The Open Group **R. Salz (OpenGroup)**
Request For Comments: 100.0
January, 1997

DCE AND FORTEZZA

1. INTRODUCTION

This document discusses how to integrate the Fortezza card with DCE. We propose a two-phase approach: first, use of the card as an instance of the DCE 1.2.2 public-key login technology; and second, use of the card as the cryptographic engine of a full PKI.

In order for this document to be self-contained, we start with a introduction to the various aspects of DCE login. We then introduce the card.

We then discuss how the Fortezza card might be integrated into the DCE login suite. The next section briefly touches on more complete integration of the card, as part of full DCE support for ‘‘multi-crypto.’’ Finally, we list some of the issues that will need to be resolved; this last section is probably the most important area of this document.

2. Background Information

2.1. Introduction to DCE login

Logging in to DCE, also known as obtaining network credentials, is achieved by obtaining a ticket-granting-ticket, TGT, to the DCE privilege service. This is done via two API's: `sec_login_setup_identity`, and `sec_login_validate_identity`. Historically, the first requests an encrypted TGT from the security service, and the second uses the provided user password to generate a key and decrypt the TGT, allowing it to be used in the process of making all future authenticated RPC's. (The pre-authentication protocols added in DCE 1.1 shift the work a bit, so that only the username is validated in the first part, and a strongly-encrypted request proving knowledge of the password is made in the second part.)

In DCE 1.2.2, the APIs have been preserved, but the implementation has undergone significant changes to allow public-key based cryptography to be used for login. See [RFC 68] for more details.

In the protocol definitions below, the common notation `{xxx}Sc` means that `xxx` is signed by `c`, or encrypted if `Pc` is used.

```
Client-->KDC:  {TrustedRoots, Checksum, Time, Nonce, KDCName}Sc
```

That is, the client sends an optional list of the ‘‘root CA's’’ that it trusts, a checksum and a nonce (plaintext confounder), its value of the current time, and the name of the KDC that it is contacting. All of this is signed by the client's public key.

Once the KDC validates the signature, checks that the time is reasonably current, and that it is the server named by the KDCName parameter, the DCE security service sends the following reply:

```
KDC-->Client:  Ckdc, {{Kreply,Nc}Skdc}Kreply, {Kreply}Pc
```

The client receives three items of data: the KDC's certificate, and two encrypted items. The last item is a random DES-style key, encrypted in the client's private key. Once the client decrypts the DES key, it can decrypt the second item. this contains a signed pair of the original nonce and a third instance of the DES key. The signed data is encrypted to avoid tampering. The tamper-proof signing proves to the client that the proper KDC replied to the proper request of the client.

In the DCE 1.2.2 reference implementation, the certificate lists are always empty. The local-host daemon, `dced` [RFC 47], has been augmented to cache KDC certificates obtained through new management interfaces.

A new internal API, `sec_psm` has also been provided with DCE 1.2.2. It supports the common cryptographic abstractions (sign, seal, and their inverses) as well as a mix of key management functions. Most initialization routines accept a "domain" or "mechanism" identifier, primarily so that DCE licensees can provide their own key storage facilities. While this primary focus is different, `sec_psm` does resemble a generic crypto API, such as Microsoft's CAPI [CAPI].

Public-key login in DCE can be made mandatory by setting the well-known `pre_auth_req` ERA value for a principal. This is currently an ordered list, ranging from no pre-authentication, to public-key login. Unfortunately, it is an integer with no spaces between the values. We are not convinced that a strict ordering makes sense, and will have to study this issue – at least to add a value for Fortezza login – as implementation begins.

Another ERA, `DCEPKPRivateKeyStorage`, specifies where a principal's private key is stored. There is no value for cryptographic token; using the "file" value may suffice.

2.2. Introduction to Fortezza

The Fortezza card is a cryptographic (hardware) device known as a "smart card.". It's tamperproof. It is a PCMCIA card, designed to plug into a standard slot supported on many desktop and portable computer systems; SCSI card readers are also widely available.

Fortezza supports multiple operating modes divided into the major categories of initialization, when security personnel must program the PINs and other data into the card; user, when the card is capable of standard use. This document does not address any mode other than the Standby and Ready states, for normal use [Fapi].

The current version of the Fortezza card supports a classified secret-key algorithm known as SKIPJACK. It also provides a classified key-exchange algorithm, KEA, and a public Digital Signature Standard, DSS. The KEA and DSS public keys are available through the card's X.509v1 certificate. X.509v3 support is currently available, and there will later be separate X.509v3 certificates for the DSS and KEA keys. Finally, the card provides some time and random-number generation.

The standard API to the Fortezza card is known the as Fortezza Cryptologic Interface library, or CI library [Fint]. It is expected that this library will either be provided on our development platforms, or that we will be able to port it with a modicum of effort.

3. Phase One: Fortezza Login

Most of the `sec_psm` and lower-level code is implemented as a `switch` statements, dispatching to domain-specific code. The primary work of supporting the Fortezza card is adding new values to the switches and adding the requisite CI library calls.

Within the public-key storage API, `sec_pubkey`, we will add the following new mechanism: `sec_pk_domain_fortezza`, indicating that the certificate is stored in a Fortezza card. Note that we are using the card's name as the domain, and we will follow this practice throughout. Future cards will support multiple cryptographic algorithms, so we will also define "aliases" making specific that this is the `*default*` `mechanism used by the card or defined by DCE. This use parallels the rpc_c_ns_syntax_dce and rpc_c_ns_syntax_default constants used in the DCE name service interface.`

`Within the public-key base interface, sec_pk_base, we will define a new domain, sec_pk_domain_fortezza_login. This defines the how the public-key technology is going to be used. We are defining a new domain because we have not yet decided how to implement the login protocol. A simple method is to follow the steps outlines in [Fapi] and do an authenticated key exchange to establish mutual authentication, and then encrypt a DES key using the card, and return both the key and the TGT. The other option is to transliterate the protocol, substituting SKIPJACK for RSA encryption, above. This decision will have be made during` implementation.

We will need to define a SKIPJACK Algorithm ID if someone else (such as Spyrus) has not already done so. For DCE's use, we will create a UUID for that ID.

We will need to extend the sec_psm handle so that it can store the CI library state. We will need to provide a once-only CI library initialization path.

Once these modifications are made, and the issues listed below addressed, we believe that allowing use of the Fortezza card to provide DCE login will be straightforward.

4. Phase Two: Fortezza Cryptography

In future releases of DCE, we want to support multiple cryptographic mechanisms. Right now, DCE is limited to using DES for signing and sealing both its authorization and its application-level data. We are evaluating how to support pluggable cryptography, and we feel it important that the Fortezza card be supportable in this environment.

We also want to use our implementation experience here to help "push back" on general multi-crypto designs; it is important that the Fortezza card not require special treatment, in API's or protocols.

This section is primarily a place-holder to capture this desire; see also the open issues below.

5. Open Issues

5.1. Phase One

(a) The CI library is not multi-threaded. Every call to the CI library must be protected by a mutex. We propose to do this by writing a thin veneer over the CI API. In addition to ensuring that the mutex calls are always performed, this layer will provide a more "DCE-like" API. For example:

```
int
CI_Initialize(
    int *numSockets
);
```

Will be modified to return the status code as an output parameter, and also to return a handle to be used in all other calls, viz:

```
void
dce_fza_initialize(
    int *numSockets,
    dce_fza_handle_t *h,
    error_status_t *st
);
```

We will also map all CI error codes into the DCE status code space, thereby providing serviceability information. In order to support non-DCE Fortezza applications, we might have to provide a "CI library global mutex" and access to it.

(b) From a hardware point of view, the Fortezza card is a high-latency low-bandwidth device. Since the amounts of data being based are very small, the context-switch and memory-copy times will dominate. While this may not be a factor in desktop operating systems, it might be an issue in Unix[tm]-style multi-tasking systems. We will have to be aware of this, but currently have no plans to address it. This will be of particular concern of the security server.

(c) The Fortezza card has a limited amount of on-card memory; it is probably capable of storing only a dozen or so certificates. We expect that deleting certificates and loading new ones (that is, using the card as a cache) is high-overhead. This will affect the scalability of the security server. In Phase Two, clients may be limited by the number of servers they can work with concurrently.

(d) We are not sure of the best way to support long-running servers such as Time and Directory. It is reasonable to allow the DCE security service to be a special-case that requires the PIN to be entered by a user (probably the Site Security Officer or his designee) at boot-time. For other services, particularly application programs, we need an API and possibly storage facility not unlike the DCE `keytab` facility.

(e) A DCE host can be thought of as the longest-running DCE server of all. The host principal is used primarily for pre-authentication, which is presumably not needed here. It also provides the DCE credentials for legacy services (such as `sendmail`) that might "inadvertently" need DFS credentials. There are also services, such as DTS, that truly correspond to the "local root.". (It's also possible, using delegation and a custom server, to build location-based ACL's, such as a group containing all machines in a specific secure room.) Supporting this gets involved with host bootup procedures, and is likely to be very system-specific.

(f) Code containing or using the Fortezza library is export-controlled. We believe that by implementing our support as a module within the `sec_psm` layer our source code will be exportable; this is of great concern to The Open Group. The thin veneer, since it merely masks APIs, may itself be exportable.

(g) Multi-user (or multi-service) machines will have more than one card reader. The documentation does not explain how to specify a particular reader, to reliably know which one is free for use, and how to convey that information to the user.

(h) Is it necessary to support "duress login" mode?

(i) We will need to deal with the pragmatic issue of getting Fortezza cards, PCMCIA readers and drivers, on our development platforms. This might be an issue, since it would be too much overhead for "step zero" to read "first port DCE.". If the CI library is not available on our development platforms, we will have to port it or contract someone to do it for us.

5.2. Phase 2

Support for the Fortezza card might influence how we implement the "cryptographic provider" facilities in a multi-crypto DCE [APKI]. Obvious candidates to consider are the Microsoft CAPI, Netscape's CDSA, and the Open Group's GCS. We note in passing that Spyrus is doing a CAPI CSP for Fortezza.

Writing this document has brought forth a new issue in multi-crypto DCE: we need to handle the case where the crypto-mechanism is decided by policy or law, and should not be overridden by the user.

Is it a requirement to support full off-line use? The Fortezza documents allow for CRL's, validation, and hint toward future integration with a full PKI. We will need to track this as DCE and the Open Group progress with their PKI plans.

6. REFERENCES

[APKI] B. Blakley, et. al, *Architecture for a Public-Key Infrastructure*, IETF Internet Draft.

[BLAZE] Doing crypto with high-latency cards.

[CAPI] Microsoft Cryptographic API, *http://www.microsoft.com/intdev/security*, on the World-Wide Web.

[Fapi] National Security Agency, *Fortezza Application Implementors Guide*, Product Designation: MD4002101-1.52, March 5, 1996.

[Fint] National Security Agency, *Fortezza Cryptologic Interfrace Programmers Guide*, Product Designation: MD4000501-1.52, January 30, 1996.

[RFC 47] J. Bowe, D. Mackey, R. Salz, P. Wang, *DCED: The DCE Host Daemon -- Functional Specification*, April 1994.

[RFC 68] A. Anderson, S. Cuti, OSF-RFC 68.3 *DCE 1.2.2 Public Key Login*, In preparation.

[RFC 98] W. Tuvell, OSF-RFC 98.0 *Challenges Concerning Public-Key in DCE*, December, 1996.

7. AUTHOR'S ADDRESS

Rich Salz
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA

Internet email: r.salz@opengroup.org
Telephone: +1-617-621-7253