Open Software Foundation
Request For Comments: 80.0
January 1995
J. Wray (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 (at least) DCE hierarchical-cells and a PEM-like model.

1.1. Changes Since Last Publication

This is the first issue of this document (as an RFC). It's still incomplete. Note in particular that many of the details of this service will depend on the results on ongoing license negotiations (see the Open Issues section below).

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. Caching of certificates will be supported in a per-process cache. The API design will allow for system-wide certificate caching, although provision of such a facility is not committed. The use of CRLs will be supported, should trust policies require them.

3. GOALS AND NON-GOALS

The goal is to provide a name to public-key translation service, using an application-chosen trust policy. The service will simply return public keys; No cryptographic services will be provided, other than (subject to licensing) certificate-chain verification.

It is highly desirable to provide sufficient support to enable additional signature algorithms and trust models to be added at a later date.

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.

5. REQUIREMENTS

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

Support of the RSA algorithm for certificate signatures.

6. FUNCTIONAL DEFINITION

6.1. Low-Level Certificate API

This API is intended for use by a policy module. Policy modules must be registered with the certification API before they can be invoked. The low-level certificate API allows certificates to be read into the cache, and performs certificate parsing functions. This API also provides certificate cache management functions.

6.2. Policy and Crypto Module Registration

These APIs are used to register policy modules and cryptographic signature verification modules, and to determine which policies and algorithms are already known. Direct `secd` lookup, DASS and PEM-like policies will be provided, as will the following signature algorithms: `md2WithRSA`, `md5WithRSA`.

Additional policies and algorithms may be supplied.

6.3. High-level public-key API

This API provides a name to public-key retrieval service. The application indicates a desired a policy or policies, a target principal name, and possibly additional key-discrimination data. The service returns the key of the target principal.

7. USER INTERFACES

No user interfaces will be provided.

8. DATA STRUCTURES AND APPPLICATION PROGRAMMING INTERFACES

8.1. Low-Level Certificate API

The following data-types and routines make up the low-level certificate API:

```
/* A data-type to hold a handle to a parsed certificate            */
typedef struct {
    void * cert_handle;
} pkc_certificate_handle_t;

/* A data-type to hold a handle to a set of parsed certificates    */
typedef struct {
    void * cert_set_handle;
} pkc_certificate_set_handle_t;

/* A data-type to hold a certificate-cache handle                  */
typedef struct {
```

```
    void * cache_handle;
} pkc_certificate_cache_handle_t;

/* A data-type to hold a certificate-cache session handle          */
typedef struct {
    void * session_handle;
} pkc_cache_session_handle_t;

/* A data-type that describes the current state of a               */
/* certificate-cache.                                              */
typedef struct {
    OM_uint32 version;
    void * cache_id;
    int backed;
    int modified;
} pkc_certificate_cache_details_t;

/* A data-type containing flag-bits that modify the certificate    */
/* retrieval algorithm.                                            */
typedef unsigned32 pkc_cert_lookup_flags_t;

/* A backing-store.  A policy-module that wishes to create         */
/* persistent certificate-caches must register a backing-store     */
/* handler with the certification API.  A backing-store contains   */
/* routines for reading and writing data from persistent storage.  */
/* It is registered with the certification API using the           */
/* following data-structure: The cache_id supplied to these        */
/* routines is a caller-defined object specified when the cache    */
/* object is created.  It can be used to hold a name for the       */
/* cache to allow the backing-store module to choose an            */
/* appropriate file-name for the persistent storage, for example.  */

typedef struct {
    /* Open existing backing-store for reading.                     */
    unsigned32 open_backing_store(void * cache_id,
                                  void ** stream_id);

    /* Create new backing-store for writing.                        */
    unsigned32 create_backing_store(void * cache_id,
                                    void ** stream_id);

    /* Close an open backing-store.                                 */
    unsigned32 close_backing_store(void * stream_id);

    /* Write a block of data into a backing-store opened for        */
    /* writing.                                                     */
    unsigned32 write_data(void * stream_id,
                          unsigned char * data_ptr,
                          size_t length);

    /* Read a block of data from a backing-store opened for         */
    /* reading.                                                     */
    unsigned32 read_data(void * stream_id,
                         unsigned char * data_ptr,
                         size_t length);
```

```
} pkc_certificate_backing_store_t;

/****************************************************************/
/* Certificate-cache management routines.  All certificates must   */
/* be stored in the cache before use.  The application is          */
/* responsible for creating the cache, and deleting it when        */
/* finished.  A cache may be saved to disk, to allow               */
/* communication between images, or to preserve cache data across  */
/* multiple image activations;  Policy-modules are responsible     */
/* for controlling such disk operation.                            */
/****************************************************************/

/* Create an empty certificate-cache.                              */
unsigned32 pkc_create_cert_cache(
                void * cache_id,
                pkc_certificate_cache_handle_t * cache);

/* Attach a backing-store module to a certificate-cache.           */
unsigned32 pkc_attach_backing_store(
                pkc_certificate_cache_handle_t * cache,
                const pkc_certificate_backing_store_t *
                                                      backing_store);

/* Load certificate-cache data from backup.                        */
unsigned32 pkc_restore_cert_cache_data(
                pkc_certificate_cache_handle_t * cache);

/* Save certificate-cache to disk.                                 */
unsigned32 pkc_backup_cert_cache_data(
                const pkc_certificate_cache_handle_t * cache);

/* Delete a certificate-cache.                                     */
unsigned32 pkc_delete_cert_cache(
                pkc_certificate_cache_handle_t * cache);

/* Perform a global operation on a certificate-cache.              */
unsigned32 pkc_global_operation(
                pkc_certificate_cache_handle_t * cache,
                unsigned32 op_rtn(pkc_certificate_handle_t * cert));

/* Interrogate a certificate-cache.  The returned                  */
/* pkc_certificate_cache_details_t structure will have been        */
/* allocated with the pkc_alloc routine defined in pkc_base.h;     */
/* should be freed by the caller using pkc_free().                 */
unsigned32 pkc_inquire_cert_cache(
                const pkc_certificate_cache_handle_t * cache,
                pkc_certificate_cache_details_t ** details);

/****************************************************************/
/* Certificate retrieval routines.  All certificates are stored    */
/* in the directory service as attributes of the subject's entry.  */
/* Before manipulating a certificate, it must be read into a       */
/* certificate-cache using one of the pkc_read_certificates        */
/* routines.  Certificates in the cache are either permanent       */
/* residents, or transient certificates.  Transient certificates   */
```

```
/* may be flushed from the cache at the end of the current cache   */
/* session; permanent certificates are never automatically         */
/* flushed.                                                         */
/*******************************************************************/

/* Start a cache session.                                           */
unsigned32 pkc_open_cert_cache_session(
                    pkc_certificate_cache_handle_t * cache,
                    pkc_cache_session_handle_t * session);

/* Finish a cache session.                                          */
unsigned32 pkc_close_cert_cache_session(
                    pkc_cache_session_handle_t * session);

/* For policy-modules that wish to interact with the name-service  */
/* themselves, the following routine allows an unparsed            */
/* certificate to be directly injected into the cache.             */
unsigned32 pkc_insert_cert(
                    pkc_cache_session_handle_t * session,
                    unsigned char * certificate_data,
                    size_t certificate_length);

/* For policy-modules that wish to interact with the name-service  */
/* themselves, the following routine allows an unparsed            */
/* certificate to be directly injected into the cache, and marked  */
/* as a permanent resident.                                         */
unsigned32 pkc_insert_cert_perm(
                    pkc_certificate_cache_handle_t * cache,
                    unsigned char * certificate_data,
                    size_t certificate_length);

/* Read transient certificates for the specified principal from    */
/* the namespace into the certificate cache.                        */
unsigned32 pkc_read_certs_by_name(
                    pkc_cache_session_handle_t * session,
                    pkc_principal_name_t * principal_name,
                    pkc_cert_lookup_flags_t flags);

/* Read permanent certificates for the specified principal from    */
/* the namespace into the certificate cache.                        */
unsigned32 pkc_read_certs_by_name_perm(
                    pkc_certificate_cache_handle_t * cache,
                    pkc_principal_name_t * principal_name,
                    pkc_cert_lookup_flags_t flags);

/* Retrieve certificates from a certificate-cache.  The cache may  */
/* be searched on any field or fields.  Any field may be           */
/* wildcarded by passing NULL as the value.                         */
unsigned32 pkc_lookup_certs(
                    pkc_certificate_cache_handle_t * cache,
                    pkc_principal_name_t * subject_name,
                    pkc_principal_name_t * issuer_name,
                    pkc_certificate_set_handle_t * cert_set);

/* Retrieve the first certificate from a set of certificates.      */
```

```
unsigned32 pkc_count_certs(pkc_certificate_set_handle_t * cert_set,
                           size_t * cert_count);

/* Retrieve the first certificate from a set of certificates.      */
unsigned32 pkc_first_cert(pkc_certificate_set_handle_t * cert_set,
                          pkc_certificate_handle_t * cert);

/* Retrieve the next certificate from a set of certificates.       */
unsigned32 pkc_next_cert(pkc_certificate_set_handle_t * cert_set,
                         pkc_certificate_handle_t * cert);

/* Delete a set of certificates.  Only the set will be released;   */
/* the individual certificates will remain in the cache.           */
unsigned32 pkc_free_cert_set(pkc_certificate_set_handle_t *
                                                           cert_set);

/*****************************************************************/
/* Certificate interrogation routines.  To avoid requiring that    */
/* policy-modules must understand ASN.1, certificates are parsed   */
/* by the certification API, and their contents is made available  */
/* to policy-modules via interrogation functions.  As always, any  */
/* storage returned to the policy-module via these routines must   */
/* be released with the pkc_free() routine.  The following         */
/* functions are provided:                                         */
/*****************************************************************/

unsigned32 pkc_check_signature(pkc_certificate_handle_t certificate,
                               unsigned char * public_key,
                               size_t public_key_length);

unsigned32 pkc_inq_cert_subject(pkc_certificate_handle_t certificate,
                                pkc_principal_name_t * subject_name);

unsigned32 pkc_inq_cert_issuer(pkc_certificate_handle_t certificate,
                               pkc_principal_name_t * issuer_name);

unsigned32 pkc_inq_cert_version(pkc_certificate_handle_t certificate,
                                int * version);

unsigned32 pkc_inq_cert_serial_number(
                                pkc_certificate_handle_t certificate,
                                unsigned char **serial_number_buffer,
                                size_t * length);

unsigned32 pkc_inq_cert_start_time(
                                pkc_certificate_handle_t certificate,
                                utc_t * time);

unsigned32 pkc_inq_cert_end_time(
                                pkc_certificate_handle_t certificate,
                                utc_t * time);

unsigned32 pkc_inq_cert_signature_alg(
                                pkc_certificate_handle_t certificate,
                                OM_object_identifier * alg_id,
```

```
                                unsigned char ** parameters,
                                size_t * parameter_length);

unsigned32 pkc_inq_cert_subject_key(
                                pkc_certificate_handle_t certificate,
                                OM_object_identifier * alg_id,
                                unsigned char ** public_key,
                                size_t * public_key_bitlength);

unsigned32 pkc_inq_cert_subject_uuid(
                                pkc_certificate_handle_t certificate,
                                unsigned char ** subject_uuid,
                                size_t * subject_uuid_bitlength);

unsigned32 pkc_inq_cert_issuer_uuid(
                                pkc_certificate_handle_t certificate,
                                unsigned char ** issuer_uuid,
                                size_t * issuer_uuid_bitlength);

/*
 * Additional certificate interrogation functions will be added to
 * support X9.xx-1994 and PKCS #6 certificates.
 */
```

8.2. Crypto Module Registration API

The following data-types and routines make up the crypto-module registration API:

```
/* This datatype is used to scroll through the list of registered  */
/* signature algorithms.                                            */

typedef struct {
    void * data;
} pkc_alg_cursor_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             */
    OM_object_identifier alg_id;
    /* Object identifier that names the algorithm - the OID that   */
    /* appears in certificates signed by the algorithm.            */

    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>), the certification API will invoke the 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,
                      unsigned char * data,
                      size_t data_length,
                      unsigned char * public_key,
                      size_t public_key_length,
                      unsigned char * parameters,
                      size_t parameter_length,
                      unsigned char * signature,
                      size_t signature_length);
/* A routine that checks the supplied signature against the    */
/* supplied data.  The <parameters> are as specified in the    */
/* certificate; the <data> is the entire certificateInfo, and  */
/* the <public_key> is the public-key of the certificate       */
/* issuer.  Both <parameters> and <public_key> may be encoded  */
/* in BER, and the helper functions below may be used by crypto*/
/* modules to aid in parsing BER, if desired.  If the algorithm*/
/* does not require parameters, certificates signed with the   */
/* algorithm may contain either an ASN.1 NULL value, or simply */
/* omit the certificate parameters field; In both cases, void  */
/* shall be passed as the <parameters> argument to the verify  */
/* function.  The <parameter_length> argument will contain 0   */
/* if the value was omitted from the certificate, or a non-zero*/
/* value if an ASN.1 NULL value was supplied, although it is   */
/* expected that a verify function should not need to          */
/* distinguish between these two possibilities.                */
/* 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,
                    unsigned char * data,
                    size_t data_length,
                    unsigned char * private_key,
                    size_t private_key_length,
                    unsigned char ** parameters,
                    size_t * parameter_length,
```

```
                        unsigned char ** signature,
                        size_t * signature_length);
    /* A routine that calculates a signature over the supplied data*/
    /* using the specified key.  The <key> parameter will be       */
    /* appropriate for the algorithm (presumably encoded in BER).  */
    /* The <parameters> and <signature> fields should be returned  */
    /* by the function; storage allocation should be performed by  */
    /* calling the pkc_alloc() & pkc_free() functions defined      */
    /* above.                                                      */
} pkc_signature_algorithm_t;

unsigned32 pkc_register_signature_alg(pkc_signature_algorithm_t *
                                                                alg);

unsigned32 pkc_initialize_algorithm_cursor(pkc_alg_cursor_t *
                                                             cursor);

unsigned32 pkc_advance_algorithm_cursor(pkc_alg_cursor_t * cursor);

unsigned32 pkc_inquire_algorithm_details(
                            pkc_alg_cursor_t * cursor,
                            pkc_signature_algorithm_t ** details);

unsigned32 pkc_delete_algorithm_cursor(pkc_alg_cursor_t * cursor);
```

8.3. Policy-Module Registration API

The following data-types and routines make up the policy-module registration API:

```
/*******************************************************************/
/*                                                                 */
/* A policy module is registered much like a crypto implementation */
/* is registered: The application provides a pkc_policy_t data     */
/* structure, along with an OID identifying the policy.            */
/*                                                                 */
/*******************************************************************/

typedef struct {
    void * data;
} pkc_policy_cursor_t;

/* This data-type is used to register a new policy-module with the */
/* certification API.  It describes a policy algorithm, and        */
/* provides entry-points to its key retrieval 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              */
    OM_object_identifier policy_id;
    /* Object identifier that names the policy.                    */

    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_alloc()         */
    /* function defined in pkc_base.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> function; other functions are optional.*/

    unsigned32 (*open) (void** context);
    unsigned32 (*close) (void** context);
    /* Before invoking any policy routines (eg <retrieve>), the    */
    /* certification API will invoke the 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 <retrieve> 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 policy  */
    /* module.  If the <open> routine stores any state in the      */
    /* context parameter, the <close> routine should free this     */
    /* storage.                                                    */

    unsigned32 (*retrieve) (void ** context,
                            pkc_principal_name_t target,
                            unsigned char ** public_key,
                            size_t * public_key_length);
    /* A routine that should return the public key for the         */
    /* specified principal.                                        */
} pkc_policy_t;

unsigned32 pkc_register_policy(pkc_policy_t * alg);

unsigned32 pkc_initialize_policy_cursor(pkc_policy_cursor_t *
                                                            cursor);

unsigned32 pkc_advance_policy_cursor(pkc_policy_cursor_t * cursor);

unsigned32 pkc_inquire_policy_details(pkc_policy_cursor_t * cursor,
                                      pkc_policy_t ** details);

unsigned32 pkc_delete_policy_cursor(pkc_policy_cursor_t * cursor);
```

8.4. High-Level Public-Key API

The following data-types and routines make up the high-level public-key API:

<TBD>

9. REMOTE INTERFACES

No new interfaces will be provided.

10. MANAGEMENT INTERFACES

Unless the system-wide certificate cache facility is implemented, there will be no management interfaces.

11. RESTRICTIONS AND LIMITATIONS

<TBD>

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 existing name-service remote interfaces for certificate and CRL retrieval.

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 the use of RSADSI BSAFE within DCE for non-US source licensees. I believe that we can simply remove the BSAFE library from the framework, and require overseas source licencees to provide their own cryptographic implementation. BSAFE itself provides callable confidentiality services, and is probably not exportable as a result.

I believe that the services and APIs described herein will not be subject to control under US export law, since they contain no provision for confidentiality. However, this is something that must be conclusively determined.

Name-space policy for certificate storage within CDS.

Storage and retrieval of CRLs.

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

Is a CDS name sufficient to identify a particular public-key, or should the APIs support multiple keys per name, requiring an additional key-selector to be associated with each name in the API specifications?

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.

AUTHOR'S 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