A Smart card Solution for Access Control and Trust Management for Nomadic Users ⋆

Daniel Díaz Sánchez, Andrés Marín Lopez, Florina Almenárez Mendoza

Telematic Engineering Department, Carlos III University of Madrid
Avda. Universidad, 30, 28911 Leganés (Madrid), Spain
{dds, amarin, florina}@it.uc3m.es

Abstract. Increasing efforts are placed on security solutions for nomadic users. Solutions based on smart cards offer physical and logical portability, robustness, low cost, and high security. Nevertheless, such solutions concentrate only on offering the cryptographical capabilities of the smart card, together with key and user certificate storage. Advanced trust management and access control are not addressed. In this article, we propose a scheme to include trust management and attribute certificates for authorization in two widely used cryptographic APIs: Microsoft CryptoAPI and RSA labs PKCS#11.

1 Introduction

Increasing efforts are placed on security solutions for nomadic users. VPN clients and https support can be found in mobile devices such as phones and PDAs. The popularity of applications using cryptographic APIs is growing fast, specially in e-business, but also in e-government, e-learning, or e-health. Solutions based on smart cards offer portability (size, weight), robustness (humidity, temperature), wide support (standards, operating systems, readers), low cost, and high security due to their tamper-proof nature.

Smart cards are convenient wallet solutions for average user needs. They are used in a number of applications like e-purse applications, or as the basis of cryptographic APIs to store keys, or user certificates([1], [2, 3]). Similar efforts and solutions are needed for trust management, security cornerstone. Nomadic users need solutions offering trust management based on smart cards.

The list of trusted entities (CAs, signers, etc.), the purposes under which this trust is placed, or the level of trust of each entity, are data required to move along with the user, just like secret keys and personal certificates.

Besides, applications also demand access control. ACs, short for Attribute Certificates, are a standard solution [4], though popular applications don't use them yet. ACs lack of support in CryptoAPI and the support that gives PKCS#11 to them is very limited.

⋆ This work has been partially supported by UBISEC (IST STREP 506926) and Trust-ES (MEDEA+).

This work is restricted to applications using Microsoft CryptoAPI [5] (CAPI) and RSA Labs PKCS#11 [1] and PKCS#15 [6]. We are not addressing PGP [7] and GnuPG [8] here, though they are mentioned in the related work section 5. Giving support to CAPI, PKCS#11 and PKCS#15 a huge amount of platforms are covered from desktop computers to mobile phones and PDAs.

2 Problem domain

A brief definition about some topics may help the reader to understand better what is addressed in this paper:

– Nomadicity is the ability of a user to change the network access point as he moves, while the service is completely stopped and started again. Nomadicity implies discrete changes of location.
– User mobility refers to the ability of a user to maintain the same identity irrespective of the terminal used and its network point of attachment. Users on the move may use different types of terminal and applications to access services.
– Ad-hoc interaction. In many cases, a user may desire to interact directly with other peers without being connected to a local network. This situation may be common where there is no available service or just to save battery (roaming and network discovery is expensive in terms of battery life).

Web browsers and mail clients are the most used user applications and those applications typically use CAPI and PKCS#11. We have identified a number of possible scenarios showing the trust management security concerns for a mobile user.

– Alice moves among different terminals always using CAPI applications.
– Bob uses CAPI applications in some computer and PKCS#11 in others and moves among them.
– Yi moves among different terminals using always PKCS#11 applications.

When Alice moves to another computer, she may carry a smart card with her keys and personal certificates. On the other hand, she has to check manually the Certificate Trust List (CTL) in the new computer and the assigned policies and trust levels for that CTL. Everytime she decides to add or remove a new certificate to the CTL or to change the trust level associated with the CTL or the privileges, she must do this in all the computers she uses. She would benefit of carrying the CTL and the trust information in the smart card to avoid this task.

Similarly, Yi may hold a smart card with personal keys and certificates. Since PKCS#11 also allows to store in the token certificates (`CKO_CERTIFICATE`) with a boolean attribute (`CKA_TRUSTED`), which is set only by the user OS, or an initialization application. Unfortunately, few applications use this feature and do the CTL management themselves. Yi is also forced to manually control the CTL.

Bob is more flexible and can use more computers and applications. On the other hand, his life is harder: there is no communication among CAPI and PKCS#11, so he is forced to maintain his security decisions about the CTL and trust levels for both types of applications as he moves to a new machine. He will also benefit of carrying such data in his smart card. Besides, Bob will need some means to ensure that the semantics of PKCS#11 and CAPI are somehow maintained, so that the security information created by PKCS#11 applications is stored in the smart card in some way that it can be accessed by CAPI.

PKCS#15 offers more explicit support for CTL. Any number of Certificate Directory Files (CDFs) can be stored in the smart card, but in the normal case there will only be one or two (one for trusted certificates and one which the cardholder may update). The certificates themselves may reside anywhere on the card (or even outside the card, using a url and a `certHash` field for verification). Trusted certificates are used in PKCS#15 as trust chain origins (when signalled by the `implicitTrust` attribute). PKCS#15 also allows certificates to indicate the `trustUsage` which can be encrypt, decrypt, sign, signRecover, wrap, unwrap, verify, verifyRecover, derive and nonRepudiation.

Unfortunately, these features are not being exploited by applications, though mobile user benefit is clear. Trust management for mobile users implies that the security information regarding the trusted certificates and their level of trust moves along with the user. We claim that smart cards should be used for this purpose, as they are now used for key and certificate storage.

The usage of policies in certificates was the first approach taken in the PKI standard (ITU-T Rec. X.509[9], ISO/IEC 9594-8). Conceptually identity and permissions have different life and life-cycle, and they arise from different sources. This leads to scalability and management problems. A later version of the standard included PMI and attribute certificates (section 2.1), engineered as a more suitable solution for privilege management. Attribute certificates are supported in PKCS#11 and PKCS#15, but not in CAPI.

2.1 Attribute Certificates

An Attribute Certificate (AC) [4] is a structure similar to a Public Key Certificate (PKC) but ACs bind identities to privileges and contain no public key (Fig. 1). ACs are issued by Attribute Authorities (AAs). AAs store and manage them independently from Certificate Authorities (CAs). AAs are the source of authorization information, just like CAs are the source of authentication information.

Typically the AAs store the attribute certificates in a Lightweight Directory Access Protocol (LDAP). In this case, both the entity and the service perform authentication. Then the server requests or “pulls” the AC from a repository or an AA. Pull mode allows the use of ACs without changing the clients. This model is more appropriate for inter-domain access where privileges are assigned in the domain of the server, see Fig. 2.

In other environments, as ad-hoc, is more suitable for an entity to “push” ACs to the server. This model simplifies services, because no new connections

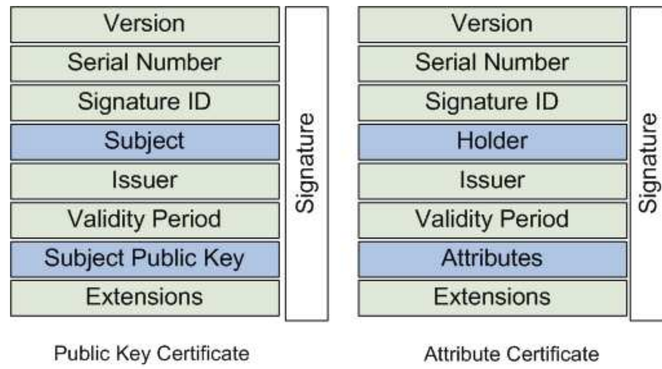

Fig. 1. AC structure

are needed to obtain authorization information. The push model improves performance since servers have instantly access to the authorization data and fit in ad-hoc environments where services can be located in peers and the access to Internet is not assured. Details about X509 identity certificate validation (necessary for AC exchange) in ad-hoc environments is covered in section 4.1.

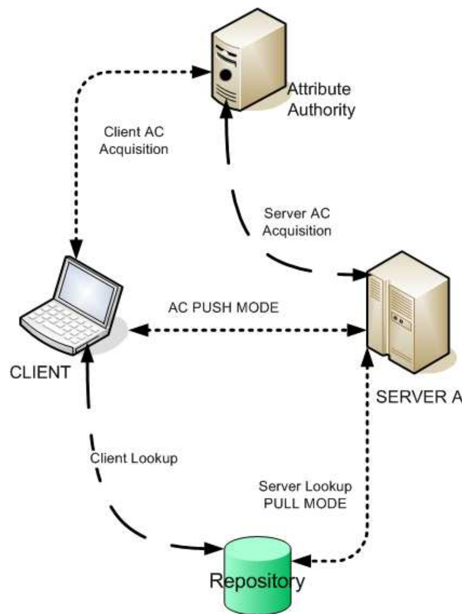

Fig. 2. AC exchanges

3 The cryptographic APIs

In this section we start by describing briefly the main features of PKCS#11 and Microsoft CryptoAPI. After that, we present why a bridge between them is needed, and the requirements for this bridge.

3.1 PKCS#11: Cryptoki

PKCS#11 (Public Key Cryptographic Standard) specifies an API called Cryptoki, short for Cryptographic Token Interface Standard, to devices that hold cryptographic information and performs cryptographic operations. The aim of this standard is to provide a simple object-based approach that allows applications to be independent of the underlying cryptographic hardware. Cryptographic devices are known, in Cryptoki words, as "*Cryptographic tokens* " or "*tokens*". Cryptoki defines a logical model that makes any device logically equal to any other regardless of the devices different technologies. Therefore, outside the library, the hardware details are hidden and the application has only access to the logical view of the *token*.

Cryptographic *tokens* are plugged in the system using "*slots*", which can correspond, for instance, to a smart card reader [10, 11] plugged in the system. *Tokens* can be, as mentioned before, any removable or not removable device plugged in a *slot*.

Cryptoki defines tree classes of objects: *data*, *certificate* and *keys*. *Data objects* hold application information, *certificate objects* store certificates and *key objects* store cryptographic keys that may be public, private or secret. Objects can be classified by visibility and lifetime in: *Token Objects*, that are accessible to applications connected to the *token* that are logged in and have sufficient permission. These objects remain on the *token* after all the sessions are closed and the *token* is removed from the *slot*. *Session Objects* remain on the *token* only within a session and are removed when the session finishes. Objects are also classified by accessibility in public and private objects. Public objects can be accessed by an application without login in the *token*, while private objects can only be accessed by authenticated applications (using, for instance, a PIN).

An application should open one or more sessions with a *token* to gain access to the objects contained in the *token*. A *session* is the logical connection between the *token* and the application. *Sessions* can be read-only and read-write ones. The read-only or read-write state refers only to the *token objects* so *session objects* can be written in a *read-only session*.

Sessions are referenced outside Cryptoki by the use of handlers. A *session handler* value is different from others. *Object handler* values can be equals for two or more *sessions* regardless of the object that is referenced since any *session* has it own handler space to reference objects.

3.2 PKCS#15 and JCCM

Cryptoki alone can not offer interoperability, since it is an API specification aimed at offering applications a uniform interface to cryptographic tokens. Dif-

ferent tokens require different PKCS#11 drivers, and users need to install the different drivers to use different tokens.

There are two workarounds for this problem. PKCS#15 [6] ensures interoperability by establishing a standard which ensures that users, in fact, will be able to use cryptographic tokens to identify themselves to multiple, standards-aware applications; regardless of the application's cryptoki (or other token interface) provider. PKCS#15 does this by establishing a syntax for storing digital credentials (keys, certificates, etc) on the tokens, and how this information are to be accessed. PKCS#15 only requires card compatibility with ISO/IEC 7816-4, ISO/IEC 7816-5, ISO/IEC 7816-6 and ISO/IEC 7816-15 [12]. Extended features, especially advanced PIN management functions and higher level security operations may require support for ISO/IEC 7816-8 or ISO/IEC 7816-9.

Another solution is the one proposed by Java Card Certificate Management [13]. JCCM proposes to move part of Cryptoki semantics to the smart card, so that the host library (cryptoki driver) is the same for all manufacturers. We have used this approach in this work.

3.3 CryptoAPI

CryptoAPI [5] is the security API for Microsoft Windows platforms from desktop computers to mobile smartphones and PDAs. It offers an API to: certificate stores, simplified messages, low level messages (PKCS#7), and certificate encode helped by extensible modules that interact each others see Fig. 3.

CAPI manages certificates providing a set of functions that allow searching, retrieving, deleting, and classifying certificates.

The low level cryptographic functions that CAPI export are served by a set of selectable modules or CSPs.

3.4 CSPs: Cryptographic Service Providers

A CSP is a software module that implements CryptoSPI API. This API provides to the operating system the ability to manage keys for both symmetrical and asymmetrical algorithms and to perform cryptographic operations as hashing, encrypting and signing. CSPs may interact with hardware, for instance, a card, usb-token or any tamper-proof device providing secure key management.

CryptoSPI is managed by the operating system, who blocks the software module to avoid for instance a man-in-the-middle attack. CryptoSPI is exported outside the operating system to be used by applications as a set of CAPI functions.

The Independent Software Vendor (ISV) model managed by the operating system allows to select among the available CSPs according to user preferences or domain policy enforcements. Using ISV model, developers can interact with more than one CSP to increase security and strength of applications. This structure allows application to have available providers that implements different public key algorithms, symmetric ciphers, and hash algorithms including communication with cryptographic devices (Fig.3).

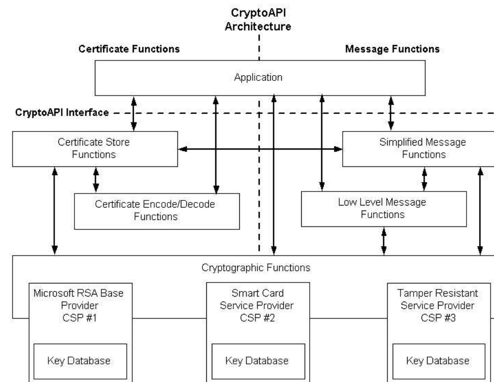

Fig. 3. CryptoAPI structure (from MSDN)

A CSP maintains a repository of keys organized by *containers*. Containers' names are given by the applications and each *container* has a unique name in the scope of a CSP. A CSP guards *container* structure and maintains the public/private keys stored in those containers from session to session. However, session keys are not preserved from one session to other.

CAPI specifies that CSP *containers* should support at least two types of key pair: "*signature key pair*"and "*exchange key pair*". Moreover, any number of key pairs can be stored besides the default ones, but it depends on the particular CSP implementation. The communication between the application and the CSP is materialized in a *context*. A context should be acquired before performing any operation with a CSP.

3.5 Certificate Manager

CAPI exports functions to store, list, retrieve, delete, and verify X.509 certificates. The API offers two main categories of functions for managing certificates: functions that manage **certificate stores**, and functions that work with the certificates, certificate revocation lists (CRLs), and certificate trust lists (CTLs) within those stores.

When a certificate is retrieved from a Store, it is represented by a **Certificate Context** read-only structure. A certificate context can be used for signing, encryption or for authentication processes. CAPI do not attach trust information to a certificate context but allows to develop custom **trust providers** by third parties to manage this type of information, through the *WinTrust* service. Trust providers perform trust verification on a specified object.

Certificate stores can be **System Stores** or **Physical Stores**. System stores are composed by a set of Physical Stores. Physical Stores are organized in store locations served by different **Certificate Store Providers**: system registry, disk file, and memory. These stores classify certificates depending on the intended use.

Certificates for trusted certificate issuers are generally kept in the Root store, which is currently stored under a registry subkey.

Besides, each user has a personal "My"store where user's personal certificates are stored; these certificates are used for signing, decrypting user's messages, and mail encryption. "My" store can be at many physical locations including the registry on a local or remote computer, a disk file, a database, directory service, a smart card, or another location. Private keys are generated and stored in key containers in CSPs. Furthermore, CSPs store the keys of the certificates generated by third-party CAs that have been imported to the system.

3.6 Bridging Microsoft CryptoAPI to PKCS#11

It is common for nomadic users to use both CAPI applications (like Internet Explorer, Outlook, etc.) and Cryptoki (Mozilla, Firebird, etc.). These users require some means to ensure that the semantics of PKCS#11 and CAPI is somehow maintained, so that the security information created by PKCS#11 applications is stored in the smart card in some way that can be accessed by CAPI.

A solution can be built upon PKCS#15, but this will still require some code to ensure that the information present in Certificate Stores is stored in the smart card.

Another solution is to provide a bridge from CAPI to Cryptoki, so that Cryptoki data objects are used to store such information. Since we work with JCCM, we ensure that the same host library is needed for different smart card providers. The actual version of CAPI does not support attribute certificates, and it needs to be extended for this purpose. Such extension is also considered in the bridge we propose.

4 Prototype

User life is not focused on security. Handling different certificates, using different personal devices may be hard unless a common certificate base, trust information and user preferences are shared among applications. This should be done regardless the Cryptographic API used. Our work covers four security cornerstones, by providing a seamless integration of two APIs and sharing credentials and cryptographic capabilities among them. It gives support for nomadic users allowing them to use a card for multiple purposes: authentication, authorization and trust management as well as tamper-proof capabilities.

The middleware developed to achieve this functionality can be divided in four blocks:

- **Trust Manager**: This module handles the trust in PKI and PTM.
- **Public Key Certificate Manager** provides tools to delete, to store, to search and to inspect certificates.
- **Attribute Certificate Manager** allows to handle, to store, to delete and to search Attribute Certificates
- **Key management and ciphersuite** provides cryptographic support to manage keys and to perform cryptographic operations

4.1 Trust Manager

Trust information is stored in the card for PKCS#11-aware applications and for CAPI-aware applications through our middleware . Trust is fundamental to validate user's certificates and their certification path. Such validation depends on the list of certificates stored as trustworthy (CTL), therefore, this module calls the CryptoAPI trust chain building functions, `CertGetCertificateChain` and `CertVerifyCertificateChainPolicy`.

After checking the validity of the type of certificate, expiration period, and certificate integrity, is needed to validate the trust chain. CryptoAPI functions perform the traditional validation process, which checks a valid certification path. In this process, the chain is built from the end certificate to a root CA; this latest must be in the Trusted Root Certification Authorities (TRCA) store, or the issuing CA must be in a trusted certification hierarchy or a CTL. During chain building, each certificate in the path will be validated, therefore, communication with each CA is required. When a problem occurs with one of the certificates, for instance, it is revoked, or if it cannot find a certificate, the certification path is discarded as a nontrusted certification path. Finally, the policy constrains are also validated.

Nevertheless, What happens if the root CA is not in the TRCA store or the user is unknown? Would this validation process be suitable for devices with restricted capabilities? or Is remote connectivity always guaranteed? In these cases, we need to use a different trust provider from CryptoAPI functions. Certificates would be validated taking the CTL into account, in this way, each certificate is a trust source. Furthermore, in ad-hoc environments, we could additionally use the cooperation between closed entities to trust in a specific user, instead of verifying long certification chains. In [14], we describe a trust management model (PTM) for open and dynamic spaces, where the presence of ad-hoc networks and peer-to-peer might be frequent. PTM would act as a trust provider according to the context, which is possible due to Windows security architecture allows several trust providers to verity trust in certificates [15].

4.2 Public Key Certificate Manager

This module acts as a bridge between CryptoAPI Certificate Manager and PKCS#11. The goal is to provide access to certificates stored in PKCS#11 modules from CryptoAPI Certificate Manager.

We have chosen to access to PKCS#11 modules from CryptoAPI and not the other way around. The reason is that most of the parameters that CryptoAPI attaches to cryptographic information can be extracted from Cryptoki attributes. Besides this “natural”mapping, Cryptoki supports application dependent data management and this can be used to store the information that CAPI needs to handle and the preferences without modifying PKCS#11 API.

It is necessary to describe CAPI internals to find out possible problems when dealing with different repositories: one for certificates and other for private keys.

As mention in related work (section 5), other bridges among APIs do not cover certificates but only key management and cryptographic capabilities.

When importing a certificate to a personal store, for instance the System Store “My”, the certificate and its public/private key pair are stored separately. Certificates themselves are stored in a Physical Store served by a Certificate Store Provider, that belongs to a System Store, and the private keys in CSPs.

There are some extended properties of a certificate help to solve the problem of having separate repositories, for certificates and private keys, these properties include data that:

- Pertains to the private key to be used with the certificate.
- Indicates the type of hashes to be performed on the certificate.
- Provides user-defined information associated to the certificate.

On Microsoft platforms, values for these properties are attached to a certificate context and move with it, but are not part of the certificate itself. Currently, predefined properties tie a certificate to a particular CSP and, within that CSP, to a particular private key. This properties set also the type of key: signature or key-exchange. This is the way to couple certificates stored in Certificate Stores and private keys guarded by the CSPs at API level. Moreover, these properties describe user preferences so they should be stored in the card in order to support nomadic users as explain the section 4.4.

CAPI Certificate Manager allows to be extended by developing custom certificate store providers, registering the appropriate callback functions in the operating system and registering new Physical Stores within this provider. We have developed a Certificate Store Provider called *PKCS11Store*. This provider is used to register new sibling physical stores under the well known system stores:

- “MY” personal certificates.
- “Root” Trusted root CAs certificates can be loaded into the card from either CryptoAPI or PKCS#11 aware applications and carried by nomadic users
- “TrustedPublisher” maintains a trusted list of software publishers
- “TrustedPeople” certificates from trusted people.

When applications use any of the prior certificate stores, *PKCS11Store* communicates with any PKCS#11 token registered in the system. Then *PKCS11Store* updates the system certificate repository with the information of the card, as can be seen in Fig. 4.

4.3 Attribute Certificate Manager

This module behaves in the same manner as Public Key Certificate Manager does, but it provides another kind of certificates, Attribute Certificates (ACs), to the system.

When an application accesses to certificate stores, it provides, as a parameter, the intended encoding type of the information requested. Currently, CryptoAPI supports the value of the bitwise OR operation of flags `X509_ASN_ENCODING` or `CRYPT_ASN_ENCODING` and `PKCS_7_ASN_ENCODING`.

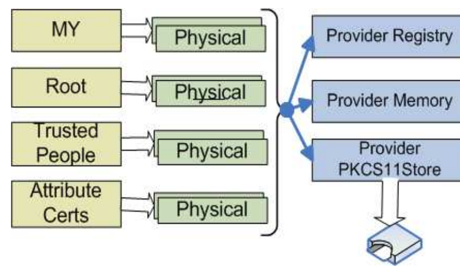

Fig. 4. Certificate stores and providers

To distinguish among different types of certificates, a new encoding type flag for ACs has been defined (X509AC_ASN_ENCODING). This enables the use of the same certificate store provider module, in our case *PKCS11Store*, to handle both Public Key Certificates and Attribute Certificates.

PKCS11Store in its current state of development is able to extract ACs from PKCS#11 modules and to accept requests from the system through a System Store registered with the convenient encoding type.

At the end of this paragraph, it is shown the "C" declaration of the Public Key Certificate context (defined in CAPI) and the Attribute Certificate context (defined by us) returned by the provider to the O.S. and from the O.S. to the application:

```
typedef struct _CERT_CONTEXT {
    DWORD                   dwCertEncodingType;
    BYTE                    *pbCertEncoded;
    DWORD                   cbCertEncoded;
    PCERT_INFO              pCertInfo;
    HCERTSTORE              hCertStore;
} CERT_CONTEXT, *PCERT_CONTEXT;

typedef struct _ACERT_CONTEXT {
    DWORD dwCertEncodingType;
    BYTE* pbCertEncoded;
    DWORD cbCertEncoded;
    PACERT_INFO pACertInfo;
    HCERTSTORE hCertStore;
} ACERT_CONTEXT, *PACERT_CONTEXT;
```

Where PCERT_INFO and PACERT_INFO contain the most relevant fields of those certificates. The main difference is that attribute certificates do not contain public key.

4.4 Key Management and ciphersuite

As explained, extended properties of certificates are used to "tell" the operating system where to find the private keys of that certificates (in which CSP and container are stored) and to handle user preferences as certificates usages not described in X.509 certificate extensions.

We have built a CSP, *PKCS11CSP*, which is able to interact with PKCS#11 modules. When *PKCS11Store* reads a certificate from a PKCS#11 module and publishes it in "MY" certificate system store, it sets the appropriate extended properties. These properties points to *PKCS11CSP* so the operating system uses *PKCS11CSP* to perform cryptographic operations with the certificate private key.

Bridging CryptoAPI and Cryptoki makes necessary to map both cryptographic APIs at logical level. This is the hardest design constrain, because the logical structure can not be easily mapped from one to other. This section covers the low level mapping from CAPI to PKCS#11: CSPs to PKCS#11.

Some approaches to designing the middleware are described in the following lines. Fist approach may map CSPs to *slots* and *containers* to *tokens*. So *slots* can have plugged more than one *token* as CSPs can manage more than one *container*. The disadvantage is that the creation of a new *container* is associated to the creation of a *token*, but: 1) *tokens* can be physical devices, and 2) Cryptoki does not support *token* creation.

In a second approach, *containers* may be assigned to *certificates* stored in the *token* [16] so container names are derived from the certificates or any of their properties. The problem arises when using a read-write *token* and an application creates a *container*. If no certificate is created according to the name of the container, next time may be infeasible to derive the container name from the certificate, so this approach is only applicable to read-only tokens.

As defined in CryptoAPI a *container* may hold an unlimited number of key pairs but at least two are mandatory: a "`signature key pair`" and a "`key exchange key pair`". Thus, assigning a *container* to each *certificate* may not behave properly.

PKCS11CSP uses a general purpose object, a Cryptoki *data object*, to store user customization and preferences of CAPI that cannot be derived from the Cryptoki object attributes. Storing this information in a data object enables the user to reproduce his/her preferences everywhere. Data objects are covered by PKCS#11 and Cryptoki does not attach to them any special meaning. The data object (`CKO_DATA` type) is a persistent object (attribute `CKA_TOKEN` true), visible without authentication (`CKA_PRIVATE` false), but it cannot be modified unless the user is authenticated. We store the information in the data object encoded with DER .

The core of the *Key Management and Ciphersuite* module, *PKCS11CSP*, can create a non limited number of *containers* and any *container* can guard "`signature key pair`", "`key exchange key pair`" and more. To distinguish among different key types, this information is stored in the data object of the PKCS#11 module and extracted when it is necessary. Furthermore, the con-

tainer structure is committed to the token data object. Fig. 5 may help the reader to understand the middleware and how blocks interacts each other.

This design doesn't use anything out of the scope of the PKCS#11 standard, so any commercial PKCS#11 [17–19] module, capable of storing data objects, can be used. Read only tokens can be used, but obviously no changes are committed to the card.

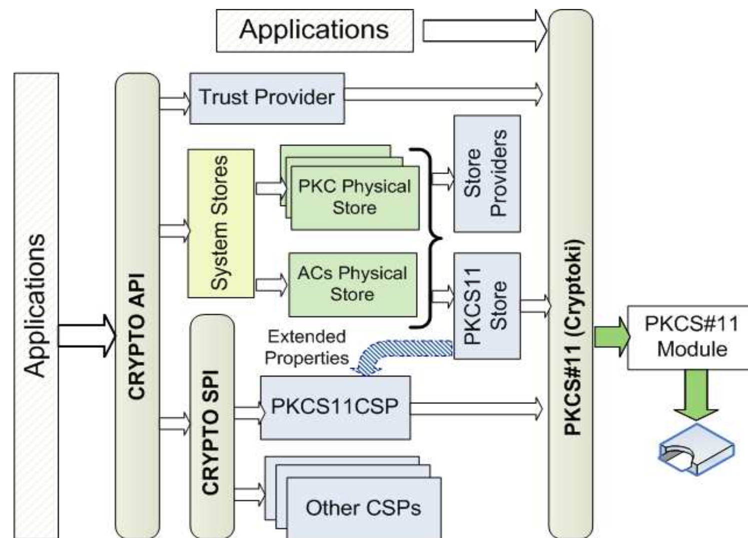

Fig. 5. Middleware structure

Context Mapping: Depending on the type of context requested by the application to the *PKCS11CSP*, the bridge should behave in a different fashion [2]. *PKCS11CSP* provides a GUI to enter PIN when access to private objects of PKCS#11 is requested. The following lines briefly describe most relevant types of CSP *context* and the mapping to PKCS#11 sessions:

- `Normal context (zero value)`: If operations requiring access to private token objects are requested, the context maps to a *R/W user function*. If not, just *R/O access* to public token objects and *R/W access* to session objects will be requested to the PKCS#11 module.
- `CRYPT_VERIFYCONTEXT`: This flag is intended for applications that do not use public/private keys. When this flag is set, the *context* is mapped to a *R/O Public Session*. Attempts to access to private objects should fail.
- `CRYPT_NEWKEYSET`: To create a new *container* with the specified name. R/W access to the token is needed: the data object stored in the *token* needs to be updated with new information. (PIN is needed).

– `CRYPT_DELETEKEYSET`: To delete a *container*. R/W access to token objects is needed to update the data object and to delete the key pairs (PIN is needed).
– `CRYPT_SILENT`: The *PKCS11CSP* should not display any user interface. It is used for unattended applications (*R/O Public Session*).

Function mapping: Key generation and exchange functions exchange, create, configure, and destroy cryptographic keys. In general, functions that require to create a new object should be mapped through calls to Cryptoki function `C_CreateObject`. If any persistent element is created, the *PKCS11CSP* should update the data object including the name of the involved container and additional data. The additional data will help to find the object faster in following *sessions*, and ensures that the user will access to the same object structure as the last *session*.

Data encryption functions support encryption and decryption. CAPI `CPDecrypt` and `CPEncrypt` functions support both single-part operation and multi-part operation (used in hashing and block ciphering). These functions will be mapped with a first call to `C_EncryptInit` to set up the algorithm. Then, for single-part operations, a unique call to `C_Encrypt` should be done. In case of multi-part operations, it is necessary to perform a set of calls to `C_EncryptUpdate` and a last call to `C_EncryptFinal`.

Hashing and digital signature functions compute hashes and create and verify digital signatures. Cryptoki does not define hash objects. A hash exists as a value during the active Cryptoki hash operation, but not as an object. CSPs define hash objects so they can be destroyed and duplicated. Due to this, no mapping is possible for CSP functions that allow creation, deletion and manipulation of hash objects. These hash objects will be handled by the *PKCS11CSP* library internally, without Cryptoki object manager intervention, and will disappear once the library unloads from memory (session objects). In any case, the hash value will be calculated in the token although the object is managed by the CSP.

Digital signatures generated by `CPSignHash` (CryptoAPI) will be mapped into calls to `C_SignInit`, `C_Sign`, `C_SignUpdate` and `C_SignFinal`. Verification process will be handled by calls to `C_Verify` and `C_Verify-Init-Update-Final`.

5 Related work

There have been a number of approaches to the interoperation of PKCS#11 and CSPs. The International Cryptography Experiment [20] aimed at a layered cryptographic service architecture, allowing the separation among cryptographic security and applications. ICE provided a bridge from CSP to PKCS#11 similar to CSP11 [20], or ilex's Generic CSP [16]. All them restrict themselves to PKCS#11 basic trust management, and do not take into account CTLs in the smart cards. These approaches lack of mobility support, since users have to set up cryptographic low level details in their personal profiles. GnuPGP [8] deals with CTLs in the smart cards, but the trust model is not compatible with PKI. Our proposal deals with CTLs and builds up a trust model compatible with PKI.

We also approach the smart card as attribute certificate holder. Being the smart card an extremely mobile device, it allows a push model more suited for disconnected situations and ad-hoc networks. Besides, the user can require privacy for some of his/her privileges. There are also works for achieving privacy through anonymity. [21] presents an approach to extend X.509 attribute certificates with anonymity, and a protocol based on fair blind signature to obtain certificates preserving user anonymity.

6 Conclusions and future work

Smart cards offer portability and security at a reasonable price. Nomadic users benefit from existing smart card based solutions, but they require extra management efforts when moving to different platforms. In this article we propose a scheme to include trust management and attribute certificates for authorization in two widely used cryptographic APIs: Microsoft CryptoAPI and RSA labs PKCS#11. Our solution is truly mobile, since platform cryptographic details are stored in the smart card, as a user profile, minimizing thus users' management effort when moving.

We are actually testing the implementation in PCs and Windows CE handhelds (not PDAs). Future work will include usability tests and porting the implementation to Windows Mobile devices (PDAs), which do not support yet PC/SC smart card readers.

Microsoft CAPI will be held in Windows Vista but it will be deprecated in some future version. The brand new API that supersedes CAPI in Windows Vista is known as Cryptographic New Generation (CNG). The information made available by Microsoft about this new API is not detailed enough to review the design at the time this paper was written.

One of the changes that Microsoft provides for new Windows Vista affects our design: the new Smart Card Infrastructure. Currently, a monolithic approach has been used to develop the CSP. A CSP implements the CryptoSPI interface and the communication with the card through PC/SC. CNG provides a Smart Card KSP (Key Store Provider), that seems to be a common middleware, which interacts with other modules that implements the details (i.e. RSA card module). Our next efforts should be directed to implement one of those modules that cover the specific smart card details. This module will be in charge of interacting with PKCS#11. Other parts of our middleware will need to have access to a more detailed documentation from us to be reviewed.

References

1. RSALabs: Pkcs#11 v2.11: Cryptographic token interface standard (2004)
2. Microsoft: The smart card cryptographic service provider cookbook (2002) http://msdn.microsoft.com/library/en-us/dnscard/html/smartcardcspcook.asp.
3. Microsoft: Writing a csp (2004) http://msdn.microsoft.com/library/en-us/dnscard/html/smartcardcspcook.asp.

4. Farrell, S., Housley, R.: An internet attribute certificate profile for authorization. Technical Report RFC 3281, IETF PKIX Working Group (2002)
5. Microsoft: Cryptography reference (2004) http://msdn.microsoft.com/library/ default.asp?url=/library/en-us/seccrypto/security/cryptography_portal.asp.
6. RSALabs: Pkcs#15 v1.1: Cryptographic token information format standard (2000)
7. Zimmermann, P.R.: The Official PGP User's Guide. MIT Press, Cambridge, MA, USA (95)
8. Team, T.G.: Gnupg (2005)
9. Union, I.T.: The directory: Public-key and attribute certificate frameworks. Technical Report X.509, International Telecommunication Union (2000)
10. ISO/IEC: 7816-4: Integrated circuit(s) cards with contacts. part 4: Interindustry commands for interchange (1995)
11. ISO/IEC: 7816-3: Integrated circuit(s) cards with contacts. part 3: Electronic signals and transmission protocols (1997)
12. ISO/IEC: 7816-15: Integrated circuit(s) cards with contacts. part 15: Cryptographic information application (1997)
13. Campo, C., Marin, A., Garcia, A., Diaz, I., Breuer, P., Delgado, C., Garcia., C.: JCCM: flexible certificates for smartcards with java card. In: Smart Card Programming and Security. Proceedings of the international Conference on Research in Smart Cards, E-Smart 2001, Springer-Verlag (2001)
14. Almenárez, F., Marín, A., Campo, C., García, C.: PTM: A Pervasive Trust Management Model for Dynamic Open Environments. In: First Workshop on Pervasive Security, Privacy and Trust PSPT'04 in conjuntion with Mobiquitous 2004. (2004)
15. Almenarez, F., Diaz, D., Marin, A.: Secure ad-hoc mbusiness: Enhancing windows ce security. In: Trust and Privacy in digital business. First International Conference, (TrustBus 2004, Zaragoza, Spain). Number 3184 in Lecture Notes in Computer Science, Heidelberg, Germany, Springer-Verlag (2004)
16. TEAM, I.S.: Pkcs_csp (2003) http://www.ilex.fr.
17. Gemplus: Gemsafe products: Gemxpresso pkcs#11 documentation (2004) http://www.gemplus.com/products/software/gemsafe_xpresso/.
18. Cucinotta, T.: Smart sign pkcs#11 modules (2005) http://sourceforge.net/projects/smartsign.
19. Axalto: Cyberflex access sdk: Pkcs#11 module for cyberflex (2004) http://www.axalto.com/infosec/cyberflex_access.asp.
20. Libre-entreprise, R.: Cryptographic service provider number 11: How it works (2004) http://csp11.labs.libre-entreprise.org.
21. V.Benjumea, J.Lopez, J.A.Montenegro, J.M.Troya: A first approach to provide anonymity in attribute certificates. In: PKC 2004 International Workshop on Practice and Theory in Public Key Cryptography. LNCS 2947, Springer-Verlag (2004)