_DEL or FGD */
    sec_acl_e_type_other_obj_deleg, /* 18 -- OTHER_OBJ_DEL or OD */
    sec_acl_e_type_for_other_deleg, /* 19 -- FOREIGN_OTHER_DEL or FOD */
    sec_acl_e_type_any_other_deleg  /* 20 -- ANY_OTHER_DEL or AOD */
}                                         sec_acl_entry_type_t;
```

Its semantics are that it indicates the type of an ACLE (the significance of which is manifested in access determination algorithms) — see Section 7.1.5 on page 313.

7.1.3 ACLE Permission Sets

A permission set; that is, the set of permissions associated to (or ''carried by'') an ACLE, is represented by the **sec_acl_permset_t** data type, which is defined as follows:

```
typedef unsigned32                              sec_acl_permset_t;
```

Its semantics are that the individual bits (called *permission bits*) of a permission set indicate the access rights (up to 32 of them) granted or denied (masked) by an ACLE. The actual *access semantics* (that is, the ''meaning'' in the sense of access control) of these access rights is the responsibility of the ACL manager type associated with the ACL in which the ACLE occurs (see Section 1.9 on page 46 and Chapter 8).

7.1.4 Extended ACLE Information

Extended ACLEs (that is, ACLEs of EXTENDED type) carry information that is represented by the **sec_acl_extend_info_t** data type, which is defined to be a pickle. In the terminology and notation of Section 2.1.7 on page 132, this pickle's type UUID (*H***.pkl_type**) and its body datastream (which is an NDR-marshalled value of an IDL-defined data type) are to be interpreted on an application-specific basis; none are further specified in this revision of DCE. (Some such values may be registered and specified in future revisions of DCE.)

The *rationale* for extended ACLEs is as follows. Future revisions of DCE may add new ACLE types not present in previous revisions. Those new ACLE types are of course unknown to ''old'' ACL clients (such as, for example, ACL editor programs) conforming to the prevision revision. Therefore, new servers supporting the new ACLE types are expected to recognise (for example, via RPC interface version numbers) when an ACL operation (such as *rdacl_lookup*( ) or *rdacl_replace*( )) comes from an old client, and to encode/decode the new ACLE types into/from the EXTENDED ACLE type (which the old client can handle at least sanely, if not intelligently). Thus, at this initial revision of DCE, the EXTENDED ACLE type is supported at specification level, though no servers actually need to encode/decode ACLEs into the EXTENDED type until such time as additional ACLEs are actually defined. ACL clients need to handle the EXTENDED type in order to migrate smoothly into the future, however.

7.1.5 ACLEs

ACLEs are represented by the **sec_acl_entry_t** data type, which is defined as follows:

```
typedef struct {
    sec_acl_permset_t               permset;
    union sec_acl_entry_u
switch (sec_acl_entry_type_t        entry_type) tagged_union {

        case sec_acl_e_type_user_obj:
        case sec_acl_e_type_group_obj:
        case sec_acl_e_type_other_obj:
        case sec_acl_e_type_mask_obj:
        case sec_acl_e_type_unauthenticated:
        case sec_acl_e_type_any_other:
        case sec_acl_e_type_user_obj_deleg:
        case sec_acl_e_type_group_obj_deleg:
        case sec_acl_e_type_other_obj_deleg:
        case sec_acl_e_type_any_other_deleg:
            /*empty*/           /*... just the permset_t... */;
        case sec_acl_e_type_user:
        case sec_acl_e_type_group:
        case sec_acl_e_type_foreign_other:
        case sec_acl_e_type_user_deleg:
        case sec_acl_e_type_group_deleg:
        case sec_acl_e_type_for_other_deleg:
            sec_id_t                local_id;
        case sec_acl_e_type_foreign_user:
        case sec_acl_e_type_foreign_group:
        case sec_acl_e_type_for_user_deleg:
        case sec_acl_e_type_for_group_deleg:
            sec_id_foreign_t        foreign_id;
        case sec_acl_e_type_extended:
            [ptr] sec_acl_extend_info_t
                                    *extended_info;
    }                                   entry_info;
}                                           sec_acl_entry_t;
```

Its semantics are that it indicates one entry of an ACL (see Section 1.8.1 on page 40 for generalities on the concept of ACLEs). Its fields are the following:

- **permset**

  The permission set associated with this ACLE.

- **entry_type**

  The ACLE type of this ACLE.

- **entry_info**

  Additional information associated with this ACLE. The additional information consists of the following, according to this ACLE's type:

  — USER_OBJ, GROUP_OBJ, OTHER_OBJ, MASK_OBJ, UNAUTHENTICATED, ANY_OTHER, USER_OBJ_DEL, GROUP_OBJ_DEL, OTHER_OBJ_DEL, ANY_OTHER_DEL

    No additional information (just the **permset**).

— USER, GROUP, FOREIGN_OTHER, USER_DEL, GROUP_DEL, FOREIGN_OTHER_DEL

A tag (**local_id**), indicating that this ACLE refers to a particular user or group in the ''default cell (of the ACL in which this ACLE occurs)'' (see below), or to a particular ''non-default cell''.

— FOREIGN_USER, FOREIGN_GROUP, FOREIGN_USER_DEL, FOREIGN_GROUP_DEL

A tag (**foreign_id**), indicating that this ACLE refers to a particular user or group in a (specified) ''non-default cell (of the ACL in which this ACLE occurs)''.

— EXTENDED

Extended information (**extended_info**) associated with this ACLE. See Section 7.1.4 on page 313 for details.

7.1.6 ACLs

ACLs are represented by the **sec_acl_t** data type, which is defined as follows:

```
typedef struct {
    sec_id_t                            default_cell;
    uuid_t                              sec_acl_manager_type;
    unsigned32                          count;
    [ptr, size_is(count)]
        sec_acl_entry_t                 *sec_acl_entries;
}                                           sec_acl_t;
```

Its semantics are that it indicates an access control list (see Section 1.8 on page 40 for generalities on the concept of ACLs). Its fields are the following:

- **default_cell**

  The ''default cell'' associated with this ACL (see Section 7.1.5 on page 313 and the Common Access Determination Algorithm in Chapter 8).

- **sec_acl_manager_type**

  The ACL manager type that can interpret this ACL (see Chapter 8).

- **count**

  The number of ACLEs in this ACL.

- **sec_acl_entries**

  The actual ACLEs in this ACL.

7.1.7 ACL Types

ACL types are represented by the **sec_acl_type_t** data type, which is defined as follows:

```
typedef enum {
    sec_acl_type_object,           /* 0 */
    sec_acl_type_default_object,   /* 1 */
    sec_acl_type_default_container /* 2 */
}                                          sec_acl_type_t;
```

Its semantics are that it indicates the ''type'' of ACL associated to an object, as follows.

- **sec_acl_type_object**

Indicates a protection ACL attached to an object (either simple object or container object).

- **sec_acl_type_default_object**

  Indicates a default object creation ACL attached to a container object.

- **sec_acl_type_default_container**

  Indicates a default container creation ACL attached to a container object.

These ACL types are used for inheritance purposes, as specified in Section 1.8.2 on page 44.

7.2 Common ACLs

In principle, a ‘‘legal’’ ACL (in the absolute sense of the generic ACL Facility mechanism itself, as opposed to the relative sense of the specific subset of well-formed ACLs supported by the policies of any specific ACL Manager) can contain any number of ACLEs of any types. But in the case of Common ACL Managers (see Section 1.9 on page 46 and Chapter 8), any ACL managed by a Common ACL Manager is required to satisfy the following conditions. (In the context of Common ACL Managers, these conditions are known as *common ACL formation rules*, and such an ACL is known as a (*well-formed*) *common ACL*.)

- It contains only ACLEs of types specified by **sec_acl_entry_type_t** (see Section 7.1.2 on page 312).
- It contains no EXTENDED ACLEs (see Section 7.1.4 on page 313 for an explanation of EXTENDED ACLEs).
- Its total number of ACLEs is in the range [0, 2^{32}–1].
- It contains at most one USER_OBJ ACLE.
- All its USER ACLEs (if any) refer to principals distinct from one another (though not necessarily distinct from the principal referred to by the USER_OBJ ACLE, if present).
- It contains at most one GROUP_OBJ ACLE.
- All its GROUP ACLEs (if any) refer to groups distinct from one another (though not necessarily distinct from the group referred to by the GROUP_OBJ ACLE, if present).
- It contains at most one OTHER_OBJ ACLE.
- All its FOREIGN_USER ACLEs (if any) refer to principals distinct from one another, and from the principals referred to by the USER ACLEs if present (though not necessarily distinct from the principal referred to by the USER_OBJ ACLE, if present).
- All its FOREIGN_GROUP ACLEs (if any) refer to groups distinct from one another, and from the groups referred to by the GROUP ACLEs if present (though not necessarily distinct from the group referred to by the GROUP_OBJ ACLE, if present).
- All its FOREIGN_OTHER ACLEs (if any) refer to cells distinct from one another, and from the cell referred to by the OTHER_OBJ ACLE if present.
- It contains at most one ANY_OTHER ACLE.
- It contains at most one MASK_OBJ ACLE.
- It contains at most one UNAUTHENTICATED ACLE.

The rules above that forbid ‘‘collisions’’ of ACLEs (that is, those that require ACLEs to be ‘‘distinct from one another’’), are usually summarised by the paraphrase: ‘‘The ACLEs of a (well-formed) common ACL must all be of *different specificity*’’ (with the possible exceptions of USER_OBJ and GROUP_OBJ, depending on whether or not one considers these to be of the ‘‘same specificity’’ as USER/FOREIGN_USER and GROUP/FOREIGN_GROUP, respectively).

Note: The above DCE formation rules should be compared with the following draft-POSIX formation rule (which is present in *some drafts* of the POSIX ACL standard): ‘‘Every ACL must have exactly one each of USER_OBJ, GROUP_OBJ, OTHER_OBJ; and if it has any USER, GROUP or application-defined entries, then it must have exactly one MASK_OBJ entry’’. This rule is not required by DCE. (This represents one of the ways that the ACL model supported by DCE generalises, in ways sanctioned by POSIX, that of the draft-POSIX models.)

Chapter 8

ACL Managers

This chapter specifies the common ACL managers supported by DCE. See Section 1.9 on page 46 for generalities on ACL managers, and for the definition of *Common ACL Managers*.

8.1 Data Types

This section defines (in IDL) the data types associated with ACL Managers.

8.1.1 Common Permissions

There are 7 permission bits, given in the following list, that are distinguished by the ACL Facility, by virtue of their being given specified names (in the C programming language) and values. These 7 distinguished permissions are said to be *common permissions* because of their support by Common ACL Managers (see Section 1.9 on page 46). (The ‘‘colloquial’’ names of these permissions, as used in Section 1.9 on page 46, are given by the terminal substring following the last underscore character of their C names.)

```
const sec_acl_permset_t   sec_acl_perm_read    = 0x00000001;
const sec_acl_perm_set_t  sec_acl_perm_write   = 0x00000002;
const sec_acl_perm_set_t  sec_acl_perm_execute = 0x00000004;
const sec_acl_perm_set_t  sec_acl_perm_control = 0x00000008;
const sec_acl_perm_set_t  sec_acl_perm_insert  = 0x00000010;
const sec_acl_perm_set_t  sec_acl_perm_delete  = 0x00000020;
const sec_acl_perm_set_t  sec_acl_perm_test    = 0x00000040;
```

It is beyond the scope of the generic ACL Facility itself to specify the *access semantics* of these common permissions — that is the responsibility of individual ACL managers themselves. (For their semantics in the case of Common ACL Managers, see Section 1.9 on page 46.)

8.1.2 Printstrings and Helpstrings

The printstring and helpstring associated with a permission bit are represented by the **sec_acl_printstring_t** data type, which is defined as follows:

```
const signed32 sec_acl_printstring_len = 16;
const signed32 sec_acl_printstring_help_len = 64;

typedef struct {
    [string] char         printstring[sec_acl_printstring_len];
    [string] char         helpstring[sec_acl_printstring_help_len];
    sec_acl_permset_t     perm;
}                                sec_acl_printstring_t;
```

Its semantics are that it specifies the printstring and helpstring associated with the permission bit(s) **perm**. Its fields are the following:

- **printstring**

  The printstring associated to **perm**. Its character elements are to be drawn from the alphanumeric characters (**a**–**zA**–**Z0**–**9**) of the Portable Character Set (see Appendix G, Portable Character Set, of the referenced X/Open DCE RPC Specification). Every common

ACL manager is required to associate *distinct* printstrings, of length ≥ 1, with each permission it supports (distinct because typical user interfaces to ACL editors use these printstrings to refer to permissions). (However, it is not required that each printstring consists of a single character, nor that the set of characters present in any one printstring supported by an ACL manager are disjoint from those of any other printstring it supports.)

- **helpstring**

  The helpstring associated to **perm**. It contains a description of the semantics of **perm**. Its character elements are to be drawn from the Portable Character Set (see Appendix G, Portable Character Set, of the referenced X / Open DCE RPC Specification).

- **perm**

  The bit representation of the permission for which this **sec_acl_printstring_t** specifies the printstring and helpstring. It must be a single bit; that is, its value must be a power of 2 (2^k, $0 \le k \le 31$).

The **sec_acl_printstring_t** is also used to describe ACL managers as a whole, not just their individual permission bits (see Section 10.1.9 on page 352).

8.1.2.1 Common Printstrings

There are 7 (single-character) printstrings, given in the following list, that are distinguished by virtue of their being the printstrings associated with the 7 common permission bits of Common ACL Managers (for this reason, they are called *common printstrings*).

- Read: ‘‘**r**’’.
- Write: ‘‘**w**’’.
- Execute: ‘‘**x**’’.
- Control (or Change, or Write-ACL): ‘‘**c**’’.
- Insert: ‘‘**i**’’.
- Delete: ‘‘**d**’’.
- Test: ‘‘**t**’’.

8.1.2.2 Common Helpstrings

There are 7 helpstrings, given in the following list, that are distinguished by virtue of their being the recommended helpstrings associated with the 7 common permission bits of Common ACL Managers (for this reason, they are called *common helpstrings*), at least in the ‘‘C locale’’.

- Read: ‘‘**read**’’.
- Write: ‘‘**write**’’.
- Execute: ‘‘**execute**’’.
- Control (or Change, or Write-ACL): ‘‘**control**’’.
- Insert: ‘‘**insert**’’.
- Delete: ‘‘**delete**’’.
- Test: ‘‘**test**’’.

8.2 Common Access Determination Algorithm

There is one access determination algorithm, specified by pseudocode in this section, that is distinguished by virtue of its being supported by Common ACL Managers (for this reason, it is called the *common access determination algorithm*). See Figure 1-8 on page 50 for a memorisable mental image of this pseudocode. The pseudocode is presented in three steps below. Recall that the ACLs supported by Common ACL Managers satisfy the conditions of Section 7.2 on page 317.

Note: The common access determination algorithm depends only upon:

1. the client's PAC or EPAC

   For DCE 1.1 and newer versions, an EPAC is used to encode the information that used to be provided by the PAC. An EPAC also contains additional attribute information notably that required for delegation support.

   Note that the steps described in this section, unless noted, may still be used for access determination using a PAC.

2. the object's ACL

   For DCE 1.1 and newer versions, new entries have been added to the ACL. These extensions have been added as additional values for the existing **sec_acl_entry_type_t** and defined in Section 7.1.2 on page 312. The key and permission fields of the new ACL entries are defined exactly as they would for other DCE ACLEs. Because of this, users of ACLs who do not enable delegation will continue to operate as before, with no change in behavior.

   Notes:

   1. Because new entries for delegation have been added as new ACLEs, no wire protocol changes are necessary to support these new types.

   2. It is possible to support delegation without using the new ACLE types that have been added to the ACL. While this provides a simple migration path, it has the consequence that every intermediary involved in an operation (request) is granted the ability to perform the operation of their own initiative (assuming their permissions are sufficient). However, that behavior is strongly discouraged by this specification. The new entries for delegation should be used. In this manner, intermediaries can be listed on the ACL without granting the intermediary the ability to operate on the target object directly.

3. the nature of the access request itself (that is, the set of permissions required by the operation requested by the client to be performed on the object).

Significantly, it does not depend on the *name* or *path* that the client uses to specify the object. This is in contradistinction to certain other systems, notably POSIX, whose access semantics support a notion of ''pathname resolution'', whereby a ''search'' (''traverse'') permission is required of intermediate naming nodes in addition to the access permissions of the ultimate target (leaf) object. For this reason, the common access determination algorithm is said to be ''object-based'', as opposed to ''name-based''. (If a name-based access model is required, as, for example, in a POSIX-conformant distributed filesystem, it can of course be implemented within the

context of this specification via a special-purpose (non-common) ACL manager.)

8.2.1 First Step: Reduction

In the first step of the algorithm, the overall determination of access is reduced from the full access request (consisting of a subset of the primitive permissions supported by the Common ACL Manager) to the individual primitive permissions themselves (or ''permission bits'') comprising the access request:

```
/* reduction step -- check each perm bit */
if (for every permission in the (non-empty) access request,
    the matching step of the algorithm (below) grants access) {
        GRANT access;
} else {
        DENY access;
}
```

Note that the second leg of the above pseudocode is entered (resulting in a denial of access) precisely when the matching step of the algorithm (below) denies access for at least one permission in the (non-empty) access request.

8.2.2 Second Step: Matching

In the second step of the algorithm, the determination of access for an individual primitive permission is reduced to a sequence of *attempted matches* against ACLE types (the notion of ''matching'' is *defined* in the subalgorithms themselves). This step is subdivided into two parts:

I. Determination of access for the client, in the non-delegation case, or determination of access for the initiator, in the delegation case, by a sequence of *attempted matches* against ACLE types, below:

```
/* matching step -- match PAC or EPAC against ACLEs, stop at first match */
if (PAC or EPAC  matches ACL's USER_OBJ ACLE) {
        invoke USER_OBJ subalgorithm;
} else if (PAC matches one of ACL's USER or FOREIGN_USER ACLEs) {
        invoke USER's/FOREIGN_USER's subalgorithm;
} else if (PAC matches any of ACL's GROUP_OBJ, GROUP or
           FOREIGN_GROUP ACLEs /*union model here*/) {
        invoke GROUP_OBJ/GROUP's/FOREIGN_GROUP's subalgorithm;
} else if (PAC matches ACL's OTHER_OBJ ACLE) {
        invoke OTHER_OBJ subalgorithm;
} else if (PAC matches one of ACL's FOREIGN_OTHER ACLEs) {
        invoke FOREIGN_OTHER's subalgorithm;
} else if (PAC matches ACL's ANY_OTHER ACLE) {
        invoke ANY_OTHER subalgorithm;
} else {
        DENY access;
}
```

Note: In the non_delegation case, the next substep is not executed. Thus, when delegation is not in effect, the decision for the client is to either GRANT or DENY access at this point.

II. Determination of access for each intermediary in the traced delegation case, by a sequence of *attempted matches* against ACLE types, below:

```
/* matching step -- match EPAC against ACLEs, stop at first match */
if (EPAC  matches ACL's USER_OBJ_DEL ACLE) {
        invoke USER_OBJ_DEL subalgorithm;
} else if (EPAC matches one of ACL's USER_DEL or FOREIGN_USER_DEL ACLEs) {
        invoke USER_DEL's/FOREIGN_USER_DEL's subalgorithm;
} else if (EPAC matches any of ACL's GROUP_OBJ_DEL, GROUP_DEL or
           FOREIGN_GROUP_DEL ACLEs /*union model here*/) {
        invoke GROUP_OBJ_DEL/GROUP_DEL's/FOREIGN_GROUP_DEL's subalgorithm;
} else if (EPAC matches ACL's OTHER_OBJ_DEL ACLE) {
        invoke OTHER_OBJ_DEL subalgorithm;
} else if (EPAC matches one of ACL's FOREIGN_OTHER_DEL ACLEs) {
        invoke FOREIGN_OTHER_DEL's subalgorithm;
} else if (EPAC matches ACL's ANY_OTHER_DEL ACLE) {
        invoke ANY_OTHER_DEL subalgorithm;
} else {
        DENY access;
}
```

Note that the final leg of the pseudocode in the access determination checking in either of the two substeps above is entered (resulting in a denial of access) if and only if the EPAC (or PAC) matches no ACLE of the ACL. This is, in particular, the case if the ACL in question is *empty* (that is, has an empty list of ACLEs). (An object protected by an empty ACL is *inaccessible*, even for modifying its ACL; ACL Managers will typically enforce a minimal, non-empty configuration for their ACLs, so that this can't happen, but DCE does not specify such.)

8.2.2.1 Combined First and Second Steps

Note also that the first and second steps of the algorithm as presented thus far are ''*interchangeable*'', and thus can be combined to give the following *equivalent* algorithm (and this is the form in which the *paraphrase* associated with Figure 1-8 on page 50 was couched):

```
/* combined matching and reduction steps */
/* for client or, for delegation, initiator */
if (PAC matches ACL's USER_OBJ ACLE) {
        if (for every permission in the (non-empty) access request,
            the USER_OBJ subalgorithm grants access) {
                GRANT access;
        } else {
                DENY access;
        }
} else /* ··· similarly for the remaining subalgorithms ··· */
```

This compined set of steps also applies to intermediaries in the case of traced delegation, using the ACLEs for delegation. Since the combined steps are intuitively obvious, they are not explicitely shown here.

8.2.3 Third Step: Subalgorithms

The third step of the algorithm is to invoke the subalgorithm determined by the second step. These subalgorithms determine access for an individual requested permission, and are described below. Throughout the following subalgorithms, to say that a permission is *granted* (resp., *denied*) by an ACLE means that the *bit* in the ACLE's permissions field representing that permission is *set* (resp., *reset*). Also, the following *textual* (''*macro*'') *substitutions* are employed:

```
/* MASK_OBJ ACLE masking */
#define MASK_OBJ-TEST-OK \
        (  (MASK_OBJ ACLE is not present in ACL) \
        || (permission is granted by MASK_OBJ ACLE) )

/* authentication flag test, and UNAUTHENTICATED ACLE masking */
#define AUTHENTICATION-TEST-OK \
        (  (PAC's authentication flag is TRUE) \
        || (  (UNAUTHENTICATED ACLE is present in ACL) \
           && (permission is granted by UNAUTHENTICATED ACLE) ) )
```

Thus note:

- If the MASK_OBJ ACLE is not present in the ACL, the behaviour of the MASK_OBJ-TEST-OK macro is the same ''as if'' it were present and *granted* all permissions.
- If the UNAUTHENTICATED ACLE is not present in the ACL, the behaviour of the AUTHENTICATION-TEST-OK macro is the same ''as if'' it were present and *denied* all permissions.

The subalgorithms are divided into two categories according to the substeps of Section 8.2.2 on page 322. Thus, if either delegation is not enabled, or the authorisation is for the initiator of a request, the non-intermediary subalgorithms in Section 8.2.4 are used. Otherwise, the intermediary subalgorithms in Section 8.2.5 on page 326 are used.

8.2.4 Non-intermediary Subalgorithms

8.2.4.1 USER_OBJ Subalgorithm

This subalgorithm is invoked when the PAC matches the ACL's USER_OBJ ACLE, in the following sense. There is a USER_OBJ ACLE present in the ACL (there can be at most one, by Section 7.2 on page 317), and the principal to which the PAC refers is equal to the principal to which the USER_OBJ refers.

```
/* USER_OBJ subalgorithm */
if ((permission is granted by USER_OBJ ACLE)
&&  AUTHENTICATION-TEST-OK) {
        GRANT access;
} else {
        DENY access;
}
```

8.2.4.2 USER/FOREIGN_USER Subalgorithm

This subalgorithm is invoked when the PAC matches one of the ACL's USER or FOREIGN_USER ACLEs, in the following sense (at most one ACLE can be matched). There are (one or more) USER and/or FOREIGN_USER ACLEs present in the ACL, and the principal to which the PAC refers is equal to the principal to which some USER or FOREIGN_USER ACLE refers.

```
/* USER's/FOREIGN_USER's subalgorithm */
if ((permission is granted by matched USER or FOREIGN_USER ACLE)
&&  MASK_OBJ-TEST-OK
&&  AUTHENTICATION-TEST-OK) {
        GRANT access;
} else {
        DENY access;
}
```

8.2.4.3 GROUP_OBJ/GROUP/FOREIGN_GROUP Subalgorithm

This subalgorithm is invoked when the PAC matches any of the ACL's GROUP_OBJ, GROUP or FOREIGN_GROUP ACLEs, in the following sense (one or more ACLEs can be matched). There are (one or more) GROUP_OBJ, GROUP and/or FOREIGN_GROUP ACLEs present in the ACLE, and some primary group, local secondary group or foreign secondary group to which the PAC refers is equal to some group to which some GROUP_OBJ, GROUP or FOREIGN_GROUP refers.

```
/* GROUP_OBJ/GROUP's/FOREIGN_GROUP's subalgorithm */
if ((permission is granted by (at least one) matched GROUP_OBJ,
     GROUP or FOREIGN_GROUP ACLE)
&&  MASK_OBJ-TEST-OK
&&  AUTHENTICATION-TEST-OK) {
        GRANT access;
} else {
        DENY access;
}
```

8.2.4.4 OTHER_OBJ Subalgorithm

This subalgorithm is invoked when the PAC matches the ACL's OTHER_OBJ ACLE, in the following sense. There is an OTHER_OBJ ACLE present in the ACL (there can be at most one), and the cell to which the PAC refers is equal to the cell to which the ACL refers.

```
/* OTHER_OBJ subalgorithm */
if ((permission is granted by OTHER_OBJ ACLE)
&&  AUTHENTICATION-TEST-OK) {
        GRANT access;
} else {
        DENY access;
}
```

8.2.4.5 FOREIGN_OTHER Subalgorithm

This subalgorithm is invoked when the PAC matches one of the ACL's FOREIGN_OTHER ACLEs, in the following sense (at most one ACLE can be matched). There are (one or more) FOREIGN_OTHER ACLEs present in the ACL, and the cell to which the PAC refers is equal to the cell to which some FOREIGN_OTHER ACLE refers.

```
/* FOREIGN_OTHER's subalgorithm */
if ((permission is granted by matched FOREIGN_OTHER ACLE)
&&  MASK_OBJ-TEST-OK
&&  AUTHENTICATION-TEST-OK) {
        GRANT access;
} else {
        DENY access;
}
```

8.2.4.6 ANY_OTHER Subalgorithm

This subalgorithm is invoked when the PAC matches the ACL's ANY_OTHER ACLE, in the following sense. There is an ANY_OTHER ACLE present in the ACL (there can be at most one), and none of the preceding subalgorithms has been invoked. That is, *every* PAC ''matches'' the ANY_OTHER ACLE (if it is present), including a NULL PAC (which is considered to be ''unauthenticated'').

```
/* ANY_OTHER subalgorithm */
if ((permission is granted by ANY_OTHER ACLE)
&&  MASK_OBJ-TEST-OK
&&  AUTHENTICATION-TEST-OK) {
        GRANT access;
} else {
        DENY access;
}
```

8.2.5 Intermediary Subalgorithms

8.2.5.1 USER_OBJ_DEL Subalgorithm

This subalgorithm is invoked when the EPAC matches the ACL's USER_OBJ_DEL ACLE, in the following sense. There is a USER_OBJ_DEL ACLE present in the ACL (there can be at most one, by POSIX-allowable delegation extensions to Section 7.2 on page 317), and the principal to which the EPAC refers is equal to the principal to which the USER_OBJ_DEL refers.

```
/* USER_OBJ_DEL subalgorithm */
if ((permission is granted by USER_OBJ_DEL ACLE)
&&  AUTHENTICATION-TEST-OK) {
        GRANT access;
} else {
        DENY access;
}
```

8.2.5.2 USER_DEL/FOREIGN_USER_DEL Subalgorithm

This subalgorithm is invoked when the EPAC matches one of the ACL's USER_DEL or FOREIGN_USER_DEL ACLEs, in the following sense (at most one ACLE can be matched). There are (one or more) USER_DEL and/or FOREIGN_USER_DEL ACLEs present in the ACL, and the principal to which the EPAC refers is equal to the principal to which some USER_DEL or FOREIGN_USER_DEL ACLE refers.

```
/* USER_DEL's/FOREIGN_USER_DEL's subalgorithm */
if ((permission is granted by matched USER_DEL or FOREIGN_USER_DEL ACLE)
&&  MASK_OBJ-TEST-OK
&&  AUTHENTICATION-TEST-OK) {
        GRANT access;
} else {
        DENY access;
}
```

8.2.5.3 GROUP_OBJ_DEL/GROUP_DEL/FOREIGN_GROUP_DEL Subalgorithm

This subalgorithm is invoked when the EPAC matches any of the ACL's GROUP_OBJ_DEL, GROUP_DEL or FOREIGN_GROUP_DEL ACLEs, in the following sense (one or more ACLEs can be matched). There are (one or more) GROUP_OBJ_DEL, GROUP_DEL and/or FOREIGN_GROUP_DEL ACLEs present in the ACLE, and some primary group, local secondary group or foreign secondary group to which the EPAC refers is equal to some group to which some GROUP_OBJ_DEL, GROUP_DEL or FOREIGN_GROUP_DEL refers.

```
/* GROUP_OBJ_DEL/GROUP_DEL's/FOREIGN_GROUP_DEL's subalgorithm */
if ((permission is granted by (at least one) matched GROUP_OBJ_DEL,
     GROUP_DEL or FOREIGN_GROUP_DEL ACLE)
&&  MASK_OBJ-TEST-OK
&&  AUTHENTICATION-TEST-OK) {
        GRANT access;
} else {
        DENY access;
}
```

8.2.5.4 OTHER_OBJ_DEL Subalgorithm

This subalgorithm is invoked when the EPAC matches the ACL's OTHER_OBJ_DEL ACLE, in the following sense. There is an OTHER_OBJ_DEL ACLE present in the ACL (there can be at most one), and the cell to which the EPAC refers is equal to the cell to which the ACL refers.

```
/* OTHER_OBJ_DEL subalgorithm */
if ((permission is granted by OTHER_OBJ_DEL ACLE)
&&  AUTHENTICATION-TEST-OK) {
        GRANT access;
} else {
        DENY access;
}
```

8.2.5.5 FOREIGN_OTHER_DEL Subalgorithm

This subalgorithm is invoked when the EPAC matches one of the ACL's FOREIGN_OTHER_DEL ACLEs, in the following sense (at most one ACLE can be matched). There are (one or more) FOREIGN_OTHER_DEL ACLEs present in the ACL, and the cell to which the EPAC refers is equal to the cell to which some FOREIGN_OTHER_DEL ACLE refers.

```
/* FOREIGN_OTHER_DEL's subalgorithm */
if ((permission is granted by matched FOREIGN_OTHER_DEL ACLE)
&&  MASK_OBJ-TEST-OK
&&  AUTHENTICATION-TEST-OK) {
        GRANT access;
} else {
        DENY access;
}
```

8.2.5.6 ANY_OTHER_DEL Subalgorithm

This subalgorithm is invoked when the EPAC matches the ACL's ANY_OTHER_DEL ACLE, in the following sense. There is an ANY_OTHER_DEL ACLE present in the ACL (there can be at most one), and none of the preceding subalgorithms has been invoked. That is, *every* EPAC ''matches'' the ANY_OTHER_DEL ACLE (if it is present), including a NULL EPAC (which is considered to be ''unauthenticated'').

```
/* ANY_OTHER_DEL subalgorithm */
if ((permission is granted by ANY_OTHER_DEL ACLE)
&&  MASK_OBJ-TEST-OK
&&  AUTHENTICATION-TEST-OK) {
        GRANT access;
} else {
        DENY access;
}
```

Chapter 9

Protected RPC

This chapter specifies how the security services specified in the preceding chapters are supported by the DCE RPC facility, thereby presenting a simplified programming model of security services to RPC programmers and securing applications against many passive and active network attacks.

This chapter depends strongly on the referenced X/Open DCE RPC Specification. *The reader of this chapter is assumed to have detailed familiarity with that specification* (especially its Chapters 12 and 13 and Appendix P, including the notation established there), since this chapter does not review the information available there. Also, for the cyclic redundancy checksum $CRC^{\S}_{32}$ used in this chapter, see Section 2.2 on page 136.

The following list specifies all *currently supported* RPC protocols, authentication protocols and authorisation types; this whole chapter is therefore restricted to these only:

- *RPC protocols*
  - — Connectionless (**CL**) RPC protocol.
  - — Connection-oriented (**CO**) RPC protocol.
- *Authentication protocols*
  - — None (**dce_c_rpc_authn_protocol_none**); that is, ‘‘unprotected RPC’’.
  - — Kerberos (**dce_c_rpc_authn_protocol_krb5**); that is, ‘‘protected RPC’’, of various *protection levels* (see Section 1.10 on page 54).
- *Authorisation types*
  - — Name-based (**dce_c_authz_name**).
  - — PAC-based (**dce_c_authz_dce**).

9.1 What is Specified in this Chapter

Recall that all RPC PDUs, as specified in the referenced X/Open DCE RPC Specification (for both the CL and CO protocols), can be regarded as bit-vectors (actually, *byte*-vectors — see below) having a common structure, which in this chapter will be denoted:

PDU = <*H*, *B*, *V*>

where:

- *H* is the PDU’s **header**. It is metadata, describing the actual data carried by the PDU, and is never empty.
- *B* is the PDU’s **body**. It embodies the actual data carried by the PDU, and may be empty. It consists of IDL-defined NDR-marshalled data, generated by a client or server stub or by the RPC runtime itself — possibly encrypted, as specified in this chapter.
- *V* is the PDU’s (*authentication/security*) *verifier*. It represents the security attributes of the PDU, and may be empty.

Each of *H*, *B* and *V* is actually a *byte*-sequence (that is, has *bit*-length a non-negative integral multiple of 8); they are henceforth always regarded as byte-sequences, not bit-sequences. In

particular, the *length* of such a (byte-)vector *M* henceforth in this chapter, will always mean its length *in bytes*, and denoted as:

$\Lambda(M)$

(as opposed to $\lambda(M)$, which denotes length *in bits*; that is, $\lambda(M) = 8 \cdot \Lambda(M)$).

The referenced X/Open DCE RPC Specification also specifies *alignment rules* for PDUs, as part of its definitions of *H*, *B* and *V* (these rules are not repeated here).

The verifier *V* is said to be *present* in a given PDU if it has length > 0. The referenced X/Open DCE RPC Specification specifies the conditions under which *V* is present and *B* (if present) is encrypted (that is, *B* is the ciphertext of a corresponding plaintext **raw body** *R*, which itself generally consists of IDL-defined NDR-marshalled data), namely:

- *CL case*

  V is present if and only if *H***.auth_proto** ≠ **dce_c_rpc_authn_protocol_none**; that is, if and only if (currently) *H***.auth_proto** = **dce_c_authn_level_protocol_krb5**. In that case, *V* is represented by the data type **auth_verifier_cl_t**. Further, *B* is encrypted if and only if *V***.protection_level** = **dce_c_authn_level_privacy**.

- *CO case*

  V is present if and only *H***.auth_length** (= $\Lambda(V)$) is > 0. In that case, *V* is represented by the data type **auth_verifier_co_t**, with *V***.auth_type** ≠ **dce_c_rpc_authn_protocol_none**; that is, with (currently) *V***.auth_type** = **dce_c_rpc_authn_protocol_krb5**. Further, *B* is encrypted if and only if *V***.auth_level** = **dce_c_authn_level_privacy**.

The conditions under which all PDU types are transmitted are completely specified in the referenced X/Open DCE RPC Specification, and there is nothing further to say about that in this chapter. The formats and contents of all PDU types (that is, their headers, bodies and verifiers) are also completely specified in the referenced X/Open DCE RPC Specification, except for certain security-related items — those were explicitly deferred to this specification, and it is the specification of them that forms the contents of this chapter.

Namely, the following is an exhaustive list of the RPC security-related material that was deferred from the referenced X/Open DCE RPC Specification to this specification, and is to be specified in this chapter:

- *Establishment of credentials*

  As specified in the referenced X/Open DCE RPC Specification, credentials (authentication and authorisation information) are established in different ways in the CL and CO cases. Thus, the following need to be specified, in both the **dce_c_authz_name** and **dce_c_authz_dce** cases:

  — *CL case*

    The *in_data* and *out_data* parameters of the *conv_who_are_you_auth*( ) conversation manager operation (these parameters are part of the bodies of the corresponding *conv_who_are_you_auth*( ) request and response PDUs) need to be specified.

  — *CO case*

    The verifier field *V***.auth_value** needs to be specified for **bind**, **bind_ack**, **alter_context** and **alter_context_response** PDUs, provided that *H***.auth_length** > 0.

- *Integrity protection*

  RPC integrity protection is implemented by cryptographic checksums of PDU headers and bodies, carried in PDU verifiers. Thus:

  — *CL case*

  The verifier field *V***.auth_value** needs to be specified when *V***.protection_level** is one of: **dce_c_authn_level_pkt**, **dce_c_authn_level_integrity** or **dce_c_authn_level_privacy**.

  — *CO case*

  The verifier field *V***.auth_value** needs to be specified when *V***.auth_level** is one of: **dce_c_authn_level_pkt**, **dce_c_authn_level_integrity** or **dce_c_authn_level_privacy**.

- *Confidentiality (privacy) protection*

  RPC confidentiality (privacy) protection is implemented by encrypting PDU bodies. Thus:

  — *CL case*

  The body *B* needs to be specified when *V***.protection_level** = **dce_c_authn_level_privacy**.

  — *CO case*

  The body *B* needs to be specified when *V***.auth_level** = **dce_c_authn_level_privacy**.

These will now be specified, first in the CL case (Section 9.2 on page 332), then in the CO case Section 9.3 on page 337).

9.2 Security in the CL RPC Protocol

This section specifies the security-related material listed in Section 9.1 on page 329 for the CL protocol.

9.2.1 CL Establishment of Credentials (Conversation Manager)

See Section 13.3.3, Conversation Manager Encodings, of the referenced X/Open DCE RPC Specification for an explanation of when the conversation manager protocol is invoked. In particular, recall that *conv_who_are_you_auth*( ) is used as an ''(RPC) callback'' that is triggered by an ''original (application-level) RPC''; that is, the invoker of *conv_who_are_you_auth*( ) is actually an application-level *server*, and the invokee is an application-level *client* which is in the process of invoking an original application-level RPC request to an application-level server — that is what triggers the *conv_who_are_you_auth*( ) callback. (Thus, the words ''client'' and ''server'' as used in this section refer to these application-level entities, as opposed to the system-level invoker and invokee of the *conv_who_are_you_auth*( ) callback.)

9.2.1.1 Conversation Manager in_data

The conversation manager *in_data* parameter has as its value the following 12-byte vector:

```
<key_seq_num, challenge>
```

Its components have the following formats and semantics:

- *Key version number*

  The field *in_data***.key_seq_num** is a 4-byte value, big/big-endian representing an encryption key version number (an integer), as specified in Section 4.3.5 on page 187. Its value must be in the range [0, 255], despite the fact that this field could potentially hold values in a larger range. (This is because it is stored elsewhere in the CL RPC protocol in an 8-bit field — see Section 13.3.4, Authentication Verifier Encodings, referenced X/Open DCE RPC Specification.) Its semantics are that it indicates the key version number to be used to ''respond to this challenge''; that is, to construct the *out_data* response — see Section 9.2.1.2.

- *Challenge*

  The field *in_data***.challenge** is an 8-byte value representing a nonce value (see Section 4.3.1 on page 183). Its semantics are that it indicates a nonce to be used by the original RPC server to match this *in_data* request message with its corresponding *out_data* response message (see Section 9.2.1.2).

9.2.1.2 Conversation Manager out_data

The conversation manager *out_data* parameter represents an authentication header (that is, a value of type **AuthnHeader** as specified in Section 4.6 on page 202) or a privilege authentication header (that is, a value of data type **PAuthnHeader** as specified in Section 5.2.8 on page 282), with the supplements indicated below. (Note that the client sends an **AuthnHeader** or **PAuthnHeader** in *out_data* according to its original RPC request specified authorisation type **dce_c_authz_name** or **dce_c_authz_dce**, respectively.) In particular, the field *out_data***.authnHdr-Tkt** carries a ticket that authenticates the client to the server.

- *Options*

  The field *out_data***.authnHdr-Flags** contains no selected options.

- *Conversation key*

  The field *out_data***.authnHdr-EncryptAuthnr.authnr-ConversationKey** is omitted.

  Note: The **key** subfield of the checksum value field (*out_data***.authnHdr-EncryptAuthnr.authnr-Cksum.cksum-Value** — see below) carries a conversation key. Historically, the CL RPC protocol was defined before the conversation key negotiation challenge/response capability was added to the Kerberos RFC 1510 protocol (by means of the conversation keys, ''K^'' and ''K^^'', of the Kerberos authentication header and reverse-authentication header; see Section 4.5 on page 200 and Section 4.7 on page 205).

- *Sequence number*

  The field *out_data***.authnHdr-EncryptAuthnr.authnr-SeqNum** is omitted.

- *Authorisation data*

  The field *out_data***.authnHdr-EncryptAuthnr.authnr-AuthzData** is omitted.

- *Checksum*

  The field *out_data***.authnHdr-EncryptAuthnr.authnr-Cksum.cksum-Type** has the value **chksumType-CL-RPC** (see Section 4.3.4.1 on page 185), which is defined as follows. The field *out_data***.authnHdr-EncryptAuthnr.authnr-Cksum.cksum-Value**, which will be denoted *checksum* here, has as its value the following 40-byte vector:

  ```
  <challenge, protection_level, key_seq_num, key_type,
      key_length, key>
  ```

  (Note that this usage of the **authnr-Cksum** field is quite different from the ''normal'' usage of checksums as discussed in Chapter 2 and in Section 4.3.4 on page 185. The present usage does meet the syntactic definition of the **authnr-Cksum** field, with a ''similar though non-standard'' semantic. This is discussed further at the end of this section.) The components of *checksum* have the following formats and semantics:

  — The field **challenge** is an 8-byte vector, equal to the *in_data***.challenge** field of the corresponding request message. In this manner, **challenge** represents a nonce that the RPC server uses to (securely) match this *out_data* response message with the corresponding *in_data* request message.

  — The field **protection_level** is a 4-byte vector, big/big-endian representing an integer equal to the protection level of the original RPC PDU that this authentication header is authenticating (as specified in the referenced X/Open DCE RPC Specification).

  — The field **key_seq_num** is a 4-byte vector, big/big-endian representing an integer equal to the encryption key version number (in the sense of Section 4.3.5 on page 187) of **key**. It is equal to *in_data***.key_seq_num**.

  — The field **key_type** is a 4-byte vector, big/big-endian representing an integer equal to the encryption key type (in the sense of Section 4.3.3 on page 184) of **key**. The only value currently supported is 1.

  — The field **key_length** is a 4-byte vector, big/big-endian representing an integer equal to Λ(**key**), depending on the value of **key_type**. For **key_type** = 1, **key_length** is 16.

  — The field **key** is a byte vector, representing an encryption key of type indicated by **key_type**. In the case **key_type** = 1, **key** is a 16-byte vector, consisting of two 8-byte subvectors, which are denoted:

```
<des_key, des_iv>
```

Here, **des_key** represents an encryption key of type **encKeyType-DES** (see Section 4.3.3.1 on page 184), and **des_iv** represents a DES initialisation vector (see Section 3.2 on page 148). This key and initialisation vector are used to protect the ensuing session/conversation between the client and server (that is, for integrity and confidentiality protection of the PDU verifiers and bodies as specified in the remainder of this chapter). Consequently, all such keys must be distinct over all <RPC activity, **key_seq_num**> pairs. The activity UUID uniquely identifies a ''shared state'' (in the sense of ''connection'' or ''association'') between client and server. All requests with the same activity UUID must use the same protection level, and all responses must be sent at the same protection level (and with the same key) as the requests that induced them.

This Conversation Manager *in_data*/*out_data* mechanism thus represents yet another way to establish a conversation key between client and server. Extending the notations ''K^'' and ''K^^'' of Section 4.5 on page 200 and Section 4.7 on page 205 (though those do not occur in the CL RPC protocol, as already noted), the key *checksum***.key.des_key** may be denoted ''K^^^''. Like K^, K^^^ is chosen by the client (not the server).

The semantics of *out_data* are that it authenticates (in the sense specified in Section 4.13.1 on page 232 and Section 5.5.1 on page 296) the original RPC request message (the one triggering this invocation of the *conv_who_are_you_auth*( ) callback). Note that *out_data* is bound to the client's original RPC request because that request is protected with *checksum***.key**. Finally, no explicit reverse (privilege) authentication header is generated corresponding to the (privilege) authentication header carried by *out_data* — nevertheless, the conversation between the client and server is indeed implicitly *mutually* authenticated, by virtue of the fact that the key K^^^ is used to protect the communications.

Note: Concerning the above-mentioned non-standard usage of **authnr-Cksum**, note that the usual intention of the **authnr-Cksum** field in the Kerberos protocol is to cryptographically bind the authentication header to the client's message. In the present application of this idea to RPC, as will be detailed in the remainder of this chapter, this binding is done indirectly, in that **authnr-Cksum** is used to cryptographically bind the authentication header to the RPC session/conversation between client and server. The security of this approach relies on the fact that **authnr-Cksum** is protected for both integrity and privacy.

9.2.2 CL Integrity and Confidentiality (PDU Verifiers and Bodies)

Let <*H*, *B*, *V*> be a PDU protected for integrity and/or confidentiality. This section specifies the format and semantics of its body *B* and its (16-byte) verifier field *V***.auth_value**.

Throughout, the following notation is used. Let *authnHdr* denote the authentication header associated with the given PDU — it has been transmitted from the client to the server as the *out_data* parameter of a conversation manager *conv_who_are_you_auth*( ) request (see above).

- *R* denotes raw (plaintext) body data (generally consisting of IDL-defined NDR-marshalled data generated by a client or server stub, or by RPC runtime itself), as specified in the referenced X/Open DCE RPC Specification.
- *K* = *authnHdr***.authnHdr-EncryptAuthnr.authnr-Cksum.cksum-Value.key.des_key**.
- *IV* = *authnHdr***.authnHdr-EncryptAuthnr.authnr-Cksum.cksum-Value.key.des_iv**.

9.2.2.1 CL dce_c_authn_level_pkt

If *V*.**protection_level** = **dce_c_authn_level_pkt**, then the body *B* and verifier field *V*.**auth_value** are specified as follows.

In pseudocode:

```
B = <R, 0(-Λ(R))(mod 8)>;
struct {unsigned32 _1_; unsigned32 _2_;} seq_frag =
    {is_client_pdu?H.seqnum:(H.seqnumˆ2^31), H.fragnum};
enc_seq_frag = DES-CBC(K, IV, seq_frag);
V.auth_value = <enc_seq_frag, 0_8>;
```

In words: The body *B* is set to the raw data *R* (which may be empty) appended with a 0-vector of length (–Λ(*R*))(mod 8), so that *B* has length a multiple of 8. Next, define *seq_frag* to be the IDL-defined NDR-marshalled 8-byte quantity consisting of the 4-byte **direction-bit-adjusted** PDU sequence number *H*.**seqnum** (namely, if this is a server PDU; that is, is transmitted from server to client, then invert (complement) the most significant bit (that is, bitwise-XOR with 0x80000000 = 2^{31})), and the 4-byte PDU fragment number *H*.**fragnum**. (The lack of provision for overflow of *H*.**seqnum** is not considered to be a significant security exposure. Also, note that *H*.**fragnum** is in the range [0, 2^{16}–1], which is a subset of [0, 2^{32}–1], so it can certainly (by ‘‘casting’’) be marshalled as an IDL **unsigned32** — and the size of this range is also not considered to be a significant security exposure.) Let *enc_seq_frag* be the indicated 8-byte DES-CBC encryption of *seq_frag*. Then the 16-byte *V*.**auth_value** is the concatenation of *enc_seq_frag* with an 8-byte 0-vector.

Security interpretation: The PDU’s direction-bit-adjusted sequence number and fragment number (which are carried in the header *H*) are integrity-protected (and bound together) by the verifier *V*. The PDU’s body *B* (*R*) is unprotected.

9.2.2.2 CL dce_c_authn_level_integrity

If *V*.**protection_level** = **dce_c_authn_level_integrity**, then the body *B* and verifier field *V*.**auth_value** are specified as follows.

In pseudocode:

```
B = <R, 0(-Λ(R))(mod 8)>;
hdr_bdy = <H, B>;
cksum_hdr_bdy = MD5(hdr_bdy);
V.auth_value = DES-CBC(K, IV, cksum_hdr_bdy);
```

In words: The body *B* is set to the raw data *R* (it may be empty) appended with a 0-vector of length (–Λ(*R*))(mod 8), so that *B* has length a multiple of 8. Next, define *hdr_bdy* to be the concatenation of the header *H* (recall that Λ(*H*) = 80) and the body *B*. Let *cksum_hdr_bdy* be the 16-byte MD5 checksum of *hdr_bdy*. Then the 16-byte *V*.**auth_value** is the indicated DES-CBC encryption of *cksum_hdr_bdy*.

Security interpretation: The PDU’s header *H* and body *B* (*R*) are integrity-protected (and bound together) by the verifier *V*.

9.2.2.3 CL dce_c_authn_level_privacy

If *V*.**protection_level** = **dce_c_authn_level_privacy**, then the body *B* and verifier field *V*.**auth_value** are specified as follows.

In pseudocode:

```
struct {byte _1_[4]; unsigned32 _2_;} conf_len =
    {RANDOM4( ), Λ(R)};
crc_conf_len = CRC§32(04, conf_len);
enc_conf_len = DES-CBC(K, IV, conf_len);
if (Λ(R) == 0) {
    crc_conf_len_bdy = crc_conf_len;
    B = R; /*that is, B = empty*/
    cksum_conf_len_bdy = 08;
} else {
    R' = <R, 0(-Λ(R))(mod 8)>;
    crc_conf_len_bdy = CRC§32(crc_conf_len, R');
    B = DES-CBC(K, enc_conf_len, R');
    cksum_conf_len_bdy = DES-CBC-CKSUM(K, enc_conf_len, R');
}
crc_conf_len_bdy_hdr = CRC§32(crc_conf_len_bdy, H);
cksum_conf_len_bdy_hdr = DES-CBC-CKSUM(K, cksum_conf_len_bdy, H);
struct {unsigned32 _1_; unsigned32 _2_;} seq_crc_conf_len_bdy_hdr =
    {H.seqnum, crc_conf_len_bdy_hdr};
enc_cksum_seq_crc_conf_len_bdy_hdr =
    DES-CBC(K, cksum_conf_len_bdy_hdr, seq_crc_conf_len_bdy_hdr);
V.auth_value = <enc_conf_len, enc_cksum_seq_crc_conf_len_bdy_hdr>;
```

In words: Set *conf_len* to the IDL-defined NDR-marshalled 8-byte quantity consisting of a 4-byte random vector, and the integer $\Lambda(R)$ (which is ≤ 65528, by Section 12.5.2.15, PDU Body Length, of the referenced X/Open DCE RPC Specification). Let *crc_conf_len* be the 4-byte CRC of *conf_len*, with 4-byte 0-vector as seed. Let *enc_conf_len* be the indicated 8-byte DES-CBC encryption of *conf_len*. If $\Lambda(R) = 0$: let *crc_conf_len_bdy* be the 4-byte *crc_conf_len*; let *B* be *R* (that is, empty); and let *cksum_conf_len_bdy* be the 8-byte 0-vector. If $\Lambda(R) > 0$: let *R´* be the raw data *R* appended with a 0-vector of length $(-\Lambda(R))(\text{mod } 8)$, so that *R´* has length a positive multiple of 8; let *crc_conf_len_bdy* be the 4-byte CRC of *R´* with seed *crc_conf_len*; let *B* be the indicated DES-CBC encryption of *R´*; and let *cksum_conf_len_bdy* be the indicated DES-CBC-CKSUM of *R´*. Let *crc_conf_len_bdy_hdr* be the CRC of the 80-byte header *H* with seed *crc_conf_len_bdy*. Let *cksum_conf_len_bdy_hdr* be the indicated DES-CBC-CKSUM of *H*. Let *seq_crc_conf_len_bdy_hdr* be the IDL-defined NDR-marshalled 8-byte quantity consisting of the 4-byte PDU sequence number *H*.**seqnum** (*not* direction-bit-adjusted, because the header *H* contains the CL packet type (**ptype** field), which itself serves as a ''direction bit''), and the 4-byte *crc_conf_len_hdr_bdy* regarded as integer by the big/big-endian mapping. Let *enc_cksum_seq_crc_conf_len_bdy_hdr* be the indicated 8-byte DES-CBC encryption of *seq_crc_conf_len_bdy_hdr*. Finally, *V*.**auth_value** is the 16-byte concatenation of *enc_conf_len* and *enc_cksum_seq_crc_conf_len_bdy_hdr*.

Security interpretation: The PDU's body *B* (*R*) is confidentiality-protected. The PDU's header *H* and body *B* (*R*) are integrity-protected (and bound together) by the verifier *V*, and by the use of the DES-CBC-CKSUM as the initialisation vector for the DES-CBC encryption of the body.

9.3 Security in the CO RPC Protocol

This section specifies the security-related material listed in Section 9.1 on page 329 for the CO protocol.

Recall that the following quantities are associated with any PDU (see Section 13.2.1, Client Association State Machine, of the referenced X/Open DCE RPC Specification for definitions and details). The formats used for them are specified here:

- *crc_assoc_uuid*

  The PDU's *CRC of association UUID*. As explained in Section 13.2.1, Client Association State Machine, of the referenced X/Open DCE RPC Specification (see also Section 9.3.1.1 on page 338 below), every PDU has an association UUID attached to it, *assoc_uuid*, which is a 16-byte NDR-marshalled value of the IDL **uuid_t** data type. However, this association UUID is known only by the client, not the server — all that is known by the server is *crc_assoc_uuid* (in the referenced X/Open DCE RPC Specification, this was denoted **assoc_uuid_cksum**). It is defined as follows:

  $$\texttt{crc\_assoc\_uuid} = \texttt{CRC}^{\S}_{32}(0_4, \texttt{assoc\_uuid});$$

  In words: *crc_assoc_uuid* is the indicated 4-byte CRC of *assoc_uuid* with 4-byte 0-vector as seed.

  Security interpretation: *crc_assoc_uuid* is a uniformly distributed 32-bit hash of the 128-bit *assoc_uuid* (that is, even though CRCs are not cryptographic hashes, the probability of *crc_assoc_uuid* colliding with another such hash is probablistically insignificant).

  It is viewed (and formatted) as a 4-byte little/big-endian integer (even though only its bit pattern is of significance, never its integral value — that is, the only operation ever performed on it is testing for equality, never arithmetic operations such as addition).

  Note also that (as seen by the remainder of this section) the *uniqueness* of *crc_assoc_uuid*s is relied upon (though not in a trusted way) to distinguish between associations. This in turn relies upon the uniqueness of UUIDs, and in fact on the actual algorithm for generating UUIDs (see Section A.2, Algorithms for Creating a UUID, of the referenced X/Open DCE RPC Specification). The secrecy of *crc_assoc_uuid*s is never relied upon, and collisions of *crc_assoc_uuid*s can at worst result in a denial of service. In particular, *uuid_create*( ) need not be considered part of the local TCB.

- *dir_seq*

  The PDU's *direction-bit-adjusted sequence number*; that is, its sequence number with the most significant bit inverted (complemented) in the case of a server(-to-client) PDU. This is specified in the referenced X/Open DCE RPC Specification (note that it is separately maintained locally on the client and on the server, and is not directly transmitted in the PDU, though it is used in the cryptographic computations below). It is formatted as a 4-byte little/big-endian integer. (The lack of provision for overflow of the sequence number is not considered to be a significant security exposure.)

- *sub_type*

  The PDU's *encryption/checksum subtype identifier*. It is an integer value in the range [0, 2^8–1], formatted into 1 byte via the big-endian mapping. It specifies a combined encryption/checksum mechanism depending on the value of *sub_type*, which is denoted:

  $$\text{ENCCKSUM}_{sub\_type}(K, CRCUUID, DIRSEQ, M)$$

  where the 4 input items are: *K* is an 8-byte vector (in fact, for both of the registered *sub_type*s listed below, *K* must actually be a DES key; that is, must be in odd-parity normal form);

CRCUUID is a 4-byte vector; *DIRSEQ* is a 4-byte vector; and *M* is a byte-vector of length $\Lambda(M) > 0$. The currently registered values for *sub_type*, and the definitions of their corresponding encryption/checksum mechanisms, are the following:

— **dce_c_cn_sub_type_des** = 0

This yields as output an 8-byte encryption/checksum value defined by the following pseudocode:

```
crcuuid_dirseq = <CRCUUID, DIRSEQ>;
M' = <M, 0(-Λ(M))(mod 8)>;
ENCCKSUMdce_c_cn_sub_type_des(K, CRCUUID, DIRSEQ, M) =
    DES-CBC(K, crcuuid_dirseq, M');
```

In words: Set *crcuuid_dirseq* to the 8-byte concatenation of *CRCUUID* and *DIRSEQ*. Let *M´* be *M* padded with a 0-vector of length $(-\Lambda(M))(\text{mod } 8)$, so that $\Lambda(M')$ is a positive multiple of 8. Then ENCCKSUM**dce_c_cn_sub_type_des**(*K*, *CRCUUID*, *DIRSEQ*, *M*) is set to the indicated 8-byte DES-CBC encryption of *M´*.

— **dce_c_cn_sub_type_md5** = 1

This yields as output a 16-byte encryption/checksum value defined by the following pseudocode:

```
crcuuid = <CRCUUID, 04>;
enc_crcuuid = DES-CBC(K, 08, crcuuid);
msg_enc_crcuuid_dirseq = <M, enc_crcuuid, DIRSEQ>;
ENCCKSUMdce_c_cn_sub_type_md5(K, CRCUUID, DIRSEQ, M) =
    MD5(msg_enc_crcuuid_dirseq);
```

In words: Set *crcuuid* to the 8-byte concatenation of the 4-byte CRCUUID and the 4-byte 0-vector. Let *enc_crcuuid* be the indicated 8-byte DES-CBC encryption of *crcuuid*. Next let *msg_enc_crcuuid_dirseq* be the concatenation of *M* (not padded), the 8-byte *enc_crcuuid*, and the 4-byte *DIRSEQ*. Then ENCCKSUM**dce_c_cn_sub_type_md5**(*K*, *CRCUUID*, *DIRSEQ*, *M*) is set to the 16-byte MD5 checksum of *msg_enc_crcuuid_dirseq*.

Security interpretation (in both of the above two cases): *CRCUUID*, *DIRSEQ* and *M* are integrity-protected (and bound together) by ENCCKSUM$_{sub_type}$(*K*, *CRCUUID*, *DIRSEQ*, *M*).

9.3.1 CO Establishment of Credentials (bind, bind_ack, alter_context, alter_context_response)

Let <*H*, *B*, *V*> be a PDU of one of the types **bind**, **bind_ack**, **alter_context** or **alter_context_response** such that *H***.auth_length** > 0. This section specifies the verifier field *V***.auth_value** in these cases.

9.3.1.1 CO Verifier auth_value.assoc_uuid_crc

This section specifies the verifier field *V***.auth_value.assoc_uuid_crc**. It depends on the PDU type, as follows:

- *The case of a* **bind** *or* **alter_context** *PDU*.

  *V***.auth_value.assoc_uuid_crc** is defined as follows:

  ```
  V.auth_value.assoc_uuid_crc = crc_assoc_uuid;
  ```

  In words: The client sets *V***.auth_value.assoc_uuid_crc** to the 4-byte CRC of association UUID, *crc_assoc_uuid*, as defined in Section 9.3 on page 337. (This is how the server learns *crc_assoc_uuid*, but not the actual *assoc_uuid* itself.)

- *The case of a* **bind_ack** *or* **alter_context_response** *PDU*.

  *V***.auth_value.assoc_uuid_crc** is defined as follows:

  ```
  V.auth_value.assoc_uuid_crc = 'unspecified';
  ```

  In words: *V***.auth_value.assoc_uuid_crc** is set to a 4-byte 0-vector by the server, and is ignored by the client.

9.3.1.2 CO Verifier auth_value.checksum

This section specifies the verifier field *V***.auth_value.checksum**. It depends on the value of *V***.auth_level**, as follows:

- *The case V***.auth_level** = **dce_c_authn_level_none** — *V***.auth_value.checksum** is defined by the following pseudocode:

  ```
  V.auth_value.checksum = 'unspecified';
  ```

  In words: *V***.auth_value.checksum** is set by the sender to an 8-byte or 16-byte 0-vector, according as *V***.auth_value.sub_type** is **dce_c_cn_sub_type_des** or **dce_c_cn_sub_type_md5** respectively; its value is ignored by the receiver.

  Security interpretation: The PDU is not protected by its verifier field *V***.auth_value.checksum**.

- *The case V***.auth_level** = **dce_c_authn_level_connect**, **dce_c_authn_level_call** *or* **dce_c_authn_level_pkt**.

  *V***.auth_value.checksum** is defined by the following pseudocode:

  $$\texttt{V.auth\_value.checksum} = \texttt{ENCCKSUM}_{\texttt{V.auth\_value.sub\_type}}(\texttt{K, crc\_assoc\_uuid, dir\_seq, } 0_{8 \text{ or } 16});$$

  In words: The field *V***.auth_value.checksum** is set to the indicated (8-byte or 16-byte) ENCCKSUM of an 8-byte or 16-byte 0-vector, according as *V***.auth_value.sub_type** *is* **dce_c_cn_sub_type_des** *or* **dce_c_cn_sub_type_md5** *respectively*..

  *Security interpretation: The PDU's CRC of association UUID and its direction-bit-adjusted sequence number are integrity-protected (and bound together) by the verifier field V***.auth_value.checksum**.

- *The case V***.auth_level** = **dce_c_authn_level_pkt_integrity** *or* **dce_c_authn_level_pkt_privacy**.

  *V***.auth_value.checksum** is defined by the following pseudocode:

  $$\begin{aligned}&\texttt{struct \{unsigned32 \_1\_; unsigned8 \_2\_; unsigned8 \_3\_;}\\&\texttt{\quad unsigned16 \_4\_;\} vrf =}\\&\texttt{\quad\quad \{V.auth\_value.assoc\_uuid\_crc, V.auth\_value.sub\_type,}\\&\texttt{\quad\quad V.auth\_value.checksum\_length, V.auth\_value.cred\_length\};}\\&\texttt{hdr\_bdy\_vrf = <H, B, vrf>;}\\&\texttt{V.auth\_value.checksum =}\\&\texttt{\quad ENCCKSUM}_{\texttt{V.auth\_value.sub\_type}}\texttt{(K, crc\_assoc\_uuid, dir\_seq,}\\&\texttt{\quad\quad hdr\_bdy\_vrf);}\end{aligned}$$

  In words: Let *vrf* be the IDL-defined NDR-marshalled 8-byte data indicated (it is the first 8 bytes of the verifier's *V***.auth_value** field; that is, it is *V***.auth_value** excluding the *V***.auth_value.credentials** and *V***.auth_value.checksum** fields). Let *hdr_bdy_vrf* be the concatenation of the PDU header *H*, the PDU body *B*, and *vrf*. Then *V***.auth_value.checksum** is set to the indicated (8-byte or 16-byte) ENCCKSUM of *hdr_bdy_vrf*.

Security interpretation: The PDU's header *H*, body *B* (*R*), and the specified fields of the verifier *V* are integrity-protected (and bound together) by the verifier field *V***.auth_value.checksum**.

9.3.1.3 CO Verifier auth_value.credentials

This section specifies the verifier field *V***.auth_value.credentials**. Most of this specification has already been given in Section 13.2.6.3, Credentials Encoding, of the referenced X/Open DCE RPC Specification, so this section just gives the specification of the components there were left unspecified there: namely, the components that were there called *name*, *pac*, *request*, *response* and *error*.

- *name*

  This is unsupported (it will be removed from future versions of the referenced X/Open DCE RPC Specification and this specification).

- *pac*

  This is unsupported (it will be removed from future versions of the referenced X/Open DCE RPC Specification and this specification).

- *request*

  This is (the NDR-marshalled IDL **byte[]** data type underlying) either an **AuthnHeader** or a **PAuthnHeader** data type, dependent on whether the **dce_c_authz_name** or **dce_c_authz_dce** authorisation service has been specified respectively (this is used to authenticate the client to the server, in the sense of Section 4.13 on page 231), with the following supplements:

  — *Options*

    The field *V***.auth_value.credentials.authnHdr-Flags** contains no selected options.

  — *Conversation key*

    The field *V***.auth_value.credentials.authnHdr-EncryptAuthnr.authnr-ConversationKey** is omitted.

  — *Sequence number*

    The field *V***.auth_value.credentials.authnHdr-EncryptAuthnr.authnr-SeqNum** is omitted by the client, and is ignored by the server.

  — *Authorisation data*

    The field *V***.auth_value.credentials.authnHdr-EncryptAuthnr.authnr-AuthzData** is omitted.

  — *Checksum*

    The field *V***.auth_value.credentials.authnHdr-EncryptAuthnr.authnr-Cksum.cksum-Type** has the value **chksumType-CO-RPC** (see Section 4.3.4.1 on page 185), which is defined as follows. The field *V***.auth_value.credentials.authnHdr-EncryptAuthnr.authnr-Cksum.cksum-Type** has as its value the (unique) vector of length 0 (that is, there is no checksum data).

- *response*

  This is (the NDR-marshalled IDL **byte[]** data type underlying) either a **RevAuthnHeader** or a **PRevAuthnHeader** data type, dependent on whether the **dce_c_authz_name** or **dce_c_authz_dce** authorisation service has been specified respectively (this is used to reverse-authenticate the server to the client, in the sense of Section 4.13 on page 231), with the

following supplements:

— *Conversation key*

The field *V***.auth_value.credentials.revAuthnHdr-EncryptPart.revAuthnHdr-ConversationKey** is omitted.

— *Sequence number*

The field *V***.auth_value.credentials.revAuthnHdr-EncryptPart.revAuthnHdr-SeqNum** is omitted by the server, and is ignored by the client.

- *error*

  This is (the NDR-marshalled IDL **byte[]** data type underlying) a **KDSError** data type.

9.3.2 CO Integrity and Confidentiality (PDU Verifiers and Bodies)

Let <*H*, *B*, *V*> be a PDU protected for integrity and/or confidentiality. This section specifies the format and semantics of its body *B* and its verifier field *V***.auth_value**.

Throughout, the following notation is used. Let *authnHdr* denote the authentication header associated with the given PDU — it has been transmitted from the client to the server in a PDU's *V***.auth_value.credentials** field (see above).

- *R* denotes raw (plaintext) body data (generally consisting of IDL-defined NDR-marshalled data generated by a client or server stub, or by the RPC runtime itself) as specified in the referenced X/Open DCE RPC Specification.
- *K* = *authnHdr***.authnHdr-Tkt.tkt-EncryptPart.tkt-SessionKey**.
- (No initialisation vector, *IV*, is used below.)

9.3.2.1 CO dce_c_authn_level_pkt

If *V***.auth_level** = **dce_c_authn_level_pkt**, then the body *B* and verifier field *V***.auth_value.checksum** are specified as follows.

In pseudocode:

```
B = R;
V.auth_value.checksum =
    ENCCKSUM_V.auth_value.sub_type
(K, crc_assoc_uuid, dir_seq, 0_8 or 16);
```

In words: The body *B* is set to the raw data *R* (it may be empty). The field *V***.auth_value.checksum** is set to the indicated (8-byte or 16-byte) ENCCKSUM of a 0-vector (of length 8 or 16 bytes, dependent on whether *V***.auth_value.sub_type** is **dce_c_cn_sub_type_des** or **dce_c_cn_sub_type_md5** respectively).

Security interpretation: The PDU's CRC of association UUID and its direction-bit-adjusted sequence number are integrity-protected (and bound together) by the verifier field *V***.auth_value.checksum**. The body *B* (*R*) is unprotected.

9.3.2.2 CO dce_c_authn_level_pkt_integrity

If *V*.**auth_level** = **dce_c_authn_level_pkt_integrity**, then the body *B* and verifier field *V*.**auth_value.checksum** are specified as follows.

In pseudocode:

```
B = R;
hdr_bdy = <H, B>;
V.auth_value.checksum =
    ENCCKSUM_V.auth_value.sub_type(K, crc_assoc_uuid,
dir_seq, hdr_bdy);
```

In words: The body *B* is set to the raw data *R* (it may be empty). Let *hdr_bdy* be the concatenation of the header *H* and body *B*. Then the field *V*.**auth_value.checksum** is set to the indicated (8-byte or 16-byte) ENCCKSUM of *hdr_bdy*.

Security interpretation: The PDU's CRC of association UUID, direction-bit-adjusted sequence number, header *H* and body *B* (*R*) are integrity-protected (and bound together) by the verifier field *V*.**auth_value.checksum**.

9.3.2.3 CO dce_c_authn_level_pkt_privacy

If *V*.**auth_level** = **dce_c_authn_level_pkt_privacy**, then the body *B* and verifier field *V*.**auth_value.checksum** are specified as follows.

If *R* is empty, in pseudocode:

```
B = R; /*that is, B = empty*/
V.auth_value.checksum =
    ENCCKSUM_V.auth_value.sub_type(K, crc_assoc_uuid, dir_seq, H);
```

In words: This is identical to the **dce_c_authn_level_pkt_integrity** case (in the case *R* is empty), above.

If *R* is non-empty, in pseudocode:

```
enccksum_crc_assoc_uuid_dir_seq_hdr =
    ENCCKSUM_V.auth_value.sub_type(K, crc_assoc_uuid, dir_seq, H);
if (V.auth_value.sub_type == dce_c_cn_sub_type_md5) {
    enccksum_crc_assoc_uuid_dir_seq_hdr =
        FOLD_8(enccksum_crc_assoc_uuid_dir_seq_hdr);
}
R' = <R, 0_(3-Λ(R))(mod 8), (3-Λ(R))(mod 8)>;
crc_bdy = CRC§32(0_4, R');
R'' = <R', crc_bdy>;
B = DES-CBC(K, enccksum_crc_assoc_uuid_dir_seq_hdr, R'');
V.auth_value.checksum = 0_8 or 16;
```

In words: Let *enccksum_crc_assoc_uuid_dir_seq_hdr* be the indicated (8-byte or 16-byte) ENCCKSUM of the header *H*. If *V*.**auth_value.sub_type** = **dce_c_cn_sub_type_md5**, then ''fold the 16-byte *enccksum_crc_assoc_uuid_dir_seq_hdr* in half'' (making it into an 8-byte vector), as defined by the following pseudocode for any 16-byte vector $<C_0, C_1, \cdots, C_{15}>$:

$FOLD_8(<C_0, C_1, \cdots, C_{15}>) = <C_0 \hat{} C_8, C_1 \hat{} C_9, \cdots, C_7 \hat{} C_{15}>;$

Next, let *R´* be the concatenation of the raw data *R*, a 0-vector of length (3–Λ(*R*))(mod 8), and a 1-byte big-endian integer whose value is (3–Λ(*R*))(mod 8); thus, $\Lambda(R') \equiv 4 \pmod 8$. Let *crc_bdy* be the 4-byte CRC of *R´*, with 4-byte 0-vector as seed. Then let *R´´* be the concatenation of *R´* and *crc_bdy*, so that Λ(*R´´*) is a multiple of 8. Then *B* is the indicated DES-CBC encryption of *R´´*. Finally, *V***.auth_value.checksum** is set to an 8-byte or 16-byte 0-vector, dependent on whether *V***.auth_value.sub_type** is **dce_c_cn_sub_type_des** or **dce_c_cn_sub_type_md5** respectively.

Security interpretation: The PDU's CRC of association UUID, direction-bit-adjusted sequence number, header *H* and body *B* (*R*) are integrity-protected (and bound together), and the body *B* (*R*) is confidentiality-protected, all via the encrypted data *B* (not via the verifier field *V***.auth_value.checksum**).

Chapter 10

ACL Editor RPC Interface

This chapter specifies the RPC interface supporting *ACL Editors*, namely the **rdacl** RPC interface (the corresponding **sec_acl** API is specified in Chapter 15). Recall that, by definition, ''ACL Editors'' are just RPC clients that invoke the operations defined in this chapter to access and manipulate (via the RPC server and its ACL managers) the ACLs on protected objects, without actually accessing the protected objects themselves.

Required background for this chapter appears in Section 1.11 on page 55, Chapter 7 and Chapter 8.

10.1 The rdacl RPC Interface

This section specifies (in IDL / NDR) the **rdacl** RPC interface. All servers that support protected objects must export this RPC interface in order for ACL Editors to be able to manage their ACLs. They must also, of course, *protect* it (at a level consistent with the policy of the server) in order to guarantee the security of the server's ACLs and protected objects (see Chapter 9 for generalities on Protected RPC, and see the *required rights* specifications on the **rdacl** operations below).

This section begins with some remarks about identifying protected objects and ACLs, followed by definitions of some common data types, and the **rdacl** interface interspersed with commentary explaining it.

10.1.1 Identifying Protected Objects and ACLs

The **rdacl** interface represents references to ACLs by a 4-tuple of data items, namely:

- An RPC binding handle, identifying the server that manages the protected object whose ACL is being referenced.
- A string, which ''further identifies'', on an application-specific basis (that is, by a ''server-supported namespace'' — see Section 1.11 on page 55), the protected object within the server whose ACL is being referenced.
- An ACL manager type UUID, identifying the ACL manager type of the ACL being referenced (and thereby identifying the ACL manager within the server that can interpret the ACL).
- An ACL type, identifying the ACL type of the ACL being referenced (protection ACL, default object creation ACL, default container creation ACL).

A consistent notation for these items is used throughout this section, namely (in the order listed above):

```
handle_t                    rpc_handle;
sec_acl_component_name_t    component_name;
uuid_t                      *manager_type;
sec_acl_type_t              acl_type;
```

This identification scheme implies that servers supporting the **rdacl** interface must *uniquely identify* all their ACLs by means of such a 4-tuple. Within this constraint, however, the server's management of its protected objects and ACLs is not further specified here, so is application-specific.

Note: Typically, it is the case that the first 2 of the above items (*rpc_handle*, *component_name*) together uniquely identify the protected object itself, and the last 2 items (*manager_type*, *acl_type*) together uniquely identify a particular ACL associated with that protected object. However, this is not a requirement. For example, it is conceivable (though atypical) that a server (identified by *rpc_handle*) could support more than one protected object having the same stringname (*component_name*), provided those were disambiguated (for ACL editing purposes) by being protected by ACL managers of different type UUIDs (*manager_type*). Again, it is entirely conceivable that a server could support protected objects that are uniquely identified by stringname (*component_name*), but having more than one ACL per object (these being disambiguated by their *manager_type*s). This latter is the case of so-called ''polymorphic types'', and is especially useful when the server supports a hierarchical namespace whose internal nodes are not only ''mere'' directories but also participate in some other role (such as a directory that participates in some of the qualities of a leaf node).

10.1.2 Common Data Types and Constants for rdacl Interface

This section specifies (in IDL/NDR) common data types and constants used by the **rdacl** interface (additional data types not defined in this chapter are defined in preceding chapters and in the referenced X/Open DCE RPC Specification).

10.1.2.1 sec_acl_component_name_t

The **sec_acl_component_name_t** data type is a string for ''further identifying'' (in an application-specific manner) a protected object (and hence its ACL) — see Section 10.1.1 on page 345. Its character elements are to be drawn from the Portable Character Set (see Appendix G, Portable Character Set, of the referenced X/Open DCE RPC Specification).

```
typedef [string, ptr] unsigned char   *sec_acl_component_name_t;
```

10.1.2.2 sec_acl_p_t

The **sec_acl_p_t** data type represents a pointer to an ACL.

```
typedef [ptr] sec_acl_t               *sec_acl_p_t;
```

10.1.2.3 sec_acl_list_t

The **sec_acl_list_t** data type represents a list (array) of (pointers to) ACLs.

```
typedef struct {
    unsigned32                        count;
    [size_is(count)] sec_acl_p_t      sec_acls[];
}                                        sec_acl_list_t;
```

10.1.2.4 sec_acl_result_t

The **sec_acl_result_t** data type represents a ''performance-optimised'' version of the **sec_acl_list_t** data type. In the success case (*status* = **error_status_ok**), **sec_acl_result_t** represents a (pointer to a) value of type **sec_acl_list_t**; in the error case (*status* ≠ **error_status_ok**) it is empty (thereby preventing unnecessary marshalling/unmarshalling of data in the error case).

```
typedef union switch (error_status_t status) {
    case error_status_ok:
        [ptr] sec_acl_list_t          acl_list;
    default:
        /*empty*/                     /*empty*/;
}                                         sec_acl_result_t;
```

10.1.2.5 sec_acl_twr_ref_t

The **sac_acl_twr_ref_t** data type represents a pointer to an RPC protocol tower (**twr_t**, defined in Appendix L, Protocol Tower Encoding, of the referenced X / Open DCE RPC Specification).

```
typedef [ref] twr_t                   *sec_acl_twr_ref_t;
```

10.1.2.6 sec_acl_tower_set_t

The **sec_acl_tower_set_t** data type represents a pointer to a list of (pointers to) RPC protocol towers.

```
typedef [ptr] struct {
    unsigned32                          count;
    [size_is(count)] sec_acl_twr_ref_t  towers[];
}                                           *sec_acl_tower_set_t;
```

10.1.2.7 sec_acl_posix_semantics_t

The **sec_acl_posix_semantics_t** data type is a flag word indicating the extent of POSIX semantics supported by an ACL manager.

```
typedef unsigned32                          sec_acl_posix_semantics_t;
const sec_acl_posix_semantics_t             sec_acl_posix_no_semantics
                                                = 0x00000000;
const sec_acl_posix_semantics_t             sec_acl_posix_mask_obj
                                                = 0x00000001;
```

The following values are currently registered:

- **sec_acl_posix_no_semantics**

  ACL manager supports ACLs as described throughout DCE, with the exception that the MASK_OBJ ACLE type is *not* supported (that is, is not present on any ACL).

- **sec_acl_posix_mask_obj**

  ACL manager supports ACLs as described throughout DCE (including the MASK_OBJ ACLE type).

Note: Arguably, DCE ‘‘should’’ support a **sec_acl_semantics_t** flag word data type, encompassing ‘‘all’’ kinds of semantics, not just those related to POSIX. Such support is for future study. For the time being, DCE restricts itself to ‘‘POSIX-like’’ semantics, and the intention is that the bits of the **sec_acl_posix_semantics_t** indicate various ‘‘levels’’ or ‘‘flavours’’ of POSIX-like semantics.

10.1.2.8 Status Codes

The following status codes (transmitted as values of the type **error_status_t**) are specified for the **rdacl** interface. Only their values are specified here — their use is specified in context elsewhere in this specification.

```
const unsigned32  sec_acl_not_implemented        = 0x17122016;
const unsigned32  sec_acl_cant_allocate_memory   = 0x17122017;
const unsigned32  sec_acl_invalid_site_name      = 0x17122018;
const unsigned32  sec_acl_unknown_manager_type   = 0x17122019;
const unsigned32  sec_acl_object_not_found       = 0x1712201a;
const unsigned32  sec_acl_no_acl_found           = 0x1712201b;
const unsigned32  sec_acl_invalid_entry_name     = 0x1712201c;
const unsigned32  sec_acl_expected_user_obj      = 0x1712201d;
const unsigned32  sec_acl_expected_group_obj     = 0x1712201e;
const unsigned32  sec_acl_invalid_entry_type     = 0x1712201f;
const unsigned32  sec_acl_invalid_acl_type       = 0x17122020;
const unsigned32  sec_acl_bad_key                = 0x17122021;
const unsigned32  sec_acl_invalid_manager_type   = 0x17122022;
const unsigned32  sec_acl_read_only              = 0x17122023;
const unsigned32  sec_acl_site_read_only         = 0x17122024;
const unsigned32  sec_acl_invalid_permission     = 0x17122025;
const unsigned32  sec_acl_bad_acl_syntax         = 0x17122026;
const unsigned32  sec_acl_no_owner               = 0x17122027;
const unsigned32  sec_acl_invalid_entry_class    = 0x17122028;
const unsigned32  sec_acl_unable_to_authenticate = 0x17122029;
const unsigned32  sec_acl_name_resolution_failed = 0x1712202a;
const unsigned32  sec_acl_rpc_error              = 0x1712202b;
const unsigned32  sec_acl_bind_error             = 0x1712202c;
const unsigned32  sec_acl_invalid_acl_handle     = 0x1712202d;
const unsigned32  sec_acl_no_update_sites        = 0x1712202e;
const unsigned32  sec_acl_missing_required_entry = 0x17122030;
const unsigned32  sec_acl_duplicate_entry        = 0x17122031;
const unsigned32  sec_acl_bad_parameter          = 0x17122032;
const unsigned32  sec_acl_not_authorized         = 0x17122033;
const unsigned32  sec_acl_server_bad_state       = 0x17122034;
const unsigned32  sec_acl_invalid_dfs_acl        = 0x17122035;
const unsigned32  sec_acl_bad_permset            = 0x17122037;
```

10.1.3 Interface UUID and Version Number for rdacl Interface

The interface UUID and version number for the **rdacl** interface are given by the following:

```
[uuid(47b33331-8000-0000-0d00-01dc6c000000), version(0.0)]
interface rdacl
{ /* begin running listing of rdacl interface */
```

10.1.3.1 Implementation Variability regarding Required Rights

The RPC (**rdacl**) operations specified in this chapter are intended to be implemented by various kinds of servers, whose detailed specification is in general beyond the scope of this specification, and all of whose security requirements cannot therefore be anticipated by DCE. In particular, the authorisation policies (‘‘required rights’’) to the server’s ACLs are the responsibility of the server itself (or more precisely, of its ACL manager(s)), and are in general beyond the scope of this document. As such, the specifications of required rights in this chapter are to be interpreted

as suggestions only, whose exact interpretation must be specified by the implementing servers. To emphasise this, the permissions mentioned in the required rights of this chapter are given hypothetical ‘‘names’’, explicitly denoted with *quotation marks* (indicating that their exact interpretation is to be specified by the specific server in question). Typical interpretations of these hypothetical permissions are given as suggestions.

For an explicit example of a concrete interpretation, namely in the specific case of the RS server (whose specification *is* within the scope of this document), see Section 11.1 on page 358.

10.1.4 rdacl_lookup()

The *rdacl_lookup*() operation retrieves (‘‘reads’’) all ACLs specified by an <*rpc_handle*, *component_name*, *manager_type*, *acl_type*> 4-tuple, creating a copy locally on the client.

```
void
rdacl_lookup (
    [in]        handle_t                    rpc_handle,
    [in]        sec_acl_component_name_t    component_name,
    [in]        uuid_t                      *manager_type,
    [in]        sec_acl_type_t              acl_type,
    [out]       sec_acl_result_t            *acl_result );
```

The *rpc_handle* parameter identifies the server that manages the protected object.

The *component_name* parameter further identifies the protected object within the server.

The *manager_type* parameter identifies an ACL manager type UUID.

The *acl_type* parameter identifies an ACL type.

The *acl_result* parameter returns the result of the operation.

Required rights (suggested): This operation succeeds only if the calling client has ‘‘rdacl-lookup’’ permission (to the specified protected object, according to the specified ACL manager’s policy — see Section 10.1.3.1 on page 348). Typically, servers will grant this permission to clients which have Read-ACL permission to the protected object in question (which, in turn, is typically interpreted as *any* access — see Section 1.9 on page 46).

10.1.5 rdacl_replace()

The *rdacl_replace*() operation applies (‘‘writes’’) all ACLs specified by an <*rpc_handle*, *component_name*, *manager_type*, *acl_type*> 4-tuple. This operation is *atomic*, in the sense that it manipulates a whole ACL (not its individual ACLEs, and it cannot be ‘‘interrupted’’ by another invocation of *rdacl_replace*()) — thus, the specified new ACL entirely replaces the old (existing) ACL on the protected object.

```
void
rdacl_replace (
    [in]        handle_t                    rpc_handle,
    [in]        sec_acl_component_name_t    component_name,
    [in]        uuid_t                      *manager_type,
    [in]        sec_acl_type_t              acl_type,
    [in]        sec_acl_list_t              *acl_list,
    [out]       error_status_t              *status );
```

The *rpc_handle* parameter identifies the server that manages the protected object.

The *component_name* parameter further identifies the protected object within the server.

The *manager_type* parameter identifies an ACL manager type UUID.

The *acl_type* parameter identifies an ACL type.

The *acl_list* parameter specifies the new ACLs to replace the old ACLs.

The *status* parameter returns the status of the operation.

Required rights (suggested): This operation succeeds only if the calling client has ''rdacl-replace'' permission (to the specified protected object, according to the specified ACL manager's policy — see Section 10.1.3.1 on page 348). Typically, servers will grant this permission to clients which have Control (Write-ACL) permission to the protected object in question (see Section 1.9 on page 46).

10.1.6 rdacl_get_access( )

The *rdacl_get_access*( ) operation determines the calling client's access rights to the protected object specified by an <*rpc_handle*, *component_name*, *manager_type*> 3-tuple.

```
void
rdacl_get_access (
    [in]        handle_t                    rpc_handle,
    [in]        sec_acl_component_name_t    component_name,
    [in]        uuid_t                      *manager_type,
    [out]       sec_acl_permset_t           *access_rights,
    [out]       error_status_t              *status );
```

The *rpc_handle* parameter identifies the server that manages the protected object.

The *component_name* parameter further identifies the protected object within the server.

The *manager_type* parameter identifies an ACL manager type UUID.

The *access_rights* parameter returns the calling client's access rights to the specified protected object. This is the client's ''maximum'' access rights; that is, the ''union'' (bitwise OR) of all the permission bits granted to the client, according to the ACLs on the protected object of ACL type **sec_acl_type_object**.

The *status* parameter returns the status of the operation.

Required rights (suggested): This operation succeeds only if the calling client has ''rdacl-get-access'' permission (to the specified protected object, according to the specified ACL manager's policy — see Section 10.1.3.1 on page 348). Typically, servers will grant this permission to clients which have Read-ACL permission to the protected object in question (which, in turn, is typically interpreted as *any* access — see Section 1.9 on page 46).

10.1.7 rdacl_test_access( )

The *rdacl_test_access*( ) operation determines (''tests'') whether or not the calling client has the specified access rights to a protected object.

```
boolean32
rdacl_test_access (
    [in]        handle_t                    rpc_handle,
    [in]        sec_acl_component_name_t    component_name,
    [in]        uuid_t                      *manager_type,
    [in]        sec_acl_permset_t           access_rights,
    [out]       error_status_t              *status );
```

The *rpc_handle* parameter identifies the server that manages the protected object.

The *component_name* parameter further identifies the protected object within the server.

The *manager_type* parameter identifies an ACL manager type UUID of the protected object.

The *access_rights* parameter identifies the specific access rights (according to the ACLs on the protected object of ACL type **sec_acl_type_object**) to be tested.

The *status* parameter returns the status of the operation.

The **boolean32** return value of this operation, which is valid only when *status* returns **error_status_ok**, returns 0 (''false'') if the calling client is denied access, non-0 (''true'') if the client is granted access.

Required rights (suggested): This operation succeeds only if the calling client has ''rdacl-test-access'' permission (to the specified protected object, according to the specified ACL manager's policy — see Section 10.1.3.1 on page 348). Typically, servers will grant this permission to *all* clients (that is, *no* permissions are required).

10.1.8 rdacl_place_holder_1( )

The *rdacl_place_holder_1* ( ) operation is merely an IDL place-holder; it is a ''no-op'' (that is, the specification of its semantics is empty). (The presence of this operation is anachronistic, but necessary.)

```
boolean32
rdacl_place_holder_1 (
    [in]        handle_t                    rpc_handle,
    [in]        sec_acl_component_name_t    _1_,
    [in]        uuid_t                      *_2_,
    [in, ptr]   sec_id_pac_t                *_3_,
    [in]        sec_acl_permset_t           _4_,
    [out]       error_status_t              *status );
```

The *rpc_handle* parameter identifies the server that manages the protected object.

The *_1_* parameter has unspecified semantics.

The *_2_* parameter has unspecified semantics.

The *_3_* parameter has unspecified semantics.

The *_4_* parameter has unspecified semantics.

The *status* parameter returns the status of the operation. It always returns **sec_acl_not_implemented**.

The **boolean32** return value of this operation always returns 0 (''false'').

Required rights (suggested): None.

10.1.9 rdacl_get_manager_types()

The *rdacl_get_manager_types*( ) operation retrieves the types of ACL managers protecting a protected object.

```
void
rdacl_get_manager_types (
    [in]        handle_t                    rpc_handle,
    [in]        sec_acl_component_name_t    component_name,
    [in]        sec_acl_type_t              acl_type,
    [in]        unsigned32                  count_max,
    [out]       unsigned32                  *count,
    [out]       unsigned32                  *num_manager_types,
    [out, size_is(count_max), length_is(*count)]
                uuid_t                      manager_types[],
    [out]       error_status_t              *status );
```

The *rpc_handle* parameter identifies the server that manages the protected object.

The *component_name* parameter further identifies the protected object within the server.

The *acl_type* parameter identifies an ACL type.

The *count_max* parameter identifies the maximum number of ACL manager type UUIDs to be returned (in *manager_types*).

The *count* parameter identifies the actual number of ACL manager type UUIDs returned (in *manager_types*).

The *num_manager_types* parameter identifies the total number of ACL manager types, of ACL type *acl_type*, at the heads of chains (see *rdacl_get_printstring*( ), Section 10.1.10), protecting the protected object — thus, an invocation of this operation is ''completely successful'' only if *count* = *num_manager_types*.

The *manager_types* parameter is an array (of size *count*) of distinct UUIDs identifying different ACL manager types protecting the protected object (in the case of a chain of ACL managers, each supporting ≤ 32 permission bits, only the first ACL manager in the chain is returned in this way, and the rest are returned by calls to *rdacl_get_printstring*( ) — see Section 10.1.10).

The *status* parameter returns the status of the operation.

Required rights (suggested): This operation succeeds only if the calling client has ''rdacl-get-manager-types'' permission (to the specified protected object, according to the specified server's policy (which may, in turn, depend on the policies of the protected object's ACL managers) — see Section 10.1.3.1 on page 348). Typically, servers will grant this permission to *all* clients (that is, *no* permissions are required).

10.1.10 rdacl_get_printstring()

The *rdacl_get_printstring*( ) operation retrieves printstring representations for the permission bits that an ACL manager supports.

```
void
rdacl_get_printstring (
    [in]        handle_t                    rpc_handle,
    [in]        uuid_t                      *manager_type,
    [in]        unsigned32                  count_max,
    [out]       uuid_t                      *manager_type_next,
    [out]       sec_acl_printstring_t       *manager_info,
    [out]       boolean32                   *tokenize,
    [out]       unsigned32                  *num_printstrings,
    [out]       unsigned32                  *count,
    [out, size_is(count_max), length_is(*count)]
                sec_acl_printstring_t       printstrings[],
    [out]       error_status_t              *status );
```

The *rpc_handle* parameter identifies the server that manages the ACL manager.

The *manager_type* parameter identifies an ACL manager type UUID.

The *count_max* parameter identifies the maximum number of printstrings to be returned (in *printstrings*).

The *manager_type_next* parameter, if not equal to **uuid_nil**, identifies the next ACL manager type in a linked list or ‘‘chain’’ of ACL manager types, which can be successively followed until the chain is exhausted (for example, such a chain can be used to support > 32 permission bits). The value **uuid_nil** indicates the end of this chain.

The *manager_info* parameter provides a name and help information for the *manager_type* ACL manager as a whole (as opposed to any of its specific permission bits — those are described in the *printstrings* parameter), as well as a complete list of all its supported permission bits (represented as the union (bitwise OR) of all those supported bits). Concerning this bitwise OR, it is suggested that, by convention, ‘‘simple’’ or ‘‘primitive’’ permission bits (for example, permissions having semantics not interpretable in terms of other permissions) appear in low-order bit positions (and in particular, that the 7 common permission bits, if present, occupy their usual places in the low-order 7 bit positions, 0x0000007f — see Section 8.1.1 on page 319), and that ‘‘complex’’ or ‘‘combination’’ permission bits (for example, permissions having semantics interpretable in terms of other permissions) appear near the high-order end. Thus, for example, for an ACL manager supporting the first 6 of the 7 common permission bits (**rwxcid** but not **t**) plus 2 combination bits, one with value 0x00000080 and having the semantics ‘‘Read and Write’’, and the other with value 0x00000100 and having the semantics ‘‘Read or Write’’, this bitwise OR would be equal to 0x000001bf.

Note: This example, using ‘‘combined rights’’, is contrived to illustrate the problem of tokenisation (below) — it is not necessarily a realistic example. Combining rights in this manner is not necessarily to be encouraged, and the merits/demerits of doing so are not debated here.)

The *tokenize* parameter identifies potential ambiguity in the concatenation of permission printstrings (that is, in the **printstring** fields of the elements of the *printstrings[]* array) — when *tokenize* is 0 (‘‘false’’), the permission printstrings may be concatenated into a single string without ambiguity; when non-0 (‘‘true’’), this property does not hold, and the permission printstrings must be ‘‘tokenised’’ (that is, separated by disambiguating characters; for example, by non-alphanumeric characters, such as whitespace) to avoid ambiguity when concatenated. The consumer of the *tokenize* parameter is typically a user interface program (for example, an ACL editor) which wants to display printstrings to users, and must do so unambiguously. Thus in the example above, if the two combination permissions were represented by, say, the printstrings **raw** (for ‘‘Read and Write’’) and **row** (for ‘‘Read or Write’’), then *tokenize* would be

non-0 (because these printstrings have characters in common with one another, as well as with the common printstrings **r** (''Read'') and **w** (''Write''), and this could potentially be confusing to human users). In this example, a user interface program could display the 8 supported permissions to humans in a manner such as: **r□w□x□c□i□d□raw□row**, where ''□'' denotes the space character (for visual clarity). Alternatively (and probably preferably), tokenisation could have been avoided in this example by choosing different printstrings for the combination permissions, such as **a** and **o** instead of **raw** and **row**, in which case a user interface program could display the supported permissions as **rwxcidao**. (Note that the order of display of permission bits need not be correlated to the order of their values — that's an implementation decision of the user interface program itself, though the reverse-value order indicated here is suggested, for consistency with the well-known 7 common permissions, **rwxcidt**.)

The *num_printstrings* parameter identifies the total number (≤ 32) of permission bits and printstrings ''supported'' (in the sense of *rdacl_get_printstring*( ), as indicated in the following sentence) by the ACL manager (thus, an invocation of this operation is ''completely successful'' only if *count* = *num_printstrings*). In the example above, the value of *num_printstrings* is 9, even though only 8 permission bits are ''actually supported'' by the *manager_type* ACL manager (the **t** bit is not supported, but its position is reserved by convention, so must be skipped).

The *count* parameter identifies the actual number of printstrings returned (in the *printstrings* array). In the example above, the value of *count* is 9.

The *printstrings* parameter is an array (of size *count*) of printstrings returning information about the permission bits (see *manager_info* above) supported by the ACL manager. (See Section 8.1.2 on page 319.) The correspondence between supported permission bits and information about those bits is realised ''as if'' it were given by an array of permission bit / printstring pairs (unsupported permission bits corresponding to NULL printstrings). That is, if permission bit 2^k ($0 \leq k \leq 31$) is supported (this bit is set in the **perm** field of *manager_info* — see Section 8.1.2 on page 319), then *printstring[k]* contains the information about this permission bit. Unsupported permissions correspond to NULL printstrings. Thus in the example above, *printstring[*6*]* is NULL because it corresponds to the unsupported permission bit **t**.

The *status* parameter returns the status of the operation.

Required rights (suggested): This operation succeeds only if the calling client has ''rdacl-get-printstring'' permission (to the specified ACL manager, according to the specified ACL manager's policy — see Section 10.1.3.1 on page 348). Typically, servers will grant this permission to *all* clients (that is, *no* permissions are required).

10.1.11 rdacl_get_referral( )

The *rdacl_get_referral*( ) operation supports a server replication strategy wherein ACL ''updates'' (*rdacl_replace*( )) may not be supported at all ''sites'' (that is, by all server instances or replicas). This operation, which refers clients to sites where ACLs *can* be updated, is to be invoked by a client after an update operation targeted to an ACL site yields a **sec_acl_site_read_only** status error ≠ **error_status_ok**. (Non-replicated servers never return a **sec_acl_site_read_only** status value, so they can provide a trivial implementation of this operation; for example, by returning **sec_acl_not_implemented** as the *status* parameter.)

```
void
rdacl_get_referral (
    [in]        handle_t                    rpc_handle,
    [in]        sec_acl_component_name_t    component_name,
    [in]        uuid_t                      *manager_type,
    [in]        sec_acl_type_t              acl_type,
    [out]       sec_acl_tower_set_t         *tower_set,
    [out]       error_status_t              *status );
```

The *rpc_handle* parameter identifies the server that manages the ACLs in question.

The *component_name* parameter further identifies a protected object within the server.

The *manager_type* parameter identifies an ACLE manager type UUID.

The *acl_type* parameter identifies an ACL type.

The *tower_set* parameter identifies the actual update referral information itself (represented as RPC towers, to which the client can rebind).

The *status* parameter returns the status of the operation.

Required rights (suggested): This operation succeeds only if the calling client has ‘‘rdacl-get-referral’’ permission (to the specified server, according to the server’s policy — see Section 10.1.3.1 on page 348). Typically, servers will grant this permission to *all* clients (that is, *no* permissions are required).

10.1.12 rdacl_get_mgr_types_semantics( )

The *rdacl_get_mgr_types_semantics*( ) operation determines the types of ACL managers protecting a protected object, and the extent of their conformance to POSIX semantics.

```
void
rdacl_get_mgr_types_semantics (
    [in]        handle_t                    rpc_handle,
    [in]        sec_acl_component_name_t    component_name,
    [in]        sec_acl_type_t              acl_type,
    [in]        unsigned32                  count_max,
    [out]       unsigned32                  *count,
    [out]       unsigned32                  *num_manager_types,
    [out, size_is(count_max), length_is(*count)]
                uuid_t                      manager_types[],
    [out, size_is(count_max), length_is(*count)]
                sec_acl_posix_semantics_t   posix_semantics[],
    [out]       error_status_t              *status );

} /* end running listing of rdacl interface */
```

The description of *rdacl_get_mgr_types_semantics*( ) is identical to that of *rdacl_get_manager_types*( ), except that it returns the additional *posix_semantics* array parameter.

The *posix_semantics* parameter is an array of flag words indicating the semantics that the corresponding ACL manager in the *manager_types* array supports.

Required rights (suggested): This operation succeeds only if the calling client has ‘‘rdacl-get-mgr-types-semantics’’ permission (to the specified protected object, according to the specified server’s policy — which may, in turn, depend on the policies of the protected object’s ACL managers — see Section 10.1.3.1 on page 348).

Typically, servers will grant this permission to *all* clients (that is, *no* permissions are required).

Chapter 11

RS Editor RPC Interfaces

This chapter specifies the RPC interfaces supporting *RS* (or *Registry*) *Editors*. These interfaces are:

- **rs_bind** for registry binding operations,
- **rs_policy** for registry policy and properties operations,
- **rs_pgo** for PGO item management,
- **rs_acct** for account management,
- **rs_misc** for miscellaneous registry operations,
- **rs_attr** for manipulating registry attributes,
- **rs_attr_schema** for manipulating registry attribute schemas,
- **rs_prop_acct** for propagating registry account information,
- **rs_prop_acl** for propagating registry ACL information,
- **rs_prop_attr** for propagating registry attributes,
- **rs_prop_attr_schema** for propagating registry attribute schemas,
- **rs_prop_pgo** for propagating registry PGO items,
- **rs_prop_plcy** for propagating registry policy information,
- **rs_prop_replist** for propagating registry replica list,
- **rs_repadm** for replica administration,
- **rs_replist** for replica list administration,
- **rs_repmgr** for replica management,
- **rs_rpladmn** for replica administration,
- **rs_update** for updating replica information,
- **rs_pwd_mgmt** for password management between clients and security daemons,
- **rs_qry** for finding a registry server, and
- **rs_unix** for UNIX interface operations.

The APIs that correspond to these RPC interfaces are specified in Chapter 16.

Recall that, by definition, *RS Editors* are just RPC clients that invoke the operations defined in this chapter to access and manipulate (via the RS server and its ACL managers) the RS datastore items. (See Section 1.12 on page 60 for background.)

Note that the RS supports some RPC interfaces other than those specified in this chapter (they are specified in subsequent chapters).

11.1 RS Protected Objects and their ACL Manager Types

The RS regards its datastore items as protected objects, so it supports the **rdacl** RPC interface, and its ACLs can be managed via ACL Editors. The RS supports seven kinds of protected objects, each of which is managed by a single ACL Manager Type (see Section 1.12 on page 60).

For DCE 1.1 (and newer versions), the RS ACL Managers have been enhanced to support generic ‘‘attribute persmssions’’ that (cell) administrators may assign for access control for attribute types of their choice. The set of new permission bits in the set {O, P, Q, ..., X, Y, Z} are supported by the ACL Manager for each registry object type that supports Extended Registry Attributes (ERAs). In other words, the list of valid permissions for each ACL Manager type listed in Table 11-2 has been augmented with the ‘O’ through ‘Z’ permission bits. All uses of these additional attribute permission bits:

- in the Access Permsets fields of schema entries,
- on ACLs, and
- in policies regarding their use,

will be controlled by the cell administrator. The DCE security services will not interpret or assign meaning to these bits other than what is implied by their inclusion in a schema entry.

Information on Extended Registry Attributes can be found in Section 1.21 on page 100.

The *ACL Manager Type UUIDs* of these five ACL Managers are as follows:

Manager Type	ACL Manager Type UUID
Policy	`06ab8f10-0191-11ca-a9e8-08001e039d7d`
Directory	`06ab9c80-0191-11ca-a9e8-08001e039d7d`
Principal	`06ab9320-0191-11ca-a9e8-08001e039d7d`
Group	`06ab9640-0191-11ca-a9e8-08001e039d7d`
Organisation	`06ab9960-0191-11ca-a9e8-08001e039d7d`
Replist	`2ac24970-60c3-11cb-b261-08001e039d7d`
Attr_schema	`755cd9ce-ded3-11cc-8d0a-080009353559`

Table 11-1 ACL Managers Supported by RS

The *permissions* supported by the RS’s ACL Managers are as follows (see also Section 10.1.3.1 on page 348). Note that those marked with ‘‘ERA’’ are supported as generic permissions only:

Manager Type	Supported Permissions															
	r	c	i	d	t	D	n	f	m	a	u	g	M	A	I	{O-Z}
Policy	r	c							m	a				A		ERA
Directory	r	c	i	d		D	n									ERA
Principal	r	c				D	n	f	m	a	u	g				ERA
Group	r	c			t	D	n	f	m				M			ERA
Organization	r	c			t	D	n	f	m				M			ERA
Replist		c	i	d					m					A	I	ERA
Attr_schema	r	c	i	d					m				M			ERA

Table 11-2 ACL Permissions Supported by RS

The *ACLE Types* supported by the RS's ACL Managers are as follows:

- Only the *Group* Manager Type supports ACLE type GROUP_OBJ (GO).
- Only the *Principal* Manager Type supports ACLE type USER_OBJ (UO).
- ALL Manager Types support all other ACLE types.

This is illustrated in the following two tables:

Manager Type	**Supported ACLE Types**										
	UO	U	GO	G	O	FU	FG	FO	AO	UN	E
Policy		U		G	O	FU	FG	FO	AO	UN	E
Directory		U		G	O	FU	FG	FO	AO	UN	E
Principal	UO	U		G	O	FU	FG	FO	AO	UN	E
Group		U	GO	G	O	FU	FG	FO	AO	UN	E
Organization		U		G	O	FU	FG	FO	AO	UN	E
Replist		U		G	O	FU	FG	FO	AO	UN	E
Attr_schema		U		G	O	FU	FG	FO	AO	UN	E

Table 11-3 ACLE Types Supported by RS

The meanings of the abbreviations of the ACLE types given in this and the preceeding table can be found in the representation of the **sec_acl_entry_type_t** data type in Section 7.1.2 on page 312.

Manager Type	**Supported Delegation ACLE Types**								
	UOD	UD	FUD	GOD	GD	FGD	OD	FOD	AOD
Policy	UOD	UD	FUD	GOD	GD	FGD	OD	FOD	AOD
Directory	UOD	UD	FUD	GOD	GD	FGD	OD	FOD	AOD
Principal	UOD	UD	FUD	GOD	GD	FGD	OD	FOD	AOD
Group	UOD	UD	FUD	GOD	GD	FGD	OD	FOD	AOD
Organization	UOD	UD	FUD	GOD	GD	FGD	OD	FOD	AOD
Replist	UOD	UD	FUD	GOD	GD	FGD	OD	FOD	AOD
Attr_schema	UOD	UD	FUD	GOD	GD	FGD	OD	FOD	AOD

Table 11-4 Delegation ACLE Types Supported by RS

11.1.1 Supported Permissions

The *printstrings*, *bit representations* and *semantics* of the supported permissions (listed in Table 11-2 on page 358) are specified as follows (see also Section 10.1.3.1 on page 348):

- **Read**: ''**r**'', 0x00000001

  Disclose an item's information.

- **Control** (**Change**, **Write-ACL**): ''**c**'', 0x00000008

  Modify an item's ACL.

- **Insert**: ''**i**'', 0x00000010

  Insert into a directory.

- **Delete**: ‘‘**d**’’, 0x00000020

  Delete from a directory.

- **Test**: ‘‘**t**’’, 0x00000040

  Test a principal’s membership in a group or organisation.

- **Delete Item**: ‘‘**D**’’, 0x00000080

  Delete an item.

- **Name**: ‘‘**n**’’, 0x00000100

  Modify name of an item.

- **Fullname**: ‘‘**f**’’, 0x00000200

  Modify the (human-friendly) ‘‘full name’’ (or other annotation).

- **Management Info**: ‘‘**m**’’, 0x00000400

  Modify management information.

- **Authentication Info**: ‘‘**a**’’, 0x00000800

  Modify authentication information.

- **User Info**: ‘‘**u**’’, 0x00001000

  Modify user information.

- **Group**: ‘‘**g**’’, 0x00002000

  Add a principal to a group.

- **Membership**: ‘‘**M**’’, 0x00004000

  Add or delete principals from a group or organisation.

- **Administer**: ‘‘**A**’’, 0x00008000

  Administer the RS.

- **Initialise**: ‘‘**I**’’, 0x00010000

  Initialise replica list.

- **Generic ERA**: ‘‘**O-Z**’’, 0x00020000, 0x00040000, ··· - 0x08000000, 0x10000000

  Generic ERA support only. Not interpreted or assigned meaning by DCE Security Services (controlled by cell administrator). These are marked by an ‘‘ERA’’ in Table 11-2 on page 358.

- **Serviceability**: ‘‘**s**’’, 0x20000000

  Support ERA Serviceability information for the RS.

The RS’s Directory ACL Manager (which is the only one of the seven RS ACL Managers that supports container objects) supports the inheritance model specified in Section 1.8.2 on page 44.

The *required rights* for successful invocation of RS RPC interface operations (and therefore of the routines based on them) are specified in context in the remaining sections of this chapter.

11.2 Common Data Types and Constants for RS Editors

This section specifies (in IDL / NDR) common data types and constants used in the RS's RPC interfaces.

11.2.1 bitset

The **bitset** data type represents a 32-bit flag word.

```
typedef    unsigned32    bitset;
```

11.2.2 sec_timeval_sec_t

The **sec_timeval_sec_t** data type represents the number of seconds elapsed since the epoch 00:00.00 January 1, 1970 AD.

```
typedef    signed32    sec_timeval_sec_t;
```

11.2.3 sec_timeval_t

The **sec_timeval_t** data type consists of a structure containing the full UNIX time. The structure contains two 32-bit integers that indicate seconds (**sec**) and microseconds (**usec**) since 00:00.00 January 1, 1970 AD.

```
typedef struct {
    signed32    sec;
    signed32    usec;
} sec_timeval_t;
```

11.2.4 sec_rgy_name_t — Short and Long PGO Names

The **sec_rgy_name_t** data type represents stringnames, usually short PGO names in the RS-supported PGO namespaces (on occasion, as in Section 11.3.3 on page 364, this data type is used to indicate names other than short PGO names). These short PGO names are also called *security-*(*domain-*)*relative* names, indicating that they are subordinate to a specified PGO *naming domain*, of type **sec_rgy_domain_t**, as specified in Section 11.5.1.1 on page 379. They are the names that appear at the level of the RPC interfaces supported by RS Editors.

```
const signed32           sec_rgy_name_max_len = 1024;
const signed32           sec_rgy_name_t_size  = 1025;
typedef [string] char    sec_rgy_name_t[sec_rgy_name_t_size];
```

The syntax of **sec_rgy_name_t** is the same as the CDS naming syntax (as specified in the referenced X / Open DCE Directory Services Specification), with the additional stipulation that these names always have length in the interval [1, 1024]. In particular, a short PGO name cannot begin or end with the character **/** (slash).

In all applications where a **sec_rgy_name_t** occurs, the context of a PGO naming domain must also be indicated, explicitly or implicitly. Then, a long PGO name (or *security-relative name*) may be formed by prepending the stringname associated with the PGO domain to the short PGO name (**sec_rgy_name_t**), separated by a **/** (slash).

For example, the short PGO name **foo/bar** in the context of the PGO principal domain (**sec_rgy_domain_person**, which has associated stringname **person** — see Section 11.5.1.1 on page 379), yields the long PGO name **person/foo/bar**. (And then this would further appear in the global namespace, via the junction architecture (Section 1.11 on page 55), as the name **/.../***name-of-server*/**person/foo/bar** — for example, as **/.../***name-of-cell*/**sec/person/foo/bar**.)

11.2.5 sec_rgy_pname_t

The **sec_rgy_pname_t** data type represents printable stringnames.

```
const signed32          sec_rgy_pname_max_len = 256;
const signed32          sec_rgy_pname_t_size  = 257;
typedef [string] char   sec_rgy_pname_t[sec_rgy_pname_t_size];
```

The characters of **sec_rgy_pname_t** are to be taken from the DCE Portable Character Set.

11.2.6 sec_rgy_login_name_t

The **sec_rgy_login_name_t** data type represents (the name of) an account.

```
typedef struct {
    sec_rgy_name_t      pname;
    sec_rgy_name_t      gname;
    sec_rgy_name_t      oname;
} sec_rgy_login_name_t;
```

Its fields are the following:

- **pname**
  Principal name associated with the account (subordinate to the **sec_rgy_domain_person** domain).
- **gname**
  Group name (subordinate to the **sec_rgy_domain_group** domain).
- **oname**
  Organisation name (subordinate to the **sec_rgy_domain_org** domain).

11.2.7 sec_rgy_cursor_t

The **sec_rgy_cursor_t** data type is used as a datastore *cursor* (that is, a position indicator) for incremental operations in the RS datastore.

```
typedef struct {
    uuid_t            source;
    signed32          position;
    boolean32         valid;
} sec_rgy_cursor_t;
```

Its fields are the following:

- **source**
  The object UUID of the RS server, identifying the RS server/datastore for which a cursor is meaningful.

  Note: This is not merely the cell UUID of such an RS server; this is especially significant in a cell with replicated RS servers, as cursors are not meaningful across RS servers/datastores/replicas.
- **position**
  Position in the datastore of the RS server indicated by **source**. Its use is RS server/datastore implementation-dependent, in the sense that only the RS server indicated by **source** can interpret it. In particular, unless specified otherwise in DCE, any implied ordering of the various kinds of data in the RS datastore (for example, the order in which data is returned by multiple invocations of an operation that takes a cursor as in input/output parameter) is also implementation-dependent (for example, alphabetical by principal name, chronological by

creation time, random, and so on).

- **valid**
  If 0 (FALSE), this cursor represents the initial position of an RS datastore; if non-0 (TRUE), a non-initial position (as determined by **source** and **position**) is indicated. Clients must set **valid** to 0 the first time a cursor is used, to initiate an RS datastore retrieval request; after this initial request, clients must treat a cursor as an opaque data object (that is, not accessing any of its fields), interpretable only by the RS server indicated by **source**.

11.2.8 rs_cache_data_t

The **rs_cache_data_t** data type represents RS datastore modification time information. This information is intended to be used by RS Editor clients to manage their caches of RS datastore information they may maintain.

Note: Maintaining a cache, as well as the details of its maintenance, is implementation-specific, and so is not further specified by DCE.

```
typedef struct {
    uuid_t              site_id;
    sec_timeval_sec_t   person_dtm;
    sec_timeval_sec_t   group_dtm;
    sec_timeval_sec_t   org_dtm;
} rs_cache_data_t;
```

Its fields are the following (see Section 11.5.1.1 on page 379 and Section 11.5.1.4 on page 380 for definitions of terms used here):

- **site_id**
  The object UUID of the RS server to which this modification time information pertains.
- **person_dtm**
  Time of last deletion (or change) of name, UUID or local-ID, to a PGO item in the principal domain.
- **group_dtm**
  Time of last deletion (or change) of name, UUID or local-ID, to a PGO item in the group domain.
- **org_dtm**
  Time of last deletion (or change) of name, UUID or local-ID, to a PGO item in the organisation domain.

Note: The **dtm** (date/time modified) information conveyed by the **rs_cache_data_t** data type is fairly coarse-grained, but since PGO nodes are only infrequently modified, a more sophisticated caching scheme is not considered necessary.

11.2.9 sec_rgy_handle_t

The **sec_rgy_handle_t** data type represents a pointer to a RS server handle. The RS server is bound to a handle with the *sec_rgy_site_open*( ) routine.

```
typedef void    *sec_rgy_handle_t;
```

11.3 The rs_bind RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_bind** RPC interface.

11.3.1 Common Data Types and Constants for rs_bind

The following are common data types and constants used in the **rs_bind** interface.

11.3.1.1 rs_replica_name_p_t

The **rs_replica_name_p_t** data type represents the (cell-relative) RPC binding stringname of an RS server.

```
typedef [string] unsigned char   *rs_replica_name_p_t;
```

11.3.1.2 rs_replica_twr_vec_p_t

The **rs_replica_twr_vec_p_t** data type represents a vector of (pointers to) RPC protocol towers.

```
typedef struct
{
    unsigned32                          num_towers;
    [size_is(num_towers)] twr_p_t       towers[];
} rs_replica_twr_vec_t, *rs_replica_twr_vec_p_t;
```

Its fields are the following:

- **num_towers**
  The number of elements in the **towers[ ]** array.
- **towers[ ]**
  The actual vector (of size **num_towers**) of (pointers to) RPC protocol towers. (For the IDL definition of the **twr_p_t** data type, see Appendix N, IDL Data Type Declarations, of the referenced X / Open DCE RPC Specification.)

11.3.2 Interface UUID and Version Number for rs_bind

The interface UUID and version number for the **rs_bind** interface are given by the following:

```
[
    uuid(d46113d0-a848-11cb-b863-08001e046aa5),
    version(2.0),
    pointer_default(ptr)
]
interface rs_bind {
/* begin running listing of rs_bind interface */
```

11.3.3 rs_bind_get_update_site( )

The *rs_bind_get_update_site*( ) operation returns binding information for an update RS server (as opposed to a query server).

```
void
rs_bind_get_update_site (
    [in]     handle_t                rpc_handle,
    [out]    sec_rgy_name_t          cellname,
    [out]    boolean32               *update_site,
    [out]    rs_replica_name_p_t     *update_site_name,
    [out]    uuid_t                  *update_site_id,
    [out]    rs_replica_twr_vec_p_t  *update_site_twrs,
    [out]    error_status_t          *status );
}
```

The *rpc_handle* parameter identifies the RS server (called the *bound server* in this section).

The *cellname* parameter indicates the bound server's cell.

The *update_site* parameter indicates whether the bound server is an update server or not: if non-0 (TRUE), the bound server is an update server; if 0 (FALSE), it is only a query (non-update) server.

The *update_site_name* parameter indicates the cell-relative RS binding stringname (relative to the cell *cellname*) of an update server in the cell *cellname*.

The semantics of the *update_site_id* parameter are unspecified in this revision of DCE.

Note: It is intended that *update_site_id* indicates the object UUID of an update server replica in the cell *cellname*, but this notion is specific to the replication model, which is not specified in this revision of DCE (it is intended for inclusion in a future revision). (The notion of object UUID of update server replica is not be confused with the RS's object UUID in the sense of RPC, which is registered in the CDS namespace, and which represents all replicas, not just this update server replica.)

The *update_site_twrs* parameter indicates the protocol tower set of an update server in the cell *cellname*.

The *status* parameter returns the status of the operation.

Required rights: None.

11.4 The rs_policy RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_policy** RPC interface.

11.4.1 Common Data Types and Constants for rs_policy

The following are common data types and constants used in the **rs_policy** interface.

11.4.1.1 sec_timeval_period_t

The **sec_timeval_period_t** data type measures time intervals in seconds.

```
typedef signed32       sec_timeval_period_t;
```

11.4.1.2 sec_rgy_properties_flags_t

The **sec_rgy_properties_flags_t** data type is a flag word representing attributes of an RS server / datastore.

```
typedef  bitset      sec_rgy_properties_flags_t;
const unsigned32     sec_rgy_prop_readonly           = 0x1;
const unsigned32     sec_rgy_prop_auth_cert_unbound  = 0x2;
const unsigned32     sec_rgy_prop_shadow_passwd      = 0x4;
const unsigned32     sec_rgy_prop_embedded_unix_id   = 0x8;
```

The following values are currently registered:

- **sec_rgy_prop_readonly**
  Read-only RS site (that is, this replica of the RS server will not service operations that modify data held in the RS datastore). The following are the RS operations that can potentially modify data held in RS datastores (and, therefore, whose service is prevented by the **sec_rgy_prop_readonly** flag): *rdacl_replace*( ), *rs_acct_add*( ), *rs_acct_delete*( ), *rs_acct_rename*( ), *rs_acct_replace*( ), *rs_auth_policy_set_info*( ), *rs_pgo_add*( ), *rs_pgo_add_member*( ), *rs_pgo_delete*( ), *rs_pgo_delete_member*( ), *rs_pgo_rename*( ), *rs_pgo_replace*( ), *rs_policy_set_info*( ), *rs_properties_set_info*( ).

- **sec_rgy_prop_auth_cert_unbound**
  All certificates (tickets and privilege tickets) generated at this RS site are to be usable at any client site; that is, are not bound to the host from which the client requested the certificate — they contain no client addresses; see Section 4.4.1 on page 195.)

- **sec_rgy_prop_shadow_passwd**
  No (protected) passwords are to be transmitted remotely. (Note that the (protected) passwords in question are those relevant to local operating systems — the passwords relevant to the Login Facility specified by DCE are not stored in the RS datastore.) The following are the operations that honour this property: *rs_acct_lookup*( ), *rs_login_get_info*( ).

  The word ''shadow'' is used here in analogy with the notion of ''shadow password file'', with the same semantics: ''as a general statement of intent, passwords, even protected ones, should never be visible''. In the cases where (protected) passwords would normally be transmitted, operations that honour this RS property must transmit something else which cannot be interpreted as a (protected) password (the character string of length 1, ***** (asterisk) (expressed in C-like pseudocode), is recommended, though not required).

- **sec_rgy_prop_embedded_unix_id**
  The UUIDs of principals (except for KDS principals), groups and organisations contain embedded local-IDs (see Section 5.2.1.1 on page 278).

11.4.1.3 sec_rgy_properties_t

The **sec_rgy_properties_t** data type represents the attributes of the RS's Policy item.

```
typedef struct {
    signed32                    read_version;
    signed32                    write_version;
    sec_timeval_period_t        minimum_ticket_lifetime;
    sec_timeval_period_t        default_certificate_lifetime;
    unsigned32                  low_unix_id_person;
    unsigned32                  low_unix_id_group;
    unsigned32                  low_unix_id_org;
    unsigned32                  max_unix_id;
    sec_rgy_properties_flags_t  flags;
    sec_rgy_name_t              realm;
    uuid_t                      realm_uuid;
    signed32                    unauthenticated_quota;
} sec_rgy_properties_t;
```

Its fields are the following:

- **read_version**
  The semantics of this field are currently unspecified (it is intended that when it is specified, it will denote a property of the underlying RS datastore, indicating the earliest version of the RS software that can correctly read the datastore). Implementations must set it to 1.
- **write_version**
  The semantics of this field are currently unspecified (it is intended that when it is specified, it will denote a property of the underlying RS datastore, indicating the earliest version of the RS software that can correctly write the datastore). Implementations must set it to 1.
- **minimum_ticket_lifetime**
  Cell-wide minimum period of time for which a ticket is to be valid. In the case of the Kerberos authentication service (which is the only authentication service currently supported), ''ticket'' here is interpreted as ''Kerberos ticket'' (see Chapter 4), and this field is the same as the *cell-wide minimum ticket lifetime* listed in Section 4.11 on page 217.
- **default_certificate_lifetime**
  Cell-wide default period of time for which a certificate is to be valid. In the case of Kerberos authentication, ''certificate'' here is interpreted as ''Kerberos ticket-granting-ticket'', and this field is the same as the *cell-wide default ticket-granting-ticket lifetime* listed in Section 4.11 on page 217.
- **low_unix_id_person**
  The minimum local-ID that the RS will assign to a newly created principal.
- **low_unix_id_group**
  The minimum local-ID that the RS will assign to a newly created group.
- **low_unix_id_org**
  The minimum local-ID that the RS will assign to a newly created organisation.

- **max_unix_id**
  The maximum local-ID that the RS will assign to any newly created entity (in any PGO domain).
- **flags**
  Flag word.
- **realm**
  Name of the cell to which the RS belongs (and holds the security information for).
- **realm_uuid**
  UUID of cell to which RS belongs.
- **unauthenticated_quota**
  Quota for unauthenticated users. Concerning the general definition of quotas in the RS datastore, see Section 11.5.1.4 on page 380. Relative to that definition, the notion of **unauthenticated_quota** is used for unauthenticated callers of *rs_pgo_add*( ) and *rs_acct_add*( ).

11.4.1.4 sec_rgy_plcy_pwd_flags_t

The **sec_rgy_plcy_pwd_flags_t** data type is a flag word indicating password policy restrictions.

```
typedef bitset      sec_rgy_plcy_pwd_flags_t;
const unsigned32    sec_rgy_plcy_pwd_no_spaces    = 0x1;
const unsigned32    sec_rgy_plcy_pwd_non_alpha    = 0x2;
```

The following values are currently registered:

- **sec_rgy_plcy_pwd_no_spaces**
  Password must not contain the space character.
- **sec_rgy_plcy_pwd_non_alpha**
  Password must contain a non-alphabetic character.

11.4.1.5 sec_rgy_plcy_t

The **sec_rgy_plcy_t** data type represents an organisation's policy information (or that of the cell).

```
typedef struct {
    signed32                    passwd_min_len;
    sec_timeval_period_t        passwd_lifetime;
    sec_timeval_sec_t           passwd_exp_date;
    sec_timeval_period_t        acct_lifespan;
    sec_rgy_plcy_pwd_flags_t    passwd_flags;
} sec_rgy_plcy_t;
```

Its fields are the following:

- **passwd_min_len**
  The minimum allowable length, in characters (or, equivalently, bytes) for a password. Currently, this value is intended to be enforced by client-side password changing utilities, not by the RS server (this may change in future releases).
- **passwd_lifetime**
  The lifetime for a password (or, equivalently, a long-term key). Its value must be ≥ 0. Currently, this value is intended to be enforced by client-side login utilities, not by the RS server (this may change in future releases). See next item.
- **passwd_exp_date**
  The lifetime for a password (or, equivalently, a long-term key). Its value must be ≥ 0.

Currently, this value is intended to be enforced by client-side login utilities, not by the RS server (this may change in future releases).

The time at which an account's password will expire (that is, after which logging in to this account will be granted only after the password has been changed), denoted here as *acct-passwd-exp-time*, is intended to be calculated by the client-side login code via the following algorithm, where **passwd_lifetime** and **passwd_exp_date** are from the effective policy for the account (determined by choosing the strictest (smallest, earliest) policy parameters from the account's organisation policy and its cell policy), and where *acct-passwd-last-update* (whose value must be ≥ 0) is the time of the most recent change of the account's password, or equivalently, long-term key (thus, it is the same as **passwd_dtm** in Section 11.6.1.15 on page 397):

— If **passwd_lifetime** = 0, then *acct-passwd-exp-time* is equal to **passwd_exp_date**, unless *acct-passwd-last-update* is greater (later) than **passwd_exp_date** (that is, the account's password/key has been changed since **passwd_exp_date**), in which case, **passwd_exp_date** is equal to 0 (meaning the password/key does not expire).

— If **passwd_lifetime** > 0 and **passwd_exp_date** = 0, then *acct-passwd-exp-time* is equal to *acct-passwd-last-update* + **passwd_lifetime**.

— If **passwd_lifetime** > 0 and **passwd_exp_date** > 0, then *acct-passwd-exp-time* is equal to the minimum (earliest) of the two values determined by the two calculations in the two preceding cases, where the value 0 (meaning the password/key does not expire) is considered to be later than any non-zero value.

The value of *acct-passwd-exp-time* must be ≥ 0; the value 0 indicates that the password/key does not expire.

- **acct_lifespan**
  The length of time an account is valid. Its value must be ≥ 0. Currently, this value is intended to be enforced by client-side login utilities, not by the RS server (this may change in future releases).

  The time at which an account will expire (that is, after which logging in to this account will be denied), denoted here *acct-exp-time*, is intended to be calculated by the client-side login code via the following algorithm, where **acct_lifespan** is from the effective policy for the account (see previous item for definition of this), and where **creation_date** and **expiration_date** (both of whose values must be ≥ 0) are from the account's administrative information (see Section 11.6.1.5 on page 392). (The account can only be reactivated by the administrative action of modifying **expiration_date**.)

  — If **acct_lifespan** = 0, then *acct-exp-time* is equal to **expiration_date**.

  — If **acct_lifespan** > 0 and **expiration_date** = 0, then *acct-exp-time* is equal to **creation_date** + **acct_lifespan**.

  — If **acct_lifespan** > 0 and **expiration_date** > 0, then *acct-exp-time* is equal to the minimum (earliest) of the two values determined by the two calculations in the two preceding cases.

  The value of *acct-exp-time* must be ≥ 0; the value 0 indicates that the account does not expire.

- **passwd_flags**
  Flag word.

11.4.1.6 sec_rgy_plcy_auth_t

The **sec_rgy_plcy_auth_t** data type represents cell-wide authentication policy.

```
typedef struct {
    sec_timeval_period_t     max_ticket_lifetime;
    sec_timeval_period_t     max_renewable_lifetime;
} sec_rgy_plcy_auth_t;
```

Its fields are the following:

- **max_ticket_lifetime**
  Maximum ticket lifetime. In the case of Kerberos authentication, ''ticket'' here is interpreted as ''Kerberos ticket'' (see Chapter 4), and this field is the same as the *cell-wide maximum ticket lifetime* listed in Section 4.11 on page 217.
- **max_renewable_lifetime**
  Maximum renewable ticket lifetime. In the case of Kerberos authentication, ''ticket'' here is interpreted as ''Kerberos ticket'' (see Chapter 4), and this field is same as the *cell-wide maximum renewable ticket lifetime* listed in Section 4.11 on page 217.

11.4.1.7 Status Codes

The following status codes (transmitted as values of the type **error_status_t**) are specified for the RS editor interfaces. Only their values and a short one-line description of them are specified here — their detailed usage is specified in context elsewhere in this chapter.

Values:

```
const unsigned32  sec_rgy_not_implemented         = 0x17122073;
const unsigned32  sec_rgy_bad_domain              = 0x17122074;
const unsigned32  sec_rgy_object_exists           = 0x17122075;
const unsigned32  sec_rgy_name_exists             = 0x17122076;
const unsigned32  sec_rgy_unix_id_changed         = 0x17122077;
const unsigned32  sec_rgy_is_an_alias             = 0x17122078;
const unsigned32  sec_rgy_no_more_entries         = 0x17122079;
const unsigned32  sec_rgy_object_not_found        = 0x1712207a;
const unsigned32  sec_rgy_server_unavailable      = 0x1712207b;
const unsigned32  sec_rgy_not_member_group        = 0x1712207c;
const unsigned32  sec_rgy_not_member_org          = 0x1712207d;
const unsigned32  sec_rgy_not_member_group_org    = 0x1712207e;
const unsigned32  sec_rgy_incomplete_login_name   = 0x1712207f;
const unsigned32  sec_rgy_passwd_invalid          = 0x17122080;
const unsigned32  sec_rgy_not_authorized          = 0x17122081;
const unsigned32  sec_rgy_read_only               = 0x17122082;
const unsigned32  sec_rgy_bad_alias_owner         = 0x17122083;
const unsigned32  sec_rgy_bad_data                = 0x17122084;
const unsigned32  sec_rgy_cant_allocate_memory    = 0x17122085;
const unsigned32  sec_rgy_dir_not_found           = 0x17122086;
const unsigned32  sec_rgy_dir_not_empty           = 0x17122087;
const unsigned32  sec_rgy_bad_name                = 0x17122088;
const unsigned32  sec_rgy_dir_could_not_create    = 0x17122089;
const unsigned32  sec_rgy_dir_move_illegal        = 0x1712208a;
const unsigned32  sec_rgy_quota_exhausted         = 0x1712208b;
const unsigned32  sec_rgy_foreign_quota_exhausted = 0x1712208c;
const unsigned32  sec_rgy_no_more_unix_ids        = 0x1712208d;
```

```
const unsigned32  sec_rgy_uuid_bad_version        = 0x1712208e;
const unsigned32  sec_rgy_key_bad_version         = 0x1712208f;
const unsigned32  sec_rgy_key_version_in_use      = 0x17122090;
const unsigned32  sec_rgy_key_bad_type            = 0x17122091;
const unsigned32  sec_rgy_crypt_bad_type          = 0x17122092;
const unsigned32  sec_rgy_bad_scope               = 0x17122093;
const unsigned32  sec_rgy_object_not_in_scope     = 0x17122094;
const unsigned32  sec_rgy_cant_authenticate       = 0x17122095;
const unsigned32  sec_rgy_alias_not_allowed       = 0x17122096;
const unsigned32  sec_rgy_bad_chksum_type         = 0x17122097;
const unsigned32  sec_rgy_bad_integrity           = 0x17122098;
const unsigned32  sec_rgy_key_bad_size            = 0x17122099;
const unsigned32  sec_rgy_mkey_cant_read_stored   = 0x1712209a;
const unsigned32  sec_rgy_mkey_bad_stored         = 0x1712209b;
const unsigned32  sec_rgy_mkey_bad                = 0x1712209c;
const unsigned32  sec_rgy_bad_handle              = 0x1712209d;
const unsigned32  sec_rgy_s_pgo_is_required       = 0x1712209e;
const unsigned32  sec_rgy_host_context_not_avail  = 0x1712209f;
const unsigned32  sec_rgy_mkey_file_io_failed     = 0x171220a0;
const unsigned32  sec_rgy_tower_rebind_failed     = 0x171220a1;
const unsigned32  sec_rgy_site_not_absolute       = 0x171220a2;
const unsigned32  sec_rgy_bad_nameservice_name    = 0x171220a3;
const unsigned32  sec_rgy_log_entry_out_of_range  = 0x171220a4;
const unsigned32  sec_rgy_era_pwd_mgmt_auth_type  = 0x171220a5;
const unsigned32  sec_rgy_passwd_too_short        = 0x171220a6;
const unsigned32  sec_rgy_passwd_non_alpha        = 0x171220a7;
const unsigned32  sec_rgy_passwd_spaces           = 0x171220a8;
```

Descriptions:

- **sec_rgy_not_implemented**
  Operation not (or not yet) implemented.
- **sec_rgy_bad_domain**
  Operation not supported on specified domain.
- **sec_rgy_object_exists**
  Object already exists.
- **sec_rgy_name_exists**
  Name already exists.
- **sec_rgy_unix_id_changed**
  Local ID changed on an alias add.
- **sec_rgy_is_an_alias**
  Query returned an alias but aliases were not allowed to satisfy the query (see Section 11.5.7 on page 386).
- **sec_rgy_no_more_entries**
  No more matching entries.
- **sec_rgy_object_not_found**
  Registry object not found.
- **sec_rgy_server_unavailable**
  Registry server unavailable.

- **sec_rgy_not_member_group**
  User is not member of specified group.
- **sec_rgy_not_member_org**
  User is not member of specified organisation.
- **sec_rgy_not_member_group_org**
  User is not member of specified group and organisation.
- **sec_rgy_incomplete_login_name**
  Incomplete login name specification.
- **sec_rgy_passwd_invalid**
  Invalid password.
- **sec_rgy_not_authorized**
  User is not authorised to update record.
- **sec_rgy_read_only**
  Registry is read only; updates are not allowed.
- **sec_rgy_bad_alias_owner**
  PGO alias entry has an invalid owner.
- **sec_rgy_bad_data**
  Invalid data record.
- **sec_rgy_cant_allocate_memory**
  Unable to allocate memory.
- **sec_rgy_dir_not_found**
  Directory not found.
- **sec_rgy_dir_not_empty**
  Directory not empty.
- **sec_rgy_bad_name**
  Illegal PGO or directory name.
- **sec_rgy_dir_could_not_create**
  Unable to create directory.
- **sec_rgy_dir_move_illegal**
  Directory move not allowed.
- **sec_rgy_quota_exhausted**
  Principal quota exhausted.
- **sec_rgy_foreign_quota_exhausted**
  Foreign quota for realm exhausted.
- **sec_rgy_no_more_unix_ids**
  Local ID space for domain has been exhausted.
- **sec_rgy_uuid_bad_version**
  UUID version invalid.
- **sec_rgy_key_bad_version**
  Key version number out of range.
- **sec_rgy_key_version_in_use**
  Key version number currently in use.

- **sec_rgy_key_bad_type**
  Key type not supported.
- **sec_rgy_crypt_bad_type**
  Encrypt/decrypt type not supported.
- **sec_rgy_bad_scope**
  Scope doesn't name existing directory or PGO.
- **sec_rgy_object_not_in_scope**
  Object found was not in scope.
- **sec_rgy_cant_authenticate**
  Can't establish authentication to security server.
- **sec_rgy_alias_not_allowed**
  Can't add alias for this name or principal (for example, **krbtgt**).
- **sec_rgy_bad_chksum_type**
  Checksum type not supported.
- **sec_rgy_bad_integrity**
  Data integrity error.
- **sec_rgy_key_bad_size**
  Invalid size for key data.
- **sec_rgy_mkey_cant_read_stored**
  Can't read stored master key.
- **sec_rgy_mkey_bad_stored**
  Stored master key is bad.
- **sec_rgy_mkey_bad**
  Supplied master key is bad.
- **sec_rgy_bad_handle**
  Bad security context handle.
- **sec_rgy_s_pgo_is_required**
  PGO/account is required and can't be deleted.
- **sec_rgy_host_context_not_avail**
  Login context of local host principal not available.
- **sec_rgy_mkey_file_io_failed**
  Master key file I/O operation failed.
- **sec_rgy_tower_rebind_failed**
  No usable tower entries.
- **sec_rgy_site_not_absolute**
  Registry site name must be absolute.
- **sec_rgy_bad_nameservice_name**
  Invalid nameservice name.
- **sec_rgy_log_entry_out_of_range**
  Invalid log entry module or operation.
- **sec_rgy_era_pwd_mgmt_auth_type**
  Authorization type for pwd_mgmt_binding ERA binding cannot be none.

- **sec_rgy_passwd_too_short**
  Password is too short.
- **sec_rgy_passwd_non_alpha**
  Passwords must contain at least one non-alphanumeric character.
- **sec_rgy_passwd_spaces**
  Passwords must contain at least one non-blank character.

11.4.2 Interface UUID and Version Number for rs_policy

The interface UUID and version number for the **rs_policy** interface are given by the following:

```
[
    uuid(4C878280-4000-0000-0D00-028714000000),
    version(1)
]
interface rs_policy {
/* begin running listing of rs_policy interface */
```

11.4.3 rs_properties_get_info()

The *rs_properties_get_info*( ) operation retrieves (reads) the RS Policy's property information.

```
[idempotent] void
rs_properties_get_info (
    [in]     handle_t                rpc_handle,
    [out]    sec_rgy_properties_t    *properties,
    [out]    rs_cache_data_t         *cache_info,
    [out]    error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *properties* parameter indicates the RS Policy's properties.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Read (**r**) permission on the RS Policy object.

11.4.4 rs_properties_set_info()

The *rs_properties_set_info*( ) operation modifies (writes) the RS Policy's property information.

```
void
rs_properties_set_info (
    [in]     handle_t              rpc_handle,
    [in]     sec_rgy_properties_t  *properties,
    [out]    rs_cache_data_t       *cache_info,
    [out]    error_status_t        *status );
```

The *rpc_handle* parameter identifies the RS server.

The *properties* parameter indicates the RS Policy's properties.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Management Info (**m**) permission on the RS's Policy object.

11.4.5 rs_policy_get_info()

The *rs_policy_get_info*( ) operation retrieves (reads) an organisation's policy information (or that of the cell).

```
[idempotent] void
rs_policy_get_info (
    [in]     handle_t          rpc_handle,
    [in]     sec_rgy_name_t    organization,
    [out]    sec_rgy_plcy_t    *policy_data,
    [out]    rs_cache_data_t   *cache_info,
    [out]    error_status_t    *status
);
```

The *rpc_handle* parameter identifies the RS server.

The *organization* parameter indicates the organisation whose policy information is to be retrieved; or, if NULL, the RS Policy's policy information is indicated. (If non-NULL, this string is interpreted as a short PGO name, subordinate to the organisation naming domain — see Section 11.2.4 on page 361.)

The *policy_data* parameter indicates the retrieved policy information.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Read (**r**) permission on the object specified by the *organization* parameter.

11.4.6 rs_policy_set_info()

The *rs_policy_set_info*( ) modifies (writes) an organisation's policy information (or that of the cell).

```
void
rs_policy_set_info (
    [in]     handle_t          rpc_handle,
    [in]     sec_rgy_name_t    organization,
    [in]     sec_rgy_plcy_t    *policy_data,
    [out]    rs_cache_data_t   *cache_info,
    [out]    error_status_t    *status );
```

The *rpc_handle* parameter identifies the RS server.

The *organization* parameter indicates the organisation whose policy information is to be modified; or, if NULL, the RS Policy's policy information is indicated. (If non-NULL, this string is interpreted as a short PGO name, subordinate to the organisation naming domain — see Section 11.2.4 on page 361.)

The *policy_data* parameter indicates the policy information that is to be written.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Management Info (**m**) permission on the object specified by the *organization* parameter.

11.4.7 rs_policy_get_effective()

The *rs_policy_get_effective*( ) operation returns an organisation's effective policy information; that is, the more restrictive of the organisation's policy information and the cell's policy information.

```
[idempotent] void
rs_policy_get_effective (
    [in]      handle_t            rpc_handle,
    [in]      sec_rgy_name_t      organization,
    [out]     sec_rgy_plcy_t      *policy_data,
    [out]     rs_cache_data_t     *cache_info,
    [out]     error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *organization* parameter indicates the organisation whose effective policy information is to be retrieved; or, if NULL, the RS Policy's policy information is indicated. (If non-NULL, this string is interpreted as a short PGO name, subordinate to the organisation naming domain — see Section 11.2.4 on page 361.)

The *policy_data* parameter indicates the returned effective policy information.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Read (**r**) permission on the object(s) specified by the *organization* parameter (that is, on both organisation and RS Policy object, or just RS Policy object).

11.4.8 rs_auth_policy_get_info()

The *rs_auth_policy_get_info*( ) operation retrieves (reads) an account's authentication policy information (or that of the cell).

```
[idempotent] void
rs_auth_policy_get_info (
    [in]      handle_t                rpc_handle,
    [in]      sec_rgy_login_name_t    *account,
    [out]     sec_rgy_plcy_auth_t     *auth_policy,
    [out]     rs_cache_data_t         *cache_info,
    [out]     error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *account* parameter indicates the account whose authentication policy information is to be retrieved; or if all the fields of *account* are NULL or empty strings the RS Policy's authentication policy information is indicated.

The *auth_policy* parameter indicates the retrieved policy information.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Read (**r**) permission on the object specified by the *account* parameter.

11.4.9 rs_auth_policy_get_effective()

The *rs_auth_policy_get_effective*( ) operation returns an account's effective authentication policy information; that is, for each **sec_rgy_plcy_auth_t** field the more restrictive of the account's authentication policy information and the cell's authentication policy information.

```
[idempotent] void
rs_auth_policy_get_effective (
    [in]    handle_t                rpc_handle,
    [in]    sec_rgy_login_name_t    *account,
    [out]   sec_rgy_plcy_auth_t     *auth_policy,
    [out]   rs_cache_data_t         *cache_info,
    [out]   error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *account* parameter indicates the account whose effective authentication policy information is to be retrieved; or if all the fields of *account* are NULL or empty strings the RS Policy's authentication policy information is indicated.

The *auth_policy* parameter indicates the retrieved policy information.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Read (**r**) permission on the object(s) specified by the *account* parameter (that is, on both account and RS Policy object, or just RS Policy object).

11.4.10 rs_auth_policy_set_info()

The *rs_auth_policy_set_info*( ) operation modifies (writes) an account's authentication policy information (or that of the cell).

```
void
rs_auth_policy_set_info (
    [in]      handle_t                rpc_handle,
    [in]      sec_rgy_login_name_t    *account,
    [in]      sec_rgy_plcy_auth_t     *auth_policy,
    [out]     rs_cache_data_t         *cache_info,
    [out]     error_status_t          *status );
}
/* end running listing of rs_policy interface */
```

The *rpc_handle* parameter identifies the RS server.

The *account* parameter indicates the account whose authentication policy information is to be modified; or if all the fields of *account* are NULL or empty strings the RS Policy's authentication policy information is indicated.

The *auth_policy* parameter indicates the policy information that is to be written.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Authentication Info (**a**) permission on the object specified by the *account* parameter.

11.5 The rs_pgo RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_pgo** RPC interface.

11.5.1 Common Data Types and Constants for rs_pgo

The following are common data types and constants used in the **rs_pgo** interface.

11.5.1.1 sec_rgy_domain_t

The **sec_rgy_domain_t** data type represents the RS datastore's PGO domains. For naming purposes (see Section 11.2.4 on page 361), these domains also have stringnames associated with them (and therefore they are also known as *naming domains*).

```
typedef signed32    sec_rgy_domain_t;
const signed32      sec_rgy_domain_person = 0;
const signed32      sec_rgy_domain_group  = 1;
const signed32      sec_rgy_domain_org    = 2;
```

The following values are currently registered:

- **sec_rgy_domain_person**
  The principal domain. Its associated stringname is **person**.
- **sec_rgy_domain_group**
  The group domain. Its associated stringname is **group**.
- **sec_rgy_domain_org**
  The organisation domain. Its associated stringname is **org**.

Note that three PGO domains are identified in the **rs_pgo** interface described in this section via the indicated **sec_rgy_domain_t** values, not via the indicated stringnames. These stringnames occur only in fully-qualified global names, as used by the **rdacl** RPC interface and **sec_acl** API (see Chapter 10 and Chapter 15).

11.5.1.2 sec_rgy_member_t

The **sec_rgy_member_t** data type represents names in the RS-supported namespace.

```
typedef char    sec_rgy_member_t[sec_rgy_name_t_size];
```

Note: There is no substantive difference between the data types **sec_rgy_name_t** (see Section 11.2.4 on page 361) and **sec_rgy_member_t**. The existence of the two types is best thought of as being historical.

11.5.1.3 sec_rgy_pgo_flags_t

The **sec_rgy_pgo_flags_t** data type represents a flag word of attributes on PGO items.

```
typedef bitset      sec_rgy_pgo_flags_t;
const unsigned32    sec_rgy_pgo_is_an_alias    = 0x1;
const unsigned32    sec_rgy_pgo_is_required    = 0x2;
const unsigned32    sec_rgy_pgo_projlist_ok    = 0x4;
```

The following values are currently registered:

- **sec_rgy_pgo_is_an_alias**
  PGO item is an alias.
- **sec_rgy_pgo_is_required**
  PGO item is required (cannot be deleted).
- **sec_rgy_pgo_projlist_ok**
  For a group item, indicates that the group can occur in a concurrent group set; that is, can occur as a secondary (non-primary) group associated with some principal or account. Has no meaning on a principal item or an organisation item.

11.5.1.4 sec_rgy_pgo_item_t

The **sec_rgy_pgo_item_t** data type represents the RS datastore information associated with PGO items.

```
typedef struct {
    uuid_t                  id;
    signed32                unix_num;
    signed32                quota;
    sec_rgy_pgo_flags_t     flags;
    sec_rgy_pname_t         fullname;
} sec_rgy_pgo_item_t;
```

Its fields are the following:

- **id**
  UUID associated with item. This is the definitive identifier of a PGO item, in the sense that it cannot be changed (see *rs_pgo_replace*( ), Section 11.5.5 on page 385).
- **unix_num**
  Local-ID associated with item.

  Note: The semantics of the **unix_num** field does not depend on the **sec_rgy_prop_embedded_unix_id** property of RS servers (see Section 11.4.1.2 on page 366). The **sec_rgy_prop_embedded_unix_id** property is important only for those environments/applications that need to extract principals' local-IDs from their principal UUIDs. Other environments/applications are advised instead to use the RS's services (namely, *rs_pgo_get*( ), see Section 11.5.7 on page 386) to do the UUID→local-ID mapping, thereby eliminating their dependency on the **sec_rgy_prop_embedded_unix_id** property.
- **quota**
  Quota associated with a principal item (this is not meaningful for group or organisation items).

  The RS server associates to each principal a *quota*; that is, the maximum number of PGO items and accounts (combined) that may be added to the RS datastore by the principal (by using *rs_pgo_add*( ) and/or *rs_acct_add*( )). A quota value of –1 indicates an unlimited quota (no restrictions). If the quota is greater than 0, the quota is decremented on each successful PGO item or account added by the indicated principal. If the quota equals zero, the RS server will fail attempts to add PGO items or accounts (with status **sec_rgy_quota_exhausted**). The only way to increase a quota value is to explicitly modify the quota field (with *rs_pgo_replace*( ) or *rs_properties_set_info*( ); the quota is not incremented on PGO or account deletions operations, for example).
- **flags**
  Flag word associated with item.

- **fullname**
  Human-friendly full name of item — or, for that matter, any other human-friendly informative (annotation) data that may be convenient. For example, a PGO item named **principal/root** might have a **fullname** of ‘‘M. Root, the Superuser’’. (Note that this example is strictly illustrative — there is no notion in DCE Security that corresponds to the traditional notion of superuser as defined in the security architecture of some local systems.)

11.5.1.5 rs_pgo_id_key_t

The **rs_pgo_id_key_t** data type represents the data necessary for a lookup-by-UUID query in the RS datastore.

```
typedef struct {
      uuid_t            id;
      sec_rgy_name_t    scope;
} rs_pgo_id_key_t;
```

Its fields are the following:

- **id**
  UUID of the object being sought (that is, the item to be matched by the query).
- **scope**
  The scope of the lookup-by-UUID. This is either the sought object’s name, or the name of a directory which is an ancestor (not necessarily an immediate parent) of the sought object.

11.5.1.6 rs_pgo_unix_num_key_t

The **rs_pgo_unix_num_key_t** data type represents the data necessary for a lookup-by-local-ID query in the RS datastore.

```
typedef struct {
    signed32            unix_num;
    sec_rgy_name_t      scope;
} rs_pgo_unix_num_key_t;
```

Its fields are the following:

- **unix_num**
  Local-ID of the object being sought (that is, the item to be matched by the query).
- **scope**
  The scope of the lookup-by-local-ID. This is either the sought object’s name, or the name of a directory which is an ancestor (not necessarily an immediate parent) of the sought object.

11.5.1.7 rs_pgo_query_t

The **rs_pgo_query_t** data type indicates the type of an RS datastore query key (see **rs_pgo_query_key_t**, Section 11.5.1.8 on page 382).

```
typedef enum {
    rs_pgo_query_name,          /* 0 */
    rs_pgo_query_id,            /* 1 */
    rs_pgo_query_unix_num,      /* 2 */
    rs_pgo_query_next,          /* 3 */
    rs_pgo_query_none           /* 4 */
} rs_pgo_query_t;
```

The following values are currently registered:

- **rs_pgo_query_name**
  Query keyed by name (**sec_rgy_name_t**).
- **rs_pgo_query_id**
  Query keyed by UUID (**rs_pgo_id_key_t**).
- **rs_pgo_query_unix_num**
  Query keyed by local-ID (**rs_pgo_unix_num_key_t**).
- **rs_pgo_query_next**
  Query keyed by the next PGO item following the current cursor position. (See the discussion at *rs_pgo_get*( ), see Section 11.5.7 on page 386.)
- **rs_pgo_query_none**
  Indicates an empty **rs_pgo_query_key_t** (that is, a non-existent value of this data type). It is used in error cases, to indicate that a item being sought could not be found.

11.5.1.8 rs_pgo_query_key_t

The **rs_pgo_query_key_t** data type represents an RS datastore query (lookup) key.

```
typedef union switch (rs_pgo_query_t query) tagged_union {
    case rs_pgo_query_name:
        sec_rgy_name_t          name;
    case rs_pgo_query_id:
        rs_pgo_id_key_t         id_key;
    case rs_pgo_query_unix_num:
        rs_pgo_unix_num_key_t   unix_num_key;
    case rs_pgo_query_next:
        sec_rgy_name_t          scope;
    default:
        /*empty*/               /*empty*/;
} rs_pgo_query_key_t;
```

Note that the **rs_pgo_query_none** value of the **query** discriminator matches the **default** arm of the union switch.

11.5.1.9 rs_pgo_result_t

The **rs_pgo_result_t** data type represents the result of an RS datastore query (lookup).

```
typedef struct {
      sec_rgy_name_t        name;
      sec_rgy_pgo_item_t    item;
} rs_pgo_result_t;
```

Its fields are the following:

- **name**
  Name of item.
- **item**
  RS datastore information associated with item.

11.5.1.10 rs_pgo_query_result_t

The **rs_pgo_query_result_t** data type represents a performance-optimised version of the **rs_pgo_result_t** data type. In the success case (*status* = **error_status_ok**), **rs_pgo_query_result_t** represents a value of type **rs_pgo_result_t**); in the error case (*status* ≠ **error_status_ok**) it is empty (thereby preventing unnecessary marshalling/unmarshalling of data in the error case).

```
typedef union switch (signed32 status) tagged_union {
    case error_status_ok:
        rs_pgo_result_t        result;
    default:
        /*empty*/              /*empty*/;
} rs_pgo_query_result_t;
```

11.5.2 Interface UUID and Version Number for rs_pgo

The interface UUID and version number for the **rs_pgo** interface are given by the following:

```
[
    uuid(4c878280-3000-0000-0d00-028714000000),
    version(1.0)
]
interface rs_pgo {
/* begin running listing of rs_pgo interface */
```

11.5.3 rs_pgo_add()

The *rs_pgo_add*() operation creates (adds) to the RS's datastore a PGO item that does not currently exist.

```
void
rs_pgo_add (
    [in]        handle_t            rpc_handle,
    [in]        sec_rgy_domain_t    name_domain,
    [in]        sec_rgy_name_t      pgo_name,
    [in]        sec_rgy_pgo_item_t  *pgo_item,
    [out]       rs_cache_data_t     *cache_info,
    [out]       error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *name_domain* parameter identifies the PGO domain in which the new item is to be created.

The *pgo_name* parameter identifies the (name of the) PGO item, subordinate to *name_domain*, that is to be created. Only terminal objects (that is, PGO items per se) are valid named items in this context — not intermediate directories (between the root of *name_domain* and the named PGO item), which are created automatically by this operation if necessary (namely, if they don't already exist), and cannot be created directly.

The *pgo_item* parameter indicates the information that is to be stored in the RS's datastore for the newly created PGO item. The fields of *pgo_item* indicate the following:

- **id**
  If **id** is a nil-valued UUID (see referenced X/Open DCE RPC Specification), the RS server will generate a non-nil value for the UUID of the added item. If the **sec_rgy_prop_embedded_unix_id** property is set for the RS server, the generated UUID will be a security-version UUID (see Section 5.2.1.1 on page 278) and will contain an embedded local-ID that is either passed in as the **unix_num** field or generated by the RS server.

If **id** is non-nil, it will be used as the UUID of the created item. If the **sec_rgy_prop_embedded_unix_id** property is set, the UUID will be inspected for the correct (security) version (see Section 5.2.1.1 on page 278) and checked to ensure that its embedded local-ID matches the **unix_num** field (if **unix_num** is not specified, it will be derived from the embedded local-ID of **id**).

- **unix_num**
  If **unix_num** = –1, the RS server will generate the local-ID of the created item; or, if the RS server's **sec_rgy_prop_embedded_unix_id** property is set and the **id** field is non-nil, the local-ID of the created item will be extracted from **id**.

  If **unix_num** ≠ –1, it will be used as the local-ID of the created item. If the RS server's **sec_rgy_prop_embedded_unix_id** property is set and the **id** field is non-nil, then **unix_num** must match the local-ID embedded in **id**.

- **quota**
  The quota for created principal items. (Has no meaning for group or organisation items.)

- **flags**
  Flag word associated with this PGO item.

- **fullname**
  Human-friendly full name associated with this PGO item.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Insert (**i**) permission on the parent container of the created PGO item (that is, the container in which the PGO item is to be created).

11.5.4 rs_pgo_delete()

The *rs_pgo_delete*( ) operation deletes from the RS's datastore an existing PGO item, together with all accounts depending on that PGO item (that is, having that PGO item as an element).

```
void
rs_pgo_delete (
    [in]        handle_t            rpc_handle,
    [in]        sec_rgy_domain_t    name_domain,
    [in]        sec_rgy_name_t      pgo_name,
    [out]       rs_cache_data_t     *cache_info,
    [out]       error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *name_domain* parameter identifies the PGO domain from which the PGO item is to be deleted.

The *pgo_name* parameter identifies the PGO item, subordinate to *name_domain*, to be deleted. Only terminal objects (that is, PGO items per se) are valid named items in this context — not intermediate directories (between the root of *name_domain* and the named item), which are deleted automatically by this operation if necessary (namely, if this operation deletes the last entry subordinate to them), and cannot be deleted directly.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Delete Item (**D**) permission on the PGO item to be deleted, and Delete (**d**) permission on the parent container of that PGO item.

11.5.5 rs_pgo_replace()

The *rs_pgo_replace*( ) operation modifies (writes) an existing PGO item in the RS's datastore.

```
void
rs_pgo_replace (
    [in]        handle_t            rpc_handle,
    [in]        sec_rgy_domain_t    name_domain,
    [in]        sec_rgy_name_t      pgo_name,
    [in]        sec_rgy_pgo_item_t  *pgo_item,
    [out]       rs_cache_data_t     *cache_info,
    [out]       error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *name_domain* parameter identifies the PGO domain of the PGO item which is to be modified.

The *pgo_name* parameter identifies the PGO item, subordinate to *name_domain*, to be modified.

The *pgo_item* parameter indicates the new information that is to be written in the RS's datastore for the indicated PGO item. The fields of *pgo_item* that are generally modifiable are: **quota**, **flags**, and **fullname**. In general, the **id** field (and hence the **unix_num** field) cannot be modified by *rs_pgo_replace*( ) (such an effect could be achieved through a combination of *rs_pgo_delete*( ) and *rs_pgo_add*( ), but it is inadvisable). (This lack of modifiability of the UUID is the sense in which the UUID is the definitive identifier of the PGO item.)

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: If **quota** or **flags** or **unix_num** is being changed, this operation succeeds only if the calling client has Management Info (**m**) permission on the PGO item. If **fullname** is being changed, this operation succeeds only if the calling client has Fullname (**f**) permission on the PGO item.

11.5.6 rs_pgo_rename()

The *rs_pgo_rename*( ) operation modifies the name of a PGO item in the RS's datastore. This operation can modify any portion of a PGO item's name (either its leaf portion, or move the PGO item from one container to another), but it is limited to operate within a single PGO domain.

```
void
rs_pgo_rename (
    [in]        handle_t            rpc_handle,
    [in]        sec_rgy_domain_t    name_domain,
    [in]        sec_rgy_name_t      old_name,
    [in]        sec_rgy_name_t      new_name,
    [out]       rs_cache_data_t     *cache_info,
    [out]       error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *name_domain* parameter identifies the PGO domain of the PGO item which is to be renamed.

The *old_name* parameter identifies the PGO item, subordinate to *name_domain*, whose name is to be modified.

The *new_name* parameter indicates the new name of the PGO item whose name is being modified.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Name (**n**) permission on the PGO item. If only the last component of the PGO item's name is being changed, the calling client requires no other permissions. If the PGO item is being moved from one container to another (that is, if some part of the PGO item's name other than just its last component is being changed), this operation succeeds only if the calling client has in addition Delete (**d**) permission on the parent container of *old_name*, and Insert (**i**) permission on the parent container of *new_name*.

11.5.7 rs_pgo_get()

The *rs_pgo_get*( ) operation searches for and retrieves (reads) data from an RS server/datastore. This operation returns the datastore information of the next PGO item (with respect to the RS server/datastore's implementation-dependent ordering of PGO items) at or following a specified cursor position, whose stored data matches that indicated by a specified query (lookup) key (where the notion of matching is query-key-specific, as defined below in this section).

```
[idempotent] void
rs_pgo_get (
    [in]        handle_t                rpc_handle,
    [in]        sec_rgy_domain_t        name_domain,
    [in]        rs_pgo_query_key_t      *key,
    [in]        boolean32               allow_aliases,
    [in, out]   sec_rgy_cursor_t        *item_cursor,
    [out]       rs_cache_data_t         *cache_info,
    [out]       rs_pgo_query_result_t   *result );
```

The *rpc_handle* parameter identifies the RS server.

The *name_domain* parameter identifies the PGO domain to be searched. (Searches are limited to a single domain per call.)

The *key* parameter identifies the query (or lookup) key on which the search is to be based. Namely, the **query** discriminator of *key* (see Section 11.5.1.8 on page 382) takes one of the following values:

- **rs_pgo_query_name**
  This indicates a lookup-by-name search. The PGO item (there can be at most one) at or following *item_cursor* and having the name specified by **(*****key***).tagged_union.name** is to be matched. (No scope information is relevant in this case.)
- **rs_pgo_query_id**
  This indicates a lookup-by-UUID search. The next PGO item at or following *item_cursor* and having the UUID specified by **(*****key***).tagged_union.id_key.id**, whose name lies within the scope specified by **(*****key***).tagged_union.id_key.scope**, is to be matched. Called successively with the same UUID, *rs_pgo_get*( ) returns first the distinguished (non-alias) item, then all alias items matching this UUID.
- **rs_pgo_query_unix_num**
  This indicates a lookup-by-local-ID search. The next PGO item at or following *item_cursor*

and having the local-ID specified by **(****key***).tagged_union.unix_num_key.unix_num**, whose name lies within the scope specified by **(****key***).tagged_union.unix_num_key.scope**, is to be matched. Called successively with the same local-ID, *rs_pgo_get*( ) returns first the distinguished (non-alias) item, then all alias items matching this local-ID.

- **rs_pgo_query_next**
  This indicates that the next PGO item at or following *item_cursor*, and whose name lies within the scope specified by **(****key***).tagged_union.scope**, is to be matched. Thus, successive queries of this type retrieve all PGO items whose names lie within the indicated scope (or within the entire indicated *name_domain*, in the case where the scope is the NULL string, indicating the root directory of *name_domain*).

The *allow_aliases* parameter, if non-0 (TRUE), indicates that an alias PGO item matching *key* satisfies the query request. If 0 (FALSE), only a distinguished (that is, non-alias) PGO item can satisfy the query request.

On input, the *item_cursor* parameter indicates the current cursor position (so that the PGO item it indicates is eligible to be matched, as are all the items following it); on output it indicates the cursor position next following the retrieved PGO item. The *item_cursor* does not automatically wrap around from the end of the datastore to its beginning: instead, the **sec_rgy_no_more_entries** status value is returned in the *result* parameter if the requested item is not matched between the current cursor position and the end of the datastore (and *item_cursor* remains indicating the end of the datastore in this case).

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *result* parameter returns the retrieved information for the matched PGO item.

The status of this operation is indicated by the **result** element of the *result* parameter.

The ritual for using *rs_pgo_get*( ) is as follows. Before making the first *rs_pgo_get*( ) call, the *item_cursor* parameter must be initialised to the beginning of the RS's datastore (modulo any scoping information), by setting the ******item_cursor***.valid** field to 0 (FALSE); the other fields of ******item_cursor* may be set to any convenient value. In the returned value of *item_cursor*, the RS server will set values that it (and only it) can interpret. The client uses this returned value of *item_cursor*, without modifying it, for a subsequent call. This ritual may be followed for a sequence of successive calls, until the end of the datastore is reached (that is, **sec_rgy_no_more_entries** status value is returned). In any such sequence of calls, the same *name_domain* and *key* parameters must be used (otherwise, the results are undefined).

Required rights: This operation succeeds only if the calling client has Read (**r**) permission on the matched PGO item.

11.5.8 rs_pgo_key_transfer( )

The *rs_pgo_key_transfer*( ) operation converts one of an RS datastore item's query keys (PGO item's name, UUID or local-ID) into another. Using this operation may be more efficient and/or convenient than using *rs_pgo_get*( ) when one query key is known, another is desired, and other datastore information is not needed. In other words, this operation implements the six mappings: name→UUID, UUID→name, name→local-ID, local-ID→name, UUID→local-ID, and local-ID→UUID.

```
[idempotent] void
rs_pgo_key_transfer (
    [in]        handle_t            rpc_handle,
    [in]        sec_rgy_domain_t    name_domain,
    [in]        rs_pgo_query_t      requested_result_type,
    [in, out]   rs_pgo_query_key_t  *key,
    [out]       rs_cache_data_t     *cache_info,
    [out]       error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *name_domain* parameter identifies the PGO domain to be searched.

The *requested_result_type* parameter indicates the query key to which *key* is to be converted.

On input, the *key* parameter indicates a query key which is to be converted. On output, it indicates the converted query key.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: In the cases where no name query key is involved (that is, the UUID→local-ID and local-ID→UUID cases), this operation succeeds unconditionally. In the cases where a name query key is involved (that is, the name→UUID, UUID→name, name→local-ID and local-ID→name cases), this operation succeeds only if the calling client has at least *some* (any) access permission to the PGO item.

11.5.9 rs_pgo_add_member( )

The *rs_pgo_add_member*( ) operation adds a member principal to the membership list of a group or organisation.

```
void
rs_pgo_add_member (
    [in]        handle_t            rpc_handle,
    [in]        sec_rgy_domain_t    name_domain,
    [in]        sec_rgy_name_t      go_name,
    [in]        sec_rgy_name_t      person_name,
    [out]       rs_cache_data_t     *cache_info,
    [out]       error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *name_domain* parameter indicates the domain of interest (it must be **sec_rgy_domain_group** or **sec_rgy_domain_org**; it may not be **sec_rgy_domain_person**).

The *go_name* parameter indicates the group or organisation item, subordinate to *name_domain*, to which the member is to be added.

The *person_name* parameter indicates the principal item to be added to the *go_name* item.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Membership (**M**) permission on the item indicated by *go_name*. If further *go_name* is a group (as opposed to an organisation), then the calling client must also have Group (**g**) permission on the principal indicated by *person_name*.

11.5.10 rs_pgo_delete_member( )

The *rs_pgo_delete_member*( ) operation deletes a member principal from the membership list of a group or organisation, together with all accounts depending on that member (that is, having the specified principal, and group or organisation).

```
void
rs_pgo_delete_member (
    [in]        handle_t            rpc_handle,
    [in]        sec_rgy_domain_t    name_domain,
    [in]        sec_rgy_name_t      go_name,
    [in]        sec_rgy_name_t      person_name,
    [out]       rs_cache_data_t     *cache_info,
    [out]       error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *name_domain* parameter indicates the domain of interest (it must be **sec_rgy_domain_group** or **sec_rgy_domain_org**; it may not be **sec_rgy_domain_person**).

The *go_name* parameter indicates the group or organisation item, subordinate to *name_domain*, from which the member is to be deleted.

The *person_name* parameter indicates the principal item to be deleted from the *go_name* item.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Membership (**M**) permission on the item indicated by *go_name*.

11.5.11 rs_pgo_is_member( )

The *rs_pgo_is_member*( ) operation determines whether a principal is a member of a group or organisation.

```
[idempotent] boolean32
rs_pgo_is_member (
    [in]        handle_t            rpc_handle,
    [in]        sec_rgy_domain_t    name_domain,
    [in]        sec_rgy_name_t      go_name,
    [in]        sec_rgy_name_t      person_name,
    [out]       rs_cache_data_t     *cache_info,
    [out]       error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *name_domain* parameter indicates the domain of interest (it must be **sec_rgy_domain_group** or **sec_rgy_domain_org**; it may not be **sec_rgy_domain_person**).

The *go_name* parameter indicates the name of the group or organisation item, subordinate to *name_domain*, for which membership is being queried.

The *person_name* parameter indicates the principal item whose membership in the *go_name* item is being queried.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

The **boolean32** return value returns non-0 (TRUE) if *person_name* is a member of *go_name*, and 0 (FALSE) otherwise.

Required rights: This operation succeeds only if the calling client has Test (**t**) permission on the item indicated by *go_name*.

11.5.12 rs_pgo_get_members( )

The *rs_pgo_get_members*( ) operation retrieves the list of member principals of a group or organisation, or the list of groups to which a principal belongs.

```
[idempotent] void
rs_pgo_get_members (
    [in]        handle_t                    rpc_handle,
    [in]        sec_rgy_domain_t            name_domain,
    [in]        sec_rgy_name_t              go_name,
    [in, out]   sec_rgy_cursor_t            *member_cursor,
    [in]        signed32                    max_members,
    [out, length_is(*number_supplied), size_is(max_members)]
                sec_rgy_member_t            member_list[ ],
    [out]       signed32                    *number_supplied,
    [out]       signed32                    *number_members,
    [out]       rs_cache_data_t             *cache_info,
    [out]       error_status_t              *status );

} /* end running listing of rs_pgo interface */
```

The *rpc_handle* parameter identifies the RS server.

The *name_domain* parameter indicates the PGO domain of interest. If *name_domain* is **sec_rgy_domain_group** or **sec_rgy_domain_org**, the principals which are members of the group or organisation indicated by *go_name* are to be returned in *member_list*[ ]. If *name_domain* is **sec_rgy_domain_person**, the groups of which the principal indicated by *go_name* is a member are to be returned in *member_list*[ ].

The *go_name* parameter indicates the PGO item, subordinate to *name_domain*, whose membership is being queried.

The *member_cursor* parameter indicates, on input, the current cursor position (so that the items at and following this position are eligible to be retrieved on the current invocation of this operation); on output, it indicates the cursor position next following the retrieved item(s).

The *max_members* parameter indicates the maximum number of members to be retrieved in *member_list*[ ].

The *member_list* parameter indicates the retrieved members.

The *number_supplied* parameter indicates the actual number of retrieved members.

The *number_members* parameter indicates the total number of members available for the *go_name* item. (The *member_cursor* parameter is used to coordinate multiple invocations to retrieve the complete list of members.)

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Read (**r**) permission on the *go_name* item.

11.6 The rs_acct RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_acct** RPC interface.

11.6.1 Common Data Types and Constants for rs_acct

The following are common data types and constants used in the **rs_acct** interface.

11.6.1.1 sec_rgy_acct_key_t

The **sec_rgy_acct_key_t** data type indicates the *minimal portion* of an account's <P, G, O> name triple that is required to unambiguously identify the account — that is, the minimal name information (or abbreviation) that constitutes an RS datastore query key for the account.

```
typedef signed32    sec_rgy_acct_key_t;
const signed32      sec_rgy_acct_key_person    = 1;
const signed32      sec_rgy_acct_key_group     = 2;
const signed32      sec_rgy_acct_key_org       = 3;
```

The following values are currently registered:

- **sec_rgy_acct_key_person**
  The principal name identifies the account; that is, the account (<P, G, O>) is uniquely determined by its <P> component.

  This is the only value required to be supported by this revision of DCE. (That is, an RS datastore in which no principal is associated with more than one account is conformant with DCE.)

- **sec_rgy_acct_key_group**
  The principal and group names identify the account; that is, the account (<P, G, O>) is uniquely determined by its <P, G> component pair — and moreover this is a minimal query key (that is, the account's <P> component does not constitute a query key).

- **sec_rgy_acct_key_org**
  The principal, group and organisation names identify the account — and moreover this is a minimal query key (that is, the account's <P, G> component pair does not constitute a query key).

11.6.1.2 sec_rgy_acct_admin_flags_t

The **sec_rgy_acct_admin_flags_t** data type is a flag word representing administrative flags.

```
typedef bitset          sec_rgy_acct_admin_flags_t;
const unsigned32        sec_rgy_acct_admin_valid    = 0x1;
const unsigned32        sec_rgy_acct_admin_server   = 0x4;
const unsigned32        sec_rgy_acct_admin_client   = 0x8;
```

The following values are currently registered:

- **sec_rgy_acct_admin_valid**
  Account is valid for logging in.

- **sec_rgy_acct_admin_server**
  Allow account to be a server. (That is, this account's principal name may be used as a targeted server name in tickets issued in this cell.)

- **sec_rgy_acct_admin_client**
  Allow account to be a client. (That is, this account's principal name may be used as a named client name in tickets issued in this cell.)

11.6.1.3 sec_rgy_acct_auth_flags_t

The **sec_rgy_acct_auth_flags_t** data type is a flag word representing account authentication flags.

```
typedef  bitset     sec_rgy_acct_auth_flags_t;
const unsigned32    sec_rgy_acct_auth_tgt        = 0x4;
```

The following values are currently registered:

- **sec_rgy_acct_auth_tgt**
  Allow issuance of certificates based on (privilege-)ticket-granting-ticket authentication. This must always be set (to 1).

11.6.1.4 sec_rgy_foreign_id_t

The **sec_rgy_foreign_id_t** data type represents identities (principals, groups and organisations, despite the field name **principal** in the defining structure below) from arbitrary cells (including the local interpreting cell; that is, not necessarily a strictly foreign non-local cell). This data type is defined as follows (compare to the **sec_id_foreign_t** data type, Section 5.2.2 on page 279):

```
typedef struct sec_rgy_foreign_id_t {
    uuid_t      principal;
    uuid_t      cell;
} sec_rgy_foreign_id_t;
```

Its fields are the following:

- **principal**
  The UUID of the principal, within its cell.
- **cell**
  The UUID of the cell.

11.6.1.5 sec_rgy_acct_admin_t

The **sec_rgy_acct_admin_t** data type represents administration-level information about accounts held in the RS datastore.

```
typedef struct {
    sec_rgy_foreign_id_t          creator;
    sec_timeval_sec_t             creation_date;
    sec_rgy_foreign_id_t          last_changer;
    sec_timeval_sec_t             change_date;
    sec_timeval_sec_t             expiration_date;
    sec_timeval_sec_t             good_since_date;
    sec_rgy_acct_admin_flags_t    flags;
    sec_rgy_acct_auth_flags_t     authentication_flags;
} sec_rgy_acct_admin_t;
```

Its fields are the following:

- **creator**
  Identity of the principal who created this account. (Set by the RS server, not directly by the administrator.)
- **creation_date**
  Time of creation of this account. (Set by the RS server, not directly by the administrator.)

- **last_changer**
  Identity of the last principal to modify this account. (Set by the RS server, not directly by the administrator.)
- **change_date**
  Time of last modification of this account. (Set by the RS server, not directly by the administrator.)
- **expiration_date**
  Last date of validity of this account. (Typically, this indicates a date by which a reactivation of the account must be performed; for example, its password must be changed.)
- **good_since_date**
  First date of validity of this account. (The KDS will not honour tickets issued before this date.)
- **flags**
  Flag word for this account.
- **authentication_flags**
  Authentication flag word for this account.

11.6.1.6 sec_rgy_acct_user_flags_t

The **sec_rgy_acct_user_flags_t** data type represents a flag word of attributes about users.

```
typedef  bitset    sec_rgy_acct_user_flags_t;
const unsigned32   sec_rgy_acct_user_passwd_valid   = 0x1;
```

The following values are currently registered:

- **sec_rgy_acct_user_passwd_valid**
  This password is good. The absence of this bit forces the user to change his or her password.

11.6.1.7 sec_passwd_type_t

The **sec_passwd_type_t** data type represents password type (either a true password to be mapped to a cryptographic key, or else a cryptographic key; for usage, see Section 11.6.1.11 on page 395).

```
typedef enum {
    sec_passwd_none,     /* 0 */
    sec_passwd_plain,    /* 1 */
    sec_passwd_des       /* 2 */
} sec_passwd_type_t;
```

The following values are currently registered:

- **sec_passwd_none**
  Invalid password / key.
- **sec_passwd_plain**
  Plaintext password.
- **sec_passwd_des**
  DES key.

11.6.1.8 sec_key_version_t

The **sec_key_version_t** data type indicates the version number of a long-term cryptographic key associated to an account held in the RS datastore. It is used in identifying the version number of uninterpreted byte strings of encrypted network data. See Section 11.6.1.20 on page 399 for its usage.

```
typedef unsigned32    sec_key_version_t;
```

11.6.1.9 sec_passwd_version_t

The **sec_passwd_version_t** data type indicates the version number of a long-term cryptographic key (and hence, by extension, its associated password, if there is one) associated to an account held in the RS datastore.

```
typedef  unsigned32    sec_passwd_version_t;
const    unsigned32    sec_c_key_version_none     = 0;
const    unsigned32    sec_passwd_c_version_none  = 0;
```

Key version numbers have the following semantics. Every account has zero or more keys associated with it; every valid account has exactly one key associated with it, with the potential (notable) exceptions of the RS (**dce-rgy**), PS (**dce-ptgt**) and KDS (or cell, **krbtgt/***cell-name*) principals in every cell. (Typically, implementations will support multiple simultaneous keys for these principals, so that their keys can be updated regularly without invalidating the many unexpired (ticket-granting) tickets that have been issued (protected) under previous keys; when such an update occurs, the old key is retained until the account's maximum renewable ticket lifetime is reached (see Section 11.4.1.6 on page 370), at which time it can be discarded.) Every such key has one and only one version number attached to it (a value of type **sec_passwd_version_t**). All the keys associated with a given account must all be distinct, and must all have different key version numbers attached to them. No key may ever have the version number **sec_c_key_version_none**. These key version numbers are used to distinguish among the various keys associated with an account, as follows.

Under normal circumstances (that is, in the stable operating configuration), an account has exactly one key associated with it. But due to security considerations (namely, the longer a key has been used, and the greater the volume of traffic it has encrypted, the more susceptible it is to compromise), the value of this key needs to be changed from time to time. (For a human user, this is manifested as changing the user's password; for a non-human server, it is manifested as changing the server's key.) However, when a new key is associated with an account, the old key needs to be retained until it times out; that is, until all the tickets that could legitimately be protected by the old key have expired. Thus, KDSs must always use accounts' *most recent* keys to protect all tickets. Finally, all KDSs must implement the successive versions of keys by means of *strictly increasing* version numbers (typically, incrementing by 1 for each new version number), and the most recent key must always be that having the *highest* version number (no provision is specified here for overflow of the **sec_passwd_version_t** data type — note, however, that keys are typically changed at most once an hour, and that 2^{32} hours ≈ 490,000 years, so the likelihood of colliding key version numbers is insignificant).

Note: The condition given here (namely, that the most recent key must correspond to the highest version number, and the related restriction to strictly increasing version numbers) might reasonably be expected to be implementation details, as opposed to specification requirements. Nevertheless, the description as given here is consistent with the Kerberos specification. [RFC 1510: 4.1]

The following values are currently registered:

- **sec_c_key_version_none**
  Invalid key. This is a reserved value to be used in certain service requests; for example, in key lookup requests for the current key whose version number is not known *a priori*, or in key update requests where the next available version number is to be assigned to the new key.
- **sec_passwd_c_version_none**
  Invalid password. This is a reserved value to be used in certain service requests; for example, in password initialization for cell initialization when a (new) cell is being added, or when initializing the registry information.

11.6.1.10 sec_passwd_des_key_t

The **sec_passwd_des_key_t** data type represents a (64-bit) DES key.

```
const unsigned32  sec_passwd_c_des_key_size = 8;
typedef byte      sec_passwd_des_key_t[sec_passwd_c_des_key_size];
```

The mapping of bit-vectors to byte-vectors is the usual one (see Section 2.1.3 on page 128), namely: if $K = \langle k_0, \cdots, k_{63} \rangle$ is a 64-bit DES key, then it is represented as the 8-byte **sec_passwd_des_key_t** vector $\langle\langle k_0, \cdots, k_7\rangle, \cdots, \langle k_{56}, \cdots, k_{63}\rangle\rangle$.

11.6.1.11 sec_passwd_rec_t

The **sec_passwd_rec_t** data type represents a password or cryptographic key record.

```
typedef struct {
    sec_passwd_version_t            version_number;
    [string, ptr] char              *pepper;
    union switch (sec_passwd_type_t key_type) {
        case sec_passwd_plain:
            [string, ptr] char      *plain;
        case sec_passwd_des:
            sec_passwd_des_key_t    des_key;
    } key;
} sec_passwd_rec_t;
```

Its fields are the following:

- **version_number**
  Version number.
- **pepper**
  A string used to modify the password-to-key generation algorithm (see Section 4.3.6.1 on page 190, where it was called **salt**).
- **key**
  Either a plaintext password (**plain**) or a DES key (**des_key**), depending on whether **key_type** is equal to **sec_passwd_plain** or **sec_passwd_des**, respectively.

For a description of how this data type is used, see Section 11.6.1.21 on page 400.

11.6.1.12 sec_chksum_type_t

The **sec_chksum_type_t** data type represents checksum type.

```
typedef enum {
    sec_chksum_none,          /* 0 */
    sec_chksum_crc32,         /* 1 */
    sec_chksum_des_cbc,       /* 2 */
    sec_chksum_rsa_md4,       /* 3 */
    sec_chksum_rsa_md4_des    /* 4 */
} sec_chksum_type_t;
```

The following values are currently registered:

- **sec_chksum_none**
  Invalid checksum.
- **sec_chksum_crc32**
  $CRC^{\S}_{G}$ checksum, with seed 0 (see Section 2.2 on page 136).
- **sec_chksum_rsa_md4**
  RSA-MD4-CKSUM which is non-invertible and of length 128 bits, derived from an arbitrary length bit-message (see Chapter 2 on page 127 and Section 1.3 on page 16).
- **sec_chksum_rsa_md4_des**
  RSA-MD4-DES-CKSUM which is non-invertible and of length 128 bits, derived from an arbitrary length bit-message by prepending an 8 octet confounder and applying the RSA-MD4-CKSUM, and with initialisation vector zero (see Chapter 2 on page 127 and Section 1.3 on page 16).
- **sec_chksum_des_cbc**
  DES-CBC-CKSUM, with a specified key, and with initialisation vector equal to the same key (see Section 3.3 on page 150).

11.6.1.13 sec_chksum_t

The **sec_chksum_t** data type represents a checksum.

```
typedef struct {
    sec_chksum_type_t           chksum_type;
    unsigned32                  len;
    [size_is(len), ptr] byte    *chksum;
} sec_chksum_t;
```

Its fields are the following:

- **chksum_type**
  Type of checksum data.
- **len**
  Length of checksum data. This is 0 when **chksum_type** is **sec_chksum_none**; 4 when **chksum_type** is **sec_chksum_crc32**; 8 when **chksum_type** is **sec_chksum_des_cbc**; 16 when **chksum_type** is either **sec_chksum_rsa_md4** or **sec_chksum_rsa_md4_des**.
- **chksum**
  The checksum data itself.

For a description of how this data type is used, see Section 11.6.1.21 on page 400.

11.6.1.14 sec_rgy_unix_passwd_buf_t

The **sec_rgy_unix_passwd_t** data type represents a (protected) (local) password (as opposed to a long-term key); that is, one whose exact interpretation (for example, the manner of its protection, or its usage by local operating systems) is implementation-specific (and whose further specification is therefore beyond the scope of DCE). (The use of the substring **unix** in the name of this data type is historical.) (Protected passwords are sometimes said to be encrypted, but this is usually a misnomer because passwords are usually protected by non-invertible cryptographic checksum techniques, not by invertible encryption / decryption techniques.)

```
const unsigned32 sec_rgy_max_unix_passwd_len = 16;
typedef [string] char
     sec_rgy_unix_passwd_buf_t[sec_rgy_max_unix_passwd_len];
```

11.6.1.15 sec_rgy_acct_user_t

The **sec_rgy_acct_user_t** data type represents user-level information about accounts held in the RS datastore.

```
typedef struct {
    sec_rgy_pname_t             gecos;
    sec_rgy_pname_t             homedir;
    sec_rgy_pname_t             shell;
    sec_passwd_version_t        passwd_version_number;
    sec_timeval_sec_t           passwd_dtm;
    sec_rgy_acct_user_flags_t   flags;
    sec_rgy_unix_passwd_buf_t   passwd;
} sec_rgy_acct_user_t;
```

Its fields are the following:

- **gecos**
  An annotation field, holding any convenient information (similar to the **fullname** field of **sec_rgy_pgo_item_t**).
- **homedir**
  An annotation field, holding any convenient information (typically, this field is interpreted as a user's home directory, in the sense of a POSIX-conformant operating system).
- **shell**
  An annotation field, holding any convenient information (typically, this field is interpreted as a user's login shell, in the sense of a POSIX-conformant operating system).
- **passwd_version_number**
  Version number of long-term cryptographic key. (This number is assigned by the RS server, not by an administrator.)
- **passwd_dtm**
  Time of last password / key update (see Section 11.4.1.5 on page 368). (This number is assigned by the RS server, not by an administrator.)
- **flags**
  User flag word.
- **passwd**
  User's (protected) local password. Its semantics are implementation-dependent. Typical implementations use it to support UNIX password management, as follows.

If the *key* input parameter to *rs_acct_replace*( ) (see Section 11.6.7 on page 405) has **key_type** (in the sense of Section 11.6.1.21 on page 400) **sec_key_plain** (that is, the **key** field of the *key* parameter contains a plaintext password instead of a DES key), then the RS will use that plaintext and a **salt** (in the sense of UNIX password management) to generate a UNIX key via *crypt*( ).) (If a DES key is passed instead of a plaintext password, the RS will copy the fixed string **CIPHER** into this UNIX key.)

This field is ignored on input to *rs_acct_add*( ) or *rs_acct_replace*( ), but it contains the UNIX key just described on output from *rs_acct_lookup*( ). Since this **passwd** field is not used by the authentication services specified in DCE (it merely provides some compatibility and coexistence with the UNIX operating system to clients that care about such things), further details are not relevant to DCE.

11.6.1.16 rs_acct_parts_t

The **rs_acct_parts_t** data type is a flag word used to indicate portions of an account's datastore information. In the current revision of DCE, the only place this is used is in conjunction with the *rs_acct_replace*( ) operation (see Section 11.6.7 on page 405), where it indicates what parts of the account's information are being updated by a given invocation of *rs_acct_replace*( ). (This allows, for example, multiple partial APIs (unspecified in this revision of DCE) to be layered over the single complete RPC *rs_acct_replace*( ) operation.)

```
typedef  bitset     rs_acct_parts_t;
const unsigned32    rs_acct_part_user       = 0x1;
const unsigned32    rs_acct_part_admin      = 0x2;
const unsigned32    rs_acct_part_passwd     = 0x4;
const unsigned32    rs_acct_part_login_name = 0x10;
```

The following values are currently registered (see Section 11.6.7 on page 405 for details):

- **rs_acct_part_user**
  Indicates user-level information.
- **rs_acct_part_admin**
  Indicates administrative-level information.
- **rs_acct_part_passwd**
  Indicates password / key information.
- **rs_acct_part_login_name**
  Indicates datastore query key information.

11.6.1.17 rs_encrypted_pickle_t

The **rs_encrypted_pickle_t** data type represents a cryptographically encrypted pickle (see Section 2.1.7 on page 132 for the definition of pickles). (The encryption type and key are not specified here, but must be specified by other means in any application of this data type.)

```
typedef struct {
    unsigned32                              enc_pickle_len;
    [ref, size_is(enc_pickle_len)] byte     *enc_pickle;
} rs_encrypted_pickle_t;
```

Its fields are the following:

- **enc_pickle_len**
  The number of bytes of encrypted data.

- **enc_pickle**
  The encrypted pickle itself.

11.6.1.18 sec_etype_t

The **sec_etype_t** data type indicates encryption type.

```
typedef enum {
    sec_etype_none,          /* 0 */
    sec_etype_des_cbc_crc    /* 1 */
} sec_etype_t;
```

The following values are currently registered:

- **sec_etype_none**
  Trivial encryption, as described in Section 4.3.5.1 on page 188 (**encType-TRIVIAL**).
- **sec_etype_des_cbc_crc**
  DES-CBC-CRC encryption, as described in Section 4.3.5.1 on page 188 (**encType-DES-CBC-CRC**).

11.6.1.19 sec_bytes_t

The **sec_bytes_t** data type represents a generic pickle type for uninterpreted byte strings of network data.

```
typedef struct {
    unsigned32                          num_bytes;
    [size_is(num_bytes), ptr] byte      *bytes;
} sec_bytes_t;
```

Its fields are the following:

- **num_bytes**
  The number of bytes of network data.
- **bytes**
  The bytes of network data. This network data is a pickle of some sort.

11.6.1.20 sec_encrypted_bytes_t

The **sec_encrypted_bytes_t** data type represents a generic pickle type for encrypting byte strings of network data.

```
typedef struct  {
    sec_etype_t             etype;
    sec_key_version_t       ekvno;
    sec_bytes_t             ebytes;
} sec_encrypted_bytes_t;
```

Its fields are the following:

- **etype**
  An encryption type for encrypting data. The supported **etype**'s are enumerated in Section 11.6.1.18. The **etype** is used in conjunction with the session key type and is extracted from the global cryptosystem information. Presently, only one encryption type is supported.
- **ekvno**
  A version number representing the generic encryption type. Presently, the only version number supported for this generic type is 0.

- **ebytes**
  The actual bytes of encrypted data that are available to servers.

11.6.1.21 rs_acct_key_transmit_t

The **rs_acct_key_transmit_t** data type represents cryptographic keying information (cryptographic key or DES key), protected for transmittal in communications.

```
typedef struct {
    sec_etype_t                 enc_type;
    sec_passwd_type_t           enc_keytype;
    sec_passwd_version_t        enc_key_version;
    unsigned32                  key_pickle_len;
    [ref] rs_encrypted_pickle_t *key;
    [ref] rs_encrypted_pickle_t *checksum;
} rs_acct_key_transmit_t;
```

Its fields are the following (for further elucidation of all these fields and how they are used, see below):

- **enc_type**
  Encryption type used by the client to encrypt **key** and **checksum**.
- **enc_keytype**
  Key type of the client's key, used by the client to encrypt **key** and **checksum**.
- **enc_key_version**
  Key version number of the client's key, used by the client to encrypt **key** and **checksum**.
- **key_pickle_len**
  Length (in bytes) of the (unencrypted) **sec_passwd_rec_t** pickle indicated by **key** — as opposed to the length of the encrypted pickle (which is already determined by **key** itself), which might include some padding appended to the unencrypted **sec_passwd_rec_t** pickle, depending on the encryption algorithm used.
- **key**
  Encrypted **sec_passwd_rec_t** pickle. In the terminology and notation of Section 2.1.7 on page 132, this (pre-encrypted) pickle's type UUID (*H***.pkl_type**) is d52ef390-49da-11ca-b2ac-08001e022936, and its body datastream is an NDR-marshalled **sec_passwd_rec_t**.
- **checksum**
  Encrypted **sec_chksum_t** pickle. In the terminology and notation of Section 2.1.7 on page 132, this (pre-encrypted) pickle's type UUID (*H***.pkl_type**) is d20b05c8-49da-11ca-996d-08001e022936, and its body datastream is an NDR-marshalled **sec_chksum_t**.

This data type is used (in *rs_acct_add*( ) and *rs_acct_replace*( )) as follows.

A principal (acting as an RS RPC client) stores keying information (password or cryptographic key) in a **sec_passwd_rec_t** record (see Section 11.6.1.11 on page 395), and pickles this **sec_passwd_rec_t**. The principal then encrypts the **sec_passwd_rec_t** pickle (using **sec_chksum_none** when **enc_type sec_etype_none**, and using **sec_chksum_des_cbc** when **enc_type** = **sec_etype_des_cbc_crc** — see Section 11.6.1.12 on page 396 and Section 11.6.1.13 on page 396) in its (the principal's) own long-term key — this encrypted pickle is the **key** field. The principal also computes the checksum over the (unencrypted) **sec_passwd_rec_t** pickle (using **cksumType-TRIVIAL** when **enc_type** = **sec_etype_none**, and using **DES-CBC-CKSUM** with initialisation vector 0 when **enc_type** = **sec_etype_des_cbc_crc** — see Section 3.3 on page 150 and Section 4.3.4.1 on page 185), stores that in a **sec_chksum_t** data type, pickles it, and encrypts that pickle in its long-term key — this encrypted pickle is the **checksum** field. The client stores

the encrypting information used for these encryptions (that is, information about its own long-term key) in the **enc_type**, **enc_keytype**, **enc_key_version** and in **key_pickle_len** fields.

On the receiving end, the RS server retrieves the client principal's (authenticated) identity from the protected RPC runtime (see Chapter 9), retrieves the client's long-term key using this identity and the **enc_type**, **enc_keytype**, **enc_key_version** fields of the **rs_acct_key_transmit_t**, then uses this information and **key_pickle_len** to decrypt **key** and **checksum**, thereby retrieving the **sec_passwd_rec_t** and **sec_chksum_t** pickles, and thence the original transmitted keying data.

11.6.1.22 sec_rgy_sid_t

The **sec_rgy_sid_t** data type represents an account's UUIDs.

```
typedef struct sec_rgy_sid_t {
    uuid_t          person;
    uuid_t          group;
    uuid_t          org;
} sec_rgy_sid_t;
```

Its fields are the following:

- **person**
  Principal UUID.
- **group**
  Group UUID.
- **org**
  Organisation UUID.

11.6.1.23 sec_rgy_unix_sid_t

The **sec_rgy_unix_sid_t** data type represents an account's local-IDs.

```
typedef struct {
    signed32    person;
    signed32    group;
    signed32    org;
} sec_rgy_unix_sid_t;
```

Its fields are the following:

- **person**
  Principal local-ID.
- **group**
  Group local-ID.
- **org**
  Organisation local-ID.

11.6.1.24 rs_acct_info_t

The **rs_acct_info_t** data type represents a performance-optimised data type for returning account data. In the success case (*status* = **error_status_ok**), **rs_acct_info_t** represents a value of type **struct result** (see definition below); in the error case (*status* ≠ **error_status_ok**) it is empty (thereby preventing unnecessary marshalling/unmarshalling of data in the error case).

```
typedef union switch (signed32 status) {
    case error_status_ok:
        struct {
            sec_rgy_acct_key_t      key_parts;
            sec_rgy_sid_t           sid;
            sec_rgy_unix_sid_t      unix_sid;
            sec_rgy_acct_admin_t    admin_part;
            sec_rgy_acct_user_t     user_part;
        } result;
    default:
        /*empty*/                   /*empty*/;
} rs_acct_info_t;
```

The fields of **result** are the following:

- **key_parts**
  Indicates the minimal portion of the account's <P, G, O> name triple that is required to identify the account.
- **sid**
  The account's UUID.
- **unix_sid**
  The account's local-ID.
- **admin_part**
  The account's administration-level information.
- **user_part**
  The account's user-level information.

11.6.2 Interface UUID and Version Number for rs_acct

The interface UUID and version number for the **rs_acct** interface are given by the following:

```
[
    uuid(4c878280-2000-0000-0d00-028714000000),
    version(1.0)
]
interface rs_acct {
/* begin running listing of rs_acct interface */
```

11.6.3 rs_acct_add( )

The *rs_acct_add*( ) operation adds (or registers) an account to the RS datastore.

```
void
rs_acct_add (
    [in]        handle_t                rpc_handle,
    [in]        sec_rgy_login_name_t    *login_name,
    [in, out]   sec_rgy_acct_key_t      *key_parts,
    [in]        sec_rgy_acct_user_t     *user_part,
    [in]        sec_rgy_acct_admin_t    *admin_part,
    [in, ref]   rs_acct_key_transmit_t  *key,
    [in]        sec_passwd_type_t       new_keytype,
    [out]       sec_passwd_version_t    *new_key_version,
    [out]       rs_cache_data_t         *cache_info,
    [out]       error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *login_name* parameter identifies the (name of the) account to be registered (added).

The *key_parts* parameter indicates the minimal portion of *login_name*'s <P, G, O> name triple that is required to identify the account.

The *user_part* parameter indicates the user-level portion of the account's datastore data.

The *admin_part* parameter indicates the administrative-level portion of the account's datastore data.

The *key* parameter indicates the long-term cryptographic key to be stored in the RS's datastore for this account. Namely, if *key* carries a cryptographic key, that is stored as the account's long-term key; if *key* carries a password (and possibly also pepper), then that is transformed into a cryptographic key according to the type specified by the *new_keytype* parameter (see Section 4.3.6.1 on page 190), and that is stored as the account's long-term key.

The *new_keytype* parameter indicates the type of the long-term cryptographic key determined by *key*. (If *key* carries a cryptographic key instead of a password, then the type of that key must be the same as *key*.)

The *new_key_version* parameter indicates the version number of the long-term cryptographic key determined by *key*. This version number may be specified by the client (in the **version_number** field of the **sec_passwd_rec_t** structure of *key*). If the client specifies **sec_c_key_version_none**, then the server assigns it. In either case, the version number is returned to the client in *new_key_version*. (In the **sec_c_key_version_none** case, the server typically assigns the next smallest available version number; that is, 1 in the case of a new account, incrementing the old version number by 1 in the case of a pre-existing account.)

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has the following permissions on the principal component of the account to be added (**(*****login_name***).pname**): Management Info (**m**), Authentication Info (**a**) and User Info (**u**).

11.6.4 rs_acct_delete( )

The *rs_acct_delete*( ) operation deletes (removes) an account from the RS datastore.

```
void
rs_acct_delete (
    [in]      handle_t                rpc_handle,
    [in]      sec_rgy_login_name_t    *login_name,
    [out]     rs_cache_data_t         *cache_info,
    [out]     error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *login_name* parameter identifies the (name of the) account to be deleted.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has the following permissions on the principal component of the account to be deleted (**(****login_name***).pname**): Management Info (**m**), Authentication Info (**a**) and User Info (**u**).

11.6.5 rs_acct_rename( )

The *rs_acct_rename*( ) operation renames an account; that is, it associates a different <P, G, O> name triple to an existing account's data in the RS datastore, but with the restriction that the principal component cannot be changed.

```
void
rs_acct_rename (
    [in]        handle_t                rpc_handle,
    [in]        sec_rgy_login_name_t    *old_login_name,
    [in]        sec_rgy_login_name_t    *new_login_name,
    [in, out]   sec_rgy_acct_key_t      *new_key_parts,
    [out]       error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *old_login_name* parameter identifies the existing name associated with the account data.

The *new_login_name* parameter identifies the new name to be associated with the account data. The new principal component (**(****new_login_name***).pname**) must be the same as the old principal component (**(****old_login_name***).pname**).

The *new_key_parts* parameter indicates the minimal portion of *new_login_name*'s <P, G, O> name triple that is required to identify the account.

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Management Info (**m**) permission on the principal component of the existing account (**(****old_login_name***).pname**).

11.6.6 rs_acct_lookup( )

The *rs_acct_lookup*( ) operation retrieves (reads) account data from the RS server / datastore. This operation returns the datastore information of the next account, at or following a specified cursor position, whose name matches that indicated by a specified account <P, G, O> name triple (where the notion of matching recognises wildcards).

```
[idempotent] void
rs_acct_lookup (
    [in]        handle_t                    rpc_handle,
    [in, out]   sec_rgy_login_name_t        *login_name,
    [in, out]   sec_rgy_cursor_t            *cursor,
    [out]       rs_cache_data_t             *cache_info,
    [out]       rs_acct_info_t              *result );
```

The *rpc_handle* parameter identifies the RS server.

The *login_name* parameter identifies the (name of the) account to be read from. On input, any of its components (**(***login_name***).pname**, **(***login_name***).gname** or **(***login_name***).oname**) which is empty (that is, of length 0) is considered to be a wildcard (that is, not used as a matching criterion — *every* name matches an empty string). On output, all components of *login_name* are non-empty, and indicate the account whose data is retrieved in *result*.

The *cursor* parameter indicates, on input, the current cursor position (so that the accounts at and following this position are eligible to be retrieved on the current invocation of this operation). On output, the *cursor* parameter indicates the cursor position next following the retrieved account(s). (See Section 11.2.7 on page 362.)

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *result* parameter represents the result of this operation.

The status of this operation is indicated by the *status* of the *result* parameter.

Required rights: This operation succeeds only if the calling client has Read (**r**) permission on the principal component of the matched account (**(***login_name***).pname**).

11.6.7 rs_acct_replace( )

The *rs_acct_replace*( ) operation modifies (writes) account data in the RS server / datastore.

```
void
rs_acct_replace (
    [in]        handle_t                  rpc_handle,
    [in]        sec_rgy_login_name_t      *login_name,
    [in, out]   sec_rgy_acct_key_t        *key_parts,
    [in]        rs_acct_parts_t           modify_parts,
    [in]        sec_rgy_acct_user_t       *user_part,
    [in]        sec_rgy_acct_admin_t      *admin_part,
    [in, ptr]   rs_acct_key_transmit_t    *key,
    [in]        sec_passwd_type_t         new_keytype,
    [out]       sec_passwd_version_t      *new_key_version,
    [out]       rs_cache_data_t           *cache_info,
    [out]       error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *login_name* parameter identifies the (name of the) account to be replaced (modified, written, updated).

The *key_parts* parameter indicates the minimal portion of the *login_name*'s <P, G, O> name triple that is required to identify the account. (Even though *key_parts* is specified as both an input and output parameter, it is used only as an input parameter for this operation.)

The *modify_parts* parameter indicates which portion(s) of the account's datastore information is to be updated:

- If the **rs_acct_part_user** bit is set, the account's user-level information is to be replaced with *user_part* (if this bit is reset, *user_part* is ignored).
- If the **rs_acct_part_admin** bit is set, the account's administration-level information is to be replaced with *admin_part* (if this bit is reset, *admin_part* is ignored).
- If the **rs_acct_part_passwd** bit is set, the account's long-term cryptographic key is to be replaced with the keying information contained in (or generatable from) *key* and *new_keytype* (if this bit is reset, *key* and *new_keytype* are ignored).
- If the **rs_acct_part_login_name** bit is set, the minimal portion of the account's <P, G, O> name triple that is required to identify the account is to be replaced with *key_parts* (if this bit is reset, *key_parts* is ignored).

The *user_part* parameter indicates new user-level information.

The *admin_part* parameter indicates new administrative-level information.

The *key* parameter indicates a new long-term cryptographic key. (See the description of the *key* parameter of *rs_acct_add*( ).)

The *new_keytype* parameter indicates the type of the long-term cryptographic key determined by *key*. (See the description of the *new_keytype* parameter of *rs_acct_add*( ).)

The *new_key_version* parameter indicates the version number or number of the long-term cryptographic key determined by *key*. (See the description of the *new_key_version* parameter of *rs_acct_add*( ).)

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has the following permission(s) on the principal component of the account to be modified (**(***login_name***).pname**):

- Management Info (**m**), if **(***admin_part***).flags** or **(***admin_part***).expiration_date** is to be changed (to a value different from the previously stored value).
- Authentication Info (**a**), if **(***admin_part***).authentication_flags** or **(***admin_part***).good_since_date** is to be changed (to a value different from the previously stored value).
- User Info (**u**), if the *key* or any field of **(***user_part***)** is to be written (that is, if the **rs_acct_part_passwd** or **rs_acct_part_user** bit of the *modify_parts* parameter is set).

11.6.8 rs_acct_get_projlist( )

The *rs_acct_get_projlist*( ) operation retrieves an account's project list; that is, the primary group and the concurrent secondary group list associated with the principal component of the account (**(***login_name***).pname**).

```
[idempotent] void
rs_acct_get_projlist (
    [in]        handle_t                rpc_handle,
    [in]        sec_rgy_login_name_t    *login_name,
    [in, out]   sec_rgy_cursor_t        *projlist_cursor,
    [in]        signed32                max_number,
    [out]       signed32                *supplied_number,
    [out, length_is(*supplied_number),size_is(max_number)]
                uuid_t                  id_projlist[],
    [out, length_is(*supplied_number),size_is(max_number)]
                signed32                unix_projlist[],
    [out]       signed32                *num_projects,
    [out]       rs_cache_data_t         *cache_info,
    [out]       error_status_t          *status );

} /* end running listing of rs_acct interface */
```

The *rpc_handle* parameter identifies the RS server.

The *login_name* parameter identifies the (name of the) account whose project list is to be retrieved.

The *projlist_cursor* parameter indicates, on input, the current cursor position (so that the concurrent groups at and following this position are eligible to be retrieved on the current invocation of this operation). On output, *projlist_cursor* indicates the cursor position next following the retrieved concurrent group(s). (See Section 11.2.7 on page 362.)

The *max_number* parameter indicates the maximum number of concurrent groups to be retrieved.

The *supplied_number* parameter indicates the number of retrieved concurrent groups.

The *id_projlist* parameter indicates the UUIDs of retrieved concurrent groups.

The *unix_projlist* parameter indicates the local-IDs of retrieved concurrent groups.

The *num_projects* parameter indicates the total number of concurrent groups associated with the account (that is, with **(***login_name***).pname**). (The *projlist_cursor* parameter is used to coordinate multiple invocations to retrieve the complete project list of concurrent groups.)

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

Required rights: This operation succeeds only if the calling client has Read (**r**) permission on the principal component of the account (**(***login_name***).pname**).

11.7 The rs_misc RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_misc** RPC interface.

11.7.1 Common Data Types and Constants for rs_misc

The following are common data types and constants used in the **rs_misc** interface.

11.7.1.1 rs_login_info_t

The **rs_login_info_t** data type represents account data, appropriate for network and local system login usage. It is performance-optimised (see **rs_pgo_query_result_t**) so that in the success case (*status* = **error_status_ok**), **rs_login_info_t** represents account data; in the error case (*status* ≠ **error_status_ok**) it is empty (thereby preventing unnecessary marshalling / unmarshalling of data in the error case).

```
typedef union switch (long status) tagged_union {
    case error_status_ok:
        struct {
            sec_rgy_acct_key_t      key_parts;
            sec_rgy_sid_t           sid;
            sec_rgy_unix_sid_t      unix_sid;
            sec_rgy_acct_user_t     user_part;
            sec_rgy_acct_admin_t    admin_part;
            sec_rgy_plcy_t          policy_data;
            sec_rgy_name_t          cell_name;
            uuid_t                  cell_uuid;
        } result;
    default:
        /*empty*/                   /*empty*/;
} rs_login_info_t;
```

The fields of *result* are the following:

- **key_parts**
  Indicates the minimal portion of the account's <P, G, O> name triple that is required to identify the account.
- **sid**
  The account's UUIDs.
- **unix_sid**
  The account's local-IDs.
- **user_part**
  The account's user-level information.
- **admin_part**
  The account's administration-level information.
- **policy_data**
  The account's effective policy data.
- **cell_name**
  The account's home cell's name.
- **cell_uuid**
  The account's home cell's UUID.

11.7.1.2 rs_update_seqno_t

The **rs_update_seqno_t** data type represents an event's monotonically increasing sequence number assigned by the master.

```
typedef struct
{
    unsigned32        high;
    unsigned32        low;
} rs_update_seqno_t;
```

11.7.2 Interface UUID and Version Number for rs_misc

The interface UUID and version number for the **rs_misc** interface are given by the following:

```
[
    uuid(4c878280-5000-0000-0d00-028714000000),
    version(1.0)
]
interface rs_misc {
/* begin running listing of rs_misc interface */
```

11.7.3 rs_login_get_info()

The *rs_login_get_info*( ) operation retrieves (reads) account data from the RS server / datastore.

```
[idempotent] void
rs_login_get_info (
    [in]        handle_t              rpc_handle,
    [in, out]   sec_rgy_login_name_t  *login_name,
    [out]       rs_cache_data_t       *cache_info,
    [out]       rs_login_info_t       *result,
    [in]        signed32              max_number,
    [out]       signed32              *supplied_number,
    [out, length_is(*supplied_number),size_is(max_number)]
                uuid_t                id_projlist[],
    [out, length_is(*supplied_number),size_is(max_number)]
                signed32              unix_projlist[],
    [out]       signed32              *num_projects );
}
```

The *rpc_handle* parameter identifies the RS server.

The *login_name* parameter identifies the (name of the) account whose information is to be retrieved.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *result* parameter represents the bulk of the information returned by this operation.

Note: The project list information returned by this operation does not occur in the **rs_login_info_t** data type because the IDL language does not permit conformant arrays in union arms.

The *max_number* parameter indicates the maximum number of concurrent groups to be retrieved.

The *supplied_number* parameter indicates the number of retrieved concurrent groups.

The *id_projlist* parameter indicates the UUIDs of retrieved concurrent groups.

The *unix_projlist* parameter indicates the local-IDs of retrieved concurrent groups.

The *num_projects* parameter indicates the total number of concurrent groups associated with the account (that is, with **(***login_name***).pname**).

The status of this operation is indicated by the *status* of the *result* parameter.

Required rights: This operation supports name-based authorisation for *local* principals only (that is, those in the same cell as this RS server), in addition to the usual EPAC-based authorisation supported by other RS RPC operations (that is a special characteristic of this operation). In either case, this operation succeeds only if the calling client has Read (**r**) permission on the principal component of the account (**(***login_name***).pname**).

This operation provides all of the credentials and login policy information required for both network and local logins. Apart from its characteristic support of name-based authorisation, this operation is a mere optimisation; it duplicates in a single operation the core functionality that is embodied in four other RS operations, namely *rs_properties_get_info*( ), *rs_policy_get_info*( ), *rs_acct_lookup*( ) and *rs_acct_get_projlist*( ).

11.7.4 rs_wait_until_consistent( )

The *rs_wait_until_consistent*( ) operation returns a value of **TRUE** once all replicas have been updated. Otherwise, at least one replica is incommunicado.

```
boolean32
rs_wait_until_consistent (
    [in]        handle_t            rpc_handle,
    [out]       rs_cache_data_t     *cache_info,
    [out]       error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.7.5 rs_check_consistency( )

The *rs_check_consistency*( ) operation performs a non-blocking check for replica consistency.

```
boolean32
rs_check_consistency (
    [in]        handle_t            rpc_handle,
    [out]       boolean32           *retry,
    [in,out]    rs_update_seqno_t   *seqno,
    [out]       rs_cache_data_t     *cache_info,
    [out]       error_status_t      *status );

} /* end running listing of rs_misc interface */
```

The *rpc_handle* parameter identifies the RS server.

The *retry* parameter, if **TRUE**, indicates that a replica is responsive but not consistent (out of sync).

As input, the *seqno* parameter is set to NULL by the client. As output, *seqno* contains the reference sequence number required for subsequent polling attempts to a responsive but inconsistent replica.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

The client calls *rs_check_consistency*( ) initially with a NULL *seqno*. The operation returns a *retry* value of **TRUE** and a reference *seqno* to be used for subsequent polling attempts to any replica that is responsive but inconsistent (out of sync).

This routine returns a value of **TRUE** if none of the polled replicas are incommunicado.

11.8 The rs_attr RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_attr** RPC interface.

11.8.1 Common Data Types and Constants for rs_attr

The following are common data types and constants used in the **rs_attr** interface.

11.8.1.1 sec_attr_component_name_t

The **sec_attr_component_name_t** data type is a pointer to a character string used to further specify the object to which the attribute is attached. (Note that this data type is analogous to the **sec_acl_component_name_t** data type in the ACL interface.)

```
typedef [string, ptr] unsigned char *sec_attr_component_name_t;
```

11.8.1.2 rs_attr_cursor_t

The **rs_attr_cursor_t** data type provides a datastore scan *cursor* (that is, a position indicator) for iterative database operations for schema and attribute interfaces.

```
typedef struct {
    uuid_t          source;
    unsigned32      object;
    unsigned32      list;
    unsigned32      entry;
    unsigned32      num_entries_left;
    boolean32       valid;
} rs_attr_cursor_t;
```

Its fields are the following:

- **source**
  Object UUID of the RS server that initialized the cursor.
- **object**
  The RS object upon which an operation is currently being performed. For schema operations, this object is identified by the *schema_name* parameter of the operation; for attribute operations, it is identified by the *component_name* parameter.
- **list**
  Optionally identifies a list of entries to be operated on, identified by the *attr_list* parameter of the operation (not used for schema operations).
- **entry**
  The datastore entry for the current operation. For schema operations, this is the sequential id of the current schema entry. For attribute operations, this is the number of entries remaining in the datastore.
- **num_entries_left**
  The approximate number of datastore entries that remain to be seen.
- **valid**
  If 0, an uninitialized cursor. Set to 1 at initialization.

11.8.1.3 sec_attr_bind_auth_info_type_t

The **sec_attr_bind_auth_info_type_t** data type is an enumeration that defines whether or not an RPC binding is authenticated. This data type is used in conjunction with the **sec_attr_bind_auth_info_t** data type to set up the authorization method and parameters for an RPC binding.

```
typedef enum {
        sec_attr_bind_auth_none,
        sec_attr_bind_auth_dce
} sec_attr_bind_auth_info_type_t;
```

The following values are currently registered:

- **sec_attr_bind_auth_none**
  The binding is not authenticated.
- **sec_attr_bind_auth_dce**
  The binding uses DCE shared-secret key authentication.

11.8.1.4 sec_attr_bind_auth_info_t

The **sec_attr_bind_auth_info_t** data type is a discriminated union that defines authorization and authentication parameters for an RPC binding. This data type is used in conjunction with the **sec_attr_bind_auth_info_type_t** data type to set up the authorization method and parameters for an RPC binding.

```
typedef union
    switch (sec_attr_bind_auth_info_type_t    info_type)
    tagged_union {
        case sec_attr_bind_auth_none:
            ;
        case sec_attr_bind_auth_dce:
            struct {
                [string, ptr] char            *svr_princ_name;
                unsigned32                    protect_level;
                unsigned32                    authn_svc;
                unsigned32                    authz_svc;
            } dce_info;
    } sec_attr_bind_auth_info_t;
```

The **sec_attr_bind_auth_info_t** data type consists of the following elements:

- **info_type**
  Specifies whether or not the binding is authenticated.
- **dce_info**
  A tagged union specifying the method of authorization and the authorization parameters. For unauthenticated bindings (**info_type** is **sec_attr_bind_auth_none**), no parameters are supplied. For authenticated bindings (**info_type** is **sec_attr_bind_auth_dce**), the following union is supplied:

  — **svr_princ_name**
  The principal name of the RS server referenced by the binding handle.

  — **protect_level**
  The protection level for RPC calls made using the binding handle. The protection level determines the degree to which authenticated communications between the client and the

server are protected by the authentication service specified by **authn_svc**.

If the RPC runtime or the RPC protocol in the bound protocol sequence does not support a specified **protect_level**, the level is automatically upgraded to the next higher supported level. The possible protection levels are as follows:

- **rpc_c_protect_level_default**
  Uses the default protection level for the specified authentication service. (The default protection level for DCE shared-secret key authentication is **rpc_c_protect_level_pkt**.)
- **rpc_c_protect_level_none**
  Performs no authentication; tickets are not exchanged, session keys are not established, client EPACs or names are not certified, and transmissions are in the clear. Note that although uncertified EPACs should not be trusted, they may be useful for debugging, tracing, and measurement purposes.
- **rpc_c_protect_level_connect**
  Authenticates only when the client establishes a relationship with the server.
- **rpc_c_protect_level_call**
  Authenticates only at the beginning of each RPC call when the RS server receives the request.

  This level does not apply to RPC calls made over a connection-based protocol sequence (that is, **ncacn_ip_tcp**). If this level is specified and the binding handle uses a connection-based protocol sequence, the routine uses the **rpc_c_protect_level_pkt** level instead.
- **rpc_c_protect_level_pkt**
  Ensures that all data received is from the expected client.
- **rpc_c_protect_level_pkt_integ**
  Ensures and verifies that none of the data transferred between client and server has been modified. This is the highest protection level that is guaranteed to be present in the RPC runtime.
- **rpc_c_protect_level_pkt_privacy**
  Authenticates as specified by all of the previous levels and also encrypts each RPC argument value. This is the highest protection level, but is not guaranteed to be present in the RPC runtime.

— **authn_svc**
Specifies the authentication service to use. The exact level of protection provided by the authentication service is specified by **protect_level** (see above). The supported authentication services are as follows:

- **rpc_c_authn_none**
  No authentication; no tickets are exchanged, no session keys established, client EPACs or names are not transmitted, and transmissions are in the clear. Specify **rpc_c_authn_none** to turn authentication off for RPC calls made using this binding.
- **rpc_c_authn_dce_secret**
  DCE shared-secret key authentication.

- **rpc_c_authn_default**
  Default authentication service. The current default authentication service is DCE shared-secret key; therefore, specifying **rpc_c_authn_default** is equivalent to specifying **rpc_c_authn_dce_secret** .

— **authz_svc**
Specifies the authorization service implemented by the server for the interface. The validity and trustworthiness of authorization data, like any application data, is dependent on the authentication service and protection level specified. The supported authorization services are as follows:

- **rpc_c_authz_none**
  Server performs no authorization. This is valid only if **authn_svc** is set to **rpc_c_authn_none** (see above), specifying that no authentication is being performed.
- **rpc_c_authz_name**
  Server performs authorization based on the client principal name. This value cannot be used if **authn_svc** is **rpc_c_authn_none** (see above).
- **rpc_c_authz_dce**
  Server performs authorization using the client's DCE Extended Privilege Attribute Certificate (EPAC) sent to the server with each RPC call made with this binding. Generally, access is checked against DCE Access Control Lists (ACLs).

11.8.1.5 sec_attr_bind_type_t

The **sec_attr_bind_type_t** data type specifies the binding type for attribute operations.

```
typedef unsigned32  sec_attr_bind_type_t;
const unsigned32    sec_attr_bind_type_string       = 0;
const unsigned32    sec_attr_bind_type_twrs         = 1;
const unsigned32    sec_attr_bind_type_svrname      = 2;
```

The following values are currently registered:

- **sec_attr_bind_type_string**
  RPC string binding.
- **sec_attr_bind_type_twrs**
  DCE protocol tower representation of bindings.
- **sec_attr_bind_type_svrname**
  Name of trigger server for lookup in a directory service.

11.8.1.6 sec_attr_twr_ref_t

The **sec_attr_twr_ref_t** data type represents a pointer to an RPC protocol tower (**twr_t**, defined in Appendix L, Protocol Tower Encoding, of the referenced X/Open DCE RPC Specification).

```
typedef [ptr] twr_t *sec_attr_twr_ref_t;
```

11.8.1.7 sec_attr_twr_set_t

The **sec_attr_twr_set_t** data type is a structure that defines an array of towers. This data is used by the client to pass an unallocated array of towers, which the server must allocate.

```
typedef struct {
    unsigned32                              count;
    [size_is(count)] sec_attr_twr_ref_t     towers[];
} sec_attr_twr_set_t;

typedef [ptr] sec_attr_twr_set_t *sec_attr_twr_set_p_t;
```

Its fields are the following:

- **count**
  The number of towers in the array.
- **towers**[ ]
  Pointers to RPC protocol towers.

11.8.1.8 sec_attr_bind_svrname

The **sec_attr_bind_svrname** data type specifies the name of the server for lookup in a directory service.

```
typedef struct {
    unsigned32              name_syntax;
    [string, ptr] char      *name;
} sec_attr_bind_svrname;
```

This data type contains the following elements:

- **name_syntax**
  The binding type used for this server (see Section 11.8.1.5 on page 415).
- **name**
  The actual string representation of the server name.

11.8.1.9 sec_attr_binding_t

The **sec_attr_binding_t** data type is the trigger server's binding union.

```
typedef union
    switch (sec_attr_bind_type_t           bind_type)
    tagged_union {
        case sec_attr_bind_type_string:
            [string, ptr] char             *string_binding;
        case sec_attr_bind_type_twrs:
            [ptr] sec_attr_twr_set_t       *twr_set;
        case sec_attr_bind_type_svrname:
            [ptr] sec_attr_bind_svrname    *svrname;
} sec_attr_binding_t;

typedef [ptr] sec_attr_binding_t *sec_attr_binding_p_t;
```

This data type contains the following elements:

- **string_binding**
  RPC string binding.
- **twr_set**
  DCE protocol tower representation of binding.

- **svrname**
  Name of trigger server in directory namespace.

11.8.1.10 sec_attr_bind_info_t

The **sec_attr_bind_info_t** data type specifies attribute trigger binding information.

```
typedef struct {
    sec_attr_bind_auth_info_t                   auth_info;
    unsigned32                                  num_bindings;
    [size_is(num_bindings)] sec_attr_binding_t  bindings[];
} sec_attr_bind_info_t;
```

This data type contains the following elements:

- **auth_info**
  The binding authorization information.
- **num_bindings**
  The number of binding handles in **bindings**.
- **bindings**[ ]
  An array of RPC binding handles.

11.8.1.11 sec_attr_enc_printstring_p_t

The **sec_attr_enc_printstring_p_t** data type is a pointer to a printstring encoding type structure.

```
typedef [string, ptr] unsigned char *sec_attr_enc_printstring_p_t;
```

11.8.1.12 sec_attr_enc_str_array_t

The **sec_attr_enc_str_array_t** data type defines a printstring array.

```
typedef struct {
    unsigned32                      num_strings;
    [size_is(num_strings)]
    sec_attr_enc_printstring_p_t    strings[];
} sec_attr_enc_str_array_t;
```

This data type contains the following elements:

- **num_strings**
  The number of strings in the array.
- **strings**[ ]
  An array of pointers to printstrings.

11.8.1.13 sec_attr_enc_bytes_t

The **sec_attr_enc_bytes_t** data type defines the length of attribute encoding values for attributes whose values are defined to be byte strings.

```
typedef struct {
    unsigned32                length;
    [size_is(length)] byte    data[];
} sec_attr_enc_bytes_t;
```

This data type contains the following elements:

- **length**
  The size of the **data** array.
- **data**[ ]
  The length of attribute encoding data.

11.8.1.14 sec_attr_i18n_data_t

The **sec_attr_i18n_data_t** data type defines the codeset and value length of the encoding values of attributes whose values are defined to be internationalized byte strings.

```
typedef struct {
    unsigned32              codeset;
    unsigned32              length;
    [size_is(length)] byte  data[];
} sec_attr_i18n_data_t;
```

This data type contains the following elements:

- **codeset**
  Identifier for the OSF-registered codeset used to encode the data.
- **length**
  The size of the **data** array.
- **data**[ ]
  The length of attribute encoding data.

11.8.1.15 sec_attr_enc_attr_set_t

The **sec_attr_enc_attr_set_t** data type supplies the UUIDs of each member of a set of attributes.

```
typedef struct {
    unsigned32                      num_members;
    [size_is(num_members)] uuid_t   members[];
} sec_attr_enc_attr_set_t;
```

This data type contains the following elements:

- **num_members**
  The total number of attributes in the attribute set.
- **members**[ ]
  An array containing the UUID for each member in the set.

11.8.1.16 sec_attr_encoding_t

The **sec_attr_encoding_t** data type is an enumerator that contains attribute encoding tags used to define the legal encodings for attribute values.

```
typedef enum {
    sec_attr_enc_any,
    sec_attr_enc_void,
    sec_attr_enc_integer,
    sec_attr_enc_printstring,
    sec_attr_enc_printstring_array,
    sec_attr_enc_bytes,
    sec_attr_enc_confidential_bytes,
    sec_attr_enc_i18n_data,
    sec_attr_enc_uuid,
    sec_attr_enc_attr_set,
    sec_attr_enc_binding,
    sec_attr_enc_trig_binding
} sec_attr_encoding_t;
```

This data type contains the following elements:

- **sec_attr_enc_any**
  The attribute value can be of any legal encoding type. This encoding tag is legal for a schema entry only; an attribute entry must contain a specified encoding type.
- **sec_attr_enc_void**
  The attribute has no value.
- **sec_attr_enc_integer**
  The attribute value is a signed 32-bit integer.
- **sec_attr_enc_printstring**
  The attribute value is a printable IDL string in DCE Portable Character Set.
- **sec_attr_enc_printstring_array**
  The attribute value is an array of printstrings.
- **sec_attr_enc_bytes**
  The attribute value is a string of bytes. The string is assumed to be a pickle or some other self-describing type.
- **sec_attr_enc_confidential_bytes**
  The attribute value is a string of bytes that have been encrypted in the key of the principal object to which the attribute is attached. The string is assumed to be a pickle or some other self-describing type. This encoding type is useful only when attached to a principal object, where it is decrypted and encrypted each time the principal's password changes.
- **sec_attr_enc_i18n_data**
  The attribute value is an internationalized string of bytes with a tag identifying the OSF-registered codeset used to encode the data.
- **sec_attr_enc_uuid**
  The attribute value is a UUID, of type **uuid_t**.
- **sec_attr_enc_attr_set**
  The attribute value is an attribute set, a vector of attribute UUIDs used to associate multiple related attribute instances which are members of the set.
- **sec_attr_enc_binding**
  The attribute value is a **sec_attr_bind_info_t** data type that specifies DCE server binding information.

- **sec_attr_enc_trig_binding**
  This encoding type, returned by an **rs_attr_lookup** call, informs the client agent of the trigger binding information of an attribute with a query trigger.

Attribute values must conform to the attribute's encoding type.

11.8.1.17 sec_attr_value_t

The **sec_attr_value_t** data type defines the values of attributes.

```
typedef union sec_attr_u
    switch (sec_attr_encoding_t                 attr_encoding)
    tagged_union {
        case sec_attr_enc_void:
            ;
        case sec_attr_enc_integer:
            signed32                            signed_int;
        case sec_attr_enc_printstring:
            sec_attr_enc_printstring_p_t        printstring;
        case sec_attr_enc_printstring_array:
            [ptr] sec_attr_enc_str_array_t      *string_array;
        case sec_attr_enc_bytes:
        case sec_attr_enc_confidential_bytes:
            [ptr] sec_attr_enc_bytes_t          *bytes;
        case sec_attr_enc_i18n_data:
            [ptr] sec_attr_i18n_data_t          *idata;
        case sec_attr_enc_uuid:
            uuid_t                              uuid;
        case sec_attr_enc_attr_set:
            [ptr] sec_attr_enc_attr_set_t       *attr_set;
        case sec_attr_enc_binding:
            [ptr] sec_attr_bind_info_t          *binding;
    } sec_attr_value_t;
```

This data type contains the following elements:

- **attr_encoding**
  An attribute encoding tag that specifies the type of attribute value (see Section 11.8.1.16 on page 418).
- **tagged_union**
  A tagged union whose contents depend on **attr_encoding** as follows:

If attr_encoding is...	Then tagged_union contains...
sec_attr_enc_void	NULL
sec_attr_enc_integer	**signed_int** (32-bit signed integer)
sec_attr_enc_printstring	**printstring** (string pointer)
sec_attr_enc_printstring_array	**string_array** (pointer to an array of printstrings)
sec_attr_enc_bytes **sec_attr_enc_confidential_bytes**	**bytes** (pointer to a structure of type **sec_attr_enc_bytes_t**)
sec_attr_enc_i18n_data	**idata** (pointer to a structure of type **sec_attr_i18n_data_t**)
sec_attr_enc_uuid	**uuid** (UUID)
sec_attr_enc_attr_set	**attr_set** (pointer to a structure of type **sec_attr_enc_attr_set_t**)
sec_attr_enc_binding	**binding** (pointer to a structure of type **sec_attr_binding_info_t**)

11.8.1.18 sec_attr_t

The **sec_attr_t** data type defines an attribute.

```
typedef struct {
    uuid_t                  attr_id;
    sec_attr_value_t        attr_value;
} sec_attr_t;
```

This data type contains the following elements:

- **attr_id**
  The UUID of the attribute.
- **attr_value**
  The attribute value.

11.8.1.19 sec_attr_vec_t

The **sec_attr_vec_t** data type defines an array of attributes.

```
typedef struct {
    unsigned32                  num_attrs;
    [size_is(num_attrs), ptr]
        sec_attr_t              *attrs;
} sec_attr_vec_t;
```

This data type contains the following elements:

- **num_attrs**
  The number of elements in the **attrs** array.
- **attrs**
  An array of pointers to attributes.

11.8.2 Interface UUID for rs_attr

The interface UUID for the **rs_attr** interface is given by the following:

```
[
    uuid(a71fc1e8-567f-11cb-98a0-08001e04de8c)
]
interface rs_attr {
```

11.8.3 rs_attr_cursor_init( )

The *rs_attr_cursor_init*( ) operation initializes a scan cursor.

```
void
rs_attr_cursor_init (
    [in]    handle_t                    rpc_handle,
    [in]    sec_attr_component_name_t   component_name,
    [out]   unsigned32                  *cur_num_attrs,
    [out]   rs_attr_cursor_t            *cursor,
    [out]   rs_cache_data_t             *cache_info,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *component_name* parameter identifies the RS object on which to perform this operation.

The *cur_num_attrs* parameter contains the current total number of attributes associated with the RS object at the time of this call.

The *cursor* parameter contains the cursor initialized to the first attribute in the list of attributes associated with this object.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.8.4 rs_attr_lookup_by_id()

The *rs_attr_lookup_by_id*( ) operation looks up (reads) attribute(s) by UUID.

```
void
rs_attr_lookup_by_id (
    [in]       handle_t                    rpc_handle,
    [in]       sec_attr_component_name_t   component_name,
    [in, out]  rs_attr_cursor_t            *cursor,
    [in]       unsigned32                  num_attr_keys,
    [in]       unsigned32                  space_avail,
    [in, size_is(num_attr_keys)]
               sec_attr_t                  attr_keys[],
    [out]      unsigned32                  *num_returned,
    [out, size_is(space_avail), length_is(*num_returned)]
               sec_attr_t                  attrs[],
    [out]      unsigned32                  *num_left,
    [out]      rs_cache_data_t             *cache_info,
    [out]      error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *component_name* parameter identifies the RS object on which to perform this lookup operation.

As input, the *cursor* parameter is an initialized or uninitialized cursor to the RS object. As output, *cursor* is positioned just past the attributes returned as output to this call.

The *num_attr_keys* parameter specifies the number of elements in the *attr_keys* array. If the *num_attr_keys* is set to 0, this function will return all attributes that the caller is authorized to see.

The *space_avail* parameter specifies the size of the output *attrs* array.

The *attr_keys*[ ] parameter contains the attribute type UUIDs for the attribute instance(s) requested by this lookup. If the requested attribute type is associated with a query trigger, the **attr_keys*.**attr_value** field may be used to pass in optional information required by the trigger query. If no information is to be passed in the **attr_keys*.**attr_value** field (whether the type indicates a trigger query or not), the **attr_keys*.**attr_value** encoding type should be set to **sec_attr_enc_void**.

The *num_returned* parameter specifies the number of attribute instances returned in the *attrs* array.

The *attrs*[ ] parameter contains the attributes retrieved by UUID.

The *num_left* parameter contains the approximate number of attributes matching the search criteria that could not be returned due to space constraints in the *attrs* buffer. (This number may not be precise if the server allows updates between successive query calls.)

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.8.5 rs_attr_lookup_no_expand()

The *rs_attr_lookup_no_expand*( ) operation reads attributes by UUID without expanding attribute sets to their constituent member attributes.

```
void
rs_attr_lookup_no_expand (
    [in]       handle_t                   rpc_handle,
    [in]       sec_attr_component_name_t  component_name,
    [in, out]  rs_attr_cursor_t           *cursor,
    [in]       unsigned32                 num_attr_keys,
    [in]       unsigned32                 space_avail,
    [in, size_is(num_attr_keys)]
               sec_attr_t                 attr_keys[],
    [out]      unsigned32                 *num_returned,
    [out, size_is(space_avail), length_is(*num_returned)]
               sec_attr_t                 attrs[],
    [out]      unsigned32                 *num_left,
    [out]      rs_cache_data_t            *cache_info,
    [out]      error_status_t             *status );
```

The *rpc_handle* parameter identifies the RS server.

The *component_name* parameter identifies the RS object on which to perform this operation.

As input, the *cursor* parameter is an initialized or uninitialized cursor to the RS object. As output, *cursor* is positioned just past the attributes returned as output to this call.

The *num_attr_keys* parameter specifies the number of elements in the *attr_keys* array. If the *num_attr_keys* is set to 0, this function will return all attributes that the caller is authorized to see.

The *space_avail* parameter specifies the size of the output *attrs* array.

The *attr_keys*[ ] parameter contains the attribute type UUIDs for the attribute instance(s) requested by this lookup. If the requested attribute type is associated with a query trigger, the **attr_keys*.**attr_value** field may be used to pass in optional information required by the trigger query. If no information is to be passed in the **attr_keys*.**attr_value** field (whether the type indicates a trigger query or not), the **attr_keys*.**attr_value** encoding type should be set to **sec_attr_enc_void**.

The *num_returned* parameter specifies the number of attribute instances returned in the *attrs* array.

The *attrs*[ ] parameter contains the attributes retrieved by UUID.

The *num_left* parameter contains the approximate number of attributes matching the search criteria that could not be returned due to space constraints in the *attrs* buffer. (This number may not be precise if the server allows updates between successive query calls.)

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.8.6 rs_attr_lookup_by_name()

The *rs_attr_lookup_by_name*( ) operation looks up (reads) a single attribute by name.

```
void
rs_attr_lookup_by_name (
    [in]         handle_t                    rpc_handle,
    [in]         sec_attr_component_name_t   component_name,
    [in, string] char                        *attr_name,
    [out]        sec_attr_t                  *attr,
    [out]        rs_cache_data_t             *cache_info,
    [out]        error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *component_name* parameter identifies the RS object on which to perform this operation.

The *attr_name* parameter is the name of the attribute to be retrieved.

The *attr* parameter is the first attribute instance of the named type.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.8.7 rs_attr_update()

The *rs_attr_update*( ) operation writes (and/or creates) an attribute. All attributes are written (created) or else none are modified.

```
void
rs_attr_update (
    [in]    handle_t                    rpc_handle,
    [in]    sec_attr_component_name_t   component_name,
    [in]    unsigned32                  num_to_write,
    [in, size_is(num_to_write)]
            sec_attr_t                  in_attrs[],
    [out]   signed32                    *failure_index,
    [out]   rs_cache_data_t             *cache_info,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *component_name* parameter identifies the RS object on which to perform this operation.

The *num_to_write* parameter specifies the number of attributes in the *in_attrs* array.

The *in_attrs*[ ] parameter contains the attribute instances to be written.

The *failure_index* parameter, in an error case, contains the array index of the element in *in_attrs* that caused this update to fail. If the failure cannot be attributed to a specific attribute, *failure_index* is set to -1.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.8.8 rs_attr_test_and_update()

The *rs_attr_test_and_update*( ) operation updates attributes if a set of control attributes retain specified values.

```
void
rs_attr_test_and_update (
    [in]    handle_t                    rpc_handle,
    [in]    sec_attr_component_name_t   component_name,
    [in]    unsigned32                  num_to_test,
    [in, size_is(num_to_test)]
            sec_attr_t                  test_attrs[],
    [in]    unsigned32                  num_to_write,
    [in, size_is(num_to_write)]
            sec_attr_t                  update_attrs[],
    [out]   signed32                    *failure_index,
    [out]   rs_cache_data_t             *cache_info,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *component_name* parameter identifies the RS object on which to perform this operation.

The *num_to_test* parameter specifies the number of control attributes in the *test_attrs* array.

The *test_attrs*[ ] parameter contains control attributes whose types and values must exactly match those of instances on the RS object in order for the update to take place.

The *num_to_write* parameter specifies the number of attributes in the *update_attrs* array.

The *update_attrs*[ ] parameter contains the attribute instances to be written.

The *failure_index* parameter, in an error case, contains the array index of the element in *in_attrs* that caused this update to fail. If the failure cannot be attributed to a specific attribute, *failure_index* is set to -1.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.8.9 rs_attr_delete( )

The *rs_attr_delete*( ) operation deletes attributes.

```
void
rs_attr_delete (
    [in]    handle_t                    rpc_handle,
    [in]    sec_attr_component_name_t   component_name,
    [in]    unsigned32                  num_to_delete,
    [in, size_is(num_to_delete)]
            sec_attr_t                  attrs[],
    [out]   signed32                    *failure_index,
    [out]   rs_cache_data_t             *cache_info,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *component_name* parameter identifies the RS object on which to perform this operation.

The *num_to_delete* parameter specifies the number of attributes in the *attrs* array.

The *attrs*[ ] parameter contains the attributes to be deleted.

The *failure_index* parameter, in an error case, contains the array index of the element in *in_attrs* that caused this update to fail. If the failure cannot be attributed to a specific attribute, *failure_index* is set to -1.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.8.10 rs_attr_get_referral( )

The *rs_attr_get_referral*( ) operation obtains a referral to an attribute update site.

```
void
rs_attr_get_referral (
    [in]    handle_t                    rpc_handle,
    [in]    sec_attr_component_name_t   component_name,
    [in]    uuid_t                      *attr_id,
    [out]   sec_attr_twr_set_p_t        *towers,
    [out]   rs_cache_data_t             *cache_info,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *component_name* parameter identifies the RS object on which to perform this operation.

The *attr_id* parameter specifies the UUID of the attribute for which a referral is being sought.

The *towers* parameter specifies the binding information for a suitable update site.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.8.11 rs_attr_get_effective()

The *rs_attr_get_effective*( ) operation reads the effective attributes by UUID.

```
void
rs_attr_get_effective (
    [in]       handle_t                   rpc_handle,
    [in]       sec_attr_component_name_t  component_name,
    [in]       unsigned32                 num_attr_keys,
    [in, size_is(num_attr_keys)]
               sec_attr_t                 attr_keys[],
    [out, ref] sec_attr_vec_t             *attr_list,
    [out]      rs_cache_data_t            *cache_info,
    [out]      error_status_t             *status );
```

The *rpc_handle* parameter identifies the RS server.

The *component_name* parameter identifies the RS object on which to perform this operation.

The *num_attr_keys* parameter specifies the number of elements in the *attr_keys* array.

The *attr_keys*[ ] parameter contains the attribute type UUIDs for the attribute instance(s) requested by this lookup.

The *attr_list* parameter contains an attribute vector allocated by the server containing all of the attributes matching the search criteria.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.9 The rs_attr_schema RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_attr_schema** RPC interface.

11.9.1 Common Data Types and Constants for rs_attr_schema

The following are common data types and constants used in the **rs_attr_schema** interface.

11.9.1.1 sec_attr_acl_mgr_info_t

The **sec_attr_acl_mgr_info_t** structure contains the access control information defined in a schema entry for an attribute.

```
typedef struct {
    uuid_t                  acl_mgr_type;
    sec_acl_permset_t       query_permset;
    sec_acl_permset_t       update_permset;
    sec_acl_permset_t       test_permset;
    sec_acl_permset_t       delete_permset;
} sec_attr_acl_mgr_info_t;

typedef [ptr] sec_attr_acl_mgr_info_t *sec_attr_acl_mgr_info_p_t;
```

This data type contains the following elements:

- **acl_mgr_type**
  The UUID of the ACL manager type that supports the object type to which the attribute can be attached. See Table 11-1 on page 358 for a list of ACL Manager Types UUIDs.
- **query_permset**
  The permission bits needed to access the attribute's value.
- **update_permset**
  The permission bits needed to update the attribute's value.
- **test_permset**
  The permission bits needed to test the attribute's value.
- **delete_permset**
  The permission bits needed to delete an attribute instance.

Refer to Chapter 7 for information on Access Control Lists and the definition of the **sec_acl_permset_t** data type.

11.9.1.2 sec_attr_sch_entry_flags_t

The **sec_attr_sch_entry_flags_t** data type is a flag word used to specify schema entry flags.

```
typedef unsigned32  sec_attr_sch_entry_flags_t;
const unsigned32    sec_attr_sch_entry_none         = 0x00000000;
const unsigned32    sec_attr_sch_entry_unique       = 0x00000001;
const unsigned32    sec_attr_sch_entry_multi_inst   = 0x00000002;
const unsigned32    sec_attr_sch_entry_reserved     = 0x00000004;
const unsigned32    sec_attr_sch_entry_use_defaults = 0x00000008;
```

The following values are currently registered:

- **sec_attr_sch_entry_none**
  No schema entry flags.

- **sec_attr_sch_entry_unique**
  Each instance of this attribute type must have a unique value within the cell for the object type implied by the ACL manager type. If this flag is not set, there is no check for uniqueness during attribute writes.
- **sec_attr_sch_entry_multi_inst**
  This attribute type may be multi-valued; in other words, multiple instances of the same attribute type can be attached to a single object. If this flag is not set, only one instance of this attribute can be attached to a given object.
- **sec_attr_sch_entry_reserved**
  This schema entry can not be deleted through any interface or by any user. If this flag is not set, then the entry can be deleted by any authorized user.
- **sec_attr_sch_entry_use_defaults**
  The system-defined default value (if any) for this attribute will be returned on a client query if an instance of this attribute doesn't exist for the queried object. If this flag is not set, no search/return of system default values will take place. (If the attribute does not not exist and this flag is FALSE (0), then no attribute instance will be returned.)

11.9.1.3 sec_attr_intercell_action_t

The **sec_attr_intercell_action_t** data type specifies the action that should be taken by the Privilege (PS) Service when it reads acceptable attributes from a foreign cell. A foreign attribute is acceptable only if there is either a schema entry for the foreign cell or if **sec_attr_intercell_act_accept** is set to **TRUE**.

```
typedef enum {
    sec_attr_intercell_act_accept,
    sec_attr_intercell_act_reject,
    sec_attr_intercell_act_evaluate
} sec_attr_intercell_action_t;
```

This data type contains the following elements:

- **sec_attr_intercell_act_accept**
  If the **sec_attr_sch_entry_unique** entry flag is set for this schema entry, retain the attribute from the foreign cell only if its value is unique among all attribute instances of this attribute type within the current cell. If **sec_attr_sch_entry_unique** is not set, retain the foreign attribute.
- **sec_attr_intercell_act_reject**
  Unconditionally discard the foreign attribute.
- **sec_attr_intercell_act_evaluate**
  Use the binding information in the **trig_binding** field of the **sec_attr_schema_entry_t** for this schema entry to make a *sec_attr_trig_query*( ) call to a trigger server. That server determines whether to retain the attribute value, discard the attribute value, or map the attribute to another value(s).

11.9.1.4 sec_attr_trig_type_flags_t

The **sec_attr_trig_type_flags_t** data type is a flag word used to indicate schema trigger types.

```
typedef unsigned32  sec_attr_trig_type_flags_t;
const unsigned32    sec_attr_trig_type_none         = 0x00000000;
const unsigned32    sec_attr_trig_type_query        = 0x00000001;
const unsigned32    sec_attr_trig_type_update       = 0x00000002;
```

The following values are currently registered:

- **sec_attr_trig_type_none**
  No trigger type entry flags.
- **sec_attr_trig_type_query**
  Attribute type is configured with a query trigger. When this flag is set, the following things happen during a lookup request for this attribute type:
  — Client binds to the trigger server using the **trig_binding** field of the **sec_attr_schema_entry_t** structure.
  — Client issues a *sec_attr_trig_query*( ) call, passing in the attribute keys with the **attr_value** fields provided by the calling routine that issued the lookup request.
  — If the *sec_attr_trig_query*( ) call is successful, client returns output attributes to the caller.

  If this flag is set and an update request is made for this attribute type, the input values of the update request are ignored and the client creates a ‘‘stub’’ attribute instance as a marker. Modifications to the attribute value must occur at the trigger server.
- **sec_attr_trig_type_update**
  Attribute type is configured with an update trigger. When this flag is set, the following things happen during an update request for this attribute type:
  — Client binds to the trigger server using the **trig_binding** field of the **sec_attr_schema_entry_t** structure.
  — Client issues a *sec_attr_trig_update*( ) call, passing in the attributes provided by the calling routine that issued the update request.
  — If the *sec_attr_trig_update*( ) is successful, the client stores the output attribute(s) in the ERA database and returns the output attribute(s) to the calling routine.

11.9.1.5 sec_attr_acl_mgr_info_set_t

The **sec_attr_acl_mgr_info_set_t** data type defines an attribute’s ACL manager set.

```
typedef struct {
    unsigned32                  num_acl_mgrs;
    [size_is(num_acl_mgrs)]
    sec_attr_acl_mgr_info_p_t   mgr_info[];
} sec_attr_acl_mgr_info_set_t;
```

The structure consists of the following elements:

- **num_acl_mgrs**
  Specifies the number of ACL managers in the ACL manager set.
- **mgr_info**[ ]
  Pointers to a set of ACL manager types and their associated permission sets.

11.9.1.6 sec_attr_schema_entry_t

The **sec_attr_schema_entry_t** data type defines a complete attribute entry for the schema catalog. The entry is identified by both name and UUID. Although either can be used as a retrieval key, the name should be used for interactive access to the attribute and the UUID for programmatic access.

```
typedef struct {
    [string, ptr] char                  *attr_name;
    uuid_t                              attr_id;
    sec_attr_encoding_t                 attr_encoding;
    [ptr] sec_attr_acl_mgr_info_set_t   *acl_mgr_set;
    sec_attr_sch_entry_flags_t          schema_entry_flags;
    sec_attr_intercell_action_t         intercell_action;
    sec_attr_trig_type_flags_t          trig_types;
    [ptr] sec_attr_bind_info_t          *trig_binding;
    [string, ptr] char                  *scope;
    [string, ptr] char                  *comment;
} sec_attr_schema_entry_t;
```

This data type contains the following elements:

- **attr_name**
  The name of the attribute type.
- **attr_id**
  The UUID of the attribute type.
- **attr_encoding**
  The encoding type for this attribute (see Section 11.8.1.16 on page 418 for a list and description of attribute encoding tags).
- **acl_mgr_set**
  The ACL manager types (and their associated permission bits) that support objects on which attributes of this type can be created.
- **schema_entry_flags**
  The schema entry flag settings for this attribute type (see Section 11.9.1.2 on page 428).
- **intercell_action**
  Flag indicating how the Privilege Service will handle attributes from a foreign cell (see Section 11.9.1.3 on page 429).
- **trig_types**
  Flag indicating whether a trigger can perform update or query operations for this attribute type (see Section 11.9.1.4 on page 429).
- **trig_binding**
  Binding information for the attribute trigger.
- **scope**
  Definition of the objects to which this attribute type can be attached.
- **comment**
  Comment text.

11.9.1.7 sec_attr_schema_entry_parts_t

The **sec_attr_schema_entry_parts_t** data type is a flag word used during an update to specify which fields of an input schema entry (of type **sec_attr_schema_entry_t**) contain modified information for the update.

```
typedef unsigned32 sec_attr_schema_entry_parts_t;

const unsigned32   sec_attr_schema_part_name       = 0x00000001;
const unsigned32   sec_attr_schema_part_acl_mgrs   = 0x00000002;
const unsigned32   sec_attr_schema_part_unique     = 0x00000004;
const unsigned32   sec_attr_schema_part_reserved   = 0x00000008;
const unsigned32   sec_attr_schema_part_defaults   = 0x00000010;
const unsigned32   sec_attr_schema_part_intercell  = 0x00000020;
const unsigned32   sec_attr_schema_part_trig_types = 0x00000040;
const unsigned32   sec_attr_schema_part_trig_bind  = 0x00000080;
const unsigned32   sec_attr_schema_part_comment    = 0x00000100;
```

This data type contains the following flags:

- **sec_attr_schema_part_name**
  Modified information for the **attr_name** field.
- **sec_attr_schema_part_acl_mgrs**
  Modified information for the **acl_mgr_set** field.
- **sec_attr_schema_part_unique**
  Not applicable in the current release.
- **sec_attr_schema_part_reserved**
  Modified information for the **sec_attr_sch_entry_reserved** setting of the **schema_entry_flags** field.
- **sec_attr_schema_part_defaults**
  Modified information for the **sec_attr_sch_entry_use_defaults** setting of the **schema_entry_flags** field.
- **sec_attr_schema_part_intercell**
  Modified information for the **intercell_action** field.
- **sec_attr_schema_part_trig_types**
  Not applicable in the current release.
- **sec_attr_schema_part_trig_bind**
  Modified information for the **trig_binding** field.
- **sec_attr_schema_part_comment**
  Modified information for the **comment** field.

11.9.2 Interface UUID for rs_attr_schema

The interface UUID for the **rs_attr_schema** interface is given by the following:

```
[
    uuid(b47c9460-567f-11cb-8c09-08001e04de8c)
]
interface rs_attr_schema {
```

11.9.3 rs_attr_schema_create_entry( )

The *rs_attr_schema_create_entry*( ) operation creates a new schema entry.

```
void
rs_attr_schema_create_entry (
    [in]    handle_t                    rpc_handle,
    [in]    sec_attr_component_name_t   schema_name,
    [in]    sec_attr_schema_entry_t     *schema_entry,
    [out]   rs_cache_data_t             *cache_info,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *schema_name* parameter identifies the schema object on which to perform this operation.

The *schema_entry* parameter identifies the schema entry to be created. Values must be supplied for all of the fields of the input **sec_attr_schema_entry_t** structure, with the exception of the **trig_types**, **trig_bind**, and **comment** fields, which are all optional.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.9.4 rs_attr_schema_delete_entry( )

The *rs_attr_schema_delete_entry*( ) operation deletes a schema entry. This is a radical operation that will delete or invalidate any existing attributes of this type on nodes dominated by the schema.

```
void
rs_attr_schema_delete_entry (
    [in]    handle_t                    rpc_handle,
    [in]    sec_attr_component_name_t   schema_name,
    [in]    uuid_t                      *attr_id,
    [out]   rs_cache_data_t             *cache_info,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *schema_name* parameter identifies the schema object to be deleted.

The *attr_id* parameter contains the UUID for the attribute type of the entry being deleted.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.9.5 rs_attr_schema_update_entry( )

The *rs_attr_schema_update_entry*( ) operation updates the modifiable fields of a schema entry.

```
void
rs_attr_schema_update_entry (
    [in]    handle_t                        rpc_handle,
    [in]    sec_attr_component_name_t       schema_name,
    [in]    sec_attr_schema_entry_parts_t   modify_parts,
    [in]    sec_attr_schema_entry_t         *schema_entry,
    [out]   rs_cache_data_t                 *cache_info,
    [out]   error_status_t                  *status );
```

The *rpc_handle* parameter identifies the RS server.

The *schema_name* parameter identifies the schema object on which to perform this operation.

The *modify_parts* parameter is a flag which identifies which fields of the *schema_entry* parameter are to be updated. Fields not indicated by *modify_parts*, or fields which are not permitted to be modified, will retain their current values.

The *schema_entry* parameter specifies a **sec_attr_schema_entry_t** data structure whose fields are NULL except for those fields which are being modified (as indicated by the *modify_parts* parameter).

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

If a field is indicated by its flag in *modify_parts*, that field from the input schema entry will completely replace the current field of the existing schema entry. All other fields will remain untouched.

Note: Fields which are arrays of structures (such as **acl_mgr_set**) and **trig_binding**) will be completely replaced by the new input array. This operation will not simply add one more element to the existing array.

11.9.6 rs_attr_schema_cursor_init( )

The *rs_attr_schema_cursor_init*( ) operation initializes a scan cursor.

```
void
rs_attr_schema_cursor_init (
    [in]    handle_t                    rpc_handle,
    [in]    sec_attr_component_name_t   schema_name,
    [out]   unsigned32                  *cur_num_entries,
    [out]   rs_attr_cursor_t            *cursor,
    [out]   rs_cache_data_t             *cache_info,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *schema_name* parameter identifies the schema object on which to perform this operation.

The *cur_num_entries* parameter specifies the current total number of entries in the schema at the time of this call.

The *cursor* parameter contains the cursor initialized to the first in the list of entries in the named schema.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.9.7 rs_attr_schema_scan( )

The *rs_attr_schema_scan*( ) operation reads a specified number of entries from the named schema object.

```
void
rs_attr_schema_scan (
    [in]      handle_t                    rpc_handle,
    [in]      sec_attr_component_name_t   schema_name,
    [in, out] rs_attr_cursor_t            *cursor,
    [in]      unsigned32                  num_to_read,
    [out]     unsigned32                  *num_read,
    [out, size_is(num_to_read), length_is(*num_read)]
              sec_attr_schema_entry_t     schema_entries[ ]
    [out]     rs_cache_data_t             *cache_info,
    [out]     error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *schema_name* parameter identifies the schema object on which to perform this operation.

As input, the *cursor* parameter is an initialized or uninitialized cursor to the schema object. As output, *cursor* is positioned just past the attributes returned as output to this call.

The *num_to_read* parameter specifies the size of the *schema_entries* array; that is, the maximum number of entries to be returned in this call.

The *num_read* parameter specifies the actual number of entries returned in the *schema_entries* array.

The *schema_entries*[ ] array contains the entries read.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.9.8 rs_attr_schema_lookup_by_name( )

The *rs_attr_schema_lookup_by_name*( ) operation performs a lookup (read) on a schema entry identified by name.

```
void
rs_attr_schema_lookup_by_name (
    [in]         handle_t                  rpc_handle,
    [in]         sec_attr_component_name_t schema_name,
    [in, string] char                      *attr_name,
    [out]        sec_attr_schema_entry_t   *schema_entry,
    [out]        rs_cache_data_t           *cache_info,
    [out]        error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *schema_name* parameter identifies the schema object on which to perform this operation.

The *attr_name* parameter is the name that identifies the entry to be read.

The *schema_entry* parameter contains the entry identified by *attr_name*.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.9.9 rs_attr_schema_lookup_by_id()

The *rs_attr_schema_lookup_by_id*() operation performs a lookup (read) on a schema entry identified by its UUID.

```
void
rs_attr_schema_lookup_by_id (
    [in]   handle_t                    rpc_handle,
    [in]   sec_attr_component_name_t   schema_name,
    [in]   uuid_t                      *attr_id,
    [out]  sec_attr_schema_entry_t     *schema_entry,
    [out]  rs_cache_data_t             *cache_info,
    [out]  error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *schema_name* parameter identifies the schema object on which to perform this operation.

The *attr_id* parameter contains the UUID of the attribute type identifying the entry to be read.

The *schema_entry* parameter contains the entry identified by *attr_id*.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.9.10 rs_attr_schema_get_referral()

The *rs_attr_schema_get_referral*() operation obtains a referral to a schema update site. This function is used when the current schema site yields a **sec_schema_site_readonly** error. Some replication managers will require all updates for a given object to be directed to a given replica. Clients of the generic schema interface may not know they are dealing with an object that is replicated in this way. This function allows them to recover from this problem and rebind to the proper update site.

```
void
rs_attr_schema_get_referral (
    [in]   handle_t                   rpc_handle,
    [in]   sec_attr_component_name_t  schema_name,
    [in]   uuid_t                     *attr_id,
    [out]  sec_attr_twr_set_p_t       *towers,
    [out]  error_status_t             *status );
```

The *rpc_handle* parameter identifies the RS server.

The *schema_name* parameter identifies the schema object on which to perform this operation.

The *attr_id* parameter contains the UUID that identifies the schema entry.

The *towers* parameter contains a pointer to an RPC protocol tower for the schema update site.

The *status* parameter returns the status of the operation.

11.9.11 rs_attr_schema_get_acl_mgrs( )

The *rs_attr_schema_get_acl_mgrs*( ) operation retrieves a list of the ACL Manager types that protect those objects that are associated with the named schema. The returned list is valid for use in the **acl_mgr_set** field of a **sec_attr_schema_entry_t** schema entry.

```
void
rs_attr_schema_get_acl_mgrs (
    [in]    handle_t                    rpc_handle,
    [in]    sec_attr_component_name_t   schema_name,
    [in]    unsigned32                  size_avail,
    [out]   unsigned32                  *size_used,
    [out]   unsigned32                  *num_acl_mgr_types,
    [out, size_is(size_avail), length_is(*size_used)]
            uuid_t                      acl_mgr_types[ ]
    [out]   rs_cache_data_t             *cache_info,
    [outs   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *schema_name* parameter identifies the schema object on which to perform this operation.

The *size_avail* parameter specifies the size of the *acl_mgr_types* array; that is, the maximum number of ACL manager types that can be returned by this call.

The *size_used* parameter specifies the actual number of ACL Manager types returned in the array.

The *num_acl_mgr_types* parameter specifies the total number of ACL Manager types supported for this schema. If this value is greater than *size_used*, this operation should be called again with a larger *acl_mgr_types* buffer.

The *acl_mgr_types*[ ] array contains the UUIDs for the ACL Manager types that protect the objects associated with this schema.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.9.12 rs_attr_schema_aclmgr_strings( )

The *rs_attr_schema_aclmgr_strings*( ) operation retrieves printable representations for each permission bit that the input ACL Manager type will support.

```
void
rs_attr_schema_aclmgr_strings (
    [in]        handle_t                    rpc_handle,
    [in]        sec_attr_component_name_t   schema_name,
    [in, ref]   uuid_t                      *acl_mgr_type,
    [in]        unsigned32                  size_avail,
    [out]       uuid_t                      *acl_mgr_type_chain,
    [out]       sec_acl_printstring_t       *acl_mgr_info,
    [out, ref]  boolean32                   *tokenize,
    [out]       unsigned32                  *total_num_printstrings,
    [out, ref]  unsigned32                  *size_used,
    [out, size_is(size_avail), length_is(*size_used)]
                sec_acl_printstring_t       permstrings[],
    [out]       rs_cache_data_t             *cache_info,
    [out]       error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *schema_name* parameter identifies the schema object on which to perform this operation.

The *acl_mgr_type* parameter contains the UUID of the ACL Manager type for which the printstrings are to be returned.

The *size_avail* parameter specifies the size of the *permstrings* array; that is, the maximum number of printstrings that can be returned by this call.

The *acl_mgr_type_chain* parameter, if not set to **uuid_nil**, identifies the UUID of the next ACL Manager type in a chain supporting ACL Managers with more than 32 permission bits.

The *acl_mgr_info* parameter contains printstrings provides the name, help information, and complete set of supported permission bits for this ACL Manager type.

If set, the *tokenize* parameter specifies that the permission bit strings should be tokenized.

The *total_num_printstrings* parameter specifies the total number of permission printstrings supported by this ACL manager type. If this value is greater than *size_avail*, this function should be invoked again with a buffer of the appropriate size.

The *size_used* parameter contains the number of printstrings returned in the *permstrings* array.

The *permstrings*[ ] array contains the printstrings for each permission supported by this ACL manager type.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

There may be aliases for common permission combinations; by convention simple entries should appear at the beginning of the array, and combinations should appear at the appear at the end. When the *tokenize* flag is **FALSE**, the permission printstrings are unambiguous; therefore printstrings for various permissions can be concatenated. When *tokenize* is **TRUE**, however, this property does not hold and the strings should be tokenized before input or output.

If the ACL Manager type supports more than 32 permission bits, multiple manager types can be used — one for each 32-bit wide slice of permissions. When this is the case, the *acl_mgr_type_chain* parameter is set to the UUID of the next ACL Manager type in the set. The final result for the chain returns **uuid_nil** in the *manager_type_chain* parameter.

11.10 The rs_prop_acct RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_prop_acct** RPC interface.

11.10.1 Common Data Types and Constants for rs_prop_acct

The following are common data types and constants used in the **rs_prop_acct** interface.

11.10.1.1 rs_prop_acct_add_data_t

The **rs_prop_acct_add_data_t** data type is used for bulk account add propagations during replica initialization.

The RS server stores only the current key version for most principals. If the principal has multiple key versions, the additional versions must be propagated using *rs_prop_acct_add_key_version*( ) (defined in Section 11.10.7 on page 443). Only current keys may be propagated via *rs_prop_acct_add*( ) (see Section 11.10.3 on page 441 for its definition).

```
typedef struct {
    sec_rgy_login_name_t          login_name;
    sec_rgy_acct_user_t           user_part;
    sec_rgy_acct_admin_t          admin_part;
    rs_acct_key_transmit_t        *key;
    [ptr] rs_acct_key_transmit_t  *unix_passwd;
    sec_rgy_foreign_id_t          client;
    sec_passwd_type_t             keytype;
    } rs_prop_acct_add_data_t;
```

This data type contains the following elements:

- **login_name**
  The principal name of the account.
- **user_part**
  The user portion of the account (see Section 11.6.1.15 on page 397).
- **admin_part**
  The administrative portion of the account (see Section 11.6.1.5 on page 392).
- **key**
  Indicates a new long-term cryptographic key. (See the description of the *key* parameter of *rs_acct_add*( ).)
- **unix_passwd**
  If not NULL, contains the pickled and encrypted UNIX password (generated by the UNIX *crypt*( ) function). The **plain** arm of the decrypted and unpickled **sec_passwd_rec_t** contains the UNIX passwd.
- **client**
  During initialization, **client** is filled with **nil_uuid**s to indicate that keys are encrypted under a session key. For incremental add propagations, **client** identifies the principal under whose key the account's key is encrypted.
- **keytype**
  Indicates the type of the long-term cryptographic key determined by *key*. (See the description of the *new_keytype* parameter of *rs_acct_add*( ).)

11.10.1.2 rs_prop_acct_key_data_t

The **rs_prop_acct_key_data_t** data type is used for bulking up multiple key propagations in the *rs_prop_acct_key_add_version*( ) routine.

```
typedef  struct {
    rs_acct_key_transmit_t  *key;
    boolean32               current;
    sec_timeval_sec_t       garbage_collect;
} rs_prop_acct_key_data_t;
```

This data type contains the following elements:

- **key**
  Indicates a new long-term cryptographic account key.
- **current**
  If current is true the key is added as the current version and *garbage_collect* is ignored (current keys are never garbage collected). If current is false, the key is stored as a back-version of the account's key using *garbage_collect*.
- **garbage_collect**
  The expiration time of the *key*.

11.10.1.3 rs_replica_master_info_t and rs_replica_master_info_p_t

The **rs_replica_master_info_t** and **rs_replica_master_info_p_t** data type describes the current master replica.

```
typedef struct
{
    uuid_t                  master_id;
    rs_update_seqno_t       master_seqno;
    unsigned32              master_compat_sw_rev;
    sec_timeval_t           update_ts;
    rs_update_seqno_t       update_seqno;
    rs_update_seqno_t       previous_update_seqno;
} rs_replica_master_info_t, *rs_replica_master_info_p_t;
```

This data type contains the following elements:

- **master_id**
  The **uuid_t** of the current master replica.
- **master_seqno**
  The current sequence number corresponding to database updates.
- **master_compat_sw_rev**
  The software revision number that the master replica is compatible with.
- **update_ts**
  Timestamp of the latest replica update. This would correspond to the time which the last sequence number was propagated to replicas.
- **update_seqno**
  The last sequence number to be propagated out to the slave replicas.
- **previous_update_seqno**
  The previous sequence number that has been propagated to slave replicas.

11.10.2 Interface UUID and Version Number for rs_prop_acct

The interface UUID and version number for the **rs_prop_acct** interface are given by the following:

```
[
    uuid(68097130-de43-11ca-a554-08001e0394c7),
    version(1),
    pointer_default(ptr)
]
interface rs_prop_acct {
```

11.10.3 rs_prop_acct_add()

The *rs_prop_acct_add*() operation will propagate account information in bulk to a security replica.

```
void
rs_prop_acct_add (
    [in]       handle_t                  rpc_handle,
    [in]       unsigned32                num_accts,
    [in, ref, size_is(num_accts)]
               rs_prop_acct_add_data_t   accts[],
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                 propq_only,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *num_accts* parameter identifies the number of accounts described in the *accts[ ]* array.

The *accts*[ ] parameter provides *num_accts* security account descriptions.

The *master_info* parameter provides information on the current master replica (see Section 11.10.1.3 on page 440).

If the *propq_only* flag is set the propagation information will only be added to the propagation queue. It will not be propagated.

The *status* parameter returns the status of the operation.

11.10.4 rs_prop_acct_delete()

The *rs_prop_acct_delete*() operation will propagate an account delete to a security replica.

```
void
rs_prop_acct_delete (
    [in]       handle_t                  rpc_handle,
    [in]       sec_rgy_login_name_t      *login_name,
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                 propq_only,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *login_name* parameter identifies the principal name of the account to be deleted.

The *master_info* parameter provides the security master replica information to the security slave replica.

If the *propq_only* flag is set the account delete operation is only added to the propagation queue. It is not propagated to the replicas.

The *status* parameter returns the status of the operation.

11.10.5 rs_prop_acct_rename( )

The *rs_prop_acct_rename*( ) operation will propagate an account rename to security replicas.

```
void
rs_prop_acct_rename (
    [in]       handle_t                  rpc_handle,
    [in]       sec_rgy_login_name_t      *old_login_name,
    [in]       sec_rgy_login_name_t      *new_login_name,
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                 propq_only,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *old_login_name* parameter identifies the original principal name of the account to be renamed.

The *new_login_name* parameter identifies the new principal name of the account.

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flag is set the rename information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.10.6 rs_prop_acct_replace( )

The *rs_prop_acct_replace*( ) operation will propagate an account replace to security replicas.

```
void
rs_prop_acct_replace (
    [in]       handle_t                  rpc_handle,
    [in]       sec_rgy_login_name_t      *login_name,
    [in]       rs_acct_parts_t           modify_parts,
    [in]       sec_rgy_acct_user_t       *user_part,
    [in]       sec_rgy_acct_admin_t      *admin_part,
    [in, ptr]  rs_acct_key_transmit_t    *key,
    [in, ref]  sec_rgy_foreign_id_t      *client,
    [in]       sec_passwd_type_t         new_keytype,
    [in, ptr]  rs_acct_key_transmit_t    *unix_passwd,
    [in, ref]  sec_timeval_sec_t         *time_now,
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                 propq_only,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *login_name* parameter identifies the account name to replace.

The *modify_parts* parameter is a flags describing the objects within the account to replace. (see Section 11.6.1.16 on page 398).

The *user_part* parameter describes the user portion of the account to be replaced (see Section 11.6.1.15 on page 397).

The *admin_part* parameter describes the admininstration portion of the account to be replaced (see Section 11.6.1.5 on page 392).

The *key* parameter indicates a new long-term cryptographic key. (See the description of the *key* parameter of *rs_acct_add*( ).)

The *client* parameter identifies (by **uuid**) the principal under whose key the updated *key* and *unix_passwd* are encrypted. If **client_ids** contains NIL uuids, *key* and *unix_passwd* are encrypted under a session key.

The *new_keytype* parameter Indicates the type of the long-term cryptographic key determined by *key*. (See the description of the *new_keytype* parameter of *rs_acct_add*( ).)

The *unix_passwd* parameter if not NULL, contains the pickled and encrypted UNIX password (generated by the UNIX *crypt*( ) function). The **plain** arm of the decrypted and unpickled **sec_passwd_rec_t** contains the UNIX passwd.

The *time_now* parameter is used for garbage collecting expired versions of multi-version keys.

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flag is set the account replace information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.10.7 rs_prop_acct_add_key_version( )

The *rs_prop_acct_add_key_version*( ) operation adds specific versions of an account key. This routine is used only during initialization to propagate all extant key types and versions from a surrogate master to an initializing slave.

```
void
rs_prop_acct_add_key_version (
    [in]       handle_t                  rpc_handle,
    [in]       sec_rgy_login_name_t      *login_name,
    [in]       unsigned32                num_keys,
    [in, ref, size_is(num_keys)]
               rs_prop_acct_key_data_t   keys[],
    [in]       rs_replica_master_info_t  *master_info,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *login_name* parameter identifies the account name to propagate the key types to.

The *num_keys* parameter identifies the number of entries in the *keys[ ]* array.

The *keys*[ ] parameter. If the **current** field of *keys* is **TRUE**, the key is added as the current version and the **garbage_collect** field is ignored (current keys are never garbage collected). If **current** is **FALSE**, the key is stored as a back-version of the account's key using **garbage_collect**.

The *master_info* parameter provides information on the current master replica.

The *status* parameter returns the status of the operation.

```
} /* End rs_prop_acct interface */
```

11.11 The rs_prop_acl RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_prop_acl** RPC interface.

11.11.1 Common Data Types and Constants for rs_prop_acl

The following are common data types and constants used in the **rs_prop_acl** interface.

11.11.1.1 rs_prop_acl_data_t

The **rs_prop_acl_data_t** data type is used for bulk ACL replace propagations during initialization.

```
typedef  struct {
    sec_acl_component_name_t     component_name;
    uuid_t                       manager_type;
    sec_acl_type_t               acl_type;
    sec_acl_list_t               *acl_list;
} rs_prop_acl_data_t;
```

This data type contains the following elements:

- **component_name**
  The component name specifies the entity that the sec_acl is protecting.
- **manager_type**
  Identifies the manager that is interpreting this sec_acl
- **acl_type**
  Differentiates between the different types of sec_acls that an object can possess.
- **acl_list**
  The list of acls for the *component_name*.

11.11.2 Interface UUID and Version Number for rs_prop_acl

The interface UUID and version number for the **rs_prop_acl** interface are given by the following:

```
[
    uuid(591d87d0-de64-11ca-a11c-08001e0394c7),
    version(1),
    pointer_default(ptr)
]
interface rs_prop_acl {
```

11.11.3 rs_prop_acl_replace( )

The *rs_prop_acl_replace*( ) operation will propagate acl replacements in bulk.

```
void
rs_prop_acl_replace (
    [in]       handle_t                    rpc_handle,
    [in]       unsigned32                  num_acls,
    [in, size_is(num_acls)]
               rs_prop_acl_data_t          acls[],
    [in, ref]  rs_replica_master_info_t    *master_info,
    [in]       boolean32                   propq_only,
    [out]      error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *num_acls* parameter identifies the number of entries in the *acls*[ ] array.

The *acls*[ ] parameter is an array of *num_acls* acls to replace.

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flag is set the acl replacement is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.12 The rs_prop_attr RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_prop_attr** RPC interface.

11.12.1 Common Data Types and Constants for rs_prop_attr

The following are common data types and constants used in the **rs_prop_attr** interface.

11.12.1.1 rs_prop_attr_list_t

The **rs_prop_attr_list_t** data type contains an array of security attributes associated with a security entry.

```
typedef struct {
    unsigned32                      num_attrs;
    [size_is(num_attrs)]
            sec_attr_t              attrs[];
} rs_prop_attr_list_t;
```

This data type contains the following elements:

- **num_attrs**
  The number of attributes within the *attrs[ ]* array
- **attrs**[ ]
  An array of size *num_attrs* of security attributes.

11.12.1.2 rs_prop_attr_data_t

The **rs_prop_attr_data_t** data type contains a component name and property attributes for a bulk propagation operation.

```
typedef struct {
    sec_rgy_name_t                  component_name;
    rs_prop_attr_list_t             *attr_list;
} rs_prop_attr_data_t;
```

This data type contains the following elements:

- **component_name**
  The component name for the attribute list.
- **attr_list**
  The property attributes of *component_name.*

11.12.2 Interface UUID and Version Number for rs_prop_attr

The interface UUID and version number for the **rs_prop_attr** interface are given by the following:

```
[
    uuid(0eff23e6-555a-11cd-95bf-0800092784c3),
    version(1),
    pointer_default(ptr)
]
interface rs_prop_attr {
```

11.12.3 rs_prop_attr_update()

The *rs_prop_attr_update*( ) operation will propagate component property attributes in bulk.

```
void
rs_prop_attr_update (
    [in]       handle_t                  rpc_handle,
    [in]       unsigned32                num_prop_attrs,
    [in, ref, size_is(num_prop_attrs)]
               rs_prop_attr_data_t       prop_attrs[],
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                 propq_only,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *num_prop_attrs* parameter identifies the number of property attribute entries in the *prop_attrs[ ]* array.

The *prop_attrs*[ ] parameter contains a component name and an array property attributes associated with than name. There are *num_prop_attrs* entries in the array.

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flag is set the property attribute information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.12.4 rs_prop_attr_delete()

The *rs_prop_attr_delete*( ) operation will

```
void
rs_prop_attr_delete (
    [in]       handle_t                  rpc_handle,
    [in]       unsigned32                num_prop_attrs,
    [in, ref, size_is(num_prop_attrs)]
               rs_prop_attr_data_t       prop_attrs[],
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                 propq_only,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *num_prop_attrs* parameter identifies the number of property attribute arrays in the *prop_attrs[ ]* array.

The *prop_attrs*[ ] parameter contains a component name and an array property attributes associated with than name. There are *num_prop_attrs* entries in the array.

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flag is set the property attribute delete information is only placed the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

```
} /* End rs_prop_acl interface */
```

11.13 The rs_prop_attr_schema RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_prop_attr_schema** RPC interface.

11.13.1 Common Data Types and Constants for rs_prop_attr_schema

The following are common data types and constants used in the **rs_prop_attr_schema** interface.

11.13.1.1 rs_prop_attr_sch_create_data_t

The **rs_prop_attr_sch_create_data_t** data type contains a component name and an attribute schema definition for a propagation operation.

```
typedef struct {
    sec_attr_component_name_t       schema_name;
    sec_attr_schema_entry_t         schema_entry;
} rs_prop_attr_sch_create_data_t;
```

This data type contains the following elements:

- **schema_name**
  The *schema_name* specifies the entity to which the *schema_entry* attribute is attached.
- **schema_entry**
  Defines the attribute schema for *schema_name*.

11.13.2 Interface UUID and Version Number for rs_prop_attr_schema

The interface UUID and version number for the **rs_prop_attr_schema** interface are given by the following:

```
[
    uuid(0eff260c-555a-11cd-95bf-0800092784c3),
    version(1),
    pointer_default(ptr)
]
interface rs_prop_attr_sch {
```

11.13.3 rs_prop_attr_schema_create( )

The *rs_prop_attr_schema_create*( ) operation propagates in bulk a newly created attribute schema.

```
void
rs_prop_attr_schema_create (
    [in]       handle_t                        rpc_handle,
    [in]       unsigned32                      num_schemas,
    [in, ref, size_is(num_schemas)]
               rs_prop_attr_sch_create_data_t  schemas[],
    [in, ref]  rs_replica_master_info_t        *master_info,
    [in]       boolean32                       propq_only,
    [out]      error_status_t                  *status );
```

The *rpc_handle* parameter identifies the RS server.

The *num_schemas* parameter identifies the number of entries in the *schemas* array being propagated.

The *schemas*[ ] parameter is an array of size *num_schemas* containing attribute names and schemas.

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flag is set the attribute schema create information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.13.4 rs_prop_attr_schema_delete( )

The *rs_prop_attr_schema_delete*( ) operation

```
void
rs_prop_attr_schema_delete (
    [in]       handle_t                    rpc_handle,
    [in]       sec_attr_component_name_t   schema_name,
    [in]       uuid_t                      *attr_id,
    [in, ref]  rs_replica_master_info_t    *master_info,
    [in]       boolean32                   propq_only,
    [out]      error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *schema_name* parameter specifies the entity to which the attribute is attached.

The *attr_id* parameter is the attribute type uuid identifying the schema entry to be deleted

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flag is set the attribute schema delete information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.13.5 rs_prop_attr_schema_update( )

The *rs_prop_attr_schema_update*( ) operation

```
void
rs_prop_attr_schema_update (
    [in]       handle_t                    rpc_handle,
    [in, ref]  rs_replica_master_info_t    *master_info,
    [in]       boolean32                   propq_only,
    [out]      error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flag is set the attribute schema update information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

```
} /* End rs_prop_attr_sch interface */
```

11.14 The rs_prop_pgo RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_prop_pgo** RPC interface.

11.14.1 Common Data Types and Constants for rs_prop_pgo

The following are common data types and constants used in the **rs_prop_pgo** interface.

11.14.1.1 rs_prop_pgo_add_data_t

The **rs_prop_pgo_add_data_t** data type is used for bulk pgo add propagations during initialization.

```
typedef  struct  {
    sec_rgy_name_t          name;
    sec_rgy_pgo_item_t      item;
    sec_rgy_foreign_id_t    client;
}  rs_prop_pgo_add_data_t;
```

This data type contains the following elements:

- **name**
  The PGO name to add.
- **item**
  The identifying information for the name to add.
- **client**
  The client that originated the update.

11.14.2 Interface UUID and Version Number for rs_prop_pgo

The interface UUID and version number for the **rs_prop_pgo** interface are given by the following:

```
[
    uuid(c23626e8-de34-11ca-8cbc-08001e0394c7),
    version(1),
    pointer_default(ptr)
]
interface rs_prop_pgo {
```

11.14.3 rs_prop_pgo_add()

The *rs_prop_pgo_add*() operation propagates PGO add operations in bulk during replica initializations.

```
void
rs_prop_pgo_add (
    [in]        handle_t                rpc_handle,
    [in]        sec_rgy_domain_t        domain,
    [in]        unsigned32              num_pgo_items,
    [in, size_is(num_pgo_items)]
                rs_prop_pgo_add_data_t  pgo_items[],
    [in, ref]   rs_replica_master_info_t *master_info,
    [in]        boolean32               propq_only,
    [out]       error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *domain* parameter identifies the PGO domain of the entries to add.

The *num_pgo_items* parameter is the number of entries in the *pgo_items*[ ] array.

The *pgo_items*[ ] parameter is *num_pgo_items* number of *domain* PGO items to add.

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flag is set the PGO add information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.14.4 rs_prop_pgo_delete()

The *rs_prop_pgo_delete*( ) operation propagates a PGO delete to a security replica.

```
void
rs_prop_pgo_delete (
    [in]       handle_t                  rpc_handle,
    [in]       sec_rgy_domain_t          domain,
    [in, ref]  sec_rgy_name_t            name,
    [in]       sec_timeval_sec_t         cache_info,
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                 propq_only,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *domain* parameter identifies the PGO domain of the entry to delete.

The *name* parameter identifies the PGO item to delete.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flag is set the PGO delete information is only placed on the propagation queue. It is not propagated to security replicas.

The *status* parameter returns the status of the operation.

11.14.5 rs_prop_pgo_rename()

The *rs_prop_pgo_rename*( ) operation propagates a PGO rename to a security replica.

```
void
rs_prop_pgo_rename (
    [in]       handle_t                  rpc_handle,
    [in]       sec_rgy_domain_t          domain,
    [in, ref]  sec_rgy_name_t            old_name,
    [in, ref]  sec_rgy_name_t            new_name,
    [in]       sec_timeval_sec_t         cache_info,
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                 propq_only,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *domain* parameter identifies the PGO domain of the entry to rename.

The *old_name* parameter provides the name the PGO item is changing from.

The *new_name* parameter provides the name the PGO item is changing tp.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flag is set the PGO rename information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.14.6 rs_prop_pgo_replace()

The *rs_prop_pgo_replace*( ) operation propagates a PGO replace to a security replica.

```
void
rs_prop_pgo_replace (
    [in]       handle_t                  rpc_handle,
    [in]       sec_rgy_domain_t          domain,
    [in, ref]  sec_rgy_name_t            name,
    [in, ref]  sec_rgy_pgo_item_t        *item,
    [in]       sec_timeval_sec_t         cache_info,
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                 propq_only,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *domain* parameter identifies the PGO domain of the entry to replace.

The *name* parameter provides the name the PGO item is replacing.

The *item* parameter identifies the replacement item information.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flag is set the PGO replace information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.14.7 rs_prop_pgo_add_member()

The *rs_prop_pgo_add_member*( ) operation propagates PGO add member information to a security replica.

```
void
rs_prop_pgo_add_member (
    [in]       handle_t                  rpc_handle,
    [in]       sec_rgy_domain_t          domain,
    [in]       sec_rgy_name_t            go_name,
    [in]       unsigned32                num_members,
    [in, size_is(num_members)]
               sec_rgy_member_t          members[],
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                 propq_only,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *domain* parameter identifies the PGO domain of the entry to add members to. This must be either **group** or **organization**.

The *go_name* parameter provides the PGO name to add members to.

The *num_members* parameter identifies the number of entries in the *members* array.

The *members*[ ] parameter is an array of *num_members* members to add.

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flage is set the PGO add member information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.14.8 rs_prop_pgo_delete_member( )

The *rs_prop_pgo_delete_member*( ) operation propagates PGO member delete information to security replica.

```
void
rs_prop_pgo_delete_member (
    [in]       handle_t                  rpc_handle,
    [in]       sec_rgy_domain_t          domain,
    [in, ref]  sec_rgy_name_t            go_name,
    [in, ref]  sec_rgy_name_t            person_name,
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                 propq_only,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *domain* parameter identifies the PGO domain of the entry to delete a member from. This must be either **group** or **organization**.

The *go_name* parameter identifies the PGO name to delete the member *person_name* from.

The *person_name* parameter identifies the member to to delete from the *go_name* PGO.

The *master_info* parameter provides information on the current master replica.

If the *propq_only* flag is set the PGO member delete information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

```
} /* End rs_prop_pgo interface */
```

11.15 The rs_prop_plcy RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_prop_plcy** RPC interface.

11.15.1 Interface UUID and Version Number for rs_prop_plcy

The interface UUID and version number for the **rs_prop_plcy** interface are given by the following:

```
[
    uuid(e6ac5cb8-de3e-11ca-9376-08001e0394c7),
    version(1.1),
    pointer_default(ptr)
]
interface rs_prop_plcy {
```

11.15.2 rs_prop_properties_set_info()

The *rs_prop_properties_set_info*( ) operation propagates registry property information changes to replicas.

```
void
rs_prop_properties_set_info (
    [in]       handle_t                  rpc_handle,
    [in, ref]  sec_rgy_properties_t      *properties,
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                 propq_only,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *properties* parameter contains the registry property information to be updated.

The *master_info* parameter contains the master registry information.

If the *propq_only* flag is set the property information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.15.3 rs_prop_plcy_set_info()

The *rs_prop_plcy_set_info*( ) operation propagates organzation policy information changes to replicas.

```
void
rs_prop_plcy_set_info (
    [in]       handle_t                  rpc_handle,
    [in, ref]  sec_rgy_name_t            organization,
    [in, ref]  sec_rgy_plcy_t            *policy_data,
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                 propq_only,
    [out]      error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *organization* parameter contains the organization name.

The *policy_data* parameter contains the organization policy information to be updated.

The *master_info* parameter contains the master registry information.

If the *propq_only* flag is set the policy information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.15.4 rs_prop_auth_plcy_set_info()

The *rs_prop_auth_plcy_set_info*() operation propagates account authentication policy to replicas.

```
void
rs_prop_auth_plcy_set_info (
    [in]       handle_t                 rpc_handle,
    [in, ref]  sec_rgy_login_name_t     *account,
    [in, ref]  sec_rgy_plcy_auth_t      *auth_policy,
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                propq_only,
    [out]      error_status_t           *status );
```

The *rpc_handle* parameter identifies the RS server.

The *account* parameter describes the account that the authentication policy belongs to.

The *auth_policy* parameter contains the account authentication policy to be updated.

The *master_info* parameter contains the master registry information.

If the *propq_only* flag is set the policy information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.15.5 rs_prop_plcy_set_dom_cache_info()

The *rs_prop_plcy_set_dom_cache_info*() operation is only used to send the domain *cache_info* to initialize a slave. The update is not logged; the slave will checkpoint its database to disk when initialization completes.

```
void
rs_prop_plcy_set_dom_cache_info (
    [in]       handle_t                 rpc_handle,
    [in, ref]  rs_cache_data_t          *cache_info,
    [in, ref]  rs_replica_master_info_t  *master_info,
    [in]       boolean32                propq_only,
    [out]      error_status_t           *status );
```

The *rpc_handle* parameter identifies the RS server.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *master_info* parameter contains the master registry information.

If the *propq_only* flag is set the policy information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

```
} /* End rs_prop_plcy interface */
```

11.16 The rs_prop_replist RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_prop_replist** RPC interface, which provides RS operations to propagate replica list updates from master to slave.

11.16.1 Interface UUID and Version Number for rs_prop_replist

The interface UUID and version number for the **rs_prop_replist** interface are given by the following:

```
[
    uuid(B7FB9CE8-DFD4-11CA-8016-08001E02594C),
    version(1.0),
    pointer_default(ptr)
]
interface rs_prop_replist {
```

11.16.2 rs_prop_replist_add_replica( )

The *rs_prop_replist_add_replica*( ) operation will add or replace a replica on the replist.

```
void
rs_prop_replist_add_replica(
    [in]    handle_t                    rpc_handle,
    [in]    uuid_p_t                    rep_id,
    [in]    rs_replica_name_p_t         rep_name,
    [in]    rs_replica_twr_vec_p_t      rep_twrs,
    [in]    rs_replica_master_info_p_t  master_info,
    [in]    boolean32                   propq_only,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *rep_id* parameter is a pointer to the identifier of the replica to add to the replist.

The *rep_name* parameter is a pointer to the name of the replica to add to the replist.

The *rep_twrs* parameter is a pointer to the base tower of the replica to be added.

The *master_info* parameter is the master replica information.

If the *propq_only* flag is set the replica information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

11.16.3 rs_prop_replist_del_replica( )

The *rs_prop_replist_del_replica*( ) operation will delete a replica from the replist.

```
void
rs_prop_replist_del_replica(
    [in]    handle_t                    rpc_handle,
    [in]    uuid_p_t                    rep_id,
    [in]    rs_replica_master_info_p_t  master_info,
    [in]    boolean32                   propq_only,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *rep_id* parameter is a pointer to the identifier of the replica to add to the replist.

The *master_info* parameter is the master replica information.

If the *propq_only* flag is set the replica information is only placed on the propagation queue. It is not propagated to the security replicas.

The *status* parameter returns the status of the operation.

```
} /* End rs_prop_replist interface */
```

11.17 The rs_pwd_mgmt RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_pwd_mgmt** RPC interface, which provides remote operations for password management between a client and the Security daemon.

11.17.1 Common Data Types and Constants for rs_pwd_mgmt

The following are common data types and constants used in the **rs_pwd_mgmt** interface.

11.17.1.1 rs_pwd_mgmt_plcy_t

The **rs_pwd_mgmt_plcy_t** data type specifies the policy attribute set used by the password management server to determine the password policy.

```
typedef struct {
    unsigned32                              num_plcy_args;
    [size_is(num_plcy_args)]sec_attr_t      plcy[];
} rs_pwd_mgmt_plcy_t;
```

This data type contains the following elements:

- **num_plcy_args**
  The number of policy attribute entries in the **plcy** array.
- **plcy**[ ]
  An array of policy attributes for password management.

11.17.2 Interface UUID and Version Number for rs_pwd_mgmt

The interface UUID and version number for the **rs_pwd_mgmt** interface are given by the following:

```
[
    uuid(3139a0e2-68da-11cd-91c7-080009242444),
    version(1.0),
    pointer_default(ptr)
]
interface rs_pwd_mgmt {
```

11.17.3 rs_pwd_mgmt_setup( )

The *rs_pwd_mgmt_setup*( ) operation retrieves the values stored in the **pwd_val_type** and **pwd_mgmt_binding** extended registry attributes (ERA), if these attributes exist.

```
void
rs_pwd_mgmt_setup (
    [in]      handle_t                rpc_handle,
    [in]      sec_rgy_login_name_t    login_name,
    [out]     sec_attr_bind_info_t    **pwd_mgmt_bind_info,
    [out]     rs_pwd_mgmt_plcy_t      **plcy_p,
    [out,ref] signed32                *pwd_val_type,
    [out]     error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *login_name* parameter specifies the account that is requesting the information.

The *pwd_mgmt_bind_info* parameter specifies binding information contained in the **pwd_mgmt_binding** ERA.

The *plcy_p* parameter contains the attributes that indicate the password policy; that is, the **pwd_val_type** and **pwd_mgmt_binding** ERAs.

The *pwd_val_type* parameter specifies the validation type contained in the **pwd_val_type** ERA, which can be:

- 0 — **none** (user has no password policy)
- 1 — **user_select** (user must choose his or her own password)
- 2 — **user_can_select** (user can either choose his or her own password or request a system-generated password)
- 3 — **generation_required** (user must use a system-generated password)

The *status* parameter returns the status of the operation.

```
} /* End rs_pwd_mgmt interface */
```

11.18 The rs_qry RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_qry** RPC interface.

11.18.1 Interface UUID and Version Number for rs_qry

The interface UUID and version number for the **rs_qry** interface are given by the following:

```
[
    /* V1 format UUID: 3727ee604000.0d.00.00.87.84.00.00.00 */
    uuid(3727EE60-4000-0000-0D00-008784000000),
    version(1)
]
interface rs_query {
```

11.18.2 rs_query_are_you_there()

The *rs_query_are_you_there*( ) operation finds an RS server.

```
[idempotent] void
rs_query_are_you_there (
    [in]  handle_t            rpc_handle,
    [out] error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *status* parameter returns the status of the operation.

```
} /* End rs_query interface */
```

11.19 The rs_repadm RPC Interface

This section specifies (in IDL/NDR) the RS's **rs_repadm** RPC interface that provide RS administration operations.

11.19.1 Common Data Types and Constants for rs_repadm

The following are common data types and constants used in the **rs_repadm** interface.

11.19.1.1 rs_sw_version_t

The **rs_sw_version_t** data type specifies the software version of the name service.

```
typedef unsigned char           rs_sw_version_t[64];
```

11.19.1.2 rs_replica_info_t

The **rs_replica_info_t** data type contains information about each replica.

```
typedef struct
{
    unsigned32              rep_state;
    uuid_t                  cell_sec_id;
    uuid_t                  rep_id;
    uuid_t                  init_id;
    rs_update_seqno_t       last_upd_seqno;
    sec_timeval_t           last_upd_ts;
    rs_sw_version_t         sw_rev;
    unsigned32              compat_sw_rev;
    rs_update_seqno_t       base_propq_seqno;
    boolean32               master;
    boolean32               master_known;
    uuid_t                  master_id;
    rs_update_seqno_t       master_seqno;
} rs_replica_info_t, *rs_replica_info_p_t;
```

This data type contains the following elements:

- **rep_state**
  The current replica state (See Section 11.20.1.3 on page 470).
- **cell_sec_id**
  The cell UUID.
- **rep_id**
  Instance UUID.
- **init_id**
  The UUID of the current master replica.
- **last_upd_seqno**
  The replicas latest update sequence number.
- **last_upd_ts**
  The last sequence update timestamp.
- **sw_rev**
  The replica software revision number.

- **compat_sw_rev**
  The compatible software revision number.
- **base_propq_seqno**
  If this is info from the master, seqno of last update that has been propagated to all replicas. Otherwise, this element is meaningless.
- **master**
  TRUE, if master replica.
- **master_known**
  TRUE, if master is known.
- **master_id**
  The master replica identifier.
- **master_seqno**
  Seqno when master was designated.

11.19.2 Interface UUID and Version Number for rs_repadm

The interface UUID and version number for the **rs_repadm** interface are given by the following:

```
[
    uuid(5b8c2fa8-b60b-11c9-be0f-08001e018fa0),
    version(1.1),
    pointer_default(ptr)
]
interface rs_repadm {
```

11.19.3 rs_rep_admin_stop( )

The *rs_rep_admin_stop*( ) operation stops the replica identified by this handle.

```
void
rs_rep_admin_stop (
    [in]    handle_t            rpc_handle,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *status* parameter returns the status of the operation.

11.19.4 rs_rep_admin_maint( )

The *rs_rep_admin_maint*( ) operation puts the replica in or out of maintenance mode.

```
void
rs_rep_admin_maint(
    [in]    handle_t            rpc_handle,
    [in]    boolean32           in_maintenance,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

If the *in_maintenance* flag is **TRUE** the replica is put into maintenance mode. If **FALSE** it is taken out of maintenance mode.

The *status* parameter returns the status of the operation.

11.19.5 rs_rep_admin_mkey()

The *rs_rep_admin_mkey*( ) operation changes the master key and re-encrypts the database.

```
void
rs_rep_admin_mkey(
    [in]    handle_t            rpc_handle,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *status* parameter returns the status of the operation.

11.19.6 rs_rep_admin_info()

The *rs_rep_admin_info*( ) operation gets basic information about a replica such as its state, UUID, latest update sequence number and timestamp, and whether it is the master. This operation also gets the replica's information about the master's UUID and the sequence number when the master was designated.

```
void
rs_rep_admin_info(
    [in]    handle_t            rpc_handle,
    [out]   rs_replica_info_t   *rep_info,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *rep_info* parameter is a pointer to the replica information returned by call.

The *status* parameter returns the status of the operation.

11.19.7 rs_rep_admin_info_full()

The *rs_rep_admin_info_full*( ) operation gets complete information about a replica such as its state, UUID, protocol towers, latest update sequence number and timestamp, and whether it is the master. This operation also get the replica's information about the master's UUID, protocol towers, and the sequence number when the master was designated.

```
void
rs_rep_admin_info_full(
    [in]    handle_t                rpc_handle,
    [out]   rs_replica_info_t       *rep_info,
    [out]   rs_replica_twr_vec_p_t  *rep_twrs,
    [out]   rs_replica_twr_vec_p_t  *master_twrs,
    [out]   error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *rep_info* parameter is a pointer to the replica information returned by the call.

The *rep_twrs* parameter is a pointer to the replica towers returned by the call.

The *master_twrs* parameter is a pointer to the masters towers returned by the call.

The *status* parameter returns the status of the operation.

11.19.8 rs_rep_admin_destroy( )

The *rs_rep_admin_destroy*( ) operation is a drastic operation which tells a replica to destroy its database and exit.

```
void
rs_rep_admin_destroy(
    [in]    handle_t            rpc_handle,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *status* parameter returns the status of the operation.

11.19.9 rs_rep_admin_init_replica( )

The *rs_rep_admin_init_replica*( ) operation initializes or re-initializes the slave identified by *rep_id*. This is a master-only operation.

```
void
rs_rep_admin_init_replica(
    [in]    handle_t            rpc_handle,
    [in]    uuid_p_t            rep_id,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *rep_id* parameter identifies the slave to be initialized /re-initialized.

The *status* parameter returns the status of the operation.

11.19.10 rs_rep_admin_change_master( )

The *rs_rep_admin_change_master*( ) operation changes the master to *new_master_id*. The master gracefully passes its replica list state and propq to the new master. This is a master-only operation.

```
void
rs_rep_admin_change_master(
    [in]    handle_t            rpc_handle,
    [in]    uuid_p_t            new_master_id,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *new_master_id* parameter is the id of the replica to become the new master.

The *status* parameter returns the status of the operation.

11.19.11 rs_rep_admin_become_master( )

The *rs_rep_admin_become_master*( ) operation is a drastic operation to make a slave become the master because the master has died. Normally, the **rs_rep_admin_change_master**( ) operation is used to designate a new master; this operation can cause updates to be lost.

```
void
rs_rep_admin_become_master(
    [in]    handle_t            rpc_handle,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *status* parameter returns the status of the operation.

11.19.12 rs_rep_admin_become_slave( )

The *rs_rep_admin_become_slave*( ) operation is a drastic operation to make a replica which thinks it's the master become a slave.

```
void
rs_rep_admin_become_slave(
    [in]    handle_t            rpc_handle,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *status* parameter returns the status of the operation.

```
} /* End rs_repadm interface */
```

11.20 The rs_replist RPC Interface

s_replist interfaces

This section specifies (in IDL / NDR) the RS's **rs_replist** RPC interface. The provide RS replica list management services.

11.20.1 Common Data Types and Constants for rs_replist

The following are common data types and constants used in the **rs_replist** interface.

11.20.1.1 rs_replica_item_t and rs_replica_item_p_t

The **rs_replica_item_t** data type contains replica information.

```
typedef struct
{
    uuid_t                      rep_id;
    rs_replica_name_p_t         rep_name;
    boolean32                   master;
    boolean32                   deleted;
    rs_replica_twr_vec_p_t      rep_twrs;
} rs_replica_item_t, *rs_replica_item_p_t;
```

This data type contains the following elements:

- **rep_id**
  The UUID of the replica instance.
- **rep_name**
  The (global) name service name.
- **master**
  If **TRUE**, this is a master replica.
- **deleted**
  If **TRUE**, this replica has been marked as deleted.
- **rep_twrs**
  A pointer to the base replica tower type.

11.20.1.2 Replica States

The **Replica State** describes the current state for each replica.

```
const unsigned32    rs_c_state_unknown_to_master    =   1;
const unsigned32    rs_c_state_uninitialized        =   2;
const unsigned32    rs_c_state_initializing         =   3;
const unsigned32    rs_c_state_in_service           =   4;
const unsigned32    rs_c_state_renaming             =   5;
const unsigned32    rs_c_state_copying_dbase        =   6;
const unsigned32    rs_c_state_in_maintenance       =   7;
const unsigned32    rs_c_state_mkey_changing        =   8;
const unsigned32    rs_c_state_becoming_master      =   9;
const unsigned32    rs_c_state_closed               =   10;
const unsigned32    rs_c_state_deleted              =   11;
const unsigned32    rs_c_state_becoming_slave       =   12;
const unsigned32    rs_c_state_dup_master          =   13;
```

This state contains the following elements:

- **rs_c_state_unknown_to_master**
  The current state of the replica is unknown to the master.
- **rs_c_state_uninitialized**
  The replica remains uninitialized while the database is being created. This is generally a temporary state during creation of a replica.
- **rs_c_state_initializing**
  The replica is currently being inialized by another replica.
- **rs_c_state_in_service**
  The replica is currently in service. The replica may either provide information for clients or become the master.
- **rs_c_state_renaming**
  This state is in effect during a renaming of the replica.
- **rs_c_state_copying_dbase**
  This state is active when the database is in the process of being copied to a new replica or a replica that has requested a new database.
- **rs_c_state_in_maintenance**
  The replica is in maintenance mode.
- **rs_c_state_mkey_changing**
  The current master key is in the process of being changed.
- **rs_c_state_becoming_master**
  When a replica receives the request to become a slave, this state is active.
- **rs_c_state_closed**
  A replica has closed its databases and is in the process of exiting.
- **rs_c_state_deleted**
  The replica has been deleted from the replica list. **rs_c_state_becoming_slave**
  During the time when a slave replica receives a request to become a master this state is active.
- **rs_c_state_dup_master**
  A replica that thinks it is the master has been informed by a slave that the slave believes a different replica to be the legitimate masteer.
-

11.20.1.3 rs_replica_prop_t

The **rs_replica_prop_t** data type specifies the replica propagation state.

```
typedef unsigned32  rs_replica_prop_t;

const rs_replica_prop_t rs_c_replica_prop_init      =   1;
const rs_replica_prop_t rs_c_replica_prop_initing   =   2;
const rs_replica_prop_t rs_c_replica_prop_update    =   3;
const rs_replica_prop_t rs_c_replica_prop_delete    =   4;
```

The following values are currently registered:

- **rs_c_replica_prop_init**
  An initialization request.

- **rs_c_replica_prop_initing**
  Replica is initializing.
- **rs_c_replica_prop_update**
  An update request.
- **rs_c_replica_prop_delete**
  A delete request to a slave.

11.20.1.4 rs_replica_prop_info_t

The **rs_replica_prop_info_t** data type contains information associated with a propagation request.

```
typedef struct
{
    rs_replica_prop_t   prop_type;
    boolean32           last_upd_inited;
    rs_update_seqno_t   last_upd_seqno;
    sec_timeval_t       last_upd_ts;
    unsigned32          num_updates;
} rs_replica_prop_info_t, *rs_replica_prop_info_p_t;
```

This data type contains the following elements:

- **prop_type**
  Type of propagation request.
- **last_upd_inited**
  The last propagation updated has been processed.
- **last_upd_seqno**
  The last propagation sequence number.
- **last_upd_ts**
  The last propagation update time stamp.
- **num_updates**
  Number of sequences updated in this propagation.

11.20.1.5 rs_replica_comm_t

The **rs_replica_comm_t** data type specifies the communication status between master and replica.

```
typedef unsigned32     rs_replica_comm_t;

const rs_replica_comm_t rs_c_replica_comm_ok            =   1;
const rs_replica_comm_t rs_c_replica_comm_short_failure =   2;
const rs_replica_comm_t rs_c_replica_comm_long_failure  =   3;
```

The following values are currently registered:

- **rs_c_replica_comm_ok**
  Communications between replicas is normal.
- **rs_c_replica_comm_short_failure**
  Communications between replicas have failed but have not exceeded the maximum number of retry attempts.

- **rs_c_replica_comm_long_failure**
  Communications between replicas have failed and exceeded the number of retry attempts.

11.20.1.6 rs_replica_comm_info_t

The **rs_replica_comm_info_t** data type contains summary information about the communication state between the master and a replica.

```
typedef struct
{
    rs_replica_comm_t   comm_state;
    error_status_t      last_status;
    signed32            twr_offset;
} rs_replica_comm_info_t, *rs_replica_comm_info_p_t;
```

This data type contains the following elements:

- **comm_state**
  Replica communications state.
- **last_status**
  Last known replica communications status.
- **twr_offset**
  The offset in tower vector to current comm tower. If set to **-1**, there is no current comm tower.

11.20.1.7 rs_replica_item_full_t

The **rs_replica_item_full_t** data type contains public information about a replica. This is the information managed by the master.

```
typedef struct
{
    uuid_t                      rep_id;
    rs_replica_name_p_t         rep_name;
    boolean32                   master;
    boolean32                   deleted;
    rs_replica_prop_info_t      prop_info;
    rs_replica_comm_info_t      comm_info;
    rs_replica_twr_vec_p_t      rep_twrs;
} rs_replica_item_full_t, *rs_replica_item_full_p_t;
```

This data type contains the following elements:

- **rep_id**
  Instance UUID.
- **rep_name**
  The (global) name service name.
- **master**
  If **TRUE**, this is a master replica.
- **deleted**
  If **TRUE**, this replica is marked as deleted.
- **prop_info**
  Propagation information.

- **comm_info**
  Communication information.
- **rep_twrs**
  A pointer to the base replica tower type.

11.20.2 Interface UUID and Version Number for rs_replist

The interface UUID and version number for the **rs_replist** interface are given by the following:

```
[
    uuid(850446B0-E95B-11CA-AD90-08001E0145B1),
    version(1.0),
    pointer_default(ptr)
]
interface rs_replist {
```

11.20.3 rs_replist_add_replica( )

The *rs_replist_add_replica*( ) operation adds a replica to the replica list. This is a master-only operation.

```
void
rs_replist_add_replica(
    [in]    handle_t                rpc_handle,
    [in]    uuid_p_t                rep_id,
    [in]    rs_replica_name_p_t     rep_name,
    [in]    rs_replica_twr_vec_p_t  rep_twrs,
    [out]   error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *rep_id* parameter provides the UUID identifier of the replica to add.

The *rep_name* parameter identifies the string name of the replica. The *rep_twrs* parameter contains a pointer to the replica tower type.

The *status* parameter returns the status of the operation.

11.20.4 rs_replist_replace_replica( )

The *rs_replist_replace_replica*( ) operation replaces information about replica *rep_id* on the replica list. This is a master-only operation.

```
void
rs_replist_replace_replica(
    [in]    handle_t                rpc_handle,
    [in]    uuid_p_t                rep_id,
    [in]    rs_replica_name_p_t     rep_name,
    [in]    rs_replica_twr_vec_p_t  rep_twrs,
    [out]   error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *rep_id* contains the UUID identifier of the replica to be replaced.

The *rep_name* contains the replica name.

The *rep_twrs* parameter contains a pointer to the replica tower type.

The *status* parameter returns the status of the operation.

11.20.5 rs_replist_delete_replica()

The *rs_replist_delete_replica*() operation deletes the replica identified by *rep_id* The master may not be deleted with this operation, which is a master-only operation.

```
void
rs_replist_delete_replica(
    [in]    handle_t                rpc_handle,
    [in]    uuid_p_t                rep_id,
    [in]    boolean32               force_delete,
    [out]   error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *rep_id* parameter identifies the replica to be deleted.

If the *force_delete* parameter is **FALSE**, send the delete to the replica identified by *rep_id* as well as the other replicas. If **TRUE**, do not send the delete to the replica identified by *rep_id*; it has been killed off some other way.

The *status* parameter returns the status of the operation.

11.20.6 rs_replist_read()

The *rs_replist_read*() operation reads the replica list.

```
void
rs_replist_read(
    [in]        handle_t            rpc_handle,
    [in, out]   uuid_t              *marker,
    [in]        unsigned32          max_ents,
    [out]       unsigned32          *n_ents,
    [out, length_is(*n_ents), size_is(max_ents)]
                rs_replica_item_t   replist[],
    [out]       error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *marker* parameter specifies the starting item in the replica list to be read. If *marker* is set to **uuid_nil**, the read starts at the beginning of the replica list. Information about a specific replica can be read by setting *marker* to its UUID and *max_ents* to 1.

The *max_ents* parameter specifies the number of entries to be read.

The *n_ents* parameter specifies the number of entries that have been read.

The *replist*[ ] array is a pointer array of *n_ents* **rs_replica_item_t**s.

The *status* parameter returns the status of the operation.

11.20.7 rs_replist_read_full()

The *rs_replist_read_full*( ) operation reads the replica list, obtaining additional propagation information about each replica.

```
void
rs_replist_read_full(
    [in]       handle_t                rpc_handle,
    [in, out]  uuid_t                  *marker,
    [in]       unsigned32              max_ents,
    [out]      unsigned32              *n_ents,
    [out, length_is(*n_ents), size_is(max_ents)]
               rs_replica_item_full_t  replist[],
    [out]      error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *marker* parameter specifies the starting item in the replica list to be read. If *marker* is set to **uuid_nil**, the read starts at the beginning of the replica list. Information about a specific replica can be read by setting *marker* to its UUID and *max_ents* to 1. As output, *marker* contains the uuid of the next replica on the list. When *marker* contains **uuid_nil**, there are no more replicas on the list.

The *max_ents* parameter specifies the number of entries to be read.

The *n_ents* parameter specifies the number of entries that have been read.

The *replist*[ ] array is a pointer array of *n_ents* **rs_replica_item_t**s.

The *status* parameter returns the status of the operation.

```
} /* End rs_replistinterface */
```

11.21 The rs_repmgr RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_repmgr** RPC interface. The **rs_repmgr** interface provides operations between RS replicas.

11.21.1 Common Data Types and Constants for rs_repmgr

The following are common data types and constants used in the **rs_repmgr** interface.

11.21.1.1 rs_replica_auth_t and rs_replica_auth_p_t

The **rs_replica_auth_t** data type contains authentication information used between replicas.

```
typedef struct
{
    unsigned32                  info_type;
    unsigned32                  info_len;
    [size_is(info_len)] byte    info[];
} rs_replica_auth_t, *rs_replica_auth_p_t;
```

This data type contains the following elements:

- **info_type**
  The Privilege Ticket-Granting Ticket type. The only currently supported type is krb5.
- **info_len**
  The Privilege Ticket-Granting Ticket byte length.
- **info**[ ]
  This is an array of bytes containing the ticket data.

11.21.2 Interface UUID and Version Number for rs_repmgr

The interface UUID and version number for the **rs_repmgr** interface are given by the following:

```
[
    uuid(B62DC198-DFD4-11CA-948F-08001E02594C),
    version(2.0),
    pointer_default(ptr)
]
```

11.21.3 rs_rep_mgr_get_info_and_creds( )

The *rs_rep_mgr_get_info_and_creds*( ) operation gets a replica's basic state information and credentials in order to authenticate to the replica.

```
void
rs_rep_mgr_get_info_and_creds(
    [in]    handle_t              rpc_handle,
    [out]   rs_replica_info_t     *rep_info,
    [out]   rs_replica_auth_p_t   *rep_auth_info,
    [out]   error_status_t        *status );
```

The *rpc_handle* parameter identifies the RS server.

The *rep_info* pointer contains a structure describing the replica.

The *rep_auth_info* contains the information to authenticate the replica.

The *status* parameter returns the status of the operation.

11.21.4 rs_rep_mgr_init()

The *rs_rep_mgr_init*( ) operation tells a replica to initialize itself from another replica.

```
void
rs_rep_mgr_init(
    [in]  handle_t                  rpc_handle,
    [in]  uuid_p_t                  init_id,
    [in]  unsigned32                nreps,
    [in, size_is(nreps)]
          uuid_p_t                  init_from_rep_ids[],
    [in, size_is(nreps)]
          rs_replica_twr_vec_p_t    init_from_rep_twrs[],
    [in]  rs_replica_master_info_p_t master_info,
    [out] uuid_t                    *from_rep_id,
    [out] rs_update_seqno_t         *last_upd_seqno,
    [out] sec_timeval_t             *last_upd_ts,
    [out] error_status_t            *status );
```

The *rpc_handle* parameter identifies the RS server.

The *init_id* parameter identifies the initialize event to prevent redundant initializations.

The *nreps* parameter specifies the number of replicas in the *init_from_rep_ids* array.

The *init_from_rep_ids*[ ] array contains a list of replicas from which the slave is to initialize.

The *init_from_rep_twrs*[ ] array contains a list of replicas from which the slave is to initialize.

The *master_info* parameter contains propagation information held by the master replica.

The *from_rep_id* parameter contains the id of the replica from which the slave has initialized.

The *last_upd_seqno* parameter contains the sequence number of the last update.

The *last_upd_ts* parameter contains the timestamp of the last update.

The *status* parameter returns the status of the operation.

11.21.5 rs_rep_mgr_init_done()

The *rs_rep_mgr_init_done*( ) operation lets a slave tell a master that it has finished initializing itself from another replica.

```
void
rs_rep_mgr_init_done(
    [in]    handle_t            rpc_handle,
    [in]    uuid_p_t            rep_id,
    [in]    uuid_p_t            init_id,
    [in]    uuid_p_t            from_rep_id,
    [in]    rs_update_seqno_t   *last_upd_seqno,
    [in]    sec_timeval_t       *last_upd_ts,
    [in]    error_status_t      *init_st,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *rep_id* parameter specifies the replica which has been initialized.

The *init_id* parameter rs_replist2_master_init_rep_don

The *from_rep_id* parameter contains the UUID of the replica from which the *rep_id* replica has initialized.

The *last_upd_seqno* parameter indicates the last sequence number which was updated.

The *last_upd_ts* parameter indicates the timestamp the last sequence was updated.

The *init_st* parameter indicates whether or not the initialization actually succeeded.

The *status* parameter returns the status of the operation.

11.21.6 rs_rep_mgr_i_am_slave( )

The *rs_rep_mgr_i_am_slave*( ) operation sends a message from a slave to the master when the slave restarts. The slave sends the master its current tower set and software compatability version.

```
void
rs_rep_mgr_i_am_slave(
    [in]    handle_t                    rpc_handle,
    [in]    uuid_p_t                    rep_id,
    [in]    unsigned32                  compat_sw_rev,
    [in]    rs_replica_twr_vec_p_t      rep_twrs,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *rep_id* parameter specifies the UUID of the slave replica that has restarted.

The *compat_sw_rev* parameter contains the software compatibility version number.

The *rep_twrs* parameter is a pointer to the slaves tower set.

The *status* parameter returns the status of the operation.

11.21.7 rs_rep_mgr_i_am_master( )

The *rs_rep_mgr_i_am_master*( ) operation allows a new master to tell a slave that it is now the master.

```
void
rs_rep_mgr_i_am_master(
    [in]    handle_t                    rpc_handle,
    [in]    rs_replica_master_info_p_t  master_info,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *master_info* parameter supplies the slave with all of the necessary master replica information.

The *status* parameter returns the status of the operation.

11.21.8 rs_rep_mgr_become_master( )

The *rs_rep_mgr_become_master*( ) operation lets a master replica tell a slave to become the new master.

```
void
rs_rep_mgr_become_master(
    [in]    handle_t                    rpc_handle,
    [in]    rs_update_seqno_t           base_prop_seqno,
    [in]    rs_replica_master_info_p_t  old_master_info,
    [out]   rs_replica_master_info_t    *new_master_info,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *base_prop_seqno* parameter is the sequence number of the earliest update currently on the prop queue.

The *old_master_info* parameter is the master replica information from the current master.

The *new_master_info* parameter is the master replica information from the new master.

The *status* parameter returns the status of the operation.

11.21.9 rs_rep_mgr_copy_all( )

The *rs_rep_mgr_copy_all*( ) operation is a request sent from one replica to another asking the latter replica to push its entire database to the requesting replica.

```
void
rs_rep_mgr_copy_all(
    [in]      handle_t                    rpc_handle,
    [in]      rs_replica_master_info_p_t  copy_master_info,
    [in]      rs_replica_auth_p_t         rep_auth_info,
    [in, ptr] rs_acct_key_transmit_t      *encryption_key,
    [out]     rs_update_seqno_t           *last_upd_seqno,
    [out]     sec_timeval_t               *last_upd_ts,
    [out]     error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *copy_master_info* parameter contains the master information that the providing replica supplies along with the database updates.

The *rep_auth_info* parameter includes a session key which is used by the two replicas to authenticate one other.

The *encryption_key* parameter is a key (pickled and encrypted with the session key) that the providing replica will use to encrypt the account authentication keys during propagation.

If the update is successful, the *last_upd_seqno* parameter contains the sequence number of the last update.

If the update is successful, the *last_upd_ts* parameter contains the timestamp of the last update.

The *status* parameter returns the status of the operation.

11.21.10 rs_rep_mgr_copy_propq( )

The *rs_rep_mgr_copy_propq*( ) operation carries a request from a slave replica, which is becoming the master, to the old master to send the new master the propagation queue.

```
void
rs_rep_mgr_copy_propq(
    [in]    handle_t          rpc_handle,
    [out]   error_status_t    *status );
```

The *rpc_handle* parameter identifies the RS server.

The *status* parameter returns the status of the operation.

11.21.11 rs_rep_mgr_stop_until_compat_sw( )

The *rs_rep_mgr_stop_until_compat_sw*( ) operation lets a master replica tell a slave not to run until its software has been updated to the software version number contained in *compat_sw_rev*.

```
void
rs_rep_mgr_stop_until_compat_sw(
    [in]    handle_t                    rpc_handle,
    [in]    unsigned32                  compat_sw_rev,
    [in]    rs_replica_master_info_p_t  master_info,
    [out]   error_status_t              *status );
```

The *rpc_handle* parameter identifies the RS server.

The *compat_sw_rev* parameter specifies the software version number that the replica must have in order to run.

The *master_info* parameter specifies all of the master replica information. This is necessary to verify sequence updates.

The *status* parameter returns the status of the operation.

11.22 The rs_rpladmn RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_rpladmn** RPC interface.

11.22.1 Interface UUID and Version Number for rs_rpladmn

The interface UUID and version number for the **rs_rpladmn** interface are given by the following:

```
[
    uuid(5b8c2fa8-b60b-11c9-be0f-08001e018fa0),
    version(1)
]
interface rs_rpladmn {
```

11.22.2 rs_rep_admin_stop( )

The *rs_rep_admin_stop*( ) operation stops the replica identified by *rpc_handle*.

```
void
rs_rep_admin_stop (
    [in]    handle_t            rpc_handle,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifes the replica to be stopped.

The *status* parameter returns the status of the operation.

11.22.3 rs_rep_admin_maint( )

The *rs_rep_admin_maint*( ) operation puts a replica in or out of maintenance mode.

```
void
rs_rep_admin_maint(
    [in]    handle_t            rpc_handle,
    [in]    boolean32           in_maintenance,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifies the replica to be put into (or out of) maintenance mode.

The *in_maintenance* parameter specifies the mode in which the replica is to be placed.

The *status* parameter returns the status of the operation.

11.22.4 rs_rep_admin_mkey( )

The *rs_rep_admin_mkey*( ) operation changes the master key and re-encrypt the database.

```
void
rs_rep_admin_mkey(
    [in]    handle_t            rpc_handle,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *status* parameter returns the status of the operation.

```
} /* End rs_rpladmninterface */
```

11.23 The rs_unix RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_unix** RPC interface.

11.23.1 Common Data Types and Constants for rs_unix

The following are common data types and constants used in the **rs_unix** interface.

11.23.1.1 rs_unix_query_t

The **rs_unix_query_t** data type specifies the criteria used in a query to the RS for UNIX information.

```
typedef enum {
     rs_unix_query_name,
     rs_unix_query_unix_num,
     rs_unix_query_none
} rs_unix_query_t;
```

This data type contains the following elements:

- **rs_unix_query_name**
  Query the RS server for the name of a principal.
- **rs_unix_query_unix_num**
  Query the RS server for the local-ID of a principal.
- **rs_unix_query_none**
  No query criteria.

11.23.1.2 rs_unix_query_key_t

The **rs_unix_query_key_t** data type provides a union to contain the key information for a UNIX query.

```
typedef union switch (rs_unix_query_t query) {
    case rs_unix_query_name:
        struct {
            signed32            name_len;
            sec_rgy_name_t      name;
        } name;
    case rs_unix_query_unix_num:
        signed32                unix_num;
    default:
        ;               /* Empty default branch */
} rs_unix_query_key_t;
```

This data type contains the following elements:

- **name_len**
  The length of the **name** element.
- **name**
  The name of the principal.
- **unix_num**
  The local-ID of the principal.

11.23.1.3 sec_rgy_unix_login_name_t

The **sec_rgy_unix_login_name_t** data type contains a UNIX login name.

```
typedef [string] char sec_rgy_unix_login_name_t[sec_rgy_name_t_size];
```

11.23.1.4 sec_rgy_unix_gecos_t

The **sec_rgy_unix_gecos_t** data type contains UNIX gecos data.

```
typedef [string] char sec_rgy_unix_gecos_t[292];
```

11.23.1.5 sec_rgy_unix_passwd_t

The **sec_rgy_unix_passwd_t** data type contains UNIX account information associated with one entry in the *passwd* data file.

```
typedef struct {
    sec_rgy_unix_login_name_t   name;
    sec_rgy_unix_passwd_buf_t   passwd;
    signed32                    uid;
    signed32                    gid;
    signed32                    oid;
    sec_rgy_unix_gecos_t        gecos;
    sec_rgy_pname_t             homedir;
    sec_rgy_pname_t             shell;
} sec_rgy_unix_passwd_t;
```

This data type contains the following elements:

- **name**
  Principal or UNIX account name.
- **passwd**
  The encrypted representation of the UNIX account password.
- **uid**
  The UNIX user id for the account.
- **gid**
  The UNIX primary group id for the account.
- **oid**
  The account primary organization id.
- **gecos**
  The UNIX gecos data for the account.
- **homedir**
  The UNIX home directory for the account.
- **shell**
  The UNIX primary shell for the account.

11.23.1.6 sec_rgy_member_buf_t

The **sec_rgy_member_buf_t** data type contains a comma-separated ASCII list of group names.

```
typedef [string] char sec_rgy_member_buf_t[sec_rgy_name_t_size * 10];
```

11.23.1.7 sec_rgy_unix_group_t

The **sec_rgy_unix_group_t** data type contains a principal name and a primary and secondary list of group names with which the principal is associated.

```
typedef struct {
    sec_rgy_name_t              name;
    signed32                    gid;
    sec_rgy_member_buf_t        members;
} sec_rgy_unix_group_t;
```

This data type contains the following elements:

- **name**
  The principal name.
- **gid**
  The principal's primary group ID.
- **members**
  The secondary group list of the principal.

11.23.2 Interface UUID and Version Number for rs_unix

The interface UUID and version number for the **rs_unix** interface are given by the following:

```
[
    /* V1 format UUID: 361993c0b000.0d.00.00.87.84.00.00.00 */
    uuid(361993C0-B000-0000-0D00-008784000000),
    version(1)
]
interface rs_unix {
```

11.23.3 rs_unix_getpwents( )

The *rs_unix_getpwents*( ) operation returns an array of UNIX password account entries.

```
[idempotent] void
rs_unix_getpwents (
     [in]     handle_t                rpc_handle,
     [in]     rs_unix_query_key_t     *key,
     [in]     unsigned32              num,
     [in,out] sec_rgy_cursor_t        *cursor,
     [out]    unsigned32              *num_out,
     [out, last_is(*num_out), max_is(num)]
              sec_rgy_unix_passwd_t   result[],
     [out]    rs_cache_data_t         *cache_info,
     [out]    error_status_t          *status );
```

The *rpc_handle* parameter identifies the RS server.

The *key* parameter identifies the entries, either by principal name or by local-ID.

The *num* parameter specifies the size of the *result* array; that is, the maximum number of entries that can be returned by this call.

As input, the *cursor* parameter is an initialized or uninitialized cursor to the RS object. As output, *cursor* is positioned just past the entries returned as output to this call.

The *num_out* parameter is the actual number of result entries returned in the *result* array.

The *result*[ ] array contains UNIX account information associated with UNIX **passwd** data file entries.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *status* parameter returns the status of the operation.

11.23.4 rs_unix_getmemberents( )

The *rs_unix_getmemberents*( ) operation returns an array of UNIX group or organization account entries.

```
[idempotent] void
rs_unix_getmemberents (
     [in]     handle_t               rpc_handle,
     [in]     sec_rgy_domain_t       domain,
     [in]     rs_unix_query_key_t    *key,
     [in]     signed32               max_num_results,
     [in,out] sec_rgy_cursor_t       *member_cursor,
     [in,out] sec_rgy_cursor_t       *cursor,
     [out]    signed32               *num_members,
     [out, last_is(*num_members), max_is(max_num_results)]
              sec_rgy_member_t       members[],
     [out]    rs_cache_data_t        *cache_info,
     [out]    sec_rgy_unix_group_t   *result,
     [out]    error_status_t         *status );
```

The *rpc_handle* parameter identifies the RS server.

The *domain* parameter specifies whether the entries are person, group, or organization entries.

The *key* parameter identifies the entries, either by name or by local-ID.

The *max_num_results* parameter specifies the size of the *members* array; that is, the maximum number of entries that can be returned by this call.

As input, the *member_cursor* parameter is an initialized or uninitialized cursor to the member list. As output, *member_cursor* is positioned just past the current member entry returned as output to this call.

As input, the *cursor* parameter is an initialized or uninitialized cursor to the attribute list. As output, *cursor* is positioned just past the attributes returned as output to this call.

The *num_members* parameter is the actual number of member entries returned in the *members* array.

The *members*[ ] array contains member entries.

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

The *result* parameter is a UNIX group describing a principal name and the associated primary and secondary group list.

The *status* parameter returns the status of the operation.

```
} /* End rs_unix interface */
```

11.24 The rs_update RPC Interface

This section specifies (in IDL / NDR) the RS's **rs_update** RPC interface. This interface is registered in the nameservice by the master replica so that clients can locate the master.

11.24.1 Interface UUID and Version Number for rs_update

The interface UUID and version number for the **rs_update** interface are given by the following:

```
[
    uuid(3B11D6A8-2A9C-11CB-BE8A-08001E0238CA),
    version(1.0),
    pointer_default(ptr)
]
interface rs_update {
```

11.24.2 rs_rep_admin_info()

The *rs_rep_admin_info*( ) operation retrieves basic information about a replica.

```
void
rs_rep_admin_info(
    [in]    handle_t            rpc_handle,
    [out]   rs_replica_info_t   *rep_info,
    [out]   error_status_t      *status );
```

The *rpc_handle* parameter identifies the RS server.

The *rep_info* parameter contains the information about the replica, including its state, UUID, latest update sequence number and timestamp, whether the replica is a master replica, and information about the replica's master.

The *status* parameter returns the status of the operation.

```
} /* End rs_update interface */
```

Chapter 12

ID Map Facility RPC Interface

This chapter specifies the RPC interface supporting the *ID Map Facility*, namely the **secidmap** RPC interface (the corresponding **ID Map** (or **sec_id**) API is specified in Chapter 17). See Section 1.13 on page 67 for the background to this chapter.

12.1 The secidmap RPC Interface

This section specifies (in IDL / NDR) the **secidmap** RPC interface, which is supported by every RS server.

12.1.1 Common Data Types and Constants for the secidmap Interface

The following are common data types and constants used in the **secidmap** interface.

12.1.1.1 rsec_id_output_selector_t

The **rsec_id_output_selector_t** data type is used to control the services of the **secidmap** interface operations.

```
typedef bitset       rsec_id_output_selector_t;
const unsigned32     rsec_id_output_select_gname = 0x1;
const unsigned32     rsec_id_output_select_cname = 0x2;
const unsigned32     rsec_id_output_select_pname = 0x4;
const unsigned32     rsec_id_output_select_cuuid = 0x8;
const unsigned32     rsec_id_output_select_puuid = 0x10;
```

The following values are currently registered:

- **rsec_id_output_select_gname**
  Return global PGO name.
- **rsec_id_output_select_cname**
  Return cell name.
- **rsec_id_output_select_pname**
  Return cell-relative PGO (principal, group or organisation) name.
- **rsec_id_output_select_cuuid**
  Return cell UUID.
- **rsec_id_output_select_puuid**
  Return PGO (principal, group or organisation) UUID.

Selectors (that is, parameters of type **rsec_id_output_selector_t**) occurring in the operations of this chapter are to some extent considered to be ‘‘hints’’, informing the RS server what services the client is interested in (that is, what information an operation should return in order to be considered successful). Namely, selected bits (that is, set to 1) indicate information that the client *is* interested in, and unselected bits (that is, set to 0) indicate information that the client is *not* interested in.

However, there are some situations in which the RS server does or does not return the requested information to clients — without reflecting an error to the client. For example, certain selections do not make sense for certain operations, so the RS server does not return selected information

in those cases (this is indicated in the descriptions of the operations, below). Furthermore, in the case of foreign cell referrals (see Section 12.1.1.2), RS servers are required to always return certain information, whether the client has selected that information or not (this is also indicated in the descriptions below). Finally, the RS server does not consider it an error if no bits at all are selected by the client, or if bits that are currently unregistered are selected (this is not repeated in the descriptions below).

12.1.1.2 Global PGO Names

Throughout this chapter the notion of *global PGO name* is used to mean a stringname of one of the following (fully-qualified or partially-qualified) forms:

- **/.../***cell-name/cell-relative-pgo-name*
- **/.:/***cell-relative-pgo-name*
- *cell-relative-pgo-name* (see Section 11.2.4 on page 361).

Of course, the latter two forms listed here are to be interpreted in the context of (relative to the home cell of) the user of these names.

Note that it is impossible to tell from the mere syntax of a global PGO name whether it names a principal or group or organisation (it may even name all three) — to disambiguate it, a specified PGO naming domain (see Section 11.5.1.1 on page 379) is required. (Concerning naming syntax, see also Section 1.18 on page 84.)

12.1.1.3 Status Codes

The following status codes (transmitted as values of the type **error_status_t**) are specified for the **secidmap** interface. Only their values and a short one-line description of them are specified here — their detailed usage is specified in context elsewhere in this book.

Note: In cases where exception statuses are not currently described in this specification, it is intended to supply them in the next revision of DCE.

Values:

```
const unsigned32  sec_id_e_name_too_long         = 0x171220d4;
const unsigned32  sec_id_e_bad_cell_uuid         = 0x171220d5;
const unsigned32  sec_id_e_foreign_cell_referral = 0x171220d6;
```

Descriptions:

- **sec_id_e_name_too_long**
  A presented name is too long to be processed by RS server's internal datastore implementation.

  For example, the DCE global naming syntax uses an initial **/.../**, but this may be converted for storage in the RS datastore (depending on the implementation) with an initial **krbtgt/** instead; in this case, 2 extra characters are needed to represent the name, so if the name were already at the RS datastore's length limit prior to conversion, it would exceed the limit after the conversion. (In practical implementations, this event is negligible.)

- **sec_id_e_bad_cell_uuid**
  A presented cell UUID is not known to an RS server.

- **sec_id_e_foreign_cell_referral**
  This status code is not considered a hard failure error; rather, it triggers a *foreign cell referral* action to occur, as follows. (See also Section 1.11 on page 55 for the general idea of a *junction architecture* for *federated* namespace organisation.)

Namely, when an RS server processes a client request naming a (purported) PGO item that the RS server does not hold in its own datastore, but which it does hold a referral to (referring to another RS server), it resolves the name as far as it can, and returns to the client the required referral information together with a **sec_id_e_foreign_cell_referral** status code. The client is then expected to retry the operation at the referred-to RS server. This ritual may be iterated.

Commonly, this referral activity arises when an RS server detects that a prefix of a presented name appears as a *KDS principal* or *cell principal* (that is, a principal which is a descendant of the RS server's **krbtgt** top-level directory of the principal domain). In this case the RS server returns this foreign cell name to the client with a **sec_id_e_foreign_cell_referral** status code, so the client can bind to an RS server in the foreign cell and retry the operation. (For RS binding, see Section 1.12.2 on page 61 and Chapter 16.)

This discussion of referrals has been a general one, necessarily leaving some details vague. Those details are to be supplied in explicit instances of referrals (as, for example, the operations specified in this chapter).

12.1.2 Interface UUID and Version Number for the secidmap Interface

The interface UUID and version number for the **secidmap** interface are given by the following:

```
[uuid(0d7c1e50-113a-11ca-b71f-08001e01dc6c), version(1.0)]
interface secidmap
{
/* begin running listing of secidmap interface */
```

12.1.3 rsec_id_parse_name( )

The *rsec_id_parse_name*( ) operation parses a global PGO name into its cell name and a cell-relative PGO name constituents, together with the UUIDs associated with them (depending on selected options).

```
void
rsec_id_parse_name (
    [in]        handle_t                    rpc_handle,
    [in]        sec_rgy_domain_t            name_domain,
    [in]        sec_rgy_name_t              global_name,
    [in]        rsec_id_output_selector_t   selector,
    [out]       sec_rgy_name_t              cell_namep,
    [out]       uuid_t                      *cell_idp,
    [out]       sec_rgy_name_t              princ_namep,
    [out]       uuid_t                      *princ_idp,
    [out]       error_status_t              *status );
```

The *rpc_handle* parameter identifies the target RS server.

The *name_domain* parameter indicates the PGO domain of interest.

The *global_name* parameter indicates the global PGO name to be parsed.

The *selector* parameter indicates the information to be returned:

- If the **rsec_id_output_select_gname** bit is selected, it is ignored.
- If the **rsec_id_output_select_cname** bit is selected, then the parsed cell name is returned in *cell_namep*.

- If the **rsec_id_output_select_pname** bit is selected, then the parsed cell-relative PGO name is returned in *princ_namep*.
- If the **rsec_id_output_select_cuuid** bit is selected, then the parsed cell UUID is returned in *cell_idp*.
- If the **rsec_id_output_select_puuid** bit is selected, then the parsed PGO UUID is returned in *princ_idp*.

The *cell_namep* parameter indicates the parsed cell name. In the case of a foreign cell referral, it indicates the referred-to foreign cell.

The *cell_idp* parameter indicates the parsed cell UUID.

The *princ_namep* parameter indicates the parsed cell-relative PGO name.

The *princ_idp* parameter indicates the parsed PGO name.

The *status* parameter returns the status of the operation.

Required rights: No permissions are required, unless the **rsec_id_output_select_puuid** bit is selected; in that case, this operation succeeds only if the calling client has *some* permission (of any kind) on the PGO item indicated by *name_domain* and *global_name* (and if the client does not have such permission, the operation fails completely; that is, the RS server does not fill in all but the *princ_idp* parameter).

12.1.4 rsec_id_gen_name( )

The *rsec_id_gen_name*( ) operation generates a global PGO name, and parses it into its associated cell name and cell-relative name, from specified cell and PGO UUIDs (depending on selected options).

```
void
rsec_id_gen_name (
    [in]        handle_t                    rpc_handle,
    [in]        sec_rgy_domain_t            name_domain,
    [in]        uuid_t                      *cell_idp,
    [in]        uuid_t                      *princ_idp,
    [in]        rsec_id_output_selector_t   selector,
    [out]       sec_rgy_name_t              global_name,
    [out]       sec_rgy_name_t              cell_namep,
    [out]       sec_rgy_name_t              princ_namep,
    [out]       error_status_t              *status );
```

The *rpc_handle* parameter identifies the target RS server.

The *name_domain* parameter indicates the PGO domain of interest.

The *cell_idp* parameter indicates the cell UUID.

The *princ_idp* parameter indicates the PGO UUID.

The *selector* parameter indicates the information to be returned:

- If the **rsec_id_output_select_gname** bit is selected, then the generated global PGO name is returned in *global_name*.
- If the **rsec_id_output_select_cname** bit is selected, then the generated cell name is returned in *cell_namep*.

- If the **rsec_id_output_select_pname** bit is selected, then the generated cell-relative PGO name is returned in *princ_namep*.
- If the **rsec_id_output_select_cuuid** bit is selected, it is ignored.
- If the **rsec_id_output_select_puuid** bit is selected, it is ignored.

The *global_name* parameter indicates the generated global PGO name.

The *cell_namep* parameter indicates the generated cell name. In the case of a foreign cell referral, it indicates the referred-to foreign cell.

The *princ_namep* parameter indicates the generated PGO name.

The *status* parameter returns the status of the operation.

Note: A **sec_id_e_foreign_cell_referral** status is generated by this operation only if one of the bits **rsec_id_output_select_gname** or **rsec_id_output_select_pname** is selected. Otherwise, the RS server either already holds the required information, or it doesn't hold enough information to generate a referral.

Required rights: This operation succeeds only if the calling client has *some* permission (of any kind) on the PGO item determined by the specified UUIDs (*cell_idp*, *princ_idp*).

12.1.5 rsec_id_parse_name_cache( )

The *rsec_id_parse_name_cache*( ) operation is identical to *rsec_id_parse_name*( ), with the addition of cache management.

```
void
rsec_id_parse_name_cache (
    [in]        handle_t                    rpc_handle,
    [in]        sec_rgy_domain_t            name_domain,
    [in]        sec_rgy_name_t              global_name,
    [in]        rsec_id_output_selector_t   selector,
    [out]       sec_rgy_name_t              cell_namep,
    [out]       uuid_t                      *cell_idp,
    [out]       sec_rgy_name_t              princ_namep,
    [out]       uuid_t                      *princ_idp,
    [out]       rs_cache_data_t             *cache_info,
    [out]       error_status_t              *status );
```

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

Required rights: Same as *rsec_id_parse_name*( ).

12.1.6 rsec_id_gen_name_cache( )

The *rsec_id_gen_name_cache*( ) operation is identical to *rsec_id_gen_name*( ), with the addition of cache management.

```
void
rsec_id_gen_name_cache (
    [in]        handle_t                    rpc_handle,
    [in]        sec_rgy_domain_t            name_domain,
    [in]        uuid_t                      *cell_idp,
    [in]        uuid_t                      *princ_idp,
    [in]        rsec_id_output_selector_t   selector,
    [out]       sec_rgy_name_t              global_name,
    [out]       sec_rgy_name_t              cell_namep,
    [out]       sec_rgy_name_t              princ_namep,
    [out]       rs_cache_data_t             *cache_info,
    [out]       error_status_t              *status );

} /* end running listing of secidmap interface */
```

The *cache_info* parameter indicates cache information (see Section 11.2.8 on page 363).

Required rights: Same as *rsec_id_gen_name*( ).

Chapter 13

Key Management Facility RPC Interface

This chapter specifies the RPC interface supporting the *Key Management Facility* (the corresponding *Key Management* — or **sec_key_mgmt** — API is specified in Chapter 18). See Section 1.14 on page 69 for the background to this chapter.

13.1 The Key Management RPC Interface

There are no special RPC interfaces (beyond those already specified elsewhere in this specification) to support the Key Management Facility.

13.1.1 Common Data Types and Constants for Key Management

The following are common data types and constants used for key management.

13.1.1.1 Status Codes

The following status codes (transmitted as values of the type **error_status_t**) are specified for key management. Only their values are specified here — their use is specified in context elsewhere in this specification.

```
const unsigned32  sec_key_mgmt_e_key_unavailable  = 0x17122043;
const unsigned32  sec_key_mgmt_e_authn_invalid    = 0x17122044;
const unsigned32  sec_key_mgmt_e_auth_unavailable = 0x17122045;
const unsigned32  sec_key_mgmt_e_unauthorized     = 0x17122046;
const unsigned32  sec_key_mgmt_e_key_unsupported  = 0x17122047;
const unsigned32  sec_key_mgmt_e_key_version_ex   = 0x17122048;
const unsigned32  sec_key_mgmt_e_not_implemented  = 0x17122049;
const unsigned32  sec_key_mgmt_e_keytab_not_found = 0x1712204a;
const unsigned32  sec_key_mgmt_e_ktfile_err       = 0x1712204b;
```

Chapter 14

Login Facility and Security Client Daemon (SCD) RPC Interface

This chapter specifies the RPC interface supporting the *Login Facility*, namely the **scd** RPC interface supported by the Security Client Daemon (the corresponding **Login** (or **sec_login**) API is specified in Chapter 19). See Section 1.15 on page 71 for the background to this chapter.

14.1 The scd RPC Interface

This section specifies (in IDL / NDR) the SCD's **scd** RPC interface.

14.1.1 Common Data Types and Constants for scd Interface

The following are common data types and constants used in the Login Facility and **scd** RPC interface.

14.1.1.1 Status Codes

The following status codes (transmitted as values of the type **error_status_t**) are specified for the Login Facility and **scd** RPC interface. Only their values are specified here — their use is specified in context elsewhere in this specification.

```
const unsigned32  sec_login_s_no_memory          = 0x171220e8;
const unsigned32  sec_login_s_auth_local         = 0x171220e9;
const unsigned32  sec_login_s_handle_invalid     = 0x171220ea;
const unsigned32  sec_login_s_context_invalid    = 0x171220eb;
const unsigned32  sec_login_s_no_current_context = 0x171220ec;
const unsigned32  sec_login_s_groupset_invalid   = 0x171220ed;
const unsigned32  sec_login_s_info_not_avail     = 0x171220ee;
const unsigned32  sec_login_s_already_valid      = 0x171220ef;
const unsigned32  sec_login_s_default_use        = 0x171220f0;
const unsigned32  sec_login_s_privileged         = 0x171220f1;
const unsigned32  sec_login_s_not_certified      = 0x171220f2;
const unsigned32  sec_login_s_config             = 0x171220f3;
const unsigned32  sec_login_s_internal_error     = 0x171220f4;
const unsigned32  sec_login_s_acct_invalid       = 0x171220f6;
const unsigned32  sec_login_s_null_password      = 0x171220f7;
const unsigned32  sec_login_s_unsupp_passwd_type = 0x171220f8;
const unsigned32  sec_login_s_refresh_ident_bad  = 0x171220fa;
```

14.1.2 Interface UUID and Version Number for scd Interface

The interface UUID and version number for the **scd** interface are given by the following:

```
[uuid(c57e83f0-58be-11ca-901c-08001e039448), version(1.0)]
interface scd
```

14.1.3 scd_protected_noop()

The *scd_protected_noop*( ) operation determines whether or not the calling client can ''successfully'' (in the sense of authentication) execute a protected ''dummy'' operation (actually, a ''no-op'') on an SCD server — that is, whether or not the client and SCD server are authenticated to one another. This operation is used to support the notion of *certification* (see Section 1.15.2 on page 77): to the extent that the client trusts that its invocation of *scd_protected_noop*( ) has actually been handled by the genuine SCD server on its local host (which is in the local host's TCB), successful execution of this operation has the semantic of ''certifying'' (in the sense of Section 1.15.2 on page 77) to the client the login context it used in invoking *scd_protected_noop*( ).

```
{ /* begin running listing of scd interface */
    void
    scd_protected_noop (
        [in]        handle_t                    rpc_handle,
        [out]       error_status_t              *status );

} /* end running listing of scd interface */
```

The *rpc_handle* parameter identifies the SCD server.

The *status* parameter returns the status of the operation. (See description below.)

Required rights: None (but see below).

The SCD's handler (manager routine) of *scd_protected_noop*( ) always returns **error_status_ok** in the *status* parameter (though this may not always be returned to the client; it may be overridden in the RPC runtime system; for example, by an authentication failure). Similarly, no specific permissions are required by the manager routine itself; however, the SCD server registers itself (see *rpc_server_register_auth_info*( ) in the referenced X/Open DCE RPC Specification) under its principal name and with an appropriate authentication service (only **rpc_c_authn_dce_secret** (Kerberos) is currently supported) and authorisation service (**rpc_c_authz_dce**, so that the client's PAC is transmitted to the server), and the client invokes *scd_protected_noop*( ) on a binding that is protected (see *rpc_binding_set_auth_info*( ) in the referenced X/Open DCE RPC Specification) at protection level **rpc_c_protect_level_pkt_integ** — it is the responsibility of the RPC runtime system to report to the client (via the *status* parameter) whether or not this operation succeeds (that is, whether the client and SCD server are authenticated to one another via the same authentication service (Kerberos)).

CAE Specification

Part 3
Security Application Programming Interface

The Open Group

Chapter 15

Access Control List API

15.1 Introduction

The routines in the ACL Editor API are distinguished with names having the prefix ‘‘**sec_acl_**’’.

Background is given in Chapter 1, especially Section 1.11 on page 55.

Note: The **sec_acl** API is designed to be a general programming interface for managing all ACLs in such a way that the client is unaware of the principal identity of the server that controls the objects protected by the ACLs. As such, the server's principal name does not occur as a parameter to the **sec_acl** API (see, for example, *sec_acl_bind*( )). This implies, in particular, that the **sec_acl** API supports only *one-way* (client-to-server) authentication, not mutual (server-to-client) authentication. Applications that require mutual authentication should use the ‘‘raw’’ **rdacl** RPC protocol, not the **sec_acl** API. (Mutual authentication may be added to the **sec_acl** API in a future revision of DCE.)

NAME

<dce/aclbase.h> — Header for **sec_acl** API.

SYNOPSIS

```
#include <dce/aclbase.h>
```

DESCRIPTION

Data Types and Constants

The following data types (listed in alphabetical order) are used in the **sec_acl** API.

unsigned char *sec_acl_component_name_t
Server-supported namespace component.

struct sec_acl_entry_t
This data type represents an ACLE. It contains the following fields:

sec_acl_permset_t perms
The permissions granted to the principals identified by this ACL entry.

struct entry_info
Identifies the principals to which this ACLE ‘‘applies’’ (that is, which ‘‘match’’ this ACLE for the purposes of an access decision). It contains the following fields:

sec_acl_entry_type_t entry_type
The type of this ACLE.

union tagged_union
Information further identifying (or ‘‘tagging’’) this ACLE. It contains the following fields:

sec_id_t id
Local principal, local group or foreign cell to which this ACLE applies. This union arm is selected if **entry_type** is **sec_acl_e_type_user**, **sec_acl_e_type_group**, **sec_acl_e_type_foreign_other** **sec_acl_e_type_user_deleg**, **sec_acl_e_type_group_deleg**, or **sec_acl_e_type_for_other_deleg**.

sec_id_foreign_t foreign_id
Foreign principal or foreign group to which this ACLE applies. This union arm is selected if **entry_type** is **sec_acl_e_type_foreign_user**, **sec_acl_e_type_foreign_group**, **sec_acl_e_type_for_user_deleg**, or **sec_acl_e_type_for_group_deleg**.

sec_acl_extend_info_t *extended_info
Contents of an extended ACLE. This union arm is selected if **entry_type** is **sec_acl_e_type_extended**.

/*empty*/
The **tagged_union** field contains no valid information for any other value of **entry_type**.

enum sec_acl_entry_type_t
The ACLE type of an ACLE. It can take the following values (see Section 1.8.1 on page 40 for discussion):

sec_acl_e_type_user_obj
USER_OBJ

sec_acl_e_type_group_obj
GROUP_OBJ

sec_acl_e_type_other_obj
OTHER_OBJ

sec_acl_e_type_user_obj_deleg
USER_OBJ_DEL

sec_acl_e_type_group_obj_deleg
GROUP_OBJ_DEL

sec_acl_e_type_other_obj_deleg
OTHER_OBJ_DEL

sec_acl_e_type_user
USER

sec_acl_e_type_group
GROUP

sec_acl_e_type_user_deleg
USER_DEL

sec_acl_e_type_group_deleg
GROUP_DEL

sec_acl_e_type_mask_obj
MASK_OBJ

sec_acl_e_type_foreign_user
FOREIGN_USER

sec_acl_e_type_foreign_group
FOREIGN_GROUP

sec_acl_e_type_foreign_other
FOREIGN_OTHER

sec_acl_e_type_for_user_deleg
FOREIGN_USER_DEL

sec_acl_e_type_for_group_deleg
FOREIGN_GROUP_DEL

sec_acl_e_type_for_other_deleg
FOREIGN_OTHER_DEL

sec_acl_e_type_any_other
ANY_OTHER

sec_acl_e_type_unauthenticated
UNAUTHENTICATED

sec_acl_e_type_extended
EXTENDED

struct sec_acl_extend_info_t
Extended ACL information (see Section 7.1.4 on page 313 for discussion). It contains the following fields:

uuid_t extension_type
The type of extension this is, indicating to ACL managers whether or not they can interpret it. (ACL managers must reject any extended ACLEs they cannot interpret.)

ndr_format_t format_label
NDR format label.

unsigned32 num_bytes
Number of bytes in **pickled_data[]** array.

unsigned char pickled_data[]
The actual extended ACL information itself.

sec_acl_handle_t
An opaque (to the client) data type representing a handle to a protected object. The handle is bound to the protected object with *sec_acl_bind*( ) or *sec_acl_bind_to_addr*( ). The distinguished value **sec_acl_default_handle** signifies an unbound handle.

sec_acl_id_t
This data type is equivalent to the **sec_id_t** data type (that is, they may be used interchangeably).

unsigned32 sec_acl_permset_t
Permission bits. The following values are currently defined (see Section 1.9 on page 46 for discussion):

sec_acl_perm_read
Read. (Conventional value: 0x00000001.)

sec_acl_perm_write
Write. (Conventional value: 0x00000002.)

sec_acl_perm_execute
Execute. (Conventional value: 0x00000004.)

sec_acl_perm_control
Control (or Change, or Write-ACL). (Conventional value: 0x00000008.)

sec_acl_perm_insert
Insert. (Conventional value: 0x00000010.)

sec_acl_perm_delete
Delete. (Conventional value: 0x00000020.)

sec_acl_perm_test
Test. (Conventional value: 0x00000040.)

sec_acl_perm_unused_00000080 to **sec_acl_perm_unused_80000000**
Application-defined. There are 25 of these bits, the last 8 characters of whose names correspond to the bit-value identifiers 0x00000080–0x80000000 (and which by convention have these same bit-values).

struct sec_acl_t
This data type represents an ACL. It contains the following fields:

sec_acl_id_t default_realm
The default cell (or realm) for this ACL.

uuid_t sec_acl_manager_type
The ACL manager that can interpret this ACL.

unsigned32 num_entries
Number of ACLEs in this ACL.

sec_acl_entry_t *sec_acl_entries[]
An array containing **num_entries** pointers to the ACLEs of this ACL.

struct sec_acl_list_t
A list of ACLs. It contains the following fields:

unsigned32 num_acls
The number of ACLs contained in this list.

sec_acl_p_t sec_acls[]
Pointers to the actual ACLs in this list.

sec_acl_t *sec_acl_p_t
Pointer to a **sec_acl_t**.

struct sec_id_foreign_t
Identities of ''foreign'' entities (see Section 5.2.2 on page 279). It contains the following fields:

sec_id_t id
Identifier of the entity within its cell.

sec_id_t realm
Identifier of the entity's cell (or ''realm'' in security-specific terminology).

struct sec_id_t
Identities of cells and ''local'' entities, suitable for DCE authorisation architecture (see Section 5.2.1 on page 277). (Compare **sec_id_foreign_t**.) It contains the following fields:

uuid_t uuid
Definitive identifier of the entity.

unsigned char *name
Advisory (''optional'') identifier of the entity.

struct sec_acl_printstring_t
Information about permission bits, and about ACL managers as a whole (see Section 8.1.2 on page 319 and Section 10.1.10 on page 352 for discussion). It contains the following fields:

unsigned char *printstring
Printstring (a character string of maximum length **signed32 sec_acl_printstring_len**).

unsigned char *helpstring
Helpstring (a character string of maximum length **signed32 sec_acl_printstring_help_len**).

sec_acl_permset_t permissions
Bit representation of permission(s).

enum sec_acl_type_t
The ACL's type (see Section 1.8.2 on page 44 for discussion). The following values are currently defined:

sec_acl_type_object
Protection ACL.

sec_acl_type_default_object
Default object creation ACL.

sec_acl_type_default_container
Default container creation ACL.

sec_acl_type_unspecified_3, ···, **sec_acl_type_unspecified_7**
Application defined. (There are 5 of these identifiers; each is 26 characters long. Their first 25 characters are ''**sec_acl_type_unspecified_**'', and their last characters are, respectively: ''**3**'', ''**4**'', ''**5**'', ''**6**'', ''**7**''.)

Status Codes

The following status codes (listed in alphabetical order) are used in the **sec_acl** API.

sec_acl_bad_acl_syntax
ACL has invalid semantics (not ''syntax'').

sec_acl_bad_key
The ACLE tag (key) is not valid.

sec_acl_bad_parameter
Parameter passed is invalid.

sec_acl_bind_error
Unable to get binding to protected object.

sec_acl_cant_allocate_memory
Requested operation requires more memory than is available.

sec_acl_duplicate_entry
ACL has duplicate entries.

sec_acl_expected_group_obj
ACLE is not of type GROUP_OBJ.

sec_acl_expected_user_obj
ACLE is not of type USER_OBJ.

sec_acl_invalid_entry_name
Requested namespace entry is invalid. For example, purported component name contains an illegal character.

sec_acl_invalid_entry_type
ACLE type is not valid.

sec_acl_invalid_manager_type
Manager type is not valid.

sec_acl_invalid_permission
Permissions for this ACL are invalid.

sec_acl_invalid_site_name
Site (server instance) name is not valid.

sec_acl_invalid_acl_type
ACL type is not valid.

sec_acl_missing_required_entry
ACL is missing a required entry.

sec_acl_name_resolution_failed
Name requested in the operation cannot be resolved.

sec_acl_no_acl_found
Requested ACL was not present.

sec_acl_no_owner
Requested operation requires owner permission.

sec_acl_not_authorized
Requested operation is not allowed.

sec_acl_not_implemented
Unwilling to perform requested operation (or, colloquially, requested operation has ‘‘not been implemented’’).

sec_acl_no_update sites
No update site for this ACL operation.

sec_acl_object_not_found
Requested protected object could not be found.

sec_acl_read_only
ACL is read-only.

sec_acl_rpc_error
Operation requested failed in RPC.

sec_acl_site_read_only
ACL is read-only at this site.

sec_acl_unable_to_authenticate
Requested operation requires authentication.

sec_acl_unknown_manager_type
Manager type selected is not an available option.

sec_invalid_acl_handle
ACL binding handle is invalid.

NAME

sec_acl_bind — Obtain (''bind'') handle to a protected object identified by name.

SYNOPSIS

```
#include <dce/daclif.h>

void sec_acl_bind(
        unsigned char *name,
        boolean32 bind_to_namespace_entry,
        sec_acl_handle_t *prot_obj_handle,
        error_status_t *status);
```

PARAMETERS

Input

name

Full name (a CDS namespace entry name concatenated with a server-supported namespace name) of the protected object to which a security handle is desired.

bind_to_namespace_entry

Boolean switch, for disambiguating the cases where *name* ambiguously refers to both a (leaf) entry in the DCE namespace (as for protected object managed by a DCE namespace server), and also an application-level (that is, non-DCE-namespace-)server-supported protected object (the root of a server-supported namespace). If non-0 (''true''), the DCE namespace entry is indicated; if 0 (''false''), the (non-DCE namespace) server's protected object is indicated.

Output

prot_obj_handle

Handle to the specified protected object.

status

Completion status. On successful completion, **error_status_ok** is returned. Otherwise, an error (≠ **error_status_ok**) is returned.

DESCRIPTION

The *sec_acl_bind*( ) routine returns an opaque (to the client) handle, bound to (that is, referring to) the protected object indicated by *name*. This handle is used subsequently by other **sec_acl** routines to refer to the protected object (instead of referring to it by *name*).

NOTES

If the specified *name* is a ''junction point'' between the DCE namespace and an application server's namespace of protected objects (that is, *name* is the application server's registered/exported RPC server entry in the DCE namespace), then *name* ambiguously identifies two protected objects: the (leaf) DCE namespace entry itself, and the protected object at the root of the server's namespace of protected objects (that is, the server's protected object with empty stringname). The *bind_to_namespace_entry* flag resolves such an ambiguity. Note that if *name* refers to a DCE namespace internal node (that is, to a DCE namespace directory, not a leaf node), then there is no ambiguity (the protected object to which a handle is returned is the DCE directory, managed by a DCE namespace server).

Implementations of *sec_acl_bind*( ) must be based on a namespace ''resolution-with-residual'' runtime support routine that resolves a full name to the junction point in the namespace, and returns to the client the unresolved, ''residual'', part of the name, supported by the application server. The client then queries the resolved name for the server's binding information, binds to

the server, and presents to it the residual name for the server's internal resolution. Such a suitable CDS namespace runtime support routine is provided by *rpc_ns_entry_inq_resolution*( ).

ERRORS

error_status_ok, **sec_acl_object_not_found**, **sec_acl_no_acl_found**.

SEE ALSO

Functions: *sec_acl_bind_to_addr*( ), *sec_acl_release_handle*( ).

Protocols: *rpc_ns_entry_inq_resolution*( ), *rpc_ns_binding_**( ).

NAME

fsec_acl_bind_to_addr — Obtain (''bind'') handle to a protected object identified by address and name.

SYNOPSIS

```
#include <dce/daclif.h>

void sec_acl_bind_to_addr(
        unsigned char *addr,
        sec_acl_component_name_t component_name,
        sec_acl_handle_t *prot_obj_handle,
        error_status_t *status);
```

PARAMETERS

Input

addr
Fully qualified RPC string binding (''address'') to the server managing the protected object to which a security handle is desired.

component_name
Server-supported namespace name of the protected object to which a security handle is desired.

Output

prot_obj_handle
Handle to the specified protected object.

status
Completion status. On successful completion, **error_status_ok** is returned. Otherwise, an error (≠ **error_status_ok**) is returned.

DESCRIPTION

The *sec_acl_bind_to_addr*( ) routine is identical to *sec_acl_bind*( ), except that it identifies the protected object by server address and server-supported name (*addr* and *component_name*), instead of by full name.

NOTES

Unlike *sec_acl_bind*( ), there can be no ambiguity about the protected object that *sec_acl_bind_to_addr*( ) refers to, because the target server is referred to unambiguously by its address (RPC string binding).

ERRORS

error_status_ok, **sec_acl_object_not_found**, **sec_acl_no_acl_found**, **sec_acl_unable_to_authenticate**, **sec_acl_bind_error**, **sec_acl_invalid_site_name**, **sec_acl_cant_allocate_memory**.

SEE ALSO

Functions: *sec_acl_bind*( ), *sec_acl_release_handle*( ).

NAME

sec_acl_calc_mask — Obtain MASK_OBJ from an ACL.

SYNOPSIS

```
#include <dce/daclif.h>

void sec_acl_calc_mask(
        sec_acl_list_t acl_list,
        error_status_t *status);
```

PARAMETERS

Input/Output

acl_list
An ACL list.

Output

status
Completion status. On successful completion, **error_status_ok** is returned. Otherwise, an error (≠ **error_status_ok**) is returned.

DESCRIPTION

The *sec_acl_calc_mask*( ) routine sets the permission bits of the (**sec_acl_e_type_mask_obj**) entry (creating it first, if necessary) of each of the ACLs in the specified ACL list (*acl_list*), to the ''union'' (bitwise OR) of the permissions of all the ACL entries in the ACL of types USER, FOREIGN_USER, GROUP_OBJ, GROUP, FOREIGN_GROUP, FOREIGN_OTHER and ANY_OTHER (but not USER_OBJ, OTHER_OBJ, UNAUTHENTICATED, EXTENDED or MASK_OBJ, if these exist).

ERRORS

error_status_ok, **sec_acl_cant_allocate_memory**.

SEE ALSO

Functions: *sec_acl_get_manager_types_semantics*( ).

NAME

sec_acl_get_access — Obtain calling client's permissions to a protected object.

SYNOPSIS

```
#include <dce/daclif.h>

void sec_acl_get_access(
        sec_acl_handle_t prot_obj_handle,
        uuid_t *manager_type,
        sec_acl_permset_t *access_rights,
        error_status_t *status);
```

PARAMETERS

Input

prot_obj_handle
Handle to a protected object.

manager_type
An ACL manager type UUID of the protected object.

Output

access_rights
Calling client's access rights to the specified protected object.

status
Completion status. On successful completion, **error_status_ok** is returned. Otherwise, an error (≠ **error_status_ok**) is returned.

DESCRIPTION

The *sec_acl_get_access*( ) routine obtains (a local copy of) the complete set of access rights the calling client has to the specified protected object.

NOTES

Implementations layer this routine over the **rdacl** RPC interface operation *rdacl_get_access*( ).

ERRORS

error_status_ok.

SEE ALSO

Functions: *sec_acl_bind*( ), *sec_acl_bind_to_addr*( ), *sec_acl_get_manager_types*( ), *sec_acl_get_manager_types_semantics*( ), *sec_acl_test_access*( ), *sec_acl_test_access_on_behalf*( ).

Protocols: *rdacl_get_access*( ).

NAME

sec_acl_get_error_info — Obtain fine-grained error information related to **sec_acl** API.

SYNOPSIS

```
#include <dce/daclif.h>

error_status_t sec_acl_get_error_info(
        sec_acl_handle_t prot_obj_handle);
```

PARAMETERS

Input

prot_obj_handle
Handle to a protected object.

RETURN VALUES

The **error_status_t** return value indicates the last error issued by DCE runtime support routines supporting the **sec_acl** API.

DESCRIPTION

The *sec_acl_get_error_info*( ) routine returns DCE runtime routine error information (as opposed to **sec_acl** API error status information) associated with **sec_acl** calls to the (server managing the) indicated protected object.

NOTES

Some implementations of the **sec_acl** API may map certain DCE runtime routine errors (for example, RPC runtime errors) into certain **sec_acl** error status values. This routine recovers those runtime support errors. It is provided for those applications that require the finer-grained information provided by the routine support error values.

The specification of the service provided by this routine is implementation-specific; it is incumbent on implementations of DCE to provide such information.

ERRORS

DCE runtime support routine errors, **sec_acl_invalid_handle**.

SEE ALSO

Functions: *sec_acl_bind*( ), *sec_acl_bind_to_addr*( ).

NAME

sec_acl_get_manager_types — Obtain list of ACL manager types on a protected object.

SYNOPSIS

```
#include <dce/daclif.h>

void sec_acl_get_manager_types(
    sec_acl_handle_t prot_obj_handle,
    sec_acl_type_t acl_type,
    unsigned32 count_max,
    unsigned32 *count,
    unsigned32 *num_manager_types,
    uuid_t manager_types[],
    error_status_t *status);
```

PARAMETERS

Input

prot_obj_handle
Handle to a protected object.

acl_type
An ACL type of the protected object.

count_max
Maximum number of elements the calling client is prepared to receive in its *manager_types*[ ] array.

Output

count
Actual number of elements returned to the calling client in its *manager_types*[ ] array.

num_manager_types
Total number of ACL manager types, of ACL type *acl_type*, at the heads of chains (see *sec_acl_get_printstring( )*), protecting the protected object.

manager_types[ ]
Array of size *count*, of ACL manager types managing ACLs of ACL type *acl_type* on the protected object.

status
Completion status. On successful completion, **error_status_ok** is returned. Otherwise, an error (≠ **error_status_ok**) is returned.

DESCRIPTION

The *sec_acl_get_manager_types*( ) routine returns a list of distinct UUIDs of different ACL manager types managing ACLs of ACL type *acl_type* that are protecting the object identified by *prot_obj_handle*. In the case of a chain of ACL managers (each supporting ≤ 32 permission bits), only the first ACL manager in the chain is returned in this way, and the rest are returned by calls to *sec_acl_get_printstring*( ).

The *sec_acl_get_manager_types*( ) routine also returns, in *num_manager_types*, the total number of ACL manager types, of ACL type *acl_type*, at the heads of chains, protecting the protected object. An invocation of this routine is completely successful only if **count** = *num_manager_types*.

NOTES

Implementations layer this routine over the **rdacl** RPC interface operation *rdacl_get_manager_types*( ).

ERRORS

error_status_ok.

SEE ALSO

Functions: *sec_acl_bind*( ), *sec_acl_bind_to_addr*( ), *sec_acl_get_printstring*( ).

Protocols: *rdacl_get_manager_types*( ).

NAME

sec_acl_get_mgr_types_semantics — Obtain list of ACL manager types on a protected object, together with information about the POSIX semantics they support.

SYNOPSIS

```
#include <dce/daclif.h>

void sec_acl_get_mgr_types_semantics(
        sec_acl_handle_t prot_obj_handle,
        sec_acl_type_t acl_type,
        unsigned32 count_max,
        unsigned32 *count,
        unsigned32 *num_manager_types,
        uuid_t manager_types[],
        sec_acl_posix_semantics_t posix_semantics[],
        error_status_t *status);
```

PARAMETERS

Input

prot_obj_handle
Handle referring to a protected object.

acl_type
An ACL type of the protected object.

count_max
Maximum number of elements the calling client is prepared to receive In its *manager_types*[] and *posix_semantics*[] arrays.

Output

count
Actual number of elements returned to the calling client in its *manager_types*[] and *posix_semantics*[] arrays.

num_manager_types
Total number of ACL manager types, of ACL type *acl_type*, at the heads of chains (see *sec_acl_get_printstring*( )), protecting the protected object.

manager_types[ ]
Array of size *count*, of ACL manager types managing ACLs of ACL type *acl_type* on the protected object.

posix_semantics[ ]
Array of size *count*, indicating the ''POSIX-specific'' semantics supported by the corresponding ACL manager in the *manager_types* array.

status
Completion status. On successful completion, **error_status_ok** is returned. Otherwise, an error (≠ **error_status_ok**) is returned.

DESCRIPTION

The *sec_acl_get_mgr_types_semantics*( ) routine is identical to *sec_acl_get_manager_types*( ), except that it returns the additional *posix_semantics*[] array parameter. That array consists of flag words, each bit of which identifies a ''POSIX-specific'' semantic that the corresponding ACL manager in the *manager_types* array supports (*posix_semantics*[*k*] corresponds to *manager_types*[*k*]). See

Section 10.1.2.7 on page 347 for discussion.

NOTES

Implementations layer this routine over the **rdacl** RPC interface operation *rdacl_get_mgr_types_semantics*( ).

ERRORS

error_status_ok.

SEE ALSO

Functions: *sec_acl_bind*( ), *sec_acl_bind_to_addr*( ), *sec_acl_get_printstring*( ).

Protocols: *rdacl_get_mgr_types_semantics*( ).

NAME

sec_acl_get_printstring — Obtain human-readable representations of permissions supported by an ACL manager.

SYNOPSIS

```
#include <dce/daclif.h>

void sec_acl_get_printstring(
        sec_acl_handle_t prot_obj_handle,
        uuid_t *manager_type,
        unsigned32 count_max,
        uuid_t *manager_type_next,
        sec_acl_printstring_t *manager_info,
        boolean32 *tokenize,
        unsigned32 *num_printstrings,
        unsigned32 *count,
        sec_acl_printstring_t printstrings[],
        error_status_t *status);
```

PARAMETERS

Input

prot_obj_handle
Handle referring to a protected object.

manager_type
An ACL manager type UUID of the protected object.

count_max
Maximum number of elements the calling client is prepared to receive in its *printstrings*[ ] array.

Output

manager_type_next
Identifies the next ACL manager type in a linked list or ‘‘chain’’ of ACL manager types, which can be successively followed until the chain is exhausted (for example, such a chain can be used to support > 32 permission bits). The end of an ACL manager chain is indicated by **uuid_nil**.

manager_info
Name and help information for the ACL manager, as well as a complete set of supported permission bits.

tokenize
Identifies potential ambiguity in the concatenation of permission printstrings (that is, in the **printstring** fields of the elements of the *printstrings*[] array).

num_printstrings
Total number (1, ···, 32) of permission bits and printstrings supported by the ACL manager.

count
Actual number of printstrings returned (in *printstrings*[]).

printstrings[ ]
Array of size *count*, of printstrings representing the permission bits supported by the ACL manager.

status

Completion status. On successful completion, **error_status_ok** is returned. Otherwise, an error (≠ **error_status_ok**) is returned.

DESCRIPTION

The *sec_acl_get_printstring*( ) routine returns information about the ACL manager specified by *manager_type*, managing an ACL of the protected object specified by *prot_obj_handle*. This information is returned in the *printstrings*[] array, which contains one or more entry for each distinct permission the ACL manager supports.

The *sec_acl_get_printstring*( ) routine also returns, in *num_printstrings*, the total number of printstrings supported by the ACL manager. An invocation of this routine is completely successful only if **count** = *num_printstrings*.

In addition to returning information about the permissions themselves, this routine returns instructions in the *tokenize* parameter about concatenating the printstrings associated with them (this is useful for user interfaces to ACL editors). When *tokenize* is 0 (''false''), the permission printstrings may be concatenated without ambiguity (for example, in a user interface to an ACL editor); when non-0 (''true''), this property does not hold and the permission printstrings must be ''tokenised'' (that is, separated by disambiguating characters; for example, non-alphanumeric characters, such as whitespace) to avoid ambiguity when concatenated.

The ACL manager must support at least one *printstring*[] array element pertaining to each permission supported by the ACL manager. If it supports more than one (''aliases'') for a given permission, by convention the simpler entries appear toward the beginning of the *printstring*[] array.

For more information (and an example), see *rdacl_get_printstring*( ) (Section 10.1.10 on page 352).

NOTES

Implementations layer this routine over the **rdacl** RPC interface operation *rdacl_get_printstring*( ).

ERRORS

error_status_ok, **sec_acl_unknown_manager_type**.

SEE ALSO

Functions: *sec_acl_bind*( ), *sec_acl_get_manager_types*( ), *sec_acl_get_manager_types_semantics*( ).

Protocols: *rdacl_get_printstring*( ).

NAME

sec_acl_lookup — Retrieve (''read'') ACLs from a protected object, creating a copy locally on the client.

SYNOPSIS

```
#include <dce/daclif.h>

void sec_acl_lookup(
        sec_acl_handle_t prot_obj_handle,
        uuid_t *manager_type,
        sec_acl_type_t acl_type,
        sec_acl_list_t *acl_list,
        error_status_t *status);
```

PARAMETERS

Input

prot_obj_handle
Handle to a protected object.

manager_type
An ACL manager type UUID of the protected object.

acl_type
An ACL type of the protected object.

Output

acl_list
Copy of retrieved ACL.

status
Completion status. On successful completion, **error_status_ok** is returned. Otherwise, an error (≠ **error_status_ok**) is returned.

DESCRIPTION

The *sec_acl_lookup*( ) routine loads into local memory a copy of the specified protected object's ACLs, managed by the specified ACL manager.

NOTES

The local memory containing the retrieved ACL is dynamically allocated (see *sec_acl_release*( )).

Implementations layer this routine over the **rdacl** RPC interface operation *rdacl_lookup*( ).

ERRORS

error_status_ok, **sec_acl_unknown_manager_type**, **sec_acl_cant_allocate_memory**.

SEE ALSO

Functions: *sec_acl_bind*( ), *sec_acl_bind_to_addr*( ), *sec_acl_get_manager_types*( ), *sec_acl_get_manager_types_semantics*( ), *sec_acl_release*( ), *sec_acl_replace*( ).

Protocols: *rdacl_lookup*( ).

NAME

sec_acl_release — Free (local copy of) ACLs.

SYNOPSIS

```
#include <dce/daclif.h>

void sec_acl_release(
    sec_acl_handle_t prot_obj_handle,
    sec_acl_t *acl,
    error_status_t *status);
```

PARAMETERS

Input

prot_obj_handle
Handle to a protected object.

acl
ACL to be released.

Output

status
Completion status. On successful completion, **error_status_ok** is returned. Otherwise, an error (≠ **error_status_ok**) is returned.

DESCRIPTION

The *sec_acl_release*( ) routine releases local copies of ACLs which had previously been obtained by *sec_acl_lookup*( ).

NOTES

This is a local memory management operation, and has no effect on the protected object to which *prot_obj_handle* is bound, or to its ACLs.

Note: Note that *sec_acl_lookup*( ) returns a *list* of ACLs, while *sec_acl_release*( ) releases ACLs only one at a time. This allows applications to retain only those ACLs of interest to them, without tying up memory for ACLs it isn't interested in.

ERRORS

error_status_ok.

SEE ALSO

Functions: *sec_acl_bind*( ), *sec_acl_bind_to_addr*( ), *sec_acl_lookup*( ).

NAME

sec_acl_release_handle — Release handle to a protected object.

SYNOPSIS

```
#include <dce/daclif.h>

void sec_acl_release_handle(
        sec_acl_handle_t *prot_obj_handle,
        error_status_t *status);
```

PARAMETERS

Input

prot_obj_handle
Handle to a protected object.

Output

status
Completion status. On successful completion, **error_status_ok** is returned. Otherwise, an error (≠ **error_status_ok**) is returned.

DESCRIPTION

The *sec_acl_release_handle*( ) routine releases a handle which had previously been obtained by *sec_acl_bind*( ) or *sec_acl_bind_to_addr*( ).

NOTES

This is a local memory management operation, and has no effect on the protected object to which *prot_obj_handle* is bound, or to its ACLs, or to the server managing them.

ERRORS

error_status_ok.

SEE ALSO

Functions: *sec_acl_bind*( ), *sec_acl_bind_to_addr*( ).

NAME

sec_acl_replace — Apply (''write'') ACLs to a protected object.

SYNOPSIS

```
#include <dce/daclif.h>

void sec_acl_replace(
        sec_acl_handle_t prot_obj_handle,
        uuid_t *manager_type,
        sec_acl_type_t acl_type,
        sec_acl_list_t *acl_list,
        error_status_t *status);
```

PARAMETERS

Input

prot_obj_handle
Handle to a protected object.

manager_type
An ACL manager type UUID to the protected object.

acl_type
An ACL type of the protected object.

acl_list
New ACLs to be applied.

Output

status
Completion status. On successful completion, **error_status_ok** is returned. Otherwise, an error (≠ **error_status_ok**) is returned.

DESCRIPTION

The *sec_acl_replace*( ) routine replaces the ACL managed by the specified ACL manager on the specified protected object, by the new ACL.

NOTES

The *sec_acl_replace*( ) routine replaces the currently existing ACLs on the protected object with the specified new ones.

It is to be noted that the ''currently existing ACLs'' may not be the same as the ''old ACLs'' previously returned by *sec_acl_lookup*( ), because an intervening *sec_acl_replace*( ) may have already replaced the old ACL on the protected object (that is, no locking / transactional semantics are supported to prevent this from happening).

This routine is ''atomic'', in the sense that it deals with whole ACLs at a time, not with individual ACLEs. Also, this routine is uninterruptible by other invocations of itself (because interruptibility could compromise consistency of ACLs).

Implementations layer this routine over the **rdacl** RPC interface operation *rdacl_replace*( ).

ERRORS

error_status_ok, **sec_acl_unknown_manager_type**.

SEE ALSO

Functions: *sec_acl_bind*( ), *sec_acl_bind_to_addr*( ), *sec_acl_get_manager_types*( ), *sec_acl_get_manager_types_semantics*( ), *sec_acl_lookup*( ).

Protocols: *sec_acl_replace*( ).

NAME

sec_acl_test_access — Determine whether calling client has permission to access a protected object.

SYNOPSIS

```
#include <dce/daclif.h>

boolean32 sec_acl_test_access(
    sec_acl_handle_t prot_obj_handle,
    uuid_t *manager_type,
    sec_acl_permset_t access_rights,
    error_status_t *status);
```

PARAMETERS

Input

prot_obj_handle
Handle to a protected object.

manager_type
An ACL manager type UUID of the protected object.

access_rights
Set of access rights to the protected object.

Output

status
Completion status. On successful completion, **error_status_ok** is returned. Otherwise, an error (≠ **error_status_ok**) is returned.

RETURN VALUES

The **boolean32** return value of this routine is valid if and only if the returned *status* value is **error_status_ok**.

This routine returns non-0 (‘‘true’’) if the calling client is granted the specified access rights to the protected object by the specified ACL manager; it returns 0 (‘‘false’’) otherwise.

DESCRIPTION

The *sec_acl_test_access*( ) routine determines whether or not the calling client is granted or denied the specified access rights to the specified protected object by the specified ACL manager.

NOTES

As an example usage, a client could invoke this routine to determine the minimal access rights it needs to accomplish a proposed task, then use that information to acquire (from the DCE PS) a minimal set of credentials authorising it to actually perform the task (this implements a security policy known as ‘‘least privilege’’).

Implementations layer this routine over the **rdacl** RPC interface operation *rdacl_test_access*( ).

ERRORS

error_status_ok, **sec_acl_unknown_manager_type**.

SEE ALSO

Functions: *sec_acl_bind*( ), *sec_acl_bind_to_addr*( ), *sec_acl_get_manager_types*( ), *sec_acl_get_manager_types_semantics*( ), *sec_acl_get_access*( ), *sec_acl_test_access_on_behalf*( ).

Protocols: *rdacl_test_access*( ).

NAME

sec_acl_test_access_on_behalf — Determine whether a specified ‘‘third-party’’ subject (not necessarily the calling client) has permission to access a protected object.

SYNOPSIS

```
#include <dce/daclif.h>

boolean32 sec_acl_test_access_on_behalf(
        sec_acl_handle_t prot_obj_handle,
        uuid_t *manager_type,
        sec_id_pac_t *subject_pac,
        sec_acl_permset_t access_rights,
        error_status_t *status);
```

PARAMETERS

Input

prot_obj_handle
Handle to a protected object.

manager_type
An ACL manager type UUID of the protected object.

subject_pac
Privilege attribute certificate (PAC) of a ‘‘third-party’’ subject.

access_rights
Set of access rights to the protected object.

Output

status
Completion status. On successful completion, **error_status_ok** is returned. Otherwise, an error (≠ **error_status_ok**) is returned.

RETURN VALUES

The **boolean32** return value of this routine is valid if and only if the returned *status* value is **error_status_ok**.

This routine returns non-0 (‘‘true’’) if the specified third-party subject PAC (typically obtained by *rpc_binding_inq_auth_client*( )) grants the specified access rights to the protected object by the specified ACL manager (the calling client must also be granted some degree of ‘‘read-ACL’’ access to determine this — this is dependent on application security policy). It returns 0 (‘‘false’’) otherwise.

DESCRIPTION

The *sec_acl_test_access_on_behalf*( ) routine determines whether or not the specified third-party subject is granted the specified access rights to the specified protected object by the specified ACL manager.

NOTES

A client can combine this routine with *sec_acl_test_access*( ) and use the combined information to implement (a rather primitive form of) *delegation* (schematically characterised as: ‘‘third-party-subject (delegator) → calling-client (delegatee) → server’’).

It is anticipated that a future revision of DCE will support ‘‘true delegation’’, and for that reason *rdacl_test_access_on_behalf*( ) is considered *obsolescent*.

Implementations layer this routine over the **rdacl** RPC interface operation *rdacl_test_access_on_behalf*( ).

ERRORS

error_status_ok, **sec_acl_unknown_manager_type**.

SEE ALSO

Functions: *rpc_binding_inq_auth_client*( ), *sec_acl_bind*( ), *sec_acl_bind_to_addr*( ), *sec_acl_get_manager_types*( ), *sec_acl_get_manager_types_semantics*( ), *sec_acl_get_access*( ), *sec_acl_test_access*( ).

Protocols: *rdacl_test_access_on_behalf*( ).

Chapter 16

Registry API

16.1 Introduction

The routines in the Registry API are distinguished with names having the prefix **sec_rgy**.

Background is given in Chapter 1, especially Section 1.12 on page 60. In particular, the concepts of **RS site**, and of **query** and **update** sites, are defined there.

The routines of the Registry API are:

Binding APIs
Routines used to bind to and communicate with a Registry server have the prefix **sec_rgy_site** or (in one instance) **sec_rgy_cell**.

PGO APIs
Routines used to create and maintain PGO items in the Registry database have the prefix **sec_rgy_pgo**.

Account APIs
Routines used to create and maintain accounts in the Registry database have the prefix **sec_rgy_acct**.

Properties and Policies APIs
Routines used to manipulate cell-wide properties and policies have the prefixes **sec_rgy_auth_plcy**, **sec_rgy_plcy**, and **sec_rgy_properties**.

UNIX Structure APIs
Routines used to obtain Registry entries in a UNIX-style structure have the prefix **sec_rgy_unix**.

Extended Registry Attribute APIs
These routines are used to create and maintain extensions to the DCE Registry database. These include all routines with the prefix of **sec_rgy_attr**, except those with the prefix **sec_rgy_attr_sch** (see the following item).

DCE Attribute APIs
These routines are used to create and maintain data repositories other than the DCE Registry database, and have the prefix **sec_rgy_attr_sch**.

Miscellaneous Registry APIs
The Registry API includes the following miscellaneous routines:

- *sec_rgy_cursor_reset( )*
- *sec_rgy_login_get_info( )*
- *sec_rgy_login_get_effective( )*
- *sec_rgy_wait_until_consistent( )*

NAME

<dce/acct.h> — Header file for the **sec_rgy_acct** API

SYNOPSIS

```
#include <dce/acct.h>
```

DESCRIPTION

Header file for the Registry API used to create and maintain accounts in the Registry database. All of these routines have the prefix **sec_rgy_acct**.

Data Types and Constants

There are no particular data types or constants specific to the **sec_rgy_acct** API (other than those that have already been introduced in this specification).

Status Codes

The following status codes (listed in alphabetical order) are used in the **sec_rgy_acct** API.

error_status_ok
The call was successful.

sec_rgy_no_more_entries
The cursor is at the end of the list of projects.

sec_rgy_not_authorized
Client program is not authorized to add an account to the registry.

sec_rgy_object_not_found
The registry server could not find the specified name.

sec_rgy_not_member_group
The indicated principal is not a member of the indicated group.

sec_rgy_not_member_org
The indicated principal is not a member of the indicated organization.

sec_rgy_not_member_group_org
The indicated principal is not a member of the indicated group or organization.

sec_rgy_object_exists
The account to be added already exists.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

NAME

<dce/binding.h> — Header file for the Registry bind (**sec_rgy_bind**) routines

SYNOPSIS

```
#include <dce/binding.h>
```

DESCRIPTION

Header file for the Registry API used to bind to and communicate with a Registry server. These routines have the prefix **sec_rgy_site** or, in one case, **sec_rgy_cell**.

Data Types and Constants

The following data types (listed in alphabetical order) are used in the **sec_rgy_bind** API.

idl_void_p_t sec_login_handle_t
See **<dce/sec_login.h>** on page 736.

struct sec_rgy_bind_auth_info_t
Represents security context information pertaining to an RS session. Namely, it indicates the subject data, authentication service, authorisation service and protection level associated with protected RPCs to the RS site/server associated with an RS context handle (**sec_rgy_handle_t**). (Conceptually, the RPC binding handle to the RS server is annotated with this security information — see *rpc_binding_set_auth_info( )* in the referenced X/Open DCE RPC Specification.) It contains the following fields:

sec_rgy_bind_auth_info_type_t info_type
The kind of information contained in **dce_info**.

union tagged_union
The actual security context information. It contains the following fields:

/*empty*/
If **info_type** = **sec_rgy_bind_auth_none**, then **tagged_union** is empty.

struct dce_info
If **info_type** = **sec_rgy_bind_auth_dce**, then **tagged_union** holds a **struct dce_info**, which contains the following fields:

unsigned32 authn_level
Protection level.

unsigned32 authn_svc
Authentication service.

unsigned32 authz_svc
Authorisation service.

sec_login_handle_t identity
The login context in effect. If NULL, it refers to the default login context.

enum sec_rgy_bind_auth_info_type_t
Represents the kind of security context pertaining to an RS session. Its currently registered values are the following:

sec_rgy_bind_auth_none = 0
No security.

sec_rgy_bind_auth_dce = 1
Security based on the security services currently supported by DCE.

idl_void_p_t sec_rgy_handle_t
An RS site handle. This is a pointer to a data structure representing a client's *RS* (*session*) *context* (the pointed-to structure is not further specified; that is, **sec_rgy_handle_t** is an opaque pointer).

The RS context contains all the information relevant to the client's session with an RS site (as defined above). In typical implementations, this includes (among other things) an RPC binding handle to an RS server. (Intuitively, this notion of RS site handle or RS session context amounts to an API-level analog of the RPC-level notion of RS server binding.)

Status Codes

The following status codes (listed in alphabetical order) are used in the **sec_rgy_bind** API.

error_status_ok
Routine completed successfully.

sec_login_s_no_current_context
No currently established network identity for which context exists.

sec_rgy_cant_allocate_memory
Can't allocate memory.

sec_rgy_object_not_found
Registry object not found.

sec_rgy_server_unavailable
Server unavailable.

NAME

<dce/misc.h> — Header file for miscellaneous Registry APIs

SYNOPSIS

```
#include <dce/misc.h>
```

DESCRIPTION

Header file for the following miscellaneous Registry routines:

- *sec_rgy_cursor_reset( )*
- *sec_rgy_login_get_info( )*
- *sec_rgy_login_get_effective( )*
- *sec_rgy_wait_until_consistent( )*

Data Types and Constants

There are no particular data types or constants specified to these miscellaneous routines (other than those that have already been introduced in this specification).

Status Codes

The following status codes (listed in alphabetical order) are used in the miscellaneous Registry API.

error_status_ok
The call was successful.

sec_rgy_object_not_found
The specified account could not be found.

sec_rgy_read_only
The Registry is read only; updates are not allowed.

sec_rgy_server_unavailable
The Registry server is unavailable.

NAME

<dce/pgo.h> — Header file for the **sec_rgy_pgo** API

SYNOPSIS

```
#include <dce/pgo.h>
```

DESCRIPTION

Header file for the Registry API used to create and maintain PGO items in the Registry database. All of these routines have the prefix **sec_rgy_pgo**.

Data Types and Constants

There are no particular data types or constants specific to the **sec_rgy_pgo** API (other than those that have already been introduced in this specification).

Status Codes

The following status codes (listed in alphabetical order) are used in the **sec_rgy_pgo** API.

error_status_ok
The call was successful.

sec_rgy_bad_domain
An invalid domain was specified.

sec_rgy_no_more_entries
The cursor is at the end of the list of entries.

sec_rgy_not_authorized
The client is not authorized to add, delete, or modify the specified record.

sec_rgy_object_exists
An object of that name already exists.

sec_rgy_object_not_found
The Registry server could not find the specified name.

sec_rgy_server_unavailable
The Registry server is unavailable.

sec_rgy_unix_id_changed
The UNIX number of the item was changed.

NAME

<dce/policy.h> — Header file for the Registry properties and policies API

SYNOPSIS

```
#include <dce/policy.h>
```

DESCRIPTION

Header file for the Registry API used to manipulate cell-wide properties and policies. These routines have the prefixes **sec_rgy_auth_plcy**, **sec_rgy_plcy**, and **sec_rgy_properties**.

Data Types and Constants

There are no particular data types or constants specific to the security properties and policy API (other than those that have already been introduced in this specification).

Status Codes

The following status codes (listed in alphabetical order) are used in the properties and policies API.

error_status_ok
The call was successful.

sec_rgy_not_authorized
User is not authorized to update record.

sec_rgy_object_not_found
The specified account could not be found.

sec_rgy_server_unavailable
The Registry server is unavailable.

NAME

<dce/rgynbase.h> — Header file for the **sec_rgy_unix** API

SYNOPSIS

```
#include <dce/rgynbase.h>
```

DESCRIPTION

Header file for the Registry API used to obtain Registry entries in a UNIX-style structure. All of these routines have the prefix **sec_rgy_unix**.

Data Types and Constants

There are no particular data types or constants specific to the **sec_rgy_unix** API (other than those that have already been introduced in this specification).

Status Codes

The following status codes (listed in alphabetical order) are used in the **sec_rgy_unix** API.

error_status_ok
The call was successful.

sec_rgy_bad_data
The name supplied as input was too long.

sec_rgy_no_more_entries
The end of the list of entries has been reached.

sec_rgy_server_unavailable
The Registry server is unavailable.

NAME

<dce/sec_rgy_attr.h> — Header file for the **sec_rgy_attr** API

SYNOPSIS

```
#include <dce/sec_rgy_attr.h>
```

DESCRIPTION

Header file for the Registry API used to create and maintain extensions to the DCE Registry database. This API includes all routines with the prefix of **sec_rgy_attr**, except those with the prefix **sec_rgy_attr_sch** (see **<dce/sec_rgy_attr_sch.h>** on page 538).

Data Types and Constants

There are no particular data types or constants specific to the **sec_rgy_attr** API (other than those that have already been introduced in this specification).

Status Codes

The following status codes (listed in alphabetical order) are used in the **sec_rgy_attr** API.

error_status_ok
The call was successful.

sec_attr_bad_encoding_type
Invalid encoding type specified.

sec_attr_bad_type
Invalid or unsupported attribute type.

sec_attr_inst_exists
Attribute instance already exists.

sec_attr_not_unique
Attribute value is not unique.

sec_attr_rgy_obj_not_found
Registry object not found.

sec_attr_svr_read_only
Server is read only.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_trig_svr_unavailable
Trigger server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

NAME

<dce/sec_rgy_attr_sch.h> — Header file for the **sec_rgy_attr_sch** API

SYNOPSIS

```
#include <dce/sec_rgy_attr_sch.h>
```

DESCRIPTION

Header file for the Registry API used to create and maintain data repositories other than the DCE Registry database. This API includes all routines with the prefix of **sec_rgy_attr_sch**.

Data Types and Constants

There are no particular data types or constants specific to the **sec_rgy_attr_sch** API (other than those that have already been introduced in this specification).

Status Codes

The following status codes (listed in alphabetical order) are used in the **sec_rgy_attr_sch** API.

error_status_ok
The call was successful.

sec_attr_bad_acl_mgr_set
Invalid acl_mgr_set specified.

sec_attr_bad_acl_mgr_type
Invalid acl manager type.

sec_attr_bad_bind_authn_svc
Invalid authentication service specified in binding auth_info.

sec_attr_bad_bind_authz_svc
Invalid authorization service specified in binding auth_info.

sec_attr_bad_bind_info
Invalid binding information.

sec_attr_bad_bind_prot_level
Invalid protection level specified in binding auth_info.

sec_attr_bad_bind_svr_name
Invalid server name specified in binding auth_info.

sec_attr_bad_comment
Invalid comment text specified.

sec_attr_bad_cursor
Invalid cursor.

sec_attr_bad_encoding_type
Invalid encoding type specified.

sec_attr_bad_intercell_action
Invalid intercell action specified.

sec_attr_bad_name
Invalid attribute name specified.

sec_attr_bad_permset
Invalid permission set.

sec_attr_bad_scope
Invalid scope specified.

sec_attr_bad_uniq_query_accept
Invalid combination of unique_flag=true, query trigger, and intercell_action=accept.

sec_attr_field_no_update
Field not modifiable.

sec_attr_name_exists
Attribute name already exists.

sec_attr_no_memory
Unable to allocate memory.

sec_attr_sch_entry_not_found
Schema entry not found.

sec_attr_svr_read_only
Server is read only.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_trig_bind_info_missing
Trigger binding info must be specified.

sec_attr_type_id_exists
Attribute type id already exists.

sec_attr_unauthorized
Unauthorized to perform this operation.

NAME

sec_rgy_acct_add — Adds an account for a login name

SYNOPSIS

```
#include <dce/acct.h>

void sec_rgy_acct_add(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *login_name,
       sec_rgy_acct_key_t *key_parts,
       sec_rgy_acct_user_t *user_part,
       sec_rgy_acct_admin_t *admin_part,
       sec_passwd_rec_t *caller_key,
       sec_passwd_rec_t *new_key,
       sec_passwd_type_t new_keytype,
       sec_passwd_version_t *new_key_version,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

login_name
A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. All three names must be completely specified.

key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

user_part
A pointer to the **sec_rgy_acct_user_t** structure containing the user part of the account data. This represents such information as the account password, home directory, and default shell.

admin_part
A pointer to the **sec_rgy_acct_admin_t** structure containing the administrative part of an account's data. This information includes the account creation and expiration dates and flags describing limits to the use of privilege attribute certificates, among other information.

caller_key
A key to use to encrypt *new_key* for transmission to the registry server.

new_key
The password for the new account. During transmission to the registry server, it is encrypted with *caller_key*.

new_keytype
The type of the new key. The server uses this parameter to decide how to encode *new_key* if it is sent as plain text.

Output

new_key_version
The key version number returned by the server. If the client requests a particular key version number (via the **version_number** field of the *new_key* input parameter), the server returns the requested version number back to the client.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_acct_add( )* routine adds an account with the specified login name. The login name is given in three parts, corresponding to the principal, group, and organization names for the account.

The *key_parts* variable specifies the minimum login abbreviation for the account. If the requested abbreviation duplicates an existing abbreviation for another account, the routine supplies the next shortest unique abbreviation and returns this abbreviation in *key_parts*. Abbreviations are not currently implemented.

Permissions Required

The *sec_rgy_acct_add( )* routine requires the following permissions on the account (principal) that is to be added:

- The **m** (**mgmt_info**) permission to change management information.
- The **a** (**auth_info**) permission to change authentication information.
- The **u** (**user_info**) permission to change user information.

NOTES

The constituent principal, group, and organization (PGO) items for an account must be added before the account can be created. (See the *sec_rgy_pgo_add( )* routine). Also, the principal must have been added as a member of the specified group and organization. (See the *sec_rgy_pgo_add_member( )* routine).

FILES

/usr/include/dce/acct.idl The **idl** file from which **dce/acct.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The client program is not authorized to add an account to the registry.

sec_rgy_not_member_group
The indicated principal is not a member of the indicated group.

sec_rgy_not_member_org
The indicated principal is not a member of the indicated organization.

sec_rgy_not_member_group_org
The indicated principal is not a member of the indicated group or organization.

sec_rgy_object exists
The account to be added already exists.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_acct_delete*( ), *sec_rgy_login_get_info*( ), *sec_rgy_pgo_add*( ), *sec_rgy_pgo_add_member*( ), *sec_rgy_site_open*( ).

NAME

sec_rgy_acct_admin_replace — Replaces administrative account data

SYNOPSIS

```
#include <dce/acct.h>

void sec_rgy_acct_admin_replace(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *login_name,
       sec_rgy_acct_key_t *key_parts,
       sec_rgy_acct_admin_t *admin_part,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

login_name

A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. For the group and organization names, blank strings can serve as wildcards, matching any entry. The principal name must be input.

key_parts

A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

admin_part

A pointer to the **sec_rgy_acct_admin_t** structure containing the administrative part of an account's data. This information includes the account creation and expiration dates and flags describing limits to the use of privilege attribute certificates, among other information, and can be modified only by an administrator. The **sec_rgy_acct_admin_t** structure contains the following fields:

creator

The identity of the principal who created this account in **sec_rgy_foreign_id_t** form. This field is set by the registry server.

creation_date

The date (**sec_timeval_sec_t**) the account was created. This field is set by the registry server.

last_changer

The identity of the principal who last modified any of the account information (user or administrative). This field is set by the registry server.

change_date

The date (**sec_timeval_sec_t**) the account was last modified (either user or administrative data). This field is set by the registry server.

expiration_date

The date (**sec_timeval_sec_t**) the account will cease to be valid.

good_since_date
This date (**sec_timeval_sec_t**) is for Kerberos-style, ticket-granting ticket revocation. Ticket-granting tickets issued before this date will not be honored by authenticated network services.

flags
Contains administration flags used as part of the administrator's information for any registry account. This field is in **sec_rgy_acct_admin_flags_t** form.

authentication_flags
Contains flags controlling use of authentication services. This field is in **sec_rgy_acct_auth_flags_t** form.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_acct_admin_replace( )* routine replaces the administrative information in the account record specified by the input login name. The administrative information contains limitations on the account's use and privileges. It can be modified only by a registry administrator; that is, a user with the **auth_info** (abbreviated as **a**) privilege for an account.

The *key_parts* variable identifies how many of the *login_name* parts to use as the unique abbreviation for the account. If the requested abbreviation duplicates an existing abbreviation for another account, the routine supplies the next shortest unique abbreviation and returns this abbreviation using *key_parts*.

Permissions Required

The *sec_rgy_acct_admin_replace( )* routine requires the following permissions on the account principal:

- The **m** (**mgmt_info**) permission, if **flags** or **expiration_date** is to be changed.
- The **a** (**auth_info**) permission, if **authentication_flags** or **good_since_date** is to be changed.

NOTES

All users need the **w** (**write**) privilege in the appropriate ACL entry to modify any account information.

FILES

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The client program is not authorized to change the administrative information for the specified account.

sec_rgy_object_not_found
The registry server could not find the specified name.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO
Functions: *sec_rgy_acct_user_replace*( ), *sec_rgy_acct_replace_all*( ), *sec_rgy_acct_lookup*( ).

NAME

sec_rgy_acct_delete — Deletes an account

SYNOPSIS

```
#include <dce/acct.h>

void sec_rgy_acct_delete(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *login_name,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

login_name
A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. Only the principal name is required to perform the deletion.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_acct_delete( )* routine deletes from the registry the account corresponding to the specified login name.

Permissions Required

The *sec_rgy_acct_delete( )* routine requires the following permissions on the account principal:

- The **m** (**mgmt_info**) permission to remove management information.
- The **a** (**auth_info**) permission to remove authentication information.
- The **u** (**user_info**) permission to remove user information.

NOTES

Even though the account is deleted, the PGO items corresponding to the account remain. These must be deleted with separate calls to *sec_rgy_pgo_delete( )*.

FILES

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The client program is not authorized to delete the specified account.

sec_rgy_object_not_found
No PGO item was found with the given name.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO
Functions: *sec_rgy_acct_add*( ), *sec_rgy_pgo_delete*( ).

NAME

sec_rgy_acct_get_projlist — Returns the projects in an account's project list

SYNOPSIS

```
#include <dce/acct.h>

void sec_rgy_acct_get_projlist(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *login_name,
       sec_rgy_cursor_t *projlist_cursor,
       signed32 max_number,
       signed32 *supplied_number,
       uuid_t id_projlist[],
       signed32 unix_projlist[],
       signed32 *num_projects,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

login_name
A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. For the group and organization names, blank strings can serve as wildcards, matching any entry. The principal name must be input.

max_number
The maximum number of projects to be returned by the call. This must be no larger than the allocated size of the *projlist*[ ] arrays.

Input/Output

projlist_cursor
An opaque pointer indicating a specific project in an account's project list. The *sec_rgy_acct_get_projlist( )* routine returns the project indicated by *projlist_cursor*, and advances the cursor to point to the next project in the list. When the end of the list is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use *sec_rgy_cursor_reset( )* to reset the cursor.

Output

supplied_number
A pointer to the actual number of projects returned. This will always be less than or equal to the *max_number* supplied on input. If there are more projects in the account list, *sec_rgy_acct_get_projlist( )* sets *projlist_cursor* to point to the next entry after the last one in the returned list.

id_projlist[ ]
An array to receive the UUID of each project returned. The size allocated for the array is given by *max_number*. If this value is less than the total number of projects in the account project list, multiple calls must be made to return all of the projects.

unix_projlist[ ]
An array to receive the UNIX number of each project returned. The size allocated for the array is given by *max_number*. If this value is less than the total number of projects in the account project list, multiple calls must be made to return all of the projects.

num_projects
A pointer indicating the total number of projects in the specified account's project list.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_acct_get_projlist( )* routine returns members of the project list for the specified account. It returns the project information in two arrays. The *id_projlist*[ ] array contains the UUIDs for the returned projects. The *unix_projlist*[ ] array contains the UNIX numbers for the returned projects.

The project list cursor, *projlist_cursor*, provides an automatic place holder in the project list. The *sec_rgy_acct_get_projlist( )* routine automatically updates this variable to point to the next project in the project list. To return an entire project list, reset *projlist_cursor* with *sec_rgy_cursor_reset( )* on the initial call and then issue successive calls until all the projects are returned.

Permissions Required

The *sec_rgy_acct_get_projlist( )* routine requires the **r** (**read**) permission on the account principal for which the project list data is to be returned.

CAUTIONS

There are several different types of cursors used in the registry API. Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to *sec_rgy_acct_get_projlist( )* and *sec_rgy_pgo_get_next( )*. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use *sec_rgy_cursor_reset( )* to refresh a cursor for use with another call or for another server.

FILES

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_no_more_entries
The cursor is at the end of the list of projects.

sec_rgy_not_authorized
The client program is not authorized to see a project list for this principal.

sec_rgy_object exists
The account to be added already exists.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO
Functions: *sec_rgy_cursor_reset*( ), *sec_rgy_pgo_get_next*( ).

NAME

sec_rgy_acct_lookup — Returns data for a specified account

SYNOPSIS

```
#include <dce/acct.h>

void sec_rgy_acct_lookup(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *name_key,
       sec_rgy_cursor_t *account_cursor,
       sec_rgy_login_name_t *name_result,
       sec_rgy_sid_t *id_sid,
       sec_rgy_unix_sid_t *unix_sid,
       sec_rgy_acct_key_t *key_parts,
       sec_rgy_acct_user_t *user_part,
       sec_rgy_acct_admin_t *admin_part,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_key

A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. Blank strings serve as wildcards, matching any entry.

Input/Output

account_cursor

An opaque pointer to a specific account in the registry database. If *name_key* is blank, *sec_rgy_acct_lookup( )* returns information about the account to which the cursor is pointing. On return, the cursor points to the next account in the database after the returned account. If *name_key* is blank and the *account_cursor* has been reset with *sec_rgy_cursor_reset( )*, *sec_rgy_acct_lookup( )* returns information about the first account in the database. When the end of the list of accounts in the database is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use *sec_rgy_cursor_reset( )* to refresh the cursor.

Output

name_result

A pointer to the full login name of the account (including all three names) for which the information is returned. The remaining parameters contain the information belonging to the returned account.

id_sid

A structure containing the three UUIDs of the principal, group, and organization for the account.

unix_sid
A structure containing the three UNIX numbers of the principal, group, and organization for the account.

key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

user_part
A pointer to the **sec_rgy_acct_user_t** structure containing the user part of the account data. This represents such information as the account password, home directory, and default shell, all of which are accessible to, and may be modified by, the account owner.

admin_part
A pointer to the **sec_rgy_acct_admin_t** structure containing the administrative part of an account's data. This information includes the account creation and expiration dates and flags describing limits to the use of privilege attribute certificates, among other information, and can be modified only by an administrator.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_acct_lookup( )* routine returns all the information about an account in the registry database. The account can be specified either with *name_key* or *account_cursor*. If *name_key* is completely blank, the routine uses the *account_cursor* value instead.

For *name_key*, a zero-length principal, group, or organization key serves as a wildcard. For example, a login name key with the principal and organization fields blank returns the next (possibly first) account whose group matches the input group field. The full login name of the returned account is passed back in *name_result*.

The *account_cursor* provides an automatic place holder in the registry database. The routine automatically updates this variable to point to the next account in the database, after the account for which the information was returned. If *name_key* is blank and the *account_cursor* has been reset with *sec_rgy_cursor_reset( )*, *sec_rgy_acct_lookup( )* returns information about the first account in the database.

Permissions Required

The *sec_rgy_acct_lookup( )* routine requires the **r** (**read**) permission on the account principal to be viewed.

CAUTIONS

There are several different types of cursors used in the registry API. Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to *sec_rgy_acct_get_projlist( )* and *sec_rgy_pgo_get_next( )*. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use *sec_rgy_cursor_reset( )* to renew a cursor for use with another call or for another server.

FILES

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_no_more_entries
The cursor is at the end of the accounts in the registry.

sec_rgy_object_not_found
The input account could not be found by the registry server.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_cursor_reset*( ), *sec_rgy_acct_replace_all*( ), *sec_rgy_acct_admin_replace*( ), *sec_rgy_acct_user_replace*( ).

NAME

sec_rgy_acct_passwd — Changes the password for an account

SYNOPSIS

```
#include <dce/acct.h>

void sec_rgy_acct_passwd(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *login_name,
       sec_passwd_rec_t *caller_key,
       sec_passwd_rec_t *new_key,
       sec_passwd_type_t new_keytype,
       sec_passwd_version_t *new_key_version,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

login_name
A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. All three strings must be completely specified.

caller_key
A key to use to encrypt the key for transmission to the registry server. If communications secure to the **rpc_c_authn_level_pkt_privacy** level are available on a system, then this parameter is not necessary, and the packet encryption is sufficient to ensure security.

new_key
The password for the new account. During transmission to the registry server, it is encrypted with *caller_key*.

new_keytype
The type of the new key. The server uses this parameter to decide how to encode *new_key* if it is sent as plain text.

Output

new_key_version
The key version number returned by the server. If the client requests a particular key version number (via the **version_number** field of the *new_key* input parameter), the server returns the requested version number back to the client.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_acct_passwd( )* routine changes an account password to the input password character string. Wildcards (blank fields) are not permitted in the specified account name; the principal, group, and organization names of the account must be completely specified.

Permissions Required

The *sec_rgy_acct_passwd( )* routine requires the **u** (**user_info**) permission on the account principal whose password is to be changed.

FILES

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The client program is not authorized to change the password of this account.

sec_rgy_object_not_found
The account to be modified was not found by the registry server.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

NAME

sec_rgy_acct_rename — Changes an account login name

SYNOPSIS

```
#include <dce/acct.h>

void sec_rgy_acct_rename(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *old_login_name,
       sec_rgy_login_name_t *new_login_name,
       sec_rgy_acct_key_t *new_key_parts,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

old_login_name
A pointer to the current account login name. The login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. All three strings must be completely specified.

new_login_name
A pointer to the new account login name. Again, all three component names must be completely specified.

Input/Output

new_key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_acct_rename( )* routine changes an account login name from *old_login_name* to *new_login_name*. Wildcards (empty fields) are not permitted in either input name; both the old and new login names must completely specify their component principal, group, and organization names. Note, though, that the principal component in a login name cannot be changed.

The *new_key_parts* variable identifies how many of the *new_login_name* parts to use as the unique abbreviation for the account. If the requested abbreviation duplicates an existing abbreviation for another account, the routine identifies the next shortest unique abbreviation and returns this abbreviation using *new_key_parts*.

Permissions Required

The *sec_rgy_acct_rename( )* routine requires the **m** (**mgmt_info**) permission on the account principal to be renamed.

NOTES

The *sec_rgy_acct_rename( )* routine does not affect any of the registry PGO data. The constituent principal, group, and organization items for an account must be added before the account can be created. (See the *sec_rgy_pgo_add( )* routine). Also, the principal must have been added as a member of the specified group and organization. (See the *sec_rgy_pgo_add_member( )* routine).

FILES

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The client program is not authorized to make the changes.

sec_rgy_object_not_found
The account to be modified was not found by the registry server.

sec_rgy_name_exists
The new account name is already in use by another account.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_acct_add*( ).

NAME

sec_rgy_acct_replace_all — Replaces all account data for an account

SYNOPSIS

```
#include <dce/acct.h>

void sec_rgy_acct_replace_all(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *login_name,
       sec_rgy_acct_key_t *key_parts,
       sec_rgy_acct_user_t *user_part,
       sec_rgy_acct_admin_t *admin_part,
       boolean32 set_password,
       sec_passwd_rec_t *caller_key,
       sec_passwd_rec_t *new_key,
       sec_passwd_type_t new_keytype,
       sec_passwd_version_t *new_key_version,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

login_name
A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. For the group and organization names, blank strings can serve as wildcards, matching any entry. The principal name must be input.

user_part
A pointer to the **sec_rgy_acct_user_t** structure containing the user part of the account data. This represents such information as the account password, home directory, and default shell, all of which are accessible to, and may be modified by, the account owner.

admin_part
A pointer to the **sec_rgy_acct_admin_t** structure containing the administrative part of an account's data. This information includes the account creation and expiration dates and flags describing limits to the use of privilege attribute certificates, among other information, and can be modified only by an administrator.

set_passwd
The password reset flag. If you set this parameter to TRUE, the account's password will be changed to the value specified in *new_key*.

caller_key
A key to use to encrypt the key for transmission to the registry server. If communications secure to the **rpc_c_authn_level_pkt_privacy** level are available on a system, then this parameter is not necessary, and the packet encryption is sufficient to ensure security.

new_key
The password for the new account. During transmission to the registry server, it is encrypted with *caller_key*.

new_keytype
The type of the new key. The server uses this parameter to decide how to encode the plaintext key.

Input/Output

key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

Output

new_key_version
The key version number returned by the server. If the client requests a particular key version number (via the **version_number** field of the *new_key* input parameter), the server returns the requested version number back to the client.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_acct_replace_all( )* routine replaces both the user and administrative information in the account record specified by the input login name. The administrative information contains limitations on the account's use and privileges. The user information contains such information as the account home directory and default shell. Typically, the administrative information can only be modified by a registry administrator (users with **auth_info** (**a**) privileges for an account), while the user information can be modified by the account owner (users with **user_info** (**u**) privileges for an account).

Use the *set_passwd* parameter to reset the account password. If you set this parameter to TRUE, the account's password is changed to the value specified in *new_key*.

The *key_parts* variable identifies how many of the *login_name* parts to use as the unique abbreviation for the replaced account. If the requested abbreviation duplicates an existing abbreviation for another account, the routine identifies the next shortest unique abbreviation and returns this abbreviation using *key_parts*.

Permissions Required

The *sec_rgy_acct_replace_all( )* routine requires the following permissions on the account principal:

- The **m** (**mgmt_info**) permission, if **flags** or **expiration_date** is to be changed.
- The **a** (**auth_info**) permission, if **authentication_flags** or **good_since_date** is to be changed.
- The **u** (**user_info**) permission, if user **flags**, **gecos**, **homedir** (home directory), **shell**, or **passwd** (password) are to be changed.

NOTES

All users need the **w** (**write**) privilege to modify any account information.

FILES

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The client program is not authorized to change account information.

sec_rgy_object_not_found
The specified account could not be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_acct_add*( ), *sec_rgy_acct_admin_replace*( ), *sec_rgy_acct_rename*( ), *sec_rgy_acct_user_replace*( ).

NAME

sec_rgy_acct_user_replace — Replaces user account data

SYNOPSIS

```
#include <dce/acct.h>

void sec_rgy_acct_user_replace(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *login_name,
       sec_rgy_acct_user_t *user_part,
       boolean32 set_passwd,
       sec_passwd_rec_t *caller_key,
       sec_passwd_rec_t *new_key,
       sec_passwd_type_t new_keytype,
       sec_passwd_version_t *new_key_version,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

login_name

A pointer to the account login name. A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. For the group and organization names, blank strings can serve as wildcards, matching any entry. The principal name must be input.

user_part

A pointer to the **sec_rgy_acct_user_t** structure containing the user part of the account data. This represents such information as the account password, home directory, and default shell, all of which are accessible to, and may be modified by, the account owner. The structure contains the following fields:

gecos

A character string containing information about the account owner. This often includes such information as their name and telephone number.

homedir

The default directory upon login for the account.

shell

The default shell to use upon login.

passwd_version_number

The password version number, a 32-bit unsigned integer, set by the registry server.

passwd_dtm

The date and time of the last password change (in **sec_timeval_sec_t** form), also set by the registry server.

flags

A flag set of type **sec_rgy_acct_user_flags_t**.

passwd
The account's encrypted password.

set_passwd
The password reset flag. If you set this parameter to TRUE, the user's password will be changed to the value specified in *new_key*.

caller_key
A key to use to encrypt the key for transmission to the registry server. If communications secure to the **rpc_c_authn_level_pkt_privacy** level are available on a system, then this parameter is not necessary, and the packet encryption is sufficient to ensure security.

new_key
The password for the new account. During transmission to the registry server, it is encrypted with *caller_key*.

new_keytype
The type of the new key. The server uses this parameter to decide how to encode the plaintext key.

Output

new_key_version
The key version number returned by the server. If the client requests a particular key version number (via the **version_number** field of the *new_key* input parameter), the server returns the requested version number back to the client.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_acct_user_replace( )* routine replaces the user information in the account record specified by the input login name. The user information contains such information as the account home directory and default shell. Typically, the the user information can be modified by the account owner (users with **user_info** (**u**) privileges for an account).

Use the *set_passwd* parameter to reset the user's password. If you set this parameter to TRUE, the user's password is changed to the value specified in *new_key*.

Permissions Required

The *sec_rgy_acct_user_replace( )* routine requires the **u** (**user_info**) permission on the account principal.

NOTES

All users need the **w** (**write**) privilege to modify any account information.

FILES

/usr/include/dce/acct.idl
The **idl** file from which **dce/acct.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The client program is not authorized to modify the account data.

sec_rgy_object_not_found
The specified account could not be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_acct_add*( ), *sec_rgy_acct_admin_replace*( ), *sec_rgy_acct_rename*( ), *sec_rgy_acct_replace_all*( ).

NAME

sec_rgy_attr_cursor_alloc — Allocates resources to a cursor used by the *sec_rgy_attr_lookup_by_id( )* call

SYNOPSIS

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_cursor_alloc(
       sec_attr_cursor_t *cursor,
       error_status_t *status);
```

PARAMETERS

Output

cursor
A pointer to a **sec_attr_cursor_t**.

status
A pointer to the completion status. On successful completion, the call returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_cursor_alloc( )* call allocates resources to a cursor used with the *sec_rgy_attr_lookup_by_id( )* call. This routine, which is a local operation, does not initialize *cursor*.

The *sec_rgy_attr_cursor_init( )* routine, which makes a remote call, allocates and initializes the cursor. In addition, *sec_rgy_attr_cursor_init( )* returns the total number of attributes attached to the object as an output parameter; *sec_rgy_attr_cursor_alloc( )* does not.

Permissions Required

None

FILES

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_object_not_found
Registry object not found.

SEE ALSO

Functions: *sec_rgy_attr_cursor_init*( ), *sec_rgy_attr_cursor_release*( ), *sec_rgy_attr_cursor_reset*( ), *sec_rgy_attr_lookup_by_id*( ).

NAME

sec_rgy_attr_cursor_init — Initialize a cursor used by the *sec_rgy_attr_lookup_by_id( )* call

SYNOPSIS

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_cursor_init (
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t name,
       unsigned32 *cur_num_attrs,
       sec_attr_cursor_t *cursor,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

A value of type **sec_rgy_domain_t** that identifies the registry domain in which the object specified by *name* resides. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name

A pointer to a **sec_rgy_name_t** character string containing the name of the person, group, or organization to which the attribute to be scanned is attached.

Output

cur_num_attrs

A pointer to an unsigned 32-bit integer that specifies the number of attributes currently attached to the object.

cursor

A pointer to a **sec_rgy_cursor_t** positioned at the first attribute in the list of the object's attributes.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_cursor_init( )* routine initializes a cursor of type **sec_attr_cursor_t** (used with the *sec_rgy_attr_lookup_by_id( )* call) and initializes the cursor to the first attribute in the specified object's list of attributes. This call also supplies the total number of attributes attached to the

object as part of its output. The cursor allocation is a local operation. The cursor initialization is a remote operation and makes a remote call to the Registry.

Use the *sec_rgy_attr_cursor_release( )* call to release all resources allocated to a **sec_attr_cursor_t** cursor.

Permissions Required

The *sec_rgy_attr_cursor_init( )* routine requires at least one permission (of any type) on the person, group, or organization to which the attribute to be scanned is attached.

FILES

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_object_not_found
Registry object not found.

SEE ALSO

Functions: *sec_rgy_attr_lookup_by_id*( ), *sec_rgy_attr_cursor_release*( ).

NAME

sec_rgy_attr_cursor_release — Release a cursor of type **sec_attr_cursor_t** that was allocated with either the *sec_rgy_attr_cursor_init( )* or *sec_rgy_attr_cursor_alloc( )* call

SYNOPSIS

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_cursor_release (
       sec_attr_cursor_t *cursor,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

Input/Output

cursor
As an input parameter, a pointer to an uninitialized cursor of type **sec_attr_cursor_t**. As an output parameter, a pointer to an uninitialized cursor of type **sec_attr_cursor_t** with all resources released.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_cursor_release( )* routine releases all resources allocated to a **sec_attr_cursor_t** by the *sec_rgy_attr_cursor_init( )* or *sec_rgy_attr_cursor_alloc( )* call.

This is a local-only operation and makes no remote calls.

Permissions Required

None.

FILES

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_object_not_found
Registry object not found.

SEE ALSO

Functions: *sec_rgy_attr_cursor_init*( ), *sec_rgy_attr_cursor_alloc*( ), *sec_rgy_attr_lookup_by_id*( ).

NAME

sec_rgy_attr_cursor_reset — Reinitializes a cursor that has been allocated with either *sec_rgy_attr_cursor_init( )* or *sec_rgy_attr_cursor_alloc( )*

SYNOPSIS

```
#include <dce/sec_rgy_attr.h>

void sec_attr_cursor_reset(
       sec_attr_cursor_t *cursor,
       error_status_t *status);
```

PARAMETERS

Input/Output

cursor

A pointer to a **sec_attr_cursor_t**. As an input parameter, an initialized *cursor*. As an output parameter, *cursor* is reset to the first attribute in the schema.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_cursor_reset( )* routine resets a **dce_attr_cursor_t** that has been allocated by either a *sec_rgy_attr_cursor_init( )* or *sec_rgy_attr_cursor_alloc( )* call. The reset cursor can then be used to process a new *sec_rgy_attr_lookup_by_id( )* query by reusing the cursor instead of releasing and re-allocating it. This is a local operation and makes no remote calls.

Permissions Required

None.

FILES

/usr/include/dce/sec_rgy_attr.idl

The **idl** file from which **dce/sec_rgy_attr.h** was derived.

ERRORS

error_status_ok

The call was successful.

SEE ALSO

Functions: *sec_rgy_attr_cursor_init*( ), *sec_rgy_attr_cursor_alloc*( ), *sec_rgy_attr_lookup_by_id*( ).

NAME

sec_rgy_attr_delete — Deletes specified attributes for a specified object

SYNOPSIS

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_delete (
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t name,
       unsigned32 num_to_delete,
       sec_attr_t attrs[ ],
       signed32 *failure_index,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

A value of type **sec_rgy_domain_t** that identifies the registry domain in which the object identified by *name* resides. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name

A character string of type **sec_rgy_name_t** specifying the name of the person, group, or organization to which the attributes are attached.

num_to_delete

A 32-bit integer that specifies the number of elements in the *attrs* array. This integer must be greater than 0.

attrs[ ]

An array of values of type **sec_attr_t** that specifies the attribute instances to be deleted. The size of the array is determined by *num_to_delete*.

Output

failure_index
In the event of an error, *failure_index* is a pointer to the element in the *attrs* array that caused the update to fail. If the failure cannot be attributed to a specific attribute, the value of *failure_index* is **-1**.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_delete( )* routine deletes attributes. This is an atomic operation: if the deletion of any attribute in the *attrs* array fails, all deletions are aborted. The attribute causing the delete to fail is identified in *failure_index*. If the failure cannot be attributed to a given attribute, *failure_index* contains **-1**.

The *attrs* array, which specifies the attributes to be deleted, contains values of type **sec_attr_t**. These values consist of:

- **attr_id**, a UUID that identifies the attribute type
- **attr_value**, values of **sec_attr_value_t** that specify the attribute's encoding type and values.

To delete attributes that are not multi-valued and to delete all instances of a multi-valued attribute, an attribute UUID is all that is required. For these attribute instances, supply the attribute UUID in the input array and set the attribute encoding (in **sec_attr_encoding_t**) to **sec_attr_enc_void**.

To delete a specific instance of a multi-valued attribute, supply the UUID and value that uniquely identify the multi-valued attribute instance in the input array.

Note that if the deletion of any attribute instance in the array fails, all fail. However, to help pinpoint the cause of the failure, the call identifies the first attribute whose deletion failed in a failure index by array element number.

Permissions Required

The *sec_rgy_attr_delete( )* routine requires the delete permission set for each attribute type identified in the *attrs* array. These permissions are defined as part of the ACL manager set in the schema entry for the attribute type.

FILES

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_bad_type
Invalid or unsupported attribute type.

sec_attr_svr_read_only
Server is read only.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO
Functions: *sec_rgy_attr_update*( ).

NAME

sec_rgy_attr_get_effective — Reads effective attributes by ID

SYNOPSIS

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_get_effective(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t name,
       unsigned32 num_attr_keys,
       sec_attr_t attr_keys[],
       sec_attr_vec_t *attr_list,
       error_status_t status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

A value of type **sec_rgy_domain_t** that identifies the domain in which the named object resides. The valid values are as follows:

sec_rgy_domain_person
The *name* identifies a principal.

sec_rgy_domain_group
The *name* identifies a group.

sec_rgy_domain_org
The *name* identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name

A pointer to a **sec_rgy_name_t** character string containing the name of the person, group, or organization to which the attribute is attached.

num_attr_keys

An unsigned 32-bit integer that specifies the number of elements in the the *attr_keys* array. If *num_attr_keys* is set to 0, all of the effective attributes that the caller is authorized to see are returned.

attr_keys[ ]

An array of values of type **sec_attr_t** that specify the UUIDs of the attributes to be returned if they are effective. If the attribute type is associated with a query attribute trigger, the **sec_attr_t attr_value** field can be used to pass in optional information required by the attribute trigger query. If no information is to be passed in the **attr_value** field (whether the type indicates an attribute trigger query or not), set the attribute's encoding type to **sec_rgy_attr_enc_void**. The size of the *attr_keys* array is determined by the *num_attr_keys* parameter.

Output

attr_list

A pointer an attribute vector allocated by the server containing all of the effective attributes matching the search criteria (defined in *num_attr_keys* or *attr_keys*). The server allocates a buffer large enough to return all the requested attributes so that subsequent calls are not necessary.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_get_effective( )* routine returns the UUIDs of a specified object's effective attributes. Effective attributes are determined by setting of the schema entry **sec_attr_sch_entry_use_defaults** flag:

- If the flag is set off, only the attributes directly attached to the object are effective.
- If the flag is set on, the effective attributes are obtained by performing the following steps for each attribute identified by UUID in the *attr_keys* array:

  1. If the object named by *name* is a principal and if the a requested attribute exists on the principal, that attribute is effective and is returned. If it does not exist, the search continues.
  2. The next step in the search depends on the type of object:

     For principals with accounts:

     a. The organization named in the principal's account is examined to see if an attribute of the requested type exists. If it does, it is effective and is returned; then the search for that attribute ends. If it does not exist, the search for that attribute continues to the **policy** object as described in b, below.

     b. The registry **policy** object is examined to see if an attribute of the requested type exits. If it does, it is returned. If it does not, a message indicating the no attribute of the type exists for the object is returned.

     For principals without accounts, for groups, and for organizations:

     The registry **policy** object is examined to see if an attribute of the requested type exits. If it does, it is returned. If it does not, a message indicating the no attribute of the type exists for the object is returned.

For multi-valued attributes, the call returns a **sec_attr_t** for each value as an individual attribute instance. For attribute sets, the call returns a **sec_attr_t** for each member of the set; it does not return the set instance.

If the attribute instance to be read is associated with a query attribute trigger that requires additional information before it can process the query request, use a **sec_attr_value_t** to supply the requested information. To do this:

- Set the **sec_attr_encoding_t** to an encoding type that is compatible with the information required by the query attribute trigger.
- Set the **sec_attr_value_t** to hold the required information.

If the attribute instance to be read is not associated with a query trigger or no additional information is required by the query trigger, an attribute UUID is all that is required. For these attribute instances, supply the attribute UUID in the input array and set the attribute encoding (in **sec_attr_encoding_t**) to **sec_attr_enc_void**.

If the requested attribute type is associated with a query trigger, the value returned for the attribute will be the binding (as set in the schema entry) of the trigger server. The caller must bind to the trigger server and pass the original input query attribute to the *sec_attr_trig_query( )* call in order to retrieve the attribute value.

FILES

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

ERRORS

error_status_ok
The call was successful.

NAME

sec_rgy_attr_lookup_by_id — Reads a specified object's attribute(s), expanding attribute sets into individual member attributes

SYNOPSIS

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_lookup_by_id (
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t name,
       sec_attr_cursor_t *cursor,
       unsigned32 num_attr_keys,
       unsigned32 space_avail,
       sec_attr_t attr_keys[ ],
       unsigned32 *num_returned,
       sec_attr_t attrs[ ],
       unsigned32 *num_left,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

A value of type **sec_rgy_domain_t** that identifies the registry domain in which the object specified by *name* resides. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name

A pointer to a **sec_rgy_name_t** character string containing the name of the person, group, or organization to which the attribute is attached.

num_attr_keys

An unsigned 32-bit integer that specifies the number of elements in the *attr_keys* array. Set this parameter to 0 to return all of the object's attributes that the caller is authorized to see.

space_avail

An unsigned 32-bit integer that specifies the size of the *attr_keys* array.

attr_keys[ ]

An array of values of type **sec_attr_t** that identify the attribute type ID of the attribute instance(s) to be looked up. If the attribute type is associated with a query attribute trigger, the **sec_attr_t attr_value** field can be used to pass in optional information required by the

attribute trigger query. If no information is to be passed in the **attr_value** field (whether the type indicates an attribute trigger query or not), set the attribute's encoding type to **sec_rgy_attr_enc_void**.

The size of the *attr_keys* array is determined by the *num_attr_keys* parameter.

Input/Output

cursor
A pointer to a **sec_attr_cursor_t**. As an input parameter, *cursor* is a pointer to a **sec_attr_cursor_t** initialized by a *sec_rgy_attr_srch_cursor_init( )* call. As an output parameter, *cursor* is a pointer to a **sec_attr_cursor_t** that is positioned past components returned in this call.

Output

num_returned
A pointer to a 32-bit unsigned integer that specifies the number of attribute instances returned in the *attrs* array.

attrs[ ]
An array of values of type **sec_attr_t** that contains the attributes retrieved by UUID. The size of the array is determined by *space_avail* and the length by *num_returned*.

num_left
A pointer to a 32-bit unsigned integer that supplies the number of attributes that were found but could not be returned because of space constraints in the *attrs* buffer. To ensure that all the attributes will be returned, increase the size of the *attrs* array by increasing the size of *space_avail* and *num_returned*.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**, or, if the requested attributes were not available, it returns the message **not_all_available**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_lookup_by_id( )* function reads those attributes specified by UUID for an object specified by name. This routine is similar to the *sec_rgy_attr_lookup_no_expand( )* routine with one exception: for attribute sets, the *sec_rgy_attr_lookup_no_expand( )* routine returns a **sec_attr_t** for the set instance only; it does not expand the set and return a **sec_attr_t** for each member in the set. This call expands attribute sets and returns a **sec_attr_t** for each member in the set.

If the *num_attr_keys* parameter is set to 0, all of the object's attributes that the caller is authorized to see are returned. This routine is useful for programmatic access.

For multi-valued attributes, the call returns a **sec_attr_t** for each value as an individual attribute instance. For attribute sets, the call returns a **sec_attr_t** for each member of the set; it does not return the set instance.

The *attr_keys* array, which specifies the attributes to be returned, contains values of type **sec_attr_t**. These values consist of:

- **attr_id**, a UUID that identifies the attribute type
- **attr_value**, values of **sec_attr_value_t** that specify the attribute's encoding type and values.

Use the **attr_id** field of each *attr_keys* array element, to specify the UUID that identifies the attribute type to be returned.

If the attribute instance to be read is not associated with a query trigger or no additional information is required by the query trigger, an attribute UUID is all that is required. For these attribute instances, supply the attribute UUID in the input array and set the attribute encoding (in **sec_attr_encoding_t**) to **sec_attr_enc_void**.

If the attribute instance to be read is associated with a query attribute trigger that requires additional information before it can process the query request, use a **sec_attr_value_t** to supply the requested information. To do this:

- Set the **sec_attr_encoding_t** to an encoding type that is compatible with the information required by the query attribute trigger.
- Set the **sec_attr_value_t** to hold the required information.

Note that if you set *num_attr_keys* to zero to return all of the object's attributes and that attribute is associated with a query attribute trigger, the attribute trigger will be called with no input attribute information (that would normally have been passed in via the **attr_value** field).

The *cursor* parameter specifies a cursor of type **sec_attr_cursor_t** initialized to the point in the attribute list at which to start processing the query. Use the *sec_attr_cursor_init( )* function to initialize *cursor*. If *cursor* is uninitialized, the server begins processing the query at the first attribute that satisfies the search criteria.

The *num_left* parameter contains the number of attributes that were found but could not be returned because of space constraints of the *attrs* array. (Note that this number may be inaccurate if the target server allows updates between successive queries.) To obtain all of the remaining attributes, set the size of the *attrs* array so that it is large enough to hold the number of attributes listed in *num_left*.

Permissions Required

The *sec_rgy_attr_lookup_by_id( )* routine requires the query permission set for each attribute type identified in the *attr_keys* array. These permissions are defined as part of the ACL manager set in the schema entry of each attribute type.

FILES

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_trig_svr_unavailable
Trigger server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_lookup_no_expand* ( ), *sec_rgy_attr_attr_lookup_by_name* ( ).

NAME

sec_rgy_attr_lookup_by_name — Read a single attribute instance for a specific object

SYNOPSIS

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_lookup_by_name(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t name,
       char *attr_name,
       sec_attr_t *attr,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain
A value of type **sec_rgy_domain_t** that identifies the domain in which the named object resides. The valid values are as follows:

sec_rgy_domain_person
The *name* identifies a principal.

sec_rgy_domain_group
The *name* identifies a group.

sec_rgy_domain_org
The *name* identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name
A pointer to a **sec_rgy_name_t** character string containing the name of the person, group, or organization to which the attribute is attached.

attr_name
An pointer to a character string that specifies the name of the attribute to be retrieved.

Output

attr
A pointer to a **sec_attr_t** that contains the first instance of the named attribute.

status
A pointer to the completion status. The completion status can be one of the following:

- **error_status_ok** if all instances of the value are returned with no errors.
- **more_available** if a multi-valued attribute was specified as *name* and the routine completed successfully. For multi-valued attributes, this routine returns the first instance of the attribute.
- **attribute_set_instance** if an attribute set was specified as *name* and the routine completed successfully.

- An error message if the routine did not complete successfully.

DESCRIPTION

The *sec_rgy_attr_lookup_by_name( )* routine returns the named attribute for a named object. This routine is useful for an interactive editor.

For multi-valued attributes, this routine returns the first instance of the attribute. To retrieve every instance of the attribute, use the *sec_rgy_attr_lookup_by_id( )* call, supplying the attribute UUID returned in the *attr* parameter.

For attribute sets, the routine returns the attribute set instance, not the member instances. To retrieve all members of the set, use the *sec_rgy_attr_lookup_by_id( )* call, supplying the the attribute set UUID returned in the *attr* parameter.

Warning

This routine does not provide for input data to an attribute trigger query operation. If the named attribute is associated with a query attribute trigger, the attribute trigger will be called with no input attribute value information.

Permissions Required

The *sec_rgy_attr_lookup_by_name( )* routine requires the query permission set for the attribute type of the attribute instance identified by *attr_name*. These permissions are defined as part of the ACL manager set in the schema entry of each attribute type.

FILES

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_trig_svr_unavailable
Trigger server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_lookup_by_id* ( ), *sec_rgy_attr_lookup_no_expand* ( ).

NAME

sec_rgy_attr_lookup_no_expand — Reads a specified object's attribute(s), without expanding attribute sets into individual member attributes

SYNOPSIS

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_lookup_no_expand(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t name,
       sec_attr_cursor_t *cursor,
       unsigned32 num_attr_keys,
       unsigned32 space_avail,
       uuid_t attr_keys[ ],
       unsigned32 *num_returned,
       sec_attr_t attr_sets[ ],
       unsigned32 *num_left,
       error_status_t status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain
A value of type **sec_rgy_domain_t** that identifies the domain in which the named object resides. The valid values are as follows:

sec_rgy_domain_person
The *name* identifies a principal.

sec_rgy_domain_group
The *name* identifies a group.

sec_rgy_domain_org
The *name* identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name
A pointer to a **sec_rgy_name_t** character string containing the name of the person, group, or organization to which the attribute is attached.

num_attr_keys
An unsigned 32-bit integer that specifies the number of elements in the the *attr_keys* array. If *num_attr_keys* is set to 0, all attribute sets that the caller is authorized to see are returned.

space_avail
An unsigned 32-bit integer that specifies the size of the *attrs_sets* array.

attr_keys[ ]
An array of values of type **uuid_t** that specify the UUIDs of the attribute sets to be returned. The size of the *attr_keys* array is determined by the *num_attr_keys* parameter.

Input/Output

cursor
A pointer to a **sec_attr_cursor_t**. As an input parameter, *cursor* is a pointer to a **sec_attr_cursor_t** that is initialized by the *sec_rgy_attr_cursor_init( )* routine. As an output parameter, *cursor* is a pointer to a **sec_attr_cursor_t** that is positioned past the attribute sets returned in this call.

Output

num_returned
A pointer to a 32-bit integer that specifies the number of attribute sets returned in the *attrs* array.

attr_sets
An array of values of type **sec_attr_t** that contains the attribute sets retrieved by UUID. The size of the array is determined by *space_avail* and the length by *num_returned.*

num_left
A pointer to a 32-bit unsigned integer that supplies the number of attribute sets that were found but could not be returned because of space constraints in the *attr_sets* buffer. To ensure that all the attributes will be returned, increase the size of the *attr_sets* array by increasing the size of *space_avail* and *num_returned.*

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_lookup_no_expand( )* routine reads attribute sets. This routine is similar to the *sec_rgy_attr_lookup_by_id( )* routine with one exception: for attribute sets, *sec_rgy_attr_lookup_by_id( )* expands attribute sets and returns a **sec_attr_t** for each member in the set. This call does not. Instead it returns a **sec_attr_t** for the set instance only. The *sec_rgy_attr_lookup_no_expand( )* routine is useful for programmatic access.

cursor is a cursor of type **sec_attr_cursor_t** that establishes the point in the attribute set list from which the server should start processing the query. Use the *sec_rgy_attr_cursor_init( )* function to initialize *cursor*. If *cursor* is uninitialized, the server begins processing the query with the first attribute that satisfies the search criteria.

The *num_left* parameter contains the number of attribute sets that were found but could not be returned because of space constraints of the *attr_sets* array. (Note that this number may be inaccurate if the target server allows updates between successive queries.) To obtain all of the remaining attribute sets, set the size of the *attr_sets* array so that it is large enough to hold the number of attributes listed in *num_left*.

Permissions Required

The *sec_rgy_attr_lookup_no_expand( )* routine requires the query permission set for each attribute type identified in the *attr_keys* array. These permissions are defined as part of the ACL manager set in the schema entry of each attribute type.

FILES

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_bad_type
Invalid or unsupported attribute type.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_lookup_by_id* ( ), *sec_rgy_attr_lookup_by_name* ( ).

NAME

sec_rgy_attr_sch_aclmgr_strings — Returns printable ACL strings associated with an ACL manager protecting a bound to schema object

SYNOPSIS

```
#include <dce/sec_rgy_attr_sch.h>

void sec_rgy_attr_sch_aclmgr_strings(
       sec_rgy_handle_t context,
       sec_attr_component_name_t schema_name,
       uuid_t *acl_mgr_type,
       unsigned32 size_avail,
       *uuid_t *acl_mgr_type_chain,
       sec_acl_printstring_t *acl_mgr_info,
       boolean32 *tokenize,
       unsigned32 *total_num_printstrings,
       unsigned32 *size_used,
       sec_acl_printstring_t permstrings[],
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

schema_name
Reserved for future use.

acl_manager_type
A pointer to the UUID identifying the type of the ACL manager in question. There may be more than one type of ACL manager protecting the schema object whose ACL is bound to the input handle. Use this parameter to distinguish them. Use *sec_rgy_attr_sch_get_acl_mgrs( )* to acquire a list of the manager types protecting a given schema object.

size_avail
An unsigned 32-bit integer containing the allocated length of the *permstrings* array.

Output

acl_mgr_type_chain
If the target object ACL contains more than 32 permission bits, chains of manager types are used: each manager type holds one 32-bit segment of permissions. The UUID returned in *acl_mgr_type_chain* refers to the next ACL manager in the chain. If there are no more ACL managers in the chain, **uuid_nil** is returned.

acl_mgr_info
A pointer to a printstring that contains the ACL manager type's name, help information, and set of supported of permission bits.

tokenize
A pointer to a variable that specifies whether or not printstrings will be passed separately:

- TRUE indicates that the printstrings must be printed or passed separately.
- FALSE indicates that the printstrings are unambiguous and can be concatenated when printed without confusion.

total_num_printstrings
A pointer to an unsigned 32-bit integer containing the total number of permission entries supported by this ACL manager type.

size_used
A pointer to an unsigned 32-bit integer containing the number of permission entries returned in the *permstrings* array.

permstrings[ ]
An array of printstrings of type **sec_acl_printstring_t**. Each entry of the array is a structure containing the following three components:

printstring
A character string of maximum length **sec_acl_printstring_len** describing the printable representation of a specified permission.

helpstring
A character string of maximum length **sec_acl_printstring_help_len** containing some text that can be used to describe the specified permission.

permissions
A **sec_acl_permset_t** permission set describing the permissions that are represented with the companion printstring.

The array consists of one such entry for each permission supported by the ACL manager identified by *acl_mgr_type*.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_sch_aclmgr_strings( )* routine returns an array of printable representations (called ''printstrings'') for each permission bit or combination of permission bits the specified ACL manager supports. The ACL manager type specified by *acl_mgr_type* must be one of the types protecting the schema object bound to by *context*.

In addition to returning the printstrings, this routine also returns instructions about how to print the strings in the *tokenize* variable. If this variable is set to FALSE, the printstrings can be concatenated. If it is set to TRUE, the printstrings cannot be concatenated. For example a printstrings of `r` or `w` could be concatenated as `rw` without any confusion. However, printstrings in a form of `read` or `write`, should not be concatenated.

ACL managers often define aliases for common permission combinations. By convention, simple entries appear at the beginning of the *printstrings*[ ] array, and combinations appear at the end.

Permissions Required

The *sec_rgy_attr_sch_aclmgr_strings( )* routine requires the **r** permission on the schema object.

FILES

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_no_memory
Unable to allocate memory.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_sch_get_acl_mgrs*( ).

NAME

sec_rgy_attr_sch_create_entry — Create a schema entry

SYNOPSIS

```
#include <dce/sec_rgy_attr_sch.h>

void sec_rgy_attr_sch_create_entry(
       sec_rgy_handle_t context,
       sec_attr_component_name_t schema_name,
       sec_attr_schema_entry_t *schema_entry,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

schema_name
Reserved for future use.

schema_entry
A pointer to a **sec_attr_schema_entry_t** that contains the schema entry values for the schema in which the entry is to be created.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_sch_create_entry( )* routine creates schema entries that define attribute types.

Permissions Required

The *sec_rgy_attr_sch_create_entry( )* routine requires **i** permission on the schema object.

FILES

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_bad_acl_mgr_set
Invalid acl_mgr_set specified.

sec_attr_bad_acl_mgr_type
Invalid acl manager type.

sec_attr_bad_bind_authn_svc
Invalid authentication service specified in binding auth_info.

sec_attr_bad_bind_authz_svc
Invalid authorization service specified in binding auth_info.

sec_attr_bad_bind_info
Invalid binding information.

sec_attr_bad_bind_prot_level
Invalid protection level specified in binding auth_info.

sec_attr_bad_bind_svr_name
Invalid server name specified in binding auth_info.

sec_attr_bad_comment
Invalid comment text specified.

sec_attr_bad_encoding_type
Invalid encoding type specified.

sec_attr_bad_intercell_action
Invalid intercell action specified.

sec_attr_bad_name
Invalid attribute name specified.

sec_attr_bad_permset
Invalid permission set.

sec_attr_bad_scope
Invalid scope specified.

sec_attr_bad_uniq_query_accept
Invalid combination of unique_flag=true, query trigger, and intercell_action=accept.

sec_attr_name_exists
Attribute name already exists.

sec_attr_no_memory
Unable to allocate memory.

sec_attr_svr_read_only
Server is read only.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_trig_bind_info_missing
Trigger binding info must be specified.

sec_attr_type_id_exists
Attribute type id already exists.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_sch_delete_entry*( ), *sec_rgy_attr_sch_update*( ).

NAME

sec_rgy_attr_sch_cursor_alloc — Allocates resources to a cursor used with the *sec_rgy_attr_sch_scan( )* call

SYNOPSIS

```
#include <dce/sec_rgy_attr_sch.h>

void sec_rgy_attr_sch_cursor_alloc(
       dce_attr_cursor_t *cursor,
       error_status_t *status);
```

PARAMETERS

Output

cursor
A pointer to a **sec_attr_cursor_t**.

status
A pointer to the completion status. On successful completion, the call returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_sch_cursor_alloc( )* call allocates resources to a cursor used with the *sec_rgy_attr_sch_scan( )* call. This routine, which is a local operation, does not initialize *cursor*.

The *sec_rgy_attr_sch_cursor_init( )* routine, which makes a remote call, allocates and initializes the cursor. In addition, *sec_rgy_attr_sch_cursor_init( )* returns the total number of entries found in the schema as an output parameter; *sec_rgy_attr_sch_cursor_alloc( )* does not.

Permissions Required

None.

FILES

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.id** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_no_memory
Unable to allocate memory.

SEE ALSO

Functions: *sec_rgy_attr_sch_cursor_init*( ), *sec_rgy_attr_sch_cursor_release*( ), *sec_rgy_attr_sch_scan*( ).

NAME

sec_rgy_attr_sch_cursor_init — Initialize and allocate a cursor used with the *sec_rgy_attr_sch_scan( )* call

SYNOPSIS

```
#include <dce/sec_rgy_attr_sch.h>

void sec_rgy_attr_cursor_init(
       sec_rgy_handle_t context,
       sec_attr_component_name_t schema_name,
       unsigned32 *cur_num_entries,
       sec_attr_cursor_t *cursor,
       error_status_t status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

schema_name
Reserved for future use.

Output

cur_num_entries
A pointer to an unsigned 32-bit integer that specifies the total number of entries contained in the schema at the time of this call.

cursor
A pointer to a **sec_attr_cursor_t** that is initialized to the first entry in the the schema.

status
A pointer to the completion status. On successful completion, the call returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_sch_cursor_init( )* call initializes and allocates resources to a cursor used with the *sec_rgy_attr_sch_scan( )* call. This call makes remote calls to initialize the cursor. To limit the number of remote calls, use the *sec_rgy_attr_sch_cursor_alloc( )* call to allocate *cursor*, but not initialize it. Be aware, however, that the the *sec_rgy_attr_sch_cursor_init( )* call supplies the total number of entries found in the schema as an output parameter; the *sec_rgy_attr_sch_cursor_alloc( )* call does not.

If the cursor input to *sec_rgy_attr_sch_scan( )* has not been initialized, the *sec_rgy_attr_sch_scan( )* call will initialize it; if it has been initialized, *sec_rgy_attr_sch_scan( )* advances it.

Permissions Required

None.

FILES

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_no_memory
Unable to allocate memory.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_sch_cursor_release*( ), *sec_rgy_attr_sch_scan*( ), *sec_rgy_attr_sch_cursor_alloc*( ).

NAME

sec_rgy_attr_sch_cursor_release — Release states associated with a cursor used by the *sec_rgy_attr_sch_scan( )* routine

SYNOPSIS

```
#include <dce/sec_rgy_attr_sch.h>

void sec_rgy_attr_cursor_release(
       sec_attr_cursor_t *cursor,
       error_status_t *status);
```

PARAMETERS

Input/Output

cursor

A pointer to a **sec_attr_cursor_t**. As an input parameter, *cursor* must have been initialized to the first entry in a schema. As an output parameter, *cursor* is uninitialized with all resources releases.

Output

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_sch_cursor_init( )* routine releases the resources allocated to the cursor used by the *sec_rgy_attr_sch_scan( )* routine. This call is a local operation and makes no remote calls.

Permissions Required

None.

FILES

/usr/include/dce/sec_rgy_attr_sch.idl

The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

ERRORS

error_status_ok

The call was successful.

SEE ALSO

Functions: *sec_rgy_attr_sch_cursor_init*( ), *sec_rgy_attr_sch_cursor_allocate*( ), *sec_rgy_attr_sch_scan*( ).

NAME

sec_rgy_attr_sch_cursor_reset — Resets a cursor that has been allocated with either *sec_rgy_attr_sch_cursor_init( )* or *sec_rgy_attr_sch_cursor_alloc( )*

SYNOPSIS

```
#include <dce/sec_rgy_attr_sch.h>

void dce_attr_cursor_reset(
       sec_attr_cursor_t *cursor,
       error_status_t *status);
```

PARAMETERS

Input/Output

cursor

A pointer to a **sec_attr_cursor_t**. As an input parameter, an initialized *cursor*. As an output parameter, *cursor* is reset to the first attribute in the schema.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_sch_cursor_reset( )* routine resets a **dce_attr_cursor_t** that has been allocated by either a *sec_rgy_attr_sch_cursor_init( )* or *sec_rgy_attr_sch_cursor_alloc( )*. The reset cursor can then be used to process a new *sec_rgy_attr_sch_scan* query by reusing the cursor instead of releasing and re-allocating it. This is a local operation and makes no remote calls.

Permissions Required

None.

FILES

/usr/include/dce/sec_rgy_attr_sch.idl

The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

ERRORS

error_status_ok

The call was successful.

sec_attr_bad_cursor

Invalid cursor.

SEE ALSO

Functions: *sec_rgy_attr_sch_cursor_init*( ), *sec_rgy_attr_sch_cursor_alloc*( ), *sec_rgy_attr_sch_scan*( ).

NAME

sec_rgy_attr_sch_delete_entry — Delete a schema entry

SYNOPSIS

```
#include <dce/sec_rgy_attr_sch.h>

void sec_rgy_attr_sch_delete_entry(
       sec_rgy_handle_t context,
       sec_attr_component_name_t schema_name,
       uuid_t *attr_id,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

schema_name
Reserved for future use.

attr_id
A pointer to a **uuid_t** that identifies the schema entry to be deleted.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_sch_delete_entry( )* routine deletes a schema entry. Because this is a radical operation that invalidates any existing attributes of this type on objects dominated by the schema, access to this operation should be severely limited.

Permissions Required

The *sec_rgy_attr_sch_delete_entry( )* routine requires the **d** permission on the schema object.

FILES

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_no_memory
Unable to allocate memory.

sec_attr_sch_entry_not_found
Schema entry not found.

sec_attr_svr_read_only
Server is read only.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_sch_create_entry*( ), *sec_rgy_attr_sch_update_entry*( ).

NAME

sec_rgy_attr_sch_get_acl_mgrs — Retrieve the manager types of the ACLs protecting the objects dominated by a named schema

SYNOPSIS

```
#include <dce/sec_rgy_attr_sch.h>

void sec_rgy_attr_sch_get_acl_mgrs(
       sec_rgy_handle_t context,
       sec_attr_component_name_t schema_name,
       unsigned32 size_avail,
       unsigned32 *size_used,
       unsigned32 *num_acl_mgr_types,
       uuid_t acl_mgr_types[],
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

schema_name
Reserved for future use.

size_avail
An unsigned 32-bit integer containing the allocated length of the *acl_manager_types* array.

Output

size_used
An unsigned 32-bit integer containing the number of output entries returned in the *acl_mgr_types*[ ] array.

num_acl_mgr_types
An unsigned 32-bit integer containing the number of types returned in the *acl_mgr_types* array. This may be greater than *size_used* if there was not enough space allocated by *size_avail* for all the manager types in the *acl_manager_types* array.

acl_mgr_types[ ]
An array of the length specified in *size_avail* to contain UUIDs (of type **uuid_t**) identifying the types of ACL managers protecting the target object.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_sch_get_acl_mgrs( )* routine returns a list of the manager types protecting the schema object identified by *context*.

ACL editors and browsers can use this operation to determine the ACL manager types protecting a selected schema object. Then, using the *sec_rgy_attr_sch_aclmgr_strings( )* routine, they can determine how to format for display the permissions supported by that ACL manager type.

Permissions Required

The *sec_rgy_attr_sch_get_acl_mgrs( )* routine requires the **r** permission on the schema object.

FILES

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_no_memory
Unable to allocate memory.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_sch_aclmgr_strings*( ).

NAME

sec_rgy_attr_sch_lookup_by_id — Read a schema entry identified by UUID

SYNOPSIS

```
#include <dce/sec_rgy_attr_sch.h>

void sec_rgy_attr_sch_lookup_by_id(
       sec_rgy_handle_t context,
       sec_attr_component_name_t schema_name,
       uuid_t *attr_id,
       sec_attr_schema_entry_t *schema_entry,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

schema_name
Reserved for future use.

attr_id
A pointer to a **uuid_t** that identifies a schema entry.

Output

schema_entry
A **sec_attr_schema_entry_t** that contains an entry identified by *attr_id*.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_sch_lookup_by_id( )* routine reads a schema entry identified by *attr_id*. This routine is useful for programmatic access.

Permissions Required

The *sec_rgy_attr_sch_lookup_by_id( )* routine requires the **r** permission on the schema object.

FILES

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_no_memory
Unable to allocate memory.

sec_attr_sch_entry_not_found
Schema entry not found.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_sch_lookup_by_name*( ), *sec_rgy_attr_sch_scan*( ).

NAME

sec_rgy_attr_sch_lookup_by_name — Read a schema entry identified by name

SYNOPSIS

```
#include <dce/sec_rgy_attr_sch.h>

void sec_rgy_attr_sch_lookup_by_name(
       sec_rgy_handle_t context,
       sec_attr_component_name_t schema_name,
       char *attr_name,
       sec_attr_schema_entry_t *schema_entry,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

schema_name
Reserved for future use.

attr_name
A pointer to a character string that identifies the schema entry.

Output

schema_entry
A **sec_attr_schema_entry_t** that contains the schema entry identified by *attr_name*.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_sch_lookup_by_name( )* routine reads a schema entry identified by name. This routine is useful for use with an interactive editor.

Permissions Required

The *sec_rgy_attr_sch_lookup_by_name( )* routine requires the **r** permission on the schema object.

FILES

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_bad_name
Invalid attribute name specified.

sec_attr_no_memory
Unable to allocate memory.

sec_attr_sch_entry_not_found
Schema entry not found.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_sch_lookup_by_id*( ), *sec_rgy_attr_sch_scan*( ).

NAME

sec_rgy_attr_sch_scan — Read a specified number of schema entries

SYNOPSIS

```
#include <dce/sec_rgy_attr_sch.h>

void sec_rgy_attr_sch_scan(
       sec_rgy_handle_t context,
       sec_attr_component_name_t schema_name,
       sec_attr_cursor_t *cursor,
       unsigned32 num_to_read,
       unsigned32 *num_read,
       sec_attr_schema_entry_t schema_entries[],
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

schema_name
Reserved for future use.

num_to_read
An unsigned 32-bit integer specifying the size of the *schema_entries*[ ] array and the maximum number of entries to be returned.

Input/Output

cursor
A pointer to a **sec_attr_cursor_t**. As input *cursor* must be allocated and can be initialized. If *cursor* is not initialized, *sec_rgy_attr_sch_scan( )* will initialize. As output, *cursor* is positioned at the first schema entry after the returned entries.

Output

num_read
A pointer an unsigned 32-bit integer specifying the number of entries returned in *schema_entries*.

schema_entries[ ]
A **sec_attr_schema_entry_t** that contains an array of the returned schema entries.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_sch_scan( )* routine reads schema entries. The read begins at the entry at which the input *cursor* is positioned and ends after the number of entries specified in *num_to_read*.

The input *cursor* must have been allocated by either the *sec_rgy_attr_sch_cursor_init( )* or the *sec_rgy_attr_sch_cursor_alloc( )* call. If the input *cursor* is not initialized, *sec_rgy_attr_sch_scan( )* initializes it; if cursor is initialized, *sec_rgy_attr_sch_scan( )* simply advances it.

To read all entries in a schema, make successive *sec_rgy_attr_sch_scan( )* calls. When all entries have been read, the call returns the message **no_more_entries**.

This routine is useful as a browser.

Permissions Required

The *sec_rgy_attr_sch_scan( )* routine requires **r** permission on the schema object.

FILES

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_bad_cursor
Invalid cursor.

sec_attr_no_memory
Unable to allocate memory.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_sch_cursor_init*( ), *sec_rgy_attr_sch_cursor_alloc*( ), *sec_rgy_attr_sch_cursor_release*( ).

NAME

sec_rgy_attr_sch_update_entry — Update a schema entry

SYNOPSIS

```
#include <dce/sec_rgy_attr_sch.h>

void sec_rgy_attr_sch_update_entry(
       sec_rgy_handle_t context,
       sec_attr_component_name_t schema_name,
       sec_attr_schema_entry_parts_t modify_parts,
       sec_attr_schema_entry_t *schema_entry,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

schema_name
Reserved for future use.

modify_parts
A value of type **sec_attr_schema_entry_parts_t** that identifies the fields in *schema_entry* that can be modified.

schema_entry
A pointer to a **sec_attr_schema_entry_t** that contains the schema entry values for the schema entry to be updated.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_sch_update_entry( )* routine modifies schema entries. Only those schema entry fields set to be modified in the **sec_attr_schema_entry_parts_t** data type can be modified.

Some schema entry components can never be modified. Instead to make any changes to these components, the schema entry must be deleted (which deletes all attribute instances of that type) and recreated. The schema entry components that can never be modified are listed below:

- Attribute name
- Reserved flag
- Apply defaults flag
- Intercell action flag
- Trigger binding
- Comment

Fields that are arrays of structures (such as **acl_mgr_set** and **trig_binding**) are completely replaced by the new input array. This operation cannot be used to add a new element to the existing array.

Permissions Required

The *sec_rgy_attr_sch_update_entry( )* routine requires the **M** permission on the schema object.

FILES

/usr/include/dce/sec_rgy_attr_sch.idl
The **idl** file from which **dce/sec_rgy_attr_sch.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_bad_acl_mgr_set
Invalid acl_mgr_set specified.

sec_attr_bad_acl_mgr_type
Invalid acl manager type.

sec_attr_bad_bind_authn_svc
Invalid authentication service specified in binding auth_info.

sec_attr_bad_bind_authz_svc
Invalid authorization service specified in binding auth_info.

sec_attr_bad_bind_info
Invalid binding information.

sec_attr_bad_bind_prot_level
Invalid protection level specified in binding auth_info.

sec_attr_bad_bind_svr_name
Invalid server name specified in binding auth_info.

sec_attr_bad_comment
Invalid comment text specified.

sec_attr_bad_intercell_action
Invalid intercell action specified.

sec_attr_bad_name
Invalid attribute name specified.

sec_attr_bad_permset
Invalid permission set.

sec_attr_bad_uniq_query_accept
Invalid combination of unique_flag=true, query trigger, and intercell_action=accept.

sec_attr_field_no_update
Field not modifiable.

sec_attr_name_exists
Attribute name already exists.

sec_attr_no_memory
Unable to allocate memory.

sec_attr_sch_entry_not_found
Schema entry not found.

sec_attr_svr_read_only
Server is read only.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_trig_bind_info_missing
Trigger binding info must be specified.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_sch_delete_entry*( ), *sec_rgy_attr_sch_create_entry*( ).

NAME

sec_rgy_attr_test_and_update — Updates specified attribute instances for a specified object only if a set of control attribute instances match the object's existing attribute instances

SYNOPSIS

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_test_and_update (
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t name,
       unsigned32 num_to_test,
       sec_attr_t test_attrs[ ],
       unsigned32 num_to_write,
       sec_attr_t update_attrs[ ],
       signed32 *failure_index,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

A value of type **sec_rgy_domain_t** that identifies the registry domain in which the object specified by *name* resides. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name

A character string of type **sec_rgy_name_t** specifying the name of the person, group, or organization to which the attribute is attached.

num_to_test

An unsigned 32-bit integer that specifies the number of elements in the *test_attrs* array. This integer must be greater than 0.

test_attrs[ ]

An array of values of type **sec_attr_t** that specifies the control attributes. The update takes place only if the types and values of the control attributes exactly match those of the attribute instances on the named registry object. The size of the array is determined by *num_to_test*.

num_to_write

A 32-bit integer that specifies the number of attribute instances returned in the *update_attrs* array.

update_attrs
An array of values of type **sec_attr_t** that specifies the attribute instances to be updated. The size of the array is determined by *num_to_write*.

Output

failure_index
In the event of an error, *failure_index* is a pointer to the element in the *update_attrs* array that caused the update to fail. If the failure cannot be attributed to a specific attribute, the value of *failure_index* is **-1**.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_test_and_update( )* routine updates an attribute only if the set of control attributes specified in the *test_attrs* match attributes that already exist for the object.

This update is an atomic operation: if any of the control attributes do not match existing attributes, none of the updates are performed, and if an update should be performed, but the write cannot occur for whatever reason to any member of the *update_attrs* array, all updates are aborted. The attribute causing the update to fail is identified in *failure_index*. If the failure cannot be attributed to a given attribute, *failure_index* contains **-1**.

If an attribute instance already exists which is identical in both **attr_id** and **attr_value** to an attribute specified in *in_attrs*, the existing attribute information is overwritten by the new information. For multi-valued attributes, every instance with the same **attr_id** is overwritten with the supplied values.

If an attribute instance does not exist, it is created.

If you specify an attribute set for updating, the update applies to the set instance, the set itself, not the members of the set. To update a member of an attribute set, supply the UUID of the set member.

If an input attribute is associated with an update attribute trigger server, the attribute trigger server is invoked (by the *sec_attr_trig_update( )* function) and the *in_attr* array is supplied as input. The output attributes from the update attribute trigger server are stored in the registry database and returned in the *out_attrs* array. Note that the update attribute trigger server may modify the values before they are used to update the registry database. This is the only circumstance under which the values in the *out_attrs* array differ from the values in the *in_attrs* array.

Permissions Required

The *sec_rgy_attr_test_and_update( )* routine requires the test permission and the update permission set for each attribute type identified in the *test_attrs* array. These permissions are defined as part of the ACL manager set in the schema entry of each attribute type.

FILES

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_bad_encoding_type
Invalid encoding type specified.

sec_attr_bad_type
Invalid or unsupported attribute type.

sec_attr_not_unique
Attribute value is not unique.

sec_rgy_read_only
Registry is read only.

sec_attr_svr_read_only
Server is read only.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_trig_svr_unavailable
Trigger server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_update*( ), *sec_rgy_attr_delete*( ).

NAME

sec_rgy_attr_update — Creates and updates attribute instances for a specified object

SYNOPSIS

```
#include <dce/sec_rgy_attr.h>

void sec_rgy_attr_update (
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t name,
       unsigned32 num_to_write,
       unsigned32 space_avail,
       sec_attr_t in_attrs[],
       unsigned32 *num_returned,
       sec_attr_t out_attrs[],
       unsigned32 *num_left,
       signed32 *failure_index,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

A value of type **sec_rgy_domain_t** that identifies the registry domain in which the object specified by *name* resides. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

This parameter is ignored if *name* is **policy** or **replist**.

name

A character string of type **sec_rgy_name_t** specifying the name of the person, group, or organization to which the attribute is attached.

num_to_write

A 32-bit unsigned integer that specifies the number of elements in the *in_attrs* array. This integer must be greater than 0.

space_avail

A 32-bit unsigned integer that specifies the size of the *out_attrs* array. This integer must be greater than 0.

in_attrs[ ]

An array of values of type **sec_attr_t** that specifies the attribute instances to be updated. The size of the array is determined by *num_to_write*.

Output

num_returned
A pointer to an unsigned 32-bit integer that specifies the number of attribute instances returned in the *out_attrs* array.

out_attrs[ ]
An array of values of type **sec_attr_t** that specifies the updated attribute instances. Not that only if these attributes were processed by an update attribute trigger server will they differ from the attributes in the *in_attrs* array. The size of the array is determined by *space_avail* and the length by *num_returned*.

num_left
A pointer to an unsigned 32-bit integer that supplies the number of attributes that could not be returned because of space constraints in the *out_attrs* buffer. To ensure that all the attributes will be returned, increase the size of the *out_attrs* array by increasing the size of *space_avail* and *num_returned*.

failure_index
In the event of an error, *failure_index* is a pointer to the element in the *in_attrs* array that caused the update to fail. If the failure cannot be attributed to a specific attribute, the value of *failure_index* is **-1**.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_attr_update( )* routine creates new attribute instances and updates existing attribute instances attached to a object specified by name and Registry domain. The instances to be created or updated are passed as an array of **sec_attr_t** data types. This is an atomic operation: if the creation of any attribute in the *in_attrs* array fails, all updates are aborted. The attribute causing the update to fail is identified in *failure_index*. If the failure cannot be attributed to a given attribute, *failure_index* contains **-1**.

The *in_attrs* array, which specifies the attributes to be created, contains values of type **sec_attr_t**. These values are:

- **attr_id**, a UUID that identifies the attribute type
- **attr_value**, values of **sec_attr_value_t** that specify the attribute's encoding type and values.

If an attribute instance already exists which is identical in both **attr_id** and **attr_value** to an attribute specified in *in_attrs*, the existing attribute information is overwritten by the new information. For multi-valued attributes, every instance with the same **attr_id** is overwritten with the supplied values.

If an attribute instance does not exist, it is created.

For multi-valued attributes, because every instance of the multi-valued attribute is identified by the same UUID, every instance is overwritten with the supplied value. To change only one of the values, you must supply the values that should be unchanged as well as the new value.

To create instances of multi-valued attributes, create individual **sec_attr_t** data types to define each multi-valued attribute instance and then pass all of them in in the input array.

If an input attribute is associated with an update attribute trigger server, the attribute trigger server is invoked (by the *sec_attr_trig_update( )* function) and the *in_attr* array is supplied as input. The output attributes from the update attribute trigger server are stored in the registry

database and returned in the *out_attrs* array. Note that the update attribute trigger server may modify the values before they are used to update the registry database. This is the only circumstance under which the values in the *out_attrs* array differ from the values in the *in_attrs* array.

Permissions Required

The *sec_rgy_attr_update( )* routine requires the update permission set for each attribute type identified in the *in_attrs* array. These permissions are defined as part of the ACL manager set in the schema entry of each attribute type.

FILES

/usr/include/dce/sec_rgy_attr.idl
The **idl** file from which **dce/sec_rgy_attr.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_attr_bad_encoding_type
Invalid encoding type specified.

sec_attr_bad_type
Invalid or unsupported attribute type.

sec_attr_inst_exists
Attribute instance already exists.

sec_attr_not_unique
Attribute value is not unique.

sec_rgy_read_only
Registry is read only.

sec_attr_svr_read_only
Server is read only.

sec_attr_svr_unavailable
Server is unavailable.

sec_attr_trig_svr_unavailable
Trigger server is unavailable.

sec_attr_unauthorized
Unauthorized to perform this operation.

SEE ALSO

Functions: *sec_rgy_attr_delete*( ), *sec_rgy_attr_test_and_update*( ).

NAME

sec_rgy_auth_plcy_get_effective — Returns the effective authentication policy for an account

SYNOPSIS

```
#include <dce/policy.h>

void sec_rgy_auth_plcy_get_effective(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *account,
       sec_rgy_plcy_auth_t *auth_policy,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

account

A pointer to the account login name (type **sec_rgy_login_name_t**). A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. If all three fields contain empty strings, the authentication policy returned is that of the registry.

Output

auth_policy

A pointer to the **sec_rgy_plcy_auth_t** structure to receive the authentication policy. The authentication policy structure contains the maximum lifetime for an authentication ticket, and the maximum amount of time for which one can be renewed.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_auth_plcy_get_effective( )* routine returns the effective authentication policy for the specified account. The authentication policy in effect is the more restrictive of the registry and the account policies for each policy category. If no account is specified, the registry's authentication policy is returned.

Permissions Required

The *sec_rgy_auth_plcy_get_effective( )* routine requires the **r** (**read**) permission on the policy object from which the data is to be returned. If an account is specified and an account policy exists, the routine also requires the **r** (**read**) permission on the account principal.

FILES

/usr/include/dce/policy.idl

The **idl** file from which **dce/policy.h** was derived.

ERRORS

error_status_ok

The call was successful.

sec_rgy_object_not_found
The specified account could not be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO
Functions: *sec_rgy_auth_plcy_get_info* ( ), *sec_rgy_auth_plcy_set_info* ( ).

NAME

sec_rgy_auth_plcy_get_info — Returns the authentication policy for an account

SYNOPSIS

```
#include <dce/policy.h>

void sec_rgy_auth_plcy_get_info(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *account,
       sec_rgy_plcy_auth_t *auth_policy,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

account

A pointer to the account login name (type **sec_rgy_login_name_t**). A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account.

Output

auth_policy

A pointer to the **sec_rgy_plcy_auth_t** structure to receive the authentication policy. The authentication policy structure contains the maximum lifetime for an authentication ticket, and the maximum amount of time for which one can be renewed.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_auth_plcy_get_info( )* routine returns the authentication policy for the specified account. If no account is specified, the registry's authentication policy is returned.

Permissions Required

The *sec_rgy_auth_plcy_get_info( )* routine requires the **r** (**read**) permission on the policy object or account principal from which the data is to be returned.

NOTES

The actual policy in effect will not correspond precisely to what is returned by this call if the overriding registry authentication policy is more restrictive than the policy for the specified account. Use *sec_rgy_auth_plcy_get_effective( )* to return the policy currently in effect for the given account.

FILES

/usr/include/dce/policy.idl

The **idl** file from which **dce/policy.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_object_not_found
No account with the given login name could be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_auth_plcy_get_effective*( ), *sec_rgy_auth_plcy_set_info*( ).

NAME

sec_rgy_auth_plcy_set_info — Sets the authentication policy for an account

SYNOPSIS

```
#include <dce/policy.h>

void sec_rgy_auth_plcy_set_info(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *account,
       sec_rgy_plcy_auth_t *auth_policy,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

account

A pointer to the account login name (type **sec_rgy_login_name_t**). A login name is composed of three character strings, containing the principal, group, and organization (PGO) names corresponding to the account. All three names must be completely specified.

auth_policy

A pointer to the **sec_rgy_plcy_auth_t** structure containing the authentication policy. The authentication policy structure contains the maximum lifetime for an authentication ticket, and the maximum amount of time for which one can be renewed.

Output

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_auth_plcy_set_info( )* routine sets the indicated authentication policy for the specified account. If no account is specified, the authentication policy is set for the registry as a whole.

Permissions Required

The *sec_rgy_auth_plcy_set_info( )* routine requires the **a** (**auth_info**) permission on the policy object or account principal for which the data is to be set.

NOTES

The policy set on an account may be less restrictive than the policy set for the registry as a whole. In this case, the change in policy has no effect, since the effective policy is the most restrictive combination of the principal and registry authentication policies. (See the *sec_rgy_auth_plcy_get_effective( )* routine).

FILES

/usr/include/dce/policy.idl

The **idl** file from which **dce/policy.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The user is not authorized to update the specified record.

sec_rgy_object_not_found
No account with the given login name could be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_auth_plcy_get_effective*( ), *sec_rgy_auth_plcy_get_info*( ).

NAME

sec_rgy_cell_bind — Binds to a registry in a cell

SYNOPSIS

```
#include <dce/binding.h>

void sec_rgy_cell_bind(
       unsigned_char_t *cell_name,
       sec_rgy_bind_auth_info_t *auth_info,
       sec_rgy_handle_t *context,
       error_status_t *status);
```

PARAMETERS

Input

cell_name

A character string (type **unsigned_char_t**) containing the name of the cell in question. Upon return, a Security Server for that cell is associated with *context*, the registry server handle. The cell must be specified completely and precisely. This routine offers none of the pathname resolving services of *sec_rgy_site_bind( )*.

auth_info

A pointer to the **sec_rgy_bind_auth_info_t** structure that identifies the authentication protocol, protection level, and authorization protocol to use in establishing the binding. (See the *rpc_binding_set_auth_info( )* routine).

Output

context

A pointer to a **sec_rgy_handle_t** variable. Upon return, this contains a registry server handle indicating (bound to) the desired registry site.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_cell_bind( )* routine establishes a relationship with a registry site at an arbitrary level of security. The *cell_name* parameter identifies the target cell.

FILES

/usr/include/dce/binding.idl

The **idl** file from which **dce/binding.h** was derived.

ERRORS

error_status_ok

The call was successful.

sec_rgy_server_unavailable

The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_site_bind*( ).

NAME

sec_rgy_cursor_reset — Resets the registry database cursor

SYNOPSIS

```
#include <dce/misc.h>

void sec_rgy_cursor_reset(
       sec_rgy_cursor_t *cursor);
```

PARAMETERS

Input/Output

cursor
A pointer into the registry database.

DESCRIPTION

The *sec_rgy_cursor_reset( )* routine resets the database cursor to return the first suitable entry. A cursor is a pointer into the registry. It serves as a place holder when returning successive items from the registry.

A cursor is bound to a particular server. In other words, a cursor that is in use with one replica of the registry has no meaning for any other replica. If a calling program attempts to use a cursor from one replica with another, the cursor is reset and the routine for which the cursor was specified returns the first item in the database.

A cursor that is in use with one call cannot be used with another. For example, you cannot use the same cursor on a call to *sec_rgy_acct_get_projlist( )* and *sec_rgy_pgo_get_next( )*. The behavior in this case is undefined.

FILES

/usr/include/dce/misc.idl
The **idl** file from which **dce/misc.h** was derived.

EXAMPLES

The following example illustrates use of the cursor within a loop. The initial *sec_rgy_cursor_reset( )* call resets the cursor to point to the first item in the registry. Successive calls to *sec_rgy_pgo_get_next( )* return the next PGO item and update the cursor to reflect the last item returned. When the end of the list of PGO items is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter.

```
sec_rgy_cursor_reset(&cursor);
do {
    sec_rgy_pgo_get_next(context, domain, scope, &cursor,
                    &item, name &status);
    if (status == error_status_ok) {
         /* Print formatted PGO item info */
    }
}while (status == error_status_ok);
```

SEE ALSO

Functions: *sec_rgy_acct_get_projlist*( ), *sec_rgy_acct_lookup*( ), *sec_rgy_pgo_get_by_id*( ), *sec_rgy_pgo_get_by_name*( ), *sec_rgy_pgo_get_by_unix_num*( ), *sec_rgy_pgo_get_members*( ), *sec_rgy_pgo_get_next*( ).

NAME

sec_rgy_login_get_effective — Returns the effective login data for an account

SYNOPSIS

```
#include <dce/misc.h>

void sec_rgy_login_get_effective(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *login_name,
       sec_rgy_acct_key_t *key_parts,
       sec_rgy_sid_t *sid,
       sec_rgy_unix_sid_t *unix_sid,
       sec_rgy_acct_user_t *user_part,
       sec_rgy_acct_admin_t *admin_part,
       sec_rgy_plcy_t *policy_data,
       signed32 max_number,
       signed32 *supplied_number,
       uuid_t id_projlist[],
       signed32 unix_projlist[],
       signed32 *num_projects,
       sec_rgy_name_t cell_name,
       uuid_t *cell_uuid,
       sec_override_fields_t *overridden,
       error_status_t *status);
```

PARAMETERS

Input

context
The registry server handle.

max_number
The maximum number of projects to be returned by the call. This must be no larger than the allocated size of the *projlist* arrays.

Input/Output

login_name
A pointer to the account login name. A login name is composed of the names for the account's principal, group, and organization (PGO) items.

Output

key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

sid
A pointer to a **sec_rgy_sid_t** structure to receive the returned Subject Identifier (SID) for the account. This structure consists of the UUIDs for the account's PGO items.

unix_sid
A pointer to a **sec_rgy_unix_sid_t** structure to receive the returned UNIX Subject Identifier (SID) for the account. This structure consists of the UNIX numbers for the account's PGO items.

user_part
A pointer to a **sec_rgy_acct_user_t** structure to receive the returned user data for the account.

admin_part
A pointer to a **sec_rgy_acct_admin_t** structure to receive the returned administrative data for the account.

policy_data
A pointer to a **sec_rgy_policy_t** structure to receive the policy data for the account. The policy data is associated with the account's organization, as identified in the login name.

supplied_number
A pointer to the actual number of projects returned. This will always be less than or equal to the *max_number* supplied on input.

id_projlist[ ]
An array to receive the UUID of each project returned. The size allocated for the array is given by *max_number*. If this value is less than the total number of projects in the account project list, multiple calls must be made to return all of the projects.

unix_projlist[ ]
An array to receive the UNIX number of each project returned. The size allocated for the array is given by *max_number*. If this value is less than the total number of projects in the account project list, multiple calls must be made to return all of the projects.

num_projects
A pointer indicating the total number of projects in the specified account's project list.

cell_name
The name of the account's cell.

cell_uuid
The UUID for the account's cell.

overridden
A pointer to a 32-bit set of flags identifying the local overrides, if any, for the account login information.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_login_get_effective( )* routine returns effective login information for the specified account. Login information is extracted from the account's entry in the registry database. Effective login information is a combination of the login information from the registry database and any login overrides defined for the account on the local machine.

The *overridden* parameter indicates which, if any, of the following local overrides have been defined for the account:

- The UNIX user ID

- The group ID
- The encrypted password
- The account's miscellaneous information (**gecos**) field
- The account's home directory
- The account's login shell

Local overrides for account login information are defined in the **/etc/passwd_override** file and apply only to the local machine.

FILES

/usr/include/dce/misc.idl
The **idl** file from which **dce/misc.h** was derived.

/etc/passwd_override
The file that defines local overrides for account login information.

ERRORS

error_status_ok
The call was successful.

sec_rgy__object_not_found
The specified account could not be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_acct_add*( ), *sec_rgy_login_get_info*( ).

NAME

sec_rgy_login_get_info — Returns login information for an account

SYNOPSIS

```
#include <dce/misc.h>

void sec_rgy_login_get_info(
       sec_rgy_handle_t context,
       sec_rgy_login_name_t *login_name,
       sec_rgy_acct_key_t *key_parts,
       sec_rgy_sid_t *sid,
       sec_rgy_unix_sid_t *unix_sid,
       sec_rgy_acct_user_t *user_part,
       sec_rgy_acct_admin_t *admin_part,
       sec_rgy_plcy_t *policy_data,
       signed32 max_number,
       signed32 *supplied_number,
       uuid_t id_projlist[],
       signed32 unix_projlist[],
       signed32 *num_projects,
       sec_rgy_name_t cell_name,
       uuid_t *cell_uuid,
       error_status_t *status);
```

PARAMETERS

Input

context
The registry server handle.

max_number
The maximum number of projects to be returned by the call. This must be no larger than the allocated size of the *projlist* arrays.

Input/Output

login_name
A pointer to the account login name. A login name is composed of the names for the account's principal, group, and organization (PGO) items.

Output

key_parts
A pointer to the minimum abbreviation allowed when logging in to the account. Abbreviations are not currently implemented and the only legal value is **sec_rgy_acct_key_person**.

sid
A pointer to a **sec_rgy_sid_t** structure to receive the UUID's representing the account's PGO items.

unix_sid
A pointer to a **sec_rgy_unix_sid_t** structure to receive the UNIX numbers for the account's PGO items.

user_part
A pointer to a **sec_rgy_acct_user_t** structure to receive the returned user data for the account.

admin_part
A pointer to a **sec_rgy_acct_admin_t** structure to receive the returned administrative data for the account.

policy_data
A pointer to a **sec_rgy_policy_t** structure to receive the policy data for the account. The policy data is associated with the account's organization, as identified in the login name.

supplied_number
A pointer to the actual number of projects returned. This will always be less than or equal to the *max_number* supplied on input.

id_projlist[ ]
An array to receive the UUID of each project returned. The size allocated for the array is given by *max_number*. If this value is less than the total number of projects in the account project list, multiple calls must be made to return all of the projects.

unix_projlist[ ]
An array to receive the UNIX number of each project returned. The size allocated for the array is given by *max_number*. If this value is less than the total number of projects in the account project list, multiple calls must be made to return all of the projects.

num_projects
A pointer indicating the total number of projects in the specified account's project list.

cell_name
The name of the account's cell.

cell_uuid
The UUID for the account's cell.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_login_get_info( )* routine returns login information for the specified account. This information is extracted from the account's entry in the registry database. To return any local overrides for the account's login data, use *sec_rgy_login_get_effective( )*.

Permissions Required

The *sec_rgy_login_get_info( )* routine requires the **r** (**read**) permission on the account principal from which the data is to be returned.

FILES

/usr/lib/dce/misc.idl
The **idl** file from which **dce/misc.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_object_not_found
The specified account could not be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO
Functions: *sec_rgy_acct_add*( ), *sec_rgy_login_get_effective*( ).

NAME

sec_rgy_pgo_add — Adds a PGO item to the registry database

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_add(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t name,
       sec_rgy_pgo_item_t *pgo_item,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

name
A pointer to a **sec_rgy_name_t** character string containing the name of the new PGO item.

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure containing the data for the new PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item may have (or belong to) a concurrent group set.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_add( )* routine adds a PGO item to the registry database.

The PGO data consists of the following:

- The Universal Unique Identifier (UUID) of the PGO item. Specify **NULL** to have the registry server create a new UUID for an item.

- The UNIX number for the PGO item. If the registry uses embedded UNIX IDs (where a subset of the UUID bits represent the UNIX ID), then the specified ID must match the UUID, if both are specified. Use a value of -1 for the UNIX number to match any value.
- The quota for subaccounts allowed for this item entry.
- The full name of the PGO item.
- Flags (in the **sec_rgy_pgo_flags_t** format) indicating whether
  - — A principal item is an alias.
  - — The PGO item can be deleted from the registry.
  - — A principal item can have a concurrent group set.
  - — A group item can appear in a concurrent group set.

Permissions Required

The *sec_rgy_pgo_add( )* routine requires the **i** (**insert**) permission on the parent directory in which the the PGO item is to be created.

NOTES

An account can be added to the registry database only when all its constituent PGO items are already in the database, and the appropriate membership relationships between them are established. For example, to establish an account with principal name **tom**, group name **writers**, and organization name **hp**, all three names must exist as independent PGO items in the database. Furthermore, **tom** must be a member of **writers**, which must be a member of **hp**. (See *sec_rgy_acct_add( )* to add an account to the registry.)

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The client program is not authorized to add the specified PGO item.

sec_rgy_object_exists
A PGO item already exists with the name given in *name*.

sec_rgy_server_unavailable
The Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_delete*( ), *sec_rgy_pgo_rename*( ), *sec_rgy_pgo_replace*( ), *sec_rgy_acct_add*( ).

NAME

sec_rgy_pgo_add_member — Adds a person to a group or organization

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_add_member(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t go_name,
       sec_rgy_name_t person_name,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

This variable identifies the type of the person, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_group

The *go_name* parameter identifies a group.

sec_rgy_domain_org

The *go_name* parameter identifies an organization.

go_name

A character string (type **sec_rgy_name_t**) containing the name of the group or organization to which the specified person will be added.

person_name

A character string (type **sec_rgy_name_t**) containing the name of the person to be added to the membership list of the group or organization specified by *go_name*.

Output

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_add_member( )* routine adds a member to the membership list of a group or organization in the registry database.

Permissions Required

The *sec_rgy_pgo_add_member( )* routine requires the **M** (**Member_list**) permission on the group or organization item specified by *go_name*. If *go_name* specifies a group, the routine also requires the **g** (**groups**) permission on the principal *person_name*.

NOTES

An account can be added to the registry database only when all its constituent PGO items are already in the database, and the appropriate membership relationships between them are established. For example, to establish an account with person name **tom**, group name **writers**, and organization name **hp**, all three names must exist as independent PGO items in the database. Furthermore, **tom** must be a member of **writers**, which must be a member of **hp** (See the *sec_rgy_acct_add( )* routine to add an account to the registry.)

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_bad_domain
An invalid domain was specified. A member can be added only to a group or organization, not a person.

sec_rgy_not_authorized
The client program is not authorized to add members to the specified group or organization.

sec_rgy_object_not_found
The registry server could not find the specified name.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_delete_member*( ), *sec_rgy_pgo_get_members*( ), *sec_rgy_pgo_is_member*( ).

NAME

sec_rgy_pgo_delete — Deletes a PGO item from the registry database

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_delete(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t name,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

This variable identifies the type of principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

name

A pointer to a **sec_rgy_name_t** character string containing the name of the PGO item to be deleted.

Output

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_delete( )* routine deletes a PGO item from the registry database. Any account depending on the deleted PGO item is also deleted.

Permissions Required

The *sec_rgy_pgo_delete( )* routine requires the following permissions:

- The **d** (**delete**) permission on the parent directory that contains the the PGO item to be deleted.
- The **D** (**Delete_object**) permission on the PGO item itself.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The client program is not authorized to delete the specified item.

sec_rgy_object_not_found
The registry server could not find the specified item.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ).

NAME

sec_rgy_pgo_delete_member — Deletes a member of a group or organization

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_delete_member(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t go_name,
       sec_rgy_name_t person_name,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain
This variable identifies the type of the person, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_group
The *go_name* parameter identifies a group.

sec_rgy_domain_org
The *go_name* parameter identifies an organization.

go_name
A character string (type **sec_rgy_name_t**) containing the name of the group or organization from which the specified person will be deleted.

person_name
A character string (type **sec_rgy_name_t**) containing the name of the person to be deleted from the membership list of the group or organization specified by *go_name*.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_delete_member( )* routine deletes a member from the membership list of a group or organization. Any accounts in which the person holds the deleted group or organization membership are also deleted.

Permissions Required

The *sec_rgy_pgo_delete_member( )* routine requires the **M** (**Member_list**) permission on the group or organization item specified by *go_name*.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_bad_domain
An invalid domain was specified. Members can exist only for groups and organizations, not for persons.

sec_rgy_not_authorized
The client program is not authorized to delete the specified member.

sec_rgy_object_not_found
The specified group or organization was not found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_add_member*( ).

NAME

sec_rgy_pgo_get_by_eff_unix_num — Returns the name and data for a PGO item identified by its effective UNIX number

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_get_by_eff_unix_num(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t scope,
       signed32 unix_id,
       boolean32 allow_aliases,
       sec_rgy_cursor_t *item_cursor,
       sec_rgy_pgo_item_t *pgo_item,
       sec_rgy_name_t name,
       boolean32 *overridden,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The UNIX number identifies a principal.

sec_rgy_domain_group
The UNIX number identifies a group.

Note that this function does *not* support the value **sec_rgy_domain_org**.

scope

A character string (type **sec_rgy_name_t**) containing the scope of the desired search. The registry database is designed to accommodate a tree-structured name hierarchy. The scope of a search is the name of the branch under which the search takes place. For example, all names in a registry might start with **/alpha**, and be divided further into **/beta** or **/gamma**. To search only the part of the database under **/beta**, the scope of the search would be **/alpha/beta**, and any resulting PGO items would have names beginning with this string. Note that these naming conventions need not have anything to do with group or organization PGO item membership lists.

unix_id

The UNIX number of the desired registry PGO item.

allow_aliases

A **boolean32** value indicating whether to search for a primary PGO item, or whether the search can be satisfied with an alias. If TRUE, the routine returns the next entry found for the PGO item. If FALSE, the routine returns only the primary entry.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The *sec_rgy_pgo_get_next( )* routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use *sec_rgy_cursor_reset( )* to reset the cursor.

Output

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure to receive the data for the returned PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item, if a principal, may have a concurrent group set. The data is as it appears in the registry, for that UNIX number, even though some of the fields may have been overridden locally.

name
A pointer to a **sec_rgy_name_t** character string containing the returned name for the PGO item. This string might contain a local override value if the supplied UNIX number is found in the **passwd_override** or **group_override** file.

overridden
A pointer to a **boolean32** value indicating whether or not the supplied UNIX number has an entry in the local override file (**passwd_override** or **group_override**).

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_get_by_eff_unix_num( )* routine returns the name and data for a PGO item. The desired item is identified by its type (domain) and its UNIX number.

This routine is similar to the *sec_rgy_pgo_get_by_unix_num( )* routine. The difference between the routines is that *sec_rgy_pgo_get_by_eff_unix_num( )* first searches the local override files for the respective *name_domain* for a match with the supplied UNIX number. If an override match is found, and an account or group name is found in that entry, then that name is used to obtain PGO data from the registry and the value of the *overridden* parameter is set to TRUE.

The *item_cursor* parameter specifies the starting point for the search through the registry database. It provides an automatic place holder in the database. The routine automatically updates this variable to point to the next PGO item after the returned item. The returned cursor location can be supplied on a subsequent database access call that also uses a PGO item cursor.

Permissions Required

The *sec_rgy_pgo_get_by_eff_unix_num( )* routine requires the **r** (**read**) permission on the PGO item to be viewed.

CAUTIONS

There are several different types of cursors used in the registry Application Programmer Interface (API). Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to *sec_rgy_acct_get_projlist( )* and *sec_rgy_pgo_get_next( ).* The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use *sec_rgy_cursor_reset( )* to renew a cursor for use with another call or for another server.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

group_override
The local group override file.

passwd_override
The local password override file.

ERRORS

error_status_ok
The call was successful.

sec_rgy_no_more_entries
The cursor is at the end of the list of PGO items.

sec_rgy_object_not_found
The specified PGO item was not found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_get_by_id*( ), *sec_rgy_pgo_get_by_name*( ), *sec_rgy_pgo_get_by_unix_num*( ), *sec_rgy_pgo_get_next*( ), *sec_rgy_pgo_id_to_name*( ), *sec_rgy_pgo_id_to_unix_num*( ), *sec_rgy_pgo_name_to_id*( ), *sec_rgy_pgo_unix_num_to_id*( ), *sec_rgy_cursor_reset*( ).

NAME

sec_rgy_pgo_get_by_id — Returns the name and data for a PGO item identified by its UUID

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_get_by_id(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t scope,
       uuid_t *item_id,
       boolean32 allow_aliases,
       sec_rgy_cursor_t *item_cursor,
       sec_rgy_pgo_item_t *pgo_item,
       sec_rgy_name_t name,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The UUID identifies a principal.

sec_rgy_domain_group
The UUID identifies a group.

sec_rgy_domain_org
The UUID identifies an organization.

scope

A character string (type **sec_rgy_name_t**) containing the scope of the desired search. The registry database is designed to accommodate a tree-structured name hierarchy. The scope of a search is the name of the branch under which the search takes place. For example, all names in a registry might start with **/alpha**, and be divided further into **/beta** or **/gamma**. To search only the part of the database under **/beta**, the scope of the search would be **/alpha/beta**, and any resulting PGO items would have names beginning with this string. Note that these naming conventions need not have anything to do with group or organization PGO item membership lists.

item_id

A pointer to the **uuid_t** variable containing the UUID (Unique Universal Identifier) of the desired PGO item.

allow_aliases

A **boolean32** value indicating whether to search for a primary PGO item, or whether the search can be satisfied with an alias. If TRUE, the routine returns the next entry found for the PGO item. If FALSE, the routine returns only the primary entry.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The *sec_rgy_pgo_get_by_id( )* routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns **sec_rgy_no_more_entries** in the *status* parameter. Use *sec_rgy_cursor_reset( )* to reset the cursor.

Output

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure to receive the data for the returned PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item, if a principal, may have a concurrent group set.

name
A pointer to a **sec_rgy_name_t** character string containing the returned name for the PGO item.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_get_by_id( )* routine returns the name and data for a PGO item. The desired item is identified by its type (domain) and its UUID.

The *item_cursor* parameter specifies the starting point for the search through the registry database. It provides an automatic place holder in the database. The routine automatically updates this variable to point to the next PGO item after the returned item. The returned cursor location can be supplied on a subsequent database access call that also uses a PGO item cursor.

Permissions Required

The *sec_rgy_pgo_get_by_id( )* routine requires the **r** (**read**) permission on the PGO item to be viewed.

CAUTIONS

There are several different types of cursors used in the registry API. Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to *sec_rgy_acct_get_projlist( )* and *sec_rgy_pgo_get_next( )*. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use *sec_rgy_cursor_reset( )* to renew a cursor for use with another call or for another server.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_no_more_entries
The cursor is at the end of the list of PGO items.

sec_rgy_object_not_found
The specified PGO item was not found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_get_by_name*( ), *sec_rgy_pgo_get_by_unix_num*( ), *sec_rgy_pgo_get_next*( ), *sec_rgy_pgo_id_to_name*( ), *sec_rgy_pgo_id_to_unix_num*( ), *sec_rgy_pgo_name_to_id*( ), *sec_rgy_pgo_unix_num_to_id*( ), *sec_rgy_cursor_reset*( ).

NAME

sec_rgy_pgo_get_by_name — Returns the data for a named PGO item

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_get_by_name(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t pgo_name,
       sec_rgy_cursor_t *item_cursor,
       sec_rgy_pgo_item_t *pgo_item,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

pgo_name
A character string (type **sec_rgy_name_t**) containing the name of the principal, group, or organization to search for.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The *sec_rgy_pgo_get_by_name( )* routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use *sec_rgy_cursor_reset( )* to reset the cursor.

Output

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure to receive the data for the returned PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item, if a principal, may have a concurrent group set.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_get_by_name( )* routine returns the data for a named PGO item from the registry database. The desired item is identified by its type (*name_domain*) and name.

The *item_cursor* parameter specifies the starting point for the search through the registry database. It provides an automatic place holder in the database. The routine automatically updates this variable to point to the next PGO item after the returned item. The returned cursor location can be supplied on a subsequent database access call that also uses a PGO item cursor.

Permissions Required

The *sec_rgy_pgo_get_by_name( )* routine requires the **r** (**read**) permission on the PGO item to be viewed.

CAUTIONS

There are several different types of cursors used in the registry API. Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to *sec_rgy_acct_get_projlist( )* and *sec_rgy_pgo_get_next( )*. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use *sec_rgy_cursor_reset( )* to renew a cursor for use with another call or for another server.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_no_more_entries
The cursor is at the end of the list of PGO items.

sec_rgy_object_not_found
The specified PGO item was not found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_get_by_id*( ), *sec_rgy_pgo_get_by_unix_num*( ), *sec_rgy_pgo_get_next*( ), *sec_rgy_pgo_id_to_name*( ), *sec_rgy_pgo_id_to_unix_num*( ), *sec_rgy_pgo_name_to_id*( ), *sec_rgy_pgo_unix_num_to_id*( ), *sec_rgy_cursor_reset*( ).

NAME

sec_rgy_pgo_get_by_unix_num — Returns the name and data for a PGO item identified by its UNIX ID

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_get_by_unix_num(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t scope,
       signed32 unix_id,
       boolean32 allow_aliases,
       sec_rgy_cursor_t *item_cursor,
       sec_rgy_pgo_item_t *pgo_item,
       sec_rgy_name_t name,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The UNIX number identifies a principal.

sec_rgy_domain_group
The UNIX number identifies a group.

sec_rgy_domain_org
The UNIX number identifies an organization.

scope

A character string (type **sec_rgy_name_t**) containing the scope of the desired search. The registry database is designed to accommodate a tree-structured name hierarchy. The scope of a search is the name of the branch under which the search takes place. For example, all names in a registry might start with **/alpha**, and be divided further into **/beta** or **/gamma**. To search only the part of the database under **/beta**, the scope of the search would be **/alpha/beta**, and any resulting PGO items would have names beginning with this string. Note that these naming conventions need not have anything to do with group or organization PGO item membership lists.

unix_id

The UNIX number of the desired registry PGO item.

allow_aliases

A **boolean32** value indicating whether to search for a primary PGO item, or whether the search can be satisfied with an alias. If TRUE, the routine returns the next entry found for the PGO item. If FALSE, the routine returns only the primary entry.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The *sec_rgy_pgo_get_by_unix_num( )* routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use *sec_rgy_cursor_reset( )* to reset the cursor.

Output

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure to receive the data for the returned PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item, if a principal, may have a concurrent group set.

name
A pointer to a **sec_rgy_name_t** character string containing the returned name for the PGO item.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_get_by_unix_num( )* routine returns the name and data for a PGO item. The desired item is identified by its type (domain) and its UNIX number.

The *item_cursor* parameter specifies the starting point for the search through the registry database. It provides an automatic place holder in the database. The routine automatically updates this variable to point to the next PGO item after the returned item. The returned cursor location can be supplied on a subsequent database access call that also uses a PGO item cursor.

Permissions Required

The *sec_rgy_pgo_get_by_unix_num( )* routine requires the **r** (**read**) permission on the PGO item to be viewed.

CAUTIONS

There are several different types of cursors used in the registry API. Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to *sec_rgy_acct_get_projlist( )* and *sec_rgy_pgo_get_next( )*. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use *sec_rgy_cursor_reset( )* to renew a cursor for use with another call or for another server.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_no_more_entries
The cursor is at the end of the list of PGO items.

sec_rgy_object_not_found
The specified PGO item was not found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_get_by_id*( ), *sec_rgy_pgo_get_by_name*( ), *sec_rgy_pgo_get_next*( ), *sec_rgy_pgo_id_to_name*( ), *sec_rgy_pgo_id_to_unix_num*( ), *sec_rgy_pgo_name_to_id*( ), *sec_rgy_pgo_unix_num_to_id*( ), *sec_rgy_cursor_reset*( ).

NAME

sec_rgy_pgo_get_members — Returns the membership list for a specified group or organization or returns the set of groups in which the specified principal is a member

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_get_members(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t go_name,
       sec_rgy_cursor_t *member_cursor,
       signed32 max_members,
       sec_rgy_member_t member_list[],
       signed32 *number_supplied,
       signed32 *number_members,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a **secd** server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

This variable specifies whether *go_name* identifies a principal, group, or organization. The valid values are as follows:

sec_rgy_domain_group
The *go_name* parameter identifies a group.

sec_rgy_domain_org
The *go_name* parameter identifies an organization.

sec_rgy_domain_person
The *go_name* parameter identifies an principal.

go_name

A character string (type **sec_rgy_name_t**) that contains the name of a group, organization, or principal. If *go_name* is the name of a group or organization, the call returns the group's or organization's member list. If *go_name* is the name of a principal, the call returns a list of all groups in which the principal is a member. (Contrast this with the *sec_rgy_acct_get_proj( )* call, which returns only those groups in which the principal is a member and that have been marked to be included in the principal's project list.)

max_members

A **signed32** variable containing the allocated dimension of the *member_list*[ ] array. This is the maximum number of members or groups that can be returned by a single call.

Input/Output

member_cursor
An opaque pointer to a specific entry in the membership list or list of groups. The returned list begins with the entry specified by *member_cursor*. Upon return, the cursor points to the next entry after the last one returned. If there are no more entries, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use *sec_rgy_cursor_reset( )* to reset the cursor to the beginning of the list.

Output

member_list[ ]
An array of character strings to receive the returned member or group names. The size allocated for the array is given by *max_number*. If this value is less than the total number of members or group names, multiple calls must be made to return all of the members or groups.

number_supplied
A pointer to a **signed32** variable to receive the number of members or groups actually returned in *member_list*.

number_members
A pointer to a **signed32** variable to receive the total number of members or groups. If this number is greater than *number_supplied*, multiple calls to *sec_rgy_pgo_get_members( )* are necessary. Use the *member_cursor* parameter to coordinate successive calls.

status
A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_get_members( )* routine returns a list of the members in the specified group or organization, or a list of groups in which a specified principal is a member.

The *member_cursor* parameter specifies the starting point for the search through the registry database. It provides an automatic place holder in the database. The routine automatically updates *member_cursor* to point to the next member or group (if any) after the returned list. If not all of the members or groups are returned, the updated cursor can be supplied on successive calls to return the remainder of the list.

Permissions Required

The *sec_rgy_pgo_get_members( )* routine requires the **r** (**read**) permission on the group, organization, or principal object specified by *go_name*.

CAUTIONS

There are several different types of cursors used in the registry API. Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to *sec_rgy_acct_get_projlist( )* and *sec_rgy_pgo_get_next( )*. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use *sec_rgy_cursor_reset( )* to renew a cursor for use with another call or for another server.

RETURN VALUES

The routine returns:

- The names of the groups or members in *member_list*
- The number of members or groups returned by the call in *number_supplied*
- The total number of members in the group or organization, or the total number of groups in which the principal is a member in *number_members*

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_no_more_entries
The cursor points to the end of the membership list for a group or organization or to the end of the list of groups for a principal.

sec_rgy_object_not_found
The specified group, organization, or principal could not be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add_member*( ), *sec_rgy_cursor_reset*( ), *sec_rgy_pgo_is_member*( ), *sec_rgy_acct_get_proj*( ).

NAME

sec_rgy_pgo_get_next — Returns the next PGO item in the registry database

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_get_next(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t scope,
       sec_rgy_cursor_t *item_cursor,
       sec_rgy_pgo_item_t *pgo_item,
       sec_rgy_name_t name,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
Returns the next principal item.

sec_rgy_domain_group
Returns the next group item.

sec_rgy_domain_org
Returns the next organization item.

scope

A character string (type **sec_rgy_name_t**) containing the scope of the desired search. The registry database is designed to accommodate a tree-structured name hierarchy. The scope of a search is the name of the branch under which the search takes place. For example, all names in a registry might start with **/alpha**, and be divided further into **/beta** or **/gamma**. To search only the part of the database under **/beta**, the scope of the search would be **/alpha/beta**, and any resulting PGO items would have names beginning with this string. Note that these naming conventions need not have anything to do with group or organization PGO item membership lists.

Input/Output

item_cursor

An opaque pointer indicating a specific PGO item entry in the registry database. The *sec_rgy_pgo_get_next( )* routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns the value **sec_rgy_no_more_entries** in the *status* parameter. Use *sec_rgy_cursor_reset( )* to reset the cursor.

Output

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure to receive the data for the returned PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item, if a principal, may have a concurrent group set.

name
A pointer to a **sec_rgy_name_t** character string containing the name of the returned PGO item.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_get_next( )* routine returns the data and name for the PGO in the registry database indicated by *item_cursor*. It also advances the cursor to point to the next PGO item in the database. Successive calls to this routine return all the PGO items in the database of the specified type (given by *name_domain*), in storage order.

The PGO data consists of the following:

- The Universal Unique Identifier (UUID) of the PGO item.
- The UNIX number for the PGO item.
- The quota for subaccounts.
- The full name of the PGO item.
- Flags indicating whether
  - A principal item is an alias.
  - The PGO item can be deleted.
  - A principal item can have a concurrent group set.
  - A group item can appear on a concurrent group set.

Permissions Required

The *sec_rgy_pgo_get_next( )* routine requires the **r** (**read**) permission on the PGO item to be viewed.

CAUTIONS

There are several different types of cursors used in the registry API. Some cursors point to PGO items, others point to members in a membership list, and others point to account data. Do not use a cursor for one sort of object in a call expecting another sort of object. For example, you cannot use the same cursor on a call to *sec_rgy_acct_get_projlist( )* and *sec_rgy_pgo_get_next( )*. The behavior in this case is undefined.

Furthermore, cursors are specific to a server. A cursor pointing into one replica of the registry database is useless as a pointer into another replica.

Use *sec_rgy_cursor_reset( )* to renew a cursor for use with another call or for another server.

RETURN VALUES

The routine returns the data for the returned PGO item in *pgo_item* and the name in *name*.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_no_more_entries
The cursor is at the end of the list of PGO items.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_cursor_reset*( ), *sec_rgy_pgo_get_by_id*( ), *sec_rgy_pgo_get_by_name*( ), *sec_rgy_pgo_get_by_unix_num*( ), *sec_rgy_pgo_id_to_unix_num*( ), *sec_rgy_pgo_unix_num_to_id*( ).

NAME

sec_rgy_pgo_id_to_name — Returns the name for a PGO item identified by its UUID

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_id_to_name(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       uuid_t *item_id,
       sec_rgy_name_t pgo_name,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The *item_id* parameter identifies a principal.

sec_rgy_domain_group
The *item_id* parameter identifies a group.

sec_rgy_domain_org
The *item_id* parameter identifies an organization.

item_id

A pointer to the **uuid_t** variable containing the input UUID (Unique Universal Identifier).

Output

pgo_name

A character string (type **sec_rgy_name_t**) containing the name of the principal, group, or organization with the input UUID.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_id_to_name( )* routine returns the name of the PGO item having the specified UUID.

Permissions Required

The *sec_rgy_pgo_id_to_name( )* routine requires at least one permission of any kind on the PGO item to be viewed.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_object_not_found
No item with the specified UUID could be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_get_by_id*( ), *sec_rgy_pgo_get_by_name*( ), *sec_rgy_pgo_get_by_unix_num*( ), *sec_rgy_pgo_id_to_unix_num*( ), *sec_rgy_pgo_name_to_id*( ), *sec_rgy_pgo_unix_num_to_id*( ).

NAME

sec_rgy_pgo_id_to_unix_num — Returns the UNIX number for a PGO item identified by its UUID

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_id_to_unix_num(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       uuid_t *item_id,
       signed32 *item_unix_id,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The *item_id* parameter identifies a principal.

sec_rgy_domain_group
The *item_id* parameter identifies a group.

sec_rgy_domain_org
The *item_id* parameter identifies an organization.

item_id
A pointer to the **uuid_t** variable containing the input UUID (Unique Universal Identifier).

Output

item_unix_id
A pointer to the **signed32** variable to receive the returned UNIX number for the PGO item.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_id_to_unix_num( )* routine returns the UNIX number for the PGO item having the specified UUID.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_object_not_found
No item with the specified UUID could be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_get_by_id*( ), *sec_rgy_pgo_get_by_name*( ), *sec_rgy_pgo_get_by_unix_num*( ), *sec_rgy_pgo_id_to_name*( ), *sec_rgy_pgo_name_to_id*( ), *sec_rgy_pgo_unix_num_to_id*( ).

NAME

sec_rgy_pgo_is_member — Checks group or organization membership

SYNOPSIS

```
#include <dce/pgo.h>

boolean32 sec_rgy_pgo_is_member(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t go_name,
       sec_rgy_name_t person_name,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_group

The *go_name* parameter identifies a group.

sec_rgy_domain_org

The *go_name* parameter identifies an organization.

go_name

A character string (type **sec_rgy_name_t**) containing the name of the group or organization whose membership list is in question.

person_name

A character string (type **sec_rgy_name_t**) containing the name of the principal whose membership in the group or organization specified by *go_name* is in question.

Output

status

A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_is_member( )* routine tests whether the specified principal is a member of the named group or organization.

Permissions Required

The *sec_rgy_pgo_is_member( )* routine requires the **t** (**test**) permission on the group or organization item specified by *go_name*.

RETURN VALUES

The routine returns TRUE if the principal is a member of the named group or organization. If the principal is not a member, the routine returns FALSE.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_object_not_found
The named group or organization was not found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add_member*( ), *sec_rgy_pgo_get_members*( ).

NAME

sec_rgy_pgo_name_to_id — Returns the UUID for a named PGO item

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_name_to_id(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t pgo_name,
       uuid_t *item_id,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

pgo_name

A character string (type **sec_rgy_name_t**) containing the name of the principal, group, or organization whose UUID is desired.

Output

item_id

A pointer to the **uuid_t** variable containing the UUID (Unique Universal Identifier) of the resulting PGO item.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_name_to_id( )* routine returns the UUID associated with the named PGO item.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_object_not_found
The specified PGO item could not be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_get_by_id*( ), *sec_rgy_pgo_get_by_name*( ), *sec_rgy_pgo_get_by_unix_num*( ), *sec_rgy_pgo_id_to_name*( ), *sec_rgy_pgo_id_to_unix_num*( ), *sec_rgy_pgo_unix_num_to_id*( ).

NAME

sec_rgy_pgo_name_to_unix_num — Returns the UNIX number for a PGO item identified by its name

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_name_to_unix_num(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t pgo_name,
       signed32 *item_unix_id,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

pgo_name
A character string containing the name of the PGO item in question.

Output

item_unix_id
A pointer to the **signed32** variable to receive the returned UNIX number for the PGO item.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_name_to_unix_num( )* routine returns the UNIX number for the PGO item having the specified name.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_object_not_found
No item with the specified UUID could be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_get_by_id*( ), *sec_rgy_pgo_get_by_name*( ), *sec_rgy_pgo_get_by_unix_num*( ), *sec_rgy_pgo_id_to_name*( ), *sec_rgy_pgo_name_to_id*( ), *sec_rgy_pgo_unix_num_to_id*( ).

NAME

sec_rgy_pgo_rename — Changes the name of a PGO item in the registry database

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_rename(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t old_name,
       sec_rgy_name_t new_name,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain

This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person

The name identifies a principal.

sec_rgy_domain_group

The name identifies a group.

sec_rgy_domain_org

The name identifies an organization.

old_name

A pointer to a **sec_rgy_name_t** character string containing the existing name of the PGO item.

new_name

A pointer to a **sec_rgy_name_t** character string containing the new name for the PGO item.

Output

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_rename( )* routine renames a PGO item in the registry database.

Permissions Required

If the *sec_rgy_pgo_rename( )* routine is performing a rename within a directory, it requires the **n** (**name**) permission on the old name of the PGO item. If the routine is performing a move between directories, it requires the following permissions:

- The **d** (**delete**) permission on the parent directory that contains the PGO item.
- The **n** (**name**) permission on the old name of the PGO item.
- The **i** (**insert**) permission on the parent directory in which the PGO item is to be added under the new name.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The client program is not authorized to change the name of the specified PGO item.

sec_rgy_object_not_found
The registry server could not find the specified PGO item.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_replace*( ).

NAME

sec_rgy_pgo_replace — Replaces the data in an existing PGO item

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_replace(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       sec_rgy_name_t pgo_name,
       sec_rgy_pgo_item_t *pgo_item,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The name identifies a principal.

sec_rgy_domain_group
The name identifies a group.

sec_rgy_domain_org
The name identifies an organization.

pgo_name
A character string (type **sec_rgy_name_t**) containing the name of the principal, group, or organization whose data is to be replaced.

pgo_item
A pointer to a **sec_rgy_pgo_item_t** structure containing the new data for the PGO item. The data in this structure includes the PGO item's name, UUID, UNIX number (if any), and administrative data, such as whether the item, if a principal, may have a concurrent group set.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_replace( )* routine replaces the data associated with a PGO item in the registry database.

The UNIX ID and UUID of a PGO item cannot be replaced. To change the UNIX ID or UUID, the existing PGO item must be deleted and a new PGO item added in its place. The one exception to this rule is that the UNIX ID can be replaced in the PGO item for a cell principal. The reason for this exception is that the UUID for a cell principal does not contain an embedded UNIX ID.

Permissions Required

The *sec_rgy_pgo_replace( )* routine requires at least one of the following permissions:

- The **m** (**mgmt_info**) permission on the PGO item, if **quota** or **flags** is being set.
- The **f** (**fullname**) permission on the PGO item, if **fullname** is being set.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The client program is not authorized to replace the specified PGO item.

sec_rgy_object_not_found
No PGO item was found with the given name.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

sec_rgy_unix_id_changed
The UNIX number of the PGO item was changed.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_delete*( ), *sec_rgy_pgo_rename*( ).

NAME

sec_rgy_pgo_unix_num_to_id — Returns the UUID for a PGO item identified by its UNIX number

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_unix_num_to_id(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       signed32 item_unix_id,
       uuid_t *item_id,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain
This variable identifies the type of the principal, group, or organization (PGO) item identified by the given name. The valid values are as follows:

sec_rgy_domain_person
The *item_unix_id* parameter identifies a principal.

sec_rgy_domain_group
The *item_unix_id* parameter identifies a group.

sec_rgy_domain_org
The *item_unix_id* parameter identifies an organization.

item_unix_id
The **signed32** variable containing the UNIX number for the PGO item.

Output

item_id
A pointer to the **uuid_t** variable containing the UUID (Unique Universal Identifier) of the resulting PGO item.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_unix_num_to_id( )* routine returns the Universal Unique Identifier (UUID) for a PGO item that has the specified UNIX number.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

sec_rgy_object_not_found
No item with the specified UNIX number could be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

error_status_ok
The call was successful.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_get_by_id*( ), *sec_rgy_pgo_get_by_name*( ), *sec_rgy_pgo_get_by_unix_num*( ), *sec_rgy_pgo_id_to_name*( ), *sec_rgy_pgo_id_to_unix_num*( ), *sec_rgy_pgo_name_to_id*( ).

NAME

sec_rgy_pgo_unix_num_to_name — Returns the name for a PGO item identified by its UNIX number

SYNOPSIS

```
#include <dce/pgo.h>

void sec_rgy_pgo_unix_num_to_name(
       sec_rgy_handle_t context,
       sec_rgy_domain_t name_domain,
       signed32 item_unix_id,
       sec_rgy_name_t pgo_name,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name_domain
The type of the principal, group, or organization (PGO) item identified by *item_unix_id*. Valid values are as follows:

sec_rgy_domain_person
The *item_unix_id* parameter identifies a principal.

sec_rgy_domain_group
The *item_unix_id* parameter identifies a group.

sec_rgy_domain_org
The *item_unix_id* parameter identifies an organization.

item_unix_id
The **signed32** variable containing the UNIX number for the PGO item.

Output

pgo_name
A character string containing the name of the PGO item in question.

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_pgo_unix_num_to_name( )* routine returns the name for a PGO item that has the specified UNIX number.

Permissions Required

The *sec_rgy_pgo_unix_num_to_name( )* routine requires at least one permission of any kind on the PGO item identified by *item_unix_id*.

FILES

/usr/include/dce/pgo.idl
The **idl** file from which **dce/pgo.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_object_not_found
No item with the specified UNIX number could be found.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_pgo_add*( ), *sec_rgy_pgo_get_by_id*( ), *sec_rgy_pgo_get_by_name*( ), *sec_rgy_pgo_get_by_unix_num*( ), *sec_rgy_pgo_id_to_name*( ), *sec_rgy_pgo_id_to_unix_num*( ), *sec_rgy_pgo_name_to_id*( ).

NAME

sec_rgy_plcy_get_effective — Returns the effective policy for an organization

SYNOPSIS

```
#include <dce/policy.h>

void sec_rgy_plcy_get_effective(
       sec_rgy_handle_t context,
       sec_rgy_name_t organization,
       sec_rgy_plcy_t *policy_data,
       or_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

organization

A character string (type **sec_rgy_name_t**) containing the name of the organization for which the policy data is to be returned. If this string is empty, the routine returns the registry's policy data.

Output

policy_data

A pointer to the **sec_rgy_plcy_t** structure to receive the authentication policy. This structure contains the minimum length of a user's password, the lifetime of a password, the expiration date of a password, the lifetime of the entire account, and some flags describing limitations on the password spelling.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_plcy_get_effective( )* routine returns the effective policy for the specified organization.

The effective policy data is the most restrictive combination of the registry and the organization policies.

The policy data consists of the following:

- The password expiration date. This is the date on which account passwords will expire.
- The minimum length allowed for account passwords.
- The period of time (life span) for which account passwords will be valid.
- The period of time (life span) for which accounts will be valid.
- Flags indicating whether account passwords can consist entirely of spaces or entirely of alphanumeric characters.

Permissions Required

The *sec_rgy_plcy_get_effective( )* routine requires the **r** (**read**) permission on the policy object from which the data is to be returned. If an organization is specified, the routine also requires the **r** (**read**) permission on the organization.

NOTES

If no organization is specified, the routine returns the registry's policy data. To return the effective policy, an organization must be specified. This is because the routine compares the registry's policy data with that of the organization to determine which is more restrictive.

FILES

/usr/include/dce/policy.idl
The **idl** file from which **dce/policy.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_object_not_found
The registry server could not find the specified organization.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_plcy_get_info*( ), *sec_rgy_plcy_set_info*( ).

NAME

sec_rgy_plcy_get_info — Returns the policy for an organization

SYNOPSIS

```
#include <dce/policy.h>

void sec_rgy_plcy_get_info(
       sec_rgy_handle_t context,
       sec_rgy_name_t organization,
       sec_rgy_plcy_t *policy_data,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

organization

A character string (type **sec_rgy_name_t**) containing the name of the organization for which the policy data is to be returned. If this string is empty, the routine returns the registry's policy data.

Output

policy_data

A pointer to the **sec_rgy_plcy_t** structure to receive the authentication policy. This structure contains the minimum length of a user's password, the lifetime of a password, the expiration date of a password, the lifetime of the entire account, and some flags describing limitations on the password spelling.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_plcy_get_info( )* routine returns the policy data for the specified organization. If no organization is specified, the registry's policy data is returned.

The policy data consists of the following:

- The password expiration date. This is the date on which account passwords will expire.
- The minimum length allowed for account passwords.
- The period of time (life span) for which account passwords will be valid.
- The period of time (life span) for which accounts will be valid.
- Flags indicating whether account passwords can consist entirely of spaces or entirely of alphanumeric characters.

Permissions Required

The *sec_rgy_plcy_get_info( )* routine requires the **r** (**read**) permission on the policy object or organization from which the data is to be returned.

NOTES

The returned policy may not be in effect if the overriding registry authorization policy is more restrictive. (See the *sec_rgy_auth_plcy_get_effective( )* routine.)

FILES

/usr/include/dce/policy.idl
The **idl** file from which **dce/policy.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_object_not_found
The registry server could not find the specified organization.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_plcy_get_effective_info*( ), *sec_rgy_plcy_set_info*( ).

NAME

sec_rgy_plcy_set_info — Sets the policy for an organization

SYNOPSIS

```
#include <dce/policy.h>

void sec_rgy_plcy_set_info(
       sec_rgy_handle_t context,
       sec_rgy_name_t organization,
       sec_rgy_plcy_t *policy_data,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

organization

A character string (type **sec_rgy_name_t**) containing the name of the organization for which the policy data is to be returned. If this string is empty, the routine sets the registry's policy data.

policy_data

A pointer to the **sec_rgy_plcy_t** structure containing the authentication policy. This structure contains the minimum length of a user's password, the lifetime of a password, the expiration date of a password, the lifetime of the entire account, and some flags describing limitations on the password spelling.

Output

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_plcy_set_info( )* routine sets the authentication policy for a specified organization. If no organization is specified, the registry's policy data is set.

Policy data can be returned or set for individual organizations and for the registry as a whole.

Permissions Required

The *sec_rgy_plcy_set_info( )* routine requires the **m** (**mgmt_info**) permission on the policy object or organization for which the data is to be set.

NOTES

The policy set on an account may be less restrictive than the policy set for the registry as a whole. In this case, the changes in policy have no effect, since the effective policy is the most restrictive combination of the organization and registry authentication policies. (See the *sec_rgy_auth_plcy_get_effective( )* routine.)

FILES

/usr/include/dce/policy.idl

The **idl** file from which **dce/policy.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The user is not authorized to perform this operation.

sec_rgy_object_not_found
The registry server could not find the specified organization.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_plcy_get_effective*( ), *sec_rgy_plcy_get_info*( ).

NAME

sec_rgy_properties_get_info — Returns registry properties

SYNOPSIS

```
#include <dce/policy.h>

void sec_rgy_properties_get_info(
       sec_rgy_handle_t context,
       sec_rgy_properties_t *properties,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

Output

properties

A pointer to a **sec_rgy_properties_t** structure to receive the returned property information. A registry's property information contains information such as the default and minimum lifetime and other restrictions on privilege attribute certificates, the realm authentication name, and whether or not this replica of the registry supports updates.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_properties_get_info( )* routine returns a list of the registry properties.

The property information consists of the following:

read_version

A stamp specifying the earliest version of the registry server software that can read from this registry.

write_version

A stamp specifying the earliest version of the registry server software that can write to this registry.

minimum_ticket_lifetime

The minimum period of time for which an authentication ticket remains valid.

default_certificate_lifetime

The default period of time for which an authentication certificate (ticket-granting ticket) remains valid. A process can request an authentication certificate with a longer lifetime. Note that the maximum lifetime for an authentication certificate cannot exceed the lifetime established by the effective policy for the requesting account.

low_unix_id_person

The lowest UNIX ID that can be assigned to a principal in the registry.

low_unix_id_group

The lowest UNIX ID that can be assigned to a group in the registry.

low_unix_id_org
The lowest UNIX ID that can be assigned to an organization in the registry.

max_unix_id
The maximum UNIX ID that can be used for any item in the registry.

realm
A character string naming the cell controlled by this registry.

realm_uuid
The UUID of the cell controlled by this registry.

flags
Flags indicating whether:

sec_rgy_prop_readonly
If TRUE, the registry database is read-only.

sec_rgy_prop_auth_cert_unbound
If TRUE, privilege attribute certificates can be generated for use at any site.

sec_rgy_prop_shadow_passwd
If FALSE, passwords can be distributed over the network. If this flag is TRUE, passwords will be stripped from the returned data to the *sec_rgy_acct_lookup( )*, and other calls that return an account's encoded password.

sec_rgy_prop_embedded_unix_id
All registry UUIDs contain embedded UNIX IDs. This implies that the UNIX ID of any registry object cannot be changed, since UUIDs are immutable.

Permissions Required

The *sec_rgy_properties_get_info( )* routine requires the **r** (**read**) permission on the policy object from which the property information is to be returned.

FILES

/usr/include/dce/policy.idl
The **idl** file from which **dce/policy.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_properties_set_info*( ).

NAME

sec_rgy_properties_set_info — Sets registry properties

SYNOPSIS

```
#include <dce/policy.h>

void sec_rgy_properties_set_info(
       sec_rgy_handle_t context,
       sec_rgy_properties_t *properties,
       error_status_t *status);
```

PARAMETERS

Input

context

The registry server handle. An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

properties

A pointer to a **sec_rgy_properties_t** structure containing the registry property information to be set. A registry's property information contains information such as the default and minimum lifetime and other restrictions on privilege attribute certificates, the realm authentication name, and whether or not this replica of the registry supports updates.

Output

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_properties_set_info( )* routine sets the registry properties.

The property information consists of the following:

read_version

A stamp specifying the earliest version of the registry server software that can read from this registry.

write_version

A stamp specifying the earliest version of the registry server software that can write to this registry.

minimum_ticket_lifetime

The minimum period of time for which an authentication ticket remains valid.

default_certificate_lifetime

The default period of time for which an authentication certificate (ticket-granting ticket) remains valid. A process can request an authentication certificate with a longer lifetime. Note that the maximum lifetime for an authentication certificate cannot exceed the lifetime established by the effective policy for the requesting account.

low_unix_id_person

The lowest UNIX ID that can be assigned to a principal in the registry.

low_unix_id_group

The lowest UNIX ID that can be assigned to a group in the registry.

low_unix_id_org
The lowest UNIX ID that can be assigned to an organization in the registry.

max_unix_id
The maximum UNIX ID that can be used for any item in the registry.

realm
A character string naming the cell controlled by this registry.

realm_uuid
The UUID of the cell controlled by this registry.

flags
Flags indicating whether:

sec_rgy_prop_readonly
If TRUE, the registry database is read-only.

sec_rgy_prop_auth_cert_unbound
If TRUE, privilege attribute certificates can be generated for use at any site.

sec_rgy_prop_shadow_passwd
If FALSE, passwords can be distributed over the network. If this flag is TRUE, passwords will be stripped from the returned data to the *sec_rgy_acct_lookup( )*, and other calls that return an account's encoded password.

sec_rgy_prop_embedded_unix_id
All registry UUIDs contain embedded UNIX IDs. This implies that the UNIX ID of any registry object cannot be changed, since UUIDs are immutable.

Permissions Required

The *sec_rgy_properties_set_info( )* routine requires the **m** (**mgmt_info**) permission on the policy object for which the property information is to be set.

FILES

/usr/include/dce/policy.idl
The **idl** file from which **dce/policy.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_not_authorized
The user is not authorized to change the registry properties.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_properties_get_info*( ).

NAME

sec_rgy_site_bind — Binds to a registry site

SYNOPSIS

```
#include <dce/binding.h>

void sec_rgy_site_bind(
       unsigned_char_t *site_name,
       sec_rgy_bind_auth_info_t *auth_info,
       sec_rgy_handle_t *context,
       error_status_t *status);
```

PARAMETERS

Input

site_name

A character string (type **unsigned_char_t**) containing the name of the registry site to bind to. Supply this name in any of the following forms:

- To randomly choose a site to bind to in the named cell, specify a cell name (for example, **/.../r_d.com** or **/.:** for the local cell)
- To bind to a specific site in a specific cell, specify either the site's global name (for example, **/.../r_d.com/subsys/dce/sec/rs_server_250_2**) or the site's network address (for example, **ncadg_ip_udp:15.22.144.248**)

auth_info

A pointer to the **sec_rgy_bind_auth_info_t** structure that identifies the authentication protocol, protection level, and authorization protocol to use in establishing the binding. (See the *rpc_binding_set_auth_info( )* routine). If the **sec_rgy_bind_auth_info_t** structure specifies authenticated RPC, the caller must have established a valid network identity for this call to succeed.

Output

context

A pointer to a **sec_rgy_handle_t** variable. Upon return, this contains a registry server handle indicating (bound to) the desired registry site.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_site_bind( )* call binds to a registry site at the security level specified by the *auth_info* parameter. The *site_name* parameter identifies the registry to use. If *site_name* is **NULL**, or a zero-length string, a registry site in the local cell is selected by the client agent.

NOTES

This routine binds arbitrarily to either an update or query site. Although update sites can accept queries, query sites cannot accept updates. To specifically select an update site, use *sec_rgy_site_bind_update( )*.

FILES

/usr/include/dce/binding.idl

The **idl** file from which **dce/binding.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_login_s_no_current_context
The caller does not have a valid network login context.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_site_open*( ), *sec_rgy_cell_bind*( ).

NAME

sec_rgy_site_bind_update — Binds to a registry update site

SYNOPSIS

```
#include <dce/binding.h>

void sec_rgy_site_bind_update(
       unsigned_char_t *site_name,
       sec_rgy_bind_auth_info_t *auth_info,
       sec_rgy_handle_t *context,
       error_status_t *status);
```

PARAMETERS

Input

site_name

A character string (type **unsigned_char_t**) containing the name of the registry site to bind to. Supply this name in any of the following forms:

- To choose the update site to bind to in the named cell, specify a cell name (for example, **/.../r_d.com** or **/.:** for the local cell)
- To start the search for the update site at a specific replica in the replica's cell, specify either the replica's global name (for example, **/.../r_d.com/subsys/dce/sec/rs_server_250_2**) or the replica's network address (for example, **ncadg_ip_udp:15.22.144.248**)

auth_info

A pointer to the **sec_rgy_bind_auth_info_t** structure that identifies the authentication protocol, protection level, and authorization protocol to use in establishing the binding. (See the *rpc_binding_set_auth_info( )* routine). If the **sec_rgy_bind_auth_info_t** structure specifies authenticated RPC, the caller must have established a valid network identity for this call to succeed.

Output

context

A pointer to a **sec_rgy_handle_t** variable. Upon return, this contains a registry server handle indicating (bound to) the desired registry site.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_site_bind_update( )* routine binds to a registry update site. A registry update site is a master server that may control several satellite (query) servers. To change the registry database, it is necessary to change a registry update site, which then automatically updates its associated query sites. No changes can be made directly to a registry query database.

The *site_name* parameter identifies either the cell in which to find the update site or the replica at which to start the search for the update site. If *site_name* is **NULL**, or a zero-length string, an update site in the local cell is selected by the client agent.

The handle for the associated registry server is returned in *context*. The handle is to an update site. Use this registry context handle in subsequent calls that update or query the the registry database (for example, the *sec_rgy_pgo_add( )* or *sec_rgy_acct_lookup( )* call).

FILES

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_login_s_no_current_context
The caller does not have a valid network login context.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_site_open*( ), *sec_rgy_site_bind*( ), *sec_rgy_site_open*( ).

NAME

sec_rgy_site_binding_get_info — Returns information from the registry binding handle

SYNOPSIS

```
#include <dce/binding.h>

void sec_rgy_site_binding_get_info(
       sec_rgy_handle_t context,
       unsigned_char_t **cell_name,
       unsigned_char_t **server_name,
       unsigned_char_t **string_binding,
       sec_rgy_bind_auth_info_t *auth_info,
       error_status_t *status);
```

PARAMETERS

Input

context
A **sec_rgy_handle_t** variable that contains a registry server handle indicating (bound to) the desired registry site. To obtain information on the default binding handle, initialize *context* to **sec_rgy_default_handle**. A valid login context must be set for the process if *context* is set to **sec_rgy_default_handle**; otherwise the error **sec_under_login_s_no_current_context** is returned.

Output

cell_name
The name of the home cell for this registry.

server_name
The name of the node on which the server is resident. This name is either a global name or a network address, depending on the form in which the name was input to the call that bound to the site.

string_binding
A string containing binding information from **sec_rgy_handle_t**.

auth_info
A pointer to the **sec_rgy_bind_auth_info_t** structure that identifies the authentication protocol, protection level, and authorization protocol to use in establishing the binding. (See the *rpc_binding_set_auth_info( )* routine).

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_site_binding_get_info( )* routine returns the site name and authentication information associated with the *context* parameter. If the context is the default context, the information for the default binding is returned. Passing in a **NULL** value for any of the output values (except for *status*) will prevent that value from being returned. Memory is allocated for the string returned in the *cell_name*, *server_name*, and *string_binding* parameters. The application calls the *rpc_string_free( )* routine to deallocate that memory.

FILES

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_login_s_no_current_context
No currently established network identity for which context exists.

sec_rgy_server_unavailable
Server unavailable.

SEE ALSO

Functions: *sec_rgy_site_open*( ), *sec_rgy_site_bind*( ).

NAME

sec_rgy_site_close — Frees the binding handle for a registry server

SYNOPSIS

```
#include <dce/binding.h>

void sec_rgy_site_close(
       sec_rgy_handle_t context,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle indicating (bound to) a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

Output

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_site_close( )* routine frees the memory occupied by the specified handle and destroys its binding with the registry server.

NOTES

A handle cannot be used after it is freed.

FILES

/usr/include/dce/binding.idl

The **idl** file from which **dce/binding.h** was derived.

ERRORS

error_status_ok

The call was successful.

SEE ALSO

Functions: *sec_rgy_site_get*( ), *sec_rgy_site_is_readonly*( ), *sec_rgy_site_open*( ), *sec_rgy_site_open_query*( ), *sec_rgy_site_open_update*( ).

NAME

sec_rgy_site_get — Returns the string representation for a bound registry site

SYNOPSIS

```
#include <dce/binding.h>

void sec_rgy_site_get(
       sec_rgy_handle_t context,
       unsigned_char_t **site_name,
       error_status_t *status);
```

PARAMETERS

Input

context

An opaque handle indicating (bound to) a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle. To obtain information on the default binding handle, initialize *context* to **sec_rgy_default_handle**. A valid login context must be set for the process if *context* is set to **sec_rgy_default_handle**; otherwise the error **sec_login_s_no_current_context** is returned.

Output

site_name

A pointer to a character string (type **unsigned_char_t**) containing the returned name of the registry site associated with *context*, the given registry server handle.

The name is either a global name or a network address, depending on the form in which the name was input to the call that bound to the site.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_site_get( )* routine returns the name of the registry site associated with the specified handle. If the handle is the default context, the routine returns the name of the default context's site. Memory is allocated for the string returned in the *site_name* parameter. The application calls the *rpc_string_free( )* routine to deallocate that memory.

NOTES

To obtain binding information, the use of the *sec_rgy_site_binding_get_info( )* call is recommended in place of this call.

FILES

/usr/include/dce/binding.idl

The **idl** file from which **dce/binding.h** was derived.

ERRORS

error_status_ok

The call was successful.

sec_login_s_no_current_context

No currently established network identity for which context exists.

sec_rgy_server_unavailable

Server unavailable.

SEE ALSO

Functions: *sec_rgy_site_open*( ).

NAME

sec_rgy_site_is_readonly — Checks whether a registry site is read-only

SYNOPSIS

```
#include <dce/binding.h>

boolean32 sec_rgy_site_is_readonly(
       sec_rgy_handle_t context);
```

PARAMETERS

Input

context
An opaque handle indicating (bound to) a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

DESCRIPTION

The *sec_rgy_site_is_readonly( )* routine checks whether the registry site associated with the specified handle is a query site or an update site. A query site is a read-only replica of a master registry database. The update site accepts changes to the registry database, and duplicates the changes in its associated query sites.

RETURN VALUES

The routine returns:

- TRUE if the registry site is read-only or if there was an error using the specified handle
- FALSE if the registry site is an update site

FILES

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

SEE ALSO

Functions: *sec_rgy_site_open*( ), *sec_rgy_site_open_query*( ).

NAME

sec_rgy_site_open — Binds to a registry site

SYNOPSIS

```
#include <dce/binding.h>

void sec_rgy_site_open(
       unsigned_char_t *site_name,
       sec_rgy_handle_t *context,
       error_status_t *status);
```

PARAMETERS

Input

site_name

A pointer to a character string (type **unsigned_char_t**) containing the name of the registry site to bind to. Supply this name in any of the following forms:

- To randomly choose a site to bind to in the named cell, specify a cell name (for example, **/.../r_d.com** or **/.:** for the local cell)
- To bind to a specific site in a specific cell, specify either the site's global name (for example, **/.../r_d.com/subsys/dce/sec/rs_server_250_2**) or the site's network address (for example, **ncadg_ip_udp**:**15.22.144.248**)

Note that if you specify the name of a specific **secd** to bind to and the name is not valid, the call will bind to a random site in the cell if the specified cell name is valid.

Output

context

A pointer to a **sec_rgy_handle_t** variable. Upon return, this contains a registry server handle indicating (bound to) the desired registry site.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_site_open( )* routine binds to a registry site at the level of security specified in the *rpc_binding_set_auth_info( )* call. The *site_name* parameter identifies the registry to use. If *site_name* is **NULL**, or a zero-length string, a registry site in the local cell is selected by the client agent. The caller must have established a valid network identity for this call to succeed.

NOTES

To bind to a registry site, the use of the *sec_rgy_site_bind( )* call is recommended in place of this call.

Like *sec_rgy_site_open_query( )* routine, this routine binds arbitrarily to either an update or query site. Although update sites can accept queries, query sites cannot accept updates. To specifically select an update site, use *sec_rgy_site_open_update( ).*

FILES

/usr/include/dce/binding.idl

The **idl** file from which **dce/binding.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_login_s_no_current_context
No currently established network identity for which context exists.

sec_rgy_server_unavailable
Server unavailable.

SEE ALSO

Functions: *sec_rgy_site_close*( ), *sec_rgy_site_is_readonly*( ), *sec_rgy_site_open_query*( ), *sec_rgy_site_open_update*( ).

NAME

sec_rgy_site_open_query — Binds to a registry query site

SYNOPSIS

```
#include <dce/binding.h>

void sec_rgy_site_open_query(
       unsigned_char_t *site_name,
       sec_rgy_handle_t *context,
       error_status_t *status);
```

PARAMETERS

Input

site_name

A character string (type **unsigned_char_t**) containing the name of the registry query site to bind to. Supply this name in any of the following forms:

- To randomly choose a site to bind to in the named cell, specify a cell name (for example, **/.../r_d.com** or **/.:** for the local cell)
- To bind to a specific site in a specific cell, specify either the site's global name (for example, **/.../r_d.com/subsys/dce/sec/rs_server_250_2**) or the site's network address (for example, **ncadg_ip_udp:15.22.144.248**)

Output

context

A pointer to a **sec_rgy_handle_t** variable. Upon return, this contains a registry server handle indicating (bound to) the desired registry site.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_site_open_query( )* routine binds to a registry query site. A registry query site is a satellite server that operates on a periodically updated copy of the main registry database. To change the registry database, it is necessary to change a registry update site, which then automatically updates its associated query sites. No changes can be made directly to a registry query database.

The *site_name* parameter identifies the query site to use. If *site_name* is **NULL**, or a zero-length string, a query site in the local cell is selected by the client agent.

The handle for the associated registry server is returned in *context*.

The caller must have established a valid network identity for this call to succeed.

NOTES

To bind to a registry query site, the use of the *sec_rgy_site_bind_query( )* call is recommended in place of this call.

Like *sec_rgy_site_open( )* routine, this routine binds arbitrarily to either an update or query site. Although update sites can accept queries, query sites cannot accept updates. To specifically select an update site, use *sec_rgy_site_open_update( )*.

FILES

/usr/include/dce/binding.idl
The **idl** file from which **dce/binding.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_login_s_no_current_context
No currently established network identity for which context exists.

sec_rgy_server_unavailable
Server unavailable.

SEE ALSO

Functions: *sec_rgy_site_close*( ), *sec_rgy_site_get*( ), *sec_rgy_site_is_readonly*( ), *sec_rgy_site_open*( ), *sec_rgy_site_open_update*( ).

NAME

sec_rgy_site_open_update — Binds to a registry update site

SYNOPSIS

```
#include <dce/binding.h>

void sec_rgy_site_open_update(
       unsigned_char_t *site_name,
       sec_rgy_handle_t *context,
       error_status_t *status);
```

PARAMETERS

Input

site_name

A character string (type **unsigned_char_t**) containing the name of an update registry site to bind to. Supply this name in any of the following forms:

- To choose the update site to bind to in the named cell, specify a cell name (for example, **/.../r_d.com** or **/.:** for the local cell)
- To start the search for the update site at a specific replica in the replica's cell, specify either the site's global name (for example, **/.../r_d.com/subsys/dce/sec/rs_server_250_2**) or the site's network address (for example, **ncadg_ip_udp**:**15.22.144.248**)

Output

context

A pointer to a **sec_rgy_handle_t** variable. Upon return, this contains a registry server handle indicating (bound to) the desired registry site.

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_site_open_update( )* routine binds to a registry update site. A registry update site is a master server that may control several satellite (query) servers. To change the registry database, it is necessary to change a registry update site, which then automatically updates its associated query sites. No changes can be made directly to a registry query database.

The *site_name* parameter identifies either the cell in which to find the update site or the replica at which to start the search for the update site. If *site_name* is **NULL**, or a zero-length string, an update site in the local cell is selected by the client agent.

The handle for the associated registry server is returned in *context*. The handle is to an update site. Use this registry context handle in subsequent calls that update or query the the registry database (for example, the *sec_rgy_pgo_add( )* or *sec_rgy_acct_lookup( )* call). The caller must have established a valid network identity for this call to succeed.

NOTES

To bind to a registry update site, the use of the *sec_rgy_site_bind_update( )* call is recommended in place of this call.

FILES

/usr/include/dce/binding.idl

The **idl** file from which **dce/binding.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_login_s_no_current_context
No currently established network identity for which context exists.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_rgy_site_close*( ), *sec_rgy_site_get*( ), *sec_rgy_site_is_readonly*( ), *sec_rgy_site_open*( ), *sec_rgy_site_open_query*( ).

NAME

sec_rgy_unix_getgrgid — Returns a UNIX style group entry for the account matching the specified group ID

SYNOPSIS

```
#include <dce/rgynbase.h>

void sec_rgy_unix_getgrgid(
       sec_rgy_handle_t context,
       signed32 gid,
       signed32 max_number,
       sec_rgy_cursor_t *item_cursor,
       sec_rgy_unix_group_t *group_entry,
       signed32 *number_members,
       sec_rgy_member_t member_list[],
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

gid
A 32-bit integer specifying the group ID to match.

max_number
The maximum number of members to be returned by the call. This must be no larger than the allocated size of the *member_list* array.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The *sec_rgy_unix_getgrgid( )* routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns **sec_rgy_no_more_entries**. Use *sec_rgy_cursor_reset( )* to refresh the cursor.

Output

group_entry
A UNIX style group entry structure returned with information about the account matching *gid*.

number_members
An signed 32-bit integer containing the total number of member names returned in the *member_list* array.

member_list[ ]
An array of character strings to receive the returned member names. The size allocated for the array is given by *max_number*. If this value is less than the total number of members in the membership list, multiple calls must be made to return all of the members.

status
On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_unix_getgrgid( )* routine returns the next UNIX group structure that matches the input UNIX group ID. The structure is in the following form:

```
typedef struct {
       sec_rgy_name_t              name;
       signed32                    gid;
       sec_rgy_member_buf_t        members;
} sec_rgy_unix_group_t;
```

The structure includes

- The name of the group.
- The group's UNIX ID.
- A string containing the names of the group members. This string is limited in size by the size of the **sec_rgy_member_buf_t** type defined in **rgynbase.h**.

The routine also returns an array of member names, limited in size by the *number_members* parameter.

This call is supplied in source code form.

FILES

/usr/include/dce/rgynbase.idl
The **idl** file from which **dce/rgynbase.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_no_more_entries
The cursor is at the end of the list of entries.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

NAME

sec_rgy_unix_getgrnam — Returns a UNIX style group entry for the account matching the specified group name

SYNOPSIS

```
#include <dce/rgynbase.h>

void sec_rgy_unix_getgrnam(
       sec_rgy_handle_t context,
       sec_rgy_name_t name,
       signed32 name_length,
       signed32 max_num_members,
       sec_rgy_cursor_t item_cursor,
       sec_rgy_unix_group_t group_entry,
       signed32 number_members,
       sec_rgy_member_t member_list[],
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name
A character string (of type **sec_rgy_name_t**) specifying the group name to be matched.

name_length
A signed 32-bit integer specifying the length of *name* in characters.

max_num_members
The maximum number of members to be returned by the call. This must be no larger than the allocated size of the *member_list* array.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The *sec_rgy_unix_getgrnam( )* routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns **sec_rgy_no_more_entries**. Use *sec_rgy_cursor_reset( )* to refresh the cursor.

Output

group_entry
A UNIX style group entry structure returned with information about the account matching *name*.

number_members
An signed 32-bit integer containing the total number of member names returned in the *member_list* array.

member_list[ ]
An array of character strings to receive the returned member names. The size allocated for

the array is given by *max_number*. If this value is less than the total number of members in the membership list, multiple calls must be made to return all of the members.

status
On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_unix_getgrnam( )* routine looks up the next group entry in the registry that matches the input group name and returns the corresponding UNIX style group structure. The structure is in the following form:

```
typedef struct {
       sec_rgy_name_t              name;
       signed32                    gid;
       sec_rgy_member_buf_t        members;
} sec_rgy_unix_group_t;
```

The structure includes:

- The name of the group.
- The group's UNIX ID.
- A string containing the names of the group members. This string is limited in size by the size of the **sec_rgy_member_buf_t** type defined in **rgynbase.h**.

The routine also returns an array of member names, limited in size by the *number_members* parameter. Note that the array contains only the names explicitly specified as members of the group. A principal that was made a member of the group because that group was assigned as the principal's primary group will not appear in the array.

This call is provided in source code form.

FILES

/usr/include/dce/rgynbase.idl
The **idl** file from which **dce/rgynbase.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy bad_data
The name supplied as input was too long.

sec_rgy_no_more_entries
The cursor is at the end of the list of entries.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

NAME

sec_rgy_unix_getpwnam — Returns a UNIX-style password structure for account matching the specified name

SYNOPSIS

```
#include <dce/rgynbase.h>

void sec_rgy_unix_getpwnam (
       sec_rgy_handle_t context,
       sec_rgy_name_t name,
       unsigned32 name_len,
       sec_rgy_cursor_t *item_cursor,
       sec_rgy_unix_passwd_t *passwd_entry,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

name
A character string (of type **sec_rgy_name_t**) containing the name of the person, group, or organization whose name entry is desired.

name_len
A 32-bit integer representing the length of the *name* in characters.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The *sec_rgy_unix_getpwnam( )* routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns **sec_rgy_no_more_entries**. Use *sec_rgy_cursor_reset( )* to refresh the cursor.

Output

passwd_entry
A UNIX-style password structure returned with information about the account matching *name*.

status
On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_unix_getpwnam( )* routine returns the next UNIX-style password structure that matches the input name. The structure is in the form:

```
typedef struct {
       sec_rgy_unix_login_name_t   name;
       sec_rgy_unix_passwd_buf_t   passwd;
       signed32                    uid;
       signed32                    gid;
       signed32                    oid;
       sec_rgy_unix_gecos_t        gecos;
       sec_rgy_pname_t             homedir;
       sec_rgy_pname_t             shell;
} sec_rgy_unix_passwd_t;
```

The structure includes:

- The account's login name.
- The account's password.
- The account's UNIX ID.
- The UNIX ID of group and organization associated with the account.
- The account's GECOS information.
- The account's home directory.
- The account's login shell

This call is provided in source code form.

FILES

/usr/include/dce/rgynbase.idl
The **idl** file from which **rgynbase.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_bad_data
The name supplied as input was too long.

sec_rgy_no_more_entries
The end of the list of entries has been reached.

NAME

sec_rgy_unix_getpwuid — Returns a UNIX-style password structure for the account matching the specified UUID

SYNOPSIS

```
#include <dce/rgynbase.h>

void sec_rgy_unix_getpwuid(
       sec_rgy_handle_t context,
       signed32 uid,
       sec_rgy_cursor_t *item_cursor,
       sec_rgy_unix_passwd_t *passwd_entry,
       error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open( )* to acquire a bound handle.

uid
A 32-bit integer UNIX ID.

Input/Output

item_cursor
An opaque pointer indicating a specific PGO item entry in the registry database. The *sec_rgy_unix_getpwuid( )* routine returns the PGO item indicated by *item_cursor*, and advances the cursor to point to the next item in the database. When the end of the list of entries is reached, the routine returns **sec_rgy_no_more_entries**. Use *sec_rgy_cursor_reset( )* to refresh the cursor.

Output

passwd_entry
A UNIX style password structure returned with information about the account matching *uid*.

status
On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_unix_getpwuid( )* routine looks up the next password entry in the registry that matches the input UNIX ID and returns the corresponding **sec_rgy_unix_passwd_t** structure. The structure is in the following form:

```
typedef struct {
       sec_rgy_unix_login_name_t   name;
       sec_rgy_unix_passwd_buf_t   passwd;
       signed32                    Vuid;
       signed32                    Vgid;
       signed32                    oid
       sec_rgy_unix_gecos_t        gecos;
       sec_rgy_pname_t             homedir;
       sec_rgy_pname_t             shell;
} sec_rgy_unix_passwd_t;
```

The structure includes:

- The account's login name.
- The account's password.
- The account's UNIX ID.
- The UNIX ID of group and organization associated with the account.
- The account's GECOS information.
- The account's home directory.
- The account's login shell

This call is provided in source code form.

FILES

/usr/include/dce/rgynbase.idl
The **idl** file from which **dce/rgynbase.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_no_more_entries
The end of the list of entries has been reached.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

NAME

sec_rgy_wait_until_consistent — Blocks the caller while prior updates are propagated to the registry replicas

SYNOPSIS

```
#include <dce/misc.h>

boolean32 sec_rgy_wait_until_consistent(
       sec_rgy_handle_t context,
       error_status_t *status);
```

PARAMETERS

Input

context
The registry server handle associated with the master registry.

Output

status
A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_rgy_wait_until_consistent( )* routine blocks callers until all prior updates to the master registry have been propagated to all active registry replicas.

RETURN VALUES

The routine returns TRUE when all active replicas have received the prior updates. It returns FALSE if at least one replica did not receive the updates.

FILES

/usr/include/dce/misc.idl
The **idl** file from which **dce/misc.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_rgy_read_only
Either the master site is in maintenance mode or the site associated with the handle is a read-only (query) site.

sec_rgy_server_unavailable
The server for the master registry is not available.

Chapter 17

ID Map API

17.1 Introduction

The routines in the ID Map API are distinguished with names having the prefix **sec_id_**.

Background is given in Chapter 1, especially Section 1.13 on page 67.

NAME

<dce/secidmap.h> — Header for **sec_id** API

SYNOPSIS

```
#include <dce/secidmap.h>
```

DESCRIPTION

Data Types and Constants

There are no particular data types or constants specific to the **sec_id** API (other than those that have already been introduced in this specification).

In particular, concerning *naming syntax* as used in this chapter (such as the notion of global PGO name), see Section 12.1.1.2 on page 490 (and the other sections referenced there). (Note especially that the name components **/principal/** and **/group/**, which are used to identify RS naming domain junction points for the purpose of ACL management, *do not occur* in the cell-relative PGO names of the present chapter.)

Status Codes

The following status codes (listed in alphabetical order) are used in the **sec_id** API.

sec_id_e_bad_cell_uuid
Cell UUID is not valid.

sec_id_e_foreign_cell_referral
Global name yields an entity in foreign cell — use referral to that cell.

sec_id_e_name_too_long
Name too long (for the implementation).

NAME

sec_id_gen_group — Generate a global group name from cell and group UUIDs.

SYNOPSIS

```
#include <dce/secidmap.h>

void sec_id_gen_group(
    sec_rgy_handle_t context,
    uuid_t *cell_idp,
    uuid_t *group_idp,
    sec_rgy_name_t global_name,
    sec_rgy_name_t cell_namep,
    sec_rgy_name_t group_namep,
    error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open*( ) to acquire a bound handle.

cell_idp
A pointer to the UUID of the home cell of the group whose name is in question.

group_idp
A pointer to the UUID of the group whose name is in question.

Input/Output

global_name
The global (full) name of the group in **sec_rgy_name_t** form (see Section 12.1.1.2 on page 490).

cell_namep
The name of the group's home cell in **sec_rgy_name_t** form.

group_namep
The local (with respect to the home cell) name of the group in **sec_rgy_name_t** form.

Output

status
A pointer to the completion status. On successful completion, the function returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_id_gen_group*( ) routine generates a global name from input cell and group UUIDs. For example, given a UUID specifying the cell **/.../world/hp/brazil**, and a UUID specifying a group resident in that cell named **writers**, the routine would return the global name of that group, in this case, **/.../world/hp/brazil/writers**. It also returns the simple names of the cell and group, translated from the UUIDs.

The routine will not produce translations to any name for which a **NULL** pointer has been supplied.

FILES

/usr/include/dce/secidmap.idl
The **idl** file from which **dce/secidmap.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_id_e_bad_cell_uuid
The cell UUID is not valid.

sec_id_e_name_too_long
The name is too long for current implementation.

sec_rgy_object_not_found
The registry server could not find the specified group.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_id_gen_name*( ), *sec_id_parse_group*( ), *sec_id_parse_name*( ).

Protocols: *rsec_id_gen_name*( ).

NAME

sec_id_gen_name — Generate a global principal name from cell and principal UUIDs

SYNOPSIS

```
#include <dce/secidmap.h>

void sec_id_gen_name (
    sec_rgy_handle_t context,
    uuid_t *cell_idp,
    uuid_t *princ_idp,
    sec_rgy_name_t global_name,
    sec_rgy_name_t cell_namep,
    sec_rgy_name_t princ_namep,
    error_status_t *status );
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open*( ) to acquire a bound handle.

cell_idp
A pointer to the UUID of the home cell of the principal whose name is in question.

princ_idp
A pointer to the UUID of the principal whose name is in question.

Input/Output

global_name
The global (full) name of the principal in **sec_rgy_name_t** form (see Section 12.1.1.2 on page 490).

cell_namep
The name of the principal's home cell in **sec_rgy_name_t** form.

princ_namep
The local (with respect to the home cell) name of the principal in **sec_rgy_name_t** form.

Output

status
A pointer to the completion status. On successful completion, the function returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_id_gen_name*( ) routine generates a global name from input cell and principal UUIDs. For example, given a UUID specifying the cell **/.../world/hp/brazil**, and a UUID specifying a principal resident in that cell named **writers/tom**, the routine would return the global name of that principal, in this case, **/.../world/hp/brazil/writers/tom**. It also returns the simple names of the cell and principal, translated from the UUIDs.

The routine will not produce translations to any name for which a **NULL** pointer has been supplied.

Permissions Required

The *sec_id_gen_name*( ) routine requires at least one permission of any kind on the account associated with the input cell and principal UUIDs.

FILES

/usr/include/dce/secidmap.idl
The **idl** file from which **dce/secidmap.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_id_e_bad_cell_uuid
The cell UUID is not valid.

sec_id_e_name_too_long
The name is too long for current implementation.

sec_rgy_object_not_found
The registry server could not find the specified principal.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_id_gen_group*( ), *sec_id_parse_group*( ), *sec_id_parse_name*( ).

Protocols: *rsec_id_gen_name*( ).

NAME

sec_id_parse_group — Translates a global group name into cell name, cell-relative group name and UUIDs

SYNOPSIS

```
#include <dce/secidmap.h>

void sec_id_parse_group(
    sec_rgy_handle_t context,
    sec_rgy_name_t global_name,
    sec_rgy_name_t cell_namep,
    uuid_t *cell_idp,
    sec_rgy_name_t group_namep,
    uuid_t *group_idp,
    error_status_t *status );
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open*( ) to acquire a bound handle.

global_name
The global (full) name of the group in **sec_rgy_name_t** form (see Section 12.1.1.2 on page 490).

Input/Output

cell_namep
The output name of the group's home cell in **sec_rgy_name_t** form (see Section 12.1.1.2 on page 490).

cell_idp
A pointer to the UUID of the home cell of the group whose name is in question.

group_namep
The local (with respect to the home cell) name of the group in **sec_rgy_name_t** form (see Section 12.1.1.2 on page 490).

group_idp
A pointer to the UUID of the group whose name is in question.

Output

status
A pointer to the completion status. On successful completion, the function returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_id_parse_group*( ) routine translates a global group name into a cell name and a cell-relative group name. It also returns the UUIDs associated with the group and its home cell.

A **NULL** input to any **Input/Output** parameter suppresses parsing of that parameter.

FILES

/usr/include/dce/secidmap.idl
The **idl** file from which **dce/secidmap.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_id_e_bad_cell_uuid
The cell UUID is not valid.

sec_id_e_name_too_long
The name is too long for current implementation.

sec_rgy_object_not_found
The registry server could not find the specified group.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_id_gen_group*( ), *sec_id_gen_name*( ), *sec_id_parse_name*( ).

Protocols: *rsec_id_parse_name*( ).

NAME

sec_id_parse_name — Translates a global name into principal and cell names and UUIDs

SYNOPSIS

```
#include <dce/secidmap.h>

void sec_id_parse_name(
    sec_rgy_handle_t context,
    sec_rgy_name_t global_name,
    sec_rgy_name_t cell_namep,
    uuid_t *cell_idp,
    sec_rgy_name_t princ_namep,
    uuid_t *princ_idp,
    error_status_t *status);
```

PARAMETERS

Input

context
An opaque handle bound to a registry server. Use *sec_rgy_site_open*( ) to acquire a bound handle.

global_name
The global (full) name of the principal in **sec_rgy_name_t** form (see Section 12.1.1.2 on page 490).

Input/Output

cell_namep
The output name of the principal's home cell in **sec_rgy_name_t** form (see Section 12.1.1.2 on page 490).

cell_idp
A pointer to the UUID of the home cell of the principal whose name is in question.

princ_namep
The local (with respect to the home cell) name of the principal in **sec_rgy_name_t** form (see Section 12.1.1.2 on page 490).

princ_idp
A pointer to the UUID of the principal whose name is in question.

Output

status
A pointer to the completion status. On successful completion, the function returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_id_parse_name*( ) routine translates a global principal name into a cell name and a cell-relative principal name. It also returns the UUIDs associated with the principal and its home cell.

A **NULL** input to any **Input/Output** parameter suppresses parsing of that parameter.

Permissions Required

Only if *princ_idp* is requested as output does the *sec_id_parse_name*( ) routine require a permission. In this case, the routine requires at least one permission of any kind on the account whose global principal name is to be translated.

FILES

/usr/include/dce/secidmap.idl
The **idl** file from which **dce/secidmap.h** was derived.

ERRORS

error_status_ok
The call was successful.

sec_id_e_bad_cell_uuid
The cell UUID is not valid.

sec_id_e_name_too_long
The name is too long for current implementation.

sec_rgy_object_not_found
The registry server could not find the specified principal.

sec_rgy_server_unavailable
The DCE Registry Server is unavailable.

SEE ALSO

Functions: *sec_id_gen_name*( ).

Protocols: *rsec_id_parse_name*( ).

Chapter 18

Key Management API

18.1 Introduction

The routines in the Key Management API are distinguished with names having the prefix ‘‘**sec_key_mgmt**’’.

Background is given in Chapter 1, especially Section 1.14 on page 69.

On input, those routines in this API that take a *keydata* argument expect a value of data type **sec_passwd_rec_t *****, and those that take a *keytype* argument expect a value of data type **sec_passwd_type_t *****; furthermore, both of these arguments must be *non-NULL* pointers to single values (not arrays). On output, those operations that give a *keydata* argument yield a value of data type **sec_passwd_rec_t *****, this being a pointer to the first element of an array; this array is terminated by an element whose **key_type** is **sec_passwd_none**.

Those routines in this API that take a **void *****get_key_fn_arg* argument expect a specification of ‘‘local key storage management’’, as defined in this paragraph. Any value of *get_key_fn_arg* other than the two special ones specified in the remainder of this paragraph indicates a (single) argument to be passed to an application-defined ‘‘key acquisition routine’’ (a value of type **rpc_auth_key_retrieval_fn_t**; that is, an *arg* as for *rpc_server_register_auth_info*( ) in the referenced X/Open DCE RPC Specification; see also Section D.7, Authentication, Authorisation and Protection-level Arguments of that specification). If *get_key_fn_arg* is a string value (of type **idl_char ***) that begins with the substring **FILE:** (that is, is of the form ‘‘**FILE:***xy···z*’’ where *xy···z* denotes a substring of arbitrary non-zero length) this indicates that local key storage is implemented via a default implicit implementation-defined key acquisition routine (not further specified in this specification) in the local **key table file** whose full pathname (in the local system’s file namespace) is *xy···z*. A NULL value of *get_key_fn_arg* indicates the default implicit implementation-defined key acquisition routine (defined in the previous sentence), using an implementation-defined default key table file (typically, on POSIX systems, this default key table file is named **/krb/v5srvtab**; that is, this case corresponds to the previous case with argument **FILE:/krb/v5srvtab**).

Note: Access to local resources is subject to implementation and local system access control policies. This is not further mentioned in the entries for these routines, though it does have implications for implementations. For example, local key storage implemented in a local file, such as **/krb/v5srvtab**, is subject to local access control considerations. As such, implementations should exercise due caution in protecting such files (for example, such files should not be located on partitions that can be remotely mounted in an unprotected manner via a network filesystem).

NAME

<dce/keymgmt.h> — Header for **sec_key_mgmt** API.

SYNOPSIS

```
#include <dce/keymgmt.h>
```

DESCRIPTION

Data Types

The following data types (listed in alphabetical order) are used in the **sec_key_mgmt** API.

idl_byte sec_passwd_des_key_t[8]
Indicates a DES key, represented in big/big-endian order (see Section 2.1.4.3 on page 130).

enum sec_passwd_type_t
Indicates key type. The currently registered values are:

sec_passwd_none
Indicates that no key type is present.

sec_passwd_plain
Indicates that the key is plaintext (that is, unencrypted).

sec_passwd_des
Indicates that the key is DES-encrypted.

struct sec_passwd_rec_t
Indicates a password. It contains the following fields:

unsigned32 version_number
Version number.

idl_char *pepper
A character string, to be appended to the password (**key.plain**) before an encryption key is derived from it.

struct key
A structure representing the actual password. It contains the following fields:

sec_passwd_type_t key_type
Indicates the kind of password contained in **tagged_union**.

union tagged_union
Indicates the actual password. It contains the following fields:

(No password is present)
(No password is present if **key_type** = **sec_passwd_none**.)

idl_char *plain
A plaintext (that is, unencrypted) password (this option occurs if **key_type** = **sec_passwd_plain**).

sec_passwd_des_key_t des_key
A DES-encrypted password (this option occurs if **key_type** = **sec_passwd_des**).

unsigned32 sec_key_mgmt_authn_service
Indicates the authentication service in use. The currently registered values are:

rpc_c_authn_none
No authentication.

rpc_c_authn_dce_secret
DCE secret key authentication, as specified in

Status Codes

The following status codes (listed in alphabetical order) are used in the **sec_key_mgmt** API.

error_status_ok
The call was successful.

sec_key_mgmt_e_authn_invalid
Requested authentication service not valid.

sec_key_mgmt_e_auth_unavailable
Authentication service is unavailable.

sec_key_mgmt_e_key_unavailable
Principal's current key is unavailable.

sec_key_mgmt_e_key_unsupported
Requested key type is not supported.

sec_key_mgmt_e_key_version_ex
Key with specified version number already exists in key store.

sec_key_mgmt_e_not_implemented
Unwilling to perform requested operation (or, colloquially, requested operation has ''not been implemented'').

sec_key_mgmt_e_unauthorized
Caller has insufficient authorisation to perform operation.

sec_login_s_no_memory
A memory allocation error occurred.

sec_rgy_object_not_found
No principal was found with the given name.

sec_rgy_server_unavailable
The RS server is unavailable.

sec_s_no_key_seed
Initialisation of random number generator has not been accomplished.

sec_s_no_memory
Unable to allocate memory.

NAME

sec_key_mgmt_change_key — Change (''write'') a principal's key in local key storage and in RS datastore.

SYNOPSIS

```
#include <dce/keymgmt.h>

void sec_key_mgmt_change_key(
        sec_key_mgmt_authn_service authn_service,
        void *get_key_fn_arg,
        idl_char *principal_name,
        unsigned32 key_vno,
        void *keydata,
        sec_timeval_period_t *garbage_collect_time,
        error_status_t *status);
```

PARAMETERS

Input

authn_service
Identifies the authentication service appropriate for this key.

get_key_fn_arg
Key acquisition routine argument (see **<dce/keymgmt.h>**).

principal_name
Name of the principal whose key is to be changed.

key_vno
Version number of the new key.

keydata
The supplied key data (see **<dce/keymgmt.h>**).

Output

garbage_collect_time
Number of seconds (from ''now''), by which time all currently usable tickets (which are protected with the current or previous keys) will have expired (and can therefore be ''garbage collected'' by the application).

status
The completion status.

DESCRIPTION

The *sec_key_mgmt_change_key*( ) routine performs all activities necessary to update a principal's key, both locally and remotely (that is, in local key storage and in the RS datastore), to the specified value. Old keys for the principal are also garbage collected, if appropriate. For more discussion, see Section 1.14 on page 69.

If *key_vno* is specified as 0 (zero), an appropriate non-zero key version number will be selected in an implementation-defined manner.

Any error (that is, *status* ≠ **error_status_ok**) will leave the key state unchanged.

ERRORS

sec_key_mgmt_e_key_unavailable, **sec_key_mgmt_e_authn_invalid**, **sec_key_mgmt_e_auth_unavailable**, **sec_key_mgmt_e_unauthorized**,

sec_key_mgmt_e_key_unsupported, **sec_key_mgmt_e_key_version_ex**, **sec_rgy_server_unavailable**, **sec_rgy_object_not_found**, **sec_login_s_no_memory**, **error_status_ok**.

SEE ALSO

Functions: *sec_key_mgmt_generate_key*( ), *sec_key_mgmt_set_key*( ).

Protocols: *rs_acct_replace*( ).

NAME

sec_key_mgmt_delete_key — Delete specified keys from local key store.

SYNOPSIS

```
#include <dce/keymgmt.h>

void sec_key_mgmt_delete_key(
    sec_key_mgmt_authn_service authn_service,
    void *get_key_fn_arg,
    idl_char *principal_name,
    unsigned32 key_vno,
    error_status_t *status);
```

PARAMETERS

Input

authn_service
Identifies the authentication service appropriate for the keys to be deleted.

get_key_fn_arg
Key acquisition routine argument (see **<dce/keymgmt.h>**).

principal_name
Name of the principal whose key is to be deleted.

key_vno
Version number of key to be deleted.

Output

status
The completion status.

DESCRIPTION

The *sec_key_mgmt_delete_key*( ) routine deletes the specified keys (namely, those of the specified key version number, of all key types) from the local key store, thereby ‘‘revoking’’ all extant tickets protected with those keys.

Any error condition leaves the key state unchanged.

ERRORS

error_status_ok, **sec_key_mgmt_e_authn_invalid**, **sec_key_mgmt_e_key_unavailable**, **sec_key_mgmt_e_unauthorized**.

SEE ALSO

Functions: *sec_key_mgmt_delete_key_type*( ), *sec_key_mgmt_garbage_collect*( ).

NAME

sec_key_mgmt_delete_key_type — Delete a key version of a specified key type from local key store.

SYNOPSIS

```
#include <dce/keymgmt.h>

void sec_key_mgmt_delete_key_type(
        sec_key_mgmt_authn_service authn_service,
        void *get_key_fn_arg,
        idl_char *principal_name,
        void *keytype,
        unsigned32 key_vno,
        error_status_t *status);
```

PARAMETERS

Input

authn_service
Identifies the authentication service appropriate for the key to be deleted.

get_key_fn_arg
Key acquisition routine argument (see **<dce/keymgmt.h>**).

principal_name
Name of the principal whose key type is to be deleted.

keytype
Indicates the key type (see **<dce/keymgmt.h>**).

key_vno
Version number of the key to be deleted.

Output

status
The completion status.

DESCRIPTION

The *sec_key_mgmt_delete_key_type*( ) routine deletes the specified key version of the specified key type from the local key store, thereby ''revoking'' all extant tickets protected with those keys.

Any error condition leaves the key state unchanged.

ERRORS

error_status_ok, **sec_key_mgmt_e_authn_invalid**, **sec_key_mgmt_e_key_unavailable**, **sec_key_mgmt_e_unauthorized**.

SEE ALSO

Functions: *sec_key_mgmt_delete_key*( ), *sec_key_mgmt_garbage_collect*( ).

NAME

sec_key_mgmt_free_key — Free the memory used by a key value.

SYNOPSIS

```
#include <dce/keymgmt.h>

void sec_key_mgmt_free_key(
        void *keydata,
        error_status_t *status);
```

PARAMETERS

Input

keydata
The key data to be freed (see **<dce/keymgmt.h>**).

Output

status
The completion status.

DESCRIPTION

The *sec_key_mgmt_free_key*( ) routine releases any memory allocated for the indicated key data.

ERRORS

error_status_ok.

SEE ALSO

Functions: *sec_key_mgmt_get_key*( ).

NAME

sec_key_mgmt_garbage_collect — Delete unusable keys from local key store.

SYNOPSIS

```
#include <dce/keymgmt.h>

void sec_key_mgmt_garbage_collect(
    sec_key_mgmt_authn_service authn_service,
    void *get_key_fn_arg,
    idl_char *principal_name,
    error_status_t *status);
```

PARAMETERS

Input

authn_service
Identifies the authentication service appropriate for the keys to be garbage-collected.

get_key_fn_arg
Key acquisition routine argument (see **<dce/keymgmt.h>**).

principal_name
Name of the principal whose keys are to be garbage collected.

Output

status
The completion status.

DESCRIPTION

The *sec_key_mgmt_garbage_collect*( ) routine discards unusable keys (that is, keys for which there can be no outstanding ticket protected with that key) for the specified principal from local key store.

ERRORS

error_status_ok, **sec_login_s_no_memory**, **sec_key_mgmt_e_authn_invalid**, **sec_key_mgmt_e_unauthorized**, **sec_key_mgmt_e_key_unavailable**, **sec_rgy_object_not_found**, **sec_rgy_server_unavailable**.

SEE ALSO

Functions: *sec_key_mgmt_delete_key*( ), *sec_key_mgmt_delete_key_type*( ).

NAME

sec_key_mgmt_gen_rand_key — Generate a new random key of specified key type.

SYNOPSIS

```
#include <dce/keymgmt.h>

void sec_key_mgmt_gen_rand_key(
    sec_key_mgmt_authn_service authn_service,
    void *get_key_fn_arg,
    idl_char *principal_name,
    void *keytype,
    unsigned32 key_vno,
    void **keydata,
    error_status_t *status);
```

PARAMETERS

Input

authn_service
Identifies the authentication service appropriate for the generated key.

get_key_fn_arg
Key acquisition routine argument (see **<dce/keymgmt.h>**).

principal_name
Name of a principal. (This argument is for future extensibility, and is currently ignored.)

keytype
Indicates the key type (see **<dce/keymgmt.h>**).

key_vno
Version number of the new key.

Output

keydata
The generated key data (see **<dce/keymgmt.h>**).

status
The completion status.

DESCRIPTION

The *sec_key_mgmt_gen_rand_key*( ) routine generates a new random key for a specified key type. This routine does not actually change any keys, either locally or remotely, though the generated key is suitable for use with *sec_key_mgmt_set_key*( ) and *sec_key_mgmt_change_key*( ).

The storage for *keydata* is allocated dynamically; this storage may be freed with the *sec_key_mgmt_free_key*( ) function.

As an initialisation requirement (to ‘‘seed the random number generator’’), the caller of this routine must have previously made a successful protected RPC call (where ‘‘successful’’ is to be interpreted in the sense of the caller’s security runtime library; that is, it is allowed to have failed ‘‘on the network’’ or ‘‘at the server’’).

ERRORS

sec_key_mgmt_e_not_implemented, **sec_s_no_key_seed**, **sec_s_no_memory**, **error_status_ok**.

SEE ALSO

Functions: *sec_key_mgmt_change_key*( ), *sec_key_mgmt_set_key*( ).

NAME

sec_key_mgmt_get_key — Retrieve a principal's key from local storage.

SYNOPSIS

```
#include <dce/keymgmt.h>

void sec_key_mgmt_get_key(
    sec_key_mgmt_authn_service authn_service,
    void *get_key_fn_arg,
    idl_char *principal_name,
    unsigned32 key_vno,
    void **keydata,
    error_status_t *status);
```

PARAMETERS

Input

authn_service
Identifies the authentication service appropriate for this key.

get_key_fn_arg
Key acquisition routine argument (see **<dce/keymgmt.h>**).

principal_name
Name of the principal to whom the key belongs.

key_vno
The version number of the desired key.

Output

keydata
The returned key data (see **<dce/keymgmt.h>**).

status
The completion status.

DESCRIPTION

The *sec_key_mgmt_get_key*( ) routine retrieves the specified key from the local key store.

The memory for *keydata* is dynamically allocated, and is to be freed by *sec_key_mgmt_free_key*( ).

ERRORS

error_status_ok, **sec_key_mgmt_e_authn_invalid**, **sec_key_mgmt_e_key_unavailable**, **sec_key_mgmt_e_unauthorized**, **sec_s_no_memory**.

SEE ALSO

Functions: *sec_key_mgmt_free_key*( ).

NAME

sec_key_mgmt_get_next_key — Retrieve key indicated by cursor from the local key storage.

SYNOPSIS

```
#include <dce/keymgmt.h>

void sec_key_mgmt_get_next_key(
    void *cursor,
    idl_char **principal_name,
    unsigned32 *key_vno,
    void **keydata,
    error_status_t *status);
```

PARAMETERS

Input

cursor
The current retrieval position in the local key storage.

Output

principal_name
Name of the principal associated with the retrieved key.

key_vno
The version number of the extracted key.

keydata
The retrieved key data (see **<dce/keymgmt.h>**).

status
The completion status.

DESCRIPTION

The *sec_key_mgmt_get_next_key*( ) routine retrieves the key indicated by the cursor in the local key store, and updates the cursor to point to the next key. The entire local key store can be scanned by a series of calls to this routine.

ERRORS

error_status_ok, **sec_key_mgmt_e_key_unavailable**, **sec_key_mgmt_e_unauthorized**, **sec_s_no_memory**.

SEE ALSO

Functions: *sec_key_mgmt_get_key*( ), *sec_key_mgmt_initialize_cursor*( ).

NAME

sec_key_mgmt_get_next_kvno — Retrieve the next eligible key version number for a key.

SYNOPSIS

```
#include <dce/keymgmt.h>

void sec_key_mgmt_get_next_kvno(
        sec_key_mgmt_authn_service authn_service,
        void *get_key_fn_arg,
        idl_char *principal_name,
        void *keytype,
        unsigned32 *key_vno,
        unsigned32 *next_key_vno,
        error_status_t *status);
```

PARAMETERS

Input

authn_service
Identifies the authentication service appropriate for this key.

get_key_fn_arg
Key acquisition routine argument (see **<dce/keymgmt.h>**).

principal_name
Name of the principal associated with the key.

keytype
Indicates the key type (see **<dce/keymgmt.h>**).

Input/Output

key_vno
The current version number of the key. Specifying NULL prevents this value from being returned.

next_key_vno
The next eligible version number for the key. Specifying NULL prevents this value from being returned.

Output

status
The completion status.

DESCRIPTION

The *sec_key_mgmt_get_next_kvno*( ) routine returns the current and next eligible version numbers for a key from the registry server (not from the local key table).

ERRORS

error_status_ok, **sec_key_mgmt_e_authn_invalid**, **sec_key_mgmt_e_key_unavailable**, **sec_key_mgmt_e_unauthorized**, **sec_rgy_object_not_found**, **sec_rgy_server_unavailable**.

SEE ALSO

Protocols: *rs_acct_lookup*( ).

NAME

sec_key_mgmt_initialize_cursor — Initialise cursor in local key store.

SYNOPSIS

```
#include <dce/keymgmt.h>

void sec_key_mgmt_initialize_cursor(
    sec_key_mgmt_authn_service authn_service,
    void *get_key_fn_arg,
    idl_char *principal_name,
    void *keytype,
    void **cursor,
    error_status_t *status);
```

PARAMETERS

Input

authn_service
Identifies the authentication service appropriate for this key.

get_key_fn_arg
Key acquisition routine argument (see **<dce/keymgmt.h>**).

principal_name
Name of the principal whose key is to be changed.

keytype
Indicates the key type (see **<dce/keymgmt.h>**).

Output

cursor
The returned cursor value.

status
The completion status.

DESCRIPTION

The *sec_key_mgmt_initialize_cursor*( ) routine initialises the cursor in the local key store. This prepares the cursor for a scan of the local key store via a series of calls to *sec_key_mgmt_get_next_key*( ).

The storage for the cursor information is allocated dynamically, so the returned pointer actually indicates a pointer to the cursor value. The storage for this data may be freed with the *sec_key_mgmt_release_cursor*( ) routine.

ERRORS

error_status_ok, **sec_key_mgmt_e_authn_invalid**, **sec_key_mgmt_e_unauthorized**, **sec_s_no_memory**.

SEE ALSO

Functions: *sec_key_mgmt_get_next_key*( ), *sec_key_mgmt_release_cursor*( ).

NAME

sec_key_mgmt_manage_key — Automatically change a principal's key on a periodic basis.

SYNOPSIS

```
#include <dce/keymgmt.h>

void sec_key_mgmt_manage_key(
        sec_key_mgmt_authn_service authn_service,
        void *get_key_fn_arg,
        idl_char *principal_name,
        error_status_t *status);
```

PARAMETERS

Input

authn_service
Identifies the authentication service appropriate for this key.

get_key_fn_arg
Key acquisition routine argument (see **<dce/keymgmt.h>**).

principal_name
Name of the principal whose key is to be managed.

Output

status
The completion status.

DESCRIPTION

The *sec_key_mgmt_manage_key*( ) routine changes (both locally and remotely) the specified principal's key on a periodic basis, as determined by the local cell's policy. It runs indefinitely, never returning during normal operation (and therefore should be invoked only from a dedicated ''key management thread'').

Conceptually, this routine operates as follows (this description imposes no requirements on implementations). First it queries the login context to determine the password expiration date that applies to the named principal. It then idles until a ''short time'' (implementation-dependent) before the current key is due to expire, and then calls *sec_key_mgmt_gen_rand_key*( ) (or similar functionality), thereby changing both the local key store and the RS datastore to a new random key. This routine may also call *sec_key_mgmt_garbage_collect*( ) (or similar functionality) as needed to discard unusable keys from the local key store.

ERRORS

error_status_ok, **sec_rgy_object_not_found**, **sec_key_mgmt_e_authn_invalid**, **sec_key_mgmt_e_key_unavailable**, **sec_key_mgmt_e_key_unsupported**, **sec_key_mgmt_e_unauthorized**, **sec_rgy_server_unavailable**.

SEE ALSO

Functions: *sec_key_mgmt_change_key*( ), *sec_key_mgmt_gen_rand_key*( ), *sec_key_mgmt_garbage_collect*( ).

NAME

sec_key_mgmt_release_cursor — Release memory used by a cursor.

SYNOPSIS

```
#include <dce/keymgmt.h>

void sec_key_mgmt_release_cursor(
        void **cursor,
        error_status_t *status);
```

PARAMETERS

Input

cursor
Cursor value for which the memory is to be released.

Output

status
The completion status.

DESCRIPTION

The *sec_key_mgmt_release_cursor*( ) routine releases any memory allocated for the indicated cursor.

ERRORS

error_status_ok, **sec_key_mgmt_e_unauthorized**.

SEE ALSO

Functions: *sec_key_mgmt_initialize_cursor*( ).

NAME

sec_key_mgmt_set_key — Insert a key value into local storage.

SYNOPSIS

```
#include <dce/keymgmt.h>

void sec_key_mgmt_set_key(
        sec_key_mgmt_authn_service authn_service,
        void *get_key_fn_arg,
        idl_char *principal_name,
        unsigned32 key_vno,
        void *keydata,
        error_status_t *status);
```

PARAMETERS

Input

authn_service
Identifies the authentication service appropriate for this key.

get_key_fn_arg
Key acquisition routine argument (see **<dce/keymgmt.h>**).

principal_name
Name of the principal associated with the key to be set.

key_vno
Version number of the key to be set.

keydata
The key to be stored (see **<dce/keymgmt.h>**).

Output

status
The completion status.

DESCRIPTION

The *sec_key_mgmt_set_key*( ) routine sets a specified key value in local key storage. This routine does not update the RS.

There exist circumstances in which a server may only wish to change its key in the local key storage, and not in the RS datastore. For one example, when a new server principal is created, its initial key must be set in local key store ''manually'' (that is, via *sec_key_mgmt_set_key*( )). For another example, a database system may have several replicas of a master database, each managed by a server running on a different machine. Since these servers together represent only one ''service'', they may (depending on policy) all share the same key. This way, a client with a ticket to use the database can, for example, choose whichever server is least busy. To change the key of such a replicated ''service'', the master server could signal all the ''slave'' (''secondary'') servers to change the current key in their local key storage. Each of them would use *sec_key_mgmt_set_key*( ) (which does not update the key in the RS). Once all the slaves have complied, the master server can then change its own local key and the RS key.

The storage for *keydata* is allocated dynamically; this storage may be freed with *sec_key_mgmt_free_key*( ).

ERRORS

error_status_ok, **sec_key_mgmt_e_authn_invalid**, **sec_key_mgmt_e_key_unavailable**, **sec_key_mgmt_e_key_unsupported**, **sec_key_mgmt_e_key_version_ex**, **sec_key_mgmt_e_unauthorized**.

SEE ALSO

Functions: *sec_key_mgmt_change_key*( ), *sec_key_mgmt_gen_rand_key*( ).

Chapter 19

Login API

19.1 Introduction

The routines in the Login API are distinguished with names having the prefix ‘‘**sec_login_**’’.

Background is given in Chapter 1, especially Section 1.15 on page 71.

NAME

<dce/sec_login.h> — Header for **sec_login** API.

SYNOPSIS

```
#include <dce/sec_login.h>
```

DESCRIPTION

Data Types

The following data types (listed in alphabetical order) are used in **sec_login** API.

enum sec_login_auth_src_t

Indicates the source of authentication or certification (that is, the ''certification authority'') of a login context. The following values are currently registered:

sec_login_auth_src_network

Login context certified by ''network authority'' (that is, KDS/PS, in ''network TCB''). Such a login context contains usable network credentials; that is, it can be used to make protected RPCs to any other DCE subject.

sec_login_auth_src_local

Login context certified by ''local authority'' (that is, local TCB). Such a login context does not contain usable network credentials; that is, it can be used to make protected RPCs only within the context of the local TCB (that is, only to subjects represented by other processes co-located on the same host as the caller).

unsigned32 sec_login_flags_t

A flag word describing attributes of a login context. The following flag is currently registered:

sec_login_credentials_private

This login context is restricted to the current process. If this flag is not set, this login context may be shared with other processes (via *sec_login_export_context*( ) and *sec_login_import_context*( )).

Additionally, the value **sec_login_no_flags** of **sec_login_flags_t** indicates that no flags are set.

idl_void_p_t sec_login_handle_t

This is a pointer to a data structure representing an account's (''*network*'') *login context* (the pointed-to structure is not further specified; that is, **sec_login_handle_t** is an ''opaque pointer'').

Conceptually, the login context contains a copy of all the account's information contained in the RS datastore relevant to the accounts operating in a DCE security environment (as specified in this document), appropriately protected. In the case of the Kerberos authentication service (the only authentication service currently supported by DCE), a login context conceptually contains, among other things, TGTs and PTGTs (targeted to the account's local KDS as well as to remote KDSs) — also referred to colloquially as ''network credentials''.

struct sec_login_net_info_t

Indicates certain ''network information'' associated with a login context. It includes the following fields:

sec_id_pac_t pac

The login context's PAC.

unsigned32 acct_expiration_date
The login context's account expiration date (measured in seconds from midnight January 1, 1970 UTC).

unsigned32 passwd_expiration_date
The login context's long-term key expiration date (measured in seconds from midnight January 1, 1970 UTC).

unsigned32 identity_expiration_date
The login context's expiration date (measured in seconds from midnight January 1, 1970 UTC). Conceptually, this is the expiration date of the TGT to the local KDS held in the login context.

A value of 0 for any of the above expiration dates means ''forever''; that is, the information does not expire — it remains usable indefinitely.

idl_void_p_t sec_login_passwd_t
Pointer to data structure (whose internal structure is not further specified; that is, **sec_login_passwd_t** is an ''opaque pointer'') representing a *password structure*, used for local host purposes.

The detailed content of this structure is implementation-dependent. As an *example*, on POSIX-compliant operating systems, it will typically contain fields such as, or similar to, the following:

char *pw_name
User's name.

char *pw_passwd
Encrypted password.

int pw_uid
User's POSIX UID (local host user identity).

int pw_gid
User's POSIX GID (local host principal group identity).

time_t pw_change
Password expiration date.

char *pw_gecos
User's fullname (or other account information).

char *pw_dir
Home directory.

char *pw_shell
Default shell.

time_t pw_expire
Account expiration date.

struct sec_login_tkt_info_t
The structure of optional AS ticket request flags and associated data. It includes the following fields:

sec_login_tkt_flags_t options
The types of ticket options (requested). The options are listed in the **Constants** section for type **sec_login_tkt_flags_t**.

sec_timeval_period_t postdated_dormanttime
A time period expressed in seconds relative to some other well known base time. In this instance, it indicates the dormant time to be permitted. If the ticket **optionf1 field specifies a postdated ticket (flag sec_login_tkt_postdated** is set), this field must be specified.

sec_timeval_period_t renewable_lifetime
The renewable lifetime of the ticket if the **options** field specifies a renewable ticket. It must be specified if a renewable ticket is being requested (if the **sec_login_tkt_renewable** flag is set in the **options** field).

sec_timeval_period_t lifetime
A non-default ticket lifetime that is specified (in seconds) and which must be specified if a non-default ticket lifetime (**sec_login_tkt_lifetime** flag is set in the **options** field.

Constants

The following constants are used in **sec_login_** calls:

sec_login_handle_t sec_login_default_handle
The value of a login context handle before setup or validation.

sec_login_flags_t sec_login_no_flags
No flags are set.

sec_login_flags_t sec_login_credentials_private
Restricts the validated network credentials to the current process. If this flag is not set, it is permissible to share credentials with descendents of current process.

sec_login_flags_t sec_login_external_tgt
Specifies that externally obtained TGTs are to be used. This is a simple proxy mechanism.

sec_login_flags_t sec_login_machine_princ
Specifies that the login context is being created or validated by the machine principal.

In addition to those already listed above, the following constants are used in **sec_login_** calls to request various attributes associated with tickets (TGTs):

sec_login_tkt_flags_t sec_login_tkt_renewable
Request for a renewable ticket.

sec_login_tkt_flags_t sec_login_tkt_postdated
Request for a postdated ticket.

sec_login_tkt_flags_t sec_login_tkt_allow_postdate
Permit postdated tickets to be used.

sec_login_tkt_flags_t sec_login_tkt_proxiable
Permit proxiable tickets to be used.

sec_login_tkt_flags_t sec_login_tkt_forwardable
Request for a forwardable ticket.

sec_login_tkt_flags_t sec_login_tkt_renewable_ok
Instructions to accept a renewable ticked if a real ticket cannot be granted.

sec_login_tkt_flags_t sec_login_tkt_lifetime
Request for a non-default ticket lifetime.

Status Codes

The following status codes, listed in alphabetical order, are used in **sec_login** calls. The status codes used in delegation are listed separately after this list:

error_status_ok
Routine completed successfully.

sec_login_s_acct_invalid
Account is invalid.

sec_login_s_already_valid
Login context has already been validated.

sec_login_s_auth_local
Operation not valid on local context.

sec_login_s_config
Bad configuration file (or SCD could not validate the TGT).

sec_login_s_context_invalid
Context has not been validated.

sec_login_s_default_use
Illegal use of default **sec_login** handle.

sec_login_s_groupset_invalid
The group set is not valid.

sec_login_s_handle_invalid
Context handle not valid.

sec_login_s_info_not_avail
Information not available.

sec_login_s_internal_error
Internal error (for example, unexpected violation of internal invariants, I/O problems, and so on).

sec_login_s_no_current_context
No currently established network identity for which context exists.

sec_login_s_no_memory
No memory available.

sec_login_s_not_certified
Login context is (validated but) not certified.

Note: This status value is considered ''advisory'' only (advising the caller that the login context in use is not certified). Routines that return this status value are not considered to have ''failed'' (unless the caller requires a *certified* login context); in particular, valid data may be returned to the caller with this status value.

sec_login_s_null_password
Password is a NULL password.

sec_login_s_privileged
Caller is not ''privileged'', in some implementation-specific (local operating system) sense.

The routines currently specified in this chapter that can return this status value are the following: *sec_login_init_first*( ), *sec_login_setup_first*( ), *sec_login_valid_and_cert_ident*( ), *sec_login_validate_first*( ). Thus, these routines fail unless the caller is ''privileged'' (in a local-operating-system sense that must be documented in implementation-specific documentation).

In the case of POSIX-compliant operating systems, the ''classical'' interpretation of ''privileged'' is that the caller's effective POSIX UID is 0 (but note that this ''classical'' interpretation is undergoing transformation as POSIX standardisation progresses). Thus on such systems, implementations of these routines fail unless the caller has effective POSIX UID 0.

sec_login_s_refresh_ident_bad
This indicates that the calling identity has changed since the login context was created or last refreshed, in one of the following senses:

1. principal UUID or primary group UUID has changed
2. groupset UUIDs have been added to. (Deletions from the groupset are okay; if the intersection of the old and new groupsets is empty, the refreshed context will have an empty groupset.)

sec_login_s_unsupp_passwd_type
The password is an unsupported type.

Status Codes Specific to Delegation

The following status codes, listed in alphabetical order, are used in **sec_login** calls dealing with delegation:

error_status_ok
Routine completed successfully.

err_sec_login_invalid_delegate_restriction
This self-descriptive status code is reserved for future use.

err_sec_login_invalid_target_restriction
This self-descriptive status code is reserved for future use.

err_sec_login_invalid_opt_restriction
This self-descriptive status code is reserved for future use.

err_sec_login_invalid_req_restriction
This self-descriptive status code is reserved for future use.

sec_login_s_compound_delegate
Login context already specifies a delegation chain.

sec_login_s_default_use
Invalid use of default **sec_login** handle

sec_login_s_invalid_context
Context has not been validated. (Not a valid login context.)

sec_login_s_invalid_compat_mode
Invalid compatibility mode selection.

sec_cred_s_invalid_cursor
Invalid credential cursor.

sec_login_s_invalid_deleg_type
Invalid delegation type selection.

sec_login_s_deleg_not_enabled
Delegation has not been enabled.

sec_login_s_no_memory
No memory available (Unable to allcoate memory).

sec_cred_s_no_more_entries
No more entries available.

NAME

sec_login_become_delegate — Causes an intermediate server to become a delegate in traced delegation chain

SYNOPSIS

```
#include <dce/sec_login.h>

sec_login_handle_t sec_login_become_delegate (
    rpc_authz_cred_handle_t callers_identity,
    sec_login_handle_t my_login_context,
    sec_id_delegation_type_t delegation_type_permitted,
    sec_id_restriction_set_t *delegate_restrictions,
    sec_id_restriction_set_t *target_restrictions,
    sec_id_opt_req_t *optional_restrictions,
    sec_id_opt_req_t *required_restrictions,
    sec_id_compatibility_mode_t compatibility_mode,
    error_status_t *status );
```

PARAMETERS

Input

callers_identity

A handle of type **rpc_authz_cred_handle_t** to the authenticated identity of the previous delegate in the delegation chain. The handle is supplied by the *rpc_binding_inq_auth_caller*( ) call.

my_login_context

A value of **sec_login_handle_t** that provides an opaque handle to the identity of the client that is becoming the intermediate delegate. The **sec_login_handle_t** that specifies the client's identity is supplied as output of the following calls:

- *sec_login_get_current_context*( ), if the client inherited the identity of the current context
- The *sec_login_setup_identity*( ) and the *sec_login_validate_identity*( ) pair that together establish an authenticated identity if a new identity was established

Note that this identity specified by **sec_login_handle_t** must be a simple login context; it cannot be a compound identity created by a previous *sec_login_become_delegate*( ) call.

delegation_type_permitted

A value of **sec_id_delegation_type_t** that specifies the type of delegation to be enabled. The types available are as follows:

sec_id_deleg_type_none
No delegation.

sec_id_deleg_type_traced
Traced delegation.

sec_id_deleg_type_impersonation
Simple (impersonation) delegation.

Note that the initiating client sets the type of delegation. If it is set as traced, all delegates must also specify traced delegation; they cannot specify simple

delegation. The same is true if the initiating client sets the delegation type as simple; all subsequent delegates must also specify simple delegation. The intermediate delegates can, however, specify no delegation to indicate that the delegation chain can proceed no further.

delegate_restrictions
A pointer to a **sec_id_restriction_set_t** that supplies a list of servers that can act as delegates for the intermediate client identified by *my_login_context*. These servers are added to delegates permitted by the *delegate_restrictions* parameter of the *sec_login_become_initiator*( ) call.

target_restrictions
A pointer to a **sec_id_restriction_set_t** that supplies a list of servers that can act as targets for the intermediate client identified by *my_login_context*. These servers are added to targets specified by the *target_restrictions* parameter of the *sec_login_become_initiator*( ) call.

optional_restrictions
A pointer to a **sec_id_opt_req_t** that supplies a list of application-defined optional restrictions that apply to the intermediate client identified by *my_login_context*. These restrictions are added to the restrictions identified by the *optional_restrictions* parameter of the *sec_login_become_initiator*( ) call.

required_restrictions
A pointer to a **sec_id_opt_req_t** that supplies a list of application-defined required restrictions that apply to the intermediate client identified by *my_login_context*. These restrictions are added to the restrictions identified *required_restrictions* parameter of the *sec_login_become_initiator*( ) call.

compatibility_mode
A value of **sec_id_compatibility_mode_t** that specifies the compatibility mode to be used when the intermediate client operates on pre-1.1 servers. The modes available are as follows:

sec_id_compat_mode_none
Compatibility mode is off.

sec_id_compat_mode_initiator
Compatibility mode is on. The pre-1.1 PAC data is extracted from the EPAC of the initiating client.

sec_id_compat_mode_caller
Compatibility mode is on. The pre-1.1 PAC data extracted from the EPAC of the last client in the delegation chain.

Output

status
A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_login_become_delegate*( ) is used by intermediate servers to become a delegate for the client identified by *callers_identity*. The routine returns a new login context (of type **sec_login_handle_t**) that carries delegation information. This information includes the delegation type, delegate and target restrictions, and any application-defined optional and required restrictions.

The new login context created by this call can then used to to set up authenticated rpc with an intermediate or target server using the *rpc_binding_set_auth_info*( ) call.

Any delegate, target, required, or optional restrictions specified in this call are added to the restrictions specified by the initiating client and any intermediate clients.

The *sec_login_become_delegate*( ) call is run only if the initiating client enabled traced delegation by setting the *delegation_type_permitted* parameter in the *sec_login_become_initiator*( ) call to **sec_id_deleg_type_traced**.

FILES

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

ERRORS

err_sec_login_invalid_delegate_restriction

err_sec_login_invalid_target_restriction

err_sec_login_invalid_opt_restriction

err_sec_login_invalid_req_restriction

sec_login_s_invalid_context

sec_login_s_compound_delegate

sec_login_s_invalid_deleg_type

sec_login_s_invalid_compat_mode

sec_login_s_deleg_not_enabled

error_status_ok

RELATED INFORMATION

Functions: *rpc_binding_inq_auth_caller*( ), *sec_login_become_impersonator*( ), *sec_login_become_initiator*( ), *sec_login_get_current_context*( ), *sec_login_setup_identity*( ), *sec_login_validate_identity*( ).

NAME

sec_login_become_impersonator — Causes an intermediate server to become a delegate in a simple delegation chain

SYNOPSIS

```
#include <dce/sec_login.h>

sec_login_handle_t sec_login_become_impersonator (
    rpc_authz_cred_handle_t callers_identity,
    sec_id_delegation_type_t delegation_type_permitted,
    sec_id_restriction_set_t *delegate_restrictions,
    sec_id_restriction_set_t *target_restrictions,
    sec_id_opt_req_t *optional_restrictions,
    sec_id_opt_req_t *required_restrictions,
    error_status_t *status );
```

PARAMETERS

Input

callers_identity

A handle of type **rpc_authz_cred_handle_t** to the authenticated identity of the previous delegate in the delegation chain. The handle is supplied by the *rpc_binding_inq_auth_caller*( ) call.

delegation_type_permitted

A value of **sec_id_delegation_type_t** that specifies the type of delegation to be enabled. The types available are as follows:

sec_id_deleg_type_none
No delegation.

sec_id_deleg_type_traced
Traced delegation.

sec_id_deleg_type_impersonation
Simple (impersonation) delegation.

The initiating client sets the type of delegation. If it is set as traced, all delegates must also specify traced delegation; they cannot specify simple delegation. The same is true if the initiating client sets the delegation type as simple; all subsequent delegates must also specify simple delegation. The intermediate delegates can, however, specify no delegation to indicate that the delegation chain can proceed no further.

delegate_restrictions

A pointer to a **sec_id_restriction_set_t** that supplies a list of servers that can act as delegates for the client becoming the delegate. These servers are added to the delegates permitted by the *delegate_restrictions* argument of the *sec_login_become_initiator*( ) call

target_restrictions

A pointer to a **sec_id_restriction_set_t** that supplies a list of servers that can act as targets for the client becoming the delegate. These servers are added to targets specified by the *target_restrictions* argument of the *sec_login_become_initiator*( ) call.

optional_restrictions
A pointer to a **sec_id_opt_req_t** that supplies a list of application-defined optional restrictions that apply to the client becoming the delegate. These restrictions are added to the restrictions identified by the *optional_restrictions* argument of the *sec_login_become_initiator*( ) call.

required_restrictions
A pointer to a **sec_id_opt_req_t** that supplies a list of application-defined required restrictions that apply to the client becoming the delegate. These restrictions are added to the restrictions identified *required_restrictions* argument of the *sec_login_become_initiator*( ) call.

Output

status
A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_login_become_impersonator*( ) is used by intermediate servers to become a delegate for the client identified by *callers_identity*. The routine returns a new login context (of type **sec_login_handle_t**) that carries delegation information. This information includes the delegation type, delegate, and target restrictions, and any application-defined optional and required restrictions. The new login context created by this call can then used to to set up authenticated rpc with an intermediate or target server using the *rpc_binding_set_auth_info*( ) call. The effective optional and required restrictions are the union of the optional and required restrictions specified in this call and specified by the initiating client and any intermediate clients. The effective target and delegate restrictions are the intersection of the target and delegate restrictions specified in this call and specified by the initiating client and any intermediate clients. The *sec_login_become_impersonator*( ) call is call is run only if the initiating client enabled simple delegation by setting the *delegation_type_permitted* argument in the *sec_login_become_initiator*( ) call to **sec_id_deleg_type_simple**.

FILES

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

ERRORS

err_sec_login_invalid_delegate_restriction

err_sec_login_invalid_target_restriction

err_sec_login_invalid_opt_restriction

err_sec_login_invalid_req_restriction

sec_login_s_invalid_deleg_type

sec_login_s_invalid_compat_mode

sec_login_s_deleg_not_enabled

error_status_ok

SEE ALSO

Functions: *rpc_binding_inq_auth_caller*( ), *sec_login_become_initiator*( ).

NAME

sec_login_become_initiator — Constructs a new login context that enables delegation for the calling client

SYNOPSIS

```
#include <dce/sec_login.h>

sec_login_handle_t sec_login_become_initiator (
    sec_login_handle_t my_login_context,
    sec_id_delegation_type_t delegation_type_permitted,
    sec_id_restriction_set_t *delegate_restrictions,
    sec_id_restriction_set_t *target_restrictions,
    sec_id_opt_req_t *optional_restrictions,
    sec_id_opt_req_t *required_restrictions,
    sec_id_compatibility_mode_t compatibility_mode,
    error_status_t *status );
```

PARAMETERS

Input

my_login_context

A value of **sec_login_handle_t** that provides an opaque handle to the identity of the client that is enabling delegation. The **sec_login_handle_t** that specifies the client's identity is supplied as output of the following calls:

- *sec_login_get_current_context*( ) if the client inherited the identity of the current context
- The *sec_login_setup_identity*( ) and the *sec_login_validate_identity*( ) pair that together establish an authentiated identity if a new identity was established

delegation_type_permitted

A value of **sec_id_delegation_type_t** that specifies the type of delegation to be enabled. The types available are as follows:

sec_id_deleg_type_none
No delegation.

sec_id_deleg_type_traced
Traced delegation.

sec_id_deleg_type_impersonation
Simple (impersonation) delegation.

Note each subsequent intermediate delegate of the delegation chain started by the initiating client must set the delegation type to traced if the initiating client set it to traced or to simple if the initiating client set it to simple. Intermediate delegates, however, can set the delegation type to no delegation to indicate that the delegation chain can proceed no further.

delegate_restrictions

A pointer to a **sec_id_restriction_set_t** that supplies a list of servers that can act as delegates for the client initiating delegation.

target_restrictions

A pointer to a **sec_id_restriction_set_t** that supplies a list of servers that can act as targets for the client initiating delegation.

optional_restrictions
A pointer to a **sec_id_opt_req_t** that supplies a list of application-defined optional restrictions that apply to the client initiating delegation.

required_restrictions
A pointer to a **sec_id_opt_req_t** that supplies a list of application-defined required restrictions that apply to the client initiating delegation.

compatibility_mode
A value of **sec_id_compatibility_mode_t** that specifies the compatibility mode to be used when the initiating client interacts with pre-1.1 servers. The modes available are as follows:

sec_id_compat_mode_none
Compatibility mode is off.

sec_id_compat_mode_initiator
Compatibility mode is on. The pre-1.1 PAC data is extracted from the EPAC of the initiating client.

sec_id_compat_mode_caller
Compatibility mode is on. The pre-1.1 PAC data extracted from the EPAC of the last client in the delegation chain.

Output

status
A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_login_become_initiator*( ) enables delegation for the calling client by constructing a new login context (in a **sec_login_handle_t**) that carries delegation information. This information includes the delegation type, delegate, and target restrictions, and any application-defined optional and required restrictions. The new login context is then used to to set up authenticated rpc with an intermediate server using the *rpc_binding_set_auth_info*( ) call. The intermediary can continue the delegation chain by calling *sec_login_become_delegate*( ) (if the delegation type is **sec_id_deleg_type_traced**) or *sec_login_become_impersonator*( ) (if the delegation type is **sec_id_deleg_type_impersonation**).

FILES

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

ERRORS

err_sec_login_invalid_delegate_restriction

err_sec_login_invalid_target_restriction

err_sec_login_invalid_opt_restriction

err_sec_login_invalid_req_restriction

error_status_ok

sec_login_s_invalid_compat_mode

sec_login_s_invalid_context

sec_login_s_invalid_deleg_type

SEE ALSO

Functions: *sec_login_become_delegate*( ), *sec_login_become_impersonator*( ), *sec_login_get_current_context*( ), *sec_login_setup_identity*( ), *sec_login_validate_identity*( ).

NAME

sec_login_certify_identity — Certify a (validated) login context.

SYNOPSIS

```
#include <dce/sec_login.h>

boolean32 sec_login_certify_identity(
    sec_login_handle_t login_context,
    error_status_t *status);
```

PARAMETERS

Input

login_context
Login context to be certified.

Output

status
The completion status.

DESCRIPTION

The *sec_login_certify_identity*( ) routine *certifies* a (validated) login context; that is, demonstrates its trustworthiness (for the purpose of basing access decisions on information carried in it) to parties other than the principal/account to which it is associated.

In typical implementations this is accomplished by using the login context to execute a protected RPC (of authentication type **rpc_c_authn_dce_secret**, of authorisation type **rpc_c_authz_dce**, and of protection level **rpc_c_protect_level_pkt_integ**) to the local host's SCD. If an implementation of *sec_login_certify_identity*( ) does not support the same strong guarantee of ''infallible'' certification that *sec_login_valid_and_cert_ident*( ) does, this fact (as well as the information about the strength of the guarantee that actually is supported) must be noted in the implementation's documentation of *sec_login_certify_identity*( ). (See Section 1.15.2 on page 77 for details.)

Typically, this routine is called by a host's login program, which uses the information contained in the login context to set security attributes of the logging-in user (principal/account) that will be subsequently used for access control to the local host's resources (such as computing power) — see *sec_login_get_pwent*( ), *sec_login_get_groups*( ) and *sec_login_get_expiration*( ).

In typical implementations, if this operation succeeds, it updates local security registration information on the local host (information derived from information in the (now-certified) login context). This locally held information can be used for subsequent logins if the RS is unreachable (for example, because of a network partition), though such information is usable only for access to local resources (that is, it endows a process with local identity information, but not with a login context that can be used for protected RPCs).

RETURN VALUES

The routine returns a non-0 (TRUE) value if the certification was successful, and 0 (FALSE) otherwise.

ERRORS

error_status_ok

sec_login_s_config

sec_login_s_context_invalid

sec_login_s_default_use

SEE ALSO

Functions: *sec_login_get_pwent*( ), *sec_login_get_groups*( ), *sec_login_get_expiration*( ), *sec_login_valid_and_cert_ident*( ).

Protocols: *scd_protected_noop*( ).

NAME

sec_login_cred_get_delegate — Returns a handle to the privilege attributes of an intermediary in a delegation chain

SYNOPSIS

```
#include <dce/sec_login.h>

sec_cred_pa_handle_t sec_login_cred_get_delegate (
    sec_login_handle_t login_context,
    sec_cred_cursor_t *cursor,
    error_status_t *status );
```

PARAMETERS

Input

login_context

A value of **sec_login_handle_t** that provides an opaque handle to a login context for which delegation has been enabled. The **sec_login_handle_t** that specifies the identity is supplied as output of the *sec_login_become_delegate*( ) call.

Input/Output

cursor

As input, a pointer to a cursor of type **sec_cred_cursor_t** that has been initialized by the *sec_login_cred_init_cursor*( ) call. As an output argument, *cursor* is a pointer to a cursor of type **sec_cred_cursor_t** that is positioned past the principal whose privilege attributes have been returned in this call.

Output

status

A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_login_cred_get_delegate*( ) routine returns a handle of type **sec_login_handle_t** to the privilege attributes of an intermediary in a delegation chain that performed an authenticated RPC operation.

This call is used by clients. Servers use the *sec_cred_get_delegate*( ) routine to return the privilege attribute handle of an intermediary in a delegation chain.

The login context identified by *login_context* contains all members in the delegation chain. This call returns a handle (**sec_cred_pa_handle_t**) to the privilege attributes of one of the delegates in the login context. The **sec_cred_pa_handle_t** returned by this call is used in other *sec_cred_get_**( ) calls to obtain privilege attribute information for a single delegate.

To obtain the privilege attributes of each delegate in the credential handle identified by *callers_identity*, execute this call until the message **sec_cred_s_no_more_entries** is returned.

Before you execute *sec_login_cred_get_delegate*( ), you must execute a *sec_login_cred_init_cursor*( ) call to initialize a cursor of type **sec_cred_cursor_t**.

Use the *sec_cred_free_pa_handle*( ) and *sec_cred_free_cursor( )* calls to free the resources allocated to the **sec_cred_pa_handle_t** and *cursor*.

FILES

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

ERRORS

sec_cred_s_invalid_cursor

sec_cred_s_no_more_entries

error_status_ok

SEE ALSO

Functions: *sec_cred_get_deleg_restrictions*( ), *sec_cred_get_delegation_type*( ), *sec_cred_get_extended_attrs*( ), *sec_cred_get_opt_restrictions*( ), *sec_cred_get_pa_date*( ), *sec_cred_get_req_restrictions*( ), *sec_cred_get_tgt_restrictions*( ), *sec_cred_get_v1_pac*( ), *sec_login_cred_init_cursor*( ).

NAME

sec_login_cred_get_initiator — Returns information about the delegation initiator in a specified login context

SYNOPSIS

```
#include <dce/sec_login.h>

sec_cred_pa_handle_t sec_login_cred_get_initiator (
    sec_login_handle_t login_context,
    error_status_t *status );
```

PARAMETERS

Input

login_context
A value of **sec_login_handle_t** that provides an opaque handle to a login context for which delegation has been enabled.

Output

status
A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_login_cred_get_initiator*( ) routine returns a handle of type **sec_cred_pa_handle_t** to the privilege attributes of the delegation initiator.

The login context identified by *login_context* contains all members in the delegation chain. This call returns a handle (**sec_cred_pa_handle_t**) to the privilege attributes of the initiator. The **sec_cred_pa_handle_t** returned by this call is used in other *sec_cred_get_**( ) calls to obtain privilege attribute information for the initiator single delegate.

Use the *sec_cred_free_pa_handle*( ) call to free the resources allocated to the **sec_cred_pa_handle_t** handle.

FILES

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

ERRORS

sec_login_s_invalid_context

error_status_ok

SEE ALSO

Functions: *sec_cred_get_deleg_restrictions*( ), *sec_cred_get_delegation_type*( ), *sec_cred_get_extended_attrs*( ), *sec_cred_get_opt_restrictions*( ), *sec_cred_get_pa_date*( ), *sec_cred_get_req_restrictions*( ), *sec_cred_get_tgt_restrictions*( ), *sec_cred_get_v1_pac*( ).

NAME

sec_login_cred_init_cursor — Initializes a sec_cred_cursor_t

SYNOPSIS

```
#include <dce/sec_cred.h>

void sec_login_cred_init_cursor (
    sec_cred_cursor_t *cursor,
    error_status_t *status );
```

PARAMETERS

Input/Output

cursor
As input, a pointer to a **sec_cred_cursor_t** to be initialized. As output, a pointer to an initialized **sec_cred_cursor_t**.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_login_cred_init_cursor*( ) routine allocates and initializes a cursor of type **sec_cursor_t** for use with the *sec_login_cred_get_delegate*( ) call.

Use the *sec_cred_free_cursor*( ) call to free the resources allocated to *cursor*.

ERRORS

sec_cred_s_invalid_cursor

sec_login_s_no_memory

error_status_ok

SEE ALSO

Functions: *sec_login_cred_get_delegate*( ).

NAME

sec_login_disable_delegation — Disables delegation for a specified login context

SYNOPSIS

```
#include <dce/sec_login.h>

sec_login_handle_t *sec_login_disable_delegation (
    sec_login_handle_t login_context,
    error_status_t *status );
```

PARAMETERS

Input

login_context
An opaque handle to login context for which delegation has been enabled.

Output

status
A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_login_disable_delegation*( ) routine disables delegation for a specified login context. It returns a new login context of type **sec_login_handle_t** without any delegation information, thus preventing any further delegation.

FILES

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

ERRORS

sec_login_s_invalid_context

error_status_ok

SEE ALSO

Functions: *sec_login_become_delegate*( ), *sec_login_become_impersonator*( ), *sec_login_become_initiator*( ).

NAME

sec_login_export_context — Export a login context.

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_export_context (
    sec_login_handle_t login_context,
    unsigned32 count_max,
    idl_byte_t advertisement[],
    unsigned32 *count,
    unsigned32 *count_needed,
    error_status_t *status );
```

PARAMETERS

Input

login_context
The login context whose advertisement is to be created.

count_max
The maximum length (in bytes) to be returned in *advertisement*.

Input/Output

advertisement[ ]
Buffer (which is opaque; that is, whose structure and contents are implementation-dependent), of length at least *count_max*, to hold the advertisement of the login context.

Output

count
Number of bytes of *advertisement* actually occupied by the advertisement of the login context.

count_needed
If *count_max* is less than the length (in bytes) of the advertisement of the login context, the **sec_login_s_no_memory** *status* value is returned, and *count_needed* indicates the length of the advertisement.

status
The completion status.

DESCRIPTION

The *sec_login_export_context*( ) routine *exports* a login context; that is, creates an *advertisement* for it. Such an advertisement consists of information that can be communicated to other processes and enables them to (potentially) share the login context. Such sharing is restricted to processes on the local host. The advertisement can be communicated to other processes (on the local host) by any means desired by the communicating processes (it need not be a trusted communication path).

In typical implementations, the advertisement of a login context is simply the name of the login context's cache file (which is protected by local host security).

ERRORS

error_status_ok

sec_login_s_no_memory

sec_login_s_handle_invalid

sec_login_s_internal_error

sec_login_s_context_invalid

SEE ALSO

Functions: *sec_login_import_context*( ).

NAME

sec_login_free_net_info — Free memory allocated for network information.

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_free_net_info (
    sec_login_net_info_t *net_info );
```

PARAMETERS

Input/Output

net_info
The network information to be freed.

DESCRIPTION

The *sec_login_free_net_info*( ) routine frees memory allocated for a **sec_login_net_info_t** structure allocated by *sec_login_inquire_net_info*( ).

SEE ALSO

Functions: *sec_login_inquire_net_info*( ).

NAME

sec_login_get_current_context — Retrieve this process' current login context.

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_get_current_context (
    sec_login_handle_t *login_context,
    error_status_t *status );
```

PARAMETERS

Output

login_context
The retrieved current login context.

status
The completion status.

DESCRIPTION

The *sec_login_get_current_context*( ) routine retrieves a process' current login context.

In typical implementations, this routine returns the login context information contained in the cache file named by an (implementation-specific) well-known environment variable (typically, KRB5CCNAME).

ERRORS

error_status_ok

sec_login_s_internal_error

sec_login_s_no_current_context

SEE ALSO

Functions: *sec_login_set_context*( ).

NAME

sec_login_get_expiration — Retrieve the expiration date of a login context.

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_get_expiration (
    sec_login_handle_t login_context,
    unsigned32 *expiration_date,
    error_status_t *status );
```

PARAMETERS

Input

login_context
Login context whose expiration date is to be retrieved.

Output

expiration_date
The expiration date of the login context (measured in seconds from midnight January 1, 1970 UTC).

status
The completion status.

DESCRIPTION

The *sec_login_get_expiration*( ) routine retrieves the *expiration_date* of a (validated) login context, which is the date beyond which RPC binding handles annotated with the login context (in the sense of *rpc_binding_set_auth_info*( ) in the referenced X/Open DCE RPC Specification) will fail. (The RPC failure may occur at either RPC invocation time or at RPC return time, since both are authenticated — this fact is especially interesting in the case of long-lived RPC operations.)

ERRORS

error_status_ok

sec_login_s_context_invalid

sec_login_s_default_use

sec_login_s_internal_error

sec_login_s_no_current_context

sec_login_s_not_certified

SEE ALSO

Functions: *sec_login_refresh_identity*( ).

NAME

sec_login_get_groups — Retrieve (read) local host group membership information from a login context

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_get_groups (
    sec_login_handle_t login_context,
    unsigned32 *count,
    signed32 **groups,
    error_status_t *status );
```

PARAMETERS

Input

login_context
Login context from which group membership information is to be retrieved.

Output

count
Number of local groups in the array *groups*.

groups
The list of local groups indicated in the login context. (The datatype of *groups*, **unsigned32**, is intended to be converted to a host-specific datatype. For example, on POSIX-compliant operating systems, it is intended to be converted to the **gid_t** datatype.)

status
The completion status.

DESCRIPTION

The *sec_login_get_groups*( ) routine returns the local group information from a login context.

The routine works only on previously validated contexts.

ERRORS

error_status_ok

sec_login_s_context_invalid

sec_login_s_default_use

sec_login_s_info_not_avail

sec_login_s_not_certified

sec_rgy_object_not_found

sec_rgy_server_unavailable

SEE ALSO

Functions: *sec_rgy_acct_get_projlist*( ), *sec_login_get_pwent*( ).

NAME

sec_login_get_pwent — Retrieve local host information associated with a login context

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_get_pwent (
    sec_login_handle_t login_context,
    sec_login_passwd_t *pwent,
    error_status_t *status );
```

PARAMETERS

Input

login_context
Login context from which information is to be retrieved.

Output

pwent
The retrieved information.

status
The completion status.

DESCRIPTION

The *sec_login_get_pwent*( ) routine retrieves local host-specific information (represented by an implementation-specific **sec_login_passwd_t** structure) from a login context.

This routine works only on (validated) login contexts that are explicitly specified (that is, it doesn't work on the default login context indicated by NULL).

ERRORS

error_status_ok

sec_login_s_context_invalid

sec_login_s_default_use

sec_login_s_info_not_avail

sec_login_s_not_certified

sec_rgy_object_not_found

sec_rgy_server_unavailable

SEE ALSO

Functions: *sec_login_get_groups*( ).

NAME

sec_login_import_context — Import a login context

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_import_context (
    unsigned32 count,
    byte advertisement[ ],
    sec_login_handle_t *login_context,
    error_status_t *status );
```

PARAMETERS

Input

count
The length (in bytes) of the advertisement of the login context (contained in *advertisement*).

advertisement[ ]
The advertisement of the login context.

Output

login_context
The login context, created from its advertisement.

status
The completion status.

DESCRIPTION

The *sec_login_import_context*( ) routine *imports* a login context; that is, creates a login context from its advertisement.

In typical implementations, this routine reads the login context's cache file (whose name was contained in the login context's advertisement) into the calling process's address space.

ERRORS

error_status_ok

sec_login_s_context_invalid

sec_login_s_default_use

sec_login_s_internal_error

SEE ALSO

Functions: *sec_login_export_context*( ).

NAME

sec_login_init_first — Initialise process's default login context inheritance mechanism

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_init_first (
    error_status_t *status );
```

PARAMETERS

Output

status
The completion status.

DESCRIPTION

The *sec_login_init_first*( ) routine initialises the calling process' current login context inheritance mechanism, thereby making the calling process' current login context (potentially) accessible to other processes on the local host (especially those in the host's daemon process hierarchy).

In typical implementations, this routine merely records the name of a cache file (not yet populated — see *sec_login_setup_first*( )) which is to contain the host principal/account's login context in an (implementation-specific) well-known environment variable (typically, KRB5CCNAME), thereby marking it as the calling process' current login context. Child processes thus inherit the host's default login context, provided they have access privilege to the cache file, and provided the cache file is actually populated (by *sec_login_setup_first*( )).

If the default inheritance mechanism is already initialised, the operation fails.

Typically, this routine is called from a host's SCD (or from the host's initial process, sometimes called **init**) at boot time to initialise the current login context for inheritance by the host's hierarchy of daemon processes.

ERRORS

error_status_ok

sec_login_s_default_use

sec_login_s_privileged

SEE ALSO

Functions: *sec_login_setup_first*( ), *sec_login_validate_first*( ).

NAME

sec_login_inquire_net_info — Retrieve certain network information from login context

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_inquire_net_info (
    sec_login_handle_t login_context,
    sec_login_net_info_t *net_info,
    error_status_t *status );
```

PARAMETERS

Input

login_context
Login context from which network information is to be retrieved.

Output

net_info
The retrieved network information.

status
The completion status.

DESCRIPTION

The *sec_login_inquire_net_info*( ) routine returns certain network information (represented by the **sec_login_net_info_t** structure) from a login context.

The memory for *net_info* is dynamically allocated, and can be freed with *sec_login_free_net_info*( ).

ERRORS

error_status_ok

sec_login_s_auth_local

sec_login_s_context_invalid

sec_login_s_internal_error

sec_login_s_no_current_context

sec_login_s_not_certified

SEE ALSO

Functions: *sec_login_get_expiration*( ), *sec_login_free_net_info*( ).

NAME

sec_login_newgroups — Restrict group membership information of a login context

SYNOPSIS

```
#include <dce/sec_login.h>

boolean32 sec_login_newgroups (
    sec_login_handle_t login_context,
    sec_login_flags_t flags,
    unsigned32 count,
    sec_id_t groups[],
    sec_login_handle_t *restricted_login_context,
    error_status_t *status );
```

PARAMETERS

Input

login_context
Login context whose group membership information is to be changed.

flags
Flag word indicating attributes of the modified login context.

count
Number of local groups in the array *groups*.

groups[ ]
Array of groups to include in the modified login context.

Output

restricted_login_context
The restricted login context.

status
The completion status.

DESCRIPTION

The *sec_login_newgroups*( ) routine restricts the group membership information of a (validated) login context, to, effectively, the intersection of its existing group membership information and the information supplied in the *groups* array. Thus, *groups* can be viewed as the maximum group membership privilege that will be claimed by an RPC annotated (see *rpc_binding_set_auth_info*( )) with the restricted login context.

The restricted login context remains validated.

RETURN VALUES

This routine returns non-0 (TRUE) upon success, 0 (FALSE) upon failure.

ERRORS

error_status_ok

sec_login_s_auth_local

sec_login_s_default_use

sec_login_s_groupset_invalid

SEE ALSO

Functions: *sec_login_get_groups*( ).

NAME

sec_login_purge_context — Purge a login context

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_purge_context (
    sec_login_handle_t *login_context,
    error_status_t *status );
```

PARAMETERS

Input

login_context
Login context to be purged.

Output

status
The completion status.

DESCRIPTION

The *sec_login_purge_context*( ) routine purges a login context; that is, *unregisters* it in the sense of making it inaccessible to the calling process and to other processes on the local host.

In typical implementations, this routine frees local memory storage in the current address space allocated to the specified login context, and deletes the login context's on-disk cache file (first overwriting its contents with NULL bytes (that is, all bits reset to 0), to limit its exposure to compromise). (The login context remains accessible to those processes that had previously stored it in their address spaces, however.)

ERRORS

error_status_ok

sec_login_s_context_invalid

sec_login_s_default_use

SEE ALSO

Functions: *sec_login_set_context*( ), *sec_login_release_context*( ).

NAME

sec_login_purge_context_exp — Destroy expired network credentials

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_purge_context_exp (
    unsigned32 count,
    byte buf[ ],
    signed32 purge_time,
    error_status_t *status );
```

PARAMETERS

Input

count
Number of bytes in the *buf* array. This number specifies the length of *buf*.

buf[ ]
Buffer containing the expired network credentials.

purge_time
The time at which the credentials are to be purged. The credentials are purged if they have actually expired before this time.

Output

status
The completion status.

DESCRIPTION

The *sec_login_purbe_context_exp*( ) function purges a named set of network credentials that have expired before a specified time **t**, the *purge_time*.

FILES

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

ERRORS

error_status_ok
This function returned successfully. The expired network credentials were purged.

sec_login_s_default_use
The login context did not contain expired credentials. No credentials were purged.

SEE ALSO

Functions: *sec_login_purge_context*( ).

NAME

sec_login_refresh_identity — Refresh a login context

SYNOPSIS

```
#include <dce/sec_login.h>

boolean32 sec_login_refresh_identity (
    sec_login_handle_t login_context,
    error_status_t *status );
```

PARAMETERS

Input

login_context
Login context to be refreshed.

Output

status
The completion status.

DESCRIPTION

The *sec_login_refresh_identity*( ) routine refreshes a login context; that is, increases its expiration date to the maximum allowable (depending on local cell policy).

The refreshed login context reflects changes that may have been made to the principal/account's RS datastore information, but no other information associated with the login context will be modified (for example, any list of maximum group membership privilege set by *sec_login_newgroups*( ) remains in effect).

The refreshed login context is unvalidated, so it must be validated (with *sec_login_validate_identity* ( )) before it is usable.

It is an error to refresh a locally-authenticated context.

ERRORS

error_status_ok

sec_login_s_context_invalid

sec_login_s_default_use

sec_login_s_internal_error

sec_login_s_refresh_ident_bad

sec_rgy_object_not_found

sec_rgy_server_unavailable

SEE ALSO

Functions: *sec_login_get_expiration*( ).

NAME

sec_login_release_context — Release a login context

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_release_context (
    sec_login_handle_t *login_context,
    error_status_t *status );
```

PARAMETERS

Input/Output

login_context
Login context to be freed.

Output

status
The completion status.

DESCRIPTION

The *sec_login_release_context*( ) routine releases a login context; that is, frees the memory allocated to it. (This routine does not affect other processes' accessibility to the login context).

In typical implementations, this routine frees local memory storage (in the current address space) allocated to the specified login context, thereby making the login context inaccessible to the calling process (but it does not access the login context's cache file).

ERRORS

error_status_ok

sec_login_s_context_invalid

sec_login_s_default_use

SEE ALSO

Functions: *sec_login_setup_identity*( ), *sec_login_purge_context*( ).

NAME

sec_login_set_context — Set a login context (including making it current)

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_set_context (
    sec_login_handle_t login_context,
    error_status_t *status );
```

PARAMETERS

Input

login_context
The login context to be set.

Output

status
The completion status.

DESCRIPTION

The *sec_login_set_context*( ) routine sets a (validated) login context; that is, registers it in the sense of making it (potentially) accessible to other processes on the local host, and makes it the calling process' current login context.

In typical implementations, this routine writes the login context information to a file called the login context's (*credential*) *cache file* on the local host (this file contains this login context's information only), thereby making it (potentially) accessible to other processes (provided they know the name of this file and have access privilege to the file). This routine also records the name of this file in an (implementation-specific) well-known environment variable (typically, KRB5CCNAME), thereby marking it as the calling process' current login context, and (implicitly) passing the file's name to its child processes.

ERRORS

error_status_ok

sec_login_s_auth_local

sec_login_s_context_invalid

sec_login_s_default_use

sec_login_s_internal_error

SEE ALSO

Functions: *sec_login_get_current_context*( ).

NAME

sec_login_set_extended_attrs — Constructs a new login context that contains extended registry attributes

SYNOPSIS

```
#include <dce/sec_login.h>

sec_login_handle_t sec_login_set_extended_attrs (
    sec_login_handle_t my_login_context,
    unsigned32 num_attributes,
    sec_attr_t attributes[],
    error_status_t *status );
```

PARAMETERS

Input

my_login_context
A value of **sec_login_handle_t** that provides an opaque handle to the identity of the calling client.

num_attributes
An unsigned 32-bit integer that specifies the number of elements in the *attributes* array. The number must be greater than 0.

attributes[ ]
An array of values of type **sec_attr_t** that specifies the list of attributes to be set in the new login context.

Output

status
A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_login_set_extended_attrs*( ) constructs a login context that contains extended registry attributes that have been established for the object identified by *my_login_context*. The attributes themselves must have been established and attached to the object using the extended registry attribute API. The input *attributes*[ ] array of **sec_attr_t** values should specify the *attr_id* field for each requested attribute. Since the lookup is by attribute type ID only, set the *attributes***.attr_value.attr_encoding** field to **sec_attr_enc_void** for each attribute. Note that **sec_attr_t** is an extended registry attribute data type. You cannot use this call to add extended registry attributes to a delegation chain. If you pass in a login context that refers to a delegation chain, an invalid context error will be returned. The routine returns a new login context of type **sec_login_handle_t** that includes the attributes specified in the *attributes* array.

FILES

/usr/include/dce/sec_login.idl
The **idl** file from which **dce/sec_login.h** was derived.

ERRORS

sec_login_s_invalid_context

error_status_ok

SEE ALSO

Functions: *sec_login_become_impersonator*( ), *sec_login_set_context*( ), *sec_login_setup_identity*( ), *sec_login_validate_identity* ( ), *sec_rgy_attr_*( )calls.

NAME

sec_login_setup_first — Create local host's current login context

SYNOPSIS

```
#include <dce/sec_login.h>

boolean32 sec_login_setup_first (
   sec_login_handle_t *login_context,
   error_status_t *status );
```

PARAMETERS

Output

login_context
The (unvalidated) login context associated with the host's principal/account (**self**).

status
The completion status.

DESCRIPTION

The *sec_login_setup_first*( ) routine creates (see the *sec_login_setup_identity*( )routine) the local host's (unvalidated) current login context; that is, the login context associated with the host's principal/account (**self**).

If the host's current login context has previously been created (not necessarily validated) the routine fails.

Typically, this routine is called from a host's SCD (or from the host's initial process, sometimes called **init**), and inherited by the host's hierarchy of daemon processes.

This routine does not take a principal/account name as input (as does *sec_login_setup_identity*( )) — it determines the host principal/account's name in an implementation-dependent manner.

RETURN VALUES

The routine returns a non-0 (TRUE) upon success, and 0 (FALSE) upon failure.

ERRORS

error_status_ok

sec_login_s_config

sec_login_s_default_use

sec_login_s_no_current_context

sec_login_s_no_memory

sec_login_s_privileged

sec_rgy_object_not_found

sec_rgy_server_unavailable

SEE ALSO

Functions: *sec_login_init_first*( ), *sec_login_setup_identity*( ), *sec_login_validate_first*( ).

NAME

sec_login_setup_identity — Set up a login context for a principal/account

SYNOPSIS

```
#include <dce/sec_login.h>

boolean32 sec_login_setup_identity (
    unsigned_char_p_t principal,
    sec_login_flags_t flags,
    sec_login_handle_t *login_context,
    error_status_t *status );
```

PARAMETERS

Input

principal
Name of principal/account to which created login context is to refer. (Recall that an account is uniquely identified by its principal name component.)

flags
Flags indicating properties attributed to created login context.

Output

login_context
The created login context.

status
The completion status.

DESCRIPTION

The *sec_login_setup_identity*( ) routine sets up a login context; that is, creates, in the address space of the calling process, an (unvalidated and uncertified) login context for the specified principal/account.

A login context created by this routine is not usable for making protected RPCs (see *rpc_binding_set_auth_info*( )) until it has been validated, usually by *sec_login_validate_identity*( ) (see also *sec_login_valid_and_cert_identity*( )). In this sense, *sec_login_setup_identity*( ) and *sec_login_validate_identity*( ) are the two halves of a single logical operation: together they collect the data needed to establish a trusted, authenticated identity. (The rationale for making the routines independent is to make sure passwords are subjected to minimal exposure, such as sending them unprotected in a message, or retaining them in local memory for longer than absolutely necessary; for example, during a long networking delay that might be caused by an attacker.)

The memory storage for the created login context is dynamically allocated (the *sec_login_release_context*( ) function is used to free it).

In typical implementations, in the case of the Kerberos authentication service (the only authentication service currently supported by DCE), this routine calls a *kds_request*( ) RPC operation, returning a KDS Response message protected in the long-term key of the specified principal/account, which cannot therefore be used (much less trusted) until it has been decrypted (using *sec_login_validate_identity*( )). (Note that this routine only contacts the KDS (of the cell in which the specified principal/account is registered), not the PS — thus, it manipulates a TGT, not a PTGT.) An alternative implementation, in environments where holding the password in memory for a longer

period of time is not as large a threat, is to postpone the *kds_request*( ) call until *sec_login_validate_identity* ( ) is invoked.

RETURN VALUES

The routine returns non-0 (TRUE) if the login context has been successfully created, otherwise it returns 0 (FALSE). (In the success case, this return value is redundant with **error_status_ok**.)

ERRORS

error_status_ok

sec_login_s_no_memory

sec_login_s_internal_error

sec_rgy_object_not_found

sec_rgy_server_unavailable

SEE ALSO

Functions: *sec_login_release_context* ( ), *sec_login_validate_identity* ( ), *sec_login_valid_and_cert_identity* ( ).

Protocols: *kds_request*( ).

NAME

sec_login_tkt_request_options —

SYNOPSIS

```
#include <dce/sec_login.h>

void sec_login_tkt_request_options (
    sec_login_handle_t  login_context,
    sec_login_tkt_info_t *tkt_info,
    error_status_t *status );
```

PARAMETERS

Input

login_context
: A login context handle in the setup or refreshed state. The requested data is placed in the KRB_REQUEST_INFO portion of the login context upon successful completion of this function.

tkt_info
: (Pointer to) a structure which specifies the types of ticket options requested. If a renewable or postdated ticket is requested, or if a non-default ticket lifetime is requested, additional data must be provided in the respective field associated with the option. These fields are the **renewable_lifetime**, **postdated_dormanttime**, or **lifetime** fields of the **sec_login_tkt_info_t** structure, respectively.

Output

status
: The completion status.

AS ticket options
: The requested options as specified in the **options** and option-associated fields, respectively, of the **sec_login_tkt_info_t** structure.

DESCRIPTION

This function is used by a client to request specific AS ticket options. This optional function should be called after *sec_login_setup_identity*( ) or *sec_login_refresh_identity*( ) and before *sec_login_validate_identity*( ) or *sec_login_valid_and_cert_ident*( ).

Input should consist of a login context handle in the setup or refreshed state, and a structure which specifies the types of ticket options requested. If the user requests a renewable/postdated ticket, or a non-default ticket lifetime, additional data must be provided in the **renewable_lifetime**, **postdated_dormanttime**, and **lifetime** fields of the **sec_login_tkt_info_t** structure, respectively.

The data is placed in the KRB_REQUEST_INFO portion of the login context. These options will override the defaults when the ticket is requested at validation time.

FILES

/usr/include/dce/sec_login.idl
: The **idl** file from which **dce/sec_login.h** was derived.

ERRORS

error_status_ok

SEE ALSO

Functions: *sec_login_refresh_identity* ( ), *sec_login_setup_identity* ( ), *sec_login_validate_identity* ( ), *sec_login_valid_and_cert_ident* ( ).

NAME

sec_login_valid_and_cert_ident — Simultaneously validate and certify a login context

SYNOPSIS

```
#include <dce/sec_login.h>

boolean32 sec_login_valid_and_cert_ident (
    sec_login_handle_t login_context,
    sec_passwd_rec_t *passwd,
    boolean32 *reset_passwd,
    sec_login_auth_src_t *authn_src,
    error_status_t *status );
```

PARAMETERS

Input

login_context
Login context to be validated and certified.

Input/Output

passwd
Password record to be used to validate the login context.

Output

reset_passwd
Indicates whether a principal/account's password has expired.

authn_src
The source of validation (or authentication) of this login context.

status
The completion status.

DESCRIPTION

The *sec_login_valid_and_cert_ident*( ) routine validates and certifies a login context (logically combining the operations of *sec_login_validate_identity*( ) and *sec_login_certify_identity*( )), in a manner appropriate for use by privileged processes.

In typical implementations this is accomplished by impersonating the local host's SCD, which may be thought of as the local TCB invoking a protected RPC to itself, and is infallible (that is, completely secure, modulo the security of the local TCB). (See Section 1.15.2 on page 77 for details.)

Upon return, this operation destroys the contents of the input *passwd* parameter (that is, overwrites the actual password contained in it with NULL bytes — all bits reset to 0, in the caller's address space), thereby reducing its exposure to compromise).

If the network security service is unavailable, a local-host authenticated context is created, and the *authn_src* parameter is set to **sec_login_auth_src_local** (see the description of this in **<dce/sec_login.h>**).

RETURN VALUES

The routine returns non-0 (TRUE) if the login identity has been successfully validated and certified, 0 (FALSE) otherwise.

ERRORS

error_status_ok

sec_login_s_acct_invalid

sec_login_s_already_valid

sec_login_s_default_use

sec_login_s_null_password

sec_login_s_privileged

sec_login_s_unsupp_passwd_type

sec_rgy_passwd_invalid

sec_rgy_server_unavailable

SEE ALSO

Functions: *sec_login_certify_identity* ( ), *sec_login_validate_identity* ( ).

NAME

sec_login_validate_first — Validate host's current login context

SYNOPSIS

```
#include <dce/sec_login.h>

boolean32 sec_login_validate_first (
    sec_login_handle_t login_context,
    boolean32 *reset_passwd,
    sec_login_auth_src_t *authn_src,
    error_status_t *status );
```

PARAMETERS

Input

login_context
Login context to be validated.

Output

reset_passwd
Indicates whether a principal/account's password has expired.

authn_src
The source of validation (or authentication) of this login context.

status
The completion status.

DESCRIPTION

The *sec_login_validate_first*( ) routine validates (see *sec_login_validate_identity*( )) the calling process's current login context, which must be the host's login context (set up, for example, by *sec_login_setup_first*( )).

Typically, this routine is called from a host's SCD (or from the host's initial process, sometimes called **init**), to validate the host's login context for the host's hierarchy of daemon processes. This routine does not have a password parameter (as does *sec_login_validate_identity*( )) — implementations typically manage the host principal/account's key with the **sec_key_mgmt** API.

RETURN VALUES

This routine returns non-0 (TRUE) if the validation was successful, and 0 (FALSE) otherwise.

ERRORS

error_status_ok

sec_login_s_privileged

sec_rgy_server_unavailable

SEE ALSO

Functions: *sec_login_init_first*( ), *sec_login_setup_first*( ), *sec_login_validate_identity* ( ).

NAME

sec_login_validate_identity — Validate a login context

SYNOPSIS

```
#include <dce/sec_login.h>

boolean32 sec_login_validate_identity (
    sec_login_handle_t login_context,
    sec_passwd_rec_t *passwd,
    boolean32 *reset_passwd,
    sec_login_auth_src_t *authn_src,
    error_status_t *status );
```

PARAMETERS

Input

login_context
Login context to be validated.

Input/Output

passwd
Password record to be used to validate the login context.

Output

reset_passwd
Indicates whether a principal/account's password has expired.

authn_src
The source of validation (or authentication) of this login context.

status
The completion status.

DESCRIPTION

The *sec_login_validate_identity*( ) routine validates a login context; that is, makes it usable for making protected RPCs (in the sense of making it usable by *rpc_binding_set_auth_info*( )), and in the process demonstrates its trustworthiness (for use in protected RPCs) to the principal/account to which it is associated (under the assumption that the long-term key of the principal/account associated with the login context is uncompromised).

Upon return, this operation destroys the contents of the input *passwd* parameter (that is, overwrites the actual password contained in it with NULL bytes — all bits reset to 0, in the caller's address space — thereby reducing its exposure to compromise).

In typical usage, validation is accomplished by decrypting the encrypted part of the login context as obtained from *sec_login_setup_identity*( ) (and verifying that the decryption is correct), using the long-term key of the principal/account — hence, this information must have been encrypted by an entity knowing the principal/account's long-term key, which must have been an entity trusted by the caller. This routine also typically contacts the PS (of the cell in which the principal/account associated with the login context is registered), gets a PTGT for the principal/account, and decrypts the encrypted part of it. Thus, a validated login context typically contains *both* a TGT and a PTGT for the local cell (as well as other information).

If *reset_passwd* returns non-0 (TRUE), then the account's password has expired. Otherwise, *reset_password* returns 0 (FALSE).

RETURN VALUES

The routine returns non-0 (TRUE) if the login context has been successfully validated. Otherwise, it returns 0 (FALSE). (In the success case, this return value is redundant with **error_status_ok**.)

ERRORS

error_status_ok

sec_login_s_acct_invalid

sec_login_s_already_valid

sec_login_s_default_use

sec_login_s_null_password

sec_login_s_unsupp_passwd_type

sec_rgy_passwd_invalid

sec_rgy_server_unavailable

SEE ALSO

Functions: *sec_login_certify_identity* ( ), *sec_login_setup_identity* ( ), *sec_login_valid_and_cert_ident* ( ).

Chapter 20

EPAC Accessor Function (sec_cred) API

20.1 Introduction

The routines in the sec_cred API are distinguished with names having the prefix **sec_cred_**.

In this API, the status **error_status_ok** has the value 0 and indicates successful completion of the function the routine was called to perform. In the few instances where successful completion is indicated by other than **error_status_ok**, this other status will be explicitely called out. For this version of DCE, there is one routine that returns successful completion other than **error_status_ok**. This routine is *sec_cred_is_authenticated*( ).

Background is given in <REFERENCE UNDEFINED>(EPAC-Accessor) and Section 5.2.14 on page 288.

NAME

sec_cred_free_attr_cursor — Frees the local resources allocated to a **sec_attr_cursor_t**

SYNOPSIS

```
#include <dce/sec_cred.h>

void sec_cred_free_attr_cursor (
    sec_cred_attr_cursor_t *cursor,
    error_status_t *status );
```

PARAMETERS

Input/Output

cursor

As input, a pointer to a **sec_cred_attr_cursor_t** whose resources are to be freed. As output, a pointer to an initialized **sec_cred_attr_cursor_t** with allocated resources freed.

Output

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_cred_free_attr_cursor*( ) routine frees the resources assoicated with a cursor of type **sec_cred_attr_cursor_t** used by the *sec_cred_get_extended_attrs*( ) call.

FILES

/usr/include/dce/sec_cred.idl

The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

error_status_ok

SEE ALSO

Functions: *sec_cred_get_extended_attrs*( ), *sec_cred_initialize_attr_cursor*( ).

NAME

sec_cred_free_cursor — Releases local resources allocated to a **sec_cred_cursor_t**

SYNOPSIS

```
#include <dce/sec_cred.h>

void sec_cred_free_cursor (
    sec_cred_cursor_t *cursor,
    error_status_t *status );
```

PARAMETERS

Input/Output

cursor
As input, a **sec_cred_cursor_t** whose resources are to be freed. As output, a **sec_cred_cursor_t** whose resources are freed.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_cred_free_cursor*( ) routine releases local resources allocated to a **sec_cred_cursor_t** used by the *sec_cred_get_delegate*( ) call.

FILES

/usr/include/dce/sec_cred.idl
The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_login_s_no_memory

error_status_ok

SEE ALSO

Functions: *sec_cred_get_delegate*( ), *sec_cred_initialize_cursor*( ).

NAME

sec_cred_free_pa_handle — Frees the local resources allocated to a privilege attribute handle of type **sec_cred_pa_handle_t**

SYNOPSIS

```
#include <dce/sec_cred.h>

void sec_cred_free_pa_handle (
    sec_cred_pa_handle__t *pa_handle,
    error_status_t *status );
```

PARAMETERS

Input/Output

pa_handle
: As input, a pointer to a **sec_cred_pa_handle_t** whose resources are to be freed. As output, a pointer to a **sec_cred_pa_handle_t** with allocated resources freed.

Output

status
: A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_cred_free_pa_handle*( ) routine frees the resources assoicated with a privilege attribute handle of type **sec_cred_pa_handle_t** used by the *sec_cred_get_initiator*( ) and *sec_cred_get_delegate*( ) calls.

FILES

/usr/include/dce/sec_cred.idl
: The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

error_status_ok

SEE ALSO

Functions: *sec_cred_get_delegate*( ), *sec_cred_get_initiator*( ).

NAME

sec_cred_get_authz_session_info — Returns session-specific information that represents an authenticated client's credentials

SYNOPSIS

```
#include <dce/sec_cred.h>

void sec_cred_get_authz_session_info (
    rpc_authz_cred_handle_t callers_identity,
    uuid_t *session_id,
    sec_timeval_t *session_expiration,
    error_status_t *status );
```

PARAMETERS

Input

callers_identity
A credential handle of type **rpc_authz_cred_handle_t**. This handle is supplied as output of the *rpc_binding_inq_auth_caller*( ) call.

Output

session_ID
A pointer to a **uuid_t** that identifies the client's DCE authorization session.

session_expiration
A pointer to a **sec_timeval_t** that specifies the expiration time of the authenticated client's credentials.

status
A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_cred_get_authz_session_info*( ) routine retrieves session-specific information that represents the credentials of authenticated client specified by *callers_identity*. If the client is a member of a delegation chain, the information represents the credentials of all members of the chain.

The information can aid application servers in the construction of identity-based caches. For example, it could be used as a key into a cache of previously allocated delegation contexts and thus avoid the overhead of allocating a new login context on every remote operation. It could also be used as a key into a table of previously computed authorization decisions.

Before you execute this call, you must execute an *rpc_binding_inq_auth_caller*( ) call to obtain an **rpc_authz_cred_handle_t** for the *callers_identity* argument.

FILES

/usr/include/dce/sec_cred.idl
The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_cred_s_authz_cannot_comply

error_status_ok

SEE ALSO

Functions: *rpc_binding_inq_auth_caller*( ).

NAME

sec_cred_get_client_princ_name — Returns the principal name associated with a credential handle

SYNOPSIS

```
#include <dce/sec_cred.h>

void sec_cred_get_client_princ_name (
    rpc_authz_cred_handle_t callers_identity,
    unsigned_char_p_t *client_princ_name,
    error_status_t *status );
```

PARAMETERS

Input

callers_identity
A handle of type **rpc_authz_cred_handle_t** to the credentials for which to return the principal name. This handle is supplied as output of the *rpc_binding_inq_auth_caller*( ) call.

Output

client_princ_name
A pointer to the principal name of the server's RPC client.

status
A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_cred_get_client_princ_name*( ) routine extracts the principal name associated with the credentials identified by *callers_identity*.

Before you execute *sec_cred_get_client_princ_name*( ), you must execute an *rpc_binding_inq_auth_caller*( ) call to obtain an **rpc_authz_cred_handle_t** for the *callers_identity* argument.

FILES

/usr/include/dce/sec_cred.idl
The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_cred_s_authz_cannot_comply

error_status_ok

SEE ALSO

Functions: *rpc_binding_inq_auth_caller*( ), *rpc_string_free*( ).

NAME

sec_cred_get_deleg_restrictions — Returns delegate restrictions from a privilege attribute handle

SYNOPSIS

```
#include <dce/sec_cred.h>

sec_id_restriction_set_t *sec_cred_get_deleg_restrictions (
    sec_cred_pa_handle_t callers_pas,
    error_status_t *status );
```

PARAMETERS

Input

callers_pas

A value of type **sec_cred_pa_handle_t** that provides a handle to a principal's privilege attributes. This handle is supplied as output of the *sec_cred_get_initiator*( ) call, the *sec_cred_get_delegate*( ) call and the *sec_login_cred_**( ) calls.

Output

status

A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

DESCRIPTION

The *sec_cred_get_deleg_restrictions*( ) routine extracts delegate restrictions from the privilege attribute handle identified by *callers_pas*. The restrictions are returned in a **sec_id_restriction_set_t**.

Before you execute *sec_cred_get_pa_data*( ), you must execute a *sec_cred_get_initiator*( ) or *sec_cred_get_delegate*( ) call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* argument.

FILES

/usr/include/dce/sec_cred.idl

The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_cred_s_invalid_pa_handle

error_status_ok

SEE ALSO

Functions: *sec_cred_get_delegate*( ), *sec_cred_get_initiator*( ).

NAME

sec_cred_get_delegate — Returns a handle to the privilege attributes of an intermediary in a delegation chain

SYNOPSIS

```
#include <dce/sec_cred.h>

sec_cred_pa_handle_t sec_cred_get_delegate (
    rpc_authz_cred_handle_t callers_identity,
    sec_cred_cursor_t *cursor,
    error_status_t *status );
```

PARAMETERS

Input

callers_identity

A handle of type **rpc_authz_cred_handle_t**. This handle is supplied as output of the *rpc_binding_inq_auth_caller*( ) call.

Input/Output

cursor

As input, a pointer to a cursor of type **sec_cred_cursor_t** that has been initialized by the *sec_cred_initialize_cursor*( ) call. As an output argument, *cursor* is a pointer to a cursor of type **sec_attr_srch_cursor_t** that is positioned past the principal whose privilege attributes have been returned in this call.

Output

status

A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

DESCRIPTION

The *sec_cred_get_delegate*( ) routine returns a handle to the the privilege attributes of an intermediary in a delegation chain that performed an authenticated RPC operation.

This call is used by servers. Clients use the *sec_login_cred_get_delegate*( ) routine to return the privilege attribute handle of an intermediary in a delegation chain.

The credential handle identified by *callers_identity* contains authentication and authorization information for all delegates in the chain. This call returns a handle (**sec_cred_pa_handle_t**) to the privilege attributes of one of the delegates in the binding handle. The **sec_cred_pa_handle_t** returned by this call is used in other *sec_cred_get_**( ) calls to obtain privilege attribute information for a single delegate.

To obtain the privilege attributes of each delegate in the credential handle identified by *callers_identity*, execute this call until the message **sec_cred_s_no_more_entries** is returned.

Before you execute *sec_cred_get_delegate*( ), you must execute

An *rpc_binding_inq_auth_caller*( ) call to obtain an **rpc_authz_cred_handle_t** for the *callers_identity* argument.

A *sec_cred_initialize_cursor*( ) call to initialize a cursor of type **sec_cred_cursor_t**.

Use the *sec_cred_free_pa_handle*( ) call to free the resources associated with the **sec_cred_pa_handle_t**.

FILES

/usr/include/dce/sec_cred.idl
The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_cred_s_invalid_auth_handle

sec_cred_s_invalid_cursor

sec_cred_s_no_more_entries

error_status_ok

SEE ALSO

Functions: *rpc_binding_inq_auth_caller*( ), *sec_cred_free_pa_handle*( ), *sec_cred_get_deleg_restrictions*( ), *sec_cred_get_delegation_typ*( ), *sec_cred_get_extended_attrs*( ), *sec_cred_get_opt_restrictions*( ), *sec_cred_get_pa_date*( ), *sec_cred_get_req_restrictions*( ), *sec_cred_get_tgt_restrictions*( ), *sec_cred_get_v1_pac*( ), *sec_cred_initialize_cursor*( ).

NAME

sec_cred_get_delegation_type — Returns the delegation type from a privilege attribute handle

SYNOPSIS

```
#include <dce/sec_cred.h>

sec_id_delegation_type_t *sec_cred_get_delegation_type (
    sec_cred_pa_handle_t callers_pas,
    error_status_t *status );
```

PARAMETERS

Input

callers_pas

A value of type **sec_cred_pa_handle_t** that provides a handle to a principal's privilege attributes. This handle is supplied as output of either the *sec_cred_get_initiator*( ) call or *sec_cred_get_delegate*( ) call.

Output

status

A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

DESCRIPTION

The *sec_cred_get_delegation_type*( ) routine extracts the delegation type from the privilege attribute handle identified by *callers_pas* and returns it in a **sec_id_delegation_type_t**.

Before you execute *sec_cred_get_delegation_type*( ), you must execute a *sec_cred_get_initiator*( ) or *sec_cred_get_delegate*( ) call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* argument.

FILES

/usr/include/dce/sec_cred.idl

The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_cred_s_invalid_pa_handle

error_status_ok

SEE ALSO

Functions: *sec_cred_get_delegate*( ), *sec_cred_get_initiator*( ).

NAME

sec_cred_get_extended_attrs — Returns extended attributes from a privilege handle

SYNOPSIS

```
#include <dce/sec_cred.h>

void sec_cred_get_extended_attrs (
    sec_cred_pa_handle_t callers_pas,
    sec_cred_attr_cursor_t *cursor
    sec_attr_t *attr
    error_status_t *status );
```

PARAMETERS

Input

callers_pas

A handle of type **sec_cred_pa_handle_t** to the caller's privilege attributes. This handle is supplied as output of either the *sec_cred_get_initiator*( ) call or *sec_cred_get_delegate*( ) call.

Input/Output

cursor

A cursor of type **sec_cred_attr_cursor_t** that has been initialized by the *sec_cred_initialize_attr_cursor*( ) routine. As input, *cursor* must be initialized. As output, *cursor* is positioned at the first attribute after the returned attribute.

Output

attr

A pointer to a value of **sec_attr_t** that contains extended registry attributes.

status

A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

DESCRIPTION

The *sec_cred_get_extended_attrs*( ) routine extracts extended registry attributes initialized from the privilege attribute handle identified by *callers_pas*.

Before you execute call, you must execute:

A *sec_cred_get_initiator*( ) or *sec_cred_get_delegate*( ) call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* argument.

A *sec_cred_initialize_attr_cursor*( ) to initialize a **sec_attr_t**.

To obtain all the extended registry attributes in the privilege attribute handle, repeat *sec_cred_get_extended_attrs*( ) calls until the status message **no_more_entries_available** is returned.

FILES

/usr/include/dce/sec_cred.idl

The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_cred_s_invalid_pa_handle

sec_cred_s_invalid_cursor

sec_cred_s_no_more_entries

error_status_ok

SEE ALSO

Functions: *sec_cred_get_initiator*( ), *sec_cred_get_delegate*( ), *sec_cred_initialize_attr_cursor*( ).

NAME

sec_cred_get_initiator — Returns the privilege attributes of the initiator of a delegation chain

SYNOPSIS

```
#include <dce/sec_cred.h>

sec_cred_pa_handle_t sec_cred_get_initiator (
    rpc_authz_cred_handle_t callers_identity,
    error_status_t *status );
```

PARAMETERS

Input

callers_identity
A credential handle of type **rpc_authz_cred_handle_t**. This handle is supplied as output of the *rpc_binding_inq_auth_caller*( ) call.

Output

status
A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

DESCRIPTION

The *sec_cred_get_initiator*( ) routine returns a handle to the the privilege attributes of the initiator of a delegation chain that performed an authenticated RPC operation.

The credential handle identified by *callers_identity* contains authentication and authorization information for all delegates in the chain. This call returns a handle (**sec_cred_pa_handle_t**) to the privilege attributes of the client that initiated the delegation chain. The **sec_cred_pa_handle_t** returned by this call is used in other *sec_cred_get_**( ) calls to obtain privilege attribute information for the initiator.

Before you execute *sec_cred_get_initiator*( ), you must execute an *rpc_binding_inq_auth_caller*( ) call to obtain an **rpc_authz_cred_handle_t** for the *callers_identity* argument.

FILES

/usr/include/dce/sec_cred.idl
The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_cred_s_invalid_auth_handle

error_status_ok

SEE ALSO

Functions: *rpc_binding_inq_auth_caller*( ), *sec_cred_get_deleg_restrictions*( ), *sec_cred_get_delegation_type*( ), *sec_cred_get_extended_attrs*( ), *sec_cred_get_opt_restrictions*( ), *sec_cred_get_pa_date*( ), *sec_cred_get_req_restrictions*( ), *sec_cred_get_tgt_restrictions*( ), *sec_cred_get_v1_pac*( ).

NAME

sec_cred_get_opt_restrictions — Returns optional restrictions from a privilege handle

SYNOPSIS

```
#include <dce/sec_cred.h>

sec_id_opt_req_t *sec_cred_get_opt_restrictions (
    sec_cred_pa_handle_t callers_pas,
    error_status_t *status );
```

PARAMETERS

Input

callers_pas

A handle of type **sec_cred_pa_handle_t** to a principal's privilege attributes. This handle is supplied as output of either the **sec_cred_get_initiator**( ) call or **sec_cred_get_delegate**( ) call.

Output

status

A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

DESCRIPTION

The *sec_cred_get_opt_restrictions*( ) routine extracts optional restrictions from the privilege attribute handle identified by *callers_pas* and returns them in a **sec_id_restriction_set_t**.

Before you execute *sec_cred_get_pa_data*( ), you must execute a *sec_cred_get_initiator*( ) or *sec_cred_get_delegate*( ) call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* argument.

FILES

/usr/include/dce/sec_cred.idl

The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_cred_s_invalid_pa_handle

error_status_ok

SEE ALSO

Functions: *sec_cred_get_delegate*( ), *sec_cred_get_initiator*( ).

NAME

sec_cred_get_pa_data — Returns identity information from a privilege attribute handle

SYNOPSIS

```
#include <dce/sec_cred.h>

sec_id_pa_t *sec_cred_get_pa_data (
    sec_cred_pa_handle_t callers_pas,
    error_status_t *status );
```

PARAMETERS

Input

callers_pas

A handle of type **sec_cred_pa_handle_t** to a principal's privilege attributes. This handle is supplied as output of either the *sec_cred_get_initiator*( ) call or *sec_cred_get_delegate*( ) call.

Output

status

A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

DESCRIPTION

The *sec_cred_get_pa_data*( ) routine extracts identity information from the privilege attribute handle specified by *callers_pas* and returns it in a **sec_id_pa_t**. The identity information includes an identifier of the princpal's locall cell and the principal's local and foreign group sets.

Before you execute *sec_cred_get_pa_data*( ), you must execute a *sec_cred_get_initiator*( ) or *sec_cred_get_delegate*( ) call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* argument.

FILES

/usr/include/dce/sec_cred.idl

The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_cred_s_invalid_pa_handle

error_status_ok

SEE ALSO

Functions: *sec_cred_get_delegate*( ), *sec_cred_get_initiator*( ).

NAME

sec_cred_get_req_restrictions — Returns required restrictions from a privilege attribute handle

SYNOPSIS

```
#include <dce/sec_cred.h>

sec_id_opt_req_t *sec_cred_get_req_restrictions (
    sec_cred_pa_handle_t callers_pas,
    error_status_t *status );
```

PARAMETERS

Input

callers_pas

A handle of type **sec_cred_pa_handle_t** to a principal's privilege attributes. This handle is supplied as output of either the *sec_cred_get_initiator*( ) call or *sec_cred_get_delegate*( ) call.

Output

status

A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

DESCRIPTION

The *sec_cred_get_req_restrictions*( ) routine extracts required restrictions from the privilege attribute handle identified by *callers_pas* and returns them in a **sec_id_opt_req_t**.

Before you execute *sec_cred_get_req_restrictions*( ), you must execute a *sec_cred_get_initiator*( ) or *sec_cred_get_delegate*( ) call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* argument.

FILES

/usr/include/dce/sec_cred.idl

The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_cred_s_invalid_pa_handle

error_status_ok

SEE ALSO

Functions: *sec_cred_get_delegate*( ), *sec_cred_get_initiator*( ).

NAME

sec_cred_get_tgt_restrictions — Returns target restrictions from a privilege attribute handle

SYNOPSIS

```
#include <dce/sec_cred.h>

sec_id_restriction_set_t *sec_cred_get_tgt_restrictions (
    sec_cred_pa_handle_t callers_pas,
    error_status_t *status );
```

PARAMETERS

Input

callers_pas

A handle of type **sec_cred_pa_handle_t** to a principal's privilege attributes. This handle is supplied as output of either the *sec_cred_get_initiator*( ) call or *sec_cred_get_delegate*( ) call.

Output

status

A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

DESCRIPTION

The *sec_cred_get_tgt_restrictions*( ) routine extracts target restrictions from the privilege attribute handle identified by *callers_pas* and returns them in a **sec_id_restriction_set_t**.

Before you execute *sec_cred_get_tgt_restrictions*( ), you must execute a *sec_cred_get_initiator*( ) or *sec_cred_get_delegate*( ) call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* argument.

FILES

/usr/include/dce/sec_cred.idl

The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_cred_s_invalid_pa_handle

error_status_ok

SEE ALSO

Functions: *sec_cred_get_delegate*( ), *sec_cred_get_initiator*( ).

NAME

sec_cred_get_v1_pac — Returns pre-1.1 PAC from a privilege attribute handle

SYNOPSIS

```
#include <dce/sec_cred.h>

sec_id_pac_t *sec_cred_get_v1_pac (
    sec_cred_pa_handle_t callers_pas,
    error_status_t *status );
```

PARAMETERS

Input

callers_pas
A handle of type **sec_cred_pa_handle_t** to the principal's privilege attributes. This handle is supplied as output of either the *sec_cred_get_initiator*( ) call or *sec_cred_get_delegate*( ) call.

Output

status
A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**.

DESCRIPTION

The *sec_cred_get_v1_pac*( ) routine extracts the privilege attributes from a pre-1.1 PAC for the privilege attribute handle specified by *callers_pas* and returns them in a **sec_id_pa_t**.

Before you execute *sec_cred_get_v1_pac*( ), you must execute a *sec_cred_get_initiator*( ) or *sec_cred_get_delegate*( ) call to obtain a **sec_cred_pa_handle_t** for the *callers_pas* argument.

FILES

/usr/include/dce/sec_cred.idl
The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_cred_s_invalid_pa_handle

error_status_ok

SEE ALSO

Functions: *sec_cred_get_delegate*( ), *sec_cred_get_initiator*( ).

NAME

sec_cred_initialize_attr_cursor — Initializes a **sec_attr_cursor_t**

SYNOPSIS

```
#include <dce/sec_cred.h>

void sec_cred_initialize_attr_cursor (
    sec_cred_attr_cursor_t *cursor,
    error_status_t *status );
```

PARAMETERS

Input/Output

cursor

As input, a pointer to a **sec_cred_attr_cursor_t** to be initialized. As output, a pointer to an initialized **sec_cred_attr_cursor_t**.

Output

status

A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_cred_initialize_attr_cursor*( ) routine allocates and initializes a cursor of type **sec_cred_attr_cursor_t** for use with the *sec_cred_get_extended_attrs*( ) call. Use the *sec_cred_free_attr_cursor*( ) call to free the resources allocated to *cursor*.

FILES

/usr/include/dce/sec_cred.idl

The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_login_s_no_memory

error_status_ok

SEE ALSO

Functions: *sec_cred_free_attr_cursor*( ), *sec_cred_get_extended_attrs*( ).

NAME

sec_cred_initialize_cursor — Initializes a **sec_cred_cursor_t**

SYNOPSIS

```
#include <dce/sec_cred.h>

void sec_cred_initialize_cursor (
    sec_cred_cursor_t *cursor,
    error_status_t *status );
```

PARAMETERS

Input/Output

cursor
As input, a **sec_cred_cursor_t** to be initialized. As output, an initialized **sec_cred_cursor_t**.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_cred_initialize_cursor*( ) routine initializes a cursor of type **sec_cursor_t** for use with the *sec_cred_get_delegate*( ) call. Use the *sec_cred_free_cursor*( ) call to free the resources allocated to *cursor*.

FILES

/usr/include/dce/sec_cred.idl
The **idl** file from which **dce/sec_cred.h** was derived.

ERRORS

sec_login_s_no_memory

error_status_ok

SEE ALSO

Functions: *sec_cred_free_cursor*( ), *sec_cred_get_delegate*( ).

NAME

sec_cred_is_authenticated — Returns true if the supplied credentials are authenticated and false if they are not

SYNOPSIS

```
#include <dce/sec_cred.h>

boolean32 sec_cred_is_authenticated (
    rpc_authz_cred_handle_t callers_identity,
    error_status_t *status );
```

PARAMETERS

Input

callers_identity

A handle of type **rpc_authz_cred_handle_t** to the credentials to check for authentication. This handle is supplied as output of the *rpc_binding_inq_auth_caller*( ) call.

Output

status

A pointer to the completion status. On successful completion, *status* is assigned **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *sec_cred_is_authenticated*( ) routine returns true if the credentials identified by *callers_identity* are authenticated or false if they are not.

Before you execute this call, you must execute an *rpc_binding_inq_auth_caller*( ) call to obtain an **rpc_authz_cred_handle_t** for the *callers_identity* argument.

FILES

/usr/include/dce/sec_cred.idl

The **idl** file from which **dce/sec_cred.h** was derived.

RETURN VALUES

The routine returns TRUE if the credentials are authenticated; FALSE if they are not.

SEE ALSO

Functions: *rpc_binding_inq_auth_caller*( ).

Chapter 21

Miscellaneous Routines Needed for DCE Security

21.1 Introduction

The routines in the this API are miscellaneous routines needed for DCE Security Services.

In this API, the status **error_status_ok** has the value 0 and indicates successful completion of the function the routine was called to perform.

NAME

rs_ns_entry_validate — Validate that this server can use "name" as its nameservice entry

SYNOPSIS

```
void rs_ns_entry_validate (
    unsigned_char_p_t       name,
    uuid_p_t                cell_sec_id,
    uuid_p_t                rep_id,
    rpc_binding_vector_p_t  svr_bindings,
    rpc_if_handle_t         ifspec,
    error_status_t          *status
);
```

PARAMETERS

Input/Output

name
The string name associated with this registry database entry

cell_sec_id
The well-known cell security id (of this cell)

rep_id
The replica id for this entry

svr_bindings
The server binding tower associated with this entry

ifspec
The interface specification for this entry

status
Completion status.

Output

status
A pointer to the completion status. On successful completion, the routine returns **error_status_ok**. Otherwise, it returns an error.

DESCRIPTION

The *rs_ns_entry_validate*( ) routine gets the nameservice entry's data associated with *name*. This data consists of the server bindings and replica id associated with *name*, cell's security id (*cell_sec_id*), and interface specification (*ifspec*).

If there is no entry associated with *name* (the nameservice entry does not exist), then this routine returns with **error_status_ok**. If an entry exists, then it is checked to verify that it isn't someone else's entry (the bindings for the replica id, *rep_id*, are checked to ensure they are the bindings expected). If this is the correct entry, this routine returns with **error_status_ok**.

If the entry associated with *name* is does not have the expected bindings (it is someone else's

entry), then this routine returns with status (non-zero) not equal **error_status_ok**.

RETURN VALUES

The routine returns **error_status_ok** if it is okay for the nameservice to use "name" as its nameservice entry. It also returns **error_status_ok** if the entry ("name") does not exist.

ERRORS

!error_status_ok

CAE Specification

Part 4
Appendices

The Open Group

Appendix A

Symbol Mapping Table

Note: This appendix is informative, not normative. It imposes no restrictions on conforming implementations.

The table below is a ‘‘symbol mapping table’’, correlating symbols employed in this specification with symbols occurring in the source code of the standard OSF reference implementation of DCE. The symbols occurring in the reference implementation are familiar to most DCE developers, but they were not chosen with an English-language specification document (such as this specification) in mind. For example, the DCE symbol *kds_request*( ) is more ‘‘English-friendly’’ than the reference implementation’s symbol *rsec_krb5rpc_sendto_kdc*( ). This table is included solely as an aid to developers who desire to compare their implementation with this specification — it does not impose any restrictions on conforming implementations.

In the table, indentation indicates either:

1. fields in a data structure, or
2. parameters in an operation signature.

Not every symbol in this specification has been listed — where symbols differ only trivially and cause no confusion to a reader, no note is made of them. On the other hand, wherever one field / parameter is listed due to a non-trivial difference in the symbols, all fields / parameters that differ (even trivially) for that structure / operation are listed.

This Specification	OSF DCE Reference Implementation
`scd_protected_noop()`	`sec_login_validate_cert_auth()`
`rpc_mgmt_set_authorization_fcn()`	`rpc_mgmt_set_authorization_fn()`
`rpc_syntax_id_t`	
`stx_id`	`id`
`stx_version`	`version`
`idl_pkl_header_t`	[doesn’t exist — idl_pkl_header_t is merely a conceptual representation of data that’s stored as a byte stream in an idl_pkl_t]
`kds_request()`	`rsec_krb5rpc_sendto_kdc()`
`rpc_handle`	`h`
`request_count`	`len`
`request`	`message`
`response_count_max`	`out_buf_len`
`response_count`	`resp_len`
`response`	`out_buf`
`ps_message_t`	`rpriv_pickle_t`

This Specification	OSF DCE Reference Implementation
ps_request()	rpriv_get_ptgt()
rpc_handle	handle
authn_service	authn_svc
authz_service	authz_svc
request	ptgt_req
response	ptgt_rep
ps_c_authn_secret	rpc_c_authn_dce_secret
ps_c_authz_dce	rpc_c_authz_dce
sec_id_foreign_t	
cell	realm
sec_id_foreign_groupset_t	
cell	realm
count_local_groups	num_groups
local_groups	groups
sec_bytes_t	
count_bytes	num_bytes
sec_id_pac_t	
pac_format	pac_type
cell	realm
primary_group	group
count_local_groups	num_groups
count_foreign_groups	num_foreign_groups
local_groups	groups
sec_acl_entry_t	
local_id	id
sec_acl_t	
default_cell	default_realm
count	num_entries
sec_acl_printstring_t	
perm	permissions
rdacl_*()	
rpc_handle	h
acl_type	sec_acl_type
acl_list	sec_acl_list
acl_result	result
count_max	size_avail
count	size_used

This Specification	OSF DCE Reference Implementation
rdacl_get_access()	
access_rights	net_rights
rdacl_test_access()	
access_rights	desired_permset
rdacl_get_manager_types()	
num_manager_types	num_types
rdacl_get_printstring()	
manager_type_next	manager_type_chain
num_printstrings	total_num_printstrings
rdacl_get_referral()	
tower_set	towers
rdacl_get_mgr_types_semantics()	
num_manager_types	num_types
rsec_id_*()	
rpc_handle	h
domain	name_domain
cell_name	cell_namep
cell_uuid	cell_idp
pgo_name	princ_namep
pgo_uuid	princ_idp

Appendix B

Error Code Mapping List

Note: This appendix is informative, not normative. It imposes no restrictions on conforming implementations.

The list below is a ‘‘error code mapping list’’, describing the codes returned from the security API in the source code of the standard OSF reference implementation of DCE.

Name: sec_acl_bad_acl_syntax

Value: 0x17122026

Specified ACL is not valid at this ACL manager. This error can be returned if less than 1 ACL is specified on a replace operation, or if more than 1 is specified on a replace and the controlling ACL manager was only expecting 1.

Name: sec_acl_bad_key

Value: 0x17122021

Internal error.

Name: sec_acl_bad_parameter

Value: 0x17122032

Internal error, should never occur.

Name: sec_acl_bad_permset

Value: 0x17122037

One or more permissions not valid for this type of ACL.

Name: sec_acl_bind_error

Value: 0x1712202c

A binding error occurred during the requested ACL operation.

Name: sec_acl_cant_allocate_memory

Value: 0x17122017

Cannot allocate memory for requested operation.

Name: sec_acl_duplicate_entry

Value: 0x17122031

Duplicate ACL entries are not allowed.

Name: sec_acl_expected_group_obj

Value: 0x1712201e

Object has an owning group but no group_obj entry in its ACL.

Name: sec_acl_expected_user_obj

Value: 0x1712201d

Object has an owner but no user_obj entry contained in its ACL.

Name: sec_acl_invalid_acl_handle

Value: 0x1712202d

Internal error, should never occur.

Name: sec_acl_invalid_acl_type

Value: 0x17122020

Specified ACL type is out of the valid range for this type.

Name: sec_acl_invalid_dfs_acl

Value: 0x17122035

DFS ACL manager does not understand the specified ACL.

Name: sec_acl_invalid_entry_class

Value: 0x17122028

Obsolete.

Name: sec_acl_invalid_entry_name

Value: 0x1712201c

NULL or invalid entry name passed to sec_acl_bind() API.

Name: sec_acl_invalid_entry_type

Value: 0x1712201f

An ACL entry type was specified that the server does not understand. It is possible for this to occur if a client is attempting to pass new ACL entry types to an older server that cannot interpret them (eg: delegation ACL entry types).

Name: sec_acl_invalid_manager_type

Value: 0x17122022

Invalid ACL manager type specified.

Name: sec_acl_invalid_permission

Value: 0x17122025

One or more specified permissions not valid for this ACL.

Name: sec_acl_invalid_site_name

Value: 0x17122018

The ACL operation specified an invalid site name.

Name: sec_acl_mgr_file_open_error

Value: 0x1712202f

ACL manager unable to open database file on startup.

Name: sec_acl_mgr_no_space

Value: 0x17122036

ACL manager was not able to store the specified ACL.

Name: sec_acl_missing_required_entry

Value: 0x17122030

ACL is missing an entry required by this ACL manager.

Name: sec_acl_name_resolution_failed

Value: 0x1712202a

Name resolution failed on the requested ACL operation.

Name: sec_acl_no_acl_found

Value: 0x1712201b

Object found, but object has no ACL associated with it.

Name: sec_acl_no_owner
Value: 0x17122027
There must be at least one entry in the ACL that grants control over the ACL.

Name: sec_acl_not_authorized
Value: 0x17122033
Not authorized to perform the requested operation on this object.

Name: sec_acl_not_implemented
Value: 0x17122016
Requested operation is not implemented in this version of DCE.

Name: sec_acl_no_update_sites
Value: 0x1712202e
No update sites available for this ACL operation.

Name: sec_acl_object_not_found
Value: 0x1712201a
Specified ACL object was not found.

Name: sec_acl_rpc_error
Value: 0x1712202b
An RPC error was returned during the ACL operation.

Name: sec_acl_server_bad_state
Value: 0x17122034
Server is not in a state capable of performing the requested operation.

Name: sec_acl_site_read_only
Value: 0x17122024
Requested operation attempted at a read only site. This error should be trapped by the security ACL API and the operation should be resent to an update site automatically if one can be located.

Name: sec_acl_unable_to_authenticate
Value: 0x17122029
Attempt to authenticate to server controlling the object failed.

Name: sec_acl_unknown_manager_type
Value: 0x17122019
Programming error, should not happen.

Name: sec_attr_bad_acl_mgr_set
Value: 0x17122151
Application specified an invalid acl manager set for this operation.

Name: sec_attr_bad_acl_mgr_type
Value: 0x17122152
Application specified an invalid acl manager.

Name: sec_attr_bad_bind_authn_svc
Value: 0x17122161
Application specified an invalid authentication service in the binding auth info for this

operation.

Name: sec_attr_bad_bind_authz_svc
Value: 0x17122162
Application specified an invalid authz_svc parameter in a request to the privilege server.

Name: sec_attr_bad_bind_info
Value: 0x17122153
Application specified invalid binding information in an attribute update operation.

Name: sec_attr_bad_bind_prot_level
Value: 0x17122160
Application specified an invalid protection level in the binding auth info.

Name: sec_attr_bad_bind_svr_name
Value: 0x1712215f
Application specified an invalid server name in the binding auth info.

Name: sec_attr_bad_comment
Value: 0x1712215a
Attribute comment specified exceeds 1024 characters.

Name: sec_attr_bad_cursor
Value: 0x17122163
Application specified an invalid cursor.

Name: sec_attr_bad_encoding_type
Value: 0x17122158
Attribute encoding type specified is invalid.

Name: sec_attr_bad_intercell_action
Value: 0x1712215b
Intercell action specified for an attribute must be one of: accept, reject or evaluate.

Name: sec_attr_bad_name
Value: 0x17122157
Attribute name specified is NULL or exceeds 1024 characters.

Name: sec_attr_bad_object_type
Value: 0x17122166
Application specified an acl manager type for this object that is not contained in the schema entry for this attribute.

Name: sec_attr_bad_param
Value: 0x1712216d
Application specified a bad parameter for a schema or attribute operation.

Name: sec_attr_bad_permset
Value: 0x17122154
Application specified one or more invalid permissions for this type of ACL.

Name: sec_attr_bad_scope
Value: 0x17122159
Attribute scope specified exceeds 1024 characters.

Name: sec_attr_bad_trig_type

Value: 0x1712215c

Application specified a trigger type other than query in this operation. Only query triggers are supported in this release.

Name: sec_attr_bad_type

Value: 0x17122150

Application performing a lookup using the specified attribute type was unsuccessful.

Name: sec_attr_bad_uniq_query_accept

Value: 0x1712215e

If the unique flag is set to true, and a query trigger is used, the intercell action can not be set to accept.

Name: sec_attr_cant_get_attrinst

Value: 0x17122ebc

Application failed to get the next attribute instance in the attribute list.

Name: sec_attr_cant_get_attrlist

Value: 0x17122eb9

Application could not retrieve the attribute list.

Name: sec_attr_cant_get_instance

Value: 0x17122ebb

Application failed to get the last attribute instance in the attribute list.

Name: sec_attr_inst_not_found

Value: 0x17122144

Application specified an attribute instance that was not found.

Name: sec_attr_multi_inst_no_update

Value: 0x17122170

Schema multi instance flag cannot be unset.

Name: sec_attr_name_exists

Value: 0x1712214d

Application attempted to add an attribute with a duplicate name to the registry.

Name: sec_attr_no_memory

Value: 0x17122155

Unable to allocate memory.

Name: sec_attr_no_more_entries

Value: 0x1712216c

Application has exhausted the available attribute instance or schema entries.

Name: sec_attr_not_implemented

Value: 0x17122164

Application specified an operation that has not yet been implemented.

Name: sec_attr_not_multi_valued

Value: 0x17122167

Application specified more than one attribute instance for a type that is not multi valued.

Name: sec_attr_num_attr_ltzero
Value: 0x17122eba
Application attempted to decrement the number of attributes below zero.

Name: sec_attr_rgy_obj_not_found
Value: 0x17122147
Specified registry object was not found.

Name: sec_attr_schema_cant_lookup
Value: 0x17122edd
Unable to lookup the schema entry.

Name: sec_attr_schema_cant_reset
Value: 0x17122edc
Unable to reset the schema entry information.

Name: sec_attr_sch_entry_not_found
Value: 0x17122143
Application specified a schema entry that does not exist.

Name: sec_attr_sch_reserved
Value: 0x1712216e
Cannot delete schema entry with reserved flag set.

Name: sec_attr_trig_bind_info_missing
Value: 0x1712215d
Application specified a trigger without supplying binding info.

Name: sec_attr_trig_query_not_sup
Value: 0x17122165
Application specified a query trigger for an operation that does not support query triggers.

Name: sec_attr_trig_types_no_update
Value: 0x17122171
Schema trigger types cannot be updated.

Name: sec_attr_type_id_exists
Value: 0x1712214e
Application attempted to add an attribute with a duplicate id to the registry.

Name: sec_attr_unauthorized
Value: 0x17122149
The object's ACL denied the attempted operation.

Name: sec_attr_unique_no_update
Value: 0x1712216f
Schema unique flag cannot be unset.

Name: sec_attr_val_attr_set_bad
Value: 0x1712216b
Appliation specifed an improperly formatted attribute value set.

Name: sec_attr_val_bytes_bad
Value: 0x1712216a

Application specified improperly formatted bytes string.

Name: sec_attr_val_printstring_bad

Value: 0x17122168

Application specified a printstring value that exceeds 1024 characters.

Name: sec_attr_val_string_array_bad

Value: 0x17122169

Application either specified an improperly formatted attribute value string array or one or more of the attribute value strings in the array exceed 1024 characters.

Name: sec_authn_s_bad_seal

Value: 0x1712217d

Data corruption, or an attacker asserting credentials that don't match the credentials that were actually granted. If error is reproducible, then it's most likely a defect in DCE.

Name: sec_authn_s_missing_epac

Value: 0x1712217b

Improperly formed RPC authentication protocol message, probably due to a defect in DCE.

Name: sec_authn_s_no_seal

Value: 0x1712217c

Improperly formed RPC authentication protocol message, probably due to a defect in DCE.

Name: sec_buf_too_small

Value: 0x17122f60

The buffer size is smaller than the amount of data which needs to be copied into the buffer. This is an internal error.

Name: sec_crdb_at_char_in_cellname

Value: 0x17122dc8

A cell name cannot contain the **@** (at sign) character.

Name: sec_crdb_cant_add_replica

Value: 0x17122dd8

The application could not add a new replica to the master registry.

Name: sec_crdb_cant_bind_updsite

Value: 0x17122ddc

The application could not locate and bind to the master registry.

Name: sec_crdb_cant_com_master

Value: 0x17122ddd

The application could not communicate with the master registry.

Name: sec_crdb_cant_create_celluuid

Value: 0x17122dd3

Application could not create a cell UUID.

Name: sec_crdb_cant_get_hostname

Value: 0x17122dd2

Application was unable to retrieve host name.

Name: sec_crdb_cant_get_host_prname
Value: 0x17122dd5
Application was unable to retrieve the host principal name.

Name: sec_crdb_cant_register_ns
Value: 0x17122dd7
Application cannot register with the name service.

Name: sec_crdb_cant_setup_rgycreator
Value: 0x17122dd4
Problem setting up rgy_creator.

Name: sec_crdb_cant_upd_rgyst_file
Value: 0x17122dd9
Cannot update rgy_state file.

Name: sec_crdb_cl_alt_dir_no_arg
Value: 0x17122dd1
No argument was specified for alt_dir. Hence a default path was used.

Name: sec_crdb_cl_bad_name
Value: 0x17122dc9
The specified name is not a legal CDS name.

Name: sec_crdb_cl_dup_option
Value: 0x17122dca
Only one of the two options may be used.

Name: sec_crdb_cl_long_passwd
Value: 0x17122dcd
Password longer than the permitted maximum.

Name: sec_crdb_cl_long_rgyname
Value: 0x17122dcc
Registry name is longer than the permitted maximum.

Name: sec_crdb_cl_missing_arg
Value: 0x17122dcb
The specified option requires an argument which was not specified.

Name: sec_crdb_cl_null_myname
Value: 0x17122dcf
The specified name is either NULL or a null string.

Name: sec_crdb_cl_unknown_option
Value: 0x17122dce
The specified option is not valid. Consult the manual page for the correct set of options.

Name: sec_crdb_cl_usage
Value: 0x17122dd0
Bad syntax in sec_create_db command.

Name: sec_crdb_cr_db_succ
Value: 0x17122dc6

This is an informational message.

Name: sec_crdb_cr_master_db
Value: 0x17122dc4
This is an informational message.

Name: sec_crdb_cr_rep_db
Value: 0x17122dc5
This is an informational message.

Name: sec_crdb_db_exists
Value: 0x17122dc7
Registry database could not be created because one exists already.

Name: sec_crdb_db_not_created
Value: 0x17122dd6
The new database was not created.

Name: sec_crdb_inherit_hostident_err
Value: 0x17122ddb
Cannot inherit local host identity.

Name: sec_crdb_rep_not_registered
Value: 0x17122dda
The security replica has not successfully registered with the name service

Name: sec_crdb_site_file_create_fail
Value: 0x17122dc0
The pe_site file could not be created or updated because of an error. The error is logged prior to this message.

Name: sec_crdb_site_file_create_succ
Value: 0x17122dc1
The pe_site file has been successfully created. This is an informational message.

Name: sec_crdb_site_file_upd_fail
Value: 0x17122dc3
RPC bindings could not be appended to the pe_site file. The messages logged prior to this indicate why this might have happened.

Name: sec_crdb_site_file_upd_succ
Value: 0x17122dc2
RPC bindings have been successfully appended to the pe_site file. This is an informational message.

Name: sec_cred_s_authz_cannot_comply
Value: 0x17122132
Application programming error. The server is asking for information that the authorisation service used for the call cannot supply (eg, calling sec_cred_get_initiator() when the call had used authz_name).

Name: sec_cred_s_invalid_auth_handle
Value: 0x1712212f
Specified credential handle is invalid.

Name: sec_cred_s_invalid_cursor
Value: 0x17122131
Specified credential cursor is invalid.

Name: sec_cred_s_invalid_pa_handle
Value: 0x17122130
Specified privilege attribute handle is invalid.

Name: sec_cred_s_no_more_entries
Value: 0x1712212e
No more entries available (informational status code).

Name: sec_id_e_bad_cell_uuid
Value: 0x171220d5
Specified cell UUID does not match any known cell names.

Name: sec_id_e_name_too_long
Value: 0x171220d4
The specified name is too long for the current implementation.

Name: sec_key_mgmt_e_authn_invalid
Value: 0x17122044
The specified authentication service is invalid for this operation.

Name: sec_key_mgmt_e_auth_unavailable
Value: 0x17122045
Unable to contact the authentication service.

Name: sec_key_mgmt_e_keytab_not_found
Value: 0x1712204a
Unable to locate or open specified key table.

Name: sec_key_mgmt_e_key_unavailable
Value: 0x17122043
No key matching specified principal and key version found in keytable.

Name: sec_key_mgmt_e_key_unsupported
Value: 0x17122047
A key with a type unknown to this version of DCE was specified.

Name: sec_key_mgmt_e_key_version_ex
Value: 0x17122048
Specified key already exists in the specified key table.

Name: sec_key_mgmt_e_ktfile_err
Value: 0x1712204b
File found, but format does not conform to that of a key table.

Name: sec_key_mgmt_e_not_implemented
Value: 0x17122049
Specified operation not implemented in this version of DCE.

Name: sec_key_mgmt_e_unauthorized
Value: 0x17122046

The caller is unauthorized to perform the requested operation.

Name: sec_lksm_bad_input

Value: 0x17122cc4

The user supplied response to the question of whether a locksmith account should be created is not correct.

Name: sec_lksm_create_acct

Value: 0x17122cc5

This will be followed by the message in sec_lksm_def_yes_prompt.

Name: sec_lksm_def_no_prompt

Value: 0x17122cc3

If security server is started in the locksmith mode and no locksmith account exists, then the user is prompted asking whether the account should be created. This prompt is used when the default is to not to create the the locksmith account.

Name: sec_lksm_def_yes_prompt

Value: 0x17122cc2

If security server is started in the locksmith mode and no locksmith account exists, then the user is prompted asking whether the account should be created. This prompt is used when the desired default is to create the locksmith account.

Name: sec_lksm_passwd_prompt

Value: 0x17122ccf

The user is prompted for the password of the locksmith account.

Name: sec_lksm_passwd_verify

Value: 0x17122cd0

The user is prompted to reenter the password of the locksmith account.

Name: sec_lksm_set_acct_span

Value: 0x17122ccb

The policy account lifespan to set to the specified time period.

Name: sec_lksm_set_acct_valid

Value: 0x17122cc9

The specified account is designated as valid.

Name: sec_lksm_set_client_valid

Value: 0x17122cc7

The specified account is designated as a client.

Name: sec_lksm_set_date_now

Value: 0x17122cca

Setting %s account good_since_date to now\n

Name: sec_lksm_set_expire

Value: 0x17122ccc

The specified account is set to expire in the specified time period.

Name: sec_lksm_set_polpwd_expire

Value: 0x17122cce

The policy password lifetime is set to the specified period.

Name: sec_lksm_set_polpwd_expire_now

Value: 0x17122ccd

The policy password lifetime is set to the specified period. The policy password expiration time is to expire in the specified time period.

Name: sec_lksm_set_pwd_valid

Value: 0x17122cc6

Setting password valid flag for the specified account.

Name: sec_lksm_set_server_valid

Value: 0x17122cc8

The specified account is designated as a server.

Name: sec_logent_out_of_bounds

Value: 0x17122ea3

The log entry is out of bounds and so is skipped.

Name: sec_login_s_acct_invalid

Value: 0x171220f6

Attempted to login to an account that is currently disabled.

Name: sec_login_s_already_valid

Value: 0x171220ef

Attempted to validate an already valid login context.

Name: sec_login_s_auth_local

Value: 0x171220e9

Operation is not valid on the local context.

Name: sec_login_s_compound_delegate

Value: 0x17122100

Attempted to call become_initiator with a login context that already contained a delegation chain.

Name: sec_login_s_config

Value: 0x171220f3

Host security client information not available. Either unable to find file **DCELOCAL/var/security/sec_clientd.binding**
or unable to set authorisation and authentication information for a server. .

Name: sec_login_s_context_invalid

Value: 0x171220eb

Attempted to use a not-yet-validated login context for an operation that requires a validated context.

Name: sec_login_s_default_use

Value: 0x171220f0

The default security login handle was used illegally.

Name: sec_login_s_deleg_not_enabled

Value: 0x17122102

Delegation / impersonation attempted, but initiator did not enable it.

Name: sec_login_s_groupset_invalid
Value: 0x171220ed
Attempting to perform a task illegally on a default context handle.

Name: sec_login_s_handle_invalid
Value: 0x171220ea
Specified login handle does not correspond to a login context.

Name: sec_login_s_incomplete_ovrd_ent
Value: 0x171220fe
Override entry for this entry encountered, with password field specified, but other necessary field(s) missing.

Name: sec_login_s_info_not_avail
Value: 0x171220ee
The unix password information is not available.

Name: sec_login_s_internal_error
Value: 0x171220f4
Internal error, should not occur.

Name: sec_login_s_invalid_compat_mode
Value: 0x17122101
Specified compatibility mode is not supported.

Name: sec_login_s_invalid_deleg_type
Value: 0x171220ff
Specified delegation type is not supported.

Name: sec_login_s_invalid_password
Value: 0x171220fd
The specified password is invalid.

Name: sec_login_s_no_current_context
Value: 0x171220ec
Login context is no longer completely accessible.

Name: sec_login_s_no_memory
Value: 0x171220e8
Unable to allocate memory.

Name: sec_login_s_no_override_info
Value: 0x171220f5
No override information is currently available.

Name: sec_login_s_not_certified
Value: 0x171220f2
Warning only. Information was obtained from a login context that has been validated but not certified.

Name: sec_login_s_not_implemented
Value: 0x171220e7
Specified operation is not yet implemented in this version of DCE.

Name: sec_login_s_null_password
Value: 0x171220f7
Cannot log in with a zero length password.

Name: sec_login_s_override_failure
Value: 0x171220fb
Unable to determine if any override information exists, so operation must be denied.

Name: sec_login_s_ovrd_ent_not_found
Value: 0x171220fc
No matching override entry found (informational status code).

Name: sec_login_s_preauth_failed
Value: 0x17122103
The client is unable to compose the necessary preauthentication data for this principal.

Name: sec_login_s_privileged
Value: 0x171220f1
Privilege operation was attempted in an unprivileged (non-root) process.

Name: sec_login_s_refresh_ident_bad
Value: 0x171220fa
Attempted to refresh credentials for an account that is no longer valid.

Name: sec_login_s_unsupp_passwd_type
Value: 0x171220f8
Attempted to login using an unsupported password type.

Name: sec_lrgy_s_cannot_create
Value: 0x17122119
Unable to create local registry files.

Name: sec_lrgy_s_internal_error
Value: 0x1712211b
Internal error, should not occur.

Name: sec_lrgy_s_max_lt_num_entries
Value: 0x17122117
User specified a max entry value smaller than current number of entries.

Name: sec_lrgy_s_no_access
Value: 0x1712211a
Local registry exists, but cannot be accessed.

Name: sec_lrgy_s_not_found
Value: 0x17122118
No local registry files found.

Name: sec_ns_import_begin
Value: 0x17122f2c
Beginning import RPC bindings.

Name: sec_ns_import_done
Value: 0x17122f2e

Completed import of RPC bindings.

Name: sec_ns_import_next
Value: 0x17122f2d
Attempting to import next RPC binding.

Name: sec_priv_s_bad_compat_mode
Value: 0x17122063
An out of range compat mode parameter was passed to the privilege server.

Name: sec_priv_s_bad_deleg_type
Value: 0x17122064
Specified delegation type is not valid.

Name: sec_priv_s_cmode_not_enabled
Value: 0x1712206c
Delegate attempted to specify a compatibility mode not allowed by initiator.

Name: sec_priv_s_corrupt_deleg_chain
Value: 0x17122067
Internal error, should not occur.

Name: sec_priv_s_deleg_not_enabled
Value: 0x17122065
Delegation attempted, but not enabled by initiator of operation.

Name: sec_priv_s_deleg_token_exp
Value: 0x17122066
Delegation operation attempted, but delegation token has expired.

Name: sec_priv_s_intercell_deleg_req
Value: 0x17122069
Intercell delegation requests are not yet supported.

Name: sec_priv_s_invalid_authn_svc
Value: 0x1712205e
Invalid authn_svc parameter in request to privilege server.

Name: sec_priv_s_invalid_authz_svc
Value: 0x1712205f
Invalid authz_svc parameter in request to privilege server.

Name: sec_priv_s_invalid_dlg_token
Value: 0x17122068
Internal error, should not occur.

Name: sec_priv_s_invalid_principal
Value: 0x1712205b
The principal requesting privileges is not valid. Could be caused by a race condition where the principal was just deleted, or could be caused by a defect in DCE.

Name: sec_priv_s_invalid_protect_lvl
Value: 0x1712206b
Privilege client code passed in an invalid protection level.

Name: sec_priv_s_invalid_request

Value: 0x17122061

Invalid request probably caused by defect in DCE, or corrupted data passed in.

Name: sec_priv_s_invalid_server_name

Value: 0x1712206a

Privilege client code passed in an invalid server name.

Name: sec_priv_s_invalid_trust_path

Value: 0x17122060

The intercell authentication path traversed to authenticate to the DCE privilege server does not conform to the requirements for hierarchical trust in DCE.

Name: sec_priv_s_no_mem

Value: 0x1712205d

Unable to allocate memory for specified operation.

Name: sec_priv_s_not_member_any_group

Value: 0x1712205c

Principal isn't a member of any of the groups it requested for its groupset. Most likely caused by a principal's group membership being changed since they logged in.

Name: sec_priv_s_PAD00

Value: 0x17122062

Obsolete error code.

Name: sec_priv_s_server_unavailable

Value: 0x1712205a

Unable to locate an accessible privilege server.

Name: sec_prop_bad_type

Value: 0x17122e43

Internal error.

Name: sec_prop_chk_prop_slave_init

Value: 0x17122e47

Check how slave initialisation is going.

Name: sec_prop_fail

Value: 0x17122e45

The update did not propagate successfully.

Name: sec_prop_no_master_info

Value: 0x17122e42

Propagation thread tried but could not obtain information about the current master security server in the cell. This is an internal error.

Name: sec_prop_no_prop_thrs

Value: 0x17122e40

The security server creates several propagation threads to manage the propagation of updates between the master and slave security servers. This error indicates that one or more such threads have not been created.

Name: sec_prop_not_master
Value: 0x17122e41
Propagation threads can only be created in a master security server not in a slave security server. This is an internal error.

Name: sec_prop_send_delete_rep
Value: 0x17122e4a
Attempting to propogate deletion of replicas.

Name: sec_prop_send_init_slave
Value: 0x17122e46
Attempting to initialize the slave.

Name: sec_prop_slave_init_done
Value: 0x17122e48
The slave was successfully initialized.

Name: sec_prop_succ
Value: 0x17122e44
The propagation has completed successfully.

Name: sec_prop_updates_to_slaves
Value: 0x17122e49
Ppropagating updates to slaves.

Name: sec_pwd_mgmt_not_authorized
Value: 0x17122177
Caller is not authorized to communicate with the password management server.

Name: sec_pwd_mgmt_str_check_failed
Value: 0x17122176
Specified password failed password strength server checking policy.

Name: sec_pwd_mgmt_svr_error
Value: 0x17122178
The password management server has failed to complete the requested operation due to an error.

Name: sec_pwd_mgmt_svr_unavail
Value: 0x17122179
Unable to contact password management server.

Name: sec_rca_op_status
Value: 0x17122f2f
Registry operation failed.

Name: sec_rca_site_rebind
Value: 0x17122f30
Attempting to rebind to an alternate registry site and retrying operation.

Name: sec_rca_site_rebind_fail
Value: 0x17122f32
Failed to rebind to an alternate registry to retry operation.

Name: sec_rca_site_rebind_succ
Value: 0x17122f31
Successfully rebound to specified site to retry operation.

Name: sec_rep_cant_start_prop_tasks
Value: 0x17122e6e
Cannot start the propagation tasks.

Name: sec_rep_corrupt_auth_handle
Value: 0x17122e62
Corrupted replica authentication handle detected.

Name: sec_rep_dupe_cant_start
Value: 0x17122e6a
Replica is in duplicate master state and cannot be started.

Name: sec_rep_dupe_not_master
Value: 0x17122e69
Replica is in duplicate master state but is not the master.

Name: sec_rep_init_slave_fail
Value: 0x17122e74
Initialisation failed.

Name: sec_rep_init_slave_succ
Value: 0x17122e73
Initialisation completed successfully.

Name: sec_rep_invalid_auth_handle
Value: 0x17122e63
Invalid replica authentication handle.

Name: sec_rep_maint_not_master
Value: 0x17122e68
Only a master security server can be in the maintenance mode not a slave.

Name: sec_rep_mseq_not_dup_master
Value: 0x17122e67
master_seqno flag can only be applied to a duplicate master.

Name: sec_rep_msrepl_not_inited
Value: 0x17122e6d
Cannot initialize master replica list.

Name: sec_rep_nm_not_deleted
Value: 0x17122e6b
Unable to remove the specified server name from the name space.

Name: sec_rep_not_on_replist
Value: 0x17122e6c
Specified replica is not on the replica list.

Name: sec_rep_prop_in_progress
Value: 0x17122e6f

Propagation was in progress to a replica when an attempting to free the master's volatile copy of the replica list.

Name: sec_rep_prop_type_not_init
Value: 0x17122e70
Propagation type is not init or initialising.

Name: sec_rep_recv_become_master
Value: 0x17122e76
The slave received a "become master" message.

Name: sec_rep_recv_i_am_master
Value: 0x17122e75
The slave received an "I am master" message.

Name: sec_rep_recv_init_slave
Value: 0x17122e72
On receipt of this request, the slave will attempt to initialize or reinitialize (?) itself.

Name: sec_rep_recv_stop_sw_compat
Value: 0x17122e77
The slave received a "stop until software s compatible" request.

Name: sec_rep_rm_not_in_service
Value: 0x17122e66
restore_master flag can only be specified to an in service master.

Name: sec_res_acct_add_err
Value: 0x17122cd8
An error occurred while adding an account.

Name: sec_res_attr_sch_add_err
Value: 0x17122cd9
An error occurred while adding an entry to the attribute schema.

Name: sec_res_host_key_set_err
Value: 0x17122cd3
An error occurred while setting local host's key.

Name: sec_res_mem_add_err
Value: 0x17122cd7
An error occurred while adding a member.

Name: sec_res_pgo_add_err
Value: 0x17122cd6
An error occurred while adding a person, group, or orgnisation entry.

Name: sec_res_princ_cvt_err
Value: 0x17122cd4
An error occurred while converting a cell name to a local realm principal.

Name: sec_res_uuid_cvt_err
Value: 0x17122cd5
An error occurred while converting a cell UUID to a string.

Name: sec_rgy_acl_init
Value: 0x17122d13
Initializing for sec_acl wire interface failed.

Name: sec_rgy_alias_not_allowed
Value: 0x17122096
Attempted to add an alias to a principal which prohibits that operation.

Name: sec_rgy_aud_open
Value: 0x17122d0d
Fail to open dce audit file

Name: sec_rgy_auth_init
Value: 0x17122d17
Cannot register server's authentication information with RPC runtime.

Name: sec_rgy_bad_chksum_type
Value: 0x17122097
Internal error, should not occur.

Name: sec_rgy_bad_data
Value: 0x17122084
Invalid data encountered during specified registry operation.

Name: sec_rgy_bad_domain
Value: 0x17122074
Attempted an operation that is not supported by the specified domain.

Name: sec_rgy_bad_handle
Value: 0x1712209d
Internal error, should not occur.

Name: sec_rgy_bad_integrity
Value: 0x17122098
Data integrity error. Could be caused by specifying invalid password.

Name: sec_rgy_bad_name
Value: 0x17122088
Illegal name (possibly illegal character(s)) passed to the sec_rgy API.

Name: sec_rgy_bad_name service_name
Value: 0x171220a3
Internal error.

Name: sec_rgy_bad_rgy_db
Value: 0x17122eab
A bad registry database state was encountered.

Name: sec_rgy_bad_scope
Value: 0x17122093
Attempted to set scope to a name that does not exist in the registry.

Name: sec_rgy_cant_allocate_memory
Value: 0x17122085

Unable to allocate memory for the specified operation.

Name: sec_rgy_cant_authenticate
Value: 0x17122095
Can’t establish authentication to security server.

Name: sec_rgy_checkpoint
Value: 0x17122eaf
Attempting to checkpoint the registry database.

Name: sec_rgy_checkpoint_succ
Value: 0x17122eb0
Successfully checkpointed the registry database.

Name: sec_rgy_checkpt_log_file
Value: 0x17122eac
Cannot perform checkpoint on the specified log file.

Name: sec_rgy_checkpt_rename_files
Value: 0x17122ead
Cannot rename files during checkpoint.

Name: sec_rgy_checkpt_save_rep_state
Value: 0x17122eae
Cannot save replica state during checkpoint.

Name: sec_rgy_checkpt_start_task
Value: 0x17122d1b
An error occurred while when trying to start a thread to do checkpoint task.

Name: sec_rgy_chkpt_save_file
Value: 0x17122ee6
Saving the specified file.

Name: sec_rgy_chkpt_save_relation
Value: 0x17122ee7
Saving the specified relation.

Name: sec_rgy_compat_log_replay
Value: 0x17122ee8
Compatibility log replay was entered.

Name: sec_rgy_db_create
Value: 0x17122f66
Failed to create the database.

Name: sec_rgy_db_init
Value: 0x17122d01
The security server is going to attempt to read the registry database into its virtual address space.

Name: sec_rgy_db_init_err
Value: 0x17122d11
Loading or initilaizing rgy database has error.

Name: sec_rgy_dce_rgy_identity
Value: 0x17122d19
Cannot set process identity (dce-rgy) and context.

Name: sec_rgy_dir_could_not_create
Value: 0x17122089
Unable to create a directory necessary for the specified operation.

Name: sec_rgy_dir_move_illegal
Value: 0x1712208a
Attempted to make a parent directory the child of one of its descendents.

Name: sec_rgy_era_pwd_mgmt_auth_type
Value: 0x171220a5
Principal's pwd_mgmt_binding ERA authentication cannot be

Name: sec_rgy_foreign_quota_exhausted
Value: 0x1712208c
Attempt by a foreign principal to add a registry object, but quota is exhausted.

Name: sec_rgy_get_cellname
Value: 0x17122f65
Cannot retrieve the requested cell name.

Name: sec_rgy_get_local_host_princ
Value: 0x17122d26
Cannot retrieve the requested local host principal name.

Name: sec_rgy_host_identity
Value: 0x17122d16
Cannot inherit host machine context and identity.

Name: sec_rgy_incomplete_login_name
Value: 0x1712207f
Specified login name structure was not completely specified.

Name: sec_rgy_init_rpc_bind
Value: 0x17122d0f
Trying to initialize rpc binding failed.

Name: sec_rgy_key_bad_size
Value: 0x17122099
Internal error, should not occur.

Name: sec_rgy_key_bad_type
Value: 0x17122091
The key type specified was not a valid for the specified operation.

Name: sec_rgy_locksmith_init
Value: 0x17122d14
Cannot set up the requestedlocksmith account.

Name: sec_rgy_log_entry_out_of_range
Value: 0x171220a4

Internal error.

Name: sec_rgy_log_init_mgr
Value: 0x17122d10
Cannot initialize the server log managers.

Name: sec_rgy_mkey_bad
Value: 0x1712209c
Registry master key retrieved from .mkey file doesn't match master key stored in the database.

Name: sec_rgy_mkey_bad_stored
Value: 0x1712209b
Registry master key stored in .mkey file has been corrupted.

Name: sec_rgy_mkey_file_io_failed
Value: 0x171220a0
Master key file operation (create/read/write) failed.

Name: sec_rgy_mky_bad_cellname
Value: 0x17122e20
The cell name must begin with /.../ but it does not.

Name: sec_rgy_mky_gen_random
Value: 0x17122e29
Cannot create the master key because an error occurred while generating a random master key.

Name: sec_rgy_mky_get_realm_name
Value: 0x17122e24
The cell name to be converted is not a legal cell name.

Name: sec_rgy_mky_init_keyseed
Value: 0x17122e26
Problem generating a DES key from user-entered keyseed and timeofday.

Name: sec_rgy_mky_init_random
Value: 0x17122e28
Cannot create the master keybecause an error occurred while initialising the random key generator.

Name: sec_rgy_mky_not_match
Value: 0x17122e2d
The master key in memory doesn't match the master key stored in the database.

Name: sec_rgy_mky_process_keyseed
Value: 0x17122e27
Cannot create the master key because an error occured while processing the keyseed.

Name: sec_rgy_mky_process_master_key
Value: 0x17122e2a
Cannot create the master key bacause an error occured while processing the master key.

Name: sec_rgy_mky_setup_mkey_name
Value: 0x17122e25
Possibly caused by not enough memory to be allocated.

Name: sec_rgy_mky_store_db
Value: 0x17122e2c
An error occurred while storing the master key in the database.

Name: sec_rgy_mky_store_disk
Value: 0x17122e2b
Cannot create the master key because an error occurred while storing the master key on the disk.

Name: sec_rgy_name_exists
Value: 0x17122076
Attempted to add a registry object that already exists.

Name: sec_rgy_no_more_entries
Value: 0x17122079
End of list encountered while performing registry lookup.

Name: sec_rgy_no_more_unix_ids
Value: 0x1712208d
No more available Unix IDs within allowable range.

Name: sec_rgy_not_authorized
Value: 0x17122081
The object's ACL denied the attempted operation.

Name: sec_rgy_not_implemented
Value: 0x17122073
Operation is not implemented in this version of DCE.

Name: sec_rgy_not_member_group
Value: 0x1712207c
The principal specified in the account operation is not a member of the specified primary group.

Name: sec_rgy_not_member_group_org
Value: 0x1712207e
The principal specified in the account operation is not a member of the specified group or organisation.

Name: sec_rgy_not_member_org
Value: 0x1712207d
The principal specified in the account operation is not a member of the specified organisation.

Name: sec_rgy_not_root
Value: 0x17122d00
The attempted operation requires root privileges.

Name: sec_rgy_ns_register
Value: 0x17122d1a
Cannot start the name service registration task.

Name: sec_rgy_ns_svr_get_binding
Value: 0x17122d53
Cannot get server's rpc binding from its repository.

Name: sec_rgy_object_exists
Value: 0x17122075
Attempted to add a registry object that already exists.

Name: sec_rgy_object_not_found
Value: 0x1712207a
Specified registry object was not found.

Name: sec_rgy_object_not_in_scope
Value: 0x17122094
Attempted to lookup object that doesn't exist within the current scope.

Name: sec_rgy_passwd_invalid
Value: 0x17122080
Specified password is invalid.

Name: sec_rgy_passwd_non_alpha
Value: 0x171220a7
Specified password contains all alphanumberic characters, which is not allowed by current policy.

Name: sec_rgy_passwd_spaces
Value: 0x171220a8
Specified password contains no non-blank character, which is not allowed by current policy.

Name: sec_rgy_passwd_too_short
Value: 0x171220a6
Specified password is shorter than the current minimum limit.

Name: sec_rgy_quota_exhausted
Value: 0x1712208b
Principal's registry quota is exhausted and an update operation was attempted.

Name: sec_rgy_read_only
Value: 0x17122082
Registry is in a read only state and an update was attempted.

Name: sec_rgy_rep_add_master_replica
Value: 0x17122e80
When a slave database is re-initializing, in-memory data is cleared and re-created. Problem occurred when trying to add master replica to its database.

Name: sec_rgy_rep_add_my_replica
Value: 0x17122e7f
When a slave database is re-initializing, in-memory data is cleared and re-created. Problem occurred when trying to add this slave replica to its database.

Name: sec_rgy_rep_already_inited
Value: 0x171220c9
Attempt to initialize a replica that has already been initialized.

Name: sec_rgy_rep_bad_arg
Value: 0x171220c2

Invalid operation.

Name: sec_rgy_rep_bad_binding
Value: 0x171220b6
Bad binding encountered by the registry server.

Name: sec_rgy_rep_bad_db_version
Value: 0x171220aa
Version stored with registry database is not that expected by the registry software executed.

Name: sec_rgy_rep_bad_init_id
Value: 0x171220c8
Internal error, should not occur.

Name: sec_rgy_rep_bad_master_seqno
Value: 0x171220cf
Internal error, should not occur.

Name: sec_rgy_rep_bad_prop_type
Value: 0x171220ca
Internal error, should not occur

Name: sec_rgy_rep_bad_state
Value: 0x171220b4
Operation attempted while registry was in a state unable to perform that type of operation.

Name: sec_rgy_rep_bad_sw_vers
Value: 0x171220c6
Attempted to start registry server with software that is at a version incompatible with that which is stored in the registry database.

Name: sec_rgy_rep_cannot_create_db
Value: 0x171220ab
Unable to create database.

Name: sec_rgy_rep_cannot_open_db
Value: 0x171220ac
Unable to open registry database file that should already exist.

Name: sec_rgy_rep_cannot_read_db
Value: 0x171220ad
Registry server is unable to read the registry database.

Name: sec_rgy_rep_cannot_rename_db
Value: 0x171220af
The registry server was unable to rename the database files during conversion to the current database format.

Name: sec_rgy_rep_cannot_save_db
Value: 0x171220ae
Unable to save registry database to disk.

Name: sec_rgy_rep_clock_skew
Value: 0x171220b9

Clock value between registry server machines is out of tolerance.

Name: sec_rgy_rep_db_locked
Value: 0x171220b8
Database is already locked by another process.

Name: sec_rgy_rep_doppelganger
Value: 0x171220bc
Another replica with the same name or id exists.

Name: sec_rgy_rep_entry_not_found
Value: 0x17122e71
Cannot find the in-memory replica list entry in the stable replica list.

Name: sec_rgy_rep_host_identity
Value: 0x17122d18
Cannot get local host principal's context and identity.

Name: sec_rgy_rep_init_ekey_invalid
Value: 0x171220c5
Initialisation encryption key is not valid.

Name: sec_rgy_rep_init_replica
Value: 0x17122d12
Cannot initialize the server replica.

Name: sec_rgy_rep_invalid_entry
Value: 0x171220c4
Invalid replica entry encountered.

Name: sec_rgy_rep_marked_for_init
Value: 0x171220c7
Attempted to mark a replica for initialisation, that has already been marked for initialisation.

Name: sec_rgy_rep_master
Value: 0x171220b1
Specified operation may only be performed at a non-master registry replica site.

Name: sec_rgy_rep_master_bad_sw_vers
Value: 0x171220cb
Master registry is running a version of software that is not compatible with the replica registry servers.

Name: sec_rgy_rep_master_dup
Value: 0x171220cd
Duplicate master registry servers found.

Name: sec_rgy_rep_master_not_found
Value: 0x171220b0
A registry replica was unable to locate the master registry.

Name: sec_rgy_rep_master_obsolete
Value: 0x17122e4b
A slave has a higher update sequence number than this master, which implies this master has an

obsolete database; so exit itself.

Name: sec_rgy_rep_mst_restart_prop

Value: 0x17122e79

After change_master operation failed, the old master try to resume its master role but fail to restart its propagation task threads.

Name: sec_rgy_rep_not_from_master

Value: 0x171220b3

Internal error, should not happen.

Name: sec_rgy_rep_not_master

Value: 0x171220b2

Specified operation may only be performed by the master registry server.

Name: sec_rgy_rep_pack_entry

Value: 0x17122e09

When trying to log replication for add or replace, error occurrs.

Name: sec_rgy_rep_pgmerr

Value: 0x171220a9

Internal error, should not occur.

Name: sec_rgy_rep_recover_db

Value: 0x17122e7e

After an attemp to initialize a replica failed, this operation tried to clear data in memory and reload pre-initializetion database from disk, also failed.

Name: sec_rgy_rep_set_init_id

Value: 0x17122e7c

This code won't be executed; we may as well take it out :-) or replaced with same fatal message

Name: sec_rgy_rep_set_state

Value: 0x17122e7b

This code won't be executed; we may as well take it out :-) or replaced with same fatal message

Name: sec_rgy_rep_set_volatile_state

Value: 0x17122e7d

This code won't be executed; we may as well take it out :-) or replaced with same fatal message

Name: sec_rgy_rep_slave_bad_sw_vers

Value: 0x171220cc

Replica registry server is running a version of software that is not compatible with the master registry server.

Name: sec_rgy_rep_slv_restart_prop

Value: 0x17122e7a

After become_slave operation failed, the old master try to resume its master role but fail to restart its propagation task threads.

Name: sec_rgy_rep_update_seqno_high

Value: 0x171220ba

Slave must have missed / lost some update from the master.

Name: sec_rgy_rep_update_seqno_low

Value: 0x171220bb

Registry replica received an update from a master registry that is older than updates already received by the replica. Either the replica is accepting the new master and should automatically reinitialize itself from this master, or it believes that the master has an obsolete database and it will shut down.

Name: sec_rgy_rsdb_attr_delete

Value: 0x17122ebf

Cannot delete the attribute instance.

Name: sec_rgy_rsdb_attr_export

Value: 0x17122ec0

When exporting attribute values from database to sec_attr, error occurrs.

Name: sec_rgy_rsdb_attr_import

Value: 0x17122ebe

When importing attribute values in sec_attr to internal buffer area, error occurrs.

Name: sec_rgy_rsdb_attr_set_id

Value: 0x17122ebd

Cannot set the object's attribute list ID.

Name: sec_rgy_rsdb_checkpt

Value: 0x17122e78

Cannot checkpoint the database.

Name: sec_rgy_rsdb_checkpt_uninit

Value: 0x17122e81

When a slave database is re-initializing, in-memory data is cleared and re-created. Problem occurred when trying to do checkpoint on this database.

Name: sec_rgy_server_unavailable

Value: 0x1712207b

Unable to contact a registry server.

Name: sec_rgy_set_stack_size

Value: 0x17122d0e

Cannot set rpc listener thread stack size to be 64*1024.

Name: sec_rgy_shutdown_done

Value: 0x17122d06

Security server shutdown has been completed.

Name: sec_rgy_site_not_absolute

Value: 0x171220a2

A non-absolute name specified as the registry site.

Name: sec_rgy_s_pgo_is_required

Value: 0x1712209e

Attempted to delete a required PGO or account.

Name: sec_rgy_startup_done

Value: 0x17122d05
Security server initialisation has been completed.

Name: sec_rgy_svr_register
Value: 0x17122d15
Failed when registering with the rpc runtime and with the endpoint mapper.

Name: sec_rgy_svr_register_ns
Value: 0x17122d54
The server cannot register with the name service.

Name: sec_rgy_thr_exit_alert
Value: 0x17122d03
The thread is exiting with an alert exception.

Name: sec_rgy_thr_exit_exc
Value: 0x17122d04
The thread is exiting with an exception.

Name: sec_rgy_thr_join
Value: 0x17122d02
Cannot join pthread tasks.

Name: sec_rgy_thr_set_pool
Value: 0x17122d1c
Prior to setting maximum number of rpc listener threads, a call to set rpc listener threads pool queue length failed,.

Name: sec_rgy_unix_id_changed
Value: 0x17122077
The specified unix id doesn't match unix id extracted from the specified UUID.

Name: sec_rgy_uuid_bad_version
Value: 0x1712208e
Version of UUID does not match that expected for this operation. Could occur if the registry server expected a UUID containing an embedded Unix ID, but was passed a generic UUID.

Name: sec_rsdb_acct_add_curkey
Value: 0x17122eb7
An attempt to add to the current key version was detected.

Name: sec_rsdb_acct_cant_getid
Value: 0x17122eb5
The application was unable to get the account by the specified ID.

Name: sec_rsdb_acct_end_list
Value: 0x17122eb8
The end of the member list was encountered unexpectedly.

Name: sec_rsdb_acct_noaliases
Value: 0x17122eb6
There are no remaining aliases.

Name: sec_rsdb_acct_reset
Value: 0x17122eb4
Unable to reset previous account information.

Name: sec_rsdb_bad_policy_data
Value: 0x17122ecb
The policy data is not of a valid size.

Name: sec_rsdb_bad_policy_key
Value: 0x17122ecc
The key for the policy data is illegal.

Name: sec_rsdb_cant_cntr_item_name
Value: 0x17122ecd
Unable to construct the specified item name.

Name: sec_rsdb_cant_get_group_creds
Value: 0x17122ed4
Unable to obtain credentials for the specified group.

Name: sec_rsdb_cant_get_item
Value: 0x17122ec9
Unable to look up the specified item.

Name: sec_rsdb_cant_get_item_seqid
Value: 0x17122ed2
Unable to get the record of the specified item for sequential ID.

Name: sec_rsdb_cant_get_key
Value: 0x17122ed7
Unable to get the key for sequential ID.

Name: sec_rsdb_cant_get_member_data
Value: 0x17122ed6
Unable to get membership data.

Name: sec_rsdb_cant_get_mgr_typuuid
Value: 0x17122edb
Could not get a manager type UUID.

Name: sec_rsdb_cant_get_org_creds
Value: 0x17122ed5
Unable to obtain credentials for the specified organization.

Name: sec_rsdb_cant_get_person_creds
Value: 0x17122ed3
Unable to obtain credentials for the specified person.

Name: sec_rsdb_cant_get_pgo_creds
Value: 0x17122ec8
Unable to obtain credentials for the specified pgo.

Name: sec_rsdb_cant_init_acl
Value: 0x17122ed9

Could not initialize the ACL list.

Name: sec_rsdb_cant_set_auth_policy
Value: 0x17122ec6
Unable to set the authorisation policy.

Name: sec_rsdb_cant_set_policy
Value: 0x17122ec5
Unable to set the policy.

Name: sec_rsdb_cant_set_properties
Value: 0x17122ec3
Unable to set the properties.

Name: sec_rsdb_cant_set_realm
Value: 0x17122ec4
Unable to set the realm.

Name: sec_rsdb_cant_store_new_item
Value: 0x17122ed1
Could not store the new item.

Name: sec_rsdb_cant_walk_alias_chain
Value: 0x17122eca
Unable to walk the alias chain.

Name: sec_rsdb_corrupt_alias_chain
Value: 0x17122ece
The database alias chain is corrupted.

Name: sec_rsdb_db_chkpt_err
Value: 0x17122ea5
Cannot checkpoint the database.

Name: sec_rsdb_db_inconsistent
Value: 0x17122ed0
The database is inconsistent.

Name: sec_rsdb_dbstore_fail
Value: 0x17122ea9
Storage in the database failed.

Name: sec_rsdb_db_unrecog_state
Value: 0x17122ea1
The database is in an unrecognized state.

Name: sec_rsdb_db_write_fail
Value: 0x17122eb3
Write to the database failed.

Name: sec_rsdb_end_memb_list
Value: 0x17122ecf
End of membership list was reached.

Name: sec_rsdb_ent_not_xlated
Value: 0x17122ea2
The log entry could not be translated and so was skipped.

Name: sec_rsdb_fetch_error
Value: 0x17122ed8
An error occurred while fetching data.

Name: sec_rsdb_file_rename_err
Value: 0x17122ea7
Cannot rename files during checkpoint.

Name: sec_rsdb_file_stat_fail
Value: 0x17122eb1
Unable to stat the file with the specified descriptor.

Name: sec_rsdb_inconsistent_creds
Value: 0x17122ec7
The database inconsistent; the credentials item length is not valid.

Name: sec_rsdb_list_not_terminated
Value: 0x17122ec1
The ist was not properly terminated.

Name: sec_rsdb_log_chkpt_err
Value: 0x17122ea6
Cannot checkpoint log file.

Name: sec_rsdb_logent_replay_err
Value: 0x17122ea4
An error occurred while replaying the log entry and it was skipped.

Name: sec_rsdb_log_file_open
Value: 0x17122ea0
The log file is already open.

Name: sec_rsdb_no_open_slot
Value: 0x17122ec2
There is no open slot in the list.

Name: sec_rsdb_readver_fail
Value: 0x17122eb2
Unable to read version the specified version file.

Name: sec_rsdb_repl_fail
Value: 0x17122eaa
The database replace operation failed.

Name: sec_rsdb_rep_state_not_saved
Value: 0x17122ea8
Cannot save the replica state.

Name: sec_rsdb_unknown_aclmgr_type
Value: 0x17122eda

Unknown ACL manager type.

Name: sec_rs_global_lock_fatal_exc

Value: 0x17122edf

An exception occurred while a global lock was held.

Name: sec_rs_lock_fatal_exc

Value: 0x17122ede

An exception occurred while a lock was held. The first %s is the mode string which can be read, write or read-intend-to-write . The **Name**: second parameter is the type of lock and indicates on what the lock was held - database, replica list, log etc.

Name: sec_rs_log_bad_version

Value: 0x17122e00

The version of the log file is bad.

Name: sec_rs_log_base_prop_seq

Value: 0x17122e06

Logged during replay.

Name: sec_rs_log_file_closed

Value: 0x17122e02

The log file was not open.

Name: sec_rs_login_bad_name

Value: 0x17122d27

The login name is invalid.

Name: sec_rs_login_cant_refresh

Value: 0x17122d25

Unable to refresh the specified identity; will idle and retry.

Name: sec_rs_login_null_handle

Value: 0x17122d21

The registry login handle is null.

Name: sec_rs_login_null_name

Value: 0x17122d22

The login name is either a null pointer or a null string.

Name: sec_rs_login_refresh

Value: 0x17122d24

The thread will attempt to refresh to the login context using the call sec_login_refresh_identity()

Name: sec_rs_login_refresh_wait

Value: 0x17122d23

The thread will wait for the specified number of seconds before attempting to refresh the login context.

Name: sec_rs_login_wrong_call

Value: 0x17122d20

Internal error.

Name: sec_rs_log_open_fail
Value: 0x17122e01
Failed to open the log file.

Name: sec_rs_log_propq_add_fail
Value: 0x17122e03
An attempt to add information to the propagation queue failed. .

Name: sec_rs_log_replay
Value: 0x17122e04
Replay the log file.

Name: sec_rs_log_replay_entry
Value: 0x17122e07
Logged during replay.

Name: sec_rs_log_replay_err
Value: 0x17122e08
An error occurred while replaying the log.

Name: sec_rs_log_replay_succ
Value: 0x17122e05
Successfully replayed the log file.

Name: sec_rs_mkey_actver_mismatch
Value: 0x17122e21
The account’s master key version doesn’t match the old or the new.

Name: sec_rs_mkey_long_keyseed
Value: 0x17122e23
The keyseed is too long.

Name: sec_rs_mkey_unknown
Value: 0x17122e22
The master key version decrypting the account key is unrecognized.

Name: sec_rs_ns_bind_export
Value: 0x17122d4c
The security server is attempting to export the interfaces to the name space.

Name: sec_rs_ns_bind_export_succ
Value: 0x17122d4d
The bindings have been exported to name space.

Name: sec_rs_ns_bind_remove_succ
Value: 0x17122d4e
The bindings have been exported to name space.

Name: sec_rs_ns_cant_rm_member
Value: 0x17122d52
Unable to remove the old name from the group.

Name: sec_rs_ns_cant_rm_name
Value: 0x17122d51

The old name was not removed from the name service.

Name: sec_rs_ns_grp_ent_create_fail
Value: 0x17122d42
Informational event. .

Name: sec_rs_ns_grp_ent_create_succ
Value: 0x17122d43
Informational event.

Name: sec_rs_ns_grp_mbr_add_fail
Value: 0x17122d44
Informational event. .

Name: sec_rs_ns_grp_mbr_add_succ
Value: 0x17122d45
The member was added to the group.

Name: sec_rs_ns_name_del_succ
Value: 0x17122d4f
The name entry has been deleted successfully from the name space. The name entry is also no longer a member of the group entry.

Name: sec_rs_ns_name_not_removed
Value: 0x17122d50
The old name not removed from name service; it may not belong to this server.

Name: sec_rs_ns_null_profile
Value: 0x17122d40
This is an internal error.

Name: sec_rs_ns_null_v1_group
Value: 0x17122d41
This is an internal error.

Name: sec_rs_ns_prof_elt_add_fail
Value: 0x17122d46
The profile element was not added to cell-profile.

Name: sec_rs_ns_prof_elt_add_succ
Value: 0x17122d47
The profile element was added to cell-profile.

Name: sec_rs_ns_prof_elt_inq_fail
Value: 0x17122d4a
The profile element inquiry failed.

Name: sec_rs_ns_prof_elt_inq_succ
Value: 0x17122d4b
Read the catalog point from the profile.

Name: sec_rs_ns_prof_elt_rm_fail
Value: 0x17122d48
The secidmap to sec mapping could not be removed from cell-profile.

Name: sec_rs_ns_prof_elt_rm_succ
Value: 0x17122d49
The secidmap to sec mapping was removed from cell-profile.

Name: sec_rs_pipe_not_created
Value: 0x17122cd2
Unable to establish a parent-child pipe.

Name: sec_rs_pwd_bogus_pickle
Value: 0x17122cc1
Internal password representation incorrect.

Name: sec_rs_rep_incompat_version
Value: 0x17122e65
The software version is incompatible with the master's version. The server will exit.

Name: sec_rs_rep_not_master
Value: 0x17122e64
The replica is no longer the master.

Name: sec_rs_rpc_if_reg_succ
Value: 0x17122d80
Security server has successfully registered the server interfaces with the RPC runtime and the end point mapper

Name: sec_rs_rpc_if_unreg_succ
Value: 0x17122d83
Security server has unregistered the server interfaces from the RPC runtime and the end point mapper

Name: sec_rs_rpc_inq_bind_err
Value: 0x17122d85
Unable to establish the requested server bindings.

Name: sec_rs_rpc_propif_reg_succ
Value: 0x17122d81
Security server has successfully registered the interfaces required for **Name**: security replication with the RPC runtime and the end point mapper.

Name: sec_rs_rpc_propif_unreg_succ
Value: 0x17122d82
Security server has unregistered the interfaces required for **Name**: security replication from the RPC runtime and the end point mapper.

Name: sec_rs_rpc_prot_twr_err
Value: 0x17122d87
Unable to get the server's protocol towers.

Name: sec_rs_rpc_save_bind_err
Value: 0x17122d86
Unable to save the server's bindings.

Name: sec_rs_rpc_use_protseq_err

Value: 0x17122d84

Unable to listen on any protocol sequence.

Name: sec_rs_thr_create_fail

Value: 0x17122da1

The specified thread could not be created . The actual cause of failure is logged prior to this.

Name: sec_rs_thr_exit_creat_fail

Value: 0x17122da0

The security server exited because thread creation failed. To name of the thread which could not be created and the reason why it could not be created is logged by the status sec_rs_thr_create_fail.

Name: sec_rs_thr_exiting

Value: 0x17122da3

The thread is about to exit.

Name: sec_rs_thr_started

Value: 0x17122da2

The specified thread has been started. This message is logged by a thread as soon as it is created and starts running.

Name: sec_rs_vmcc_cant_register

Value: 0x17122e60

Unable to register virtual memory kerberos credential cache type.

Name: sec_rs_vmcc_cant_remove

Value: 0x17122e61

Cannot remove individual credentials from the VM cache.

Name: sec_s_authz_unsupp

Value: 0x17122001

The requested authorisation protocol is not supported by the authentication protocol requested.

Name: sec_s_bad_key_parity

Value: 0x1712200d

Specified DES key did not pass a parity check.

Name: sec_s_bad_nonce

Value: 0x17122003

Client failed challenge issued by server in RPC DG callback. Could be caused by a bug in DCE, trouble with the network, or possibly a failed security attack.

Name: sec_secd_cl_bad_arg

Value: 0x17122d09

The argument for the specified option is not correct.

Name: sec_secd_cl_bad_chkpt_interval

Value: 0x17122d0c

Checkpoint interval specified on the command line is not a positive number.

Name: sec_secd_cl_locksmith_opt

Value: 0x17122d0b
The specified option is valid and can be used only when -locksmith is also used.

Name: sec_secd_cl_missing_arg
Value: 0x17122d08
The specified option requires an argument which was not specified.

Name: sec_secd_cl_unknown_opt
Value: 0x17122d0a
The specified option is either invalid or unknown.

Name: sec_secd_cl_usage
Value: 0x17122d07
Name: secd started with incorrect arguments.

Name: sec_s_invalid_name service_entry
Value: 0x1712200b
Registry server encountered an error while processing its name service entry. May be caused by an incomplete or incorrect configuration, or by duplicate secd replicas running simultaneously.

Name: sec_site_bind_default
Value: 0x17122f24
Attempting to bind to a registry site using the specified file.

Name: sec_site_bind_fail
Value: 0x17122f22
Failed to bind to the specified registry site.

Name: sec_site_bind_start
Value: 0x17122f20
Attempting to bind to the specified registry site.

Name: sec_site_bind_succ
Value: 0x17122f21
Successfully bound to the registry site.

Name: sec_site_cell_bind_start
Value: 0x17122f23
Attempting to bind to an arbitrary registry site in the specified cell.

Name: sec_site_lookup_file
Value: 0x17122f28
Retrieving RPC string binding handles for the specified server from the specified file.

Name: sec_site_profile_search_fail
Value: 0x17122f2b
The search for a security server using the specified profile s failed.

Name: sec_site_profile_search_start
Value: 0x17122f29
Beginning a search for the security server using the specified profile.

Name: sec_site_profile_search_succ
Value: 0x17122f2a

Successfully located security the specified server using the specified profile.

Name: sec_site_rebind_fail
Value: 0x17122f27
Failed to rebind to an alternate registry site.

Name: sec_site_rebind_start
Value: 0x17122f25
Attempting to rebind to an alternate registry site.

Name: sec_site_rebind_succ
Value: 0x17122f26
Successfully rebound to the registry site.

Name: sec_s_keytype_unsupp
Value: 0x17122002
Most likely an internal error, caused by a defect in DCE.

Name: sec_s_no_key_seed
Value: 0x17122009
Security service has not yet been initialized, so there is no random key seed available.

Name: sec_s_no_memory
Value: 0x17122007
Unable to allocate memory for the requested operation.

Name: sec_s_none_registered
Value: 0x17122004
Application programming error. The server has not yet registered its identity with the security runtime.

Name: sec_s_no_pac
Value: 0x17122005
Improperly formed RPC authentication protocol message. Most likely caused by a defect in DCE.

Name: sec_s_not_implemented
Value: 0x17122006
Requested operation is not implemented by this verion of DCE.

Name: sec_s_not_trustworthy
Value: 0x17122008
Client field of an incoming ticket was not the known **Name**: security/privilege server.

Name: sec_s_pgmerr
Value: 0x1712200c
Internal security server error.

Name: sec_s_v1_1_no_support
Value: 0x1712200f
Client attempted to use a DCE1.1 security feature that the server doesn't support.

Name: sec_svc_cant_get_msg
Value: 0x17122cd1

Serviceability component returns a error.

Name: sec_svc_not_authorized
Value: 0x1712217e
Caller is not authorized to perform the requested serviceability operation.

Name: sec_sys_errno_text
Value: 0x17122f61
The function call returned -1 and errno was set.

Name: sec_sys_errno_text_only
Value: 0x17122f62
The function call returned -1 and errno was set.

Name: sec_sys_file_ftruncate_fail
Value: 0x17122f69
An attempt to truncate the file using the call ftruncate() failed.

Name: sec_sys_file_lseek_fail
Value: 0x17122f63
The file seek failed.

Name: sec_sys_file_open_fail
Value: 0x17122f64
Failed to open the specified file.

Name: sec_sys_file_read_error
Value: 0x17122f67
The requested number of bytes were not read from the file.

Name: sec_sys_file_write_error
Value: 0x17122f68
The requested number of bytes were not written to the file.

Name: sec_thr_alert
Value: 0x17122f42
Thread received an alert exception.

Name: sec_thr_exit_cancel
Value: 0x17122f41
The thread terminated execution because it received a thread cancel exception.

Name: sec_thr_exit_exc
Value: 0x17122f43
The thread terminated execution because it received an exception.

Name: sec_thr_post_cancel
Value: 0x17122f40
Posting a cancel to specified thread.

Glossary

This Glossary is intended to assist understanding and is not a substantive part of this specification.

access
The interaction of a subject with an object. See Section 1.1.3 on page 6.

access control list (ACL)
The matrix of pairs of subjects and objects, whose entries consist of the subjects' permissions to the objects. See Chapter 7, Section 1.1.3 on page 6 and Section 1.8 on page 40.

access determination algorithm
The algorithm in an ACL manager that determines whether the server should grant or deny access. See Section 1.9 on page 46.

ACL manager
A module within an RPC server that interprets ACLs. See Section 1.9 on page 46.

a priori trusted entity
One of a small number of objects whose trust is assumed. See Section 1.1.5 on page 7.

asserted
Sent to the server without authentication. See Section 1.6 on page 25.

assured service
The state of being available and obtainable for use when needed. See Section 1.1.1 on page 4.

attribute
A security aspect of a computer installation that must be protected. Security attributes studied in this specification include authenticity, confidentiality and integrity. See Section 1.1.1 on page 4.

attribute encoding type
A specifier of the data format (integer, string, uuid) of an attribute value. See Section 1.21.7 on page 104.

attribute instance
An attribute type uuid and value created according to the attribute type's semantics and attached to a registry object. Also called *attribute* or *ERA*. See Section 1.21.10 on page 106.

attribute schema
A collection of attribute type definitions or schema entries. Also called a schema. See Section 1.21.1 on page 100.

attribute set
An attribute instance with encoding type **attr_set**. Its value is a list of attribute type UUIDs that identify member attributes of this set. Attribute sets are created for the purpose of efficient queries of related attributes. See Section 1.21.9 on page 106.

attribute type
The description of the identifiers (such as name and UUID) and semantics (such as encoding type and access control parameters) of instances of this type. See Section 1.21.10 on page 106 and Section 1.21.12 on page 108.

attribute type UUID
A DCE UUID that uniquely identifies an attribute type. Also called attribute type ID or attribute ID. See Section 1.21.3 on page 101 and Section 1.21.12 on page 108.

attribute value
The data in an attribute instance.

authenticity
The state of genuinely representing reality, of actually representing that which is alleged to be represented. See Section 1.1.1 on page 4.

authorisation
The state of being granted privilege to access an object. See Section 1.1.3 on page 6.

authorisation data
The portion of a Kerberos ticket that contains data necessary for authorisation decisions. It is sometimes abbreviated *Auth_Data* or *A_D*.

authority
An entity that is trusted to know the secrets of objects other than itself. See Section 1.1.5 on page 7.

call chain
The chain of operations (RPC calls) leading from an initiator to the final target.

cell
The unit of partition of the network TCB. For security purposes, a cell is an *instance* of the three security services, termed the RS / KDS / PS triple, of the security environment. As such, each instance defines a separate cell. See Section 1.2 on page 12.

cell principal
A ticket that is targeted to a KDS server principal. See Section 1.5 on page 18.

certify
To convince a subject of the security of a credential. See Section 1.1.5 on page 7. Certification of login is an optional process undertaken to thwart a type of multi-prong attack described in Section 1.15.2 on page 77.

client
An object acts as a client when it sends an RPC to another object.

compromised
Said of a resource whose security attributes are not adequately protected. See Section 1.1.1 on page 4.

confidentiality
The state of being intrinsically unimpaired. See Section 1.1.1 on page 4.

container object
An object that contains other objects. See Section 1.8.2 on page 44.

credential
An object containing security information about a subject. See Section 1.1.5 on page 7.

cryptography
The science of using secrets to implement security mechanisms. *Cryptanalysis* is the art of analysing cryptographic mechanisms. The two together are *cryptology*. See Section 1.1.6 on page 8.

data encryption standard (DES)
An encryption/decryption algorithm in use since the late 1970's and generally considered secure. See Section 1.4 on page 17.

current login context
The login context automatically inherited by child processes. See Section 1.15 on page 71.

decode/decrypt
The inverse process of encoding or encryption, respectively. See Section 1.1.7 on page 9.

denial of service
The state of being unavailable or unobtainable for use when needed. See Section 1.1.1 on page 4.

delegation
The projection of an initiator's identity to another identity in a manner permitting the other identity to operate on behalf of the initiator.

delegate restrictions
Limits placed upon who may act as an intermediary for a particular identity. See **intermediary**.

delegation token
A checksum over the extended PAC (EPAC) data, encrypted in the PS's key, placed in the A_D field of a PTGT by the priveledge server when enabling delegation and when generating a new delegation chain or impersonated identity. See **impersonation** for the context in which this identity is used.

delegation type
Either *traced delegation* or *impersonation* (only one of which is valid for a given login context).

direct requestor
The client that operates directly on a given target. See *target*.

distributed environment
An environment in which the notion of communication is an explicit model primitive. See Section 1.1.1 on page 4.

distributed time service (DTS)
A secure source of time information which is part of the network TCB. See Section 1.16 on page 80.

domain
The scope of a security policy. See Section 1.1.2 on page 5.

encode
To semantically represent a message by an utterance, where the mapping between message and utterance is secret. See Section 1.1.7 on page 9.

encrypt
To syntactically represent a message by an utterance, where the mapping between message and utterance is secret. Typically, the encoding of the message is not secret. See Section 1.1.7 on page 9.

endianness
An attribute of bit-sequences and byte-sequences on a machine architecture that determines whether the most significant element of the sequence occurs at the high address or at the low address. See Section 2.1.4 on page 128.

EPAC
An Extended PAC available in DCE 1.1 and newer versions, that can contain specified ERAs in addition to the principal's identity and group memberships. A delegation chain is expressed by concatenating the EPACs fro the series of principals involved in an operation. See Section 1.6 on page 25.

environment_set
A set of attributes known to the server; a ''well-known'' ERA. The use of the term *environment* in this document is intended to represent aspects of a login session that are associated with a client principal but whose values are derived from the point of entry the client uses for access. These ''environment attributes'' can have static values, in which case the value is specified by an administrator when defining a point of entry for a host machine and stored in an ERA. Or thay can be dynamic, in which case their value is derived at the time of the specific login attempt and *assigned* to an ERA through the login process.

ERA
Extended Registry Attribute, an attribute (user defined) in the DCE Security Registry (Registry database). It is attached to a registry object, and created using the interfaces defined in this specification. (Also called attribute.) Each ERA has a *schema* entry that is the data dictionary entry defining the attribute type. *Instances* of the attribute containing values can be attached to principal, group, organisation or policy nodes in the Registry database. See Section 1.21 on page 100.

ERA Database
The portion of the Registry database that contains ertended registry attribute information, including schema entries and attribute instances. See Section 1.21 on page 100.

final target
The last object in a call chain.

helpstring
A human-readable string explaining the semantics of a permission in greater detail than does the *printstring*. See Section 1.9 on page 46.

home cell
The cell in whose registry a given principal's security information is held. See Section 1.2 on page 12.

insecure
Said of a resource whose security attributes are not adequately protected. See Section 1.1.1 on page 4.

integrity
The state of being unimpaired. See Section 1.1.1 on page 4.

item
An element of the registry datastore. See Section 1.12 on page 60.

immediate target
The object upon which a client performs an operation directly.

impersonation
Transmission of an initiator's identity such that the identities of participants in a call chain are not preserved.

initiator
The initial client in a call chain.

integrator
A person responsible for porting applications. This person is familiar with both the application to be proted and with the site into which the application is being added. This role involves modifying and recompiling source code.

intermediary
A server acting on behalf of an initiator, via delegation or impersonation, making requests to another target server.

intermediate service
See *intermediary*.

Kerckhoffs´ Doctrine
The idea that the entire algorithm need not be secret, provided a key is. See Section 1.1.8 on page 9.

key
A parameter to an encryption algorithm that suffices to make encryption secure even if the algorithm is not secret. See Section 1.1.8 on page 9.

derived key
A key used for encryption based upon user input, usually a password and a ‘‘confounder’’ or ‘‘salt’’.

strong key
A key that is random and which uses the full key size. These keys are more difficult to break by an intruder.

key management facility
A module that manages long-term cryptographic keys. See Section 1.14 on page 69.

login
A procedure that obtains and validates a login name to provide context for subsequent operations. This specification does not specify a login program or login command, but Section 1.15 on page 71 does list the typical behaviour of such a program or command.

login_set
A set of attributes known to a server, a ‘‘well-known’’ ERA. This set of attributes consists of client specific information derived from the identity of a client. These *login attributes* can have static values, in which case the value is specified by the administrator when defining a user and stored in an ERA. Or they can be dynamic, in which case their values is derived at the time of the specific login attempt and *assigned* to an ERA through the login process.

message
Data in communication. See Section 1.1.7 on page 9.

multi-prong attack
A security attack consisting of a counterfeit login and, simultaneously, malicious RPC servers masquerading as KDS, PS, RS and SCD servers. Defeated by certifying the login, as described in Section 1.15.2 on page 77.

multi-valued attribute
A collection of attribute instances of hte same attribute type attached to a single registry object. See Section 1.21.12.1 on page 108 and Section 1.21.4 on page 102.

name-based authorisation
A primitive authorisation alternative specified in Section 1.6.1 on page 30 but whose use is discouraged.

network login context
The information necessary for a subject to become a client. See Section 1.15 on page 71.

network TCB
Three trusted network services: a Registry, a Key Distribution Service, and a Privilege Service. See Section 1.2 on page 12.

object
The passive aspect of entities whose security attributes are to be protected. See Section 1.1.3 on page 6.

PAC
Privilege Attribute Certificate; the portion of a principal's DCE 1.0 security credentials that provides information about the principal's identity (UUID) and privileges (group memberships). See Section 1.6 on page 25.

pickle
A representation of a data type suitable for storage in the absence of a communications context. See Section 2.1.7 on page 132.

policy
Requirements or rules an organisation places on the security attributes of its assets. See Section 1.1.2 on page 5.

policy object
The registry data node, with the well-known name ''policy'' (under the Security junction point, usually /.:/sec), representing registry-wide policy information. Attributes related to cell-wide security policy should be created on the policy object. See Section 1.21.3 on page 101.

printstring
A human-readable string identifying a permission. See Section 1.9 on page 46.

privilege attribute
That portion of a client's credentials a server uses in access control decisions. See Section 1.6 on page 25.

privilege attribute certificate (PAC)
A certificate specifying the attributes of a client that a server uses to grant or deny access to its protected objects. See Section 1.2 on page 12.

quota
The maximum total number of PGO items plus accounts that may be added to the registry datastore. See Section 11.5 on page 379

PTGT
Privilege Ticket Granting Ticket.

realm
The scope of a security policy. From the strict perspective of security, a *cell* is also known as a *realm* in that it is the security domain of the network TCB. See Section 1.1.2 on page 5.

reference monitor
A trusted subject or entity that mediates all access to a protected object. See Section 1.1.5 on page 7.

registry object
A data node in the Registry database. Registry object are of the object types: principal, group, org, directory, policy, replist (replica list), and attr_schema. There are many nodes of

the principal, group, org and directory types. There is only one node each for the policy, replist and attr_schema types. See Section 1.21 on page 100.

replay attack
A security attack consisting of a retransmission of an intercepted message for the purpose of claiming to be the original sender. Thwarted by use of timestamps, as described in Section 1.16 on page 80.

schema
See attribute schema.

schema entry
A record containing the identifiers and characteristics of an attribute type. A schema entry is essentially an attribute type definition. See Section 1.21.3 on page 101.

schema object
The Registry data node, with the well-known name ‘‘xattrschema’’ (under the Security junction point, typically /.:/sec), containing the attribute schema information. Also called the attribute schema object. See Section 1.21.1 on page 100.

secret
The smallest object whose security is considered tantamount to the security of larger objects by means of trust chains. See Section 1.1.5 on page 7.

secure
Said of a resource whose security attributes are adequately protected. See Section 1.1.1 on page 4.

service
A tool available to enforce a security policy. See Section 1.1.2 on page 5.

session
An interaction between an identified client and a server for a finite time, subject to discrete authentication. See Section 1.2 on page 12.

signature
A keyed cryptographic checksum of a message. See Section 1.3 on page 16.

simple object
An object that does not contain other objects. See Section 1.8.2 on page 44.

site administrator
A person responsible for maintaining user accounts and installing new software packages. This role does not involve any source code modification.

strength
An algorithm’s resistance to cryptanalysis. See Section 1.1.8 on page 9.

subject
The active aspect of entities that interact with objects. See Section 1.1.3 on page 6.

target
Any object that is downstream in a call chain from a given target.

target restrictions
A bound upon the set of targets to whom the client’s identity may be projected.

ticket
A credential certificate representing the authenticated identity of a client. See Section 1.2 on page 12.

traced delegation
A form of delegation that preserves the identities of each participant in a call chain.

transit path
The ordered sequence of KDS servers that vouch for a ticket. See Section 1.5 on page 18.

trigger
A remote operation, associated with an attribute type, that is executed when attributes of that type are either queried or updated. See Section 1.21.8 on page 104.

trigger type
A classification, either ‘‘query’’ or ‘‘update’’, on a trigger that identifies on which attribute operation the trigger will be invoked. See Section 1.21.8.2 on page 105.

trust
Said of a subject that believes an object is secure. See Section 1.1.4 on page 7.

trusted computing base
The fundamental core set of hardware and software that must be trusted. This set is abbreviated (TCB) in this document, and is also referred to as the *network* TCB. See Section 1.1.5 on page 7.

validated login
A login context whose information has been decrypted and is trusted by the associated principal or account. See Section 1.15 on page 71.

weak password
Users typically choose passwords which are derived from words and this makes attacks on passwords easier to break than randomly generated passwords. Not to be confused with *weak key* which is a term used to refer to specific keys and how they are modified by the DES algorithm for encryption.

Index