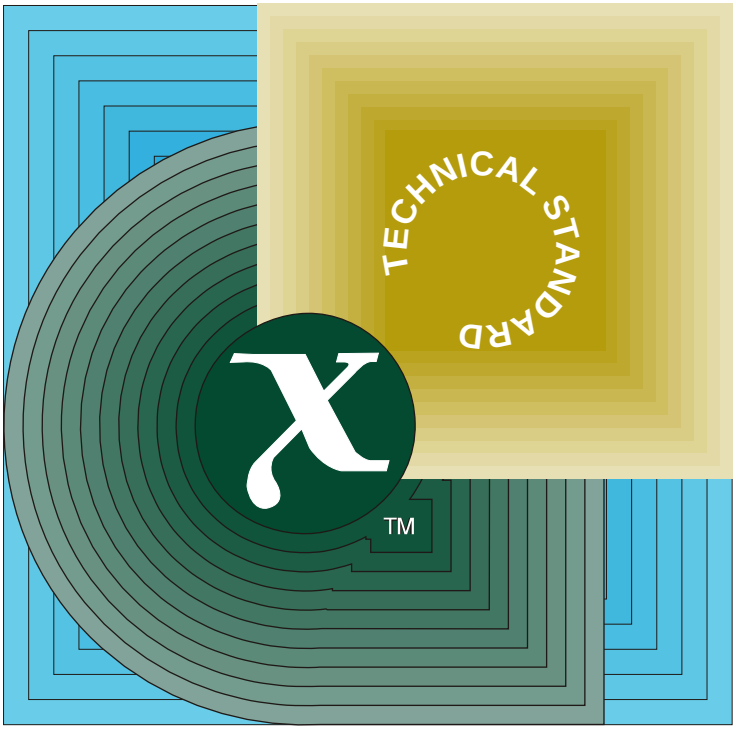
Technical Standard

Generic Security Service API (GSS-API) Base



THE Open GROUP

[This page intentionally left blank]

X/Open CAE Specification

Generic Security Service API (GSS-API)

Base

X/Open Company Ltd.

X/Open CAE Specification

Generic Security Service API (GSS-API) Base

ISBN: 1-85912-131-4

X/Open Document Number: C441

Published by X/Open Company Ltd., U.K.

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

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.org

Contents

List of Figures

List of Tables

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

  CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

  These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done either by email to the X/Open info-server or by checking the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information by email, send a message to info-server@xopen.co.uk with the following in the Subject line:

```
request corrigenda; topic index
```

This will return the index of publications for which Corrigenda exist.

This Document

This document is a *CAE Specification* (see above). It is a merged and reformatted version of the IETF's RFC 1508 and RFC 1509, together with some additional material.

This document is intended for system and application programmers. Readers may find it useful to read Appendix C first.

- Chapter 1 is an introduction to the GSS-API.
- Part 1 :
  - — Chapter 2 covers characteristics and concepts.
  - — Chapter 3 gives interface descriptions.
  - — Chapter 4 provides some example scenarios.
  - — Chapter 5 discusses related activities.
- Part 2 specifies the C-language bindings for GSS-API:
  - — Chapter 6 is an overview.
  - — Chapter 7 covers data types and calling conventions.
  - — Chapter 8 presents the C-language functions in reference manual pages.
  - — Appendix A contains the complete text of the C-language header file.
- Part 3 contains additional material that is not derived from the IETF documents RFC 1508 and RFC 1509:
  - — Appendix B discusses security aspects of GSS-API.
  - — Appendix C describes changes expected in a future version of this document.
  - — A glossary and index are provided.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for filenames, and C-language keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — C-language variable names, for example, substitutable argument prototypes
  - — C-language functions; these are shown as follows: *name*( ).
- 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 a C-language return code 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`*.
- Language-independent functions and arguments use ***bold italic*** font, for example, ***function***( ) and ***argument***.

With the exception of Part 3, which is new material, additional text providing technical enhancement of the source documents is identified by shading and placing the characters EX (meaning extension) in the left margin. For example, this text is an extension. Footnotes are extensions but are not marked. Minor editorial changes are not extensions and are not marked.

EX

Trade Marks

Kerberos™ is a trade mark of the Massachusetts Institute of Technology.

OSF™ is a trade mark of The Open Software Foundation, Inc.

X/Open® is a registered trade mark, and the ‘‘X’’ device is a trade mark, of X/Open Company Limited.

Acknowledgements

X/Open gratefully acknowledges the Internet Engineering Task Force (IETF) for permitting the use of their **Generic Security Service Application Program Interface**, RFC 1508 and **Generic Security Service API: C-bindings**, RFC 1509.

X/Open also gratefully acknowledges the work of the X/Open Security Working Group in the development of this specification.

Document Development

This specification is the result of merging two documents:

IETF RFC 1508
IETF RFC 1509

and adding material required by X/Open.

RFC 1508

The author of this document is John Linn of OpenVision Technologies; it is the result of a collaborative effort. Acknowledgements are due to the many members of the IETF Security Area Advisory Group (SAAG) and the Common Authentication Technology (CAT) Working Group for their contributions at meetings and by electronic mail. Acknowledgements are also due to Kannan Alagappan, Doug Barlow, Bill Brown, Cliff Kahn, Charlie Kaufman, Butler Lampson, Richard Pitkin, Joe Tardo, and John Wray of Digital Equipment Corporation, and John Carr, John Kohl, Jon Rochlis, Jeff Schiller, and Ted T'so of MIT and Project Athena. Joe Pato and Bill Sommerfeld of HP/Apollo, Walt Tuvell of OSF, and Bill Griffith and Mike Merritt of AT&T, provided inputs which helped to focus and clarify directions. Precursor work by Richard Pitkin, presented to meetings of the Trusted Systems Interoperability Group (TSIG), helped to demonstrate the value of a generic, mechanism-independent security service API.

RFC 1509

The author of this document is John Wray of Digital Equipment Corporation.

X/Open Security Working Group

Many members of the X/Open Security Working Group have contributed to this specification, either by providing additional material or by reviewing drafts. In particular, thanks are due to:

Dave Bauer, Bellcore
Denis Pinkas, Groupe Bull
Peter Callaway, IBM
Piers McMahon, ICL
John Linn, OpenVision Technologies
Craig Heath, SCO
Ingo Hoffmann, Siemens-Nixdorf
Joe Brame, Unisys

Referenced Documents

The following documents are referenced in this specification:

ACM

ECMA Draft Standard, Association Context Management — including Security Context Management, Draft 8, May 1993.

ASN.1

ISO 8824: 1990 Information Technology — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1).

BER

ISO / IEC 8825: 1990 (ITU-T Recommendation X.209 (1988)), Information Technology — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).

Extensions

X / Open Snapshot, January 1994, Generic Security Service API (GSS-API) Security Attribute and Delegation Extensions (ISBN: 1-85912-026-1, S307).

ISO / IEC 7498-2

ISO / IEC 7498-2: 1989, Information Processing Systems — Open Systems Interconnection — Basic Reference Model — Part 2: Security Architecture.

RFC 1508

J.Linn, Generic Security Service Application Program Interface, September 1993.

RFC 1509

J.Wray, Generic Security Service API: C-bindings, September 1993.

RFC 1510

Internet Proposed Standard, The Kerberos Network Authentication System, John Kohl, B.Clifford Neuman, issue 5.2, 21 April 1993.

X.509

ISO / IEC 9594-8: 1990 Information Technology — Open Systems Interconnection — The Directory — Part 8: Authentication Framework, together with:

Technical Corrigendum 1: 1991 to ISO / IEC 9594-8: 1990.

XOM

X / Open CAE Specification, November 1991, OSI-Abstract-Data Manipulation API (XOM) (ISBN: 1-872630-17-0, C180 or XO / CAE / 91 / 080).

Chapter 1

Introduction

With the growing awareness of network security threats, providers of distributed computing infrastructure, applications and protocols are increasingly being required to integrate security services into their systems to protect against unauthorised system access, and attacks against user and system data.

This document is the result of merging two separate documents (RFC 1508 and RFC 1509) and adding material required by X/Open.

This chapter explains the status of the source material and outlines the content of each part of this specification. This chapter also outlines the motivation for Security API standardisation and outlines the status of the Security API defined in this specification.

1.1 Status

RFC 1508 and RFC 1509 specify Internet standards track protocol for the Internet community, and request discussion and suggestions for improvements.

1.2 Abstract

Part 1 of this document is the Base Generic Security Service Application Programming Interface (GSS-API) definition.

Typically, GSS-API callers are application protocols into which security enhancements are
EX integrated through the invocation of services provided by the GSS-API. If supported by the underlying mechanism, the GSS-API allows a caller application to do one or more of the following[1]:

- Authenticate a principal identity associated with a peer application.
- Delegate rights to a peer.
- Apply security services such as confidentiality and integrity on a per-message basis.

The interfaces provide a basic set of tools, which is sufficient in many circumstances. However, some security features may be required that are not supported by the Base GSS-API. In particular, the ability to support authorisation based on a subset of the caller privileges, and more sophisticated cases of delegation may be required.

The C-language bindings in Part 2 provide details of the data types, calling conventions and function specifications for GSS-API.

The interfaces in the supplement are currently proposals to enhance GSS-API:

- the use of PACs and Authorisation Services
- additional interfaces for security attributes and delegation.

1. See Section C.5 on page 108 for additional information.

1.3 Motivation for Security API Standardisation

The priority being given to the security of computer systems and networks is growing. This is in part due to the well-publicised exploits of hackers, and threats to systems due to viruses, but it also reflects the increasing trend towards client-server distributed systems. The previously centralised approach of managing corporate data on a single secure mainframe is changing to support devolved processing on departmental servers or PCs. Consequently, the amount of sensitive data crossing insecure networks is growing.

Therefore, there is both a perceived and actual aggravation of the threats to the confidentiality, integrity and availability of users' applications and data. This means that security is featuring increasingly strongly in commercial and non-military government IT requirements. Although the levels of software assurance, and also the security policies that need to be enforced by civil systems differ from those in military systems, common security services such as authentication, secure peer-to-peer communication, access control, audit and delegation are almost always required. Developers of systems meeting these requirements need to take account of the many different technologies being produced to support distributed security, together with the legislative restrictions on cryptography which differ from country to country.

Consideration of these issues, together with the general move towards open systems, has caused increased awareness of the need to isolate application logic from the details of specific security mechanisms. The motivation to agree on a standard API to support distributed security services stems from the real needs of current commercial and military developments. In a number of predominantly U.S.-based industry and open systems standards bodies, activities are underway to scope the subject of security APIs, and define standards. The work in the Internet Engineering Task Force (IETF) has been highly influential in defining the base GSS-API, and further work continues in the IETF both on exploiting the GSS-API, and consideration of possible extensions supportive of distributed authorisation services. Also, in a European standards context, following a well-attended workshop convened in late 1992, the CEN / CENELEC ITAEGV set up a special group to consider the appropriate standardisation needed for Security APIs.

1.4 GSS-API and Current Standardisation Activities

It looks certain that the GSS-API will form the basis of standards in distributed security APIs, for the following reasons:

- Specifications submitted as IETF Internet-Drafts in early 1993 under Common Authentication Technology working group jurisdiction are now Proposed Standard RFCs. This has been based on trial implementations over Kerberos and DASS.
- OSF DCE1.1 includes an implementation of GSS-API to allow non-RPC applications access to the DCE security services.
- The May 1992 meeting of the ISO SC21 WG6 Upper Layer Security Rapporteur's group produced a new work item proposal (SC21 N6998) for **Authentication and Related Security Services for Distributed Applications**, which would include a mechanism-independent abstract service interface. This was based on input from the U.K. representative derived from GSS-API (see the GSAI paper).
- The ECMA TC36/TG9 **Security in Open Systems** group developing the Association Context Management standard is defining this standard in such a way that it can support GSS-API (see the ACM draft standard).
- The SESAME project (based on ECMA TC36/TG9 **Security in Open Systems** work) is supporting and originating extensions to the GSS-API.

X/Open CAE Specification

Part 1

GSS-API

X/Open Company Ltd.

Chapter 2

GSS-API Characteristics and Concepts

This chapter discusses the scope and purpose, operational paradigm and goals of GSS-API. It also describes the constructs and features of GSS-API, and comments on design issues.

2.1 Purpose

EX The purpose of the GSS-API is to provide a standard application programming interface to certain communication-oriented security services. An implementation of GSS-API, together with the necessary cryptographic algorithms and protocols, permits callers portably and interoperably[2] to counter a range of vulnerabilities that may affect the security of communication between applications in open networks.

Some specific threats addressed by GSS-API (assuming the appropriate functionality is supported by underlying mechanisms) are given below. Note that the last three threats may occur accidentally as well as maliciously, depending on the application environment.

- masquerade: unauthorised assertion of an identity by one peer to another peer
- disclosure: unauthorised breach of the confidentiality of a message sent between peers, typically through interception of communications in transit
- replay: unauthorised reuse of a message sent from one peer to another (for example, a message may be intercepted by a third party and retransmitted to a communicating peer at a later time)
- falsification of origin: unauthorised sending of a message from one peer to another, falsely claiming that it originated from a third peer without this being detected by the recipient
- modification: unauthorised or accidental alteration of the contents of a message in transit between two peers
- non-delivery: unauthorised or accidental misrouting or dropping of a message in transit without the intended recipient being aware of this
- re-ordering: unauthorised or accidental reordering of messages in transit without either peer being aware of this.

The countermeasures to these threats are mechanism-specific. The application is insulated from the details of how the mechanism implements security through the GSS-API.

2. Interoperability is only possible between communicating peers using GSS-API implementations that support the necessary interoperable cryptographic algorithms and protocols. GSS-API allows peers to negotiate a common security mechanism (if they possess one) thereby enabling interoperability.

2.2 Scope

EX The scope of GSS-API addresses protection of communication between distributed applications; it does not comprise an interface to other non-communication oriented security facilities within hosts. In particular:

- Use of GSS-API is relevant to communication software, and to those components of distributed applications that implement applications protocols.
- GSS-API is designed for use between pairs of communicating peers in a direct on-line client-server or message-passing environment (such as ftp or transaction processing systems). GSS-API is appropriate for use with both connectionless and connection-oriented protocols, but is not applicable to queued store-and-forward applications such as electronic mail, nor to a broadcast environment.

  The callers of the GSS-API must take responsibility for construction of application protocol messages by combining application-specific control and data elements with the necessary security structures generated by GSS-API.
- Generally, use of the GSS-API sequencing facilities is most appropriate when GSS-API is called from protocol modules whose message exchanges assume ordered sequence semantics rather than a datagram environment.

2.2.1 Non-goals

GSS-API does not:

- provide for a specific non-repudiation service but could be used as an underlying technology to support non-repudiation
- provide time-stamping
- enforce security directly — it provides services to security enforcing applications which are expected to use those services correctly.

2.3 GSS-API (Base) — Conformance

This section defines conformance criteria for implementations of the GSS-API, and also mechanism-independent use of the GSS-API by applications.

The following GSS-API implementation conformance levels are defined:

- The minimum conformance criteria which are prescribed for implementations to meet the Authentication and Integrity requirements of the X/Open GSS-API (Base) specification
- The conformance criteria for three optional services Confidentiality, Non Confidentiality and Delegation which implementations may independently support. Confidentiality and Non Confidentiality are mutually exclusive options.
- If confidentiality/non confidentiality or delegation or both are not supported then calls for these services should fail.

In addition, for GSS-API-using applications:

- The conformance criteria for use of the GSS-API by Portable Applications.

2.3.1 GSS-API (Base) — Minimal Implementation Conformance

All conforming GSS-API (Base) implementations **must** support:

- unilateral and mutual authentication
- integrity protection of messages
- replay detection or out-of-sequence detection.

All conforming GSS-API (Base) implementations **must**:

- provide a confirmed indication of the set of services available to a peer initiating a security context when mutual authentication is requested.

  In particular, where a successfully completed call to ***gss_init_sec_context*** is made, provide a correct indication via ***ret_flags*** of whether confidentiality or delegation is supported by the mechanism *and* the context acceptor.

2.3.2 GSS-API (Base) — Confidentiality Conformance

Two interworking GSS-API (Base) implementations for a mechanism which supports the Confidentiality option **must** permit a context initiator to optionally confidentiality-protect its data sent to a context acceptor.

Hence Confidentiality option implementations must:

- accept GSS_C_CONF_FLAG in ***gss_init_sec_context req_flags***, and return this option in ***ret_flags*** from ***gss_init_sec_context*** and ***gss_accept_sec_context*** respectively
- support the GSS-API (Base) caller requesting confidentiality by accepting ***conf_req_flag*** == TRUE in ***gss_wrap***
- return TRUE in the conf_state output from ***gss_wrap***, and ***gss_unwrap*** when this service is requested.

2.3.3 GSS-API (Base) — Non-confidentiality

Two interworking GSS-API (Base) implementations for a mechanism which supports the Non-confidentiality option **must not** permit a context initiator to confidentiality-protect its data sent to a context acceptor.

Hence Non Confidentiality option implementations must:

- process GSS_C_CONF_FLAG in ***gss_init_sec_context req_flags***, but not return this option in ***ret_flags*** from ***gss_init_sec_context*** and ***gss_accept_sec_context*** respectively
- not support the GSS-API (Base) caller requesting confidentiality by not accepting ***conf_req_flag*** == TRUE in ***gss_wrap***
- return FALSE in the *conf_state* output from ***gss_wrap***, and ***gss_unwrap*** and not providing the requested service.

2.3.4 GSS-API (Base) — Delegation Conformance

Two interworking GSS-API (Base) implementations for a mechanism which supports the Delegation option **must** permit a context initiator to optionally delegate its credential to a context acceptor.

Hence Delegation option implementations must:

- support the GSS-API (Base) caller requesting delegation by accepting the GSS_C_DELEG bit in ***req_flags***
- return GSS_C_DELEG in the ***ret_flags*** output from ***gss_init_sec_context*** and ***gss_accept_sec_context***, and return a valid ***delegated_cred_handle*** from ***gss_accept_sec_context***, when this service is requested.

2.4 GSS-API (Base) — Portable Application Conformance

If you use applications which use GSS-API (Base) but require no portability then you may skip this section.

Applications which use GSS-API (Base) and are required to be portable **must**:

- isolate the main application logic from the syntax of the ***input_name_buffer*** which is provided when the initiator calls ***gss_import_name*** to obtain an internal representation of the acceptor name.

  For maximal portability the default ***name_type*** GSS_C_NULL_OID should always be used to specify the default name space, and the input ***name_string*** should be treated by the client as an opaque name-space specific input.

- adopt maximally portable usage of credentials (which should support most GSS-API initiators in a typical single user session or workstation).

  This requires that the initiator should call ***gss_init_sec_context***, with GSS_C_NO_CREDENTIAL into ***cred_handle*** to specify the default credential, and GSS_C_NULL_OID into ***mech_type*** to specify the default mechanism. For maximal portability, the ***gss_accept_sec_context*** verifier ***cred_handle*** should be set to GSS_C_NO_CREDENTIAL.

- use standard levels of security appropriate to distributed applications.

  The context initiator must therefore specify its requirements for replay protection, delegation, and sequence protection via the ***gss_init_sec_context*** *req_flags* parameter. The context initiator should always request these service options (i.e. passes GSS_C_MUTUAL_FLAG | GSS_C_REPLAY_FLAG | GSS_C_SEQUENCE_FLAG into ***req_flags***).

- in the case of a strictly conforming application, additionally use appropriate defaults

- use channel bindings for the ***application_data***, and for the GSS_C_AF_NULL_ADDR and GSS_C_AF_INET address types only (noting that the latter only applies to IPV4).

  Conforming portable GSS-API applications must not otherwise use channel bindings for addresses — as the syntaxes are not defined.

2.5 Operational Paradigm

The operational paradigm in which GSS-API operates is illustrated in Figure 2-1.

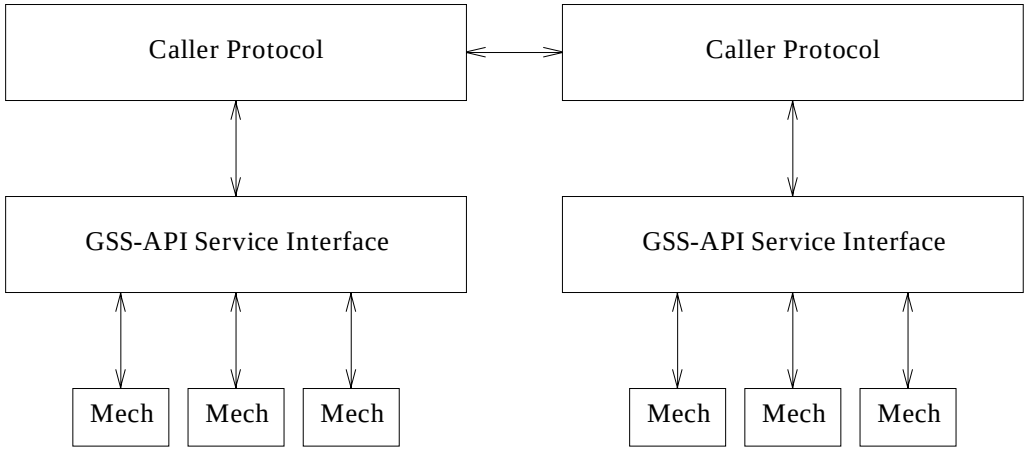


Figure 2-1 GSS-API Paradigm

A typical GSS-API caller is itself a communication protocol, calling on GSS-API in order to protect its communication with authentication, integrity and confidentiality security services. A GSS-API caller accepts tokens provided to it by its local GSS-API implementation and transfers the tokens to a peer on a remote system; that peer passes the received tokens to its local GSS-API implementation for processing. The security services available through GSS-API in this fashion are implementable (and have been implemented) over a range of underlying mechanisms based on secret-key and public-key cryptographic technologies.

This security service definition, and other definitions used in this document, correspond to that provided in the ISO / IEC 7498-2 standard. The GSS-API separates the operations of initialising a security context between peers, achieving peer entity authentication (the ***gss_init_sec_context***( ) and ***gss_accept_sec_context***( ) calls), from the operations of providing per-message data origin authentication and data integrity protection (the ***gss_get_mic***[3]( ) and ***gss_verify_mic***[4]( ) calls) for messages subsequently transferred in conjunction with that context. Per-message ***gss_wrap***[5]( ) and ***gss_unwrap***[6]( ) calls provide the data origin authentication and data integrity services that ***gss_get_mic***( ) and ***gss_verify_mic***( ) offer, and also support selection of confidentiality services as a caller option. Additional calls provide supporting functions to the GSS-API's users.

EX When establishing a security context, the GSS-API enables a context initiator optionally to permit its credentials to be delegated, meaning that the context acceptor may initiate further security contexts on behalf of the initiating caller. Figure 2-2 on page 13 provides an example illustrating the data flows involved in the use of the GSS-API by a client and server in a mechanism-independent fashion: establishing a security context and transferring a protected message. The example assumes that credential acquisition has already been completed. The

3. The old message name *gss_sign*( ) is also implemented.
4. The old message name *gss_verify*( ) is also implemented.
5. The old message name *gss_seal*( ) is also implemented.
6. The old message name *gss_unseal*( ) is also implemented. See the note on terminology in Appendix C on page 107.

example also assumes that the underlying authentication technology is capable of authenticating a client to a server using elements carried within a single token, and of authenticating the server to the client (mutual authentication) with a single returned token; this assumption holds for certain documented mechanisms but is not necessarily true for other cryptographic technologies and associated protocols.

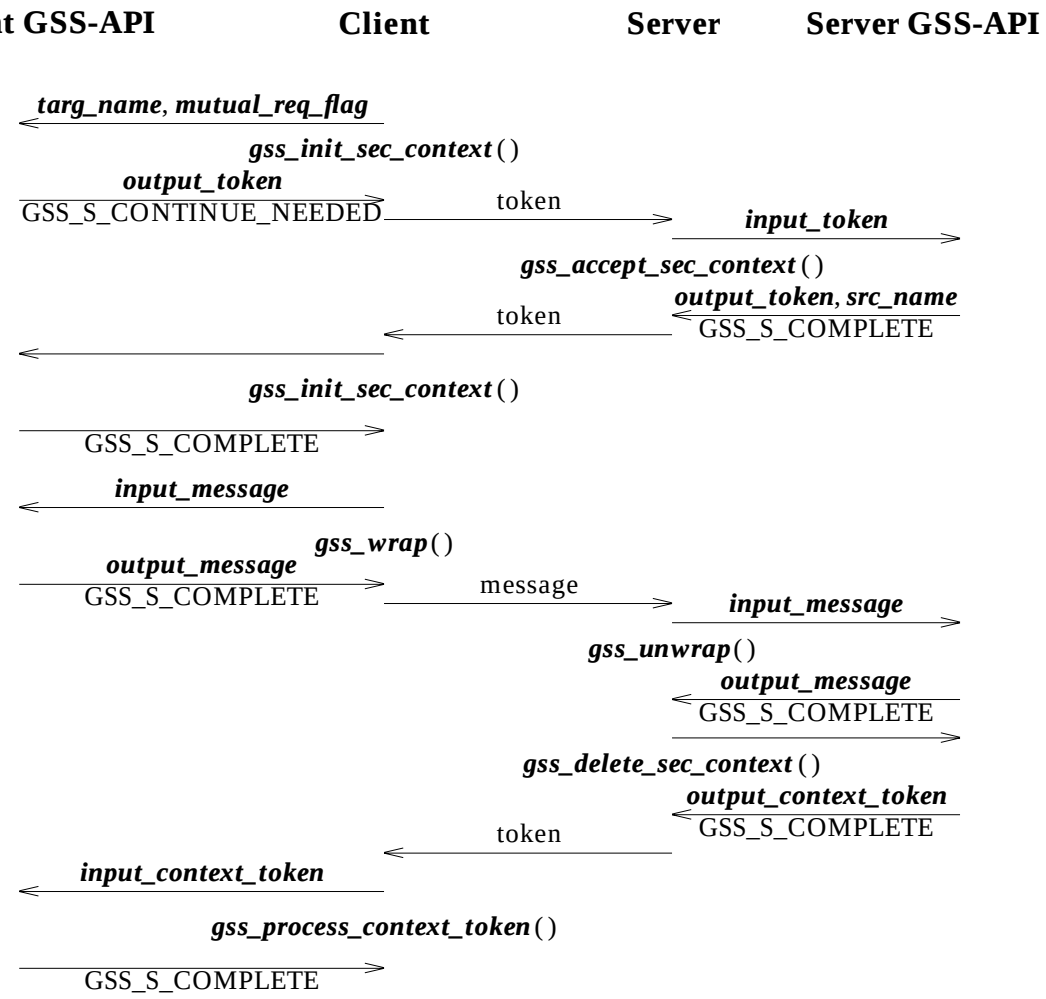


Figure 2-2 GSS-API Used by Client and Server

The client calls ***gss_init_sec_context*** ( ) to establish a security context to the server identified by ***targ_name***, and elects to set the ***mutual_req_flag*** so that mutual authentication is performed in the course of context establishment. ***gss_init_sec_context*** ( ) returns an ***output_token*** to be passed to the server, and indicates GSS_S_CONTINUE_NEEDED status pending completion of the mutual authentication sequence. If ***mutual_req_flag*** is not set, the initial call to ***gss_init_sec_context*** ( ) returns GSS_S_COMPLETE status. The client sends the ***output_token*** to the server.

The server passes the received token as the ***input_token*** parameter to ***gss_accept_sec_context*** ( ). ***gss_accept_sec_context*** ( ) indicates GSS_S_COMPLETE status, provides the client's authenticated identity in the ***src_name*** result, and provides an ***output_token*** to be passed to the client. The server sends the ***output_token*** to the client.

The client passes the received token as the ***input_token*** parameter to a successor call to ***gss_init_sec_context*** ( ), which processes data included in the token to achieve mutual authentication from the client's viewpoint. This call to ***gss_init_sec_context*** ( ) returns GSS_S_COMPLETE status, indicating successful mutual authentication and the completion of context establishment for this example.

The client generates a data message and passes it to ***gss_wrap*** ( ). ***gss_wrap*** ( ) performs data origin authentication, data integrity, and optionally, confidentiality processing on the message, and encapsulates the result into ***output_message***, indicating GSS_S_COMPLETE status. The client sends the ***output_message*** to the server.

The server passes the received message to ***gss_unwrap*** ( ). ***gss_unwrap*** ( ) inverts the encapsulation performed by ***gss_wrap*** ( ), deciphers the message if the optional confidentiality feature is applied, and validates the data origin authentication and data integrity checking quantities. ***gss_unwrap*** ( ) indicates successful validation by returning GSS_S_COMPLETE status along with the resultant ***output_message.***

For the purposes of this example, the server is assumed to know by out-of-band means that this context has no further use after one protected message is transferred from client to server. Given this premise, the server now calls ***gss_delete_sec_context*** ( ) to flush context-level information. ***gss_delete_sec_context*** ( ) returns an ***output_context_token*** for the server to pass to the client.

The client passes the returned token as ***input_context_token*** to ***gss_process_context_token*** ( ), which returns GSS_S_COMPLETE status after deleting context-level information at the client system.

2.6 Goals

The GSS-API design assumes and addresses several basic goals, including:

Mechanism independence

The GSS-API defines an interface to cryptographically-implemented strong authentication and other security services at a generic level that is independent of particular underlying mechanisms. For example, services provided by GSS-API can be implemented by secret-key technologies (for example, Kerberos) or public-key approaches (for example, X.509).

Protocol environment independence

The GSS-API is independent of the communication protocol suites with which it is employed, permitting use in a broad range of protocol environments. In appropriate
EX environments, GSS-API need not be directly invoked by applications, but may form an intermediate layer that is indirectly invoked. For example, in an RPC environment, GSS-API may be layered beneath RPC (or above it).

Protocol association independence

The GSS-API's security context construct is independent of communication protocol association constructs. This characteristic allows a single GSS-API implementation to be utilised by a variety of invoking protocol modules on behalf of those modules' calling applications. GSS-API services can also be invoked directly by applications, wholly independent of protocol associations.

Suitability to a range of implementation placements

GSS-API clients are not constrained to reside within any Trusted Computing Base (TCB) perimeter defined on a system where the GSS-API is implemented; security services are specified in a manner suitable to both intra-TCB and extra-TCB callers.

2.7 GSS-API Constructs

The basic elements comprising the GSS-API are credentials, tokens, security contexts, mechanism types, names and channel bindings.

2.7.1 Credentials

Credentials structures provide the prerequisites enabling peers to establish security contexts with each other. A caller may designate that its default credential be used for context establishment calls without presenting an explicit handle to that credential. Alternatively, those GSS-API callers that need to make explicit selection of particular credentials structures may make references to those credentials through GSS-API-provided credential handles; a credential handle argument is termed ***cred_handle.***

A single credential structure may be used for initiation of outbound contexts and acceptance of inbound contexts. Callers needing to operate in only one of these modes may designate this fact when credentials are acquired for use, allowing underlying mechanisms to optimise their processing and storage requirements. The credential elements defined by a particular mechanism may contain multiple cryptographic keys, for example, to enable authentication and message encryption to be performed with different algorithms.

A single credential structure may accommodate credential information associated with multiple underlying mechanisms; a credential structure's contents vary depending on the set of mechanism types supported by a particular GSS-API implementation. Commonly, a single mechanism type is used for all security contexts established by a particular initiator to a particular target. The primary motivation for supporting credential sets representing multiple mechanism types is to allow initiators on systems equipped to handle multiple types to initiate contexts to targets on other systems that can accommodate only a subset of the set supported at the initiator's system.

It is the responsibility of underlying system-specific mechanisms and operating system functions below the GSS-API to ensure that the ability to acquire and use credentials associated with a given identity is constrained to appropriate processes within a system. This responsibility should be taken seriously by implementors, as the ability for an entity to utilise a principal's credentials is equivalent to the entity's ability to assert that principal's identity successfully.

Once a set of GSS-API credentials is established, the transferability of that credentials set to other processes or analogous constructs within a system is a local matter, not defined by the GSS-API. An example local policy would be one in which any credentials received as a result of login to a given user account, or of delegation of rights to that account, are accessible by, or transferable to, processes running under that account.

The credential establishment process (particularly when performed on behalf of users rather than server processes) is likely to require access to passwords or other quantities which should be protected locally and exposed for the shortest time possible. As a result, it is often appropriate for preliminary credential establishment to be performed through local means at user login time, with the results cached for subsequent reference. These preliminary credentials would be set aside (in a system-specific fashion) for subsequent use, in one of the following ways:

- to be accessed by an invocation of the GSS-API ***gss_acquire_cred*** ( ) call, returning an explicit handle to reference that credential
- as the default credentials installed on behalf of a process.

2.7.2 Tokens

Tokens are data elements transferred between GSS-API callers, and are divided into two classes. Context-level tokens are exchanged in order to establish and manage a security context between peers. Per-message tokens are exchanged in conjunction with an established context to provide protective security services for corresponding data messages. The internal contents of both classes of tokens are specific to the particular underlying mechanism used to support the GSS-API; Section 5.2 on page 36 provides a uniform recommendation for designers of GSS-API support mechanisms, encapsulating mechanism-specific information along with a globally-interpretable mechanism identifier.

Tokens are opaque from the viewpoint of GSS-API callers. They are generated within the GSS-API implementation at an end system, provided to a GSS-API caller to be transferred to the peer GSS-API caller at a remote end system, and processed by the GSS-API implementation at that remote end system. Tokens may be output by GSS-API primitives (and are to be transferred to GSS-API peers) independent of the status indications indicated by those primitives. Token transfer may take place in an in-band manner, integrated into the same protocol stream used by the GSS-API callers for other data transfers, or in an out-of-band manner across a logically separate channel.

Development of GSS-API support primitives based on a particular underlying cryptographic technique and protocol does not necessarily imply that GSS-API callers invoking that GSS-API mechanism type are able to interoperate with peers invoking the same technique and protocol outside the GSS-API paradigm. For example, the format of GSS-API tokens defined in conjunction with a particular mechanism, and the techniques used to integrate those tokens into callers' protocols, may not be the same as those used by non-GSS-API callers of the same underlying technique.

2.7.3 Security Contexts

Security contexts are established between peers, using credentials established locally in conjunction with each peer or received by peers by means of delegation. Multiple contexts may exist simultaneously between a pair of peers, using the same or different sets of credentials. Coexistence of multiple contexts using different credentials allows graceful roll over when credentials expire. Distinction between multiple contexts based on the same credentials serves applications by distinguishing different message streams in a security sense.

The GSS-API is independent of underlying protocols and addressing structure, and depends on its callers to transport data elements provided by GSS-API. As a result of these factors, it is a caller responsibility to parse communicated messages, separating GSS-API-related data elements from caller-provided data. The GSS-API is independent of connection as opposed to connectionless orientation of the underlying communication service.

No correlation between security context and communication protocol association is dictated. The optional channel binding facility, discussed in Section 2.7.6 on page 19, represents an intentional exception to this rule, supporting additional protection features within GSS-API supporting mechanisms. This separation allows the GSS-API to be used in a wide range of communication environments, and also simplifies the calling sequences of the individual calls. In many cases (dependent on underlying security protocol, associated mechanism, and availability of cached information), the state information required for context set-up can be sent concurrently with initial signed user data, without interposing additional message exchanges.

2.7.4 Mechanism Types

In order to establish a security context successfully with a target peer, it is necessary to identify an appropriate underlying mechanism type supported by both initiator and target peers; a mechanism type argument is termed ***mech_type***. The definition of a mechanism embodies not only the use of a particular cryptographic technology (or a hybrid or choice between possible cryptographic technologies), but also definition of the syntax and semantics of data element exchanges employed by that mechanism to support security services.

It is recommended that callers initiating contexts specify the *default* mechanism type value, allowing system-specific functions within or invoked by the GSS-API implementation to select the appropriate mechanism type, but callers may direct that a particular mechanism type be employed when necessary.

The means for identifying a shared mechanism type to establish a security context with a peer varies in different environments and circumstances; examples include (but are not limited to):

- use of a fixed mechanism type, defined by configuration, within an environment
- syntactic convention on a target-specific basis, through examination of a target's name
- lookup of a target's name in a naming service or other database to identify mechanism types supported by that target
- explicit negotiation between GSS-API callers in advance of security context setup.

When transferred between GSS-API peers, mechanism type specifiers (represented as Object Identifiers (OIDs), as described in Section 5.2 on page 36) serve to qualify the interpretation of associated tokens. The structure and encoding of Object Identifiers is defined in the ASN.1 standard and the BER standard. Use of hierarchically structured OIDs serves to preclude ambiguous interpretation of mechanism type specifiers. The OID representing the DASS MechType, for example, is 1.3.12.2.1011.7.5.

2.7.5 Naming

The GSS-API avoids prescription of naming structures, treating the names transferred across the interface in initiating and accepting security contexts as opaque octet string quantities. This approach supports the GSS-API's goal of implementability on top of a range of underlying security mechanisms, recognising the fact that different mechanisms process and authenticate names presented in different forms. Generalised services offering translation functions between arbitrary sets of naming environments are outside the scope of the GSS-API; the availability and use of local conversion functions to translate between the naming formats supported within a given end system is expected.

Two distinct classes of name representations are used in conjunction with different GSS-API parameters:

- a printable form (denoted by OCTET STRING), for acceptance from and presentation to users; printable name forms are accompanied by OID tags identifying the name space to which they correspond
- an internal form (denoted by INTERNAL NAME), opaque to callers and defined by individual GSS-API implementations; GSS-API implementations supporting multiple name space types are responsible for maintaining internal tags to remove ambiguity from the interpretation of particular names.

Tagging of printable names allows GSS-API callers and underlying GSS-API mechanisms to disambiguate name types and to determine whether an associated name's type is one they are capable of processing, avoiding aliasing problems, which could result from misinterpreting a

name of one type as a name of another type.

In addition to providing means for names to be tagged with types, this specification defines primitives to support a level of naming environment independence for certain calling applications. To provide basic services oriented towards the requirements of callers, which need not themselves interpret the internal syntax and semantics of names, GSS-API calls are defined for:

- name comparison (***gss_compare_name***( ))
- human-readable display (***gss_display_name***( ))
- input conversion (***gss_import_name***( ))
- internal name deallocation (***gss_release_name***( )).

It is expected that these GSS-API calls will be implemented in many end systems based on system-specific name manipulation primitives already extant within those end systems. Inclusion within the GSS-API is intended to offer GSS-API callers a portable means to perform specific operations, supportive of authorisation and audit requirements, on authenticated names.

gss_import_name( ) implementations can, where appropriate, support more than one printable syntax corresponding to a given name space (for example, alternative printable representations for X.500 Distinguished Names), allowing flexibility for their callers to select between possible representations. ***gss_display_name***( ) implementations output a printable syntax selected as appropriate to their operational environments; this selection is a local matter. Callers desiring portability across alternative printable syntaxes should refrain from implementing comparisons based on printable name forms and should instead use the ***gss_compare_name***( ) call to determine whether or not one internal name format matches another.

2.7.6 Channel Bindings

The GSS-API accommodates the concept of caller-provided channel binding information, used by GSS-API callers to bind the establishment of a security context to relevant characteristics (for example, addresses, transformed representations of encryption keys) of the underlying communication channel and of protection mechanisms applied to that communication channel. Verification by one peer of channel binding information provided by the other peer to a context serves to protect against various active attacks. The caller initiating a security context must determine the channel binding values before making the ***gss_init_sec_context***( ) call, and consistent values must be provided by both peers to a context. Callers should not assume that underlying mechanisms provide confidentiality protection for channel binding information.

Use or non-use of the GSS-API channel binding facility is a caller option, and GSS-API supporting mechanisms can support operation in an environment where NULL channel bindings are presented. When non-NULL channel bindings are used, certain mechanisms offer enhanced security value by interpreting the bindings' content (rather than simply representing those bindings, or signatures[7] computed on them, within tokens) and therefore depend on presentation of specific data in a defined format. To this end, agreements among mechanism implementors define conventional interpretations for the contents of channel binding arguments, including address specifiers (with content dependent on the communication protocol environment) for context initiators and acceptors. These conventions are being

7. The term *signature* means Message Integrity Code (MIC) or cryptographic checkvalue (see Section C.1 on page 107).

incorporated into related documents. For GSS-API callers to be portable across multiple mechanisms and achieve the full security functionality available from each mechanism, it is strongly recommended that GSS-API callers provide channel bindings consistent with these conventions and those of the networking environment in which they operate.

2.8 GSS-API Features and Issues

2.8.1 Status Reporting

Each GSS-API call provides two status return values. The ***major_status*** value provides a mechanism-independent indication of call status (for example, GSS_S_COMPLETE, GSS_S_FAILURE, GSS_S_CONTINUE_NEEDED), sufficient to drive normal control flow within the caller in a generic fashion. The defined ***major_status*** return codes are listed below[8].

Fatal Error Codes	**Meaning**
GSS_S_BAD_BINDINGS	Channel binding mismatch.
GSS_S_BAD_MECH	Unsupported mechanism requested.
GSS_S_BAD_NAME	Invalid name provided.
GSS_S_BAD_NAMETYPE	Name of unsupported type provided.
GSS_S_BAD_STATUS	Invalid input status selector.
GSS_S_BAD_SIG	Token had invalid signature.
GSS_S_CONTEXT_EXPIRED	Specified security context expired.
GSS_S_CREDENTIALS_EXPIRED	Expired credentials detected.
GSS_S_DEFECTIVE_CREDENTIAL	Defective credential detected.
GSS_S_DEFECTIVE_TOKEN	Defective token detected.
GSS_S_FAILURE	Failure, unspecified at GSS-API level.
GSS_S_NO_CONTEXT	No valid security context specified.
GSS_S_NO_CRED	No valid credentials provided.

Informative Status Codes	**Meaning**
GSS_S_COMPLETE	Normal completion.
GSS_S_CONTINUE_NEEDED	Continuation call to routine required.
GSS_S_DUPLICATE_TOKEN	Duplicate per-message token detected.
GSS_S_OLD_TOKEN	Timed-out per-message token detected.
GSS_S_UNSEQ_TOKEN	Out-of-order per-message token detected.

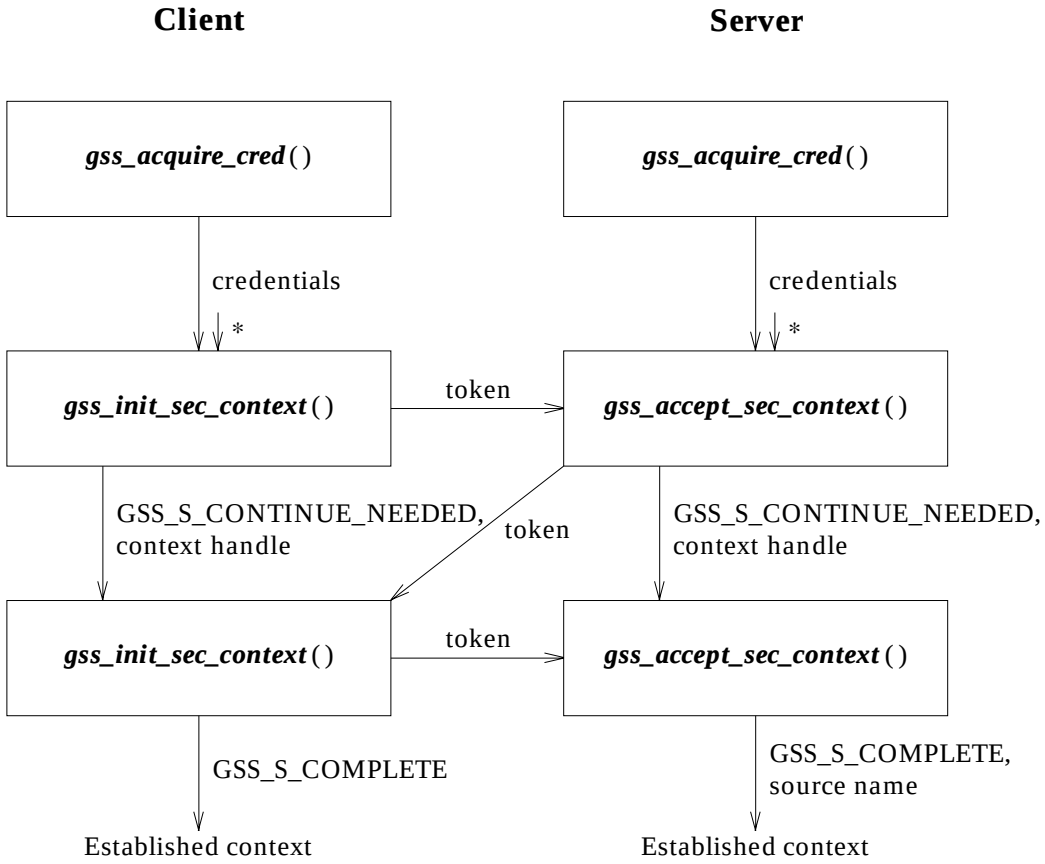
The ***minor_status*** provides more detailed status information; this may include status codes specific to the underlying security mechanism. Values for ***minor_status*** are not specified in this language-independent specification.

GSS_CONTINUE_NEEDED ***major_status*** returns, and optional message outputs, are provided in ***gss_init_sec_context***( ) and ***gss_accept_sec_context***( ) calls so that different mechanisms' employment of different numbers of messages within their authentication sequences need not be reflected in separate code paths within calling applications. Instead, such cases are accommodated with sequences of continuation calls to ***gss_init_sec_context***( ) and ***gss_accept_sec_context***( ). The same mechanism is used to encapsulate mutual authentication within the GSS-API's context initiation calls.

For mechanism types that require interactions with third-party servers to establish a security context, GSS-API context establishment calls may block pending completion of such third-party interactions. On the other hand, no GSS-API calls depend on serialised interactions with GSS-API peer entities. As a result, local GSS-API status returns cannot reflect unpredictable or asynchronous exceptions occurring at remote peers, and reflection of such status information is a caller responsibility outside the GSS-API. Figure 2-3 on page 22 illustrates a GSS-API

8. A new ***major_status*** code may be included at a later date to identify a missing token (see Section C.1 on page 107).

continuation scenario.



∗ There may be repeated calls to ***gss_init_sec_context*** ( ) and ***gss_accept_sec_context*** ( )

Figure 2-3 Example Context Establishment with Continuation

2.8.2 Per-message Security Service Availability

When a context is established, two flags are returned to indicate the set of per-message protection security services available on the context:

- the ***integ_avail*** flag indicates whether per-message integrity and data origin authentication services are available
- the ***conf_avail*** flag indicates whether per-message confidentiality services are available, and is never returned TRUE unless the ***integ_avail*** flag is also returned TRUE.

GSS-API callers desiring per-message security services should check the values of these flags at context establishment time. Callers must be aware that a returned FALSE value for ***integ_avail*** means that invocation of ***gss_get_mic*** ( ) or ***gss_wrap*** ( ) primitives on the associated context applies no cryptographic protection to user data messages.

The GSS-API per-message protection service primitives, as the category name implies, are oriented to operation at the granularity of protocol data units. They perform cryptographic operations on the data units, transfer cryptographic control information in tokens, and in the case of ***gss_wrap*** ( ), encapsulate the protected data unit. As such, these primitives are not oriented to efficient data protection for stream-paradigm protocols (for example, Telnet) if cryptography must be applied on an octet-by-octet basis.

EX If only integrity protection is required, this can be provided in two ways:

- The first approach is to obtain a signature over a message using the ***gss_get_mic***( ) call; then to verify the signature against the message using the ***gss_verify_mic***( ) call.
- The second approach is to obtain a single opaque object that contains both the message and the signature using the ***gss_wrap***( ) call; then to recover the message from the opaque object with the appropriate status information using the ***gss_unwrap***( ) call.

If both integrity protection and confidentiality protection are required, the ***gss_wrap***( ) or ***gss_unwrap***( ) call is used to provide and respectively check or recover the opaque object.

2.8.3 Per-message Replay Detection and Sequencing

Certain underlying mechanism types are expected to offer support for replay detection or sequencing of messages transferred on the contexts they support. These optionally-selectable protection features are distinct from replay detection and sequencing features applied to the context establishment operation itself. The presence or absence of context-level replay or sequencing features is wholly a function of the underlying mechanism type's capabilities, and is not selected or omitted as a caller option.

The caller initiating a context provides flags (***replay_det_req_flag*** and ***sequence_req_flag***) to specify whether the use of per-message replay detection and sequencing features is desired on the context being established. The GSS-API implementation at the initiator system can determine whether these features are supported (and whether they are optionally selectable) as a function of the mechanism type, without the need for bilateral negotiation with the target. When enabled, these features provide recipients with indicators as a result of GSS-API processing of incoming messages, identifying whether those messages were detected as duplicates or out-of-sequence. Detection of such events does not prevent a suspect message from being provided to a recipient; the appropriate course of action on a suspect message is a matter of caller policy.

The semantics of the replay detection and sequencing services applied to received messages, as visible across the interface provided by GSS-API to its clients, are determined by the value of ***replay_det_state*** and ***sequence_state***.

When ***replay_det_state*** is TRUE, the possible ***major_status*** values for well-formed and correctly signed messages[9] are as follows:

- GSS_S_COMPLETE indicates that the message is within the window (of time or sequence space) allowing replay events to be detected, and that the message is not a replay of a previously-processed message within that window.
- GSS_S_DUPLICATE_TOKEN indicates that the signature on the received message is correct, but that the message is recognised as a duplicate of a previously-processed message.
- GSS_S_OLD_TOKEN indicates that the signature on the received message is correct, but that the message is too old to be checked for duplication.

9. The term *signed messages* means integrity-protected messages (see Section C.1 on page 107).

When ***sequence_state*** is TRUE, the possible ***major_status*** values for well-formed and correctly signed messages are as follows:

- GSS_S_COMPLETE indicates that the message is within the window (of time or sequence space) allowing replay events to be detected, and that the message is not a replay of a previously-processed message within that window.
- GSS_S_DUPLICATE_TOKEN indicates that the signature on the received message is correct, but that the message is recognised as a duplicate of a previously-processed message.
- GSS_S_OLD_TOKEN indicates that the signature on the received message is correct, but that the token is too old to be checked for duplication.
- GSS_S_UNSEQ_TOKEN indicates that the signature on the received message is correct, but that it is earlier in a sequenced stream than a message already processed on the context.

  Note: Mechanisms can be architected to provide a stricter form of sequencing service, delivering particular messages to recipients only after all predecessor messages in an ordered stream have been delivered. This type of support is incompatible with the GSS-API paradigm in which recipients receive all messages, whether in order or not, and provide them (one at a time, without intra-GSS-API message buffering) to GSS-API routines for validation. GSS-API facilities provide supportive functions, aiding clients to achieve strict message stream integrity in an efficient manner in conjunction with sequencing provisions in communication protocols, but the GSS-API does not offer this level of message stream integrity service by itself.

As the message stream integrity features (especially sequencing) may interfere with certain applications' intended communication paradigms, and since support for such features is likely to be resource intensive, it is highly recommended that mechanism types supporting these features allow them to be activated selectively on initiator request when a context is established. A context initiator and target are provided with corresponding indicators (***replay_det_state*** and ***sequence_state***), signifying whether these features are active on a given context.

An example mechanism type supporting per-message replay detection could (when ***replay_det_state*** is TRUE) implement the feature as follows. The underlying mechanism inserts timestamps in data elements output by ***gss_get_mic***( ) and ***gss_wrap***( ), and maintains (within a time-limited window) a cache (qualified by originator-recipient pair) identifying received data elements processed by ***gss_verify_mic***( ) and ***gss_unwrap***( ). When this feature is active, exception status returns (GSS_S_DUPLICATE_TOKEN, GSS_S_OLD_TOKEN) are provided when ***gss_verify_mic***( ) or ***gss_unwrap***( ) is presented with a message that is either a detected duplicate of a prior message or too old to validate against a cache of recently received messages.

2.8.4 Quality of Protection

Some mechanism types provide their users with fine granularity control over the means used to provide per-message protection, allowing callers to trade off security processing overhead dynamically against the protection requirements of particular messages. A per-message quality-of-protection (QOP) parameter (analogous to quality-of-service, or QOS) selects between different QOP options supported by that mechanism. On context establishment for a multi-QOP mechanism type, context-level data provides the prerequisite data for a range of protection qualities.

It is expected that the majority of callers do not wish to exert explicit mechanism-specific QOP control and therefore request selection of a default QOP. Definitions of, and choices between, non-default QOP values are mechanism-specific, and no ordered sequences of QOP values can be assumed equivalent across different mechanisms. Meaningful use of non-default QOP values demands that callers be familiar with the QOP definitions of an underlying mechanism or mechanisms, and is therefore a non-portable construct.

Chapter 3

Interface Descriptions

This chapter describes the GSS-API's service interface, dividing the calls offered into four groups. Credential-management calls are related to the acquisition and release of credentials by principals. Context-level calls are related to the management of security contexts between principals. Per-message calls are related to the protection of individual messages on established security contexts. Support calls provide ancillary functions useful to GSS-API callers. The calls are summarised below:

Credential-management Calls	**Function**
gss_acquire_cred ( )	acquire credentials for use
gss_release_cred ( )	release credentials after use
gss_inquire_cred ( )	display information about credentials
Context-level Calls	**Function**
gss_init_sec_context ( )	initiate outbound security context
gss_accept_sec_context ( )	accept inbound security context
gss_delete_sec_context ( )	flush context when no longer needed
gss_process_context_token ( )	process received control token on context
gss_context_time ( )	indicate validity time remaining on context
Per-message Calls	**Function**
gss_get_mic ( )	apply signature, receive as token separate from message
EX	(generate a checkvalue token separate from message)
EX ***gss_verify_mic*** ( )	validate signature (checkvalue) token along with message
gss_wrap ( )	sign, optionally encrypt, encapsulate
EX	(apply integrity and optionally confidentiality, encapsulate a message within a token)
gss_unwrap ( )	decapsulate, decrypt if needed, validate signature
EX	(recover a message from a token, verify integrity)
Support Calls	**Function**
gss_display_status ( )	translate status codes to printable form
gss_indicate_mechs ( )	indicate mechanism types supported on local system
gss_compare_name ( )	compare two names for equality
gss_display_name ( )	translate name to printable form
gss_import_name ( )	convert printable name to normalised form
gss_release_name ( )	free storage of normalised-form name
gss_release_buffer ( )	free storage of printable name
gss_release_oid_set ( )	free storage of OID set object

3.1 Credential-management Calls

These GSS-API calls provide functions related to the management of credentials. Their characterisation with regard to whether or not they may block pending exchanges with other network entities (for example, directories or authentication servers) depends in part on issues specific to the operating system, therefore outside the GSS-API, and not specified here.

The ***gss_acquire_cred***( ) call is defined within the GSS-API in support of application portability, EX with a particular orientation towards support of portable server applications. It is typically only called by clients when they want to use other than the default credentials. It is recognised that (for certain systems and mechanisms) credentials for interactive users may be managed differently from credentials for server processes; in such environments, it is the GSS-API implementation's responsibility to distinguish these cases and the procedures for making this distinction are a local matter. The ***gss_release_cred***( ) call provides a means for callers to indicate to the GSS-API that use of a credentials structure is no longer required. The ***gss_inquire_cred***( ) call allows callers to determine information about a credentials structure.

3.2 Context-level Calls

This group of calls is devoted to the establishment and management of security contexts between peers. A context's initiator calls ***gss_init_sec_context***( ), resulting in generation of a token which the caller passes to the target. At the target, that token is passed to ***gss_accept_sec_context***( ). Dependent on the underlying mechanism type and specified options, additional token exchanges may be performed in the course of context establishment; such exchanges are accommodated by GSS_S_CONTINUE_NEEDED status returns from ***gss_init_sec_context***( ) and ***gss_accept_sec_context***( ). Either party to an established context may invoke ***gss_delete_sec_context***( ) to flush context information when a context is no longer required. ***gss_process_context_token***( ) is used to process received tokens carrying context-level control information. ***gss_context_time***( ) allows a caller to determine the length of time for which an established context remains valid.

3.3 Per-message Calls

This group of calls is used to perform per-message protection processing on an established security context. None of these calls block pending network interactions. These calls may be invoked by a context's initiator or by the context's target. The four members of this group should be considered as two pairs: the output from ***gss_get_mic***( ) is properly input to ***gss_verify_mic***( ); the output from ***gss_wrap***( ) is properly input to ***gss_unwrap***( ).

gss_get_mic( ) and ***gss_verify_mic***( ) support data origin authentication and data integrity services. When ***gss_get_mic***( ) is invoked on an input message, it yields a per-message token containing data items that allow underlying mechanisms to provide the specified security services. The original message, along with the generated per-message token, is passed to the remote peer; these two data elements are processed by ***gss_verify_mic***( ), which validates the message in conjunction with the separate token.

gss_wrap( ) and ***gss_unwrap***( ) support caller-requested confidentiality in addition to the data origin authentication and data integrity services offered by ***gss_get_mic***( ) and ***gss_verify_mic***( ). ***gss_wrap***( ) outputs a single data element, encapsulating optionally enciphered user data as well as associated token data items. The data element output from ***gss_wrap***( ) is passed to the remote peer and processed by ***gss_unwrap***( ) at that system. ***gss_unwrap***( ) combines decipherment (as required) with validation of data items related to authentication and integrity.

3.4 Support Calls

This group of calls provides support functions useful to GSS-API callers, independent of the state of established contexts. Their characterisation with regard to blocking or non-blocking status in terms of network interactions is unspecified.

Chapter 4

Mechanism-specific Example Scenarios

This chapter provides illustrative overviews of the use of various candidate mechanism types to support the GSS-API. These discussions are intended primarily for readers familiar with specific security technologies, demonstrating how GSS-API functions can be used and implemented by candidate underlying mechanisms. They should not be regarded as constrictive to implementations or as defining the only means through which GSS-API functions can be realised with a particular underlying technology, and do not demonstrate all GSS-API features with each technology.

4.1 Kerberos V5, Single-TGT

This example illustrates the use of the GSS-API in conjunction with Kerberos Version 5 (see RFC 1510).

Login functions specific to the operating system yield a TGT to the local realm Kerberos server; TGT is placed in a credentials structure for the client. The client calls ***gss_acquire_cred*** ( ) to acquire a credential handle to reference the credentials for use in establishing security contexts.

The client calls ***gss_init_sec_context*** ( ). If the requested service is located in a different realm, ***gss_init_sec_context*** ( ) gets the necessary TGT and key pairs needed to traverse the path from local to target realm; this data is placed in the owner's TGT cache. After any needed remote realm resolution, ***gss_init_sec_context*** ( ) yields a service ticket to the requested service with a corresponding session key; this data is stored in conjunction with the context. GSS-API code sends KRB_TGS_REQ requests and receives KRB_TGS_REP responses (in the successful case) or KRB_ERROR.

Assuming success, ***gss_init_sec_context*** ( ) builds a Kerberos-formatted KRB_AP_REQ message, and returns it in ***output_token***. The client sends the ***output_token*** to the service.

The service passes the received token as the ***input_token*** argument to ***gss_accept_sec_context*** ( ), which verifies the authenticator, provides the service with the client's authenticated name, and returns an ***output_context_handle***.

Both parties now hold the session key associated with the service ticket, and can use this key in subsequent ***gss_get_mic*** ( ), ***gss_verify_mic*** ( ), ***gss_wrap*** ( ) and ***gss_unwrap*** ( ) operations.

4.2 Kerberos V5, Double-TGT

TGT acquisition is as described in Section 4.1 on page 31.

Note: To avoid unnecessary frequent invocations of error paths when implementing the GSS-API on top of Kerberos V5, it seems appropriate to represent single-TGT K-V5 and double-TGT K-V5 with separate mechanism types; this discussion makes that assumption.

Based on the (specified or default) mechanism type, ***gss_init_sec_context*** ( ) determines that the double-TGT protocol should be employed for the specified target. ***gss_init_sec_context*** ( ) returns GSS_S_CONTINUE_NEEDED in ***major_status***, and its returned ***output_token*** contains a request to the service for the service's TGT. If a service TGT with suitably long remaining lifetime already exists in a cache, it may be usable, obviating the need for this step. The client passes the ***output_token*** to the service.

Note: This scenario illustrates a different use for the GSS_S_CONTINUE_NEEDED status return facility from that used for support of mutual authentication. Both uses can coexist as successive operations within a single context establishment operation.

The service passes the received token as the ***input_token*** argument to ***gss_accept_sec_context*** ( ), which recognises it as a request for TGT.

Note: Current Kerberos V5 defines no intra-protocol mechanism to represent such a request.

gss_accept_sec_context ( ) returns GSS_S_CONTINUE_NEEDED in ***major_status*** and provides the service's TGT in its ***output_token***. The service sends the ***output_token*** to the client.

The client passes the received token as the ***input_token*** argument to a continuation of ***gss_init_sec_context*** ( ). ***gss_init_sec_context*** ( ) caches the received service TGT and uses it as part of a service ticket request to the Kerberos authentication server, storing the returned service ticket and session key in conjunction with the context. ***gss_init_sec_context*** ( ) builds a Kerberos-formatted authenticator, and returns it in ***output_token*** along with GSS_S_COMPLETE in ***major_status***. The client sends the ***output_token*** to the service.

The service passes the received token as the ***input_token*** argument to a continuation call to ***gss_accept_sec_context*** ( ). ***gss_accept_sec_context*** ( ) verifies the authenticator, provides the service with the client's authenticated name, and returns GSS_S_COMPLETE in ***major_status***.

gss_get_mic ( ), ***gss_verify_mic*** ( ), ***gss_wrap*** ( ) and ***gss_unwrap*** ( ) are used as described in Section 4.1 on page 31.

4.3 X.509 Authentication Framework

This example illustrates the use of the GSS-API in conjunction with public-key mechanisms, consistent with the X.509 Directory Authentication Framework (see the X.509 standard).

The ***gss_acquire_cred***( ) call establishes a credentials structure, making the client's private key accessible for use on behalf of the client.

The client calls ***gss_init_sec_context***( ), which interrogates the Directory to acquire (and validate) a chain of public-key certificates, thereby collecting the public key of the service. The certificate validation operation determines that suitable signatures were applied by trusted authorities and that those certificates have not expired. ***gss_init_sec_context***( ) generates a secret key for use in per-message protection operations on the context, and enciphers that secret key under the service's public key.

The enciphered secret key, along with an authenticator quantity signed with the client's private key, is included in the ***output_token*** from ***gss_init_sec_context***( ). The ***output_token*** also carries a certification path, consisting of a certificate chain leading from the service to the client; a different approach would defer this path resolution to be performed by the service instead of being asserted by the client. The client application sends the ***output_token*** to the service.

The service passes the received token as the ***input_token*** argument to ***gss_accept_sec_context***( ). ***gss_accept_sec_context***( ) validates the certification path, and as a result determines a certified binding between the client's distinguished name and the client's public key. Given that public key, ***gss_accept_sec_context***( ) can process the ***input_token*** argument's authenticator quantity and verify that the client's private key was used to sign the ***input_token***. At this point, the client is authenticated to the service. The service uses its private key to decipher the enciphered secret key provided to it for per-message protection operations on the context.

The client calls ***gss_get_mic***( ) or ***gss_wrap***( ) on a data message, which causes per-message authentication, integrity, and optionally, confidentiality facilities to be applied to that message. The service uses the context's shared secret key to perform corresponding ***gss_verify_mic***( ) and ***gss_unwrap***( ) calls.

Chapter 5

Related Activities

This chapter considers the additional requirements for GSS-API to be implemented on top of existing, emerging and future security mechanisms. It also discusses design constraints.

Concrete language bindings are required for the programming environments in which GSS-API is to be employed; such bindings for the C language are provided in Part 2.

5.1 Identification

Object identifiers must be assigned to candidate GSS-API mechanisms and the name types they support. Concrete data element formats must be defined for candidate mechanisms. Future versions of this specification will reference appropriate standards as they exist at the time.

Calling applications must implement formatting conventions that enable them to distinguish GSS-API tokens from other data carried in their application protocols (see Section 5.2 on page 36).

5.2 Mechanism-independent Token Format

This token format defined is a mechanism-independent level of encapsulating representation for the initial token of a GSS-API context establishment sequence, incorporating an identifier of the mechanism type to be used on that context. Use of this format (with ASN.1-encoded data elements represented in BER, constrained in the interests of parsing simplicity to the Distinguished Encoding Rule (DER) BER subset defined in X.509, clause 8.7) is recommended to the designers of GSS-API implementations based on various mechanisms, so that tokens can be interpreted unambiguously at GSS-API peers. There is no requirement that the mechanism-specific innerContextToken, innerMsgToken and sealedUserData data elements be encoded in ASN.1 BER.

```
-- optional top-level token definitions to
-- frame different mechanisms

GSS-API DEFINITIONS ::=

BEGIN

MechType ::= OBJECT IDENTIFIER
-- data structure definitions

-- callers must be able to distinguish among
-- InitialContextToken, SubsequentContextToken,
-- PerMsgToken, and SealedMessage data elements
-- based on the usage in which they occur

InitialContextToken ::=
-- option indication (delegation, etc.) indicated within
-- mechanism-specific token
[APPLICATION 0] IMPLICIT SEQUENCE {
  thisMech MechType,
  innerContextToken ANY DEFINED BY thisMech
   -- contents mechanism-specific
  }

SubsequentContextToken ::= innerContextToken ANY
-- interpretation based on predecessor InitialContextToken

PerMsgToken ::=
-- as emitted by gss_get_mic and processed by gss_verify_mic
  innerMsgToken ANY

SealedMessage ::=
-- as emitted by gss_wrap and processed by gss_unwrap
-- includes internal, mechanism-defined indicator
-- of whether or not encrypted
  sealedUserData ANY

END
```

5.3 Mechanism Design Constraints

The following constraints on GSS-API mechanism designs are adopted in response to observed caller protocol requirements, and adherence thereto is expected in subsequent descriptions of GSS-API mechanisms, to be documented in future publications:

- Use of the approach defined in Section 5.2 on page 36, applying a mechanism type tag to the InitialContextToken, is required.
- It is strongly recommended that mechanisms offering per-message protection services also offer at least one of the replay detection and sequencing services, as mechanisms offering neither fail to satisfy recognised requirements of certain candidate caller protocols.

X/Open CAE Specification

Part 2

C-language Bindings

X/Open Company Ltd.

Chapter 6

GSS-API C-language Overview

This chapter explains how the C-language functions are used and provides a brief description of each routine.

6.1 Using the C-language Functions

The GSS-API C-language functions provide security services to calling applications. A communicating application can authenticate the entity associated with its peer, can delegate rights to another application, and can apply security services such as confidentiality and integrity on a per-message basis.

There are four stages to using the C-language GSS-API functions:

1. If the use of non-default credential elements is required, the application acquires a set of credentials (*gss_acquire_cred*( )) with which it demonstrates an associated identity to a peer. The application's credentials vouch for its global identity, which may or may not be related to the local user name under which it is running.

2. A pair of communicating applications establish a joint security context using their credentials (*gss_init_sec_context*( ) and *gss_accept_sec_context*( )). The security context is a pair of GSS-API data structures that contain shared state information, which is required for per-message security services.

   As part of the establishment of a security context, the context initiator is authenticated to the responder, and may require that the responder be authenticated in turn (*gss_init_sec_context*( ) and *gss_accept_sec_context*( )). The initiator may optionally give the responder the right to initiate further security contexts. This transfer of rights is termed delegation, and is achieved by creating a set of credentials, similar to those used by the originating application, but which may be used by the responder .

   To manage the state of a context once established, certain GSS-API calls return a token data structure that carries control information. The caller of such an GSS-API routine is responsible for transferring the token to the peer application, which should then pass it to a corresponding GSS-API routine (*gss_process_context_token*( )) which processes the information appropriately. The token is cryptographically protected so it can be transmitted over an insecure link.

3. Per-message services are invoked to apply either integrity and data origin authentication, or confidentiality, integrity and data origin authentication to application data, which is treated by GSS-API as arbitrary octet strings. The application transmitting a message that it wishes to protect calls the appropriate GSS-API routine (*gss_get_mic*( ) or *gss_wrap*( )) to apply protection, specifying the appropriate security context, and sends the result to the receiving application. The receiver passes the received data to the corresponding decoding routine (*gss_verify_mic*( ) or *gss_unwrap*( )) to remove the protection and validate the data.

4. At the completion of a communication session (which may extend across several connections), the peer applications call GSS-API routines to delete the security context (*gss_delete_sec_context*( )). Multiple contexts may also be used (either successively or simultaneously) within a single communication association.

6.2 GSS-API C-language Routines

The GSS-API C-language routines are listed below:

Routine	Function
gss_acquire_cred( )	assume a global identity
gss_release_cred( )	discard credentials
gss_init_sec_context( )	initiate a security context with a peer application
gss_accept_sec_context( )	accept a security context initiated by a peer application
gss_process_context_token( )	process a token on a security context from a peer application
gss_delete_sec_context( )	discard a security context
gss_context_time( )	determine for how long a context remains valid
gss_get_mic( )	sign a message; integrity service
gss_verify_mic( )	check signature on a message
gss_wrap( )	sign (optionally encrypt) a message; confidentiality service
gss_unwrap( )	verify (optionally decrypt) message
gss_display_status( )	convert an API status code to text
gss_indicate_mechs( )	determine underlying authentication mechanism
gss_compare_name( )	compare two internal names
gss_display_name( )	convert opaque name to text
gss_import_name( )	convert a textual name to internal form
gss_release_name( )	discard an internal name
gss_release_buffer( )	discard a buffer
gss_release_oid_set( )	discard a set of object identifiers
gss_inquire_cred( )	determine information about a credential

Individual GSS-API implementations may augment these routines by providing additional mechanism-specific routines if the required capability is not available from the generic forms. Applications are encouraged to use the generic routines wherever possible on portability grounds.

Chapter 7

Data Types and Calling Conventions

This chapter describes the data types used by the C-language versions of the GSS-API functions. It also explains calling conventions for these functions.

7.1 Structured Data Types

Wherever these GSS-API C-bindings describe structured data, only fields that must be provided by all GSS-API implementations are documented. Individual implementations may provide additional fields, either for internal use within GSS-API routines, or for use by non-portable applications.

7.2 Integer Types

GSS-API defines the following integer data type:

```
OM_uint32     32-bit unsigned integer
```

Where guaranteed minimum bit-count is important, this portable data type is used by the GSS-API routine definitions. Individual GSS-API implementations include appropriate **typedef** definitions to map this type onto a built-in data type.

7.3 String and Similar Data

Many of the GSS-API routines take arguments and return values that describe contiguous multi-byte data. All such data is passed between the GSS-API and the caller using the **gss_buffer_t** data type. This data type is a pointer to a buffer descriptor consisting of a **length** field, which contains the total number of bytes in the data, and a **value** field, which contains a pointer to the actual data:

```
typedef struct gss_buffer_desc_struct{
   size_t   length;
   void    *value;
} gss_buffer_desc, *gss_buffer_t;
```

Storage for data passed to the application by a GSS-API routine using the **gss_buffer_t** conventions is allocated by the GSS-API routine. The application may free this storage by invoking the *gss_release_buffer*( ) routine. Allocation of the **gss_buffer_desc** object is always the responsibility of the application; unused **gss_buffer_desc** objects may be initialised to the value GSS_C_EMPTY_BUFFER.

7.3.1 Opaque Data Types

Certain multi-octet data items are considered opaque data types at the GSS-API, because their internal structure has no significance either to the GSS-API or to the caller. Examples of such opaque data types are the *input_token* argument to *gss_init_sec_context*( ) (which is opaque to the caller), and the *input_message* argument to *gss_wrap*( ) (which is opaque to the GSS-API). Opaque data is passed between the GSS-API and the application using the **gss_buffer_t** datatype.

7.3.2 Character Strings

Certain multi-octet data items may be regarded as simple Latin-1 character strings as defined in the ISO/IEC 8859-1 standard. An example of this is the *input_name_buffer* argument to *gss_import_name*( ). Some GSS-API routines also return character strings. Character strings are passed between the application and the GSS-API using the **gss_buffer_t** data type, defined earlier.

7.4 Object Identifiers

Certain GSS-API procedures take arguments of the type **gss_OID**, or object identifier. This is a type containing ISO-defined tree-structured values, and is used by the GSS-API caller to select an underlying security mechanism. A value of type **gss_OID** has the following structure:

```
typedef struct gss_OID_desc_struct{
   OM_uint32  length;
   void      *elements;
} gss_OID_desc, *gss_OID;
```

The **elements** field of this structure points to the first byte of an octet string containing the ASN.1 BER encoding of the value of the **gss_OID**. The **length** field contains the number of bytes in this value. For example, the **gss_OID** value corresponding to:

```
{iso(1) identified-organization(3) icd-ecma(12)
member-company(2) dec(1011) cryptoAlgorithms(7) SPX(5)}
```

meaning SPX (Digital's X.509 authentication mechanism) has a length field of 7 and an elements field pointing to seven octets containing the following octal values: 53,14,2,207,163,7,5. GSS-API implementations should provide constant **gss_OID** values to allow callers to request any supported mechanism, although applications are encouraged on portability grounds to accept the default mechanism. **gss_OID** values should also be provided to allow applications to specify particular name types (see Section 7.10 on page 50). Applications should treat **gss_OID_desc** values returned by GSS-API routines as read-only. In particular, the application should not attempt to deallocate them. The **gss_OID_desc** data type is equivalent to the X/Open **OM_object_identifier** datatype (XOM).

7.5 Object Identifier Sets

Certain GSS-API procedures take arguments of the type **gss_OID_set**. This type represents one or more object identifiers (see Section 7.4). A **gss_OID_set** object has the following structure:

```
typedef struct gss_OID_set_desc_struct{
   int       count;
   gss_OID   elements;
} gss_OID_set_desc, *gss_OID_set;
```

EX The **count** field contains the number of OIDs within the set. The **elements** field is type **gss_OID**, which is a pointer to an array of **gss_OID_desc** objects, each of which describes a single OID. Type **gss_OID_set** values are used to name the available mechanisms supported by the GSS-API, to request the use of specific mechanisms, and to indicate which mechanisms a given credential supports. Storage associated with **gss_OID_set** values returned to the application by the GSS-API may be deallocated by the *gss_release_oid_set*( ) routine.

7.6 Credentials

A credential handle is a caller-opaque atomic datum that identifies a GSS-API credential data structure. It is represented by the caller-opaque type **gss_cred_id_t**, which may be implemented as either an arithmetic or a pointer type. Credentials describe a principal, and they give their holder the ability to act as that principal. The GSS-API does not make the actual credentials available to applications; instead the credential handle is used to identify a particular credential, held internally by GSS-API or the underlying mechanism. Thus the credential handle contains no security-relevant information, and requires no special protection by the application. Dependent on the implementation, a given credential handle may refer to different credentials when presented to the GSS-API by different callers. Individual GSS-API implementations should define both the scope of a credential handle and the scope of a credential itself (which must be at least as wide as that of a handle). Possibilities for credential handle scope include the process that acquired the handle, the acquiring process and its children, or all processes sharing some local identification information (for example, UID). If no handles exist by which a given credential may be reached, the GSS-API may delete the credential.

Certain routines allow credential handle arguments to be omitted to indicate the use of a default credential. The mechanism by which a default credential is established and its scope should be defined by the individual GSS-API implementation.

7.7 Contexts

The **gss_ctx_id_t** data type contains a caller-opaque atomic value that identifies one end of a GSS-API security context. It may be implemented as either an arithmetic or a pointer type. Depending on the implementation, a given **gss_ctx_id_t** value may refer to different GSS-API security contexts when presented to the GSS-API by different callers. The security context holds state information about each end of a peer communication, including cryptographic state information. Individual GSS-API implementations should define the scope of a context. Since no way is provided by which a new **gss_ctx_id_t** value may be obtained for an existing context, the scope of a context should be the same as the scope of a **gss_ctx_id_t**.

7.8 Authentication Tokens

A token is a caller-opaque type that GSS-API uses to maintain synchronisation between the context data structures at each end of a GSS-API security context. The token is a cryptographically protected bit-string, generated by the underlying mechanism at one end of a GSS-API security context for use by the peer mechanism at the other end. Encapsulation (if required) and transfer of the token are the responsibility of the peer applications. A token is passed between the GSS-API and the application using the **gss_buffer_t** conventions.

7.9 Status Values

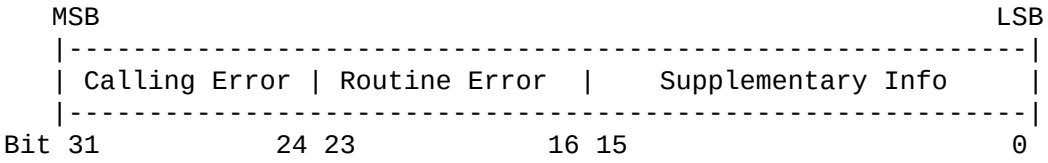
One or more status codes are returned by each GSS-API routine. Two distinct sorts of status code are returned. These are termed GSS status codes and mechanism status codes.

7.9.1 GSS Status Codes

GSS-API routines return GSS status codes as their **OM_uint32** function value. These codes indicate major status errors that are independent of the underlying mechanism used to provide the security service. The errors that can be indicated by means of a GSS status code are either generic API routine errors (errors that are defined in the GSS-API specification) or calling errors (errors that are specific to these bindings[10]).

A GSS status code can indicate a single fatal generic API error from the routine and a single calling error. In addition, supplementary status information may be indicated by setting bits in a **Supplementary Info** field in a GSS status code.

These errors are encoded into the 32-bit GSS status code as follows:



Hence if a GSS-API routine returns a GSS status code whose upper 16 bits contain a non-zero value, the call failed. If the **Calling Error** field is non-zero, the invoking application's call of the routine was erroneous. Calling errors are defined in Table 7-1. If the **Routine Error** field is non-zero, the routine failed for one of the routine-specific reasons listed in Table 7-2 on page 48. Whether or not the upper 16 bits indicate a failure or a success, the routine may indicate additional information by setting bits in the **Supplementary Info** field of the status code. The meaning of individual bits is listed in Table 7-3 on page 48.

Name	Value in Field	Meaning
[GSS_S_CALL_INACCESSIBLE_READ]	1	A required input argument cannot be read.
[GSS_S_CALL_INACCESSIBLE_WRITE]	2	A required output argument cannot be written.
[GSS_S_CALL_BAD_STRUCTURE]	3	An argument is malformed.

Table 7-1 Calling Errors

10. The included C bindings.

Name	Value in Field	Meaning
[GSS_S_BAD_MECH]	1	Unsupported mechanism requested.
[GSS_S_BAD_NAME]	2	Invalid name supplied.
[GSS_S_BAD_NAMETYPE]	3	A name supplied is of an unsupported type.
[GSS_S_BAD_BINDINGS]	4	Incorrect channel bindings supplied.
[GSS_S_BAD_STATUS]	5	Invalid status code supplied.
[GSS_S_BAD_SIG]	6	A token has an invalid signature.
[GSS_S_NO_CRED]	7	No credentials supplied.
[GSS_S_NO_CONTEXT]	8	No context established.
[GSS_S_DEFECTIVE_TOKEN]	9	Token invalid.
[GSS_S_DEFECTIVE_CREDENTIAL]	10	Credential invalid.
[GSS_S_CREDENTIALS_EXPIRED]	11	The referenced credentials have expired.
[GSS_S_CONTEXT_EXPIRED]	12	The context has expired.
[GSS_S_FAILURE]	13	Miscellaneous failure (see text).

Table 7-2 Routine Errors

Name	Value in Field	Meaning
[GSS_S_CONTINUE_NEEDED]	0 (LSB)	The routine must be called again to complete its function. See individual function descriptions in Chapter 8 on page 55 for a detailed description.
[GSS_S_DUPLICATE_TOKEN]	1	The token is a duplicate of an earlier token.
[GSS_S_OLD_TOKEN]	2	The token's validity period has expired.
[GSS_S_UNSEQ_TOKEN]	3	A later token has already been processed.

Table 7-3 Supplementary Status Bits

The function specifications also use the name [GSS_S_COMPLETE], which is a zero value, to indicate an absence of any API errors or supplementary information bits.

All [GSS_S_*] symbols equate to complete **OM_uint32** status codes, rather than to bit-field values. For example, the actual value of the symbol [GSS_S_BAD_NAMETYPE] (value 3 in the **Routine Error** field) is 3 << 16.

The macros[11]:

GSS_C_CALLING_ERROR( )
GSS_C_ROUTINE_ERROR( )
GSS_C_SUPPLEMENTARY_INFO( )

are provided, each of which takes a GSS status code and removes all but the relevant field. For example, the value obtained by applying GSS_C_ROUTINE_ERROR( ) to a status code removes the **Calling Errors** and **Supplementary Info** fields, leaving only the **Routine Errors** field. The values delivered by these macros may be directly compared with a [GSS_S_*] symbol of the

11. See Section C.4 on page 108.

appropriate type. The macro GSS_C_ERROR( ) is also provided, which when applied to a GSS status code returns a non-zero value if the status code indicates a calling or routine error, and a zero value otherwise.

A GSS-API implementation may choose to signal calling errors in a platform-specific manner instead of, or in addition to the routine value; routine errors and supplementary information should be returned by means of routine status values only.

7.9.2 Mechanism-specific Status Codes

GSS-API C-language functions return a *minor_status* argument, which is used to indicate specialised errors from the underlying security mechanism. This argument may contain a single mechanism-specific error, indicated by an **OM_uint32** value.

The *minor_status* argument is always set by a GSS-API function, even if it returns a calling error or one of the generic API errors indicated above as fatal, although other output arguments may remain unset in such cases. However, output arguments that are expected to return pointers to storage allocated by a function must always be set by the function, even in the event of an error, although in such cases the GSS-API function may elect to set the returned argument value to NULL to indicate that no storage was actually allocated. Any length field associated with such pointers (as in a **gss_buffer_desc** structure) should also be set to zero in such cases.

The GSS status code [GSS_S_FAILURE] is used to indicate that the underlying mechanism detected an error for which no specific GSS status code is defined. The mechanism status code provides more details about the error.

7.10 Names

A name is used to identify a person or entity. GSS-API authenticates the relationship between a name and the entity claiming the name.

Two distinct representations are defined for names:

- a printable form, for presentation to a user
- an internal form, for presentation at the API.

The syntax of a printable name is defined by the GSS-API implementation, and may be dependent on local system configuration, or on individual user preference. The internal form provides a canonical representation of the name that is independent of configuration.

A given GSS-API implementation may support names drawn from multiple name spaces. In such an implementation, the internal form of the name must include fields that identify the name space from which the name is drawn. The name space from which a printable name is drawn is specified by an accompanying object identifier.

The functions *gss_import_name*( ) and *gss_display_name*( ) are provided to convert names between their printable representations and the **gss_name_t** type. The function *gss_import_name*( ) may support multiple syntaxes for each supported name space, allowing users the freedom to choose a preferred name representation. The function *gss_display_name*( ) should use an implementation-chosen preferred syntax for each supported name type.

Comparison of internal names is accomplished by means of the *gss_compare_names*( ) function. This removes the need for the application program to understand the syntaxes of the various printable names that a given GSS-API implementation may support.

Storage is allocated by routines that return **gss_name_t** values. The function *gss_release_name*( ) is provided to free storage associated with a name.

7.11 Channel Bindings

GSS-API supports the use of user-specified tags to identify a given context to the peer application. These tags are used to identify the particular communication channel that carries the context. Channel bindings are communicated to the GSS-API using the following structure:

```
typedef struct gss_channel_bindings_struct{
   OM_uint32       initiator_addrtype;
   gss_buffer_desc initiator_address;
   OM_uint32       acceptor_addrtype;
   gss_buffer_desc acceptor_address;
   gss_buffer_desc application_data;
} *gss_channel_bindings_t;
```

The **initiator_addrtype** and **acceptor_addrtype** fields denote the type of addresses contained in the **initiator_address** and **acceptor_address** buffers.

EX Examples[12] of address types[13] are as follows:

- **Defined Address Types**

  GSS_C_AF_INET
  IPV4 only. Four octets in net byte order in *gss_buffer_desc*( ).

  GSS_C_AF_NULLADDR
  Zero length octet string in *gss_buffer_desc*( ).

- **Example Address Types**

  GSS_C_AF_UNSPEC
  Unspecified address type.

  GSS_C_AF_LOCAL
  Host-local address type.

  GSS_C_AF_IMPLINK
  ARPAnet IMP address type.

  GSS_C_AF_PUP
  pup protocols (for example, BSP) address type.

  GSS_C_AF_CHAOS
  MIT CHAOS protocol address type.

  GSS_C_AF_NS
  XEROX NS address type.

  GSS_C_AF_NBS
  nbs address type.

  GSS_C_AF_ECMA
  ECMA address type.

12. The text in RFC 1509 implies that this is prescriptive. At present they are examples (see Section C.3 on page 108 for information on expected future changes).

13. X/Open acknowledges the rights of the proprietors of the trade marks referred to in this section.

GSS_C_AF_DATAKIT
datakit protocols address type.

GSS_C_AF_CCITT
CCITT protocols (for example, X.25).

GSS_C_AF_SNA
IBM SNA address type.

GSS_C_AF_DECnet
DECnet address type.

GSS_C_AF_DLI
DCE address type.

GSS_C_AF_DLI
Direct data link interface address type.

GSS_C_AF_LAT
LAT address type.

GSS_C_AF_HYLINK
NSC Hyperchannel address type.

GSS_C_AF_APPLETALK
AppleTalk address type.

GSS_C_AF_BSC
BISYNC 2780/3780 address type.

GSS_C_AF_DSS
Distributed system services address type.

GSS_C_AF_OSI
OSI TP4 address type.

GSS_C_AF_X25
X.25 address type.

Note that these identify address families rather than specific addressing formats. For address families that contain several possible address forms, the **initiator_address** and **acceptor_address** fields must contain sufficient information to determine which address form is used. When not otherwise specified, addresses should be specified in network byte order.

Conceptually, the GSS-API concatenates the **initiator_addrtype**, **initiator_address**, **acceptor_addrtype**, **acceptor_address** and **application_data** to form an octet string. The mechanism signs this octet string, and binds the signature to the context establishment token emitted by *gss_init_sec_context*( ). The same bindings are presented by the context acceptor to *gss_accept_sec_context*( ), and a signature is calculated in the same way. The calculated signature is compared with that found in the token, and if the signatures differ, *gss_accept_sec_context*( ) returns a [GSS_S_BAD_BINDINGS] error, and the context is not established. Some mechanisms may include the actual channel binding data in the token (rather than just a signature); applications should therefore not use confidential data as channel-binding components. Individual mechanisms may impose additional constraints on addresses and address types that

13. X/Open acknowledges the rights of the proprietors of the trade marks referred to in this section.

may appear in channel bindings. For example, a mechanism may verify that the **initiator_address** field of the channel bindings presented to *gss_init_sec_context*( ) contains the correct network address of the host system.

7.12 Optional Arguments

Various arguments are described as optional. This means that they follow a convention whereby a default value may be requested. The following conventions are used for omitted arguments. These conventions apply only to those arguments that are explicitly documented as optional.

7.12.1 gss_buffer_t Types (Input, Output)

EX Specify GSS_C_NO_BUFFER as a value. For an input argument this signifies that default behaviour is requested, while for an input, output argument it indicates that the information that would be returned by the argument is not required by the application.

7.12.2 Integer Types (Input)

Individual argument documentation lists values to be used to indicate default actions.

7.12.3 Integer Types (Input, Output)

If the caller of the API is not interested in the return value, specify NULL as the value for the pointer.

7.12.4 Pointer Types

Specify NULL as the value.

7.12.5 Object IDs

Specify GSS_C_NULL_OID as the value.

7.12.6 Object ID Sets

Specify GSS_C_NULL_OID_SET as the value.

7.12.7 Credentials

Specify GSS_C_NO_CREDENTIAL to use the default credential handle.

7.12.8 Channel Bindings

Specify GSS_C_NO_CHANNEL_BINDINGS to indicate that channel bindings are not to be used.

Chapter 8

C-language Reference Manual Pages

This chapter specifies the GSS-API C-language functions in alphabetical order.

NAME

gss_accept_sec_context — allow a remotely initiated security context between the application and a remote peer to be established

SYNOPSIS

```
OM_uint32 gss_accept_sec_context(
    OM_uint32              *minor_status,
    gss_ctx_id_t           *context_handle,
    gss_cred_id_t           verifier_cred_handle,
    gss_buffer_t            input_token,
    gss_channel_bindings_t  input_chan_bindings,
    gss_name_t             *src_name
    gss_OID                *mech_type,
    gss_buffer_t            output_token,
    OM_uint32              *ret_flags,
    OM_uint32              *time_rec,
    gss_cred_id_t          *delegated_cred_handle
    );
```

DESCRIPTION

This call verifies the incoming token and returns the authenticated internal name and the mechanism types used. The function may return an *output_token*, which should be transferred to the peer application. The peer application presents it to *gss_init_sec_context*( ). If no token need be sent, *gss_accept_sec_context*( ) indicates this by setting the **length** field of the *output_token* argument to zero. To complete the context establishment, one or more reply tokens may be required from the peer application. If so, *gss_accept_sec_context*( ) returns a status flag of [GSS_S_CONTINUE_NEEDED], in which case it should be called again when the reply token is received from the peer application, passing the token to *gss_accept_sec_context*( ) in the *input_token* argument.

This call may block pending network interactions for those mechanism types in which a directory service or other network entity must be consulted on behalf of a context acceptor, to validate a received ***input_token***.

The ***gss_accept_sec_context*** ( ) routine is used by a context target. Using information in the credentials structure referenced by the input ***acceptor_cred_handle***, it verifies the incoming ***input_token*** and (following the successful completion of a context establishment sequence) returns the authenticated ***src_name*** and the mechanism types used. The ***acceptor_cred_handle*** must correspond to the same valid credentials structure on the initial call to ***gss_accept_sec_context*** ( ) and on any successor calls resulting from GSS_S_CONTINUE_NEEDED status returns; different protocol sequences modelled by the GSS_S_CONTINUE_NEEDED mechanism require access to credentials at different points in the context establishment sequence.

The ***input_context_handle*** argument is 0, meaning ‘‘not yet assigned’’, on the first ***gss_accept_sec_context*** ( ) call relating to a given context. That call returns an ***output_context_handle*** for future references to this context; when continuation attempts to ***gss_accept_sec_context*** ( ) are needed to perform context establishment, that handle value is entered into the ***input_context_handle*** argument.

The ***chan_bindings*** argument is used by the caller to provide information binding the security context to security-related characteristics (for example, addresses, cryptographic keys) of the underlying communication channel. See Section 2.7.6 on page 19 for more discussion of this argument’s usage.

The returned state results (***deleg_state***, ***mutual_state***, ***replay_det_state*** and ***sequence_state***) reflect the same context state values as returned to the caller of ***gss_init_sec_context*** ( ) at the initiator system.

The ***conf_avail*** return value indicates whether the context supports per-message confidentiality services, and so informs the caller whether or not a request for encryption through the ***conf_req_flag*** input to ***gss_wrap*** ( ) can be honoured. In similar fashion, the ***integ_avail*** return value indicates whether per-message integrity services are available (through either ***gss_get_mic*** ( ) or ***gss_wrap*** ( )) on the established context.

The ***lifetime_rec*** return value indicates the length of time for which the context is valid, expressed as an offset from the present. The values of ***deleg_state***, ***mutual_state***, ***replay_det_state***, ***sequence_state***, ***conf_avail***, ***integ_avail*** and ***lifetime_rec*** are undefined unless the accompanying ***major_status*** indicates GSS_S_COMPLETE.

The ***delegated_cred_handle*** result is significant only when ***deleg_state*** is TRUE, and provides a means for the target to reference the delegated credentials. The ***output_token*** result, when non-NULL, provides a context-level token to be returned to the context initiator to continue a multi-step context establishment sequence. As noted with ***gss_init_sec_context*** ( ), any returned token should be transferred to the context's peer (in this case, the context initiator), independent of the value of the accompanying returned major_status.

Note: A target must be able to distinguish a context-level ***input_token,*** which is passed to
EX ***gss_accept_sec_context*** ( ), from the per-message data elements (or tokens) passed to ***gss_verify_mic*** ( ) or ***gss_unwrap*** ( ). These data elements may arrive in a single application message, and ***gss_accept_sec_context*** ( ) must be performed before per-message processing can be performed successfully.

The values returned in the *src_name*, *ret_flags*, *time_rec* and *delegated_cred_handle* arguments are not defined unless the function returns [GSS_S_COMPLETE].

The arguments for *gss_accept_sec_context* ( ) are:

context_handle (in,out)
Context handle for a new context. Supply GSS_C_NO_CONTEXT for the first call; use the value returned in subsequent calls.

verifier_cred_handle (in)
Optional credential handle claimed by context acceptor. Specify GSS_C_NO_CREDENTIAL to use default credentials. If GSS_C_NO_CREDENTIAL is specified, but the caller has no default credentials established, an implementation-defined default credential may be used.

input_token (opaque, in)
Token obtained from remote application.

input_chan_bindings (in)
Application-specified bindings. Allows application to bind channel identification information securely to the security context.

src_name (out)
Optional authenticated name of context initiator. After use, this name should be deallocated by passing it to *gss_release_name*( ). If not required, specify NULL.

mech_type (out)
Security mechanism used. The returned OID value is a pointer into static storage, and should be treated as read-only by the caller.

output_token (opaque, out)
Token to be passed to peer application. If the **length** field of the returned token buffer is 0, no token need be passed to the peer application.

ret_flags (bit-mask, out)
This argument contains six independent flags, each of which indicates that the context supports a specific service option. Symbolic names are provided for each flag, and the symbolic names corresponding to the required flags should be logically ANDed with the *ret_flags* value to test whether a given option is supported by the context. The flags are:

GSS_C_DELEG_FLAG

True Delegated credentials are pointed to by the *delegated_cred_handle* argument.
False No credentials have been delegated.

GSS_C_MUTUAL_FLAG

True Remote peer asked for mutual authentication.
False Remote peer did not ask for mutual authentication.

GSS_C_REPLAY_FLAG

True Replay of signed or sealed messages is detected.
False Replayed messages are not detected.

GSS_C_SEQUENCE_FLAG

True Out-of-sequence signed or sealed messages are detected.
False Out-of-sequence messages are not detected.

GSS_C_CONF_FLAG

True Confidentiality service may be invoked by calling *gss_wrap*( ).
False No confidentiality service (by means of *gss_wrap*( )) available. The function *gss_wrap*( ) provides message encapsulation, data-origin authentication and integrity services only.

GSS_C_INTEG_FLAG

True Integrity service may be invoked by calling either *gss_get_mic*( ) or *gss_wrap*( ).
False Per-message integrity service unavailable.

time_rec (out)
Optional number of seconds for which the context remains valid. Specify NULL if not required.

delegated_cred_handle (out)
Credential handle for credentials received from context initiator. Only valid if GSS_C_DELEG_FLAG in *ret_flags* is true.

minor_status (out)
Mechanism-specific status code.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_CONTINUE_NEEDED]
A token from the peer application is required to complete the context, and *gss_accept_sec_context*( ) must be called again with that token.

[GSS_S_DEFECTIVE_TOKEN]
Consistency checks performed on the *input_token* failed.

[GSS_S_DEFECTIVE_CREDENTIAL]
Consistency checks performed on the credential failed.

[GSS_S_NO_CRED]
The credentials supplied are not valid for context acceptance, or the credential handle does not reference any credentials.

[GSS_S_CREDENTIALS_EXPIRED]
The referenced credentials have expired.

[GSS_S_BAD_BINDINGS]
The *input_token* contains different channel bindings from those specified by the *input_chan_bindings* argument.

[GSS_S_NO_CONTEXT]
The context handle supplied does not refer to a valid context.

[GSS_S_BAD_SIG]
The *input_token* contains an invalid message integrity code.

[GSS_S_OLD_TOKEN]
The *input_token* is too old. This is a fatal error during context establishment.

[GSS_S_DUPLICATE_TOKEN]
The *input_token* is valid, but is a duplicate of a token already processed. This is a fatal error during context establishment.

[GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

ERRORS

No other errors are defined.

NAME

gss_acquire_cred — acquire a handle for an existing credential

SYNOPSIS

```
OM_uint32 gss_acquire_cred(
    OM_uint32      *minor_status,
    gss_name_t      desired_name,
    OM_uint32       time_req,
    gss_OID_set     desired_mechs,
    int             cred_usage
    gss_cred_id_t  *output_cred_handle,
    gss_OID_set    *actual_mechs,
    OM_int32       *time_rec
    );
```

DESCRIPTION

This function allows an application to acquire a handle for an existing credential by name. GSS-API implementations shall impose a local access-control policy on callers of this function to prevent unauthorised callers from acquiring credentials to which they are not entitled. This function is not intended to provide a ''login to the network'' function, as such a function would result in the creation of new credentials rather than merely acquiring a handle to existing credentials. Such functions, if required, should be defined in implementation-specific extensions to the API.

If credential acquisition is time-consuming for a mechanism, the mechanism may choose to delay the actual acquisition until the credential is required (for example, by *gss_init_sec_context*( ) or *gss_accept_sec_context*( )). Such mechanism-specific implementation decisions should be invisible to the calling application; thus a call of *gss_inquire_cred*( ) immediately following the call of *gss_acquire_cred*( ) must return valid credential data, and may therefore incur the overhead of a deferred credential acquisition.

This call is used to acquire a handle to existing or modified credentials so that a principal (as a function of the input argument ***cred_usage***) can initiate or accept security contexts under the identity represented by the input argument ***desired_name***.

On successful completion, the returned ***output_cred_handle*** provides a handle for subsequent references to the acquired credentials. Typically, single-user client processes using only default credentials for context establishment purposes have no need to invoke this call[14].

A caller may provide the value NULL for ***desired_name***, signifying a request for credentials corresponding to a default principal identity. The procedures used by GSS-API implementations to select the appropriate principal identity in response to this form of request are local matters. It is possible that multiple pre-established credentials may exist for the same principal identity (for example, as a result of multiple user login sessions) when ***gss_acquire_cred***( ) is called; the means used in such cases to select a specific credential are local matters. The input argument ***lifetime_req*** to ***gss_acquire_cred***( ) provides information that may be useful in resolving ambiguity in a manner that best satisfies a caller's intent.

The ***lifetime_rec*** output argument indicates the length of time for which the acquired credentials are valid, as an offset from the present. A mechanism may return a reserved value indicating INDEFINITE if no constraints on credential lifetime are imposed. A caller of ***gss_acquire_cred***( )

14. A single-user client is a client acting on behalf of a single identity.

can request a length of time for which acquired credentials are to be valid (the ***lifetime_req*** input argument), beginning at the present, or can request credentials with a default validity interval. (Requests for postdated credentials are not supported within the GSS-API.)

Certain mechanisms and implementations may bind in credential validity period specifiers at a point preliminary to invocation of the ***gss_acquire_cred*** ( ) call (for example, in conjunction with user login procedures). As a result, callers requesting non-default values for ***lifetime_req*** must recognise that such requests cannot always be honoured and must be prepared to accommodate the use of returned credentials with different lifetimes as indicated in ***lifetime_rec***[15].

The caller of ***gss_acquire_cred*** ( ) can explicitly specify a set of mechanism types that are to be accommodated in the returned credentials (***desired_mechs*** argument), or can request credentials for a system-defined default set of mechanism types. Selection of the system-specified default set is recommended in the interests of application portability. The ***actual_mechs*** output argument may be interrogated by the caller to determine the set of mechanisms with which the returned credentials may be used.

The arguments for *gss_acquire_cred*( ) are:

desired_name (in)
Name of principal whose credential is required.

time_req (in)
Number of seconds that credentials remain valid.

desired_mechs (in)
Set of underlying security mechanisms that may be used. GSS_C_NULL_OID_SET may be used to obtain an implementation-specific default.

cred_usage (in)
This argument can have one of the following values:

GSS_C_BOTH
Credentials may be used either to initiate or accept security contexts.

GSS_C_INITIATE
Credentials are only used to initiate security contexts.

GSS_C_ACCEPT
Credentials are only used to accept security contexts.

output_cred_handle (out)
The returned credential handle.

actual_mechs (out)
The optional set of mechanisms for which the credential is valid. Specify NULL if not required.

time_rec (out)
The actual number of seconds for which the returned credentials remain valid; this argument is optional. If the implementation does not support expiration of credentials, the value GSS_C_INDEFINITE is returned. Specify NULL if not required.

---

15. ***lifetime_rec*** may be *greater* than ***lifetime_req***.

minor_status (out)
Mechanism-specific status code.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_BAD_MECH]
Unavailable mechanism requested.

[GSS_S_BAD_NAMETYPE]
Type contained within *desired_name* argument is not supported.

[GSS_S_BAD_NAME]
Value supplied for *desired_name* argument is ill-formed.

[GSS_S_FAILURE]
Unspecified failure. The *minor_status* argument points to more detailed information.

ERRORS

No other errors are defined.

NAME

gss_compare_name — compare two internal-form names

SYNOPSIS

```
OM_uint32 gss_compare_name(
    OM_uint32      *minor_status,
    gss_name_t      name1,
    gss_name_t      name2,
    int            *name_equal
    );
```

DESCRIPTION

Allows an application to compare two internal-form names to determine whether they refer to the same entity.

The arguments for *gss_compare_name*( ) are:

minor_status (out)
Mechanism-specific status code.

name1 (in)
Internal-form name.

name2 (in)
Internal-form name.

name_equal (boolean, out)

True	Names refer to same entity.
False	Names refer to different entities. (Strictly, the names are not known to refer to the same identity.)

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_BAD_NAMETYPE]
The type contained within either *name1* or *name2* is unrecognised, or the names are of incomparable types.

[GSS_S_BAD_NAME]
Either *name1* or *name2* (or both) is ill-formed.

ERRORS

No other errors are defined.

NAME

gss_context_time — determine time for which the context remains valid

SYNOPSIS

```
OM_uint32 gss_context_time(
    OM_uint32      *minor_status,
    gss_ctx_id_t    context_handle,
    OM_uint32      *time_rec
    );
```

DESCRIPTION

This function determines the number of seconds for which the specified context remains valid.

The arguments for *gss_context_time*( ) are:

minor_status (out)
Implementation-specific status code.

context_handle (in)
Identifies the context to be interrogated.

time_rec (out)
Number of seconds that the context remains valid. If the context has already expired, zero is returned.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_CONTEXT_EXPIRED]
The context has already expired.

[GSS_S_CREDENTIALS_EXPIRED]
The context is recognised, but associated credentials have expired.

EX [GSS_S_FAILURE]
Unspecified failure. The *minor_status* argument points to more detailed information.

[GSS_S_NO_CONTEXT]
The *context_handle* argument does not identify a valid context.

ERRORS

No other errors are defined.

NAME

gss_delete_sec_context — delete a security context

SYNOPSIS

```
OM_uint32 gss_delete_sec_context(
    OM_uint32      *minor_status,
    gss_ctx_id_t   *context_handle,
    gss_buffer_t    output_token
    );
```

DESCRIPTION

This call may block pending network interactions for mechanism types in which active notification must be made to a central server when a security context is to be deleted.

This call can be made by either peer in a security context, to flush context-specific information and to return an ***output_context_token*** which can be passed to the context's peer informing it that the peer's corresponding context information can also be flushed. Once a context is established, the peers involved are expected to retain cached credential and context-related information until the information's expiration time is reached or until a ***gss_delete_sec_context*** ( ) call is made. Attempts to perform per-message processing on a deleted context result in error returns. No further security services can be obtained using the context specified by *context_handle*.

The arguments for *gss_delete_sec_context* ( ) are:

minor_status (out)
Mechanism-specific status code.

EX *context_handle* (in, out)
Context handle identifying context to delete.

output_token (opaque, out)
Token to be sent to remote application to instruct it also to delete the context.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

[GSS_S_NO_CONTEXT]
No valid context supplied.

ERRORS

No other errors are defined.

NAME

gss_display_name — obtain a textual representation of an internal name

SYNOPSIS

```
OM_uint32 gss_display_name(
    OM_uint32      *minor_status,
    gss_name_t      input_name,
    gss_buffer_t    output_name_buffer,
    gss_OID        *output_name_type
    );
```

DESCRIPTION

This function allows an application to obtain a textual representation of an opaque internal-form name for display purposes. The syntax of a printable name is defined by the GSS-API implementation.

The arguments for *gss_display_name*( ) are:

minor_status (out)
Mechanism-specific status code.

input_name (in)
Name to be displayed.

output_name_buffer (out)
Buffer to receive textual name string.

output_name_type (out)
The type of the returned name. The returned type **gss_OID** is a pointer into static storage and should be treated as read-only by the caller.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_BAD_MECH]
Unavailable mechanism requested.

[GSS_S_BAD_NAMETYPE]
Type contained within *desired_name* argument is not supported.

[GSS_S_BAD_NAME]
Value supplied for *desired_name* argument is ill-formed.

[GSS_S_FAILURE]
Unspecified failure. The *minor_status* argument points to more detailed information.

ERRORS

No other errors are defined.

NAME

gss_display_status — obtain a textual representation of a GSS-API status code

SYNOPSIS

```
OM_uint32 gss_display_status(
    OM_uint32      *minor_status,
    OM_uint32       status_value,
    int             status_type,
    gss_OID         mech_type,
    int            *message_context,
    gss_buffer_t    status_string
    );
```

DESCRIPTION

This function is used to translate major and minor status codes into printable string representations, for display to the user or for logging purposes. Since some status values may indicate multiple errors, applications may need to call *gss_display_status*( ) multiple times, each call generating a single text string. The *message_context* argument is used to indicate which error message should be extracted from a given *status_value*. The argument *message_context* should be initialised to 0. The function *gss_display_status*( ) returns a non-zero value if there are further messages to extract.

The arguments for *gss_display_status*( ) are:

minor_status (out)
Mechanism-specific status code.

status_value (in)
Status value to be converted.

status_type (in)
This argument can take one of the following values:

GSS_C_GSS_CODE
The argument *status_value* is a GSS status code.

GSS_C_MECH_CODE
The argument *status_value* is a mechanism status code.

mech_type (in)
Optional underlying mechanism (used to interpret a *minor_status* value). Supply GSS_C_NULL_OID to obtain the system default.

message_context (in,out)
Should be initialised to zero by caller on first call. If further messages are contained in the *status_value* argument, *message_context* is non-zero on return; this value should be passed back to subsequent calls, along with the same *status_value*, *status_type* and *mech_type* arguments.

status_string (out)
Textual interpretation of the *status_value*.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_BAD_MECH]
Translation in accordance with an unsupported mechanism type requested.

[GSS_S_BAD_STATUS]
The status value is not recognised, or the status type is neither GSS_C_GSS_CODE nor GSS_C_MECH_CODE.

ERRORS

No other errors are defined.

NAME

gss_get_mic[16] — generate a cryptographic Message Integrity Code (MIC)

SYNOPSIS

```
OM_uint32 gss_get_mic(
    OM_uint32     *minor_status,
    gss_ctx_id_t   context_handle,
    int            qop_req,
    gss_buffer_t   message_buffer,
    gss_buffer_t   msg_token
    );
```

DESCRIPTION

This function generates a message integrity check value for the supplied *message_buffer*, and places the message integrity code in a token for transfer to the peer application. The *qop_req* argument allows a choice between several cryptographic algorithms, if supported by the chosen mechanism.

The arguments for *gss_get_mic*( ) are:

minor_status (out)
Implementation-specific status code.

context_handle (in)
Identifies the context on which the message is sent.

qop_req (in)
Specifies the requested quality of protection. This argument is optional. See Section 7.12 on page 54. Callers are encouraged, on portability grounds, to accept the default quality of protection offered by the chosen mechanism, which may be requested by specifying GSS_C_QOP_DEFAULT for this argument. If an unsupported protection strength is requested, *gss_get_mic*( ) returns [GSS_S_FAILURE].

message_buffer (opaque, in)
Message to be signed.

msg_token (opaque, out)
Buffer to receive token.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_CONTEXT_EXPIRED]
The context has already expired.

[GSS_S_CREDENTIALS_EXPIRED]
The context is recognised, but associated credentials have expired.

[GSS_S_NO_CONTEXT]
The *context_handle* argument does not identify a valid context.

16. The old message name *gss_sign*( ) is also implemented.
See the note on terminology in Appendix C on page 107.

[GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

ERRORS

No other errors are defined.

NAME

gss_import_name — convert a printable name to internal form

SYNOPSIS

```
OM_uint32 gss_import_name(
    OM_uint32     *minor_status,
    gss_buffer_t   input_name_buffer,
    gss_OID        input_name_type,
    gss_name_t    *output_name
    );
```

DESCRIPTION

This function converts a printable name to an internal form.

The arguments for *gss_import_name*( ) are:

minor_status (out)
Mechanism-specific status code.

input_name_buffer (in)
Buffer containing printable name to convert.

input_name_type (in)
Optional object ID specifying type of printable name. Applications may specify either GSS_C_NULL_OID, to use a local system-specific printable syntax, or an OID registered by the GSS-API implementation, to name a particular name space.

output_name (out)
Returned name in internal form.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_BAD_NAMETYPE]
The *input_name_type* is unrecognised.

[GSS_S_BAD_NAME]
The *input_name* argument cannot be interpreted as a name of the specified type.

ERRORS

No other errors are defined.

NAME

gss_indicate_mechs — determine which underlying security mechanisms are available

SYNOPSIS

```
OM_uint32 gss_indicate_mechs(
    OM_uint32     *minor_status,
    gss_OID_set   *mech_set
    );
```

DESCRIPTION

This function allows an application to determine which underlying security mechanisms are available.

The arguments for *gss_indicate_mechs*( ) are:

minor_status (out)
Mechanism-specific status code.

mech_set (out)
Set of implementation-supported mechanisms. The returned type **gss_OID_set** value is a pointer into static storage, and should be treated as read-only by the caller.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

[GSS_S_COMPLETE]
Successful completion.

ERRORS

No errors are defined.

NAME

gss_init_sec_context — initiate the establishment of a security context

SYNOPSIS

```
OM_uint32 gss_init_sec_context(
    OM_uint32             *minor_status,
    gss_cred_id_t          claimant_cred_handle,
    gss_ctx_id_t          *context_handle,
    gss_name_t             target_name,
    gss_OID                mech_type,
    int                    req_flags,
    int                    time_req,
    gss_channel_bindings_t input_chan_bindings,
    gss_buffer_t           input_token,
    gss_OID               *actual_mech_type,
    gss_buffer_t           output_token,
    OM_uint32             *ret_flags,
    OM_uint32             *time_rec
    );
```

DESCRIPTION

This function initiates the establishment of a security context between the application and a remote peer. Initially, the *input_token* argument should be specified as GSS_C_NO_BUFFER. The function may return an *output_token* which should be transferred to the peer application; the peer application presents it to *gss_accept_sec_context*( ). If no token need be sent, *gss_init_sec_context*( ) indicates this by setting the **length** field of the *output_token* argument to zero.

To complete the context establishment, one or more reply tokens may be required from the peer application. If so, *gss_init_sec_context*( ) returns [GSS_S_CONTINUE_NEEDED], in which case it should be called again when the reply token is received from the peer application, passing the token to *gss_init_sec_context*( ) by means of the *input_token* argument.

This call may block pending network interactions for some mechanism types. The mechanism types concerned are those where an authentication server or other network entity must be consulted on behalf of a context initiator in order to generate an ***output_token*** suitable for presentation to a specified target.

This call is used by a context initiator, and ordinarily emits one (or, for the case of a multi-step exchange, more than one) ***output_token*** suitable for use by the target within the selected mechanism type's protocol. Using information in the credentials structure referenced by ***claimant_cred_handle***, ***gss_init_sec_context***( ) initialises the data structures required to establish a security context with target ***targ_name***. The ***claimant_cred_handle*** must correspond to the same valid credentials structure on the initial call to ***gss_init_sec_context***( ) and on any successor calls resulting from GSS_S_CONTINUE_NEEDED status returns; different protocol sequences modelled by the GSS_S_CONTINUE_NEEDED mechanism require access to credentials at different points in the context establishment sequence.

The ***input_context_handle*** argument is 0, meaning ''not yet assigned'', on the first ***gss_init_sec_context***( ) call relating to a given context. That call returns an ***output_context_handle*** for future references to this context. When continuation attempts to ***gss_init_sec_context***( ) are needed to perform context establishment, the previously-returned non-zero handle value is entered into the ***input_context_handle*** argument and is echoed in the returned ***output_context_handle*** argument. On such continuation attempts (and only on continuation attempts) the ***input_token*** value is used to provide the token returned from the

context's target.

The ***chan_bindings*** argument is used by the caller to provide information binding the security context to security-related characteristics (for example, addresses, cryptographic keys) of the underlying communication channel. See Section 2.7.6 on page 19 for more discussion of this argument's usage.

The ***input_token*** argument contains a message received from the target, and is significant only on a call to ***gss_init_sec_context*** ( ) that follows a previous return indicating GSS_S_CONTINUE_NEEDED ***major_status*** value.

It is the caller's responsibility to establish a communication path to the target, and to transmit any returned ***output_token*** (independent of the accompanying returned ***major_status*** value) to the target over that path. The ***output_token*** can, however, be transmitted along with the first application-provided input message to be processed by ***gss_get_mic*** ( ) or ***gss_wrap*** ( ) in conjunction with a successfully-established context. On the final call in a context establishment sequence, ***output_token*** will be returned NULL to indicate that no further token need be transmitted to the peer.

The initiator may request various context-level functions through input flags: the ***deleg_req_flag*** requests delegation of access rights, the ***mutual_req_flag*** requests mutual authentication, the ***replay_det_req_flag*** requests that replay detection features be applied to messages transferred on the established context, and the ***sequence_req_flag*** requests that sequencing be enforced. See Section 2.8.3 on page 23 for more information on replay detection and sequencing features.

Not all of the optionally-requestable features are available in all underlying mechanism types; the corresponding return state values (***deleg_state***, ***mutual_state***, ***replay_det_state***, ***sequence_state***) indicate, as a function of mechanism types processing capabilities and initiator-provided input flags, the set of features active on the context. These state indicators' values are undefined unless the routine's ***major_status*** indicates GSS_COMPLETE. Failure to provide the precise set of features requested by the caller does not cause context establishment to fail; it is the caller's prerogative to delete the context if the feature set provided is unsuitable for the caller's use. The returned mechanism types value indicates the specific mechanism employed on the context, and never indicates the value for default.

The value of ***conf_avail*** indicates whether the context supports per-message confidentiality services, and so informs the caller whether or not a request for encryption through the ***conf_req_flag*** input to ***gss_wrap*** ( ) can be honoured. In similar fashion, the value of ***integ_avail*** indicates whether per-message integrity services are available (through either ***gss_get_mic*** ( ) or ***gss_wrap*** ( )) on the established context.

The ***lifetime_req*** input specifies a desired upper bound for the lifetime of the context to be established, with a value of 0 used to request a default lifetime. The ***lifetime_rec*** output indicates the length of time for which the context is valid, expressed as an offset from the present. Dependent on mechanism capabilities, credential lifetimes and local policy, it may not correspond to the value requested in ***lifetime_req***. If no constraints on context lifetime are imposed, this may be indicated by returning a reserved value representing INDEFINITE ***lifetime_req***. The values of ***conf_avail***, ***integ_avail*** and ***lifetime_rec*** are undefined unless the routine's ***major_status*** indicates GSS_COMPLETE.

If the ***mutual_state*** is TRUE, this fact is reflected within the ***output_token***. A call to ***gss_accept_sec_context*** ( ) at the target in conjunction with such a context returns a token, to be processed by a continuation call to ***gss_init_sec_context*** ( ), to achieve mutual authentication.

The values returned by means of the *ret_flags* and *time_rec* arguments are not defined unless the function returns [GSS_S_COMPLETE].

The arguments for *gss_init_sec_context*( ) are:

claimant_cred_handle (in)
Optional handle for credentials claimed. Supply GSS_C_NO_CREDENTIAL to use default credentials.

context_handle (in,out)
Context handle for new context. Supply GSS_C_NO_CONTEXT for the first call; use the value returned by the first call in continuation calls.

target_name (in)
Name of target.

mech_type (in)
Optional object ID of desired mechanism. Supply GSS_C_NULL_OID to obtain an implementation-specific default.

req_flags (bit-mask, in)
Contains four independent flags, each of which requests that the context support a specific service option. Symbolic names are provided for each flag, and the symbolic names corresponding to the required flags should be logically ORed together to form the bit-mask value. The flags are:

GSS_C_DELEG_FLAG

True Delegate credentials to remote peer.
False Do not delegate.

GSS_C_MUTUAL_FLAG

True Request that remote peer authenticate itself.
False Authenticate self to remote peer only.

GSS_C_REPLAY_FLAG

True Enable replay detection for signed or sealed messages.
False Do not attempt to detect replayed messages.

GSS_C_SEQUENCE_FLAG

True Enable detection of out-of-sequence signed or sealed messages.
False Do not attempt to detect out-of-sequence messages.

time_req (in)
Desired number of seconds for which context should remain valid. Supply 0 to request a default validity period.

input_chan_bindings (in)
Application-specified bindings. Allows application to bind channel identification information securely to the security context.

input_token (opaque, in)
Optional token received from peer application. Supply GSS_C_NO_BUFFER on initial call.

actual_mech_type (out)
Actual mechanism used.

output_token (opaque, out)
Token to be sent to peer application. If the **length** field of the returned buffer is zero, no token need be sent to the peer application.

ret_flags (bit-mask, out)
Contains six independent flags, each of which indicates that the context supports a specific service option. Symbolic names are provided for each flag, and the symbolic names corresponding to the required flags should be logically ANDed with the *ret_flags* value to test whether a given option is supported by the context. The flags are:

GSS_C_DELEG_FLAG

True Credentials have been delegated to the remote peer.
False No credentials have been delegated.

GSS_C_MUTUAL_FLAG

True Remote peer has been asked to authenticate itself.
False Remote peer has not been asked to authenticate itself.

GSS_C_REPLAY_FLAG

True Replay of signed or sealed messages is detected.
False Replayed messages are not detected.

GSS_C_SEQUENCE_FLAG

True Out-of-sequence signed or sealed messages are detected.
False Out-of-sequence messages are not detected.

GSS_C_CONF_FLAG

True Confidentiality service may be invoked by calling *gss_wrap*( ).
False No confidentiality service (by means of *gss_wrap*( )) available. The function *gss_wrap*( ) provides message encapsulation, data-origin authentication and integrity services only.

GSS_C_INTEG_FLAG

True Integrity service may be invoked by calling either *gss_get_mic*( ) or *gss_wrap*( ).
False Per-message integrity service unavailable.

time_rec (out)
Optional number of seconds for which the context remains valid. If the implementation does not support credential expiration, the value GSS_C_INDEFINITE is returned. Specify NULL if not required.

minor_status (out)
Mechanism-specific status code.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_CONTINUE_NEEDED]
A token from the peer application is required to complete the context, and *gss_init_sec_context*( ) must be called again with that token.

[GSS_S_DEFECTIVE_TOKEN]
Consistency checks performed on the *input_token* failed.

[GSS_S_DEFECTIVE_CREDENTIAL]
Consistency checks performed on the credential failed.

[GSS_S_NO_CRED]
The supplied credentials are not valid for context initiation, or the credential handle does not reference any credentials.

[GSS_S_CREDENTIALS_EXPIRED]
The referenced credentials have expired.

[GSS_S_BAD_BINDINGS]
The *input_token* contains different channel bindings from those specified by means of the *input_chan_bindings* argument.

[GSS_S_BAD_SIG]
The *input_token* contains an invalid message integrity code, or a message integrity code that could not be verified.

[GSS_S_OLD_TOKEN]
The *input_token* is too old. This is a fatal error during context establishment.

[GSS_S_DUPLICATE_TOKEN]
The *input_token* is valid, but is a duplicate of a token already processed. This is a fatal error during context establishment.

[GSS_S_NO_CONTEXT]
The supplied context handle does not refer to a valid context.

[GSS_S_BAD_NAMETYPE]
The *target_name* argument provided contains an invalid or unsupported type of name.

[GSS_S_BAD_NAME]
The *target_name* argument provided is ill-formed.

[GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

ERRORS

No other errors are defined.

NAME

gss_inquire_cred — obtain information about a credential

SYNOPSIS

```
OM_uint32 gss_inquire_cred(
    OM_uint32     *minor_status,
    gss_cred_id_t  cred_handle,
    gss_name_t    *name,
    OM_uint32     *lifetime,
    int           *cred_usage
    gss_OID_set   *mechanisms
    );
```

DESCRIPTION

This function obtains information about a credential. The caller must already have obtained a handle that refers to the credential.

The arguments for *gss_inquire_cred*( ) are:

minor_status (out)
Mechanism-specific status code.

cred_handle (in)
A handle that refers to the target credential. Specify GSS_C_NO_CREDENTIAL to inquire about the default credential.

name (out)
The name whose identity the credential asserts. Specify NULL if not required.

lifetime (out)
The number of seconds for which the credential remains valid. If the credential has expired, this argument is set to zero. If the implementation does not support credential expiration, the value GSS_C_INDEFINITE is returned. Specify NULL if not required.

cred_usage (out)
How the credential may be used. This argument takes one of the following values:

GSS_C_INITIATE
GSS_C_ACCEPT
GSS_C_BOTH

Specify NULL if not required.

mechanisms (out)
Set of mechanisms supported by the credential. Specify NULL if not required.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_NO_CRED]
The referenced credentials cannot be accessed.

[GSS_S_DEFECTIVE_CREDENTIAL]
The referenced credentials are invalid.

[GSS_S_CREDENTIALS_EXPIRED]
The referenced credentials have expired. If the *lifetime* argument is not passed as NULL, it is set to 0.

[GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

ERRORS

No other errors are defined.

NAME

gss_process_context_token — pass a token to the security service

SYNOPSIS

```
OM_uint32 gss_process_context_token(
    OM_uint32     *minor_status,
    gss_ctx_id_t   context_handle,
    gss_buffer_t   token_buffer
    );
```

DESCRIPTION

This function provides a way to pass a token to the security service. Usually, tokens are associated either with context establishment (when they would be passed to *gss_init_sec_context*( ) or *gss_accept_sec_context*( )) or with per-message security service (when they would be passed to *gss_verify_mic*( ) or *gss_unwrap*( )). Occasionally, tokens may be received at other times, and *gss_process_context_token*( ) allows such tokens to be passed to the underlying security service for processing. At present, such additional tokens may only be generated by *gss_delete_sec_context*( ). GSS-API implementations may use this service to implement deletion of the security context.

The arguments for *gss_process_context_token*( ) are:

context_handle (in)
Context handle of context on which token is to be processed.

token_buffer (opaque, in)
Pointer to first byte of token to process.

minor_status (out)
Implementation-specific status code.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_DEFECTIVE_TOKEN]
Consistency checks performed on the token fail.

[GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

[GSS_S_NO_CONTEXT]
The *context_handle* does not refer to a valid context.

ERRORS

No other errors are defined.

NAME

gss_release_buffer — free storage associated with a buffer

SYNOPSIS

```
OM_uint32 gss_release_buffer(
    OM_uint32     *minor_status,
    gss_buffer_t   buffer
    );
```

DESCRIPTION

Free storage associated with a buffer format name. The storage must have been allocated by a GSS-API function. In addition to freeing the associated storage, the function zeros the **length** field in the *buffer* argument.

The arguments for *gss_release_buffer*( ) are:

minor_status (out)
Mechanism-specific status code.

EX *buffer* (in, out)
The storage associated with the *buffer* is deleted. The **gss_buffer_desc** object is not freed, but its **length** field is zeroed.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

EX [GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

ERRORS

No errors are defined.

NAME

gss_release_cred — release credential handle

SYNOPSIS

```
OM_uint32 gss_release_cred(
    OM_uint32     *minor_status,
    gss_cred_id_t *cred_handle
    );
```

DESCRIPTION

Informs GSS-API that the specified credential handle is no longer required by the process. When all processes have released a credential, it is deleted.

Note: System-specific credential management functions are also likely to exist, for example, to ensure that credentials shared between processes are properly deleted when all affected processes terminate, even if no explicit release requests are issued by those processes. Given the fact that multiple callers are not precluded from gaining authorised access to the same credentials, invocation of ***gss_release_cred*** ( ) cannot be assumed to delete a particular set of credentials on a system-wide basis.

The arguments for *gss_release_cred*( ) are:

EX *cred_handle* (in,out)
Optional buffer containing opaque credential handle. If GSS_C_NO_CREDENTIAL is supplied, the default credential is released.

minor_status (out)
Mechanism-specific status code.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_NO_CRED]
Credentials cannot be accessed.

EX [GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

ERRORS

No other errors are defined.

NAME

gss_release_name — free storage associated with an internal form name

SYNOPSIS

```
OM_uint32 gss_release_name(
    OM_uint32     *minor_status,
    gss_name_t    *name
    );
```

DESCRIPTION

This function frees storage allocated by GSS_API that is associated with an internal form name.

The arguments for *gss_release_name*( ) are:

minor_status (out)
Mechanism-specific status code.

EX *name* (in,out)
The name to be deleted.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_BAD_NAME]
The *name* argument does not contain a valid name.

EX [GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

ERRORS

No other errors are defined.

NAME

gss_release_oid_set — free storage associated with an OID set

SYNOPSIS

```
OM_uint32 gss_release_oid_set(
    OM_uint32     *minor_status,
    gss_OID_set   *set
    );
```

DESCRIPTION

Free storage associated with a type **gss_OID_set** object. The storage must have been allocated by a GSS-API function.

The arguments for *gss_release_oid_set*( ) are:

minor_status (out)
Mechanism-specific status code.

EX *set* (in,out)
The storage associated with the type **gss_OID_set** is deleted.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

EX [GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

ERRORS

No errors are defined.

NAME

gss_unwrap[17] — convert a previously sealed message back to a usable form

SYNOPSIS

```
OM_uint32 gss_unwrap(
    OM_uint32     *minor_status,
    gss_ctx_id_t   context_handle,
    gss_buffer_t   input_message_buffer,
    gss_buffer_t   output_message_buffer,
    int           *conf_state,
    int           *qop_state
    );
```

DESCRIPTION

This function converts a previously sealed message back to a usable form, verifying the embedded message integrity code. The *conf_state* argument indicates whether the message is encrypted. The *qop_state* argument indicates the strength of protection used to provide the confidentiality and integrity services.

The arguments for *gss_unwrap*( ) are:

minor_status (out)
Mechanism-specific status code.

context_handle (in)
Identifies the context on which the message arrived.

input_message_buffer (opaque, in)
Sealed message.

output_message_buffer (opaque, out)
Buffer to receive unsealed message.

conf_state (boolean, out)

True	Confidentiality and integrity protection used.
False	Integrity service only used.

qop_state (out)
Quality of protection gained from the message integrity code.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_DEFECTIVE_TOKEN]
The token fails consistency checks.

[GSS_S_BAD_SIG]
The message integrity code is incorrect.

---

17. The old message name *gss_unseal*( ) is also implemented.
See the note on terminology in Appendix C on page 107.

[GSS_S_DUPLICATE_TOKEN]
The token is valid, and contains a correct message integrity code for the message, but it has already been processed.

[GSS_S_OLD_TOKEN]
The token is valid, and contains a correct message integrity code for the message, but it is too old.

[GSS_S_UNSEQ_TOKEN]
The token is valid, and contains a correct message integrity code for the message, but it has been verified out of sequence. An earlier token has been signed or sealed by the remote application, but not yet been processed locally.

[GSS_S_CONTEXT_EXPIRED]
The context has already expired.

[GSS_S_CREDENTIALS_EXPIRED]
The context is recognised, but associated credentials have expired.

[GSS_S_NO_CONTEXT]
The context_handle argument does not identify a valid context.

[GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

ERRORS

No other errors are defined.

NAME

gss_verify_mic[18] — verify that a cryptographic Message Integrity Code (MIC) is acceptable

SYNOPSIS

```
OM_uint32 gss_verify_mic(
    OM_uint32     *minor_status,
    gss_ctx_id_t   context_handle,
    gss_buffer_t   message_buffer,
    gss_buffer_t   token_buffer,
    int           *qop_state
    );
```

DESCRIPTION

This function verifies that a cryptographic message integrity code, contained in the *token_buffer* argument, fits the supplied message. The *qop_state* argument allows a message recipient to determine the strength of protection applied to the message.

The arguments for *gss_verify_mic*( ) are:

minor_status (out)
Mechanism-specific status code.

context_handle (in)
Identifies the context on which the message arrived.

message_buffer (opaque, in)
Message to be verified.

token_buffer (opaque, in)
Token associated with message.

qop_state (out)
Quality of protection gained from the message integrity code.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_DEFECTIVE_TOKEN]
The token fails consistency checks.

[GSS_S_BAD_SIG]
The message integrity code is incorrect.

[GSS_S_DUPLICATE_TOKEN]
The token is valid, and contains a correct message integrity code for the message, but it has already been processed.

[GSS_S_OLD_TOKEN]
The token is valid, and contains a correct message integrity code for the message, but it is too old.

---

18. The old message name *gss_verify* ( ) is also implemented.
See the note on terminology in Appendix C on page 107.

[GSS_S_UNSEQ_TOKEN]
The token is valid, and contains a correct message integrity code for the message, but has been verified out of sequence. An earlier token has been signed or sealed by the remote application, but not yet been processed locally.

[GSS_S_CONTEXT_EXPIRED]
The context has already expired.

[GSS_S_CREDENTIALS_EXPIRED]
The context is recognised, but associated credentials have expired.

[GSS_S_NO_CONTEXT]
The context_handle argument does not identify a valid context.

[GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

ERRORS

No other errors are defined.

NAME

gss_wrap[19] — sign and optionally encrypt an input message

SYNOPSIS

```
OM_uint32 gss_wrap(
    OM_uint32     *minor_status,
    gss_ctx_id_t   context_handle,
    int            conf_req_flag,
    int            qop_req,
    gss_buffer_t   input_message_buffer,
    int           *conf_state,
    gss_buffer_t   output_message_buffer
    );
```

DESCRIPTION

This function cryptographically signs and optionally encrypts the specified *input_message*. The *output_message* contains both the message integrity code and the message. The *qop_req* argument allows a choice between several cryptographic algorithms, if supported by the chosen mechanism.

The arguments for *gss_wrap*( ) are:

minor_status (out)
Mechanism-specific status code.

context_handle (in)
Identifies the context on which the message is sent.

conf_req_flag (boolean, in)

True Both confidentiality and integrity services are requested.
False Only integrity service is requested.

qop_req (in)
Specifies the required quality of protection. This argument is optional. See Section 7.12 on page 54. A mechanism-specific default can be requested by setting *qop_req* to GSS_C_QOP_DEFAULT. If an unsupported protection strength is requested, *gss_wrap*( ) returns [GSS_S_FAILURE].

input_message_buffer (opaque, in)
Message to be sealed.

conf_state (boolean, out)

True Confidentiality, data origin authentication and integrity services have been applied.
False Integrity and data origin services only have been applied.

output_message_buffer (opaque, out)
Buffer to receive sealed message.

---

19. The old message name *gss_seal*( ) is also implemented.
See the note on terminology in Appendix C on page 107.

RETURN VALUE

The following GSS status codes shall be returned:

[GSS_S_COMPLETE]
Successful completion.

[GSS_S_CONTEXT_EXPIRED]
The context has already expired.

[GSS_S_CREDENTIALS_EXPIRED]
The context is recognised, but associated credentials have expired.

[GSS_S_NO_CONTEXT]
The *context_handle* argument does not identify a valid context.

[GSS_S_FAILURE]
Failure. The *minor_status* argument points to more detailed information.

ERRORS

No other errors are defined.

Appendix A

Example C Header File <gssapi.h>

This appendix contains the source of an example header file **<gssapi.h>**.

```
#ifndef GSSAPI_H_
#define GSSAPI_H_

/*
 * First, define the platform-dependent types.
 */
typedef <platform-specific> OM_uint32;
typedef <platform-specific> gss_ctx_id_t;
typedef <platform-specific> gss_cred_id_t;
typedef <platform-specific> gss_name_t;

/*
 * Note that a platform supporting the xom.h X/Open header file
 * may make use of that header for the definitions of OM_uint32
 * and the structure to which gss_OID_desc equates.
 */

typedef struct gss_OID_desc_struct {
    OM_uint32 length;
    void      *elements;
} gss_OID_desc, *gss_OID;

typedef struct gss_OID_set_desc_struct  {
    int     count;
    gss_OID elements;
} gss_OID_set_desc, *gss_OID_set;

typedef struct gss_buffer_desc_struct {
    size_t length;
    void *value;
} gss_buffer_desc, *gss_buffer_t;

typedef struct gss_channel_bindings_struct {
    OM_uint32 initiator_addrtype;
    gss_buffer_desc initiator_address;
    OM_uint32 acceptor_addrtype;
    gss_buffer_desc acceptor_address;
    gss_buffer_desc application_data;
} *gss_channel_bindings_t;
```

```
/*
 * Six independent flags each of which indicates that a context
 * supports a specific service option.
 */
#define GSS_C_DELEG_FLAG 1
#define GSS_C_MUTUAL_FLAG 2
#define GSS_C_REPLAY_FLAG 4
#define GSS_C_SEQUENCE_FLAG 8
#define GSS_C_CONF_FLAG 16
#define GSS_C_INTEG_FLAG 32

/*
 * Credential usage options
 */
#define GSS_C_BOTH 0
#define GSS_C_INITIATE 1
#define GSS_C_ACCEPT 2

/*
 * Status code types for gss_display_status
 */
#define GSS_C_GSS_CODE 1
#define GSS_C_MECH_CODE 2

/*
 * The constant definitions for channel-bindings address families
 */
#define GSS_C_AF_UNSPEC     0
#define GSS_C_AF_LOCAL      1
#define GSS_C_AF_INET       2
#define GSS_C_AF_IMPLINK    3
#define GSS_C_AF_PUP        4
#define GSS_C_AF_CHAOS      5
#define GSS_C_AF_NS         6
#define GSS_C_AF_NBS        7
#define GSS_C_AF_ECMA       8
#define GSS_C_AF_DATAKIT    9
#define GSS_C_AF_CCITT      10
#define GSS_C_AF_SNA        11
#define GSS_C_AF_DECnet     12
#define GSS_C_AF_DLI        13
#define GSS_C_AF_LAT        14
#define GSS_C_AF_HYLINK     15
#define GSS_C_AF_APPLETALK  16
#define GSS_C_AF_BSC        17
#define GSS_C_AF_DSS        18
#define GSS_C_AF_OSI        19
#define GSS_C_AF_X25        21

#define GSS_C_AF_NULLADDR   255
```

```
#define GSS_C_NO_BUFFER ((gss_buffer_t) 0)
#define GSS_C_NULL_OID ((gss_OID) 0)
#define GSS_C_NULL_OID_SET ((gss_OID_set) 0)
#define GSS_C_NO_CONTEXT ((gss_ctx_id_t) 0)
#define GSS_C_NO_CREDENTIAL ((gss_cred_id_t) 0)
#define GSS_C_NO_CHANNEL_BINDINGS ((gss_channel_bindings_t) 0)
#define GSS_C_EMPTY_BUFFER {0, NULL}

/*
 * Define the default Quality of Protection for per-message
 * services.  Note that an implementation that offers multiple
 * levels of QOP may either reserve a value (for example zero,
 * as assumed here) to mean "default protection", or alternatively
 * may simply equate GSS_C_QOP_DEFAULT to a specific explicit QOP
 * value.
 */
#define GSS_C_QOP_DEFAULT 0

/*
 * Expiration time of 2^32-1 seconds means infinite lifetime for a
 * credential or security context
 */
#define GSS_C_INDEFINITE 0xfffffffful

/* Major status codes */

#define GSS_S_COMPLETE 0

/*
 * Some "helper" definitions to make the status code macros obvious.
 */
#define GSS_C_CALLING_ERROR_OFFSET 24
#define GSS_C_ROUTINE_ERROR_OFFSET 16
#define GSS_C_SUPPLEMENTARY_OFFSET 0
#define GSS_C_CALLING_ERROR_MASK 0377ul
#define GSS_C_ROUTINE_ERROR_MASK 0377ul
#define GSS_C_SUPPLEMENTARY_MASK 0177777ul

/*
 * The macros that test status codes for error conditions
 */
#define GSS_C_CALLING_ERROR(x) \
  (x & (GSS_C_CALLING_ERROR_MASK << GSS_C_CALLING_ERROR_OFFSET))
#define GSS_C_ROUTINE_ERROR(x) \
  (x & (GSS_C_ROUTINE_ERROR_MASK << GSS_C_ROUTINE_ERROR_OFFSET))
#define GSS_C_SUPPLEMENTARY_INFO(x) \
  (x & (GSS_C_SUPPLEMENTARY_MASK << GSS_C_SUPPLEMENTARY_OFFSET))
#define GSS_C_ERROR(x) \
  ((GSS_C_CALLING_ERROR(x) != 0) || (GSS_C_ROUTINE_ERROR(x) != 0))

/*
 * Now the actual status code definitions
 */
```

```
/*
 * Calling errors:
 */
#define GSS_S_CALL_INACCESSIBLE_READ \
                             (1ul << GSS_C_CALLING_ERROR_OFFSET)
#define GSS_S_CALL_INACCESSIBLE_WRITE \
                             (2ul << GSS_C_CALLING_ERROR_OFFSET)
#define GSS_S_CALL_BAD_STRUCTURE \
                             (3ul << GSS_C_CALLING_ERROR_OFFSET)

/*
 * Routine errors:
 */
#define GSS_S_BAD_MECH (1ul << GSS_C_ROUTINE_ERROR_OFFSET)
#define GSS_S_BAD_NAME (2ul << GSS_C_ROUTINE_ERROR_OFFSET)
#define GSS_S_BAD_NAMETYPE (3ul << GSS_C_ROUTINE_ERROR_OFFSET)
#define GSS_S_BAD_BINDINGS (4ul << GSS_C_ROUTINE_ERROR_OFFSET)
#define GSS_S_BAD_STATUS (5ul << GSS_C_ROUTINE_ERROR_OFFSET)
#define GSS_S_BAD_SIG (6ul << GSS_C_ROUTINE_ERROR_OFFSET)
#define GSS_S_NO_CRED (7ul << GSS_C_ROUTINE_ERROR_OFFSET)
#define GSS_S_NO_CONTEXT (8ul << GSS_C_ROUTINE_ERROR_OFFSET)
#define GSS_S_DEFECTIVE_TOKEN (9ul << GSS_C_ROUTINE_ERROR_OFFSET)
#define GSS_S_DEFECTIVE_CREDENTIAL (10ul << GSS_C_ROUTINE_ERROR_OFFSET)
#define GSS_S_CREDENTIALS_EXPIRED (11ul << GSS_C_ROUTINE_ERROR_OFFSET)
#define GSS_S_CONTEXT_EXPIRED (12ul << GSS_C_ROUTINE_ERROR_OFFSET)
#define GSS_S_FAILURE (13ul << GSS_C_ROUTINE_ERROR_OFFSET)

/*
 * Supplementary info bits:
 */
#define GSS_S_CONTINUE_NEEDED (1ul << (GSS_C_SUPPLEMENTARY_OFFSET + 0))
#define GSS_S_DUPLICATE_TOKEN (1ul << (GSS_C_SUPPLEMENTARY_OFFSET + 1))
#define GSS_S_OLD_TOKEN (1ul << (GSS_C_SUPPLEMENTARY_OFFSET + 2))
#define GSS_S_UNSEQ_TOKEN (1ul << (GSS_C_SUPPLEMENTARY_OFFSET + 3))

/*
 * Finally, function prototypes for the GSS-API routines.
 */

OM_uint32 gss_acquire_cred
         (OM_uint32*,       /* minor_status */
          gss_name_t,       /* desired_name */
          OM_uint32,        /* time_req */
          gss_OID_set,      /* desired_mechs */
          int,              /* cred_usage */
          gss_cred_id_t*,   /* output_cred_handle */
          gss_OID_set*,     /* actual_mechs */
          OM_uint32*        /* time_rec */
         );

OM_uint32 gss_release_cred,
         (OM_uint32*,       /* minor_status */
          gss_cred_id_t*    /* cred_handle */
         );
```

```
OM_uint32 gss_init_sec_context
         (OM_uint32*,       /* minor_status */
          gss_cred_id_t,    /* claimant_cred_handle */
          gss_ctx_id_t*,    /* context_handle */
          gss_name_t,       /* target_name */
          gss_OID,          /* mech_type */
          OM_uint32,        /* req_flags */
          OM_uint32,        /* time_req */
          gss_channel_bindings_t,
                            /* input_chan_bindings */
          gss_buffer_t,     /* input_token */
          gss_OID*,         /* actual_mech_type */
          gss_buffer_t,     /* output_token */
          int*,             /* ret_flags */
          OM_uint32*        /* time_rec */
         );

OM_uint32 gss_accept_sec_context
         (OM_uint32*,       /* minor_status */
          gss_ctx_id_t*,    /* context_handle */
          gss_cred_id_t,    /* verifier_cred_handle */
          gss_buffer_t,     /* input_token_buffer */
          gss_channel_bindings_t,
                            /* input_chan_bindings */
          gss_name_t*,      /* src_name */
          gss_OID*,         /* mech_type */
          gss_buffer_t,     /* output_token */
          OM_uint32*,       /* ret_flags */
          OM_uint32*,       /* time_rec */
          gss_cred_id_t*    /* delegated_cred_handle */
         );

OM_uint32 gss_process_context_token
         (OM_uint32*,       /* minor_status */
          gss_ctx_id_t,     /* context_handle */
          gss_buffer_t      /* token_buffer */
         );

OM_uint32 gss_delete_sec_context
         (OM_uint32*,       /* minor_status */
          gss_ctx_id_t*,     /* context_handle */
          gss_buffer_t      /* output_token */
         );

OM_uint32 gss_context_time
         (OM_uint32*,       /* minor_status */
          gss_ctx_id_t,     /* context_handle */
          OM_uint32*        /* time_rec */
         );
```

```
OM_uint32 gss_get_mic
         (OM_uint32*,       /* minor_status */
          gss_ctx_id_t,     /* context_handle */
          int,              /* qop_req */
          gss_buffer_t,     /* message_buffer */
          gss_buffer_t      /* message_token */
         );

OM_uitn32 gss_verify_mic
         (OM_uint32*,       /* minor_status */
          gss_ctx_id_t,     /* context_handle */
          gss_buffer_t,     /* message_buffer */
          gss_buffer_t,     /* token_buffer */
          int*              /* qop_state */
         );

OM_uint32 gss_wrap
         (OM_uint32*,       /* minor_status */
          gss_ctx_id_t,     /* context_handle */
          int,              /* conf_req_flag */
          int,              /* qop_req */
          gss_buffer_t,     /* input_message_buffer */
          int*,             /* conf_state */
          gss_buffer_t      /* output_message_buffer */
         );

OM_uint32 gss_unwrap
         (OM_uint32*,       /* minor_status */
          gss_ctx_id_t,     /* context_handle */
          gss_buffer_t,     /* input_message_buffer */
          gss_buffer_t,     /* output_message_buffer */
          int*,             /* conf_state */
          int*              /* qop_state */
         );

OM_uint32 gss_display_status
         (OM_uint32*,       /* minor_status */
          OM_uint32,        /* status_value */
          int,              /* status_type */
          gss_OID,          /* mech_type */
          int*,             /* message_context */
          gss_buffer_t      /* status_string */
         );

OM_uint32 gss_indicate_mechs
         (OM_uint32*,       /* minor_status */
          gss_OID_set*      /* mech_set */
         );
```

```
OM_uint32 gss_compare_name
         (OM_uint32*,       /* minor_status */
          gss_name_t,       /* name1 */
          gss_name_t,       /* name2 */
          int*              /* name_equal */
         );

OM_uint32 gss_display_name,
         (OM_uint32*,           /* minor_status */
          gss_name_t,        /* input_name */
          gss_buffer_t,      /* output_name_buffer */
          gss_OID*           /* output_name_type */
         );

OM_uint32 gss_import_name
         (OM_uint32*,       /* minor_status */
          gss_buffer_t,     /* input_name_buffer */
          gss_OID,          /* input_name_type */
          gss_name_t*       /* output_name */
         );

OM_uint32 gss_release_name
         (OM_uint32*,       /* minor_status */
          gss_name_t*       /* input_name */
         );

OM_uint32 gss_release_buffer
         (OM_uint32*,       /* minor_status */
          gss_buffer_t      /* buffer */
         );

OM_uint32 gss_release_oid_set
         (OM_uint32*,       /* minor_status */
          gss_OID_set*      /* set */
         );

OM_uint32 gss_inquire_cred
         (OM_uint32 *,      /* minor_status */
          gss_cred_id_t,    /* cred_handle */
          gss_name_t *,     /* name */
          OM_uint32 *,      /* lifetime */
          int *,            /* cred_usage */
          gss_OID_set *     /* mechanisms */
         );

#endif /* GSSAPI_H_ */
```

X/Open CAE Specification

Part 3

Supplement

X/Open Company Ltd.

Appendix B

Security

This appendix discusses the security issues for implementors of the GSS-API. Refer to the **Procurement Guide** for details of the security functionality provided by X-BASE, X-DAC, X-MAC, X-AUDIT, X-PRIV and X-DIST.

The goals of this security chapter for the GSS-API specification are:

1. To raise the level of awareness of specification writers, specification implementors and application developers who may use the GSS-API such that the security functions being utilised by means of the API are prevented from compromise through misuse or ignorance of key environmental factors.
2. To identify a set of guidelines on the use of GSS-API by specification and application developers to ensure against such compromise.
3. To provide a description of tests that specification writers using GSS-API can apply after having written the specification to ensure no key areas have been overlooked.

B.1 Threats

An implementation of the GSS-API provides peer-entity authentication, delegation, message integrity and privacy services to calling applications. In the course of providing these services certain data elements are used (for example, principal credentials) or created (for example, security contexts) which, if not suitably protected, could result in the breach of expected security. In addition, the failure of applications using the GSS-API to utilise per-message security features can leave communication open to active attack (for example, the insertion of false messages into the communication stream).

Specifically:

- If a principal's credentials are successfully obtained by an unauthorised user, they could be used to masquerade as that principal to another principal for the lifetime of the credentials.
- If the data elements representing the security context between principals is discovered by an unauthorised user, then the security of the communication channel can be breached: privacy protected messages can be read, integrity protected messages can be modified and resent, and false messages can be inserted into the communication stream.
- If the application using the GSS-API fails to establish channel bindings, then that application runs the risk of allowing an attacker to use potentially compromised credentials from any location.
- If the application using the GSS-API fails to use per-message confidentiality services, messages may be subject to unauthorised disclosure.
- If an application using the GSS-API fails to employ per-message replay or out-of-sequence protection, messages may be reordered or replayed without detection.
- If the application using the GSS-API fails to employ per-message integrity services, message recipients may not be able reliably to verify the authenticity of information received. If integrity protection, replay and sequence detection are not used, then the recipient is subject to message interception, modification and retransmission.

- If credentials are destroyed or corrupted, secure associations cannot be established between principals without credential refresh.
- If security context data elements are destroyed or corrupted, subsequent communication using secure facilities is not possible without establishment of a new security context. Messages transmitted under the prior context are not recoverable (if privacy protected) or verifiable (if integrity protected).
- If the application using the GSS-API fails to manage conditions where credentials or contexts expire, then communication subsequent to the expiration is not possible or may be rejected by the peer.

B.1.1 Basic Security Policy Requirements

Integrity

The implementation of the GSS-API shall use appropriate platform security controls to protect credentials and context data elements from modification by external processes, and shall use data-hiding techniques to prevent unintentional modification by the application process.

Availability

The implementation of the GSS-API shall provide for the proper handling of the GSS_S_CREDENTIALS_EXPIRED and GSS_S_CONTEXT_EXPIRED error conditions to ensure that applications can continue processing, or gracefully exit, when credentials and contexts expire.

Confidentiality

The implementation of the GSS-API shall use appropriate platform security controls to protect credentials and context data elements from being referenced, captured or displayed by unauthorised processes.

B.1.2 Impact on Other Specifications

Providing security for an implementation of the GSS-API does not impact other specifications.

B.2 Overview of Security Solution

B.2.1 Security Goals

The goals of the security functionality are twofold:

1. To provide a secure environment for applications using the GSS-API to run such that the critical information necessary for the establishment and maintenance of secure communication cannot be compromised or corrupted.
2. To recommend that certain necessary security services, providing secure communication facilities, should be provided by the actual mechanisms supplied with an implementation of the GSS-API.

B.2.2 Security Framework

To be added in a later version of this specification.

B.2.3 Security Functionality and Services

Availability of the GSS-API specification on a platform results in some or all of the functionality of X-DIST being made available to applications on that platform.

It is suggested that a platform with applications that use an implementation of the GSS-API should provide X-BASE and X-AUDIT security functionality to provide a secure environment for the sensitive data elements used by the GSS-API.

Implementors wishing to provide platforms of extra security should consider implementing some of the following:

X-DAC
X-MAC
X-PRIV
X-DIST.

B.2.4 Standards

None applicable.

B.2.5 Emerging Standards

None applicable.

B.3 Security Specification

B.3.1 Identification

There are no identification functional requirements, except as required by X-BASE to permit the operating environment to protect information in memory or when stored on disk.

B.3.2 Authentication

There are no authentication functional requirements, except as required by X-BASE to permit the operating environment to protect information in memory or when stored on disk.

B.3.3 Authorisation and Access Control

Authorisation and Access Control functionality is implementation-defined.

An implementation of the GSS-API must demonstrate the following:

- Only authorised processes may make use of a principal's credentials to establish a security context with a remote peer. This implies that GSS-API implementations should incorporate appropriate protection features (for example, encryption, effective use of local operating system and file system security features, other available means, or a combination of these approaches) to mediate access to credential data.

  Where appropriate on particular platforms this precludes:

  — the storage of credentials in unprotected files or shared memory

  — the storage of credentials using distributed file systems that do not provide confidentiality services between platforms.

- Only authorised processes may access data associated with a GSS-API security context. This implies that GSS-API implementations should incorporate appropriate protection features (for example, encryption, effective use of local operating system and file system security features, other available means, or combinations of these approaches) to mediate access to context data.

  Where appropriate on particular platforms this precludes the careless use of:

  — unprotected memory

  — unprotected files for temporary storage (including swap space)

  — unprotected communication (for example, shared memory, message queues, raw sockets) for transmission of context information between cooperating processes on the same (or distributed) platform.

Accountability and Audit

This specification provides no recommended functionality for auditing. In environments in which auditing is supported the following activities should generate audit records:

- credential acquisition and calls to ***gss_init_sec_context*** ( ) and ***gss_accept_sec_context*** ( )
- if any of the fatal error codes in Section 2.8.1 on page 21 are encountered
- if the GSS_S_DUPLICATE_TOKEN, GSS_S_OLD_TOKEN or GSS_S_UNSEQ_TOKEN informative status codes are encountered.

Definition of the parameters for the audit records required is left for a future issue of this specification.

Object Reuse

An implementation of the GSS-API shall ensure that data objects are not left in a state such that they could be referenced and examined by unauthorised processes. Platforms employing the object reuse functions of X-BASE should be used where possible to ensure:

- The data structures associated with a security context are erased upon successful completion of the ***gss_delete_sec_context*** ( ) call.
- Internal buffers used by ***gss_get_mic*** ( ), ***gss_wrap*** ( ), ***gss_verify_mic*** ( ), and ***gss_unwrap*** ( ) to store plain text or unsigned data are erased prior to return of control to the calling process.
- The data structures associated with credentials are erased upon successful completion of the ***gss_release_cred*** ( ) call.

Integrity

Necessary integrity functionality is described in Section B.1.1 on page 102.

Additional integrity services are implementation-defined. However, to provide basic communications security, the mechanisms supported by an implementation of the GSS-API should provide per-message integrity, data origin authentication, replay detection, and sequence detection services for applications.

Confidentiality

The necessary confidentiality functionality is described in Section B.1.1 on page 102.

Additional confidentiality services are implementation-defined. However the mechanisms supported by an implementation of the GSS-API should provide confidentiality services.

Availability of Service

The necessary service availability functionality is described in Section B.1.1 on page 102.

Appendix C

Future Directions

This appendix explains how this specification is expected to develop.

C.1 Terminology and Function Names

The names of the functions ***gss_sign***( ) and ***gss_seal***( ) can be misleading because they do not represent signing and sealing in the sense used by ISO. In this document and the referenced RFC 1508:

- The term *sign* means integrity protect.
- The term *signature* means Message Integrity Code (MIC) or cryptographic checkvalue.
- The term *seal* means protect the data with integrity, integrity and confidentiality, or nothing. A suggested alternative is *wrap*. There is no corresponding term in ISO standards.

In this specification, ***gss_get_mic***( ), ***gss_wrap***( ), ***gss_verify_mic***( ) and ***gss_unwrap***( ) are used in preference to ***gss_sign***( ), ***gss_seal***( ), ***gss_verify***( ) and ***gss_unseal***( ).

Note: Because it is expected that the IETF will change to be consistent with ISO terminology, implementors may think it advisable to support both types of name.

C.2 Additional major_status Codes

The descriptions of ***major_status*** values in Section 2.8.3 on page 23 are expected to change as follows:

‘‘When ***replay_det_state*** is TRUE and ***sequence_state*** is FALSE, the possible ***major_status*** values for well-formed and correctly integrity-protected messages are as follows:

- GSS_S_COMPLETE indicates that the message is within the window (of time or sequence space) allowing replay events to be detected, and that the message is not a replay of a previously-processed message within that window.
- GSS_S_DUPLICATE_TOKEN indicates that the cryptographic checkvalue on the received message is correct, but that the message is recognised as a duplicate of a previously-processed message.
- GSS_S_OLD_TOKEN indicates that the cryptographic checkvalue on the received message is correct, but that the message is too old to be checked for duplication.

When ***replay_det_state*** is FALSE and ***sequence_state*** is TRUE, the possible ***major_status*** values for well-formed and correctly signed messages are as follows:

- GSS_S_COMPLETE indicates that the cryptographic checkvalue of the message is correct, is not a replay from the previous message, and that no message is missing relative to the last message received whose integrity checkvalue was correct.
- GSS_S_DUPLICATE_TOKEN indicates that the cryptographic checkvalue on the received message is correct, but that the message is recognised as a duplicate of a previously-processed message.
- GSS_S_OLD_TOKEN indicates that the cryptographic checkvalue on the received message is correct, but that the token is too old to be checked for duplication.

- GSS_S_UNSEQ_TOKEN indicates that the cryptographic checkvalue of the message is correct, but that it is earlier relative to the last message received whose integrity checkvalue was correct.
- GSS_C_GAP_TOKEN indicates that the checkvalue of the message is correct but one or more messages are missing relative to the last message received whose integrity checkvalue was correct (that is, with GSS_S_COMPLETE status).

When both ***sequence_state*** and ***replay_det_state*** are TRUE, the possible ***major_status*** values for well-formed and correctly integrity-protected messages include any of the previous status values with the following meaning for GSS_S_COMPLETE:

- GSS_S_COMPLETE indicates that the cryptographic checkvalue of the message is correct, that the message is within the time window (of time or sequence space) allowing replay messages to be detected, that the message is not a replay of a previously processed message within that window, and that no message is missing relative to the last message received whose integrity checkvalue was correct.''

C.3 Channel Bindings

In a future version of this specification Section 7.11 on page 51 will be amended to add references for the address types listed. The form of the address type and values for the constants will be made normative.

C.4 Status Values

In a future version of this specification Section 7.9 on page 47 may be modified to allow errors to be returned by functions instead of macros.

C.5 Support for Anonymous Security Contexts

There has been some discussion on whether authentication is mandatory.

While the semantics of a NULL pointer are defined (that is, what is passed in), the semantics of an empty value (that is, what is passed out) are not defined. As a consumer of the API is not told otherwise, the assumption is that it expects that a completed security context results in an *src_name* being written into any supplied non-NULL reference parameter. Based on the current text, it is not proven that authentication is not mandatory. If this is intended, it should be noted explicitly.

By adding one flag to *gss_init_sec_context*( ) the client would be able to specify that anonymity is required (in) and would be informed if anonymity is supported (out).

An addition of a flag to *gss_accept_sec_context*( ) (out) could specify that the *src_name* value is not meaningful.

If this feature is considered useful and is added to GSS-API, a future version of this document will describe why and how per-message protection can be used without authentication.

Glossary

For common computing terms refer to the glossary in the **Procurement Guide** .

channel bindings
Information used by GSS-API callers to bind the establishment of a security context to relevant characteristics (for example, addresses and transformed representations of encryption keys).

confidentiality
The property that information is not disclosed without authorisation.

context
A relationship between two communicating peers.

credential
Authentication information possessed by a peer which enables it to initiate security contexts with other peers under a specific mechanism or set of mechanisms.

delegate
A peer who acts on behalf of another peer.

delegation
To authorise a principal to act on behalf of another. With GSS-API, an initiator peer may delegate its credentials to another intermediate peer when establishing a security context. The intermediate which accepted the security context can then optionally act as a delegate for the initiator, when initiating a second security context with a third peer.

IETF
Internet Engineering Taskforce.

integrity
The property that information is protected against undetected, unauthorised modification.

Kerberos
Authentication and key distribution protocol originated in MIT's Project Athena protocol specified in RFC 1511. The Kerberos RFC is RFC-1510.

mechanism
In GSS-API, a security mechanism is a specific method for implementing security functions based on conventions (which may include particular cryptographic algorithms and protocols).

OID
Object Identifier (see the XOM specification).

out of sequence detection
This is the detection of the unauthorised or accidental reordering of messages in transit without either peer being aware of it.

principal
An entity whose identity may be authenticated.

proxy
Same as delegation.

QOP
Quality of protection.

quality of protection
Strength of integrity and confidentiality protection.

replay detection
This is the detection of the unauthorised reuse of a message sent from one peer to another.

signature
Linkage of a data item to a principal, who creates the signature by signing that data item. Signing requires that a cryptographic checksum be computed of the data item, and then be encrypted by the private key of the signing principal.

TGT
Ticket Granting Ticket.

Ticket Granting Ticket
Data element of the Kerberos (see **Kerberos** on page 109) protocol.

token
Binary data, opaque to the GSS-API caller, produced or consumed by a mechanism underlying a GSS-API implementation.

Index