Open Software Foundation
Request For Comments: 80.1
December 1996

J. Wray (Digital)
M. Girkar (Digital)

DCE 1.2 CERTIFICATION API — FUNCTIONAL SPECIFICATION

1. INTRODUCTION

This document describes the DCE 1.2 certification API. This API provides a distributed name to public-key service, supporting the trust models of DCE hierarchical-cells and direct registry-based trust. Additional trust models may be added by vendors or end-users.

1.1. Changes Since Last Publication

Other than the boilerplate, this draft is almost a complete re-write.

2. TARGET

The certification service provides a single certificate infrastructure, supporting applications that wish to use public-key technology for authentication or data-protection purposes. Inclusion within DCE of such a service will encourage a single point of management for such applications, as well as encouraging the use of standard well-understood trust models.

Certificates will be stored within the CDS or X.500 GDS.

3. GOALS AND NON-GOALS

3.1. Goals

Provide a name to public-key translation service, using an application-chosen trust policy.

Provide trust policies that implement retrieval from registry and certificate hierarchy trust models.

Provide sufficient support to enable additional signature algorithms and trust models to be added at a later date.

3.2. Non-goals

No cryptographic services will be provided, other than (subject to licensing) certificate-chain verification.

No certification authority will be provided.

4. TERMINOLOGY

(a) **Asymmetric algorithm**

An encryption (or signature) algorithm utilizing two distinct keys. One key is used to encrypt (or sign) while the other is used to decrypt (or verify). Publication of one key does not reveal the other.

(b) **Key-pair**

A key for an asymmetric encryption algorithm, consisting of two components. One component (the public key) is usually published, while the other (the private key) is kept secret by the key-pair owner.

(c) **Certifying Authority (CA)**

An entity that signs certificates. A trust model determines the set of certificates for which a given CA's signature will be believed.

(d) **Certificate**

A data structure, commonly stored in the directory service that associates a name with a public key. The certificate is signed by a CA. If a principal trusts the certifying authority, and if the signature can be verified, the principal can believe that the public key belongs to the entity named by the certificate.

(e) **Certification path**

A sequence of certificates, used by one principal (the evaluator) to determine the public key of another principal (the target). The sequence begins with a certificate signed by a CA trusted by the evaluator, continues such that each certificate asserts the public key of the CA who signs the next certificate (and is trusted to assert same), and ends with a certificate giving the public key of the target.

(f) **Trust policy**

A policy that defines which certifying authorities are trusted to sign certificates for which principals. The policy may be used to determine whether a given sequence of certificates forms a certification path.

(g) **Extension module**

An implementation of some internal function that is invoked by the DCE certification module. To allow for vendor- or user-extensibility, some functionality within the certification module is invoked via routine-vectors. These routine-vectors may be replaced or added to, either at DCE build-time, or at run-time, allowing vendors or users to provide alternative implementations, or extended functionality.

5. REQUIREMENTS

Support of ISO 9594-8/X.509 certificate format, to at least 1992 standard (X.509 version 2). Possible support of draft X9.xx-1994 extended certificates, and PKCS #6 extended certificates.

Support of the RSA algorithm for certificate signatures, with MD5 hashing.

The high-level API will be callable from a C main program. A requirement for the use of C++ as an implementation language may be placed on certain extension modules.

6. FUNCTIONAL DEFINITION

6.1. Overview

6.1.1. Certificates

A certificate is a data structure that associates a name (the subject of the certificate) with a public key. The entire certificate is signed using the private key of the certificate issuer. It is used whenever one party wishes to determine the public key of another. For example, if party A wishes to send data to B in such a fashion that B can be certain that A was the sender, A can attach a digital signature to the data, constructed using A's private key. To verify such a signature, B must have access to A's public key. A could have already given his public key to B, but in general A and B may never have previously communicated. The use of certificates allows this.

Assume that B wishes to determine A's public key. B will first look for a certificate of which A is the subject. This certificate will be signed by a third party C. If B already knows C's public key, then B can verify the certificate signature. If the signature is valid, and if B is willing to trust C to vouch for A's public key, then B can believe that A's public key is the one contained within the certificate. If B does not know C's public key, then B can attempt to find a certificate in which C is the subject and repeat the verification process.

In general, several certificates may be required in order to be able to trust the public key of a principal. The first certificate will be signed by a key known to the verifier, and will give the public key of an intermediate principal. The second certificate will be signed by that intermediate, and will give the key of a second intermediate, and so on. Such intermediate principals are termed Certification Authorities — principals who have the authority to issue certificates for other principals. The final certificate will contain the key that is actually desired — A's key in the example above. Not all such chains of certificate may be trusted, even if all the certificate signatures are valid. For example, it might be unwise to place a great deal of trust in a certificate that asserted the public key belonging to Bill Clinton, if that certificate were signed by the key belonging to Fidel Castro, or vice versa. A sequence of certificates with valid signatures, where each certificate is trusted to vouch for the next is termed a ''certification path''. In the DCE Certification API, retrieval of certificates and the determination of whether those certificates form a certification path is the responsibility of a ‘‘policy module’’.

6.1.2. Certificate Revocation Lists

When a CA creates a certificate for an end-user or another CA, it places an expiration time within the certificate. This expiration time, like all other fields of the certificate, is protected against modification by the certificate signature, so the certificate cannot be used after it has expired. However, it may happen that the issuing CA decides that the subject of a certificate that it has issued is no longer trustworthy prior to the expiration of the certificate. For example, consider a CA whose responsibility is creating certificates for employees of a company. When an employee leaves the company, the CA will wish to ‘‘recall’’ the certificate(s) that it created that certify the ex-employee. Since the directory service is untrusted, it cannot be relied upon to delete the ex-employee’s certificate; also, copies of the certificate may have been cached outside the directory. Certificate Revocation Lists (CRLs) are used to counter these problems.

Each certificate contains a serial number (the CA is responsible for ensuring that each certificate is issued with a unique serial number), and the CRL is simply a list of the serial numbers of those certificates that should no longer be trusted. The list as a whole is signed by the CA. The CRL also includes a field that says when the next CRL should be issued.

CAs are expected to publish CRLs on a regular basis. The frequency of publication determines the maximum length of time that a certificate may be believed after it has been revoked. Policy modules that support CRLs should verify that they have obtained appropriate CRLs for all certification paths they return to the user. The API allows a policy module to signal the end-user if one or more CRLs for a certification path could not be found.

6.1.3. Policy Modules

Policy modules must be registered with the Certification component before they can be invoked by applications via the API. Two policy modules are automatically registered when the certification API is loaded — the registry policy(0) and the DCE hierarchical policy. DCE vendors may add to the list of pre-registered policies.[1]

The tasks that a policy module must perform include retrieving certificates from some certificate store, verifying their signatures, and verifying that a list of certificates forms a certification path. Support libraries assist with all of these tasks. The xdsxx module is a C++ wrapper around the XDS nameservice API to CDS and GDS, which simplifies the task of retrieving certificates stored in either directory. The x509 module contains C++ classes that can decode X.509 certificates and certificate revocation lists, and verify their signatures. The pkc_certs module contains classes that maintain chains of certificates. Using these support classes, construction of a policy module is a reasonably straightforward task.

6.1.4. X.509 version support

The X.509 standard defines the format of a certificate, using ASN.1 notation. X.509 has passed through two versions, and version 3 is nearing final ballot. Versions 1 and 2 differed from one another only in detail; Version 3 adds significant new features to certificates.

The DCE certification API certificate parser (module x509) currently supports V1 and V2 certificates, and accepts V3 certificates, but ignores all version 3 fields. Partial support for V3 features is implemented in module pkc_certs.

6.1.5. The Directory

X.509 certificates were conceived as a means of authenticating clients to an X.500 directory, and they have since been used as a more general authentication mechanism. The following pre-defined X.500 attributes exist to contain certificates and CRLs:

(1) UserCert — A certificate for an end-user.

(2) CACert — A certificate for an intermediate or CA.

---

1. The registry policy does not use X.509 certificates, but instead retrieves public keys directly from the DCE registry, using authenticated RPC.

(3) CrossCertPair — A pair of certificates, mutually authenticating two CAs.

(4) CRL — A list of certificates that the CA has revoked.

(5) CA-CRL — A subset of the CRL, listing only those revoked certificate that referred to other CAs.

In general, little significance should be placed upon the distinction between the various locations where certificates may be stored. The directory is assumed to be untrusted (since if it were trustworthy, public keys could simply be stored in the directory, without the overhead of certificates and signatures), so the fact that a certificate is found under the CACert attribute of a directory entry should not necessarily be interpreted as meaning that the subject of that certificate is authorized by the signatory to act as a CA — the certificate may have been created as a user certificate and simply ‘‘mis-filed’’. X.509 version 3 defines additional fields within the certificate that enables a certificate signatory to embed the purpose of the certificate within the certificate itself (thereby protected from modification by the certificate signature). Version 1 and 2 certificates do not have this feature. Policy modules are encouraged to treat all certificates equally, no matter where they are found, and to apply policy-specific constraints to determine whether a given certificate should grant its subject the right to act as a CA. This will result in correct behavior for V3 certificates, and secure behavior for V1 and V2 certificates.

6.1.6. Signature algorithms

Certificate signatures are calculated using an algorithm identified by an Object Identifier (a tree-structured value). The signature field within each certificate contains, in addition to the signature itself, the object identifier of the signature algorithm used. The DCE certification component allows for signature verification modules to be registered at run-time, and invoked according to the object identifiers found within certificates. In addition to run-time registration, two signature algorithms are automatically registered when the certification API is first loaded — MD2-with-RSA and MD5-with-RSA. Individual DCE vendors may augment this list of preloaded algorithms.

6.2. Implementation Overview

The certification API presents the process of certificate retrieval and processing as a name-to-key retrieval service for public keys. It is implemented as a strictly layered module. The diagram illustrates this layered design.

```
   |----------------------------------------------|
   |       High-level API (pkc_XXX routines)      |
   |----------------------------------------------|
                          |
   |----------------------------------------------|
   | Policy switch module (pkc_plcy_XXX routines) |
   |----------------------------------------------|
                          |
                  |--------------|
                 |--------------||
                |--------------|||
                |    Policy    |||
                | Extension(s) ||
                |--------------|
                  /          \
                 /            \
|------------------------|   |----------------------|
|     Support classes    |   | Name-service access  |
|    (x509, pkc_certs)   |   |       (xdsxx)        |
|------------------------|   |----------------------|
           |                      /          \
|------------------------|   |-------|  |---------|
|  Crypto switch module  |   |  CDS  |  |  X.500  |
|(pkc_crypto_XXX routines|   |-------|  |---------|
|------------------------|
           |
     |--------------|
    |--------------||
   |--------------|||
   |    Crypto    |||
   | Extension(s) ||
   |--------------|
```

The certification facility is intended to be extensible by vendors and users to support additional signature algorithms, certificate stores and trust models, or enhanced implementations of standard algorithms, stores or models. The common mechanism to achieve this extensibility is run-time dispatch to the appropriate implementation by a ‘‘switch’’ layer. Implementations of extension modules are registered with the approriate switch layer, either at run-time or at system build-time, and are identified by object identifiers (OIDs). Typical applications will not need to register any extension modules, and need use only the high-level API to retrieve principal public-keys.

6.2.1. High-level public-key API

This API provides a name to public-key retrieval service. The application a policy identifiers, a set of trusted public keys, and a target principal name. The service returns the key of the target principal.

6.2.2. Policy switch module

This module provides an API which is invoked by the high-level public-key API module, and an SPI through which individual policy extensions are invoked. A registration API is also provided to allow for run-time registration of policy extensions.

6.2.3. Policy extensions

These modules provide implementations of specific trust policies. A trust policy specifies how trust in a given name-key relationship is established. The reference implementation will contain two policy extensions: The DCE registry policy, which retrieves public-keys directly from the DCE registry using authenticated RPC, and the DCE hierarchical certificate policy, which implements a certificate-based trust model using certificates stored within CDS and/or X.500.

6.2.4. Certificate store API

This API is intended for use by policy modules that store their certificates within a directory-service. It is implemented as a C++ wrapper around the XDS API, and encapsulates much of the complexity of XDS relating to memory management, private vs. public objects, data structure construction and error handling.

6.2.5. Crypto switch module

This module is used by the selected policy module to verify certificate signatures. Crypto modules are implemented as extensions to allow new certificate signature algorithms to be supported transparently to applications.

6.2.6. Crypto extensions

These modules provide signature verification functions. The reference implementation contains extensions for the RSAWithMD5 and RSAWithMD2 signature algorithms.

6.3. Namespace usage

The DCE hierarchical policy, supplied with the reference implementation, retrieves certificates from a directory using the XDS API. Any directory service that can be accessed through XDS may be used as the certificate store. In the reference implementation, this allows the use of CDS or X.500.

To avoid the requirement that DCE principal certificates be stored within the root directory of a DCE cell namespace, whenever the DCE hierarchical policy module attempts to retrieve a certificate from a given directory, if first checks whether the directory contains a ‘‘Certificate SubDirectory’’ attribute (OID 1.3.24.9.15). If present, this attribute is expected to contain a text string which is appended to the directory pathname to give the name of a new directory which should be searched for certificates.

The DCE hierarchical trust policy supplied with the reference implementation assumes that principal names are hierarchical objects, with the levels of the hierarchy seperated by ‘‘/’’ characters in the the printable representation of a principal name. The DCE hierarchical trust policy supports trust paths that are similar to those supported by DCE hierarchical-cells or the DASS/RFC1507 authentication system, in that a valid trust path between principals A and B consists of 0 or more up-links, followed by 0 or 1 cross-link, followed by 0 or more down links.

Each cell is assumed to operate as a certification authority for top-level principals registered within that cell. Thus, the CA for cell A is assumed to create a certificate for principal P within A such that the certificate signatory is ‘‘A’’ and the certificate target is ‘‘A/P’’. If a cell employs structured names for principals, each level is considered to act as a CA for its subordinate. For example, if cell ‘‘A’’ contains a principal ‘‘admin/JohnSmith’’, then A/admin/JohnSmith is certified by two certificates, the first signed by ‘‘A’’ and certifying ‘‘A/admin’’, and the second signed by ‘‘A/admin’’ and certifying ‘‘A/admin/JohnSmith’’.

6.4. The DCE Hierarchical Policy

The DCE Hierarchical Policy (HRP) defines a precise trust policy for choosing certification paths in a X.500 or CDS namespace. Certificates are retrieved using the XDS API, regardless of whether the back-end used to store them is an X.500 or a CDS database. Since the XDS API always uses the X.500 representation for describing namespace entries, the HRP policy, to a large extent, is oblivious of the back-end. The policy accesses CDS entries using their X.500 representations (using the ‘‘typeless’’ attribute for components of the name). Since the HRP does not distinguish between X.500 and CDS, for simplicity we describe the HRP policy in this section assuming that the name service is provided by an X.500 directory.

The HRP policy imposes a hierarchical security model by constraining candidate certification paths to obey specific rules relating to the names of the issuer and subject of each certificate evaluated on a certification path. The evaluation process for selected certificates is independent of the attribute under which the certificate is stored. For User-Certs, CA-Certs and Cross-certs, the evaluation process determines whether the public key for the user/CA identified in these certificates can be trusted based on the previously computed trusted public keys. For CRL and CA-CRL, the evaluation process either weakens, or strengthens the ‘‘faith’’ in previously established trusted keys based on the information presented in these lists. The discussion in this section is limited to the method used in selecting Certification Paths and certificates along that path, and does not describe the evaluation process.

This division of labor is reflected in the implementation of the hierarchical policy, and is typical of a policy module — The policy module selects certificates according to its own policy-specific rules, and then passes selected certificates to the pkc_certs module for evaluation in a policy-independent manner.

6.4.1. Background

X.500 distinguished names define a tree-like hierarchy of locations within the directory. These locations, among other things, may contain one or more types of certificate. In addition to storing certificates, names are also identified with users and certification authorities.

Before going into the details of this policy, we briefly discuss our terminology used for the rest of this section via some simple examples.

The X.500 name /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=john identifies a user or CA; It also identifies a location where you can find certificates pertaining to John's public keys, or possibly certificates signed by John for some other user or CA.

The tree-like hierarchy for names impose a strict rule for defining ancestors and descendants; For example, /C=US/O=DEC/OU=lkg/OU=DCE is an ''ancestor'' of /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=john (who is a ''descendant'' of the former). In this example, /C=US/O=DEC/OU=lkg/OU=DCE is a depth of 2 away from /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=john. Provided two X.500 names share an ancestral relationship (one of them is an ancestor of the other) the depth between them is the number of Relative Distinguished Names (RDN's) that need to be appended at the end of the ancestor name to obtain the descendant name. If the depth is 1, the ancestor is a ''parent'' of the descendant (and the descendant is a ''child'' of the ancestor). Two names share a ''cross-relationship'' if they don't share an ancestral relationship. The closest ancestor for such names is the longest prefix shared between the two names; For example, /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=john and /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=mahesh share a ''cross-link relationship'' with the closest common ancestor being /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security.

6.4.2. Constraints on Certification Paths imposed by HRP

Briefly, the rule used by the HRP policy to select certification paths consists of zero of more up-links, followed by at most one cross-link, followed by zero or more down-links.

Formally, suppose N1, N2, N3, ..., Nx is a candidate certification path, starting with a certifciate signed by principal N1 and ending with a certificate that asserts Nx's public key. This path will be selected by the HRP policy if and only if it belongs to at least one of the two categories (1) or (2) described below:

(1) In this category, the path is divisible into 2 segments N1, N2, ..., Na and Na+1, Na+2, ..., Nx. The first segment is such that Ni+1 is a parent of Ni (1<=i<=a). The second segment is such that Ni+1 is a child of Ni (a < i < x). Further, Na is a parent of Na+1. Note that Na may be the root of the naming hierarchy.

For example, the following certification paths belong to Category 1 and hence will be accepted by the HRP policy.

Path 1:

(N1) /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=john
(N2) /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security
(N3) /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=mahesh

Path 2 (Note the duplicated names):

(N1) /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=john
(N2) /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security
(N3) /C=US/O=DEC/OU=lkg/OU=DCE
(N4) /C=US/O=DEC/OU=lkg
(N5) /C=US/O=DEC
(N6) /C=US
(N7) /C=US/O=DEC
(N8) /C=US/O=DEC/OU=lkg
(N9) /C=US/O=DEC/OU=lkg/OU=DCE
(N10) /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security

(N11) /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=mahesh

(2) In this category, the path is divisible into two segments N1, N2, ..., Na, and Na+1, ..., Nx. The first and second segments are identical in property to their corresponding counterparts in Category 1. However, in this category, we allow for Na and Na+1 to share a cross-link relationship.

Below, we provide one example of a certification path that belongs to Category 2:

Path 3:

(N1) /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=john
(N2) /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=mahesh

(3) The following candidate certification path (Path 4) belongs to neither category 1 nor category 2; hence, it will be rejected by the HRP policy.

Path 4 (rejected by HRP):

(N1) /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=john
(N2) /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=mary
(N3) /C=US/O=DEC/OU=lkg/OU=DCE/OU=Security/CN=mahesh

6.4.3. Certificate Types and Storage Locations

As explained earlier, there are 5 types of certificates:

(a) User Certificates (U-cert)

(b) CA Certificates (CA-Cert)

(c) Cross Certificates (Cross-Cert)

(d) Certificate Revocation List (CRL)

(e) CA Certificate Revocation List (CA-CRL).

Each type of certificate has a corresponding well-known attribute OID. Certificates are stored as values of their corresponding attributes. The attribute for U-cert, CA-cert and Cross-Cert is multi-valued. Thus, there may be more than one user/CA/Cross certificates in a X.500 entry. CRL and CA-CRL atributes should contain only one value (although when these attributes are being updated, there may be a short period when multiple CRLs appear).

We now describe in detail the mechanism used by the HRP policy to avoid requiring large numbers of certificates to be stored in the CDS root directory. HRP policy administrators may choose to store certificates in a different directory from that implied by the DN of the user or CA. This is achieved by defining a ‘‘relative’’ location in the parent directory using the ‘‘Principal Directory Attribute’’. If this attribute of directory entry /C=US/O=dec/lkg has a value ‘‘foo’’, then certificates for /C=US/O=dec/lkg will be stored in /C=US/O=dec/foo/lkg. Similarly, certificates for /C=US/O=dec/lkg/fred will be stored in /C=US/O=dec/foo/lkg/fred. On other words, the RDN ‘‘foo’’ is appended to the name of the parent directory and joined to the remainder of the name to obtain the name of the location where certificates are stored. During certificate retrieval, the HRP policy checks for the existence of the Principal Directory Attribute whenever it attempts to retrieve a certificate. If this attribute is found, it is then used to determine the alternate location.

6.4.4. Choosing Certificates for Evaluation

Once a potential certification path is found by the HRP policy, it uses well-defined rules (1-4, below) for choosing certificates found in the entries for evaluation.

Suppose the path is N1, N2, N3, ..., Nx.

(1) U-Certs. The only U-certs chosen for evaluation are those found in Nx, the target. Further, Nx should be the subject of these U-certs.

(2) A CA-Cert is chosen by evaluation in entry Ni provided:

(a) it is signed by Ni and contains the public key of Ni’s parent; OR

(b) it is signed by a child of Ni and contains the Ni's public key; OR

(c) it is signed by Ni's parent and contains Ni's public key; OR

(d) it is signed by Ni and contains the public key of a child of Ni.

CA-certs of type (a) and type (b) are called up-certificates and those of type (c) and type (d) are called down certificates.

All other CA-certs that do not adhere to rules (2a) and (2b) are considered incorrectly placed and ignored.

(3) A Cross-Cert may contain up to two certificates. The role of the issuer and signatory in one is reversed in the other. A cross-cert can be stored in either the issuer's X.500 entry or the signatory's X.500 entry (or both). A certificate in a cross-cert in Ni's entry is considered for evaluation if all the following conditions hold:

(a) Ni is either the issuer or signatory in atleast one certificate; AND the issuer and signatory share a ''cross-link'' relationship; AND the role of the issuer and signatory in two certificates is reversed.

(b) If Ni is the issuer in the first certificate, the second certificate is considered for evaluation if the signatory in the first certificate appears along the certification path, and, when the X.500 entry of the signatory is processed.

(4) CRL and CA-CRL's are unconditionally evaluated.

6.4.5. Basic Algorithm employed in the HRP policy

The algorithm used to implement the above policy works in two main steps:

Step 1:

The first step consists of establishing a ''trustbase'' given a set of pre-trusted public keys of CA's. In most organizations, there will always be a trust in the Certification Authority of the organization. The first step ''adds'' to this list of pre-trusted keys by traversing up-links starting with the X.500 entries of CA's that are pre-trusted. CA-certs encountered along the up-links (and selected as per the Rules discussed earlier) are used to increase the number of public keys that are trusted. Cross-Certs encountered along this up-traversal are stored away in the ''trustbase'' and evaluated during the second step (see below), if necessary. CRL and CA-CRL's encountered are used to prune,strengthen, or weaken the trust in some of the keys encountered or stashed away and used during the second step.

Step 2:

Once a trustbase is established, it can be used to retrieve public keys of any user. This is performed in the second step, when the namepsace is traversed using down-links from the root to the X.500 entry of the target user. During this step, cross-certs stored for a entry from the first step are evaluated, if the X.500 entry being processed is the signatory in the cross-cert. CA-certs encountered along X.500 entries are evaluated (as per the Rules described earlier). For the target user, both CA and User certs are evaluated. CRL and CA-CRL are evaluated unconditionally.

The final outcome of performing Steps 1 and 2 is an extended trustbase with some of the certification paths completing the trust-chain to the target user. The key asserted by each such path is returned to the caller of the high-level Certification API via the pkc_get_key_count/pkc_get_key_data APIs, and the details of each trust path can be obtained via the pkc_get_key_certifier_coutn/pkc_get_key_certifier_info APIs.

6.5. The Registry Retrieval API

The DCE Registry Policy is fairly simple. Public keys of users are stored in the Security Registry as Extended Registry Attributes (ERA) for a principal. Since the security server is a trusted DCE entity, retrieval of public keys is achieved by making use of secure RPC's to the security server. In this policy, we use the public DCE API available for application developers to perform the ERA lookup.

Step 1 (establishing a trustbase) in the DCE registry Policy is a no-op. Step 2 performs the ERA lookup and retrieves relevant information from the ERA. If the user performing the lookup had certified credentials before performing the lookup, the policy returns the public keys as trusted keys. Any other form of credentials (either uncertified or no credentials) is treated as an untrusted lookup; the keys are returned, but they are ''tagged'' as untrusted.

7. USER INTERFACES

No user interfaces will be provided.

8. DATA STRUCTURES AND APPPLICATION PROGRAMMING INTERFACES

8.1. High-level API

```
/*
 * @OSF_COPYRIGHT@
 * COPYRIGHT NOTICE
 * Copyright (c) 1996 Open Software Foundation, Inc.
 * ALL RIGHTS RESERVED (DCE).  See the file named COPYRIGHT.DCE for
 * the full copyright text.
 * All Rights Reserved
 */
/*
 * HISTORY
 * $Log: pkc_api.h,v $
 * Revision 1.1.2.1  1996/11/13  20:36:44  arvind
 *      OSF DCE 1.2.2 Drop 5
 *      [1996/11/13  19:54:17  arvind]
 *
 * Revision /main/DCE_1.2.2/2  1996/11/04  22:26 UTC  arvind
 *      Cert-API drop 2 from DEC
 *      [1996/11/03  23:23 UTC  arvind  /main/DCE_1.2.2/arvind_capi2/1]
 *
 * Revision /main/DCE_1.2.2/1  1996/10/18  00:39 UTC  arvind
 *      CAPI drop from DEC
 *      [1996/10/16  20:31 UTC  arvind  /main/arvind_capi/1]
 *
 * $EndLog$
 */
/*
 * @DEC_COPYRIGHT@
 * Copyright (c) 1996
 * Digital Equipment Corporation, Maynard, Massachusetts, USA
 * All Rights Reserved
 *
 * FACILITY: Certification API
 *
 * ABSTRACT:  User-API and Data-Structures for CAPI.
 *
 * AUTHOR:
 *            Mahesh B. Girkar & John Wray
 *            Distributed Processing Engineering (DPE)
 *
 *
 *
 */

#ifndef PKC_PLCY_API_
#define PKC_PLCY_API_
```

```
#ifdef __cplusplus
extern "C" {
#endif

#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <dce/dce.h>
#include <dce/utc.h>
#include <dce/gssapi.h>
#include <dce/pkc_consts.h>
#include <dce/sec_pk.h>

/*
 * High-level DCE certificate API.
 *
 *
 * This file exports the high-level DCE Certification API, a
 * general-purpose public-key retrieval facility.  The Certification
 * API supports replacable policy and crypto implementations.
 * Default policies provide a certificate-based hierarchical policy,
 * similar in action to the DCE hierarchical cells mechanism, where
 * certificates may be stored in either CDS or the X.500 GDS, and a
 * non-certificate registry retrieval policy,
 *
 */

typedef struct {
void *handle;        /* will point to a pkc_trust_list_t item */
} trustlist_t;

typedef struct {
  int version;          /* Structure version.  Currently, must be 0 */
  unsigned char *cert;  /* ASN.1 encoding of a Certificate */
  size_t size;          /* size of the encoding */
} cert_t;

typedef struct {
  int version;        /* Structure version.  Currently, must be 0 */
  unsigned char *ca;  /* x500 char * version of Certification Authority
                       * whose key this is
                       * Name; for eg. /..../foo_cell/ca or
                       * /C=US/O=dec/CN=foo_cell/ca
                       */
  unsigned char *key;  /* ca's ASN.1 key */
  size_t size;         /* size of ASN.1 key */
  utc_t startDate;
  utc_t endDate;
} trusted_key_t;

#define IS_KEY          0
#define IS_CERT         1
#define CURRENT_KEY_VERSION     0
#define CURRENT_CERT_VERSION    0
```

```
/* structure to hold either a key, or a certificate */
typedef struct {
  int type;    /* Possible values of type are:
                  IS_KEY, or IS_CERT, anything else currently
                  is an error */
  union {
    trusted_key_t k;
    cert_t        c;
  } u;
} trustitem_t;

/* this structure is defined for future enhancements so that
 * users can specify usages or other constraints on the key that
 * is being retrieved.  In the current version, it is ignored.
 */
typedef struct {
  void *reserved;
} selection_t;

/*
 * structure to hold a trust base (the result of an
 * establish_trustbase call.  Callers do not need to
 * create objects of this type; trustbase_t
 * objects are private to the certification API.
 */

typedef struct {
  gss_OID policy_oid;  /* policy OID used for computing trustbase.
                        * Computing a trust base is a time-consuming
                        * operation, and hence, we cache this in
                        * this structure -- specifically in
                        * trusbase_handle, where policy specific
                        * trustbase pointers are kept.
                        */

  utc_t time;          /* time at which you want public keys
                        * to be valid
                        */

  selection_t *sel;    /* has to be 0; ignored for now; when used, will
                        * point to a copy of the sel ptr passed in by
                        * the user in pkc_init_trustbase
                        */

  void *trustbase_handle; /* this will be the last
                           * pointer returned by ->establish
                           * routine (see pkc_policy_t
                           * structure). It points to a
                           * pkc_trust_list_t list.
                           */
} trustbase_t;

/*
 * keyinfo_t is returned to user when pkc_retrieve_keyinfo
 * is called, and contains information about a set of keys belonging
```

```
 * to a single principal.
 *
 * The user "interrogates" this handle for number of keys,
 * key value(s), certification path (incase of hierarchical policy) etc
 *
 * The structure contains a pointer to the trustbase_t that was used to
 * establish the trust from which these keys were derived, and a
 * policy-specific handle that is used for the actual key information.
 *
 * The user must ensure that the trustbase is not deleted until any
 * dependent keyinfo_t objects have been deleted.
 *
 * Callers do not need to create objects of this type; keyinfo_t
 * objects are private to the certification API.
 */

typedef struct {
  trustbase_t * base;
  char *name;
  utc_t key_date;
  selection_t *sel;
  void *handle;  /* handle points to a policy specific structure */
} keyinfo_t;

typedef enum {
   UNTRUSTED,     /* No trust (e.g. unauthenticated) */
   DIRECT_TRUST,  /* Direct trust via third party
                     (e.g. authenticated registry) */
   CERTIFIED_TRUST    /* Trust certified by caller's trustbase. */
} trust_type_t;

/*
 * A structure that describes the trust that can be placed
 * in a returned key.
 */
typedef struct {
  trust_type_t trust_type;  /* The style of trust */
  char missing_crls;        /* True if one or more CRLs were missing */
  char revoked;             /* True if any certificate has been revoked
                               (even if it were still valid at the
                               retrieval time). */
} certification_flags_t;

/*
 * pkc_get_registered_policies
 * returns a set of OID's, all of which point to pre-loaded policies.
 * An appln will call this once during the life-time of the appln.
 * An appln can then choose to use these oids in
 * pkc_init_trustbase, etc.
 */

extern unsigned32 pkc_get_registered_policies (gss_OID_set * oid_set);

/*
 * Function pkc_init_trustlist
```

```
 *
 * Creates an empty trustlist
 * If tr_list is empty, return pkc_s_asn_bad_param;
 * If we cannot malloc, returns pkc_s_nomem;
 * Otherwise, returns pkc_s_success.
 * On succesful return, tr_list will contain a PKC-allocated
 * data structure which must be released with pkc_free_trustlist()
 */

extern unsigned32 pkc_init_trustlist (trustlist_t **tr_list);

/*
 * Function pkc_append_to_trustlist:
 * Appends to pkc_trust_list_t (described by (*tr_list)->handle)
 */

extern unsigned32 pkc_append_to_trustlist (trustlist_t ** tr_list,
                                            trustitem_t *tr_item,
                                            size_t no_of_tr_items);

/*
 * pkc_init_trustbase: This is to initialize the initial trust
 * list to include all the certificates we initially trust, given
 * the initial set of trusted certificates.
 * This routine will also store the cross-cert pair certificates
 * found during the creationi of the trustbase.
 *
 * Users will normally call the pkc_ API in the following order:
 *
 *  1. pkc_get_registered_policies  (One Time for duration of Appln)
 *
 *  2. pkc_create_empty_trustlist,
 *  3. One or more pkc_append_to_trustlist
 *
 *  2 & 3 together build up an initial trust list.
 *
 *  4. pkc_init_trustbase.
 *
 *  4 computes a trustbase given an initial trust list.
 *
 *  5. One or more pkc_retrieve_keyinfo  for each individual's public
 *     key that needs to be looked up.
 *  6. pkc_free_trustlist
 *  7. pkc_free_trustbase
 *
 *
 * tr_list: trust list presented to API
 * policy_oid: distinguishes which policy to use
 * date: time at which the public key is to be valid. Can be 0.
 * sel: has to be set to 0. (this is present for a future version &
 *      ignored currently)
 * base: extended trust list + other things..
 *
 * On successful return, base will contain a PKC-allocated structure,
 * which should be released with pkc_free_trustbase()
```

```
 */

extern unsigned32 pkc_init_trustbase (trustlist_t **tr_list,
                                      gss_OID policy_oid,
                                      utc_t *time,
                                      selection_t *sel,
                                      trustbase_t **base);

/*
 * pkc_retrieve_keyinfo
 *
 * Returns a keyinfo_t structure describing the set of trusted
 * keys that are valid for the specified principal at the specified
 * date, under any additional constraints specified in key_domain,
 * key_usages or sel.
 *
 * key_domain specifies the domain of the public key(s) to be retrieved.
 * If specified as either NULL or sec_pk_domain_unspecified, keys for
 * all available domains will be returned.
 *
 * key_usages specifies the intended use(s) to which the key will be
 * put.  Key usage values are expressed as a bit-mask made up from the
 * various PKC_KEY_USAGE_xxx and PKC_CAKEY_USAGE_xxx constants defined
 * in file pkc_consts.h.  If key_usages is specified as NULL, keys for
 * all usages are returned.
 *
 * Note that the key_domain and key_usage parameters are considered as
 * advisory only.  Some policies may not distinguish keys according to
 * these properties.  The key_usage and domain parameters of returned
 * keys (output from pkc_get_key_trust_info) should always be checked
 * before attempting to use a key.
 *
 * On successful return, keyinfobase will contain a PKC-allocated
 * structure which must be released with pkc_free_keyinfo().
 */

extern unsigned32 pkc_retrieve_keyinfo (trustbase_t *base,
                                        char *name,
                                        utc_t *key_date,
                                        uuid_t * key_domain,
                                        unsigned long * key_usages,
                                        selection_t *sel,
                                        keyinfo_t **keyinfobase);

/*
 * pkc_get_key_count
 *
 * Return the number of keys within a keyinfo_t structure
 *
 */

extern unsigned32 pkc_get_key_count (keyinfo_t *keyinfobase,
                                     size_t *key_count);

/*
```

```
 * pkc_get_key_data
 *
 * Extracts an encoded public key from a keyinfo_t structure.
 * <Index> is the index of the key (ranging from 0..key_count-1).
 * On return, key_data will contain PKC-allocated storage which must
 * be released with pkc_free().
 */

extern unsigned32 pkc_get_key_data (keyinfo_t *keyinfobase,
                                    unsigned key_index,
                                    unsigned char ** key_data,
                                    size_t * key_length);

/*
 * Miscellaneous key interrogation functions
 *
 */

/*
 * pkc_get_key_trust_info
 *
 * Returns a set of flags describing the trust that can be placed in
 * the key.  See the definition of certification_flags_t for more
 * information.
 *
 * If key_domain and key_usages are passed as non-NULL pointers, on a
 * succesfull return these parameters will describe the domain and
 * permitted usage(s) of the specified key.  Policies that do not
 * distinguish keys according to domain will indicate a domain of
 * sec_pk_domain_unspecified; policies that do not distinguish keys
 * according to usage will indicate all usages are permitted.
 *
 */

extern unsigned32 pkc_get_key_trust_info (keyinfo_t  *keyinfobase,
                                          unsigned key_index,
                                          certification_flags_t * flags,
                                          uuid_t * key_domain,
                                          unsigned long * key_usages);

/*
 * pkc_get_key_certifier_count
 *
 * Returns the number of CAs in the certification path of the
 * specified key.
 *
 */

extern unsigned32 pkc_get_key_certifier_count (keyinfo_t  *keyinfobase,
                                               unsigned key_index,
                                               size_t * ca_count);

/*
 * pkc_get_key_certifier_info
 *
```

```
 * Returns information about a specific certifier from a key's
 * certification path.  Certifier 0 is the CA that vouched for
 * the key; certifier 1 is the CA that vouched for certifier 0, etc.
 * The total number of certifiers for a given key is returned by
 * pkc_get_key_certifier_count above.  Any of the return parameters
 * may be passed as NULL if the corresponding information is not
 * required.  On return, the certifier_name parameter will contain
 * PKC-allocated storage which must be released with pkc_free().
 */

extern unsigned32 pkc_get_key_certifier_info
                              (keyinfo_t *keyinfobase,
                               unsigned key_index,
                               unsigned certifier_index,
                               char ** certifier_name,
                               utc_t * certification_start,
                               utc_t * certification_expiration,
                               char * crl_valid,
                               utc_t * crl_last_seen,
                               utc_t * next_crl_expected);

/*
 * pkc_free_trustlist
 *
 */

extern unsigned32 pkc_free_trustlist(trustlist_t **tr_list);

/*
 * pkc_free_trustbase
 *
 * Note that a trustbase should not be freed until any keyinfo_t
 * structures that were derived from it have been freed.
 */

extern unsigned32 pkc_free_trustbase(trustbase_t **base);

/*
 * pkc_free_keyinfo
 *
 */

extern unsigned32 pkc_free_keyinfo(keyinfo_t **keybase);

/*
 * pkc_free
 *
 * This routine should be used to release contiguous storage returned
 * by any of the pkc_XXX routines.
 *
 */

extern void pkc_free(void *);

#ifdef __cplusplus
```

```
}
#endif

#endif
```

8.2. Policy registration API

```
/*
 * @OSF_COPYRIGHT@
 * COPYRIGHT NOTICE
 * Copyright (c) 1996 Open Software Foundation, Inc.
 * ALL RIGHTS RESERVED (DCE).  See the file named COPYRIGHT.DCE for
 * the full copyright text.
 * All Rights Reserved
 */
/*
 * HISTORY
 * $Log: pkc_plcy_reg.h,v $
 * Revision 1.1.2.1  1996/11/13  20:39:30  arvind
 *      OSF DCE 1.2.2 Drop 5
 *      [1996/11/13  19:56:34  arvind]
 *
 * Revision /main/DCE_1.2.2/2  1996/11/04  22:27 UTC  arvind
 *      Cert-API drop 2 from DEC
 *      [1996/11/03  23:23 UTC  arvind  /main/DCE_1.2.2/arvind_capi2/1]
 *
 * Revision /main/DCE_1.2.2/1  1996/10/18  00:40 UTC  arvind
 *      CAPI drop from DEC
 *      [1996/10/16  20:31 UTC  arvind  /main/arvind_capi/1]
 *
 * $EndLog$
 */
/*
 * @DEC_COPYRIGHT@
 * Copyright (c) 1996
 * Digital Equipment Corporation, Maynard, Massachusetts, USA
 * All Rights Reserved
 *
 * This software is furnished under a license and may be used and copied
 * only  in  accordance  with  the  terms  of such  license and with the
 * inclusion of the above copyright notice.  This software or any  other
 * copies thereof may not be provided or otherwise made available to any
 * other person.  No title to and ownership of the  software  is  hereby
 * transferred.
 *
 * The information in this software  is subject to change without notice
 * and  should  not be  construed  as a commitment by Digitial Equipment
 * Corporation.
 *
 * Digital assumes no responsibility for the use or  reliablity  of its
 * software on equipment which is not supplied by Digital.
 *
 * FACILITY:    Certification API
 *
 * ABSTRACT:    Policy Switch layer definitions.
 *
```

```
 *
 * ENVIRONMENT: portable
 *
 * AUTHOR:
 *              John Wray
 *              Distributed Processing Engineering (DPE)
 *
 * DATE:        March 1996
 *
 */

#ifndef PKC_PLCY_REG_
#define PKC_PLCY_REG_

#ifndef __cplusplus
#error "C++ must be used for policy module registration"
#endif

extern "C" {
#include <stdlib.h>
#include <dce/dce.h>
#include <dce/dcepkcmsg.h>
#include <dce/gssapi.h>
#include <dce/sec_pk_base.h>
#include <dce/pkc_api.h>
};
#include <dce/pkc_common.h>
#include <dce/pkc_certs.h>
#include <dce/x500name.h>

typedef struct {
    char threadsafe;    /* non-zero if retrieve_keyinfo() may be    */
                        /* called simultaneously (within a single   */
                        /* session) by multiple threads.            */
    char multi_session; /* non-zero if the policy implementation    */
                        /* supports multiple simultaneous policy    */
                        /* sessions.                                */
} pkc_plcy_flags_t;

/* This data-type is used to register a new policy-module with the  */
/* certification API.  It describes a policy implementation, and    */
/* provides entry-points to its functions.                          */

typedef struct pkc_policy_struct_t {
    /* The version and alg_id fields are required for all versions  */
    /* of this data-structure.  Other fields may be                 */
    /* version-dependent.                                           */
    OM_uint32 version;      /* must be pkc_V1 for now               */
    gss_OID_desc policy_id;
    /* Object identifier that names the policy implementation       */

    pkc_plcy_flags_t flags;
    /* Describes the policy implementation capabilities             */

    char reserved[32 - sizeof(pkc_plcy_flags_t)];
```

```
/* Reserve space for 32 bytes of flag data.                     */

char * (* name)(void);
/* Returns the policy name.  This will be used in                */
/* diagnostic or auditing messages.  The name should be          */
/* returned in storage allocated using the pkc_malloc()          */
/* function defined in pkc_common.h.  This is the only           */
/* policy-module routine that may be called without having       */
/* first called the <open> routine.                              */

/* NULL may be supplied as the address of any of the following  */
/* routines, if the policy module does not provide or require   */
/* the corresponding feature.  All policy modules should        */
/* provide a <retrieve_keyinfo> function; other functions are   */
/* optional.                                                     */

unsigned32 (*open) (void** context);
unsigned32 (*close) (void** context);
/* Before invoking any policy routines (eg <retrieve_keyinfo>), */
/* caller will establish a session with the policy              */
/* implementation by invoking the implementation's open()       */
/* function.  If the policy module maintains state information  */
/* between calls, it may use the context parameter.             */
/* This same context parameter will be passed to subsequent     */
/* routine calls made by the certification facility.  Note that */
/* once close() has been invoked, the certification facility    */
/* will invoke open() again before making any further calls to  */
/* the policy module.  If the open() routine stores any state   */
/* in the context parameter, the close() routine should free    */
/* this storage.                                                 */

unsigned32 (*establish_trustbase) (void **context,
                            const pkc_trust_list_t &initial_trust,
                            const utc_t * date,
                            char initial_explicit_policy_required,
                            void ** trust_base_handle);
/* This is a one-time call made by appln to initialize a trust */
/* base. It returns the out_trust parameter, which contains an */
/* extended trust list.                                        */
/* After this call is made, the application can call           */
/* retrieve_keyinfo to obtain the public keys of any           */
/* particular principal.  The trustbase will not be changed    */
/* as a rfesult of this call.                                  */

unsigned32 (*retrieve_keyinfo) (
                        void ** context,
                        const void *trust_base_handle,
                        const x500name & subjectName,
                        const utc_t * date,
                        const uuid_t & domain,
                        const pkc_generic_key_usage_t &
                                                   desired_usage,
                        char initial_explicit_policy_req,
                        void ** keys_handle);
/* A routine that retrieves the keys for the specified          */
```

```
    /* principal, stored in a policy-specific handle type.  The     */
    /* caller's initial trust is expressed in the <trust>           */
    /* parameter.  The desired subject name and key usage(s) are    */
    /* specified by the caller, along with an indication as to      */
    /* whether the initial certificate must explicitly contain the  */
    /* active policy in its Policies field.  The returned key       */
    /* information handle can be interrogated by various policy-    */
    /* specific pkc_cert_XXX functions to extract the actual key    */
    /* and determine the degree of trust that can be placed in that */
    /* key.                                                         */
    /* Note that some of these parameters relate to X.509 version 3 */
    /* certificates, support for which is not comitted for DCE 1.2. */
    /* The API has been designed with the intent that it be capable */
    /* of supporting all currently defined versions of X.509, so    */
    /* that it need not change when V3 support is added.            */

     unsigned32 (*delete_trustbase) (void **context,
                                     void **trust_base_handle
                                     );

     unsigned32 (*delete_keyinfo) (void **context,
                                   void **keys_handle
                                  );

/*
 * Interrogation routines
 */

    unsigned32 (*get_key_count) (void ** context,
                                 void * keys_handle,
                                 size_t * key_count);

    unsigned32 (*get_key_data) (void ** context,
                                void * keys_handle,
                                unsigned key_index,
                                unsigned char ** key_data,
                                size_t * key_length);

    unsigned32 (*get_key_trust) (void ** context,
                                 void * keys_handle,
                                 unsigned key_index,
                                 certification_flags_t * flags,
                                 uuid_t * domain,
                                 pkc_generic_key_usage_t * usages);

    unsigned32 (*get_key_certifier_count) (void ** context,
                                           void * keys_handle,
                                           unsigned key_index,
                                           size_t * ca_count);

    unsigned32 (*get_key_certifier_info) (void ** context,
                                          void * keys_handle,
                                          unsigned key_index,
                                          unsigned ca_index,
                                          char ** ca_name,
```

```
                                              utc_t * certification_start,
                                              utc_t *
                                                  certification_expiration,
                                              char  * is_crl_valid,
                                              utc_t * last_crl_seen,
                                              utc_t * next_crl_expected);
/*
 * etc.
 */
} pkc_policy_t;

#define PKC_REPLACE_NONE 0
#define PKC_REPLACE_ENTRYPOINTS 1
#define PKC_REPLACE_ALL 2

extern "C" unsigned32 pkc_plcy_register_policy(pkc_policy_t * plcy,
                                                   int replacement_policy);

    /* Register an implementation of a policy.                       */
    /* The <replacement_policy> argument determines how the PKC     */
    /* facility will handle the case where an implementation of the */
    /* policy is already registered:                                */
    /*                                                              */
    /*     PKC_REPLACE_NONE:    An error will be returned if an     */
    /*                          implementation of the policy is     */
    /*                          already registered.                 */
    /*     PKC_REPLACE_ENTRYPOINTS: Only entrypoints that the       */
    /*                          original implementation did not     */
    /*                          provide will be replaced.           */
    /*     PKC_REPLACE_ALL:     The new implementation will replace */
    /*                          the existing one.                   */
    /* The use of PKC_REPLACE_ENTRYPOINTS is not currently          */
    /* supported.                                                   */

extern "C" unsigned32 pkc_plcy_lookup_policy(gss_OID oid,
                                                 pkc_policy_t * details);
    /* The preferred way of obtaining information about a           */
    /* registered policy implementation, this function returns      */
    /* a policy implementation descriptor block for the             */
    /* specified policy, and leaves the policy list unlocked.       */
    /* The complete list of registered polcicies may be obtained    */
    /* by invoking pkc_get_registered_policies().                   */

extern "C" unsigned32 pkc_plcy_get_registered_policies(
                                          gss_OID_set * oid_set);
    /* Returns an OID-set describing the currently-registered       */
    /* policy implementations.                                      */

extern "C" unsigned32 pkc_plcy_establish_trustbase (
                            gss_OID policy,
                            const pkc_trust_list_t &initial_trust,
                            const utc_t * date,
                            char initial_explicit_policy_required,
                            void ** trust_base_handle);
```

```
extern "C" unsigned32 pkc_plcy_retrieve_keyinfo(
                            gss_OID policy,
                            const void *trust_base_handle,
                            const x500name & subjectName,
                            const utc_t * date,
                            const uuid_t & desired_domain,
                            const pkc_generic_key_usage_t &
                                                     desired_usage,
                            char initial_explicit_policy_required,
                            void ** keys_handle);

extern "C" unsigned32 pkc_plcy_delete_trustbase(
                            gss_OID policy,
                            void **trust_base_handle);

extern "C" unsigned32 pkc_plcy_delete_keyinfo(
                            gss_OID policy,
                            void **keys_handle);

extern "C" unsigned32 pkc_plcy_get_key_count (gss_OID policy,
                                              void * keys_handle,
                                              size_t * key_count);

extern "C" unsigned32 pkc_plcy_get_key_data (gss_OID policy,
                                             void * keys_handle,
                                             unsigned key_index,
                                             unsigned char ** key_data,
                                             size_t * key_length);

extern "C" unsigned32 pkc_plcy_get_key_trust
                                  (gss_OID policy,
                                   void * keys_handle,
                                   unsigned key_index,
                                   certification_flags_t * flags,
                                   uuid_t * domain,
                                   pkc_generic_key_usage_t * usages);

extern "C" unsigned32 pkc_plcy_get_key_certifier_count (
                                   gss_OID policy,
                                   void * keys_handle,
                                   unsigned key_index,
                                   size_t * ca_count);

extern "C" unsigned32 pkc_plcy_get_key_certifier_info (
                                   gss_OID policy,
                                   void * keys_handle,
                                   unsigned key_index,
                                   unsigned ca_index,
                                   char ** certifier_name,
                                   utc_t * certification_start,
                                   utc_t * certification_expiration,
                                   char  * is_crl_valid,
                                   utc_t * last_crl_seen,
                                   utc_t * next_crl_expected);
```

```
    /* The above functions search the list of registered policies   */
    /* for implementations of the specified policy. If found, the   */
    /* implementation will be opened if necessary and its           */
    /* corresponding member function will be invoked.  These        */
    /* routines provide the necessary mutex protection around       */
    /* non-threadsafe policy implementations.                       */

#endif
```

8.3. Crypto registration API

```
/*
 * @OSF_COPYRIGHT@
 * COPYRIGHT NOTICE
 * Copyright (c) 1996 Open Software Foundation, Inc.
 * ALL RIGHTS RESERVED (DCE).  See the file named COPYRIGHT.DCE for
 * the full copyright text.
 * All Rights Reserved
 */
/*
 * HISTORY
 * $Log: pkc_crypto_reg.h,v $
 * Revision 1.1.2.2  1996/11/13  20:38:26  arvind
 *      CAPI drop from DEC
 *      [1996/10/16  20:24 UTC  arvind  /main/DCE_1.2.2/arvind_capi/1]
 *
 * Revision 1.1.2.1  1996/10/03  20:28:27  arvind
 *      OSF DCE 1.2.2 Drop 4
 *      [1996/10/03  19:41:46  arvind]
 *
 * Revision /main/DCE_1.2.2/1  1996/09/09  21:25 UTC  arvind
 *      PKSS drop from DEC (DCE1.2.2)
 *      [1996/08/30  16:10 UTC  arvind  /main/arvind_pkss/1]
 *
 * $EndLog$
 */
/*
 * (c) Copyright 1996,
 * Digital Equipment Corporation, Maynard, Massachusetts, USA
 * All Rights Reserved
 *
 */

#ifndef PKC_CRYPTO_REG_
#define PKC_CRYPTO_REG_

extern "C" {
#include <stdlib.h>
#include <dce/dce.h>
#include <dce/dcepkcmsg.h>
#include <dce/gssapi.h>
#include <dce/sec_pk_base.h>
}

#include <dce/pkc_common.h>
```

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct {
    char threadsafe;    /* non-zero if sign() or verify() may be    */
                        /* may be called simultaneously (within a   */
                        /* single crypto session) by multiple       */
                        /* threads.                                 */
    char multi_session; /* non-zero if the crypto implementation    */
                        /* supports multiple simultaneous crypto    */
                        /* sessions.                                */
} pkc_alg_flags_t;

/* This data-type is used to register a new crypto-module with the  */
/* certification API.  It describes a crypto algorithm, and         */
/* provides entry-points to its sign/verify functions.              */

typedef struct {
    /* The version and alg_id fields are required for all versions  */
    /* of this data-structure.  Other fields may be                 */
    /* version-dependent.                                           */
    OM_uint32 version;      /* must be pkc_V1 for now               */
    gss_OID_desc alg_id;
    /* Object identifier that names the algorithm - the OID that    */
    /* appears in certificates signed by the algorithm.             */

    pkc_alg_flags_t flags;
    /* Describes the algorithm implementation capabilities          */

    char reserved[32 - sizeof(pkc_alg_flags_t)];
    /* Reserve space for 32 bytes of flag data.                     */

    char * (* name)(void);
    /* Returns the algorithm name.  This will be used in            */
    /* diagnostic or auditing messages.  The name should be         */
    /* returned in storage allocated using the pkc_alloc()          */
    /* function defined in pkc_base.h.  This is the only            */
    /* crypto-module routine that may be called without having      */
    /* first called the <open> routine.                             */

    /* NULL may be supplied as the address of any of the following  */
    /* routines, if the crypto module does not provide or require   */
    /* the corresponding feature.  All crypto modules should        */
    /* provide a <verify> function; other functions are optional.   */

    unsigned32 (*open) (void** context);
    unsigned32 (*close) (void** context);
    /* Before invoking any encryption routines (eg <sign> or        */
    /* <verify>), a caller will establish a crypto session with the */
    /* algorithm implementation by invoking the implementation's    */
    /* open() function.  If the crypto module maintains state       */
    /* information between calls, it may use the context parameter. */
    /* This same context parameter will be passed to subsequent     */
    /* sign(), verify() and close() calls made by the certification */
```

```
    /* facility.  Note that once close() has been invoked, the      */
    /* certification facility will invoke open() again before       */
    /* making any further calls to the crypto module.  If the       */
    /* open() routine stores any state in the context parameter,    */
    /* the close() routine should free this storage.                */

    unsigned32 (*verify) (void ** context,
                          sec_pk_gen_data_t data,
                          sec_pk_data_t public_key,
                          sec_pk_data_t signature);
    /* A routine that checks the supplied signature against the     */
    /* supplied data.  <public_key> is a SubjectPublicKeyInfo data  */
    /* structure, encoded in BER, as found within an X.509          */
    /* certificate.  The routine should return 0 for a correct      */
    /* signature, pkc_invalid_signature for an incorrect signature, */
    /* or another DCE-defined error status to indicate any other    */
    /* errors.                                                      */

    unsigned32 (*sign) (void ** context,
                        sec_pk_gen_data_t data,
                        sec_pk_data_t private_key,
                        sec_pk_data_t * signature);
    /* A routine that calculates a signature over the supplied data */
    /* using the specified key.  The <private_key> parameter will   */
    /* be a BER-encoded PrivateKeyInfo data object, as defined in   */
    /* PKCS #7. The <signature> field should be returned by the     */
    /* function; storage allocation should be performed by calling  */
    /* the pkc_alloc() & pkc_free() functions defined above.        */
    /* Crypto modules are not required to provide a sign function.  */

    unsigned32 (*generate_keypair) (void ** context,
                                    unsigned32 size,
                                    void * alg_info,
                                    sec_pk_data_t * private_key,
                                    sec_pk_data_t * public_key);
    /* A routine that generates a keypair.  The <size> parameter    */
    /* should be used by the routine to determine the key-size in   */
    /* some way (for the RSA algorithm, for example, it should be   */
    /* treated as the number of bits in the key modulus).  The      */
    /* <private_key> and <public-key> parameters should return      */
    /* BER-encoded PrivateKeyInfo and SubjectPublicKeyInfo data     */
    /* objects respectively.  The alg_info parameter can be used    */
    /* for algorithm-specific information to modify the             */
    /* key-generation process, but all crypto modules that offer    */
    /* this function should be prepared to operate when NULL is     */
    /* provided for this parameter.                                 */
    /* Crypto modules are not required to provide a                 */
    /* generate_keypair function.                                   */

} pkc_signature_algorithm_t;

#define PKC_REPLACE_NONE 0
#define PKC_REPLACE_ENTRYPOINTS 1
#define PKC_REPLACE_ALL 2
```

```
unsigned32 pkc_crypto_register_signature_alg(
                        pkc_signature_algorithm_t * alg,
                        int replacement_policy);

    /* Register an implementation of a crytptographic algorithm.    */
    /* The <replacement_policy> argumentdetermines how the PKC      */
    /* facility will handle the case where an implementation of the */
    /* algorithm is already registered:                             */
    /*                                                              */
    /*     PKC_REPLACE_NONE:    An error will be returned if an     */
    /*                          implemenation of the algorithm is   */
    /*                          already registered.                 */
    /*     PKC_REPLACE_ENTRYPOINTS: Only entrypoints that the       */
    /*                          original implementation did not     */
    /*                          provide will be replaced.           */
    /*     PKC_REPLACE_ALL:     The new implementation will replace */
    /*                          the existing one.                   */
    /* The use of PKC_REPLACE_ENTRYPOINTS is not currently          */
    /* supported.                                                   */

unsigned32 pkc_crypto_lookup_algorithm(
                            gss_OID oid,
                            pkc_signature_algorithm_t * details);
    /* The preferred way of obtaining information about a           */
    /* registered algorithm implementation, this function returns   */
    /* an algorithm implementation descriptor block for the         */
    /* specified algorithm, and leaves the algorithm list unlocked. */
    /* The complete list of registered algorithms may be obtained   */
    /* by invoking pkc_get_registered_algorithms().                 */

unsigned32 pkc_crypto_get_registered_algorithms(gss_OID_set * oid_set);
    /* Returns an OID-set describing the currently-registered       */
    /* algorithm implementations.                                   */

unsigned32 pkc_crypto_verify_signature(gss_OID algorithm,
                                       sec_pk_gen_data_t data,
                                       sec_pk_data_t public_key,
                                       sec_pk_data_t signature);

unsigned32 pkc_crypto_sign(gss_OID algorithm,
                           sec_pk_gen_data_t data,
                           sec_pk_data_t private_key,
                           sec_pk_data_t * signature);

unsigned32 pkc_crypto_generate_keypair (gss_OID algorithm,
                                        unsigned32 size,
                                        void * alg_info,
                                        sec_pk_data_t * private_key,
                                        sec_pk_data_t * public_key);

    /* The above functions search the list of registered algorithms */
    /* for an implementation of the specified algorithm.  If found, */
    /* the implementation will be opened if necessary and its sign  */
    /* or verify member function will be invoked.                   */
```

```
#ifdef __cplusplus
}
#endif

#endif
```

8.4. XDSXX API

```
/*
 * @OSF_COPYRIGHT@
 * COPYRIGHT NOTICE
 * Copyright (c) 1996 Open Software Foundation, Inc.
 * ALL RIGHTS RESERVED (DCE).  See the file named COPYRIGHT.DCE for
 * the full copyright text.
 * All Rights Reserved
 */
/*
 * HISTORY
 * $Log: xdsxx.h,v $
 * Revision 1.1.2.2  1996/11/18  19:22:38  arvind
 *      OSF DCE 1.2.2 Drop 5
 *      [1996/11/18  19:22:12  arvind]
 *
 * Revision /main/DCE_1.2.2/2  1996/10/18  00:40 UTC  arvind
 *      CAPI drop from DEC
 *      [1996/10/17  00:23 UTC  arvind  /main/DCE_1.2.2/arvind_capi/1]
 *
 * Revision /main/DCE_1.2.2/1  1996/09/09  21:26 UTC  arvind
 *      PKSS drop from DEC (DCE1.2.2)
 *      [1996/08/30  16:12 UTC  arvind  /main/arvind_pkss/1]
 *
 * $EndLog$
 */
/* (c) Copyright 1996,
 * Digital Equipment Corporation, Maynard, Massachusetts, USA
 * All Rights Reserved
 *
 */

#ifndef XDSXX_
#define XDSXX_

#ifndef __cplusplus
#error "This file requires a C++ compiler"
#endif

extern "C" {
#include <pthread.h>
#include <xom.h>
#include <xds.h>
#include <xdsbdcp.h>
#include <xdscds.h>
#include <xdsgds.h>
}
#include <dce/x500name.h>
```

```
extern int OM_object_identifier_equal(const OM_object_identifier & o1,
                                      const OM_object_identifier & o2);

typedef enum {
    XDS_BASE_ONLY,
    XDS_ONE_LEVEL,
    XDS_SUBTREE
} subset_t;

typedef enum {
    XDS_C_SUCCESS, // Indicates a success
    XDS_C_BAD_STATUS, // Indicates a problem decoding a ds_status value
    XDS_C_ABANDON_FAILED,
    XDS_C_ATTRIBUTE_ERROR,
    XDS_C_ATTRIBUTE_PROBLEM,
    XDS_C_COMMUNICATIONS_ERROR,
    XDS_C_LIBRARY_ERROR,
    XDS_C_NAME_ERROR,
    XDS_C_REFERRAL, // Should never be returned to the user
    XDS_C_SECURITY_ERROR,
    XDS_C_SERVICE_ERROR,
    XDS_C_SYSTEM_ERROR,
    XDS_C_UPDATE_ERROR
} xds_error_class_t;

typedef enum {
    VC_UNSUPPORTED,
    VC_NULL,
    VC_BOOLEAN,
    VC_INTEGER,
    VC_CHARSTRING,
    VC_OCTETSTRING,
    VC_BITSTRING
} value_class_t;

value_class_t syntaxToClass(OM_syntax s);

const char * xds_error_class_name(xds_error_class_t class_id);
const char * xds_error_code_name(OM_enumeration code_id);

class om_translation_t {
protected:
    OM_public_object copied_object;
    OM_object original_object;
protected:
    OM_public_object get_obj(void) {
        if (copied_object != NULL) return copied_object;
        else return original_object;
    };
public:
    om_translation_t(void);
    om_translation_t(OM_object & obj, int force = 0);
    virtual ~om_translation_t();
};
```

```
class xds_value_t : public om_translation_t {
public:
    value_class_t theValueType;
    OM_syntax theValueSyntax;
    long theInteger;
    OM_string theString;
    int theBoolean;

    xds_value_t(void);
    xds_value_t(OM_object o);
    xds_value_t(const xds_value_t & o);
    virtual ~xds_value_t();

    virtual unsigned32 set_value(OM_syntax s, long i);
    virtual unsigned32 set_value(OM_syntax s, const char * c);
    virtual unsigned32 set_value(OM_syntax s,
                                 const unsigned char * c, size_t l);
    virtual unsigned32 get_value_syntax(OM_syntax &s) const;
    virtual unsigned32 get_value_class(value_class_t & s) const;
// The next two methods allocate storage via pkc_malloc for their return
// parameters...
    virtual unsigned32 get_value(unsigned char * &s, size_t & l) const;
    virtual unsigned32 get_value(char * &s) const;
    virtual unsigned32 get_value(long & s) const;
};

class xds_value_list_t : public om_translation_t {
    xds_value_list_t * current_value;
public:
    xds_value_t * value;
    xds_value_list_t * next_value;
    xds_value_list_t(void);
    xds_value_list_t(OM_object o);

    virtual unsigned32 add_value(OM_syntax s, long i);
    virtual unsigned32 add_value(OM_syntax s, const char * c);
    virtual unsigned32 add_value(OM_syntax s,
                                 const unsigned char * c, size_t l);

    virtual ~xds_value_list_t();
    virtual unsigned32 copy(xds_value_list_t * &newList) const;
    virtual unsigned32 get_first_value(const xds_value_t * & val) const;
    virtual unsigned32 get_next_value(const xds_value_t * & val) const;
};

class xds_filter_t {
};

class xds_selection_t {
friend class xds_t;
    int all_attributes;
    OM_descriptor * xom_object;
public:
    xds_selection_t(void);
    virtual ~xds_selection_t();
```

```
    void select_all_attributes(void);
    void select_specified_attributes(void);
    void clear_all_attributes(void);
    void select_attribute(OM_object_identifier attr);
    void return_types_only(void);
    void return_types_and_values(void);
};

class xds_result_t : public om_translation_t {
    int AliasWasDereferenced;
public:
    unsigned32 error_status;
    xds_result_t(OM_object par);
    void scan_object(OM_object obj);
//    virtual ~xds_result_t();
    virtual int aliasDereferenced(void) const {
        return AliasWasDereferenced;
    }
};

class xds_attribute_t : public om_translation_t {
friend class xds_attribute_list_t;
friend class xds_changes_t;
    xds_attribute_t * next_attribute;
    OM_object_identifier attributeType;
    xds_value_list_t * first_attributeValue;
public:
    unsigned32 error_status;
    xds_attribute_t(OM_object apr);
    xds_attribute_t(const OM_object_identifier & attr_type,
                    const xds_value_list_t & values);
    virtual ~xds_attribute_t();
    virtual unsigned32 get_next_attribute(
                           const xds_attribute_t * & attr) const;
    virtual unsigned32 get_type(
                        const OM_object_identifier * & attr_type) const;
    virtual unsigned32 get_first_value(const xds_value_t * &s ) const;
    virtual unsigned32 get_next_value(const xds_value_t * &s ) const;
};

class xds_changes_t {
    int modification_count;
    OM_object xom_object;
    virtual unsigned32 add_attr_value_list(
                            const OM_object_identifier & attr_type,
                            const xds_value_list_t * values,
                            OM_enumeration mod_type);

public:
    xds_changes_t(void);
    virtual ~xds_changes_t();
    virtual unsigned32 build_xom_object(OM_object & o) const;
    virtual unsigned32 delete_xom_object(void) const;
    virtual unsigned32 add_attribute(
                           const OM_object_identifier & attr_type,
```

```
                              const xds_value_list_t & values);
    virtual unsigned32 add_values(
                              const OM_object_identifier & attr_type,
                              const xds_value_list_t & values);
    virtual unsigned32 remove_attribute(
                              const OM_object_identifier & attr_type);
    virtual unsigned32 remove_values(
                              const OM_object_identifier & attr_type,
                              const xds_value_list_t & values);
};

class xds_attribute_list_t : public om_translation_t {
    xds_attribute_t * first_attribute;
    OM_object xom_object;
public:
    unsigned32 error_status;
    xds_attribute_list_t(OM_object par);
    xds_attribute_list_t(void);
    virtual ~xds_attribute_list_t();
    virtual unsigned32 build_xom_object(OM_object & o) const;
    virtual unsigned32 delete_xom_object(void) const;
    virtual unsigned32 get_first_attribute(
                             const xds_attribute_t * & att) const;
    virtual unsigned32 add_attribute(
                             const OM_object_identifier &attr_type,
                             const xds_value_list_t & values);
};

class xds_entry_t : public xds_attribute_list_t {
    x500name object_name;
    int wasFromEntry;
public:
    xds_entry_t(OM_object par);
    virtual ~xds_entry_t();
    virtual unsigned32 get_object_name(
                            const x500name * & name) const;
    virtual int fromEntry(void) const { return wasFromEntry; };
};

class xds_read_result_t : public xds_result_t {
    xds_entry_t * entry;
    const xds_attribute_t * cur_attribute;
public:
    xds_read_result_t(OM_object par);
    virtual ~xds_read_result_t();
    virtual unsigned32 get_object_name(
                             const x500name *& name) const;
    virtual unsigned32 get_first_attribute(
                             const xds_attribute_t * & att) const;
    virtual unsigned32 get_next_attribute(
                             const xds_attribute_t * & att) const;
};

class xds_list_info_item_t {
friend class xds_list_result_t;
```

```
    xds_list_info_item_t * next_item;
    int entryIsAlias;
    int wasFromEntry;
    RelativeDistinguishedName rdn;
public:
    int isAlias(void) {return entryIsAlias; };
    int fromEntry(void) {return wasFromEntry; };
    unsigned32 error_status;
    xds_list_info_item_t(OM_object obj);
    virtual ~xds_list_info_item_t();
    virtual unsigned32 get_rdn(
                        const RelativeDistinguishedName * & name) const;
};

class xds_list_result_t : public xds_result_t {
protected:
    x500name object_name;
    int wasPartialOutcome;
    xds_list_info_item_t * first_item;
    xds_list_info_item_t * cur_item;
public:
    xds_list_result_t(OM_object par);
    virtual ~xds_list_result_t();
    virtual unsigned32 get_object_name(const x500name * & name)
    const {
        if (object_name.is_set()) name = &object_name;
        else return pkc_s_no_value;
        return 0;
    };
    virtual int partialOutcome(void) const {
        return wasPartialOutcome;
    };
    virtual unsigned32 get_first_item(
                        const xds_list_info_item_t * & result) const;
    virtual unsigned32 get_next_item(
                        const xds_list_info_item_t * & result) const;
};

class xds_search_result_t : public xds_result_t {
public:
   xds_search_result_t(OM_object o) : xds_result_t(o) {};
   virtual ~xds_search_result_t();
};

class xds_status_t : public om_translation_t {
protected:
    OM_object_identifier attribute_oid;
    x500name problem_name;
public:
    xds_status_t(OM_object ds_status);
    xds_status_t(void);
    virtual ~xds_status_t();
    virtual xds_status_t & operator=(const xds_status_t & o);
    virtual int isSuccess(void) const;
    virtual int isError(void) const { return !isSuccess(); };
```

```
    xds_error_class_t error_class;
    OM_enumeration error_code;
    virtual unsigned32 get_attribute_oid(
                            const OM_object_identifier * & oid) const;
    virtual unsigned32 get_name(const x500name * & name) const;
};

class xds_t {
private:
    static pthread_once_t once_block;
    static OM_workspace workspace;
    static void initialize_xds(void);
    OM_private_object session;
    static int cdsIsSupported;
    static int gdsIsSupported;
public:
    int cds_supported(void) { return cdsIsSupported; };
    int gds_supported(void) { return gdsIsSupported; };
    unsigned32 error_status;
    xds_t(void);
    virtual ~xds_t();

    virtual xds_status_t * add_entry(
                               const x500name & name,
                               const xds_attribute_list_t & entry);
    virtual xds_status_t * add_entry(const x500name & name);
    virtual xds_status_t * list(const x500name & name,
                                 xds_list_result_t *& results);
    virtual xds_status_t * modify(
                               const x500name & name,
                               const xds_changes_t & changes);
    virtual xds_status_t * modify_rdn(
                             const x500name & name,
                             const RelativeDistinguishedName & new_rdn,
                             int delete_old_rdn);
    virtual xds_status_t * read(const x500name & name,
                                 const xds_selection_t & selection,
                                 xds_read_result_t *& results);
    virtual xds_status_t * remove_entry(const x500name & name);
    virtual xds_status_t * search(const x500name & base,
                                   subset_t subset,
                                   const xds_filter_t & filter,
                                   int search_aliases,
                                   const xds_selection_t & selection,
                                   xds_search_result_t *& results);
};

#endif
```

8.5. Certificate manipulation classes

8.5.1. asn.h

```
/*
 * @OSF_COPYRIGHT@
 * COPYRIGHT NOTICE
```

```
 * Copyright (c) 1996 Open Software Foundation, Inc.
 * ALL RIGHTS RESERVED (DCE).  See the file named COPYRIGHT.DCE for
 * the full copyright text.
 * All Rights Reserved
 */
/*
 * HISTORY
 * $Log: asn.h,v $
 * Revision 1.1.2.2  1996/11/13  20:36:15  arvind
 *      Cert-API drop 2 from DEC
 *      [1996/11/03  23:23 UTC  arvind  /main/DCE_1.2.2/arvind_capi2/1]
 *
 *      CAPI drop from DEC
 *      [1996/10/16  20:24 UTC  arvind  /main/DCE_1.2.2/arvind_capi/1]
 *
 * Revision 1.1.2.1  1996/10/04  19:28:24  arvind
 *      OSF DCE 1.2.2 Drop 4
 *      [1996/10/04  19:28:10  arvind]
 *
 * Revision /main/DCE_1.2.2/1  1996/09/09  21:25 UTC  arvind
 *      PKSS drop from DEC (DCE1.2.2)
 *      [1996/08/30  16:08 UTC  arvind  /main/arvind_pkss/1]
 *
 * $EndLog$
 */

/* (c) Copyright 1996,
 * Digital Equipment Corporation, Maynard, Massachusetts, USA
 * All Rights Reserved
 *
 * Module: asn.h
 *
 * Purpose:  This module contains the class definitions for the ASN.1
 *           BER encode/decode objects
 *
 *
 */

#ifndef ASN_
#define ASN_

#ifdef __cplusplus
extern "C" {
#endif

#include <pthread.h>
/*
 * pthread needed as it might re-define read & write, so including it
 * here means that our users silently see the redefined versions.
 */
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <dce/nbase.h>
#include <dce/utc.h>
```

```
#include <dce/dcepkcmsg.h>
#include <dce/gssapi.h>

#ifdef __cplusplus
}
#endif

#define pkc_malloc_status

#ifdef __cplusplus
extern "C"
unsigned32 pkc_adjust(unsigned char * &b,
                      size_t cur_size,
                      size_t new_size);
#endif

#ifdef __cplusplus
extern "C"
#endif
void * pkc_malloc(size_t l);

#ifdef __cplusplus
extern "C"
#endif
void * pkc_realloc(void * p, size_t l);

#ifdef __cplusplus
extern "C"
#endif
void pkc_free(void * b);

#ifdef __cplusplus

#define DEFAULT_EXTENT 256

typedef enum {
    PUBLIC,
    SECURE
} secure_flag_t;

class r_buffer_t {
// A buffer descriptor (intended for read-only buffers)
public:
    unsigned char * orig_data;
    size_t orig_data_length;
    unsigned char * data;
    size_t data_length;
    r_buffer_t(void);
    void set(void);  // Take a snapshot of data/data_length
                     // for later restore
    void reset(void); // Restore data/data_length to previous settings
};

class buffer_t : public r_buffer_t {
protected:
```

```
    secure_flag_t security;
public:
    unsigned char * buffer;
    size_t buffer_length;
    buffer_t(secure_flag_t sec = PUBLIC);
    void secure(secure_flag_t sec = SECURE);
    void detach_buffer(void);
    ~buffer_t();
    unsigned32 clear(void);
    unsigned32 zero(void);
    unsigned32 empty(void);
    unsigned32 extend(size_t extra = 0);
    unsigned32 append(unsigned char c);
    unsigned32 append(const r_buffer_t & o);
    unsigned32 append(unsigned char * b, size_t l);
    unsigned32 append(char * b);
    unsigned32 append_dec(unsigned long i);
    unsigned32 append_dec(signed long i);
    unsigned32 append_hex(unsigned long i);
    unsigned32 append_hex(unsigned char i);

};

enum asn_tag_class {
    UNIVERSAL_CLASS = 0,
    APPLICATION_CLASS = 1,
    CONTEXT_CLASS = 2,
    PRIVATE_CLASS = 3
};

class asn_type_flags {
public:
    char constructed;
    char indefinite;
    asn_tag_class tag_class;
    asn_type_flags(void) {
        constructed = 0;
        indefinite = 0;
        tag_class = UNIVERSAL_CLASS;
    };
    int is_primitive(void) const { return !constructed; };
    int is_constructed(void) const { return constructed; };
    int is_definite(void) const { return !indefinite; };
    int is_indefinite(void) const { return indefinite; };
};

class asn_object {
friend class asn_constructed;
friend class asn_explicit;
protected:
    virtual unsigned32 check_default(void);
    asn_object * containing_object;

    buffer_t encode_buffer;
    char encoding_valid;
```

```
    char value_valid;
    char constant_object; // True for things like the NULL value

    char optional;
    char present;
    char isDefault;
    char defaultable;
    secure_flag_t security;

    char constructed;
    asn_tag_class tag_class;
    unsigned tag_value;

    unsigned32 parse_type_tag(unsigned char * &data,
                              size_t & data_length,
                              unsigned &tag_value,
                              asn_type_flags & flags);
    unsigned32 parse_type_tag(r_buffer_t & data_buffer,
                              unsigned &tag_value,
                              asn_type_flags & flags);

    unsigned32 parse_length(unsigned char * &data,
                            size_t & data_length,
                            size_t & value_length,
                            asn_type_flags &flags);
    unsigned32 parse_length(r_buffer_t & data_buffer,
                            size_t & value_length,
                            asn_type_flags &flags);

    int check_EOC(unsigned char * &data,
                  size_t & data_length);
    int check_EOC(r_buffer_t &data_buffer);

    virtual int check_type(unsigned char * &data,
                           size_t & data_length,
                           unsigned & ber_tag_value,
                           asn_type_flags &flags);
    int check_type(r_buffer_t &data_buffer,
                   unsigned & ber_tag_value,
                   asn_type_flags &flags) {
        return check_type(data_buffer.data,
                          data_buffer.data_length,
                          ber_tag_value,
                          flags);
    };

    virtual void invalidate_encoding(void);
// Derived classes should call this function whenever their encoding
// cache becomes invalid (for example, if the object value becomes
// invalid).  However, the base-class value_changed() member function
// calls invalidate_encoding, so many derived classes don't needd to
// explicitly call this function.

    virtual void value_changed(void);
// Derived classes should invoke this function whenever the value
```

```
// they represent changes.  This is used by the asn classes to
// maintain their encode/decode caches, and the various flags that
// are stored with each object.

    virtual unsigned32 decode_value_indefinite(
                            unsigned char * &data,
                            size_t & max_data_length);
    virtual unsigned32 decode_value_definite(
                            unsigned char * &data,
                            size_t & data_length) = 0;

    virtual unsigned32 encode_type_tag(buffer_t & data_buffer,
                                       unsigned tag_value,
                                       asn_type_flags flags);

    virtual unsigned32 encode_length(buffer_t & data_buffer,
                                     size_t value_length);

    virtual unsigned32 set_valid(char yesno = 1);

// Default assignment operator for asn_object hierarchy.  Note that
// in general there are better ways of copying than using the
// assignment operator because:
// 1) The generic assignment operator is very innefficient - it works
//    by calling write() on the source object, and read on the target
//    object.  Member-wise copies (using get_value/set_value) are
//    likely to be much more efficient.
// 2) If the source and destination object aren't of compatible types,
//    the assignment will fail with undefined results.  Since we choose
//    not to use exceptions, applications won't even be aware of such a
//    failure.
// Note that derived classes can't simply inherit this operator, as the
// compiler will execute it and then execute default assignment
// operations for data members of derived classes, which will in general
// over-write the correct results that this operator has placed there.
// Derived classes that want an assignment operator should declare one,
// and define it to be asn_object::operator=()
//
// Because of the complexities of inheriting this operator, I've decided
// to make it private.  Equivalent functionality is available from the
// public member function copy(), which assignment operators in derived
// classes could invoke.  However, since the assignment operator can't
// signal an error, end-users should use copy() directly, rather than
// sacrificing error-checking for a slightly prettier syntax.

private:
    asn_object & operator = (const asn_object & o);

// Similarly, the copy constructor is dangerous, so make it privaate
private:
    asn_object(const asn_object & o, secure_flag_t sec);

public:

    virtual unsigned32 copy (const asn_object & o);
```

```
// A way to copy the value of another (type-compatible) asn_object
// into this one.

    int is_valid(void) const {
        return value_valid;
    };
    int is_set(void) const {
        return value_valid;
    };
    virtual void set_tag(unsigned tag) {
        tag_value = tag;
    };
    void set_tag_class(asn_tag_class theClass) {
        tag_class = theClass;
    };
    unsigned get_tag_value(void) const { return tag_value; };
    asn_tag_class get_tag_class(void) const { return tag_class; };
    virtual unsigned32 calculate_value(buffer_t & buffer) const = 0;
    virtual unsigned32 calculate_default_value(buffer_t & buffer);
    virtual unsigned32 hash(buffer_t & buffer) const;
    asn_object(secure_flag_t sec);
    virtual ~asn_object();
    int is_primitive(void) const { return !constructed; };
    int is_constructed(void) const { return constructed; };
    virtual unsigned32 make_default(void);
    virtual unsigned32 set_optional(char yesno = 1);
    virtual unsigned32 set_defaultable(void);
    virtual char is_present(void) const { return present; };
    virtual char is_optional(void) const { return optional; };
    virtual char is_default(void) const { return isDefault; };
    virtual char is_defaultable(void) const { return defaultable; };

    virtual unsigned32 read(r_buffer_t & data);
// Parse the supplied data.  The buffer is updated to
// describe the unparsed portions of the input data.

    virtual unsigned32 write(buffer_t & buffer) const;
// Append the DER value of the object to the buffer.


    virtual unsigned32 display(buffer_t & buffer) const;
// Append a human-readable display of the data to the buffer
    virtual int display_quals(buffer_t & buffer) const;
// Append a human-readable display of the object qualifiers.
// Called by display().  Returns 1 if value should be printed

    virtual int operator == (const asn_object & o) const;

    virtual int operator != (const asn_object & o) const {
        return !(*this == o);
    };
};

class asn_constructed : public asn_object {
protected:
```

```
    asn_object ** contained_object;
    size_t contained_object_count;
    asn_object * operator [] (size_t i) const;
    virtual void empty(void);
public:
    int find_child(asn_object * c);
    virtual size_t child_count(void) const {
        return contained_object_count;
    };
    asn_constructed(secure_flag_t sec) : asn_object(sec) {
        constructed = 1;
        contained_object = NULL;
        contained_object_count = 0;
    };
    virtual ~asn_constructed();
    virtual void register_child(asn_object * o);
};

class asn_primitive : public asn_object {
public:
    asn_primitive(secure_flag_t sec) : asn_object(sec) {
        constructed = 0;
    };
};

enum asn_int_value_type {
    SMALLINT, LARGEINT
};

class asn_integer : public asn_primitive {
    virtual unsigned32 check_default(void);
    void cons_int(void);
    signed32 default_data;
    signed32 data;
    unsigned char * data_buffer;
    size_t data_length;
    asn_int_value_type value_type;
public:
    asn_integer(secure_flag_t sec = PUBLIC) : asn_primitive(sec) {
        cons_int();
    };
    asn_integer & operator = (signed32 i);
    asn_integer & operator = (const asn_integer & o);
    virtual unsigned32 set_default_value(signed32 i);
    virtual unsigned32 set_value(signed32 i);
    virtual unsigned32 get_value(signed32 & i) const;
    virtual unsigned32 get_value(unsigned32 & i) const;
    virtual unsigned32 set_value(const unsigned char * b, size_t l);
    virtual unsigned32 get_value(unsigned char * &b, size_t & l) const;
    virtual unsigned32 calculate_value(buffer_t & buffer) const;
    virtual unsigned32 calculate_default_value(buffer_t & buffer);
    virtual unsigned32 decode_value_definite(
                            unsigned char * &data,
                            size_t & data_length);
    virtual unsigned32 display(buffer_t & buffer) const;
```

```
};

class asn_octetstring : public asn_object {
    virtual unsigned32 check_default(void);
    void cons_oct(void);
    unsigned char * data_buffer;
    size_t data_length;
    unsigned char * default_data_buffer;
    size_t default_data_length;
public:
    virtual void set_tag(unsigned tag) {
        tag_value = tag;
    };
    virtual ~asn_octetstring();
    asn_octetstring(secure_flag_t sec = PUBLIC) : asn_object(sec) {
        cons_oct();
    };
    virtual unsigned32 set_value(const unsigned char * b, size_t l);
    virtual unsigned32 get_value(unsigned char * &b, size_t & l) const;
    virtual unsigned32 set_default_value(const unsigned char * b,
                                         size_t l);
    virtual unsigned32 calculate_default_value(buffer_t & buffer);
    virtual unsigned32 calculate_value(buffer_t & buffer) const;
    virtual unsigned32 decode_value_definite(
                            unsigned char * &data,
                            size_t & data_length);
    virtual unsigned32 decode_value_indefinite(
                            unsigned char * &data,
                            size_t & data_length);
    virtual unsigned32 display(buffer_t & buffer) const;
};

class asn_boolean : public asn_object {
    virtual unsigned32 check_default(void);
    void cons_boole(void);
    char data;
    char default_data;
public:
    asn_boolean(secure_flag_t sec = PUBLIC) : asn_object(sec) {
        cons_boole();
    };
    asn_boolean & operator = (char c);
    virtual unsigned32 set_value(char c);
    virtual unsigned32 get_value(char & c) const;
    virtual unsigned32 calculate_value(buffer_t & buffer) const;
    virtual unsigned32 set_default_value(char c);
    virtual unsigned32 calculate_default_value(buffer_t & buffer);
    virtual unsigned32 decode_value_definite(
                            unsigned char * &data,
                            size_t & data_length);
    virtual unsigned32 display(buffer_t & buffer) const;
};

class asn_null : public asn_primitive {
    void cons_null(void);
```

```
public:
    asn_null(secure_flag_t sec = PUBLIC) : asn_primitive(sec) {
        cons_null();
    };
    virtual unsigned32 calculate_value(buffer_t & buffer) const;
    virtual unsigned32 decode_value_definite(
                            unsigned char * &data,
                            size_t & data_length);
    virtual unsigned32 display(buffer_t & buffer) const {
        buffer.append("<NULL>");
        return 0;
    };
};

inline size_t bit_to_byte_length(unsigned32 b) {
    return (b+7)/8;
}

inline unsigned char unused_bit_count(unsigned32 l) {
    return ((8 - (l%8)) & 0x07);
}

inline unsigned char used_bit_mask(unsigned32 b) {
    switch(b%8) {
    case 1: return 0x80; // 7 unused bits
    case 2: return 0xc0; // 6 bits unused
    case 3: return 0xe0; // 5 bits unused
    case 4: return 0xf0; // 4 bits unused
    case 5: return 0xf8; // 3 bits unused
    case 6: return 0xfc; // 2 bits unused
    case 7: return 0xfe; // 1 bit unused
    };
    return 0xff; // All bits used
}

class asn_bitstring : public asn_object {
    virtual unsigned32 check_default(void);
    void cons_bit(void);
    unsigned char * data_buffer;
    size_t data_length;
    unsigned char unused_bits;
    unsigned char * default_data_buffer;
    size_t default_data_length;
    unsigned char default_unused_bits;
public:
    virtual ~asn_bitstring();
    asn_bitstring(secure_flag_t sec = PUBLIC) : asn_object(sec) {
        cons_bit();
    };
    virtual unsigned32 set_value(const unsigned char * b,
                                 unsigned32 l);
    virtual unsigned32 set_default_value(const unsigned char * b,
                                         unsigned32 l);
    virtual unsigned32 get_value(unsigned char * &b, unsigned32 & l)
        const;
```

```
    virtual unsigned32 calculate_value(buffer_t & buffer) const;
    virtual unsigned32 calculate_default_value(buffer_t & buffer);
    virtual unsigned32 decode_value_definite(
                            unsigned char * &data,
                            size_t & data_length);
    virtual unsigned32 decode_value_indefinite(
                            unsigned char * &data,
                            size_t & data_length);
    virtual unsigned32 display(buffer_t & buffer) const;
};

class asn_objectidentifier : public asn_primitive {
    virtual unsigned32 check_default(void);
    void cons_oid(void);
    unsigned long * subidents;
    size_t subident_count;
    unsigned long * default_subidents;
    size_t default_subident_count;
    unsigned32 encode_subident(unsigned long v, buffer_t & buffer)
        const;
    unsigned32 decode_subident(unsigned char * &data,
                               size_t & data_length,
                               unsigned long & encoding) const;
public:
    virtual ~asn_objectidentifier();
    asn_objectidentifier(secure_flag_t sec = PUBLIC)
    : asn_primitive(sec) {
        cons_oid();
    };
    asn_objectidentifier(const unsigned long * b,
                         size_t l,
                         secure_flag_t sec = PUBLIC)
    : asn_primitive(sec) {
        cons_oid();
        set_value(b, l);
    };
    asn_objectidentifier & operator = (const asn_objectidentifier & o);
    virtual unsigned32 set_value(const char * stringVal);
    virtual unsigned32 set_value(const unsigned long * b, size_t l);
    virtual unsigned32 set_value(const asn_objectidentifier &o,
                                 const unsigned long * b,
                                 size_t l);
    virtual unsigned32 set_value(const asn_objectidentifier &o,
                                 unsigned long s);
    virtual unsigned32 set_default_value(const unsigned long * b,
                                         size_t l);
    virtual unsigned32 set_default_value(
                                    const asn_objectidentifier &o,
                                    const unsigned long * b,
                                    size_t l);
    virtual unsigned32 set_default_value(const asn_objectidentifier &o,
                                         unsigned long s);
    virtual unsigned32 get_value(unsigned long *& b, size_t & l) const ;

// The next routine returns the value as a gss_OID object.  Unlike all
```

```
// other get_value members, on entry, the parameter must be either NULL
// or a valid gss_OID.
    virtual unsigned32 get_value(gss_OID & v) const ;

    virtual unsigned32 calculate_default_value(buffer_t & buffer);
    virtual unsigned32 calculate_value(buffer_t & buffer) const;
    virtual unsigned32 decode_value_definite(
                            unsigned char * &data,
                            size_t & data_length);
    virtual unsigned32 display(buffer_t & buffer) const;
    virtual unsigned32 x500_display(buffer_t & buffer) const;
    virtual int operator==(const asn_objectidentifier & o) const;
};

class asn_choice : public asn_constructed {
protected:
    size_t choice;
    virtual unsigned32 decode_value_definite(
                            unsigned char * &data,
                            size_t & data_length);
public:
    asn_choice(secure_flag_t sec = PUBLIC) : asn_constructed(sec) {
        choice = 0;
    };
    virtual unsigned32 set_value(size_t c) {
        if ((choice != c) || !value_valid) {
            choice = c;
            value_changed();
        };
        return 0;
    };
    virtual unsigned32 get_value(size_t & c) const;
    virtual unsigned32 calculate_value(buffer_t & buffer) const;
    virtual unsigned32 display(buffer_t & buffer) const;
    virtual unsigned32 read(r_buffer_t & data);
    virtual unsigned32 write(buffer_t & data) const;
};

#define STRING_SYNTAX_UNSPECIFIED 0
#define STRING_SYNTAX_C 1
#define STRING_SYNTAX_ISO646 2
#define STRING_SYNTAX_IA5 3
#define STRING_SYNTAX_T61 4

class asn_charstring : public asn_octetstring {
    int syntax;
    char * temp_string;
    void cons_chars(unsigned t) {
        syntax = STRING_SYNTAX_UNSPECIFIED;
        tag_value = t;
        temp_string = NULL;
    };
public:
    unsigned32 set_syntax(int s) { syntax = s; return 0; };
    asn_charstring(unsigned t, secure_flag_t sec = PUBLIC)
```

```
    : asn_octetstring(sec) {
        cons_chars(t);
    };
    asn_charstring(unsigned t, const char * s,
                   secure_flag_t sec = PUBLIC)
    : asn_octetstring(sec) {
        cons_chars(t);
        set_value(s);
    };
    virtual ~asn_charstring();
    virtual unsigned32 set_value(const char * s, int translate = 1);
    virtual unsigned32 set_default_value(const char * s,
                                         int translate = 1);
    virtual unsigned32 get_value(char * &s, int translate = 1) const;
    virtual unsigned32 display(buffer_t & buffer) const;
    virtual unsigned32 x500_display(buffer_t & buffer) const;
    virtual unsigned32 read(r_buffer_t & data);
    virtual unsigned32 write(buffer_t & data) const;
};

class asn_conformantstring : public asn_charstring {
// A conformantstring is one that has no fixed ASN type.  It has a
// default type, that's used when a value is loaded by set_value(),
// but any character-string ASN1 type can be succesfully read into
// it, and it's actual type will change to conform.
protected:
    unsigned default_tag_value;
    void cons_conf(unsigned t) {
        default_tag_value = t;
    };
    virtual int check_type(unsigned char * &data,
                           size_t & data_length,
                           unsigned & ber_tag_value,
                           asn_type_flags &flags);
public:
    asn_conformantstring(unsigned t, secure_flag_t sec = PUBLIC)
    : asn_charstring(t, sec) {
        cons_conf(t);
    };
    asn_conformantstring(unsigned t,
                         char * s,
                         secure_flag_t sec = PUBLIC)
    : asn_charstring(t, s, sec) {
        cons_conf(t);
    };
    virtual unsigned32 set_value(const char * s, int translate = 1) {
        asn_charstring::set_tag(default_tag_value);
        return asn_charstring::set_value(s, translate);
    };
    virtual unsigned32 set_default_value(const char * s,
                                         int translate = 1) {
        return asn_charstring::set_default_value(s, translate);
    };

};
```

```
class asn_IA5string : public asn_charstring {
public:
    asn_IA5string(secure_flag_t sec = PUBLIC)
    : asn_charstring(22, sec) { set_syntax(STRING_SYNTAX_IA5); };
};

class asn_T61string : public asn_charstring {
public:
    asn_T61string(secure_flag_t sec = PUBLIC)
    : asn_charstring(20, sec) { set_syntax(STRING_SYNTAX_T61); };
};

class asn_utctime : public asn_charstring {
    unsigned century_start;
    char ignore_seconds;
public:
    asn_utctime(secure_flag_t sec = PUBLIC);
// Override the charstring set_value function as it would
// provide a way to break DER-encoding.
// Note that if you invoke the char * version of get_value,
// you get no Y2K help.
    virtual unsigned32 set_value(const char * s) {
        return pkc_s_unimplemented;
    };
    virtual unsigned32 set_value(utc_t time);
    virtual unsigned32 get_value(utc_t & time) const;
    virtual unsigned32 set_value(unsigned year,
                                 unsigned month,            // 1..12
                                 unsigned day,              // 1..31
                                 unsigned hour,             // 0..23
                                 unsigned minute,           // 0..59
                                 unsigned sec,              // 0..59
                                 int tz_hour = 0,           // -12..+12
                                 unsigned tz_min = 0);      // 0..59
// The tz_min is intepreted as adding to the magnitude of the tz_hour,
// and having the same sign as the tz_hour.  Thus a time-zone of
// GMT-5:30 would be expressed as tz_hour = -5, and tz_min = 30
    virtual unsigned32 get_value(unsigned &year,
                                 unsigned &month,
                                 unsigned &day,
                                 unsigned &hour,
                                 unsigned &minute,
                                 unsigned &sec,
                                 int &tz_hour,
                                 unsigned &tz_min) const ;
    virtual unsigned32 set_century_rollover(unsigned s);
// set_century_rollover is a hack to avoid the Y2K problem that is
// inherent with the UTCtime type.  UTCtime only stores two digits for
// the year portion of the date, and assumes 19XX as the interpretation
// of this.  Thus the basic type is incapable of expressing dates
// after 1999.  set_century_rollover(s) tells the class to treat
// parsed years before s as 20XX.  "s" should be in the range 0 to 99.

    virtual unsigned32 set_ignore_seconds(char t = 1);
// Tells the class whether to include seconds in the generated BER,
```

```
// or whether to ignore them.  Seconds are always supressed if they
// are zero.
};

class asn_sequence : public asn_constructed {
protected:
    virtual unsigned32 decode_value_definite(
                            unsigned char * &data,
                            size_t & data_length);
    virtual unsigned32 decode_value_indefinite(
                            unsigned char * &data,
                            size_t & data_length);

    void cons_seq(void);
    asn_object * operator [] (size_t i) const {
        return asn_constructed::operator[](i);
    }
    virtual void empty(void) { asn_constructed::empty(); };
public:
    asn_sequence(secure_flag_t sec = PUBLIC)
    : asn_constructed(sec) {
        cons_seq();
    };
    virtual unsigned32 calculate_value(buffer_t & buffer) const;
    virtual unsigned32 display(buffer_t & buffer) const;
};

class asn_sortable : public asn_constructed {
protected:
    virtual void presort_elements(void) {};
    virtual void sort_elements(void);
    virtual int compare_objects(int i, int j) = 0;
// returns 1 if i > j, 0 if i = j, -1 if i < j
    virtual void swap_objects(int i, int j);
// swaps objects i & j
public:
    virtual unsigned32 write(buffer_t & buffer) const;
    asn_sortable(secure_flag_t sec) : asn_constructed(sec) {};
};

class asn_set : public asn_sortable {
protected:
    char * component_present;
    virtual unsigned32 decode_value_definite(
                            unsigned char * &data,
                            size_t & data_length);
    virtual unsigned32 decode_value_indefinite(
                            unsigned char * &data,
                            size_t & data_length);

    void cons_set(void);
    virtual unsigned32 set_child_present(int i);
    virtual unsigned32 set_child_absent(int i);
    virtual int compare_objects(int i, int j);
    virtual void swap_objects(int i, int j);
```

```
    virtual void register_child(asn_object * o) {
        asn_sortable::register_child(o);
        component_present =
            (char *)pkc_realloc(component_present,
                                contained_object_count);
        component_present[contained_object_count-1] = 0;
    };
public:
    virtual ~asn_set();
    asn_set(secure_flag_t sec = PUBLIC) : asn_sortable(sec) {
        cons_set();
    };
    virtual unsigned32 calculate_value(buffer_t & buffer) const;
    virtual unsigned32 display(buffer_t & buffer) const;
    virtual int child_present(int i);
    virtual unsigned32 set_child_present(asn_object * c);
    virtual unsigned32 set_child_absent(asn_object * c);
};

// Unlike classes with fixed elements, sequence_of and set_of classes
// own their children - that is, they are dynamically created and
// deleted.  The only way to access the children is via the class []
// operator.

template<class elem_t> class asn_sequenceof : public asn_sequence {
protected:
    virtual unsigned32 decode_value_definite(
                            unsigned char * &data,
                            size_t & data_length) {

        r_buffer_t buffer;
        unsigned32 error;
        size_t the_child = 0;

        buffer.data = data;
        buffer.data_length = data_length;

        empty();  // Remove any current children

        while (buffer.data_length != 0) {
// Create new children as needed.
            register_child(new elem_t);
            if (error =
                  asn_sequence::operator[](the_child)->read(buffer)) {
                return error;
            };
            the_child++;
        }
        data = buffer.data;
        data_length = 0;
        return 0;
    };
    virtual unsigned32 decode_value_indefinite(
                            unsigned char * &data,
                            size_t & data_length) {
```

```
        r_buffer_t buffer;
        unsigned32 error;
        size_t the_child = 0;

        buffer.data = data;
        buffer.data_length = data_length;

        empty();  // Remove any current children

        while (buffer.data_length > 0) {
            if (check_EOC(buffer)) {
                data = buffer.data;
                data_length = buffer.data_length;
                return 0;
            };
// Create new children as needed.
            register_child(new elem_t);
            if (error =
                   asn_sequence::operator[](the_child)->read(buffer)) {
                return error;
            };
            the_child++;
        }
        return pkc_s_asn_no_more_data;
    };
public:
    virtual void empty(void) {
        int i;
        for (i=0; i<child_count();i++) {
            delete (*this)[i];
        };
        asn_sequence::empty();
    };
    asn_sequenceof(secure_flag_t sec = PUBLIC) : asn_sequence(sec) {};
    virtual ~asn_sequenceof() { empty(); };
    virtual elem_t * operator [] (size_t i) const {
        return (elem_t *)(asn_sequence::operator[](i));
    };
    virtual elem_t * new_child(void) {
        elem_t * the_child;
        the_child = new elem_t;
        register_child(the_child);
        return the_child;
    };
};

template<class elem_t> class asn_setof : public asn_set {
    buffer_t ** encoding;

protected:
    virtual void register_child(asn_object * o) {
        asn_set::register_child(o);
        component_present[contained_object_count-1] = 1;
    };
    virtual void presort_elements(void) {
```

```
// Called immediately before sorting the elements.  We have to run
// through the elements and ask them to encode themeselves
        int i;

        if (component_present == NULL)
            component_present =
                  (char*)pkc_malloc(contained_object_count);

        if (encoding) pkc_free(encoding);
        encoding =
            (buffer_t **)pkc_malloc(
                          contained_object_count * sizeof(buffer_t *));

        for (i=0;i<contained_object_count;i++) {
            encoding[i] = new buffer_t(security);
            contained_object[i]->write(*encoding[i]);
// This has to be set so the set:: routines will do their stuff
            component_present[i] = 1;
        };
    };
    virtual int compare_objects(int i, int j) {
// Compare the two encodings.  If the encodings differ within the length
// of the smaller, then use lexicographic ordering; if not, then the
// shorter string is smaller.
        int x;
        size_t len;
        len = ((encoding[i]->data_length < encoding[j]->data_length)
               ?encoding[i]->data_length : encoding[j]->data_length);
        for (x=0; x < len; x++) {
            if (encoding[i]->data[x] < encoding[j]->data[x])
                return -1;
            else if (encoding[i]->data[x] > encoding[j]->data[x])
                return 1;
        };
        if (encoding[i]->data_length < encoding[j]->data_length)
            return -1;
        else if (encoding[i]->data_length > encoding[j]->data_length)
            return 1;
        else return 0;
    };
    virtual void swap_objects(int i, int j) {
        buffer_t * temp;
        asn_sortable::swap_objects(i, j);
        temp = encoding[i];
        encoding[i] = encoding[j];
        encoding[j] = temp;
    };
    virtual unsigned32 decode_value_definite(
                          unsigned char * &data,
                          size_t & data_length) {

        r_buffer_t buffer;
        unsigned32 error;
        size_t the_child = 0;
```

```
        buffer.data = data;
        buffer.data_length = data_length;

        empty();  // Remove any current children

        while (buffer.data_length != 0) {
// Create new children as needed.
            register_child(new elem_t);
            if (error = asn_set::operator[](the_child)->read(buffer)) {
                return error;
            };
            the_child++;
        }
        data = buffer.data;
        data_length = 0;
        return 0;
    };
    virtual unsigned32 decode_value_indefinite(
                            unsigned char * &data,
                            size_t & data_length) {
        r_buffer_t buffer;
        unsigned32 error;
        size_t the_child = 0;

        buffer.data = data;
        buffer.data_length = data_length;

        empty();  // Remove any current children

        while (buffer.data_length > 0) {
            if (check_EOC(buffer)) {
                data = buffer.data;
                data_length = buffer.data_length;
                return 0;
            };
// Create new children as needed.
            register_child(new elem_t);
            if (error = asn_set::operator[](the_child)->read(buffer)) {
                return error;
            };
            the_child++;
        }
        return pkc_s_asn_no_more_data;
    };
public:
    virtual void empty(void) {
        int i;
        for (i=0; i<child_count();i++) {
            delete (*this)[i];
            if (encoding) delete encoding[i];
        };
        pkc_free(encoding); encoding = NULL;
        asn_set::empty();
    };
    virtual ~asn_setof() { empty(); };
```

```
    asn_setof(secure_flag_t sec = PUBLIC) : asn_set(sec) {
        encoding = NULL;
    };
    virtual elem_t * operator [] (size_t i) const {
        return (elem_t *)(asn_set::operator[](i));
    };
    virtual elem_t * new_child(void) {
        elem_t * the_child;
        the_child = new elem_t;
        register_child(the_child);
        return the_child;
    };
};

// asn_explicit is actually implemented as an implicitly tagged
// sequence.  This construct results in correct encoding, and
// saves writing and debugging any new code.  The only downside is
// that it's possible for an application to register multiple
// children of an asn_explicit, which wouldn't be legal ASN.1.
class asn_explicit : public asn_sequence {
public:
    asn_explicit(asn_tag_class tagClass,
                 unsigned tagValue,
                 secure_flag_t sec = PUBLIC) : asn_sequence(sec) {
        set_tag(tagValue);
        set_tag_class(tagClass);
    };
    asn_explicit(unsigned tagValue,
                 secure_flag_t sec = PUBLIC) : asn_sequence(sec) {
        set_tag(tagValue);
        set_tag_class(CONTEXT_CLASS);
    };
    virtual char is_default(void) const {
        return contained_object[0]->is_default();
    };
    virtual char is_defaultable(void) const {
        return contained_object[0]->is_defaultable();
    };
    virtual unsigned32 check_default(void) {
        return contained_object[0]->check_default();
    };
    virtual unsigned32 calculate_default_value(buffer_t & buffer) {
        // We know the contained object is it's default value, so
        // just ask it to encode itself...
        return contained_object[0]->write(buffer);
    };
    virtual unsigned32 display(buffer_t & buffer) const;
};

// The class for ANY is a bit of a kludge, in that it doesn't really
// work for a true ANY value, only those values that are encoded in BER.
// It works by parsing the incoming BER stream, and stores the entire
// result as its value. Today, only definite-length-encoded ANY values
// are supported by the parser, although this restriction could be lifted
// fairly easily.
```

```
class asn_any : public asn_object {
    buffer_t value_encoding;
    buffer_t total_encoding;
    virtual int check_type(unsigned char * &data,
                           size_t & data_length,
                           unsigned & ber_tag_value,
                           asn_type_flags &flags);
public:
    asn_any(secure_flag_t sec = PUBLIC) : asn_object(sec) {
        value_encoding.empty();
        value_encoding.secure(security);
        total_encoding.secure(security);
    };
    virtual unsigned32 set_value(const asn_object & val);
    virtual unsigned32 get_value(unsigned char * &b, size_t & l) const;
    virtual unsigned32 decode_value_definite(
                            unsigned char * &data,
                            size_t & data_length) {
        value_encoding.empty();
        return value_encoding.append(data, data_length);
    };
    virtual unsigned32 calculate_value(buffer_t & buffer) const {
        return buffer.append(value_encoding);
    };
};

#endif
#endif
```

8.5.2. x509.h

```
/*
 * @OSF_COPYRIGHT@
 * COPYRIGHT NOTICE
 * Copyright (c) 1996 Open Software Foundation, Inc.
 * ALL RIGHTS RESERVED (DCE).  See the file named COPYRIGHT.DCE for
 * the full copyright text.
 * All Rights Reserved
 */
/*
 * HISTORY
 * $Log: x509.h,v $
 * Revision 1.1.2.2  1996/11/13  20:40:54  arvind
 *      OSF DCE 1.2.2 Drop 5
 *
 *      Cert-API drop 2 from DEC
 *      [1996/11/03  23:26 UTC  arvind  /main/DCE_1.2.2/arvind_capi2/1]
 *
 *      CAPI drop from DEC
 *      [1996/10/17  00:21 UTC  arvind  /main/DCE_1.2.2/arvind_capi/1]
 *      Revision 1.1.2.1  1996/10/04  19:57:32  arvind
 *      OSF DCE 1.2.2 Drop 4
 *      [1996/10/04  19:57:14  arvind]
 *
 * Revision /main/DCE_1.2.2/1  1996/09/09  21:26 UTC  arvind
 *      PKSS drop from DEC (DCE1.2.2)
```

```
 *      [1996/08/30  16:11 UTC  arvind  /main/arvind_pkss/1]
 *
 * $EndLog$
 */
/*
 *
 * (c) Copyright 1996,
 * Digital Equipment Corporation, Maynard, Massachusetts, USA
 * All Rights Reserved
 *
 */

#ifndef X509_
#define X509_

#ifdef NOT_DCE
#include "asn.h"
#include "x500name.h"
#else
#include <dce/asn.h>
#include <dce/x500name.h>
#endif

// Version in X.509 is EXPLICIT [0] INTEGER DEFAULT 0
class x509_version : public asn_explicit {
public:
    asn_integer value;
    x509_version(void) : asn_explicit(0) {
        register_child(&value);
        value.set_default_value(0);
        value.set_value(0);
    };
};

template<unsigned tag> class tagged_bitstring : public asn_explicit {
public:
    asn_bitstring value;
    tagged_bitstring(void) : asn_explicit(tag) {
        register_child(&value);
    };
};

class AlgorithmIdentifier : public asn_sequence {
public:
    asn_objectidentifier algorithm;
    asn_any parameters;
    AlgorithmIdentifier(void) {
        register_child(&algorithm);
        register_child(&parameters);
    };
    int operator == (const AlgorithmIdentifier & o) const;
};

class Validity : public asn_sequence {
public:
```

```
    asn_utctime notBefore;
    asn_utctime notAfter;
    Validity(void) {
        notBefore.set_century_rollover(88);
        notBefore.set_ignore_seconds();
        notAfter.set_century_rollover(88);
        notAfter.set_ignore_seconds();
        register_child(&notBefore);
        register_child(&notAfter);
    };
};

class SubjectPublicKeyInfo : public asn_sequence {
public:
    AlgorithmIdentifier algorithm;
    asn_bitstring subjectPublicKey;
    SubjectPublicKeyInfo(void) {
        register_child(&algorithm);
        register_child(&subjectPublicKey);
    };
};

class Extension : public asn_sequence {
public:
    asn_objectidentifier extnID;
    asn_boolean critical;
    asn_octetstring extnValue;
    Extension(void) {
        critical.set_default_value(0);
        critical.set_value(0);
        register_child(&extnID);
        register_child(&critical);
        register_child(&extnValue);
    };
};

class Extensions : public asn_sequenceof<Extension> {
};

template<unsigned tag> class tagged_extensions : public asn_explicit {
public:
    Extensions value;
    tagged_extensions(void) : asn_explicit(tag) {
        register_child(&value);
    };
};

class CertificateInfo : public asn_sequence {
public:
    x509_version version;
    asn_integer serialNumber;
    AlgorithmIdentifier signatureAlgorithm;
    x500name issuer;
    Validity validity;
    x500name subject;
```

```
    SubjectPublicKeyInfo subjectPublicKeyInfo;
    tagged_bitstring<1> issuerUniqueIdentifier;
    tagged_bitstring<2> subjectUniqueIdentifier;
    tagged_extensions<3> extensions;
    CertificateInfo(void) {
        register_child(&version);
        register_child(&serialNumber);
        register_child(&signatureAlgorithm);
        register_child(&issuer);
        register_child(&validity);
        register_child(&subject);
        register_child(&subjectPublicKeyInfo);
        register_child(&issuerUniqueIdentifier);
        register_child(&subjectUniqueIdentifier);
        register_child(&extensions);
        issuerUniqueIdentifier.set_optional();
        subjectUniqueIdentifier.set_optional();
        extensions.set_optional();
    };
};

class Certificate : public asn_sequence {
    buffer_t cached_verifier_key;
protected:
    virtual void value_changed(void);
public:
    CertificateInfo certificateInfo;
    AlgorithmIdentifier signatureAlgorithm;
    asn_bitstring signature;
    Certificate(void) {
        register_child(&certificateInfo);
        register_child(&signatureAlgorithm);
        register_child(&signature);
    };
    unsigned32 check_signature(const r_buffer_t & publicKeyInfo) const;
    unsigned32 sign(const r_buffer_t & privateKey);
};

template <unsigned tag> class taggedCertificate : public asn_explicit {
public:
    Certificate cert;
    taggedCertificate(void) : asn_explicit(tag) {
        register_child(&cert);
    };
};

class CrossCertificatePair : public asn_sequence {
public:
    taggedCertificate<0> forward;
    taggedCertificate<1> reverse;
    CrossCertificatePair(void) {
        forward.set_optional();
        reverse.set_optional();
// A note in ISO 9594-8 says that at least one component shall be
// present.  Since this restriction can't be expressed in ASN.1,
```

```
// it's up to the caller to verify that a succesful read operation
// actually consumed some bytes.
        register_child(&forward);
        register_child(&reverse);
    };
};

class Revocation : public asn_sequence {
public:
    asn_integer certificateSerialNumber;
    asn_utctime revocationDate;
    Extensions extensions;
    Revocation(void) {
        revocationDate.set_century_rollover(88);
        revocationDate.set_ignore_seconds();
        extensions.set_optional();
        register_child(&certificateSerialNumber);
        register_child(&revocationDate);
        register_child(&extensions);
    };

};

class RevokedCerts : public asn_sequenceof<Revocation> {
};

class CertificateList : public asn_sequence {
public:
    x509_version version;
    AlgorithmIdentifier signatureAlgorithm;
    x500name issuer;
    asn_utctime thisUpdate;
    asn_utctime nextUpdate;
    RevokedCerts revokedCertificates;
    tagged_extensions<0> extensions;
    CertificateList(void) {
        thisUpdate.set_century_rollover(88);
        thisUpdate.set_ignore_seconds();
        nextUpdate.set_century_rollover(88);
        nextUpdate.set_ignore_seconds();
        version.set_optional();
        revokedCertificates.set_optional();
        extensions.set_optional();
        register_child(&version);
        register_child(&signatureAlgorithm);
        register_child(&issuer);
        register_child(&thisUpdate);
        register_child(&nextUpdate);
        register_child(&revokedCertificates);
        register_child(&extensions);
    };
};

class SignedCertificateList : public asn_sequence {
    buffer_t cached_verifier_key;
```

```
protected:
    virtual void value_changed(void);
public:
    CertificateList certificateList;
    AlgorithmIdentifier signatureAlgorithm;
    asn_bitstring signature;
    SignedCertificateList(void) {
        register_child(&certificateList);
        register_child(&signatureAlgorithm);
        register_child(&signature);
    };
    unsigned32 check_signature(const r_buffer_t & publicKeyInfo) const;
    unsigned32 sign(const r_buffer_t & privateKey);
};
#endif
```

8.5.3. pkc_certs.h

```
/*
 * @OSF_COPYRIGHT@
 * COPYRIGHT NOTICE
 * Copyright (c) 1996 Open Software Foundation, Inc.
 * ALL RIGHTS RESERVED (DCE).  See the file named COPYRIGHT.DCE for
 * the full copyright text.
 * All Rights Reserved
 */
/*
 * HISTORY
 * $Log: pkc_certs.h,v $
 * Revision 1.1.2.1  1996/11/13  20:36:58  arvind
 *      OSF DCE 1.2.2 Drop 5
 *      [1996/11/13  19:54:34  arvind]
 *
 * Revision /main/DCE_1.2.2/2  1996/11/04  22:26 UTC  arvind
 *      Cert-API drop 2 from DEC
 *      [1996/11/03  23:23 UTC  arvind  /main/DCE_1.2.2/arvind_capi2/1]
 *
 * Revision /main/DCE_1.2.2/1  1996/10/18  00:39 UTC  arvind
 *      CAPI drop from DEC
 *      [1996/10/16  20:31 UTC  arvind  /main/arvind_capi/1]
 *
 * $EndLog$
 */
/*
 * @DEC_COPYRIGHT@
 * Copyright (c) 1996
 * Digital Equipment Corporation, Maynard, Massachusetts, USA
 * All Rights Reserved
 *
 * This software is furnished under a license and may be used and copied
 * only  in  accordance  with  the  terms  of such  license and with the
 * inclusion of the above copyright notice.  This software or any  other
 * copies thereof may not be provided or otherwise made available to any
 * other person.  No title to and ownership of the  software  is  hereby
 * transferred.
 *
```

```
 * The information in this software  is subject to change without notice
 * and  should  not be  construed  as a commitment by Digitial Equipment
 * Corporation.
 *
 * Digital assumes no responsibility for the use or  reliablity  of its
 * software on equipment which is not supplied by Digital.
 *
 * FACILITY:    Certification API
 *
 * ABSTRACT:    Data-Structures and proto-types for pkc_certs.cxx
 *
 *
 * ENVIRONMENT: portable
 *
 * AUTHOR:
 *              John Wray
 *              Distributed Processing Engineering (DPE)
 *
 * DATE:        March 1996
 *
 */

#ifndef PKC_CERTS_
#define PKC_CERTS_

#ifndef __cplusplus
#error "C++ must be used for direct certificate manipulation"
#endif

#include <dce/x509.h>
/*** #include <dce/pkc_platform_specific.h>  **/

extern "C" {
#include <stddef.h>
#include <dce/utc.h>
#include <dce/gssapi.h>
#include <dce/pkc_consts.h>
};

#ifdef UTC_OPERATORS

char periods_overlap(utc_t p1_start, utc_t p1_end,
                     utc_t p2_start, utc_t p2_end);
// Returns non-zero if p1 & p2 overlap

int operator == (const utc_t & o1, const utc_t & o2);
int operator != (const utc_t & o1, const utc_t & o2);
int operator <= (const utc_t & o1, const utc_t & o2);
int operator >= (const utc_t & o1, const utc_t & o2);
int operator < (const utc_t & o1, const utc_t & o2);
int operator > (const utc_t & o1, const utc_t & o2);

#endif

class pkc_generic_key_usage_t {
```

```
protected:
    virtual unsigned long pruned(unsigned long x) const;
public:
    unsigned long permitted;

    pkc_generic_key_usage_t(unsigned long permit_bits = 0xfffffffflu);
    char is_permitted(unsigned long check) const;
    char is_permitted(const pkc_generic_key_usage_t & check) const;
    void constrain(unsigned long constraint);
    void constrain(const pkc_generic_key_usage_t & constraint);
    void set(unsigned long constraints);
    pkc_generic_key_usage_t & operator = (unsigned long c);
};

class pkc_key_usage_t :  public pkc_generic_key_usage_t {
protected:
    virtual unsigned long pruned(unsigned long x) const;
public:
    pkc_key_usage_t & operator = (unsigned long c);
    pkc_key_usage_t(unsigned long c = 0xfffffffflu);
};

class pkc_ca_key_usage_t  : public pkc_generic_key_usage_t {
protected:
    virtual unsigned long pruned(unsigned long x) const;
public:
    pkc_ca_key_usage_t & operator = (unsigned long c);
    pkc_ca_key_usage_t(unsigned long c = 0xfffffffflu);
};

class pkc_key_policy_t {
public:
    gss_OID value;
    pkc_key_policy_t(void);
    virtual ~pkc_key_policy_t();
    pkc_key_policy_t & operator = (const pkc_key_policy_t & o);
    pkc_key_policy_t & operator = (const gss_OID & o);
};

class pkc_key_policies_t {
    char all_policies;
    gss_OID_set policies;
public:
    pkc_key_policies_t(void);  // Initialize to "all policies OK".
    unsigned32 set(const pkc_key_policy_t & pol);
    unsigned32 set(const gss_OID pol);  // Adds an allowed policy
    unsigned32 set_none(void); // Sets no policies permitted
    unsigned32 set_all(void); // Sets all policies permitted
    unsigned32 constrain(const pkc_key_policies_t & pol);
    pkc_key_policies_t & operator = (const pkc_key_policies_t & pol);
};

#define PKC_NAME_CONSTRAINT_NONE 0ul
#define PKC_NAME_CONSTRAINT_CA_SUBORDINATE 1ul
#define PKC_NAME_CONSTRAINT_CA_SUPERIOR_SUBORDINATE 2ul
```

```
class pkc_name_subord_constraints_t;

class pkc_name_subord_constraint_t {
private:
    pkc_name_subord_constraints_t * root;
private:
    pkc_name_subord_constraint_t & operator =
          (const pkc_name_subord_constraint_t & o) {return *this;};
public:
    pkc_name_subord_constraint_t * next;
    pkc_name_subord_constraint_t * prev;

    unsigned long constraint;
    unsigned skipCerts;

    void unlink(void);

    ~pkc_name_subord_constraint_t();
    pkc_name_subord_constraint_t(
                            pkc_name_subord_constraints_t * theRoot);
    pkc_name_subord_constraint_t(void);
    void set_constraint(unsigned long c);
    void set_skipCerts(unsigned c);
    void get_next_link_constraint(
                  pkc_name_subord_constraint_t ** new_constraint) const;
// Generate a new name-subord constraint that will be applicable to a
// certificate issued by the subject of this constraint.

    char is_permitted(const x500name & issuer_name,
                      const x500name & subject_name) const;
// Return value of 1 means permitted, 0 means forbidden, -1 means not
// relevant, -2 means not relevant, but explicit permission is required
// from another subord constraint.
};

class pkc_name_subtree_constraints_t;

class pkc_name_subtree_constraint_t {
private:
    pkc_name_subtree_constraints_t * root;
private:
    pkc_name_subtree_constraint_t & operator =
          (const pkc_name_subtree_constraint_t & o) {return *this;};
public:
    pkc_name_subtree_constraint_t * next;
    pkc_name_subtree_constraint_t * prev;

    x500name * base;
    x500name * chopBefore;
    x500name * chopAfter;
    unsigned minimum;
    unsigned maximum;

    void unlink(void);
```

```
    virtual ˜pkc_name_subtree_constraint_t();
    pkc_name_subtree_constraint_t(
                pkc_name_subtree_constraints_t * theRoot);
    void set_base(const x500name & n);
    void set_chopBefore(const x500name & n);
    void set_chopAfter(const x500name & n);
    void set_minimum(unsigned n);
    void set_maximum(unsigned n);

    char is_permitted(const x500name & subject_name) const;
// Return value of 1 means permitted, 0 means forbidden, -1 means
// not relevant, -2 means not forbidden, but explicit permission is
// required from another subtree constraint.

    void get_next_link_constraint(
                  pkc_name_subtree_constraint_t ** new_constraint) const;
// Generate a new name-subtree constraint that will be applicable to a
// certificate issued by the subject of this constraint.
};

class pkc_name_subord_constraints_t {
// A set of subordinate constraints on a name.
public:
    pkc_name_subord_constraints_t & operator =
                     (const pkc_name_subord_constraints_t & o);
public:
    pkc_name_subord_constraint_t * first;
    pkc_name_subord_constraint_t * last;
    pkc_name_subord_constraints_t(void);
    ~pkc_name_subord_constraints_t();
    char is_permitted(const x500name & ca_name,
                      const x500name & subject_name) const;
    void get_next_link_constraint(
                pkc_name_subord_constraints_t ** new_constraints) const;
    unsigned32 constrain(const pkc_name_subord_constraint_t & o);
    unsigned32 constrain(const pkc_name_subord_constraints_t & o);

};

class pkc_name_subtree_constraints_t {
// A set of subtree constraints on a name.
public:
    pkc_name_subtree_constraints_t & operator =
                        (const pkc_name_subtree_constraints_t & o);
public:
    pkc_name_subtree_constraint_t * first;
    pkc_name_subtree_constraint_t * last;
    pkc_name_subtree_constraints_t(void);
    virtual ˜pkc_name_subtree_constraints_t();
    char is_permitted(const x500name & subject_name) const;
    void get_next_link_constraint(
                pkc_name_subtree_constraints_t ** new_constraints) const;
    unsigned32 constrain(const pkc_name_subtree_constraint_t & o);
    unsigned32 constrain(const pkc_name_subtree_constraints_t & o);
};
```

```
class pkc_constraints_t {
// A class that expresses constraints on the names that can be
// certified by a given key.  At the moment, there are three
// types of constraints that can be checked: total path-length,
// name subordination and subtree constraints.

public:
    pkc_constraints_t & operator = (const pkc_constraints_t & o);
public:
    unsigned path_length;  // The maximum path-length that can be
                           // certified by the key (if the entity can act
                           // as a CA). 0xffffu means unlimited.
    pkc_name_subord_constraints_t subord_constraints;
    pkc_name_subtree_constraints_t subtree_constraints;

    pkc_constraints_t(void);
    unsigned32 constrain(const pkc_constraints_t & o);
                           // Adds the specified constraints
    char is_permitted(const x500name & ca_name,
                      const x500name & subject_name)
        const;

    void get_next_link_constraint(
                  pkc_constraints_t ** new_constraints) const;
// Generate a new name-constraint that will be applicable to a
// certificate issued by the subject of this constraint.
};

class pkc_trust_list_element_t;
class pkc_trusted_key_list_t;
class pkc_trusted_key_t {
friend class pkc_trust_list_t;
// A class that expresses trust in a public-key.  Very much like
// a certificate,  but with trust pre-established, rather than
// based on a signature.

friend unsigned32 pkc_add_trusted_key(
                         pkc_trust_list_t * trust_list,
                         const pkc_trusted_key_t & key);
// Adds a key to a trustlist.

friend unsigned32 pkc_lookup_keys_in_trustlist(
                         pkc_trust_list_t * trust_list,   /* in */
                         pkc_trusted_key_list_t * & keys, /* out */
                         size_t * key_count,              /* out */
                         const x500name & owner,          /* in */
                         utc_t * key_time,
                                                /* in, optional */
                         const pkc_generic_key_usage_t * usages);
                                                /* in, optional */
// Retrieves keys for the specified principal for the specified usages
// that are valid at the specified time.  If usages is NULL, keys for
// any usage are returned; if key-time is NULL, keys for the current
// time are returned.  Note that this routine may return a success
// status even when no keys are actually found (in this case, an
```

```
// empty pkc_trusted_key_list_t will have been returned via the
// keys parameter, and must be deleted).

friend unsigned32 pkc_lookup_key_in_trustlist(
                         pkc_trust_list_t * trust_list,
                         const pkc_trusted_key_t ** key,
                         unsigned long key_id);
// Retrieves the specified key by index.

friend unsigned32 pkc_lookup_element_in_trustlist(
                         pkc_trust_list_t * trust_list,
                         const pkc_trust_list_element_t ** key,
                         unsigned long key_id);
// Retrieves the specified key by index, returning it as a
// trust-list-element

friend unsigned32 pkc_check_cert_against_trustlist(
                         pkc_trust_list_t * trust_list,
                         const Certificate * cert,
                         int revoked_certs_permitted);
// Attempt to add a certificate to a trust-list.  The routine will
// succeed only if the trust-list contains a key that can be used
// to verify the certificate.  revoked_certs_permitted specifies
// whether it is permissible to add a certificate that appears
// on a verified CRL if there is a "window of validity" between
// the verifying key's start-time and its revocation time.

friend unsigned32 pkc_revoke_certificate(
                                 pkc_trust_list_t * trust_list,
                                 const x500name & issued_by,
                                 const asn_integer & serial_no,
                                 utc_t * invalidate_from);
// Revoke a single certificate.

friend unsigned32 pkc_revoke_certificates(
                    pkc_trust_list_t * trust_list,
                    const SignedCertificateList * crl);
// Revoke certificates based on a CRL.  If the signature on
// the CRL can be verified, it will be processed immediately.
// If not, it will be saved and each certificate subsequently
// added will attempt to verify the CRL.  As soon as the CRL
// can be verified it will be applied.

friend unsigned32 pkc_prune_trustlist(pkc_trust_list_t * trust_list,
                         const utc_t * prune_before);
// Remove certificates that have expired at the given date
// (NULL means now).

friend unsigned32 pkc_delete_trustlist(pkc_trust_list_t * trust_list);
// Delete a trust-list.

friend unsigned32 pkc_copy_trustlist(
                         const pkc_trust_list_t * input_trust_list,
                         pkc_trust_list_t * output_trust_list);
// Copy a trust-list.
```

```
friend unsigned32 pkc_display_trustlist(
                         const pkc_trust_list_t * input_trust_list);

protected:
// Fields from the certificate:
    utc_t start_date;
    utc_t end_date;
    pkc_ca_key_usage_t ca_usages;
    pkc_key_usage_t user_usages;
    pkc_key_policies_t policies;
    pkc_constraints_t constraints;

// Flags.
// Is this entry trusted (a-priori)?
    char trusted;
// Is it certified by another entry?
    char certified;

    x500name certified_by;      // Name of CA that certified this key.
    asn_integer serial_number;  // Serial no of certifying certificate

// The following fields are copied from the certifying key entry
    utc_t certified_start_date;
    utc_t certified_end_date;

    pkc_ca_key_usage_t certified_usages;
    pkc_key_policies_t certified_policies;
    pkc_constraints_t certified_constraints;

    char revoked; // non-zero if the certifying certificate
                  // has been revoked
    utc_t revocation_date; // Date from which certifier revocation
                           // is effective.
    utc_t revalidation_date; // Most recent CRL we've seen that
                             // doesn't contain this certificate.


    unsigned long key_id;       // An ID identifying this Key entry
    unsigned long ca_key_id;    // The ID of the key that certified
                                // this one.  0 means direct trust.

    unsigned long old_key_id;   // Temporary storage for use while
                                // copying
    unsigned long old_ca_key_id;

public:

    SubjectPublicKeyInfo value;  /* mbg */
    x500name owner;

    virtual unsigned32 display(buffer_t & buffer) const;
    pkc_trusted_key_t(void);
    pkc_trusted_key_t(utc_t startDate,
                      utc_t endDate,
                      pkc_ca_key_usage_t caUsages,
```

```
                      pkc_key_usage_t userUsages,
                      pkc_key_policies_t keyPolicies,
                      pkc_constraints_t keyConstraints);
    unsigned32 get_start_date(utc_t * start_date) const;
    unsigned32 get_end_date(utc_t * start_date) const;
    unsigned32 get_key_start_date(utc_t * start_date) const;
    unsigned32 get_key_end_date(utc_t * start_date) const;
    unsigned32 get_usages(pkc_generic_key_usage_t * all_usages) const;
    unsigned32 get_user_usages(pkc_key_usage_t * user_usages) const;
    unsigned32 get_ca_usages(pkc_ca_key_usage_t * ca_usages) const;
    unsigned32 get_key_policies(pkc_key_policies_t * policies) const;
    unsigned32 get_constraints(pkc_constraints_t * constraints) const;
    unsigned32 get_certifier(x500name & name) const;
    unsigned32 get_revalidation_date(utc_t * start_date) const;
    unsigned32 get_revocation_date(utc_t * start_date) const;
    unsigned long get_key_id(void) const;
    unsigned long get_ca_key_id(void) const;
    char is_revoked(void) const;
    char is_trusted(void) const;
    char is_certified(void) const;
    char valid_at(utc_t * time) const;

    pkc_trusted_key_t & operator = (const pkc_trusted_key_t & o);
    unsigned32 copy (const pkc_trusted_key_t * src);

    char may_certify(
            const x500name & subject,
            unsigned long usage = PKC_CAKEY_USAGE_KEY_CERT_SIGN) const;

    char may_certify(
            const x500name & subject,
            const pkc_ca_key_usage_t & usage) const;
};

class pkc_revocation_t;
class pkc_pending_revocation_t;
class pkc_revocation_list_t {
friend class pkc_trust_list_t;
// A list of revoked certificates and their dates.
private:
    pkc_revocation_list_t & operator =
                      (const pkc_revocation_list_t & o);
public:
    pkc_revocation_t * first;
    pkc_revocation_t * last;
    pkc_pending_revocation_t * first_pending;
    pkc_pending_revocation_t * last_pending;
    unsigned32 get_revocation_date(const pkc_revocation_t & o,
                                   utc_t * revocationDate) const;
    unsigned32 get_revocation_date(const x500name & issuer,
                                   const asn_integer & serialNumber,
                                   utc_t * revocationDate) const;
    unsigned32 add_revocation(const pkc_revocation_t & o);
    unsigned32 add_revocation(const x500name & issuer,
                              const asn_integer & serialNumber,
```

```
                              utc_t * revocationDate);
    unsigned32 add_key(pkc_trust_list_t * trust_list,
                       const SubjectPublicKeyInfo & key,
                       const x500name & subject,
                       const utc_t & start_date,
                       const utc_t & end_date,
                       const pkc_ca_key_usage_t * usages = NULL);
    pkc_pending_revocation_t * find_pending(
                                   const x500name & issuer) const;
    virtual unsigned32 display(buffer_t & buffer) const;
    pkc_revocation_list_t(void);
    virtual ~pkc_revocation_list_t();
    void empty(void);
};

class pkc_pending_revocation_t {
friend class pkc_revocation_list_t;
    pkc_revocation_list_t * root;
    pkc_pending_revocation_t * prev;
    pkc_pending_revocation_t * next;
public:
    SignedCertificateList crl;
    virtual unsigned32 display(buffer_t & buffer) const;
    pkc_pending_revocation_t(const SignedCertificateList & crl,
                             pkc_revocation_list_t * the_root = NULL);
    pkc_pending_revocation_t & operator =
                                   (const pkc_pending_revocation_t & o);
    virtual void unlink(void);
    virtual ~pkc_pending_revocation_t();
};

class pkc_trusted_key_list_element_t {
private:
    pkc_trusted_key_list_t * root;
public:
    pkc_trusted_key_t * key;
    pkc_trusted_key_list_element_t * next;
    pkc_trusted_key_list_element_t * prev;
    virtual unsigned32 display(buffer_t & buffer) const;
    pkc_trusted_key_list_element_t(pkc_trusted_key_list_t * theRoot);
    virtual ~pkc_trusted_key_list_element_t();
    virtual void unlink(void);
};

class pkc_trusted_key_list_t {
public:
    pkc_trusted_key_list_element_t * first;
    pkc_trusted_key_list_element_t * last;
public:
    virtual unsigned32 display(buffer_t & buffer) const;
    pkc_trusted_key_list_t(void);
    void empty(void);
    virtual ~pkc_trusted_key_list_t(void);
};
```

```
class pkc_revocation_t {
friend class pkc_revocation_list_t;
    pkc_revocation_list_t * root;
    pkc_revocation_t * prev;
    pkc_revocation_t * next;
public:
    x500name certIssuer;
    asn_integer certSerialNumber;
    utc_t certRevocationDate;

    virtual unsigned32 display(buffer_t & buffer) const;
    pkc_revocation_t(const x500name & issuer,
                     const asn_integer & serialNumber,
                     utc_t revocationDate,
                     pkc_revocation_list_t * the_root = NULL);
    pkc_revocation_t & operator = (const pkc_revocation_t & o);

    virtual void unlink(void);
    virtual ~pkc_revocation_t();
};

class pkc_crl_seen_date_t;
class pkc_crl_seen_date_list_t {
friend class pkc_trust_list_t;
// A list of "CRL-seen" dates.
private:
    pkc_crl_seen_date_list_t & operator =
                      (const pkc_crl_seen_date_list_t & o);
protected:
    pkc_crl_seen_date_t * find(const x500name & issuer) const;
public:
    pkc_crl_seen_date_t * first;
    pkc_crl_seen_date_t * last;
    unsigned32 get_last_crl_seen_date(const x500name & issuer,
                                      utc_t * crlSeenDate) const;
    unsigned32 get_next_crl_expected_date(const x500name & issuer,
                                          utc_t * crlExpectedDate) const;
    unsigned32 add_crl(const x500name & issuer,
                       const utc_t & crlSeenDate,
                       const utc_t & crlExpectedDate);
    virtual unsigned32 display(buffer_t & buffer) const;
    pkc_crl_seen_date_list_t(void);
    virtual ~pkc_crl_seen_date_list_t();
    void empty(void);
};

class pkc_crl_seen_date_t {
friend class pkc_crl_seen_date_list_t;
    pkc_crl_seen_date_list_t * root;
    pkc_crl_seen_date_t * prev;
    pkc_crl_seen_date_t * next;
public:
    x500name crlIssuer;
    utc_t crlDate;
    utc_t crlNextDate;
```

```
    virtual unsigned32 display(buffer_t & buffer) const;
    pkc_crl_seen_date_t(const x500name & issuer,
                        const utc_t & seenDate,
                        const utc_t & nextDate,
                        pkc_crl_seen_date_list_t * the_root = NULL);
    pkc_crl_seen_date_t & operator = (const pkc_crl_seen_date_t & o);

    virtual void unlink(void);
    virtual ~pkc_crl_seen_date_t();
};

class pkc_trust_list_t {
friend class pkc_trust_list_element_t;
public:

    pkc_revocation_list_t revocation_list;  // List of revocations
    pkc_crl_seen_date_list_t revalidation_list;  // List of revalidations

    pkc_trust_list_element_t * first;
    pkc_trust_list_element_t * last;
    pkc_trust_list_t(void);
    virtual unsigned32 display(buffer_t & buffer) const;
    void empty(void);
    virtual ~pkc_trust_list_t();
    pkc_trust_list_t & operator = (const pkc_trust_list_t & o);
    unsigned32 fixup_links(void);
    unsigned32 find_certified_key(
                          const x500name & certifier,
                          const asn_integer & certifying_serial_no,
                          pkc_trust_list_element_t ** key);
// Returns the first key entry that was created from the specified
// certificate.  Call find_next_certified_key() to return the next
// such key.

    unsigned32 find_next_certified_key(
                           pkc_trust_list_element_t ** key);
// Returns the next key entry that was created from the same
// certificate as the current entry.  "key" is both an input and
// an output.

    unsigned32 find_certified_key_by_id(unsigned long ca_key_id,
                                        pkc_trust_list_element_t ** key);
// Returns the first key entry that was certified by the specified key-id
// Call find_next_certified_key_by_id() to return the next such key.

    unsigned32 find_next_certified_key_by_id(
                                 pkc_trust_list_element_t ** key);
// Returns the next key entry that was certified by the same key as
// the current entry.  "key" is both an input and an output.

    unsigned32 process_crl(const SignedCertificateList *crl);
// Applies a previously verifed CRL to the trust-list.
};
```

```
class pkc_trust_list_element_t : public pkc_trusted_key_t {
friend class pkc_trust_list_t;
// A list of trusted keys.
private:
    pkc_trust_list_element_t & operator =
                 (const pkc_trust_list_element_t & o) {return *this;};
private:
    pkc_trust_list_t * root;
public:
    pkc_trust_list_element_t * next;
    pkc_trust_list_element_t * prev;
    void unlink(void);
    pkc_trust_list_element_t(pkc_trust_list_t & the_root);
    pkc_trust_list_element_t(pkc_trust_list_t & the_root,
                             utc_t startDate,
                             utc_t endDate,
                             pkc_ca_key_usage_t caUsages,
                             pkc_key_usage_t userUsages,
                             pkc_key_policies_t keyPolicies,
                             pkc_constraints_t keyConstraints);
    virtual ~pkc_trust_list_element_t();
    unsigned32 apply_revocation(utc_t * revocation_date);
// Apply a revocation to this key, starting at the specified date.
// If revocation_date is NULL, the key is completely revoked.
// This key, and all keys dependent on it will be revoked.

    unsigned32 get_certifier_key(const pkc_trusted_key_t ** ca) const;
    unsigned32 get_certifier(const pkc_trust_list_element_t ** ca)
        const;

};

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//
// The trustlist is the fundamental object within the certificate
// manipulation facility.  A trustlist is a set of keys which are
// trusted, plus a list of revoked certificate serial numbers.  Keys
// are inserted into a trustlist either directly (via the
// pkc_add_trusted_key() function) or indirectly (via the
// pkc_check_cert_against_trustlist() function).  The latter routine
// will only add keys if the certificate signature can be verified by a
// key already in the trustlist, and if the certificate has not been
// revoked.
//
// Currently, trustlists are relatively static objects - once a key is
// inserted, its trust properties don't change.  If, for example, a
// key is added that is capable of extending the trust in another key
// within the list, the second key isn't automatically updated.
// Auto-update is a matter for the next version.
//
// The way that a policy module is expected to use this stuff is to
// first create an initial trustlist containing the directly trusted
// keys, that is, the start point(s) of all valid trust chains.
// Typically, this set of keys will be used for multiple certificate
// chain evaluations;  If the policy wishes to impose additional path
```

```
// constraints over the constraints expressed within the certificates,
// it must maintain a master copy of the original trustlist and clone
// it to create a modifiable version for each chain the policy module
// wants to verify.  After verification of a candidate chain, the
// cloned trustlist must be discarded so that the next trial
// verification starts from a known state.
//
// Using the initial trustlist as a starting point, the policy module
// retrieves a chain of certificates and adds them to the trustlist
// one by one, starting with the certificate(s) closest to the start
// point(s).  Multiple chains may be evaluated simultaneously using a
// single trustlist for policies that do not wish to impose additional
// constraints on the trust chain;  however the policy module must
// ensure that for each trust-chain, certificates are added in the
// correct order.  A future auto-update enhancement may lift this
// requirement.
//
// ----------------------------------------------------------------------

unsigned32 pkc_add_trusted_key(
                         pkc_trust_list_t * trust_list,  /* mbg: */
                         const pkc_trusted_key_t & key);
// Adds the specified key to the specified trustlist.

unsigned32 pkc_lookup_keys_in_trustlist(
                         pkc_trust_list_t * trust_list,
                         const pkc_trusted_key_t ** key,
                         size_t * key_count,
                         const x500name & owner,
                         utc_t * key_time = NULL,
                         const pkc_generic_key_usage_t * usages = NULL);
// Searches the specified trustlist for keys owned by the specified
// principal at the specified time for the specified usage.  The keys
// are returned in an array of pointers to pkc_trusted_key_t objects,
// which is allocated on the heap; the pointers point to elements within
// the trust-list, thus the caller should copy into allocatead storage
// if they are expected to remain valid after the deletion of the
// trust-list.

unsigned32 pkc_lookup_key_in_trustlist(
                         pkc_trust_list_t * trust_list,
                         const pkc_trusted_key_t ** key,
                         unsigned long key_id);
// Searches the specified trustlist for the specified key.

unsigned32 pkc_lookup_element_in_trustlist(
                         pkc_trust_list_t * trust_list,
                         const pkc_trust_list_element_t ** key,
                         unsigned long key_id);
// Searches the specified trustlist for the specified key.

unsigned32 pkc_check_cert_against_trustlist(
                         pkc_trust_list_t * trust_list,
                         const Certificate * cert,
                         int revoked_certs_permitted);
```

```
// Checks the specified certificate against the specified list of
// trusted keys. If the certificate is valid and can be verified
// from the trustlist, its content is added to the trustlist.
// revoked_certs_permitted is a flag that says whether revoked
// certificates should still be trusted for dates prior to their
// revocation date.

unsigned32 pkc_revoke_certificate(pkc_trust_list_t * trust_list,
                                  const x500name & issued_by,
                                  const asn_integer & serial_no,
                                  utc_t * invalidate_from = NULL);
// Applies the specified revocation to the specified trustlist
// (i.e. revokes a key and all dependent keys).  If invalidate_from
// is NULL, the key is completely revoked; if a valid UTC-time is
// provided, the key is revoked from that time on.  The revocation
// is stored within the trustlist, and any subsequent attempts to
// add the certificate will be rejected.

unsigned32 pkc_revoke_certificates(
                    pkc_trust_list_t * trust_list,
                    const SignedCertificateList * crl);
// Processes a certificate revocation list.  The revocations within
// the CRL are applied to the specified trustlist (i.e. revoking
// a set of keys and all dependent keys).  The revocations are stored
// within the trustlist, and any subsequent attempts to add a revoked
// certificate will be rejected.  In addition, the CRL as a whole
// serves to revalidate certificates stored within the sphere of
// responsibility specified by <revalidate>.

unsigned32 pkc_prune_trustlist(pkc_trust_list_t * trust_list,
                               const utc_t * prune_before = NULL);
// Prunes keys that expired before the stated time from a trustlist.
// If prune_before is NULL, all keys that have expired are pruned.

unsigned32 pkc_display_trustlist(
                         const pkc_trust_list_t * input_trust_list);
// Prints the contents of the trustlist to stdout.  Intended for use
// when debugging policy modules.

#endif
```

9. REMOTE INTERFACES

No new interfaces will be provided.

10. MANAGEMENT INTERFACES

No new management interfaces will be provided.

11. RESTRICTIONS AND LIMITATIONS

TBS

12. OTHER COMPONENT DEPENDENCIES

The registry-lookup policy module will retrieve public-keys directly from the registry. Depending on the public-key login component design, this may require APIs provided by that component (or it may use standard ERA lookup).

The services will make use of the existing XDS name-service interface for certificate and CRL retrieval from either X.500 or CDS.

13. COMPATIBILITY

N/A

14. STANDARDS

The ISO 9594-8/X.509 certificate format will be supported, including the 1992 amendment.

The PKCS #6 and X9.xx-1994 extended certificate formats may also be supported.

15. OPEN ISSUES

Licensing terms of RSA algorithm and other crypto implementations.

Export implications of use of RSADSI BSAFE within DCE for non-US source licensees.

Storage and retrieval of CRLs.

Provision of dummy CA for test purposes, as opposed to set of hand-crafted certificates.

REFERENCES

[ISO 9594-8] ISO, ‘‘Information Technology — Open Systems Interconnection — The Directory — Part 8: Authentication Framework’’, 1990.

[ISO 9594-8/DAM 1] ISO, ‘‘Information Technology — Open Systems Interconnection — The Directory — Part 8: Authentication Framework, AMENDMENT 1: Access control’’, 1992.

[PKCS #6] RSA Laboratories, ‘‘Public-Key Cryptography Standards #6: Extended Certificate Syntax Standard’’, November 1, 1993.

[ANSI X9.xx-1994] ANSI, ‘‘Public Key Cryptography for the Financial Services Industry: Extensions to Public Key Certificates— Proposed Working Draft’’, November 1994.

AUTHORS’ ADDRESSES

John Wray
Digital Equipment Corporation
550 King Street
Littleton, MA 01460
USA
Internet email: wray@tuxedo.enet.dec.com
Telephone: +1-508-486-5210

Mahesh Girkar
Digital Equipment Corporation
550 King Street
Littleton, MA 01460
USA
Internet email: girkar@tuxedo.enet.dec.com
Telephone: +1-508-486-6132