UNIVERSITÀ CA' FOSCARI - VENICE

Department of Environmental Sciences, Informatics and Statistics

MSc in Computer Science

Master Thesis

Security and interoperability of APIs for cryptographic devices

Supervisor: Prof. Riccardo Focardi

Student:
Andrea Pretotto (820955)

A.Y. 2012-2013

special thanks to my parents, grandparents, and all my friends

Contents

List of Figures

Chapter 1

Introduction

In this work two important API standards are analyzed: PKCS#11 and Microsoft CryptoAPI. A security API is an *Application Program Interface* that allows users to interact with trusted modules and work with sensitive data in a secure way. Cryptography is used to ensure a communication between an untrusted source (host, application) and a trusted service, such as a web server or a hardware device (security tokens). A first degree of security is provided by PIN authentication, i.e. something that user "knows", but it can be easily intercepted then broken. The attention is to the analysis of the standards, highlighting their vulnerabilities.

Starting from a previous security analysis of the standard PKCS#11 [2, 5] we try to identify if attacks, which were discovered on PKCS#11 devices, also affect CryptoAPI. Some tokens, which we do not cite, have been taken in consideration for that purpose. Despite key management is quite different from PKCS#11, similar vulnerabilities were found, e.g. the *wrap decrypt* attack. CryptoAPI does not allow to store symmetric keys within the device, being *session keys*, and key exportation is only made through a specific structure, the *keyblob.* The work proceeds with an integration of the two standards to deeply discover more vulnerabilities, analyzing the limits and delivering new solutions.

Chapter 2

CryptoAPI standard

CryptoAPI (or CAPI) [10], by Microsoft, is a set of cryptographic functions performed by independent software modules, the Cryptographic Service Providers (CSPs), to ensure operations such as encryption, decryption and signature of messages, and secure key storage. The modules are implemented with a set of dynamic libraries, and work apart from user applications. Applications start a session (i.e. a *context*) with the cryptographic token after a call to the function *CryptAcquireContext()*, specifying the correct CSP. Each CSP consists of an unique ID name, specified by the application when the context is acquired, and a type, which defines the operations that can be carried out, providing a different implementation of the Cryptography API layer. One CSP, the Microsoft RSA Base Provider, is included with the operating system, while other cryptographic provides allow to use different and stronger cryptographic algorithms. Some of them are based on hardware components, such as smart cards and tokens.

Software CSPs offer more flexibility than hardware CSPs, which often allow a limited number of private keys to be stored (our token supports two pairs of exchange and signature keys) requiring at the same time a longer time to be generated and to sign data, but with a lower overall security. To give an example, both the Base and Enhanced Cryptographic Provider use RSA technology for signing or key exchange, however the first one public key length is set to 512 bits as default, the minimum recommended, while the stronger one reaches up to 16KB. Differences also affect the algorithms used for symmetric (or session) keys. On the other side, hardware-based cryptography and key management is more secure than software-based cryptography and key management because cryptographic operations and private keys are isolated from the operating system, with the assurance of an encrypted communication and user authentication. Software CSPs usually provide more flexibility than hardware CSPs, but at the cost of somewhat less security.

2.1 Token device for tests

The study of CAPI was conducted over software and hardware providers, such as usb smart card tokens. The hardware device considered in the analysis a token of a known brand. The token offers a two-factor authentication for secure access, i.e. a knowledge factor (the PIN) and a possession factor (the smart-card), providing generation and protection of user's sensible credentials, such as private keys, password and certificates, inside the protected internal memory. API and standards support includes CAPI and PKCS#11. Private keys can never be exposed outside the token, in contrast to some providers (e.g. Microsoft base and enhanced provider) which allow an authenticated user to export the private key into a blob. Every public key is instead exportable by default.

2.2 Keys

Each CSP contains a key database, which consists of one or more key containers, where all the cryptographic keys of a specific user are safety stored; after it has been acquired acquired, user can create a default key container, which takes user's login name as its name, or retrieve an existing one. Once a user has received the handle to the container (then he is inside a session), the application can access the objects stored on the token, i.e. keys and certificates. Only two types of keys can be stored: the exchange and signature public/private key pairs. The first type refers to keys used to encrypt session keys so that they can be securely exported and then exchanged with other users via an encrypted and secure format (key blob), while the second one is about messages authentication (digitally signing). For each key container, two private/public key pair can be stored (one for each type); our token supports up to two key containers. Keys have a set of permissions, i.e. boolean parameters that specify key roles. Some of them are specified at creation time, e.g. the parameter *exportable*, which identifies if a key is exportable outside the CSP, the other ones are automatically set depending on key type. Permissions are read by the function *CryptGetProvParam*, or via PKCS#11 (but here they are called *attributes*), but anyway any flag (encrypt, decrypt, wrap, etc.) applied to session keys has no effect, because they are not kept in memory after session has been closed. As it follows in next sections, when session keys are exchanged with another user, one key container should store user's own public/private key pairs, the other the receiver's public key (previously imported from a public key blob). The container can be protected by a PIN, or password, stored in a smart-card or usb token, and, as the keys inside, is accessed only through a handle. However, if the PIN is intercepted, a malicious user can easily access the smartcard and use all the keys that are stored inside. This vulnerability constitutes a starting point in order to carry out attacks in the token.

A key container is preserved by the CSP from session to session, including all the keys it stores, while session keys, even if kept inside the CSP for a security purpose, are not preserved being volatile, but can be saved for further use into a specific blob type. Session keys are treated as symmetric keys, being used with symmetric algorithms to encrypt and decrypt data, and should be frequently changed, better if after one cryptographic operation. Next we will use the terms symmetric and sessions without difference in meaning. Keys, after having acquired a CSP context, can be generated randomly by the functions *CryptGenKey()* or, for session keys only, by *CryptDeriveKey()*, which derives a key from an input string. According to the *ALG_ID* parameter, an user can create a pair of keys (*at_keyexchange* or *at_signature*) or a session key, specifying a valid algorithm (the list of available algorithms depends on the CSP used). If the CSP stores a user digital certificate, the related public key can just be retrieved by the function *CryptGetUserKey*.

The security of keys storage directly depends on the CSP and its design. Three rules should be applied:

- Keys, generated within CSP, are not directly accessible by the application except through handles, in order to deny access to the key material or derive the key from random sources.

- Details of cryptographic operations cannot be implemented or modified by the application. The implementation is internal to the CSP, while the algorithm (the available ones) to be performed can be chosen by the application.

- User authentication is managed by the CSP; the application does not store any user credentials.

Every operation refers to a key through its handle, but it is still possible to export the key into a particular format and save it out of the application for future uses. Sensitive keys, which are, by default, private keys, and in general all CAPI keys can be exported into a particular structure, the key blob, if it is not prevented by the CSP (e.g. our token does not allow private keys to be exported) or the flag "crypt_exportable" is not set. Except for public keys, which are exported in clear, a key is *wrapped*, i.e. exported after it has been encrypted under another key. Session keys are usually wrapped with a public key or another symmetric key, while the private key is stored into the blob "as is" or encrypted with a symmetric key. Next picture represents the relation among cryptographic components.

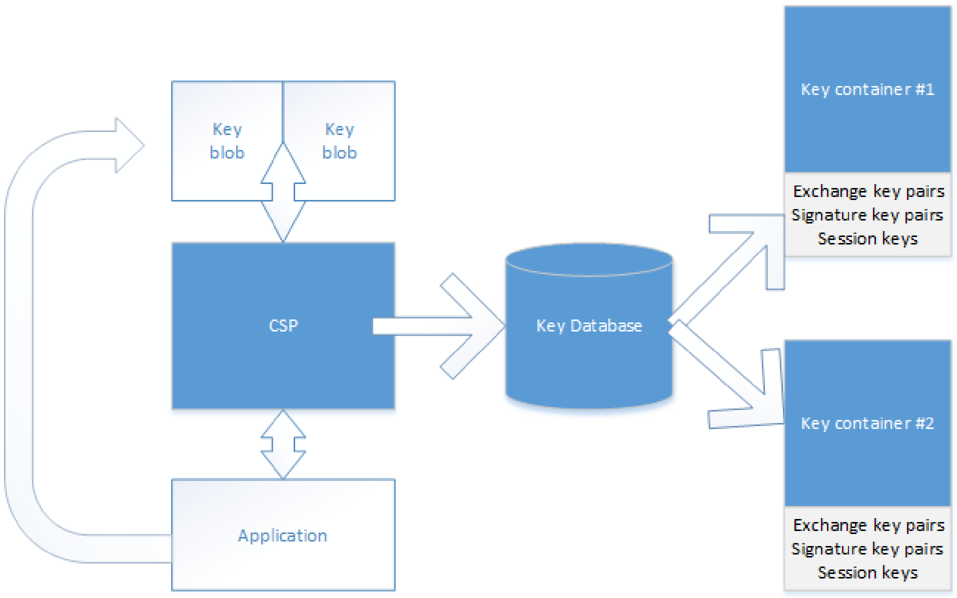


Figure 2.1: CSP and key diagram

2.3 Session keys

In CAPI, session keys can be generated by every CSP with the function *CryptGenKey()* or *CryptDeriveKey()* and exported, then stored, into the application level. The primary use consists on encryption and decryption of a message, which would be exchanged between two processes. To guarantee the security and the secrecy of a symmetric key, the latter must be encrypted with a public key (*simpleblob*), and then imported using the corresponding private key of the key pair before decrypting the encoded message. If the message has to be exchanged between two local processes, the CSP will use its own public key to export the session key into a *simple* key blob, so that the session key used can be imported into the CSP using the "local" private key. The scenario would be different if two users want to exchange a message: the sender has to wrap the symmetric key with the receiver's public key and later on the receiver will use his private key to decrypt the session key.

Key management should prevent any intruder to get a secret key during a communication between two or more host. Some rules are adopted during key generation and specific structure are used to store session keys outside trusted devices. It is very important that a reliable key is generated randomly, in order to prevent any known dictionary attack [14], and, for encryption, CBC mode (*Cipher-block chaining*) should be used in addition to an initialization vector IV of $non-zero$ elements, so that an attacker must try every possible IV before decrypting a cypher text. More details are showed in the next section.

2.4 Key blob structures

Keys, once exported from the CSP, are stored into key blobs, and then can be used by the application level. Each type of key blob has a common header, the *publickeystruc* structure, which indicates blob type and the key algorithm's ID. Key pairs can only be exported into a *publickeyblob* or *privatekeyblob* structure. The *publickeyblob* stores the public key for use with the RSA key exchange algorithm: the public exponent and the modulus info is kept inside the format. The private key can be saved "in clear" through the corresponding format, but in general CSPs, like the token's csp, do not allow it to be exported outside the key storage. Session keys can be exported in three types

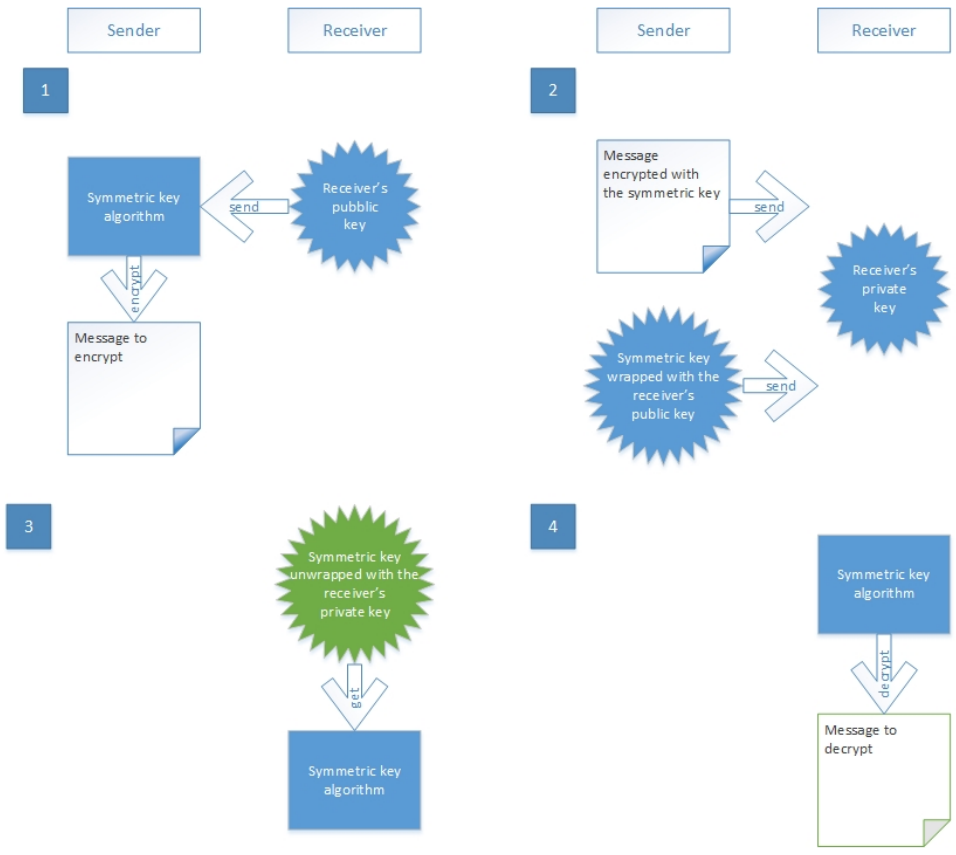


Figure 2.2: Exchanging session keys

of blobs: a *plaintextkeyblob*, that is the ”raw” key material preceded by the common blob header, a *simplekeyblob*, a ”wrapping format” in which the session key is encrypted with a public key before being exported, and a *symmetricwrapkeyblob*, a format specified in the IETF SMIME X9.42 standard [7], used to wrap session keys with another session key.

According to the function *CryptImportKey()*, setting the proper parameters, any type of key can be imported into a CSP, starting from a key blob format. A key blob can be managed outside the CSP, and then written or read to/from a text file. The structure can be manipulated (i.e. changing byte values in the correspondence of key algorithm, modulus or key exponent) or created ex-novo to insert a desired key and allow it to be imported into the CSP and then recognized as a valid key. The simplest scenario concerns session keys and the *plaintextkeyblob* structure: starting from such type of blob, it is possible to create a random or known key and save it to the structure, adding the correct parameters of key size and key algorithm. A valid key is the following:

Listing 2.1: create a random key

```
// Allocate memory for 3DES key and
// Fill key with data 1,2,3,... in this case
pbKeyMaterial = (LPBYTE)LocalAlloc(LPTR, 192/8);
for (n = 0; n < 192/8; n++)
    pbKeyMaterial[n] = n+1;
    dwKeyMaterial = 192/8;
```

The key is then inserted into a *plaintextkeyblob*, or it can be encrypted under a public key and then inserted into a simple blob. This operation is showed in the next sections, when it is given the description of an attack.

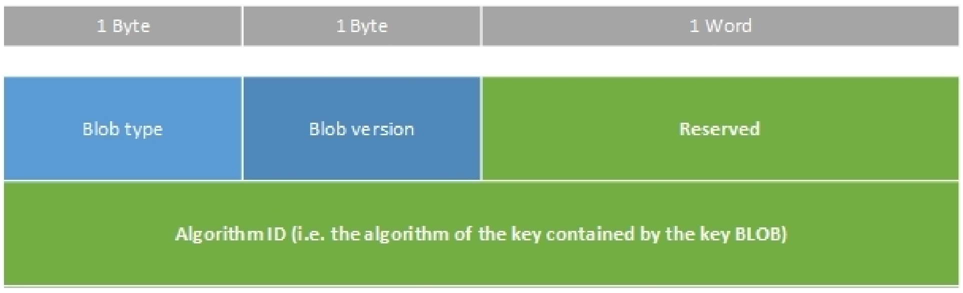

1 Byte
1 Byte
1 Word
Blob type
Blob version
Reserved
Algorithm ID (i.e. the algorithm of the key contained by the key BLOB)

Figure 2.3: Blob header (publickeystruc structure)

The *simpleblob* can be modified to set the proper key algorithm and its size to add a new key material, previously encrypted with a public key (which belongs to a exchange public/private key pair). Being part of the application level, key blobs become an interesting starting point to analyze and discover vulnerabilities of CAPI.

Chapter 3

A model for CAPI

The semantic of our model is based on the one proposed in [6] to model PKCS#11 operations. This provides a starting point to build a more generic model to describe both CAPI and PKCS#11 and their interoperability.

3.1 Primitives

Asymmetric and symmetric encryption of a plain text x under the key y:

$\{x\}_y$ = aenc(x,y)
$\{\|x\|\}_y$ = senc(x,y)

Key data stored in key blobs:

$\left[\{x\}_y\right]$= key blob with aenc(x,y)
$\left[\{\|x\|\}_y\right]$= key blob with senc(x,y)
$\left[x\right]$= key blob with x not encrypted

The functions *aenc(x,y)* and *senc(x,y)*, of arity 2, are formally defined as

$$\begin{array}{l} \text{aenc: key} \times \text{string} \rightarrow \text{string} \\ \text{senc: key} \times \text{string} \rightarrow \text{string} \end{array}$$

3.2 Basic notation

The finite set of *function symbols* is Σ, with the arity function
ar: $\Sigma \rightarrow \mathbb{N}$, where $\Sigma \subset \mathbb{N}^n$ and n the number of arguments.
Plain terms (PT) is defined as:

$$\begin{aligned} t_1, t_2 &= x \quad x \in X \\ &| = n \quad n \in N \\ &| = f(t_1 ... t_n) \quad f \in \sum \quad and \quad ar(f) = n \end{aligned}$$

with N the set of keys, nonces and data values, and X the set of variables.

Key access is modeled with functions of arity 2.

h: nonce $\times$ key $\rightarrow$ handle

We use $h(n_1, k_1)$ to define the handle n_1 of the key k_1.

To directly access the encrypted key, start from the 12^{th} byte position of key blob: this part corresponds to aenc(x,y) or x, if the blob is exported without encryption (i.e. plaintextblob, publickeyblob).

$$\left[\{x\}_y\right] \rightarrow \{x\}_y$$

If the blob is a *symmetricwrapkeyblob*, the key is stored according to the format specified in [7]. We assume to have performed all the necessary operations to get the key in the "clear" format $\{x\}_y$, with y a symmetric key.

$$\left[\{|x|\}_y\right] \rightarrow \{x\}_y$$

Implementations of symmetric key blob must protect the encrypted key, i.e. the stored secret key, in the best possible way. When a key is generated, a default IV is applied: for a better security, implementations of symmetric key wrapping must generate random initialization vectors (IVs) and padding. To decrypt a symmetric key blob (in our tests, we use 3DES symmetric keys), if no change on IV is made, we can follow the procedure about *Triple-DES Key Unwrap* in the documentation of *rfc3217* standard [7].

3.3 CAPI syntax

A first difference respect to PKCS#11 is the absence of templates (set of attributes): a key has some permissions, i.e. operations it can perform, such as decrypt, encrypt, wrap, unwrap, assigned at creation time according to key type (or specification), determined by the algorithm identifier ALG_ID, and some flags. Depending on key roles, ALG_IDs can assume different values for (symmetric) session keys. These keys can perform all operations by default.
For key pair, ALG_ID can be substituted by a key specification (ks):

$$\text{ks} = \{exchange\_key, signature\_key\}$$

In this way, the algorithm used to generate a key pair depends on the cryptographic provider. To refer to a session key, without specifying an algorithm, we add the key specification *session_key*, which does not belong to CAPI syntax. Every time a key is created with this specification, a suitable symmetric algorithm is applied. Usually, *CALG_RSA_KEYX* is used for key exchange and *CALG_RSA_SIGN* for key signature. Key specification determines a different set of permissions: e.g. for key exchange, the public key is wrap and encrypt, the private is unwrap and decrypt.
When generating a RSA key pair, CAPI maintains only one handle for both the key. The key pair consists on a set of numbers, each one specifying a key parameter, such as the private and public exponent, the modulus, the prime numbers, and so on. From a key pair k_p, we refer to the public key component writing *pub*(k_p). As described in *Cryptographic security APIs* [13]:

$$\text{pub: seed} \rightarrow \text{key}$$

In PKCS#11, to get a handle to *pub*(k_p), we must call the PKCS#11 function *CreateObject()* [11]. PKCS#11 or CAPI key pairs are stored into the cryptographic providers with the label *at_key_exchange* or *at_key_signature*. In CAPI, the choice of the key (public or private key) in each operation, is automatically selected; we only need to call the key pair handle. Other parameters used at key generation are numeric constants, the *flags* (f), which set additional properties to keys. The set of flags is defined as follows:

$$\text{FLAGS} = \{exportable, archivable, ..\}$$

To assign more than one flag f to a key, we should concatenate them. The suitable data structure is a *list*, represented with λ.

$$\lambda = f_1|f_2|\dots|f_n \quad f_1 \dots f_n \in FLAG$$

A generic key is represented by k_1, k_2, while a public/private key pair is k_p. When a key is encrypted using a key pair, i.e. it is encrypted under a public key, we use *pub(k_p)*, to state that only the public exponent and the modulus are used to perform encryption.
The set of Type Terms (TT) can be defined as:

$$\begin{aligned} tt_1, tt_2 &= (n, ks) \quad ks \in N \\ | &= (n, \lambda) \quad \lambda = f_1 \dots f_n \end{aligned}$$

Each term is a couple with a nonce (i.e. handle to the key) n and a list of flags or a key specification.
System is described with this set of rules:

$$T; \Gamma \xrightarrow{new\tilde{n}} T'; \Gamma'$$

where T, T' $\subset$ PT are plain terms, such as keys and nonces, and Γ, Γ' $\subset$ TT are type terms, i.e. conditions to apply to keys. T represents user's knowledge and Γ is the current system state before performing the attack.

3.4 CAPI operations

Main operations with CAPI:

Gen_key: the function generates a random session key, and returns a handle to the key k_1 in n_1. Generally, λ = exportable.

$$\text{Gen\_key:} \quad \xrightarrow{new n_1} h(n_1, k_1); (n_1, \lambda), (n_1, session\_key)$$

Gen_key_pair: the function generates a key pair, and returns a unique handle. According to key specification *ks*, a key pair is for key exchange or for signature. During tests, we create exchange key pairs to maintain interoperability with PKCS#11 (as we would see, a PKCS#11 key pair is recognized as an exchange key when saved in the token) with the flag *exportable* always set.

$$\text{Gen\_key\_pair:}\quad \xrightarrow{new\, n_p} h(n_p, k_p), pub(k_p); (n_p, \lambda), (n_p, ks)$$

If the flag *archivable* is specified, the function generates a key pair, stored in the device memory, and a copy of it, maintained in the current session, with the corresponding handle. When the session is closed, the session key pair is destroyed.

$$\text{Gen\_key\_pair:}\quad \xrightarrow{new\, n_{ps}} h(n_{ps}, k_{ps}), pub(k_{ps});$$
$$(n_{ps}, exchange\_key), (n_{ps}, archivable)$$

Export: the function takes a handle to the key (k_2) to be exported and, if requested, the handle to the encrypting key (it depends from the key blob type in which the key is exported to). If the blob is not encrypted, such as a public or plain text blob, no encrypted key is passed. To export the key, it must be exportable. The public key is always exportable. The function returns a key blob. For every key blob type:

$$\begin{array}{ll} \text{Export(asym):} & h(n_p, pub(k_p)), h(n_2, k_2); (n_2, exportable) \\ & (n_p, exchange\_key) \rightarrow \big[\{k_2\}_{pub(k_p)}\big] \\ \\ \text{Export(sym):} & h(n_1, k_1), h(n_2, k_2); (n_2, exportable) \\ & (n_1, session\_key) \rightarrow \big[\{\|k_2\|\}_{k_1}\big] \\ \\ \text{Export(plain):} & h(n_2, k_2); (n_2, exportable) \\ & \rightarrow \big[k_2\big] \\ \\ \text{Export(public):} & h(n_p, k_p); (n_p, exportable) \\ & \rightarrow \big[pub(k_p)\big] \end{array}$$

An additional external condition is necessary to describe if a private key is exportable: *is_private_exportable*. Even if the flag *exportable* is set to true, a private key is exportable if the current CSP allows it (Microsoft software CSPs allow it; the token which we use does not allow it).

if(is_private_exportable)

$$\begin{array}{ll}
\text{Export(private):} & h(n_p, k_p); (n_p, \mathit{exportable}) \rightarrow \big[k_p\big] \\
\text{Export(private):} & h(n_p, k_p); (n_p, \mathit{archivable}) \rightarrow \big[k_p\big] \\
\text{Export(private):} & h(n_2, k_2), h(n_p, k_p); (n_p, \mathit{exportable}) \\
 & (n_2, \mathit{session\_key}) \rightarrow \big[\{k_p\}_{k_2}\big] \\
\text{Export(private):} & h(n_2, k_2), h(n_p, k_p); (n_p, \mathit{archivable}) \\
 & (n_2, \mathit{session\_key}) \rightarrow \big[\{k_p\}_{k_2}\big]
\end{array}$$

else

$$\begin{array}{ll}
\text{Export(private):} & h(n_p, k_p); (n_p, \mathit{archivable}) \rightarrow \big[k_p\big] \\
\text{Export(private):} & h(n_2, k_2), h(n_p, k_p); (n_p, \mathit{archivable}) \\
 & (n_2, \mathit{session\_key}) \rightarrow \big[\{k_2\}_{k_2}\big]
\end{array}$$

Import: the function imports a cryptographic key from a key blob into a CSP. It takes a key blob and a handle to a decrypting key: it must be the symmetric key used to wrap the key stored in the key blob or the private key of an exchange key pair (i.e. the corresponding public key is used to encrypt the key). If the blob is not encrypted, no key is passed as first parameter.

$$\begin{array}{ll}
\text{Import(asym):} & h(n_p, k_p), \big[\{k_2\}_{pub(k_p)}\big], (n_p, \mathit{exchange\_key}) \\
 & \xrightarrow{new\, n_2} h(n_2, k_2); (n_2, \mathit{exportable}), (n_2, \mathit{session\_key}) \\
 & \\
\text{Import(sym):} & h(n_1, k_1), \big[\{\|k_2\|\}_{k_1}\big], (n_1, \mathit{session\_key}) \\
 & \xrightarrow{new\, n_2} h(n_2, k_2); (n_2, \mathit{exportable}), (n_2, \mathit{session\_key}) \\
 & \\
\text{Import(plain):} & \big[k_2\big] \xrightarrow{new\, n_2} h(n_2, k_2); \\
 & (n_2, \mathit{exportable}), (n_2, \mathit{session\_key}) \\
 & \\
\text{Import(public)} & \big[pub(k_2)\big] \xrightarrow{new\, n_2} h(n_2, pub(k_2)); \\
 & (n_2, \mathit{exportable}), (n_2, \mathit{session\_key}) \\
 & \\
\text{Import(private)} & \big[k_2\big] \xrightarrow{new\, n_p} h(n_p, k_p); \\
 & (n_2, \mathit{exportable}), (n_2, \mathit{exchange\_key})
\end{array}$$

Encrypt data (key): the function encrypts data (here data is key material) with the key k_2.

SEncrypt: $h(n_2, k_2), k_1; (n_2, session\_key) \rightarrow \{|k_1|\}_{k2}$
AEncrypt: $h(n_p, pub(k_p)), k_1; (n_2, exchange\_key) \rightarrow \{k_1\}_{pub(k_p)}$

Decrypt data (key): the function takes a key k_1 encrypted with k_2, and the handle to a decrypting key (that is k_2 if symmetric). If we start from a key blob, we only consider the section which contains encrypted key data. The encrypted data (key) is $\left[\{x\}_y\right] \rightarrow \{x\}_y$ (see next function).

SDecrypt: $h(n_2, k_2), \{|k_1|\}_{k_2}; (n_2, session\_key) \rightarrow k_1$
ADecrypt: $h(n_p, k_p), \{k_1\}_{pub(k_p)}; (n_2, exchange\_key) \rightarrow k_1$

Get encrypted key: the function (not included in CAPI) gets encrypted data from a key blob (i.e. select data from the 12^{th} byte position), where k_2 is the key encrypted under another key k_1 or k_p (if the blob is a *plaintextkeyblob*, no wrapping key is specified). We assume to get access to the encrypted key k_2 even if k_1 is a symmetric key (as already specified in model description).

Get_EKey: $\left[\{k_2\}_{k_1}\right] \rightarrow \{k_2\}_{k_1}$
Get_EKey: $\left[\{k_2\}\right] \rightarrow \{k_2\}$

GetUserKey: this function starts a search of a public/private key pair inside a CSP and returns a handle to it. We cannot use keys without handles.

GetUserKey: $k_p \xrightarrow{newn} h(n_p, k_p)$

Additional rules are defined to open and close a session:

$$\big\{T, L\big\} \xrightarrow{start-session} T, L$$

$$T', L' \xrightarrow{close-session} \big\{T'', L''\big\}$$

To have access to private objects, the normal user must log in and be authenticated (the state $\big\{\mathbf{T},\ \mathbf{L}\big\}$ means that the user is **not authenticated**). When the session is closed, all session keys are destroyed. We cannot start a new session if another one is already open or after a session key has been created (during cross-api attacks, we open a session in CAPI; in PKCS#11 no more session can be created, until the previous one is closed). It follows an example of an *archivable* private key, i.e. the key can be exported during session time.

$$\{\} \xrightarrow{start} emptysession$$

$$\text{Gen\_key\_pair:} \quad \xrightarrow{newn_{pe}} h(n_{ps}, k_{ps}), pub(k_{ps}); \\ (n_{ps}, exchange\_key), (n_{ps}, archivable)$$

$$h(n_{ps}, k_{ps}), pub(k_{ps}); (n_{ps}, exchange\_key), (n_{ps}, archivable) \\ \xrightarrow{close} \\ \{h(n_{ps}, k_{ps})\}$$

Inside a session, k_{ps} is archivable, i.e. the private key is exportable during the session. No external CSP condition, such as *is_private_exportable*, would affect this operation. When the session is closed, the key k_{ps} is no more archivable (then exportable).

Chapter 4

A model for CAPI & PKCS#11

4.1 Primitives

Asymmetric and symmetric encryption of a plain text x under the key y:

$\{x\}_y$ = aenc(x,y)
$\{\|x\|\}_y$ = senc(x,y)

For the property previously cited, if we get a key blob from CAPI, we assume to work directly with the encrypted key, i.e. $\{x\}_y$ or $\{\|x\|\}_y$, without considering the header.

4.2 Basic notation

The finite set of *function symbols* is Σ, with the arity function
ar: $\Sigma \to \mathbb{N}$, where $\Sigma \subset \mathbb{N}^n$ and n the number of arguments.
Plain terms (PT) is defined as:

$$\begin{aligned} t_1, t_2 = x \quad & x \in X \\ | = n \quad & n \in N \\ | = f(t_1 ... t_n) \quad & f \in \sum \quad and \quad ar(f) = n \end{aligned}$$

with N the set of keys, nonces and data values, and X the set of variables.

Key access is modeled with functions of arity 2.

$$\text{h: nonce} \times \text{key} \to \text{handle}$$

$h(n, k)$ is a handle n to the key k. Different handles to the same key allow user to assign more templates to keys (formally, a template is linked to a nonce).

To state a key property or set a boolean value to a key attribute, write $a(t, b)$, i.e. the boolean value b, *true* ($\top$) or *false* ($\bot$), of an attribute a is applied to the term (i.e. nonce) t. The terms b are constants, i.e. functions of arity 0. Formally:

$$\text{a: Nonce} \times \text{bool} \to \text{Attribute}$$

The ordered set of binary function symbols, the attributes, is A, disjoint from Σ:

$$A = a_1, ..., a_n$$

with a_1 = sensitive, a_2 = exportable, a_3 = modifiable, a_4 = encrypt, a_5 = decrypt, a_6 = wrap, a_7 = unwrap, a_8 = token (i.e. the key is permanently stored in the token). The set of attribute terms specify all the functions in the form $a(t, b)$.

$$AT = \{a(t, b) | a \in A, t \in P, b \in \{\top, \bot\}\}.$$

More in general, we consider a template T a set of *true* attributes:

$$a_i, ...a_j$$

The template is associated to a term t: the attributes which appear in the template have the corresponding boolean values set to true; the other ones are set to false.
To build a secure configuration [2], we should apply some fixes FX in addition to attribute templates (see the last chapter). For now we consider some basic default templates, evidencing all the vulnerabilities.

In PKCS#11 templates must be specified every time a key is created (or to search and manipulate objects), setting specific values for each attribute. The set T of standard templates is defined as:

$$T = \{t\_pkcs, t\_exchange, t\_public, t\_session\}$$

The default one, for every PKCS#11 key type, is the following:

$$\text{t\_pkcs} = \{exportable, modifiable\}$$

A PKCS#11 key has all the attributes set to false, except for *exportable* and *modifiable*.

In CAPI, templates are assigned to a public/private key pair using the key specification *at_keyexchange* or *at_signature*. We consider by default the corresponding template for exchange key pair:

$$\text{t\_exchange} = \{sensitive, encrypt, decrypt, wrap, unwrap, token\}$$

The template refers to the key pair, but each key has specific attributes: the public key is *wrap*, *encrypt* and *exportable*, the private is *unwrap*, *decrypt* and *sensitive*. Generally, a private key is not exportable. If the flag is set to true, only some CSPs allow the private key to be exported. Public key template is:

$$\text{t\_public} = \{exportable, encrypt, wrap\}$$

A CAPI session key, instead, has the following template:

$$\text{t\_session} = \{modifiable, encrypt, decrypt, wrap, unwrap\}$$

To assign a valid template *tp* ∈ T to the key handle *n*, use A(n,tp), i.e. $a_1(n, \top)$, ..., $a_n(n, \top)$.

System is described with a finite set of rules:

$$T; L \xrightarrow{new\tilde{n}} T'; L'$$

where T, T' ⊂ PT are plain terms, such as keys and nonces, and L, L' ⊂ AT are sets of attribute terms, i.e. conditions on attributes. T represents user's knowledge and L is the current system state before performing the attack.
From T and L we derive new sets of terms and attributes condition, so that:

$$x_1, x2 \in T \Rightarrow x_1, x_2 \in T'$$
$$y_1, y2 \in L \Rightarrow y_1, y_2 \in L'$$

where names(T ∪ L) = {} and names(T' $\bigcup$ L') ⊆ $\tilde{n}$.
User knowledge is updated to T' and system state satisfies L'. When T and L are evaluated, new $\tilde{n}$ means that names in $\tilde{n}$ must be replaced by names in T' and L' (e.g. during key generation).

4.3 Merge together

We consider a super model which describes both CAPI and PKCS#11 operations. Thanks to a good interoperability level, we can represent CAPI keys, then operations, through PKCS#11 notation. From the set of CAPI rules:

$$T; \Gamma \xrightarrow{new \tilde{n}} T'; \Gamma'$$

we consider a set of transformation functions *(tf)* which map CAPI rules into more general ones. The new model will result from the following rules:

$$[T]; [\Gamma] \xrightarrow{new \tilde{n}} [T']; [\Gamma']$$

Every *flag* corresponds to an attribute, and *ks* maps to a specific template (t_exchange or t_signature). In fact, if we create a key pair in CAPI, the private key is recognized in PKCS#11 with a set of attributes, which differ according to key specification. From CAPI, a type term (tt) is mapped into a plain term (pt):

$$tf(n, ks) \rightarrow A(n, ts), \text{ with } ts \in \mathrm{T}$$
$$tf(n, \lambda) \rightarrow a_1(n, \top) \ldots a_n(n, \top)$$

Each transformation is applied to a couple of type terms *tt*, to obtain a new set Γ' of attribute terms. CAPI uses the blob format to store keys, but the encrypted key can be easily extracted from a blob, then used to perform a decrypt or an unwrap. We assume to apply the function *Get_EKey()*, declared in the previous section.

$$\left[\{k_2\}_{k_1}\right] \rightarrow \{k_2\}_{k_1}$$

The new super model considers "extra" attributes, called *fixes* which can be added to enforce security configuration to any hardware devices. With fixes, attributes of a particular key are set off.

$$\{a_1(\ldots, \bot), a_2(\ldots, \bot), \ldots\}$$

The system, with the set of fix attributes FX, becomes:

$$[T]; [\Gamma] \xrightarrow{new \tilde{n}} [T']; [\Gamma']; FX$$

All the operations are executed inside sessions: after user has opened a session, an application has access to the token's public objects. Session keys, as the name suggests, are accessible during the session time: none of these keys, after the session has been closed, is permanently saved to memory.

4.4 New rules

New rules take in consideration the transformation function *tf*: from CAPI to SM we apply this function when required.

Gen_key: the function generates a random session key, and returns a handle to the key k_1 in n_1. If the key is extractable, the attribute *exportable* must be set to true. The key is *modifiable*, i.e. its attributes can be changed.

(CAPI)

$$\text{Gen\_key:} \quad \xrightarrow{new n_1} h(n_1, k_1);\ (n_1, exportable), (n_1, session\_key)$$

(SM)

$$\text{Gen\_key:} \quad \xrightarrow{new n_1} h(n_1, k_1);\ \text{exportable(n, } \top\text{)},\ A(n_1, \text{ t\_session}).$$

In CAPI, all the symmetric keys are session keys, i.e. they are available during session time. In the following example, we consider a key pair k_p, already stored in the token, and a session key generated with *Gen_key*. Session keys are destroyed when session is closed.

$$\begin{gathered}\{h(n_p, k_p); A(n_p, t\_exchange)\}\\ \xrightarrow{start-session}\\ h(n_p, k_p), A(n_p, t\_exchange).\end{gathered}$$

$$\text{Gen\_key:} \quad \xrightarrow{new n_1} h(n_1, k_1);\ \text{exportable(}n_1\text{, } \top\text{)},\ A(n_1, \text{ t\_session}).$$

$$\begin{gathered}h(n_p, k_p),\ h(n_1, k_1);\\ exportable(n_1, \top) A(n_1, t\_session), A(n_p, t\_exchange)\\ \xrightarrow{close-session}\\ \{h(n_p, k_p)\}.\end{gathered}$$

The key k_1 is destroyed, because it is a session key (*t_session*) and hence it has the attribute *token* set to false.

Gen_key_pair: the function generates a key pair. If the key pair is archivable, it is copied and the new copy is set *exportable*, and the attributes *token* and *sensitive* are unset. **(CAPI)**

$$\text{Gen\_key\_pair:} \quad \xrightarrow{new\, n_p} h(n_p, k_p), pub(k_p); (n_p, exchange\_key).$$

$$\begin{aligned}\text{Gen\_key\_pair:} \quad & \xrightarrow{new\, n_{ps}} h(n_{ps}, k_{ps}), pub(k_{ps}) \\ & (n_{ps}, exchange\_key), (n_{ps}, archivable)\end{aligned}$$

(SM)

$$\text{Gen\_key\_pair:} \quad \xrightarrow{new\, n_1} h(n_1, k_p), pub(k_p);\ A(n_1, \text{t\_exchange})$$

$$\begin{aligned}\text{Gen\_key\_pair:} \quad & \xrightarrow{new\, n_{ps}} h(n_{ps}, k_{ps}), pub(k_{ps}), h(n_p, k_{ps}), \\ & A(n_{ps}, t\_exchange), exportable(n_{ps}, \top), \\ & token(n_{ps}, \bot), sensitive(n_{ps}, \bot), \\ & A(n_p, t\_exchange)\end{aligned}$$

If a key pair is created with the flag *archivable*, it means that a copy of the key pair is maintained in the session and it becomes exportable (in general, a private key is not exportable outside the token, where it is stored).

Export/wrap: the function takes a handle to the key to be exported and, if requested, the handle to the encrypting key (it depends from the key blob type in which the key is exported to). If the blob is not encrypted, such as a public or plain text blob, no key is specified. The function returns encrypted data. The public key is referenced using the same handle n_p.
(CAPI)

$$\begin{aligned}\text{Export(asym):} \quad & h(n_p, pub(k_p)), h(n_2, k_2); (n_2, exportable), \\ & (n_p, exchange\_key) \to \left[\{k_2\}_{pub(k_p)}\right]\end{aligned}$$

$$\begin{aligned}\text{Export(sym):} \quad & h(n_1, k_1), h(n_2, k_2); (n_2, exportable), \\ & (n_1, session\_key) \to \left[\{\|k_2\|\}_{k_1}\right]\end{aligned}$$

To export the private key, we must consider the external condition *is_private_exportable*. Even if the flag *exportable* is set to true, a private key is exportable if the current CSP allows it. We consider the condition in which the private key is wrapped with a symmetric key (for CAPI and PKCS#11 interoperability). If *is_private_exportable* is set:

$$\begin{array}{ll}\text{Export(private):} & h(n_2, k_2), h(n_p, k_p); (n_p, exportable), \\ & (n_2, session\_key), \rightarrow \left[\{k_p\}_{k_2}\right]\end{array}$$

$$\begin{array}{ll}\text{Export(private):} & h(n_2, k_2), h(n_p, k_p); (n_p, archivable), \\ & (n_2, session\_key), \rightarrow \left[\{k_p\}_{k_2}\right]\end{array}$$

If the condition is unset, the private key is exportable only if it is archivable. Remember that the archivable key is a copy of a public/private key pair, which is not exportable after the session was closed. In SM we are exporting this copy.

(SM)

$$\begin{array}{ll}\text{Export(asym):} & h(n_p, pub(k_p)), h(n_2, k_2); exportable(n_2, \top), \\ & A(n_p, t\_exchange) \rightarrow \{k_2\}_{pub(k_p)}\end{array}$$

$$\begin{array}{ll}\text{Export(sym):} & h(n_1, k_1), h(n_2, k_2); exportable(n_2, \top), \\ & A(n_1, t\_session), \rightarrow \{\|k_2\|\}_{k_1}\end{array}$$

$$\begin{array}{ll}\text{Export(private):} & h(n_2, k_2), h(n_p, k_p); exportable(n_p, \top), \\ & A(n_2, t\_session) \rightarrow \{k_p\}_{k_2}\end{array}$$

$$\begin{array}{ll}\text{Export(private):} & h(n_2, k_2), h(n_p, k_p); exportable(n_p, \top), \\ & A(n_2, t\_session), token(n_p, \bot), sensitive(n_p, \bot) \\ & \rightarrow \{k_p\}_{k_2}\end{array}$$

The condition *if(is_private_exportable)*, in SM, is expressed with the attribute exportable. When, in PKCS#11, we get a handle to the private key, even if the key is created with the flag *exportable* in CAPI, the key can have the attribute set to true or false. Only private keys with the attribute *exportable* can be wrapped. Anyway, in PKCS#11, some tokens do not allow private keys to be created if the attribute exportable is set to true.

Import/unwrap: the function imports a cryptographic key. It takes encrypted key data and a handle to a decrypting key: it must be the symmetric key used to wrap the key or the private key of an exchange

key pair (i.e. the corresponding public key is used to encrypt the key).
(CAPI)

$$\begin{array}{ll} \text{Import(asym):} & h(n_p, k_p), \left[\{k_2\}_{pub(k_p)}\right]; (n_p, exchange\_key) \\ & \xrightarrow{new n_2} h(n_2, k_2); (n_p, exportable), \\ & (n_2, session\_key) \end{array}$$

$$\begin{array}{ll} \text{Import(sym):} & h(n_1, k_1), \left[\{\|k_2\|\}_{k_1}\right]; (n_1, session\_key) \\ & \xrightarrow{new n_2} h(n_2, k_2); (n_2, exportable), \\ & (n_2, session\_key) \end{array}$$

(SM)

$$\begin{array}{ll} \text{Import(asym):} & h(n_p, k_p), \{k_2\}_{pub(k_p)}; A(n_p, t\_exchange) \\ & \xrightarrow{new n_2} h(n_2, k_2); exportable(n_2, \top), \\ & A(n_2, t\_session) \end{array}$$

$$\begin{array}{ll} \text{Import(sym):} & h(n_1, k_1), \{\|k_2\|\}_{k_1}; A(n_2, t\_session) \\ & \xrightarrow{new n_2} h(n_2, k_2); exportable(n_2, \top), \\ & A(n_2, t\_session) \end{array}$$

Encrypt data (key): the function encrypts data (here data is key material) with the key k_2.
(CAPI)

$$\begin{array}{ll} \text{SEncrypt:} & h(n_2, k_2), k_1; (n_2, session\_key) \rightarrow \{|k_1|\}_{k2} \\ \text{AEncrypt:} & h(n_p, pub(k_p)), k_1; (n_p, exchange\_key) \rightarrow \{k_1\}_{pub(k_p)} \end{array}$$

(SM)

$$\begin{array}{ll} \text{SEncrypt:} & h(n_2, k_2), k_1; A(n_2, t\_session) \rightarrow \{|k_1|\}_{k2} \\ \text{AEncrypt:} & h(n_p, pub(k_p)), k_1; A(n_p, t\_exchange) \rightarrow \{k_1\}_{pub(k_p)} \end{array}$$

Decrypt data (key): the function takes key data k_1, encrypted with k_2, and the handle to a decrypting key (that is $h(n_2, k_2)$ if symmetric, or the corresponding private key if k_2 is public).
(CAPI)

$$\begin{array}{ll} \text{SDecrypt:} & h(n_2, k_2), \{|k_1|\}_{k_2}; (n_2, session\_key) \rightarrow k_1 \\ \text{ADecrypt:} & h(n_p, k_p), \{k_1\}_{pub(k_p)}; (n_p, exchange\_key) \rightarrow k_1 \end{array}$$

(SM)

$$\begin{array}{ll}\text{SDecrypt:} & h(n_2, k_2), \{|k_1|\}_{k_2}; A(n_2, t\_session) \to k_1 \\ \text{ADecrypt:} & h(n_p, k_p), \{k_1\}_{pub(k_p)}; A(n_p, t\_exchange) \to k_1\end{array}$$

Create object: the function creates a public key object starting from raw data, i.e. public information (modulus, public exponent) of a key pair.

$$\text{CreateObject:} \quad \text{pub}(k_p) \xrightarrow{new\, n_1} h(n_1, k_1),\ A(n_1,\ \text{t\_public})$$

Create key blob: the function takes key data k_1, encrypted with a key k_2 (or not), and returns a CAPI key blob. This structure is necessary to import PKCS#11 keys in CAPI.

$$\begin{array}{ll}\text{CreateBlob:} & \{k_2\}_{pub(k_p)} \to \big[\{k_2\}_{pub(k_p)}\big] \\ \text{CreateBlob:} & \{k_2\} \to \big[\{k_2\}\big]\end{array}$$

Get encrypted key:

$$\begin{array}{ll}\text{Get\_EKey:} & \big[\{k_2\}_{k_1}\big] \to \{k_2\}_{k_1} \\ \text{Get\_EKey:} & \big[\{k_2\}\big] \to \{k_2\}\end{array}$$

4.5 Correctness of the model

Our intent is to formalize every CAPI operation with a super model, which uses PKCS#11 syntax. Starting from the model of CAPI

$$T; \Gamma \xrightarrow{new\, \tilde{n}} T'; \Gamma'$$

we apply suitable transformation functions (identified with square brackets)

$$[T]; [\Gamma] \xrightarrow{new\, \tilde{n}} [T']; [\Gamma']$$

Every transformation must map a state (i.e. a sequence of operations) from CAPI to the super model SM. To prove the correctness of the model, a valid state in CAPI must be coherent in SM, i.e. all the operations performed in CAPI must be replicable in SM.

Formally:

$$\forall \text{ n,k} \in \text{T, h(n,k)} \in \text{T, } f_1 \dots f_n \in \Gamma,\ ks_1 \dots ks_n \in \Gamma$$

$$\Longrightarrow$$

$$\text{n,k} \in \big[T\big],\ \text{n,k} \in \big[T\big],\ tf(f_1) \dots tf(f_n) \in \big[\Gamma\big],\ tf(ks_1) \dots tf(ks_n) \in \big[\Gamma\big]$$

We use the following transformation functions:

- **tf(n_1,session_key)** $\rightarrow A(n_1, t\_session)$
- **tf(n_1,exchange_key)** $\rightarrow A(n_1, t\_exchange)$
- **tf(n_1,signature_key)** $\rightarrow A(n_1, t\_signature)$
- **tf(n_1,exportable)** $\rightarrow exportable(n_1, \top)$
- **tf(n_1,archivable)** $\rightarrow$ exportable(n_1,$\top$), token(n_1, $\bot$), $sensitive(n_1, \bot)$

Then we must prove that a sequence of state $S_1 \dots S_n$ in CAPI is coherent in SM: in both the model we must be able to carry out the same operations.

$$\begin{gathered} S_1 \rightarrow S_2 \cdots \rightarrow S_n \\ \big[T\big];\big[\Gamma\big] \rightarrow \big[T'\big];\big[\Gamma'\big] \cdots \rightarrow \big[T^n\big];\big[\Gamma^n\big] \end{gathered}$$

$$\Longrightarrow$$

$$\begin{gathered} S_1 \rightarrow S_2 \cdots \rightarrow S_n \\ T;L \rightarrow T';L' \cdots \rightarrow T^n;L^n \end{gathered}$$

Some key roles can be expressed more in detail by the SM model: for example, the archivable property is expressed with three attributes in PKCS#11. Still more, if we create an archivable key in CAPI, in PKCS#11 we find two keys (a session key and a permanent key), while in CAPI we have only one handle.
To prove the implication, we first how all the necessary transformations from CAPI to SM, to obtain the system

$$T;L \xrightarrow{new\tilde{n}} T';L'$$

and then we give an example of state consistency.

Gen_Key (CAPI):

Gen_key: $\xrightarrow{new n_1} h(n_1, k_1);\ (n_1, \mathit{exportable}), (n_1, \mathit{session\_key})$

We apply these transformation functions:

tf(n_1**,session_key)** $\rightarrow A(n_1, \mathit{t\_session})$
tf(n_1**,exportable)** $\rightarrow \mathit{exportable}(n_1, \top)$

Gen_Key (SM):

Gen_key: $\xrightarrow{new n_1} h(n_1, k_1)$; exportable(n, $\top$), $A(n_1$, t_session).

Gen_Key_Pair (CAPI):

$$\begin{aligned} \text{Gen\_key\_pair:}\quad & \xrightarrow{new n_{ps}} h(n_{ps}, k_{ps}), pub(k_{ps}) \\ & (n_{ps}, \mathit{exchange\_key}), (n_{ps}, \mathit{archivable}) \end{aligned}$$

We apply these transformation functions:

tf(n_{ps}**,exchange_key)** $\rightarrow A(n, \mathit{t\_exchange})$
tf(n_{ps}**,archivable)** $\rightarrow \mathit{exportable}(n, \top), \mathit{token}(n, \bot), \mathit{sensitive}(n, \bot)$

Gen_Key_Pair (SM):

$$\begin{aligned} \text{Gen\_key\_pair:}\quad & \xrightarrow{new n_{ps}} h(n_{ps}, k_{ps}), pub(k_{ps}), h(n_p, k_{ps}), pub(k_{ps}), \\ & A(n_{ps}, \mathit{t\_exchange}), \mathit{exportable}(n_{ps}, \top), \\ & \mathit{token}(n_{ps}, \bot), \mathit{sensitive}(n_{ps}, \bot), \\ & A(n_p, \mathit{t\_exchange}) \end{aligned}$$

In SM we have two handle to the key k_{ps}: one is a session key (the archivable key), the second, which is not handled in CAPI. is stored in the device (the key has the attribute *token* set to true). *Gen_key_pair* rule also applies to all key pairs.

Export(asym) (CAPI):

$$\begin{aligned} \text{Export(asym):}\quad & h(n_p, pub(k_p)), h(n_2, k_2); (n_p, \mathit{exchange\_key}), \\ & (n_2, \mathit{exportable}) \rightarrow \left[\{k_2\}_{pub(k_p)}\right] \end{aligned}$$

We apply the following transformation functions:

tf(n_p**, exchange_key)** $\rightarrow A(n_p, t\_exchange)$
tf(n_2**, exportable)** $\rightarrow exportable(n_2, \top)$

Export(asym) (SM):

$$\text{Export(asym):} \quad h(n_p, pub(k_p)), h(n_2, k_2); A(n_p, t\_exchange), exportable(n_2, \top) \rightarrow \{k_2\}_{pub(k_p)}$$

In CAPI, the result of *export* is a key blob. In SM, we consider the encrypted (wrapped) key without the key blob structure (essentially, we have a key blob without the header). We implicitly apply the function *Get_EKey()*, from CAPI to SM model. The two exported formats are considered without difference in meaning. *Export()* rule also applies to symmetric keys .

Export(private) (CAPI):

$$\text{Export(private):} \quad h(n_2, k_2), h(n_p, k_p); (n_2, session\_key), (n_p, archivable) \rightarrow \left[\{k_p\}_{k_2}\right]$$

We apply these transformation functions:

tf(n_p**, archivable)** $\rightarrow exportable(n_p, \top), token(n_p, \bot), sensitive(n_p, \bot)$
tf(n_2**, session_key)** $\rightarrow A(n_2, t\_session)$

Export(private) (SM):

$$\text{Export(private):} \quad h(n_2, k_2), h(n_p, k_p); A(n_2, t\_session), exportable(n_p, \top), token(n_p, \bot), sensitive(n_p, \bot) \rightarrow \{k_p\}_{k_2}$$

Export(private) rule also applies to private not-archivable keys.

Import(asym) (CAPI):

$$\text{Import(asym):} \quad h(n_p, k_p), \left[\{k_2\}_{pub(k_p)}\right]; (n_p, exchange\_key) \xrightarrow{new\, n_2} h(n_2, k_2); (n_2, exportable), (n_2, session\_key)$$

We apply these transformation functions:

tf(n_2**, exportable)** $\rightarrow$ $exportable(n_2, \top)$
tf(n_p**, exchange_key)** $\rightarrow A(n_p, t\_exchange)$
tf(n_2**, session_key)** $\rightarrow A(n_2, t\_session)$

Import(asym) (SM):

$$\begin{array}{ll}\text{Import(asym):} & h(n_p, k_p), \{k_2\}_{pub(k_p)} \xrightarrow{newn_2} h(n_2, k_2); \\ & A(n_2, t\_session), A(n_p, t\_exchange), \\ & exportable(n_2, \top)\end{array}$$

Import() rule also applies to symmetric keys .

AEncrypt (CAPI):

$$\text{AEncrypt:} \quad h(n_p, pub(k_p)), k_1; (n_p, exchange\_key) \rightarrow \{k_1\}_{pub(k_p)}$$

We apply these transformation functions:

tf(n_p**, exchange_key)** $\rightarrow A(n_p, t\_exchange)$

AEncrypt (SM):

$$\text{AEncrypt:} \quad h(n_2, pub(k_p)), k_1; A(n_2, t\_exchange) \rightarrow \{k_1\}_{pub(k_p)}$$

Encrypt() rule also applies to symmetric encryption .

ADecrypt (CAPI):

$$\text{ADecrypt:} \quad h(n_p, k_p), \{k_1\}_{pub(k_p)}; (n_p, exchange\_key) \rightarrow k_1$$

We apply these transformation functions:

tf(n_p**, exchange_key)** $\rightarrow A(n_p, t\_exchange)$

ADecrypt (SM):

$$\text{ADecrypt:} \quad h(n_p, k_p), \{k_1\}_{pub(k_p)}; A(n_p, t\_exchange) \rightarrow k_1$$

Encrypt() rule also applies to symmetric decryption.

The second proof is about state consistency: we consider the wrap decrypt attack (described more in detail in the next chapter): a key k_1 is encrypted with k_p and then exported. The key k_1 must be exportable, k_p must have the flag/attribute encrypt set to true (according to the function *Export(asym)* and *ADecrypt* previously defined). In CAPI, the attack is defined as follows (state by state; S_i denotes a state):

S_1	$\{\ \}$
S_2 (Gen_key_pair)	$h(n_p, k_p), pub(k_p); (n_p, t\_exchange)$
S_3 (Gen_key)	$S_2 + h(n_1, k_1); (n_1, t\_session), (n_1, exportable)$
S_4 (Export(asym))	$S_3 + h(n_p, pub(k_p)), h(n_1, k_1) \rightarrow \left[\{k_1\}_{pub(k_p)}\right]$
S_5 (ADecrypt)	$S_4 + h(n_p, k_p), \{k_1\}_{pub(k_p)} \rightarrow k_1$

The key k_p is encrypting (it is an exchange key - S_2) and k_1 is exportable (it is a session key - S_3). With SM the sequence becomes:

S_1	$\{\ \}$
S_2 (Gen_key_pair)	$h(n_p, k_p), pub(k_p); A(n_p, t\_exchange)$
S_3 (Gen_key)	$S_2 + h(n_1, k_1); A(n_1, t\_session), exportable(n_1, \top)$
S_4 (Export(asym))	$S_3 + h(n_3, pub(k_p)), h(n_1, k_1) \rightarrow \{k_1\}_{pub(k_p)}$
S_5 (ADecrypt)	$S_4 + h(n_p, k_p), \{k_1\}_{pub(k_p)} \rightarrow k_1$

The key k_p is again encrypting (we apply a t_exchange template - S_2) and k_1 is exportable (we apply a t_session template - S_2). Maintaining the same handles, we can perform the final decryption of k_1, that is our aim.

4.6 PKCS#11 operations

PKCS#11 model and key management in [5, 6] has been extended according to the previous definition of super model (SM).

Generate key pair: generates public and private key pair with a specific template. The private key must be *sensitive*, and, if the attribute *cka_token* is set to *true*, the key will be accessible via CAPI.

$$\text{Gen\_key\_pair:}\quad \xrightarrow{new\, n_p} h(n_p, k_p), pub(k_p);\ A(n_p,\ \text{t\_pkcs}),\ \text{sensitive}(n_p,\ \top)$$

Gen_key: generate a symmetric key.

$$\text{Gen\_key:}\quad \xrightarrow{new\, n_1} h(n_1, k_1);\ A(n_1,\ \text{t\_session})$$

Wrap key: this operation corresponds to *Export key* in CAPI, but a key blob structure is not created.

$$\text{Wrap(asym):}\quad h(n_p, pub(k_p)), h(n_2, k_2);\ \text{wrap}(n_p, \top),\ \text{exportable}(n_2, \top) \rightarrow \{k_2\}_{pub(k_p)}$$

$$\text{Wrap(sym):}\quad h(n_1, k_1), h(n_2, k_2);\ \text{wrap}(n_1, \top),\ \text{exportable}(n_2, \top) \rightarrow \{|k_2|\}_{k_1}$$

Unwrap key: this operation corresponds to *Import key* in CAPI, but here the function returns a string (or encrypted string) instead of a blob structure.

$$\text{Unwrap(asym):}\quad h(n_p, k_p), \{k_2\}_{k_1};\ \text{unwrap}(n_p, \top) \xrightarrow{new\, n_2} h(n_2, k_2);\ A(n_2,\ \text{t\_pkcs})$$

$$\text{Unwrap(sym):}\quad h(n_1, k_1), \{|k_2|\}_{k_1};\ \text{unwrap}(n_1, \top) \xrightarrow{new\, n_2} h(n_2, k_2);\ A(n_2,\ \text{t\_pkcs})$$

Encrypt key: similar to CAPI; specify if to use a public key.

$$\text{SEncrypt:}\quad h(n_2, k_2), k_1; encrypt(n_2, \top) \rightarrow \{|k_1|\}_{k2}$$
$$\text{AEncrypt:}\quad h(n_p, pub(k_p)), k_1;\ \text{encrypt}(n_p,\ \top) \rightarrow \{k_1\}_{pub(k_p)}$$

Decrypt key: similar to CAPI; specify if to use a private key.

$$\begin{array}{ll} \text{SDecrypt:} & h(n_2, k_2), \{|k_1|\}_{k_2};\ \text{decrypt}(n_2,\ \top) \rightarrow k_1 \\ \text{ADecrypt:} & h(n_p, k_p), \{k_1\}_{k_2};\ \text{decrypt}(n_p,\ \top) \rightarrow k_1 \end{array}$$

Set or unset attribute: these functions allow to modify some key attributes. CAPI private key permissions are not modifiable via PKCS#11. If attribute *modifiable* is set, attributes cannot be changed. Other restrictions (see next paragraph) refer to conflicting attributes, sticky_on and sticky_off attributes.

$$\begin{array}{ll} \text{Set\_Attribute:} & h(n_1, k_1); a(n_1, \bot), \rightarrow a(n_1, \top) \\ \text{Unset\_Attribute:} & h(n_1, k_1); a(n_1, \top), \rightarrow a(n_1, \bot) \end{array}$$

Later on, when we talk about fixes and solutions to attacks, we will consider restrictions about conflicting attributes and sticky_on and sticky_off attributes.

Chapter 5

Attacks

Clulow and Bortolozzo et al. [2, 4] give some demonstrations on how to perform attacks on PKCS#11 sensitive keys, exporting them and then reading the content of the key, in clear, outside the device. The operations take advantage of the conflicting roles assigned to keys, such as *wrap* and *decrypt* or *encrypt* and *unwrap*.

By the use of Tookan [3], a tool for PKCS#11 token analysis, our token was discovered vulnerable to the wrap decrypt attack, i.e. a sensitive key k_1 is wrapped with a key k_2 with the attributes *wrap* and *decrypt* set on, and then, once extracted, it is decrypted with the same key. A first remark to be done is about the wrapping format of the exported keys: CAPI allows keys to be exported into a key blob, a structure which contains the encoded key. However, the blob format can be changed favoring a way to read the wrapped key in clear.

In all the attacks (we maintain the names of the attacks described in the paper [2]), the token is assumed to be connected to a host under the control of an intruder who knows the secret PIN and can have access to the whole set of API operations. User is supposed to be able to read an encrypted key only if he knows the correct key, and cannot crack the encryption algorithm by brute-force attempts or similar means.

5.1 PKCS#11 attacks simulated with CAPI

In this section we try to replicate vulnerabilities, found in PKCS#11, with CAPI functions. A first important remark should be done: in CAPI no session key is sensitive and can be stored in the token. Session keys can be stored outside the device into (protected) key blobs: in the next chapter we will describe some cross-attacks (from CAPI to PKCS#11 or vice-versa) that can be done using such a structure. Now we give demonstration that CAPI has no restrictions on performing **conflict operations**, such as wrap/decrypt: these vulnerabilities will reflect on further attacks.

Keys are represented as $h(n_i, k_i)$: n_i is a handle for the key k_i, where k_1 is usually a sensitive key stored in the device, for which we want to discover the secret value. The intruder is given some initial knowledge, such as some keys k_i, or keys wrapped with other keys $k_{i_{k_j}}$.

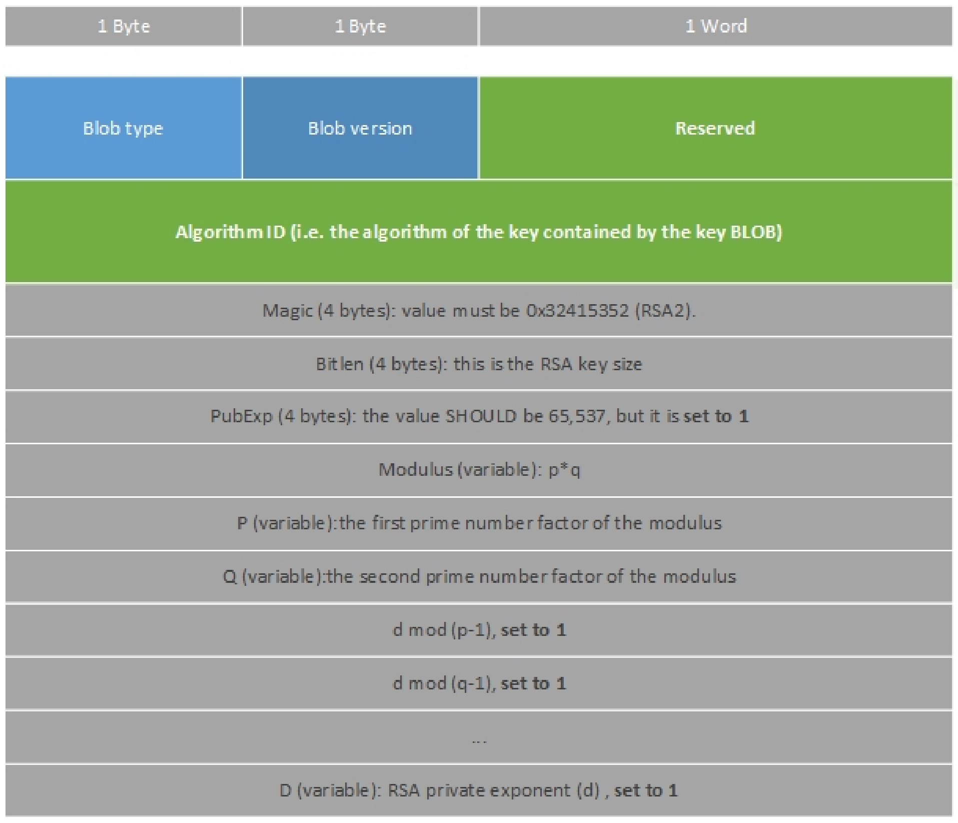
5.1.1 Wrap decrypt attack

An intruder knows the sensitive and extractable key k_1 and the wrap/decrypt key k_2. Excluding a symmetric wrap (we mainly test CAPI with asymmetric wrap/decrypt, for simplicity in operating with simple key blob structures), we consider a RSA private/public key pair k_2: a user can get it from a public certificate (already included in the token), generate a new RSA key pair or derive a *one exponent private/public key pair* [1]. This last key let the intruder export k_1 "as is", saving it into a simple key blob; k_1 is not really encrypted because of the wrapping key exponent of one. The token allows to create (or import) new keys, so the *one exponent key* can always be a valid key. A public key is, by default, always exportable, and it can perform wrap (encryption) and decryption. The *one exponent* key is created starting from a private key blob of a key pair generated by another software provider, e.g. the Microsoft Enhanced Provider, included in every Windows OS, and then it is successfully imported into the token.

The wrap/decrypt attack is simply:

$$\begin{array}{ll}
\text{Export(asym):} & h(n_2, pub(k_2)), h(n_1, k_1); \\
 & (n_1, exportable), (n_2, exchange\_key) \\
 & \rightarrow \left[\{k_1\}_{pub(k_2)}\right] \\
 & \\
\text{Get\_EKey:} & \left[\{k_1\}_{pub(k_2)}\right] \rightarrow \{k_1\}_{pub(k_2)} \\
 & \\
\text{ADecrypt:} & h(n_2, pub(k_2)), \{k_1\}_{pub(k_2)} \\
 & (n_2, exchange\_key) \rightarrow k_1
\end{array}$$

If we perform this attack with a *one exponent* key, the private key must be generated and then exported by a different CSP, because the token provider does not allow any private key to be exported (as we have remarked, it is sensitive and not exportable); then the key blob is modified to change the exponent of the key (refer to the following picture to look which bytes should be modified). Different CSP do not generate conflicts or errors during these operations. To prevent the



1 Byte	1 Byte	1 Word
Blob type	Blob version	**Reserved**
Algorithm ID (i.e. the algorithm of the key contained by the key BLOB)		
Magic (4 bytes): value must be 0x32415352 (RSA2).		
Bitlen (4 bytes): this is the RSA key size		
PubExp (4 bytes): the value SHOULD be 65,537, but it is **set to 1**		
Modulus (variable): p*q		
P (variable):the first prime number factor of the modulus		
Q (variable):the second prime number factor of the modulus		
d mod (p-1), **set to 1**		
d mod (q-1), **set to 1**		
...		
D (variable): RSA private exponent (d) , **set to 1**		

Figure 5.1: RSA private key blob

wrap decrypt attack in PKCS#11, we should avoid a key to be at the same time *wrap* and *decrypt*, adding these attributes to the set of conflicting ones. In CAPI we cannot set permissions to public/private key pair: basically every key pair can wrap (i.e. encrypt) and decrypt data. With one exponent private/public key, decryption is not necessary at all, so any conflicting role is absent.

5.1.2 Encrypt unwrap attack

The intruder knows the handles $h(n_1, k_1)$, with k_1 set sensitive and extractable, and $h(n_2, k_2)$, with k_2 a private/public key (the private has encrypt and unwrap set on), and the plain-text key k_3. Using the one exponent key pair, it is still possible to read the sensitive key. With the unwrap operation, the attack becomes the following:

Initial knowledge: $h(n_1, k_1)$ session key handle, $h(n_2, k_2)$ public/private key pair handle, k_3 key.

AEncrypt: $h(n_2, pub(k_2)), k_3; (n_2, exchange\_key)$
$\rightarrow k_{3pub(k_2)}$

CreateBlob: $k_{3pub(k_2)} \rightarrow \left[\{k_3\}_{pub(K_2)}\right]$

Import(asym): $h(n_2, k_2),\ \left[\{k_3\}_{pub(K_2)}\right], (n_2, exchange\_key)$
$\xrightarrow{new n_3} h(n_3, k_3); A(n_3, t\_session)$
$exportable(n_3, \top)$

A new key has been imported after it was encrypted into a simple key blob; then we can read k_1 value after having wrapped the key with k_3 or another *one exponent* key.

Listing 5.1: create a simpleblob key (pseudo code)

```
[...]
((blobheader*)pbPtr)->bType = simpleblob;
((blobheader*)pbPtr)->bVersion = 2;
((blobheader*)pbPtr)->reserved = 0;
((blobheader*)pbPtr)->aiKeyAlg = Algid;
dwDataLen = sizeof(ALG_ID);

//Place the encrypted key in reverse order (little-endian format)
[...]
//The key is now imported (k3)
CryptImportKey(
        token provider,
        pbData,
        datalength,
        private key,
        crypt_exportable,
        key handle
);
```

The key, once encrypted, is suitably inserted into a simpleblob structure. The simple blob structure is shown as follows:

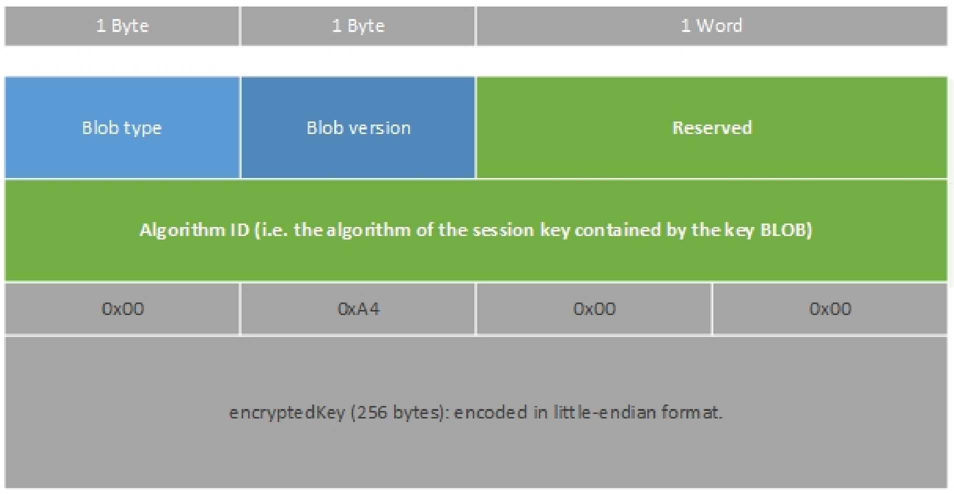

1 Byte
1 Byte
1 Word
Blob type
Blob version
Reserved
Algorithm ID (i.e. the algorithm of the session key contained by the key BLOB)
0x00
0xA4
0x00
0x00
encryptedKey (256 bytes): encoded in little-endian format.

Figure 5.2: Simplekeyblob

In PKCS#11, to prevent this attack, the attributes *encrypt* and *unwrap* are considered conflicting. In CAPI, we propose the attack using a public/key pair, to operate with the simplest key blob format. Public and private key permissions cannot be set, so it is always possible to perform this kind of operations. Anyway, in the following attacks, we will not consider keys with such conflicting attributes.

5.1.3 Re import attack 1

Without using the same key as in the previous attacks to wrap and decrypt, a key k_2 is first wrapped under k_2 itself or another key, then it is unwrapped obtaining a different handle. The same key is viewed as two different keys, which are used to wrap and then decrypt. In CAPI k_2 should be a session key or a private/public key pair. The intruder knows the handles $h(n_1, k_1)$, with k_1 set sensitive and extractable, and $h(n_2, k_2)$, set extractable.

Initial knowledge: $h(n_1, k_1)$ is a session key handle and $h(n_2, k_2)$ a public/private key pair handle.

Export(public):	$h(n_2, k_2); (n_2, \mathit{exportable}) \rightarrow \big[\mathit{pub}(k_2)\big]$
Import(public)	$\big[\mathit{pub}(k_2)\big] \xrightarrow{new n_3} h(n_3, \mathit{pub}(k_2)); (n_3, \mathit{exportable}), (n_3, \mathit{session\_key})$
Export(asym):	$h(n_2, \mathit{pub}(k_2)), h(n_1, k_1); (n_1, \mathit{exportable}), (n_2, \mathit{exchange\_key}) \rightarrow \big[\{k_1\}_{\mathit{pub}(k_2)}\big]$
Get_EKey:	$\big[\{k_1\}_{\mathit{pub}(k_2)}\big] \rightarrow \{k_1\}_{\mathit{pub}(k_2)}$
ADecrypt:	$h(n_3, k_2), \{k_1\}_{\mathit{pub}(k_2)}; (n_3, \mathit{exchange\_key}) \rightarrow k_1$

In PKCS#11, key k_2 should be wrapping and unwrapping. The attack can be prevented adding these attributes as conflicting. In CAPI it has again no effects!

5.1.4 Re import attack 2

Giving the intruder an encrypted key $k_{3_{k_2}}$, (in CAPI it must be encapsulated into a simpleblob or symmetricwrapkeyblob, encoded under k_2 key), we can unwrap twice this key, obtaining two key, one to wrap and the other to decrypt. The intruder knows the handles $h(n_1, k_1)$, with k_1 set sensitive and extractable, $h(n_2, k_2)$, set extractable, and k_{3k2} (in a blob structure). We consider the simplest case of simpleblob, with k_2 a private/public key.

Initial knowledge: $h(n_1, k_1)$ is a session key handle, $h(n_2, k_2)$ a public/private key pair handle and k_{3k2} an encrypted key. k_3 is a raw key.

Import(asym):	$h(n_2, k_2), \left[\{k_3\}_{k2}\right]; (n_2, \mathit{exchange\_key}) \xrightarrow{new n_3} h(n_3, k_3); A(n_3, t\_session)$
Import(asym):	$h(n_2, k_2), \left[\{k_3\}_{k2}\right]; (n_2, \mathit{exchange\_key}) \xrightarrow{new n_4} h(n_4, k_3); A(n_4, t\_session)$
Export(sym):	$h(n_1, k_1), h(n_3, k_3); (n_1, \mathit{session\_key})$ $\mathit{exportable}(n_1, \top) \rightarrow \left[\{\|k_1\|\}_{k_3}\right]$
Get_EKey:	$\left[\{\|k_1\|\}_{k_3}\right] \rightarrow \{\|k_1\|\}_{k_3}$
SDecrypt:	$h(n_4, k_3), \{\|k_1\|\}_{k_3} \rightarrow k_1$

An alternative to a symmetric blob decryption expects to import again the blob to get a new handle to k_1; then we export the key into a plain text blob. We can always do it in CAPI.

Import(sym):	$h(n_3, k_3), \left[\{\|k_1\|\}_{k_3}\right]; (n_3, \mathit{session\_key}) \xrightarrow{new n_5} h(n_5, k_1); A(n_5, t\_session)$ $\mathit{exportable}(n_5, \top)$
Export(plain):	$h(n_5, k_1); (n_5, \mathit{exportable}) \rightarrow \left[k_1\right]$

5.2 Notes

Vulnerabilities found in PKCS#11 are similar to the ones found in CAPI devices. While the first standard allows to set parameters to define specific key behaviors, CAPI has an option to set permissions to session keys, but there is no effect, being them volatile and not persistent to memory. Session keys has no protection, except if they are declared "not-exportable": in this way, they cannot be wrapped under any key. CAPI defines specific exporting formats, the key blobs: these structures allow at the same time a good manipulation (we saw how to create "ad hoc" simple or plain text key blob to encapsulate a known key) and a good protection, such as the symmetric key blob, hard to decrypt manually, but anyway it follows a standard. PKCS#11 does not provide these kind of structures; keys, when wrapped (then exported), are raw strings. We often use a simple key blob to simplify the decryption operation, once a key has been wrapped. The use of the *one exponent key* derives from the vulnerability of the token (vulnerable to a_2 attack; see [6]); any operation which aims to discover a key value can use this key without a final decryption.

Session keys are volatile, but they can still be saved in the application layer, differently from PKCS#11 devices which allows symmetric keys to be stored inside. Attacks will be performed on that structures, using functions of the two standards alternately and vulnerabilities we have found.

Chapter 6

Implementation with CAPI

The implementation follows a top-down approach, in order to highlight some essential phases of the attacks that we proposed. The code is written in C++ using Visual Studio 2012. All the cryptographic functions we use are included in Windows libraries: no external files are necessary. To communicate with hardware devices, we install the requested drivers. For each application, we suppose to have acquired the context of the token and its cryptographic provider, using the function *CryptAcquireContext*(). The Token is password protected, and before executing the first operation in the device it is necessary to type the PIN. The function CryptSetProvParam() with the flag PP_KEYEXCHANGE_PIN provides the PIN when requested.

Listing 6.1: acquire context

```
//provider name is specified in the token documentation
LPCTSTR provider = _T(token Cryptographic Provider);

//token is the name of key container.
//if keys are created via PKCS11, key container is
//"*********", with x >=0

//if no key inside the token, set CRYPT_NEWKEYSET
CryptAcquireContext(
        &tokenProv,
        _T("token"),
        provider,
        PROV_RSA_FULL,
        0)

//set user PIN
```

```
CryptSetProvParam(
        tokenProv,
        PP_KEYEXCHANGE_PIN,
        (PBYTE)pin,
        0)
```

A fist step is about key generation, provided by the function CryptoGenKey(). The function requires a value (ALG_ID) that identifies the algorithm to generate a key:

- a session key is DES, 3DES, etc.
- a public/private key pair is *AT_KEY_EXCHANGE*, if used to export (encrypt) and import (decrypt) session keys, i.e. to exchange them among users, or AT_SIGNATURE, if used for signature and authentication.

The algorithm depends on the provider: for tests with the token provider, 3DES is chosen for symmetric keys, CALG_RSA_KEYX (RSA public key exchange) for public and private keys. No flag can be used (or has no effect with the token provider) to set the key "sensitive" or not. Every key we create is exportable, otherwise no operation that expects to export (or wrap) a key could be done. This flag applies only to session key and private key blobs: if a session key is set not exportable, it becomes available only for the application that generated it, i.e. for the current session, and cannot be exchanged. Public keys are always exportable by default.

Listing 6.2: generate key

```
//AT_KEYEXCHANGE identifies the algorithm CALG_RSA_KEYX
//An exchange public/private key pair is created
CryptGenKey(tokenProv,AT_KEYEXCHANGE,flag,&k3)

//flag must be set to CRYPT_EXPORTABLE if the key is to
//export
```

Unlike the standard PKCS#11, we do not set key permissions to determine key roles, e.g. wrap, unwrap, decrypt or encrypt key. The function to set parameters is CryptoSetKeyParam(), available only for session keys [10]; values set by this function are not persisted to memory and can only be used within a single session. A key can be set to have a different encryption mode (the default is CBC), or contain a salt (KP_PADDING) used by the cipher or declare a different initialization vector (KP_IV), to be used in CBC. The symmetric function to get key parameters is CryptoGetKeyParam(); it is used in the function to import a key raw into the token, to get the necessary information to build a new key blob.

Listing 6.3: get key info and build a key blob

```
//a key blob needs private key's algorithm and key length
//to be set in header

// Get private key's algorithm
dwSize = sizeof(ALG_ID);
fResult = CryptGetKeyParam(hpKey,
         KP_ALGID, //get alg ID
         (LPBYTE)&dwPrivKeyAlg,
         &dwSize,
         0
);

// Get private key's length in bits
dwSize = sizeof(DWORD);
fResult = CryptGetKeyParam(
         hpKey,
         KP_KEYLEN,
         (LPBYTE)&dwPublicKeySize,
         &dwSize,
         0
);

// calculate Simple blob's length
dwSessionBlob = (dwPublicKeySize/8) + sizeof(ALG_ID)
 + sizeof(BLOBHEADER);

//create the blob
[...]
```

To perform the first attack, the wrap/decrypt attack, we suppose to have a symmetric key k_1 and a public/private key pair k_2. The first function that we call is CryptExportKey(): it requires a handle of the key to be exported, e.g. n_1, a blob type and, if requested, a handle of a key used to wrap (encrypt) the first key. If the blob type is a *publickeyblob*, no key is requested, because the public key will be exported without any encryption. A *privatekeyblob* cannot be obtained with the token provider, because the private key is set to sensitive and not exportable. A blob to be used with a session key and a private/public key pair is a *simpleblob*. Specifying k_2 as the encrypted key, k_1 will be exported encrypted, i.e. wrapped, with the public key. At this point, k_1 is stored in a key blob outside the CSP, starting from the 12^{th} byte of the structure. To perform a decryption, the key is temporary saved into a data array, then it becomes an input to the decrypt function CryptDecrypt(). The CryptDecrypt() function decrypts data using the private k_2.

Listing 6.4: decrypt a wrapped key

```
CryptExportKey(k1,k2,SIMPLEBLOB,0,keyBlob,&dwBlobLength))
//save the key in keyDecrypt
keyDecrypted = &keyBlob[12];
//update keylength
keyDecryptedLength = dwBlobLength-12;
//decrypt the key with the private key k2
CryptDecrypt(k2,0,FALSE,0,keyDecrypted,&keyDecryptedLength))
```

To unwrap a key derived by a known string, more steps are necessary (we show the solution with *simepleblob*):

1. A string of bytes, a plain text key, must be encrypted by a private/public key. The length should be equal to a known key, e.g. a 3DES key has a length of 192bit.

2. Create a simpleblob, including a *publicstruct* header, specifying the crypt provider, the encrypt algorithm (i.e. key algorithm, such as 3DES), the public/private key pair and the raw key with its length (the procedure is described in the previous chapter).

3. Encrypt the key material with *CryptEncrypt()*. The byte array (i.e. the simpleblob) contains now a *publicstruct* blob header followed by the encrypted key.

4. Import the key blob into the token with *CryptImportKey()*. A new handle to the imported key is created.

There are no restrictions on the keyblob: it can be created by the running application or by another application on a different computer. This allows an intruder to create a private blob from a random key, export it (remember, the token provider denies it), change some bytes and import it successfully into any other provider. This will be the starting point for the one-exponent key attack. The new session key is ready to perform all the cryptographic operations, and the unwrap is done. With key unwrapping it is possible to obtain different handles, giving keys different attributes (possibly not conflicting ones).

Another "bug" allow an intruder to export a session key without the knowledge of public/private key pair. As discovered by TooKan in [6], with the token it is possible to import any key without restrictions, so an user can create a private key, modify the exponent to one and import it again under a different CSP, the token's one. Main steps are:

1. Call *CryptAcquireContext()* to acquire the Microsoft Enhanced provider, and generate a public/private key pair.
2. Export a private key into a privatekeyblob, access the structure and set the RSA public and private exponent to 1.
3. Write the key blob into an external text file, then import it again into the token CSP.

In C++, the function used to perform these operations is *createOneExponentKey()*:

Listing 6.5: one exponent key generation

```
// Generate the private key
CryptGenKey(hProv, dwKeySpec, CRYPT_EXPORTABLE, hpKey);

// Export the private key, we'll convert it to a private
// exponent of one key

CryptExportKey(*hpKey, 0, PRIVATEKEYBLOB, 0, keyblob, &dwkeyblob);

// Get the bit length of the key
memcpy(&dwBitLen, &keyblob[12], 4);

//pointer
BYTE *ptr;

// Modify the Exponent in Key BLOB format
// Key BLOB format is documented in SDK

// Convert pubexp in rsapubkey to 1
ptr = &keyblob[16];
for (n = 0; n < 4; n++){
        if (n == 0)
                ptr[n] = 1;
        else ptr[n] = 0;
}

// Skip pubexp
```

```
ptr += 4;
// Skip modulus, prime1, prime2
ptr += (dwBitLen/8);
ptr += (dwBitLen/16);
ptr += (dwBitLen/16);

//Convert exponent1 to 1
for (n = 0; n < (dwBitLen/16); n++){
        if (n == 0)
                ptr[n] = 1;
        else ptr[n] = 0;
}

// Skip exponent1
ptr += (dwBitLen/16);

// Convert exponent2 to 1
for (n = 0; n < (dwBitLen/16); n++){
        if (n == 0)
                ptr[n] = 1;
        else ptr[n] = 0;
}

// Skip exponent2, coefficient
ptr += (dwBitLen/16);
ptr += (dwBitLen/16);

// Convert privateExponent to 1
for (n = 0; n < (dwBitLen/8); n++){
        if (n == 0)
                ptr[n] = 1;
        else ptr[n] = 0;
}

//write key blob to file
[...]

//Change to the token provider, read the file and import the key
//blob

// Import the exponent-of-one private key into the token!
CryptImportKey(hCryptProv, keyblob, dwkeyblob, 0, 0, hpKey)

//hpKey is the new key
```

The application returns a valid handle to the key, and it can be used to simulate a wrap of the session key k_1 into a simple blob. If k_1 is wrapped and exported into a simpleblob with the one-exponent public key, from the 12^{th} position of the blob it is possible to read the key in clear, i.e. "as is". This key can be also used to "unwrap" a key, i.e. the key is imported, again, without any encoding.

Chapter 7

CAPI and PKCS#11 keys

Behind a common set of cryptographic operations, key management plays a primary rule for security both in PKCS#11 and CAPI applications, even if keys have different uses and properties: for example, in CAPI, symmetric keys are volatile and cannot be saved in the token, and public/private key pairs result accessible through a unique handle. The token is use provides interoperability between PKCS#11 and CAPI, but with some restrictions which may vary according to token versions. The aim is to study if keys, stored in the token, are vulnerable to attacks, such as *wrap/decrypt*, using CAPI and PKCS#11 alternately (in succession or simultaneously). A first analysis concerns the public and private keys, their permissions and usage limits, then symmetric and session keys (respectively of PKCS#11 and CAPI), which appear to have the most interoperability issues.

7.1 Public and private keys

In CAPI, a public/private key pair is created by the function *CryptGenKey()* specifying the flag *at_key_exchange* or *at_signature*. This key specification determines public and private key rule: an exchange key pair can perform all kind of cryptographic operations, *wrap*, *decrypt* first of all, while a signature key is more limited. Most of the key permissions are visible via PKCS#11 when a key search is performed, but only private key can be retrieved. Each key has a list of attributes (i.e. a template), in particular boolean attributes indicate a key status or rule: for example, if *CKA_SENSITIVE* is *true*, then the key is sensitive, and its values cannot be read; if *CKA_UNWRAP* is true, the key can perform an *unwrap* operation.

A private exchange or signature key, when created in CAPI, has the following attributes set to true:

exchange key: *unwrap, decrypt, signature, sensitive, never_extractable*

signature key: *signature, encrypt, sensitive, never_extractable*

An exchange key can perform the whole set of operations (i.e. it is also a signature key) considering that the corresponding public key is *wrap* and *encrypt* (we know from CAPI). In particular, the conflict operations wrap and decrypt are carried out without any limitation, and it constitutes a security issue. Despite that, a CAPI key pair, as is, with all its permissions and functionalities, cannot be used in PKCS#11. CAPI stores into a device the only private key, so that the corresponding public key cannot be found by a public search in PKCS#11. All keys are retrieved specifying in the search template the attribute *CKA_CLASS* with the value *CKO_PUBLIC_KEY* or *CKO_PRIVATE_KEY* (i.e. a key is only accessible after user authentication). A CAPI key pair has an unique handle: the function *CryptGetUserKey()* retrieves the private key, and the public key is dynamically derived from it. The token does not allow user to get or delete only one of the two key (from a key pair) separately. As seen before, a private key blob stores private but also public information, such as the public exponent and the modulus (common attribute). Every time a public key is imported, a default set of attributes is applied, depending on private key type (if exchange or signature). CAPI attributes cannot be changed in a PKCS#11 application: when a private key is retrieved, and a call to the function *C_GetKeyAttributes()* is made, some of its attributes are visible, but they are read only. Any tempt to set or change key attributes will result in two types of errors:

- CKR_FUNCTION_FAILED, if try to set off *sensitive* and *extract*
- CKR_DEVICE_ERROR, if try to change other attributes

The attribute *sensitive* is sticky-on, while *extractable* is sticky-off. Anyway we cannot change attributes value, according to the token attributes policies. The public key should be created ”ex-novo”, with a specific public exponent and modulus, then we *may* get a valid key to use with the private. It is presented a template for public key with the attribute *wrap* set to true: it would be a security issue, because any extractable secret key could be wrapped then decrypted with the private key. In next chapter the problem is analyzed in detail, then some fixes will be proposed.

Listing 7.1: create a public key

```
//create a search template to find the private key
CK_ATTRIBUTE search_template[] = {
    {CKA_TOKEN, &yes, sizeof(true)},
    {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
    {CKA_PUBLIC_EXPONENT, &publicExponent, sizeof(publicExponent)}
};
//save public attributes

//create a template for a new public key

CK_OBJECT_CLASS keyClass = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType =  CKK_RSA;

CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &keyClass, sizeof(keyClass)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_WRAP, &yes, sizeof(true)},
    {CKA_MODULUS, modulus, sizeof(modulus)},
    {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)}
};

//create an RSA public key object
rv = (*functions->C_CreateObject)
        (session, template, 5, &publicKey);
```

In PKCS#11 a public key is created starting from a public exponent and modulus (using the function *C_CreateObject()*), or through the function C_GenerateKeyPair(), which creates a public/private key pair. Respect to CAPI key pairs, PKCS#11 maintains two handles, and each key can be retrieved in different sessions. If keys are saved into the token, CAPI can recognize, then get, the only private key, with the attributes specified in PKCS#11 at creation time. The public key is reconstructed form the private using CAPI template: even if in PKCS#11 specific attributes and permissions are set, they will be lost and substituted with default ones.

Unlike CAPI, in PKCS#11 we must specify a template when we generate a public and private key pair (one different template for each key type). A list of attributes to be excluded, for each key, is specified in the document of PKCS#11 [11], in addition to some restrictions specified in token policies (e.g. a private key cannot have the attribute *encrypt* set to true). Attributes set in PKCS#11 affect operations in CAPI: if a private key has *decrypt* set to false (anyway there is no reason to set it to false), in CAPI the key cannot be used to decrypt data, and permissions can no longer be modified.

As specified in token documentation, CAPI can read *some* keys created via PKCS#11. The first operation to be made is a key-container search, to list all the keys stored in the token: by the call to the function *CryptGetProvParam()*, all PKCS#11 key containers, which store private keys, are returned with some pseudorandom labels. If other key types (i.e. secret or public keys) are kept inside, they cannot be read by a CAPI application. This limit also affects CAPI keys to be read via PKCS#11 applications: only private keys stored in a token (remember, only one handle for each key pair created) are recognized. In the following code, the function determines the current key container (with the flag *CRYPT_FIRST set*) and the keys which are stored in the token (flag *CRYPT_NEXT set*).

Listing 7.2: search keys

```
//current key container.
if(CryptGetProvParam(
        tokenProv,
        PP_ENUMCONTAINERS,
        containerName,
        &containerNameSize,
        CRYPT_FIRST)
)

[...]

//search other key containers.
while(CryptGetProvParam(
        tokenProv,
        PP_ENUMCONTAINERS,
        containerName,
        &containerNameSize,
        CRYPT_NEXT))
{
        for(count=0; count<containerNameSize; count++)
                printf("%c", containerName[count]);
}
```

Next session is about session and secret keys interoperability.

7.2 Session and secret keys

CAPI session keys are volatile and can be stored, then exported, into key blobs, outside any token devices. PKCS#11 corresponding keys are secret keys, i.e. symmetric keys, with particular attributes and permissions, which can be saved into tokens and used in different sessions. CAPI keys can be read in PKCS#11 as encrypted (or not) data string: if a session key is encrypted with a CAPI public key, the corresponding private key is used in PKCS#11 to decrypt data and the the key is recovered, or if the key was exported into a plain text key blob, data is recovered without decryption. CAPI session keys, for they specifications, are **never sensitive**, and they are mainly used as symmetric keys (i.e. to encrypt and decrypt a message exchanged among users). Using the same application and executing both the two standards, session keys are not directly accessible via PKCS#11, but this limit can be easily by-passed decrypting a key blob which contains such a key. PKCS#11 secret keys can be sensitive and store secret data, so attacks to discover sensitive keys' value make sense. These keys cannot be directly read via CAPI at all, even if they are stored into a token. By default, keys are saved into a key store, like the one created to store public and private key pair, but two scenarios arise:

- the key container is not visible in CAPI if it only contains symmetric keys
- key handles are not valid

Anyway, if the correct PKCS#11 key container is retrieved, no secret key can be handled. To get a key we can use the function *CryptGetUserKey()*, but only the private key, if present, is returned. Symmetric keys can be imported if saved in the known key blob format.

The following CAPI wrap operation uses a PKCS#11 secret key handle generated in the same application (i.e. session), but it returns an error.

Listing 7.3: error wrapping key

```
//let k1 be a secret key
(*functions->C_GenerateKey) (session, &gmec, sTemplate, 8, &k1);

//let k2 be a public/private key pair
CryptGenKey(tokenProv,AT_KEYEXCHANGE,CRYPT_EXPORTABLE,&k2);

//wrap k1 with k2 public key
CryptExportKey(k1,k2,SIMPLEBLOB,0,NULL,&dwBlobLength);
```

The function terminates with the error *bad_key*, i.e. k_2 is not a valid key handle, so that a secret (sensitive) key cannot be directly used in CAPI and perform a wrap/decrypt attack, despite it is always possible to perform this kind of operations with session keys using public/private key pair. This is an interoperability limit, but it ensures a more secure environment. Next section introduces some attacks and the relative fixes, with the use of specific and limited key templates.

7.3 Attacks and vulnerabilities using PKCS#11 and CAPI together

CAPI is shown to be vulnerable to some attacks such as the wrap-decrypt attack, because of the property of public/private key pairs (always wrap and decrypt keys) and the facility to manipulate key blob structure to create new fake keys, like the one-exponent key, which allows to read any session key value without performing a decrypt operation. PKCS#11 standard is also vulnerable to attacks aimed at discover the plain text value of sensitive keys [5]. Some fixes to these security issues have been proposed [2]: sensitive keys should be set as not exportable, or public key should not be wrap keys, but huge limits would be imposed over standard functionalities. In addition to known attacks, the interoperability between CAPI and PKCS#11 offers some interesting ideas to exchange keys and discover secret key values. The first "cross API" attack consists on importing and/or decrypting a secret (and sensitive) key created in PKCS#11; the last one starts from CAPI and the aim is to discover the key value from an encrypted key blob.

1 - Export and import sensitive key: start from PKCS#11 with a public/private key pair k_p, and a symmetric, sensitive and exportable key k_3. Aim: wrap k_3 with $pub(k_p)$ and decrypt it with CAPI using the private key k_p to read k_3 secret value.

Gen_Key_Pair:	$\xrightarrow{new n_1, n_2} h(n_1, k_p), h(n_2, pub(k_p));$ $A(n_1,$ t_exchange$)$, $A(n_2,$ t_public$)$
Gen_Key:	$\xrightarrow{new n_3} h(n_3, k_3)$; $A(n_3,$t_session$)$, exportable$(n_3,\ \top)$.
Export(asym):	$h(n_2, pub(k_p)), h(n_3, k_3) \rightarrow \{k_3\}_{pub(k_p)}$

Now reverse $\{k_3\}_{pub(k_p)}$ bytes, because of the different encryption format. In CAPI, the function CryptGetUserKey() retrieves the private key k_p. Using the handle h(n_1, k_p) we can access to the private key, but also to a public (derived) key.

(first solution): the key is decrypted directly

$$\text{ADecrypt:} \quad h(n_1, k_p), \{k_3\}_{pub(k_p)}; \rightarrow k_3$$

(second solution): the key is imported then decrypted

Create_Blob:	$\{k_3\}_{pub(k_p)} \rightarrow \left[\{k_3\}_{pub(k_p)}\right]$
Import(asym):	$h(n_1, k_p), \{k_3\}_{pub(k_p)} \xrightarrow{new n_4} h(n_4, k_3);$ $A(n_4, t\_session), exportable(n_4, \top)$
Export(plain):	$h(n_4, k_3) \rightarrow \left[k_3\right]$

2 - Wrap/decrypt with symmetric key: start from PKCS#11 with a sensitive key k_1 and a symmetric wrapping key k_2. Keep k_2 value and save it externally, in order to build a plain text key blob, then a valid symmetric and decrypting key in CAPI. Aim: wrap k_1 with k_2 and decrypt it with CAPI using the imported k_2 to read k_1 secret value.

Gen_Key:	$\xrightarrow{new n_1} h(n_1, k_1)$; $A(n_1$, t_pkcs), exportable(n_1, $\top$), sensitive(n_1, $\top$)
Gen_Key:	$\xrightarrow{new n_2} h(n_2, k_2)$; $A(n_2$, t_pkcs), exportable(n_2, $\top$)
Export(sym):	$h(n_2, k_2), h(n_1, k_1); wrap(n_1, \top) \rightarrow \{\|k_1\|\}_{k_2}$

Now reverse $\{\|k_1\|\}_{k_2}$ bytes , because of the different encryption format. Create a plain text blob with k_2 value.

Create_Blob:	$k_2 \rightarrow \left[k_2\right]$
Import(plain):	$\left[k_2\right] \xrightarrow{new n_3} h(n_3, k_2); A(n_3, t\_session), exportable(n_3, \top)$
SDecrypt:	$h(n_3, k_2), \{\|k_1\|\}_{k_2}; \rightarrow k_1$

3 - Decrypt CAPI blob: a key k_2 is exported into a simple blob using k_3, a public/private CAPI key pair. In PKCS#11 the encrypted key $\{k_2\}_{k_3}$ is decrypted with the private key k_3. Note that in CAPI k_3 is a key pair, while in PKCS#11 only the private key is recognized (i.e. the unique object saved in the token).

Gen_Key_Pair:	$\xrightarrow{new n_3} h(n_3, k_p), pub(k_p); A(n_3, t\_exchange)$
Gen_key:	$\xrightarrow{new n_2} h(n_2, k_2); A(n_2, t\_session), exportable(n_2, \top)$
Export(asym):	$h(n_4, pub(k_p)), h(n_2, k_2) \rightarrow \{k_2\}_{pub(k_p)}$
//switch to PKCS#11	
//reverse bytes	
ADecrypt:	$h(n_3, k_p), \{k_2\}_{pub(k_p)} \rightarrow k_2$

Another attack should be found in PKCS#11 vulnerability. Clulow, in his work [4], introduced the term *key conjuring*, to refer to the ability to generate keys in any hardware device with unauthorized transactions: a public key can be generated starting from the public data contained in a private key (it is anyway not necessary to use them), a random ”raw” string can be unwrapped and become a new key, and so on. Using the two standards together, the following attack can be performed: a sensitive key is wrapped with a Trojan [4] public key, created *ad-hoc* deriving the modulus and the public exponent from the private (CAPI) key. The key can be imported into the token with the function *CryptImportKey()*, after it has been inserted into a valid simple key blob, to be used for other operations, or simply it can be decrypted to read the secret value. This sensitive key exchange is permitted if no restrictions to key template are made.

Listing 7.4: exchange key attack

```
//Initial knowledge: k1 public/private key (CAPI), k2 sensitive
//and exportable key (PKCS#11).

/Get public exponent and modulus from $h(n_1,k_1)$.

//search for private keys
CK_ATTRIBUTE private_search[] = {
        {CKA_DECRYPT, &yes, sizeof(true)}
};

(*functions->C_FindObjectsInit)
        (session, private_search, 1); //0 not match
rv = (*functions->C_FindObjects)
        (session, &foundKey, 1, &nKeys);
if(rv == CKR_OK && nKeys > 0) {
        lastKey = foundKey;
        get_key_info(session, foundKey);
        //search other keys
        rv = (*functions->C_FindObjects)
                (session, &foundKey, 1, &nKeys);
}

//generate public key object (set modulus bits: 1024)
rv = (*functions->C_CreateObject)
        (session, keyTemplate, 5, &publicKey);

//wrap
rv = (*functions->C_WrapKey)
        (session, &dec_mec, publicKey, k1, result, &lenResult);
```

```
//write to .txt file
FILE* pFile;
if(fopen_s(&pFile,"keywrappedsecret.txt","w")==0){
        for (n=0 ; n<lenResult ; n++)
                fprintf (pFile, "%02x",result[n]);
        fclose(pFile);
}

//change to CAPI calls.
//e.g. the key is 3DES. The key can be decrypted or imported.
//For simplicity, a simple key blob is first created; wrapped
//data follows little-endian format.

//we create a blob structure (keyBlob) performing a 3DES
//key export  with a public key; this would be used as model

//read data and save to keyDecrypted: its dimension
//(keyDecryptedLength) is the same of the wrapped 3DES
//key, exported previously

//create blob
PBYTE newBlob = new BYTE[keyDecryptedLength];

// Place the key material in reverse order (little-endian format)
for (n = 0; n < keyDecryptedLength; n++)
        newBlob[n] = keyDecrypted[keyDecryptedLength-n-1];

//maintain (key)blob header and write new data
for (n = 0; n < keyDecryptedLength; n++)
        keyBlob[n+12] = newBlob[n];

PBYTE finalKey = new BYTE[keyDecryptedLength];
finalKey = &keyBlob[12];

//decrypt key (1)

CryptDecrypt(k2,0,FALSE,0,finalKey,&keyDecryptedLength)

for(count=0; count<keyDecryptedLength; count++)
        printf("%02x",finalKey[count])

//if we want to import the key (2)

CryptImportKey(
        ehCryptProv,
        keyBlob,
        keyDecryptedLength+12,
        k2,
        CRYPT_EXPORTABLE,
        &k3
)

//key should be sensitive, but it is exportable and can be
//read in clear

CryptExportKey(
        k3,
        0,
        PLAINTEXTKEYBLOB,
        0,
        NULL,
        &dwKeyMaterial
)

pbKeyMaterial = new BYTE[dwKeyMaterial]

CryptExportKey(k3,
        0,
        PLAINTEXTKEYBLOB,
        0,
        pbKeyMaterial,
        &dwKeyMaterial
)

//if no errors, print the key
for(count=0; count<dwKeyMaterial; count++)
        printf("%02x", pbKeyMaterial[count]);

}
```

The attack is similar to the one proposed by Clulow, but here the public key directly uses private key information, which can be read across the two standards. The private key, in the example, is directly read in PKCS#11, then a decrypt operation can be performed in PKCS#11 without the need to use CAPI calls. More in general, as Clulow suggests, a malicious user can create a public key starting from information (modulus and public exponent) that only he knows and it is not accessible through PKCS#11 application.

Another vulnerability should be found in asymmetric key management: as described in previous sections, when a key pair is created in the token via PKCS#11, the standard maintains two handle for public and private key, while CAPI recognizes the private key only. Any template applied to the public key would not be valid in CAPI: a secure template for public key, proposed by Bortolozzo and Focardi [2], considers that public keys should not have the attribute *wrap* set on. Every time a public key is used in CAPI, it is derived from the private, with a default template. Functions such as *CryptEncrypt()* or *CryptExportKey()* can be always performed, because CAPI treats PKCS#11 keys as exchange keys, even if some permissions are changed or set off at creation time. During a common session, session keys (and hence the handles) are not recognized by both the two standards; **keys become vulnerable only if wrapped** (i.e. exported into an encrypted format). The point is: if a secret, sensitive and exportable key can be wrapped outside a PKCS#11 device with a public or a common symmetric key, using a valid private key (i.e. the private key should be a decrypt key), it becomes easy to read the clear value of the key.

The secure template for PKCS#11 public key can be easily bypassed using CAPI, but the resulting attack cannot be performed: a public key, generated or imported into the token, should have the attribute *wrap* set to false; it does not apply if the key (must be *AT_KEYEXCHANGE*) is created in CAPI. A public-private key pair, once created, has an unique handle, but it is possible to extract the public key, then import it obtaining a new valid handle. This key maintains the properties of encryption and wrap, but is not recognized as public key object in PKCS#11: the *C_WrapKey()* function terminate with error *CKR_OBJECT_HANDLE_INVALID*.

Listing 7.5: invalid public handle

```
//k3 is a AT_KEYEXCHANGE key pair

// Export the public key
CryptExportKey(k3,0,PUBLICKEYBLOB,0,NULL,&dwBlobLength)

// Import the public key under a different handle
CryptImportKey(
        tokenProv,
        keyBlob,
        dwBlobLength,
        0,
        CRYPT_EXPORTABLE,
        &k2
);

//leave the provider open. No key handle is left
```

```
//generate a sensitive key
(*functions->C_GenerateKey)
        (session, &gen_mec, skey_template, 8, &sKey);

//function exits with error
rv = (*functions->C_WrapKey)
        (session, &gen_mec2, k2, sKey, resultWrapped, &lenResult);
```

This limit of using key objects between CAPI and PKCS#11 would ensure that secure templates and Policies are good enough to avoid attacks on these devices. In next section secure templates are described in detail.

Chapter 8

Fixes and secure templates

Most tokens set some limits on templates that can be used to create a PKCS#11 key. Some attributes should be disabled in order to avoid conflicts among them and the wrap/decrypt attack: the same key should not be either wrap and decrypt, and a sensitive key must not be extractable. In the token we use for tests, a sensitive key can be created with the attribute *extractable* set to true, which makes possible all the attacks showed in [2] . Using CryptokiX [12], an extension to open-Cryptoki, we can add a list of **conflicting attributes** to token policies, in order to prevent or limit some attacks. Pairs of attributes such as *encrypt*, *decrypt* and *wrap*, *unwrap* should not belong to same key, i.e. each key maintains distinct rules. In CAPI, any exchange key pair can perform a wrap and decrypt (i.e. the public key is wrap, the private is decrypt), and permissions cannot be changed or selected at creation time, considering that there is no function or template array to call or set in CAPI. If the private key is read in PKCS#11, we find it has usually the attribute *never-extractable* set to true, and some attributes, except for *modulus* and *public-exponent*, cannot be read, because the key is set as *sensitive.*

(fix):conflicting attributes in PKCS#11: *wrap*, *unwrap*, *encrypt*, *decrypt.*
(fix):conflicting attributes in CAPI: none.

Sets of **sticky-on** and **sticky-off** attributes are already defined in the token. A private key is always sensitive, and a not-extractable key cannot be set as extractable. In addition to these known *sticky* attributes, we can add *sticky* properties to conflicting attributes, in order not to change key properties and rules once the key has been created. This property, already discussed in [6], applies to PKCS#11. If we get CAPI private keys handle and attributes list in PKCS#11, it is not possible to change attributes value. A stronger limitation than sticky properties is given by the attribute *modifiable*, set to false. This is applied by default to all CAPI private keys.

We now consider SafeNet response [8] to the attacks found in Aladdin eToken devices using the tool "Tookan" [3]. The document suggests to:

- Block sensitive key export completely: no wrap operation would have effect on the key. Key exchange is limited to symmetric (or session) keys with the attribute *sensitive* off.
- Use trust and wrap-with-trusted attributes: keys can be wrapped (i.e. exported) with trusted keys only. This is a good trade-off between a never-extractable key and an extractable key; key exchange is allowed.

The first point suggest to set every sensitive key as "not exportable", so that every attack which aims to discover sensitive key value fails as soon as the function *(Asym/Sym)Wrap* is called. With the second solution, the sensitive key must be set *exportable* under a certain *trusted* key, created at token initialization, i.e. a public key is verified and have the attribute *trusted* set on. This control ensures that only known keys can carry out secure wrap and export operations. In our analysis, we consider the scenario in which every sensitive key is also *exportable*. The set of conflicting attributes is not enough to ensure a complete solution to attacks: a key, for example, can be unwrapped more times to generate new copies with conflicting attributes. A solution is found in **secure templates** [2, 6], where attributes availability depends on key type (symmetric, public and private) and use (key generation, key unwrapping).

The question is: do secure templates apply to CAPI operations? How would they be affected? CAPI keys, excluding sensitive keys (we have only session keys, never sensitive), such as private and public keys, have not templates to set: attributes (here permissions) are automatically set, according to the key (pair) specification *at_keyexchange, at_signature*. In few words, while PKCS#11 keys templates are fixed,

CAPI does not allow to fix public/private keys permissions.
A combination of conflicting attributes and sticky properties can be summed up in one unique attribute: *modifiable.* All keys, once generated, should have the attribute *modifiable* set to false: key permissions cannot be changed and roles are kept separate, e.g. a key is encrypt and decrypt or wrap and unwrap, while it can never be wrap and decrypt or unwrap and encrypt (being pairs of conflicting attributes).

In PKCS#11 key generation, according to secure templates, one of the following templates should be applied:

Encrypt/decrypt key: sensitive = true, modifiable = false,
encrypt = true, decrypt= true, wrap = false, unwrap = false.

Wrap/unwrap key: sensitive = true, modifiable = false,
encrypt = false, decrypt= false, wrap = true, unwrap = true.

Using this configuration, a generated symmetric key cannot perform a key wrap and then a decrypt. The attack *wrap/decrypt with symmetric key* is denied if we create every key with the attribute **sensitive**: in fact, the value of symmetric wrapping key cannot be read and then used in CAPI to build a new valid decrypt key. In the model, FX will be as follows:

$$\text{FX: } \mathit{wrap}(n_i, \bot), \mathit{unwrap}(n_i, \bot), \mathit{sensitive}(n_i, \top)$$
or
$$\text{FX: } \mathit{encrypt}(n_i, \bot), \mathit{decrypt}(n_i, \bot), \mathit{sensitive}(n_i, \top)$$

Key attributes may be changed (e.g. we set wrap to false and decrypt to true), or, if we consider the *re-import* attacks, a new handle to the same key can be obtained with a key unwrap, specifying new attributes value. The first attempt is avoided by setting the attribute *modifiable* to false (as we discussed before); the second one needs a new template for unwrap keys.

Imported key template: encrypt = true, decrypt= false,
wrap = false, unwrap = true.

An imported or derived key should never have the attribute *wrap* and *decrypt* set to true, so that the re-import attacks are not allowed. In the model:

$$\text{FX: } decrypt(n_i, \perp), wrap(n_i, \perp)$$

In CAPI, wrap and decrypt operations are performed by private and public exchange keys, i.e. public for wrap and private for decrypt. The template (not modifiable) is: sensitive = true, modifiable = false, encrypt = true (public), decrypt= true (private), wrap = true (public), unwrap = true (private).
Considering that a private key (and not the public) can be handled in PKCS#11 application, we obtain a *decrypt* and *unwrap* key. If we create a corresponding wrapping public key in PKCS#11, any sensitive key can be exported and read. This cross-api attack requires first to create a private key in CAPI, then use it in combination with a public key, created with the function *C_CreateObject*. To fix this vulnerability, **the previous imported key template is applied** if *C_CreateObject* is called. Even if we are interested in the public key exponent and modulus to wrap, we need to create a valid key object to be used in the function (i.e. we cannot use directly the exponent and modulus value). Clulow trojan key attack, which considers a private key and a derived public key, is negated.

A public key, if wrapping, would be able to perform all kinds of attacks seen since now, so such kind of limit is necessary to be applied. Exchange key, from CAPI to PKCS#11 is allowed: a key, exported via CAPI using the public key, is imported (i.e. unwrapped) with the corresponding private key, because the private attribute *unwrap* remains unchanged. Anyway, the new imported key cannot wrap a sensitive key. Bortolozzo [2] suggests an additional template when the PKCS#11 function *C_GenerateKeyPair* is called.

Private/public key pair template: sensitive = true (private), modifiable = false, encrypt = true (public), decrypt= true (private), wrap = false (public), unwrap = false (private).

In the model:

$$\text{FX: } wrap(public\_key, \perp), unwrap(private\_key, \perp)$$

This template limits a lot cryptographic functionalities: in PKCS#11 this type of keys would be used in encryption and decryption only, while in CAPI, the private key (correctly recognized even if it is created in PKCS#11), cannot be used to unwrap (import) other keys. Attributes, specified in PKCS#11, become valid in CAPI for the private key, while the public key is every time reconstructed from the private key and gets default permissions (hence it is an encrypt and wrap key). To deny asymmetric wrap/decrypt, it is enough to set the attribute wrap to false; the attribute unwrap, if set to true, would be useful to import and decrypt a key previously encrypted with the public key by another user.

Actually, templates can be easily adopted to avoid all the attacks and cross-api operations. The unique way to perform a wrap is to create trusted keys, but this operation can only be performed by an administrator.

Chapter 9

Private key backup

Key backup is a kind of operation that users can perform both in CAPI and in PKCS#11, but in different contexts. Every key with the attribute *exportable* set on *true* can be exported, encrypted or not, outside of a hardware device (e.g. a token) or a software provider. With Microsoft API, all symmetric keys and public keys are exportable, by default, then *archived* into a key blob. Since symmetric keys are volatile, the only way to reuse them from session to session is to import the key blob, i.e. the key backup. Keys, in PKCS#11, can be stored in a token, if the attribute *CKA_TOKEN* is set on *true*, but anyway, once wrapped, i.e. exported under another key, they can be saved into an external file. Different treatment is for private keys: token policies do not allow them to be exported, even if they are declared *exportable.* In CAPI, a public/private key pair admits the flag *archivable*, i.e. the key pair is exportable only in the current session (the same in which the key is generated), into a private key blob. In next sessions, the private key becomes no more extractable. An application is executed running the two standards simultaneously: first a public/private *archivable* key is created in CAPI, then, without closing the current session, a public object search is made in PKCS#11. The results show two equal private keys (key modulus is a proof), but with different attributes: one key has *never_extractable* set to *false*, *extractable* set to true and *sensitive* false, while the second key has opposite attribute values. An archivable key is created via software, i.e. it is exportable but volatile, then copied and stored into the token. If the object search is performed in future sections, only the *token* key is retrieved. Application results are listed below.

```
@pkcs11 and capi.exe
Object:
Key found!
is sensitive: false
is neverExtract: false
is extractable: true
keytype: 3

Object:
Key found!
is sensitive: true
is neverExtract: true
is extractable: false
keytype: 3
```

The key type means that the object found is a private key stored in the token. The private session key is not sensitive, then all its attributes can be read. Anyway, any change on them will be lost next sessions.

```
@read keys PKCS11i.exe
Object:
Key found!
is sensitive: true
is neverExtract: true
is extractable: false
keytype: 3
```

The modulus, here not shown, is the same for all the keys retrieved.

Chapter 10

Results and conclusions

Without fixes, sensitive keys created in PKCS#11 are vulnerable and can be imported in CAPI with simple cross-api attacks. Data structures, such as key blob, can be easily manipulated (except for symmetric key blob), to obtain the encrypted key, or created ex-novo to import PKCS#11 keys. A new CAPI imported key, specially if symmetric, is considered as a session key, so it can be exported in clear without any additional operation (encryption or decryption). In short, if a symmetric key is wrapped with a public key in PKCS#11, it can be imported in CAPI then exported and read in clear. Wrap and decrypt can be executed separately via CAPI then PKCS#11 (or vice-versa). The absence of a template to be set in a huge limit in CAPI: fixes on it in PKCS#11 do not affect any imported key. An exchange key pair performs all the operations: wrap, unwrap, encrypt, decrypt, but also sign and verify. Its permissions cannot be change when the key is read by a PKCS#11 application.

Adopting partial restrictions, such as sticky-on and sticky-off attributes (already set in token policies), it does not constitute a good protection from attacks. Secure templates are a good solution, but they limit a lot CAPI functionality. We notice that if we create a key pair in PKCS#11, private key attributes affect CAPI private keys behavior, i.e. they cannot perform the unwrap, then they cannot be used for key exchange. On the opposite side, if we create a key pair in CAPI, we can initialize the exchange operation, because the public key ($pub(k_p)$) is derived and it has the wrap permission always set to true, then call the unwrap with the private key. Permissions and template should be inserted as policies.

If a public/private key pair should be only encrypt and decrypt, only (CAPI) signature keys are allowed. Key exchange with asymmetric keys would be limited or denied. A wrap/unwrap template should be avoided, because in CAPI a secret key value is read in clear in a plain text key blob. It is therefore correct to assign such a template to session keys only (as "predicted" in [2] for PKCS#11 keys).

Using the super model, we study to what extent these attacks can be performed, with fixes and without fixes.
Without fixes we **can perform** the following attacks:

1. Export and import sensitive key
2. Wrap/decrypt with symmetric (and not-sensitive) key
3. Decrypt CAPI blob

The first point needs a **wrapping** public key, in order to import the symmetric key with CAPI (or decrypt it directly); the second one considers a **not-sensitive** symmetric key, so that its value can be read then used to create a new CAPI key; the third point is about a key exchange operation: it describes how this operation can be performed across the two standards. With fixes, i.e. sticky properties, key-separation roles and secure templates, we **cannot perform** any more these attacks:

1. Export and import sensitive key - because the public key in PKCS#11 is no more wrapping
2. Wrap/decrypt with symmetric (and not-sensitive) key - because every key should be sensitive

The third point can be executed because wrap is made with the public key in CAPI, then the private is used via PKCS#11; for this key, it is not possible to apply/change key attributes. If the token policies apply these restrictions for CAPI, neither this last operation is allowed.

If we deny *wrap* to secure the token, then it means no operations with CAPI. A trade-off could be the following one:

- Asymmetric CAPI keys: wrap, without encrypt and decrypt, OR encrypt and decrypt (i.e. signature key pair).

- Symmetric CAPI keys: encrypt/decrypt

If these permissions are set, CAPI looses exchange properties, but it can export a symmetric key (can be then imported?). Any symmetric key is used to encrypt or decrypt data, such as PKCS#11 symmetric keys. The difficult is to allow basic CAPI operation and ensure a good level of security. The interoperability between the two standards gives us interesting starting points for further analysis.

Chapter 11

Future Work: Microsoft CNG

Until now, a full ”efficient” secure token configuration has not found. Security means huge limits on cryptographic operations: CAPI and PKCS#11 can be executed with a good interoperability grade, but restrictions on one standard make the other inefficient. CAPI essentially performs a key exchange and a key signature using a couple of public and private keys. No key is stored in the token. PKCS#11 stores secret keys, which value should be protected, making limits on key exporting (and so wrapping). Future Microsoft Cryptographic APIs are **Microsoft CNG**. CNG, Cryptography API Next Generation, unlike CAPI, separates cryptographic providers from key storage providers (KSPs). KSPs are then used to create, delete, export, import and store asymmetric and symmetric keys. Each key is an object with some properties (constants), which can be set by an apposite function. There are properties like ALGORITHM_NAME, BLOCK_LENGTH, KEY_LENGTH, or the more interesting Export Policy, Key Type and Key Usage properties. The export policies apply to private key, specifying how they can be exported. Key usage properties set some ”preconfigured” permissions sets: we can create Decryption keys (i.e. the new key is used for encryption and decryption), exchange or signing key, and a ”all-usage” key, similar to CAPI session keys. The **Decryption key** is interesting, because we obtain a separation from a wrapping key and a decrypting key, which was missed in CAPI.

The new set of APIs permits a higher grade of customization. It would be interesting to discover how restrictions and fixes affect or limit cryptographic operations, considering that there are about twenty new keyblob formats, the full support to suite B algorithms [9], and many other features. Is possible to create a new strong interoperability among security APIs and reach the best complete security?

Bibliography

[1] How to export and import plain text session keys by using cryptoapi. `http://support.microsoft.com/default.aspx?scid=kb;EN-US;q228786`, 2006.

[2] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing pkcs#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 260–269, New York, NY, USA, 2010. ACM.

[3] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Tookan, tool for cryptoki analysis. `http://secgroup.ext.dsi.unive.it/tookan/`, 2010.

[4] Jolyon Clulow. On the security of pkcs#11. In *In proceedings of the 5th international workshop on cryptographic hardware and embedded systems (Ches'03), volume 2779 of lncs*, pages 411–425. Springer-Verlag, 2003.

[5] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 331–344. IEEE Computer Society Press, 2008.

[6] Riccardo Focardi, Flaminia L. Luccio, and Graham Steel. Foundations of security analysis and design vi. chapter An Introduction to Security API Analysis, pages 35–65. Springer-Verlag, Berlin, Heidelberg, 2011.

[7] R. Housley. Triple-des and rc2 key wrapping. RFC 3217 (Informational), December 2001.

[8] SafeNet inc. Attacking and fixing pkcs#11 security tokens - a response by safenet. http://secgroup.ext.dsi.unive.it/wp-content/uploads/2010/10/Reponse-by-SafeNet.pdf.

[9] Microsoft. Cng, msdn documentation. http://msdn.microsoft.com/en-us/library/windows/desktop/bb204775(v=vs.85).aspx.

[10] Microsoft. Cryptography functions, msdn documentation. http://msdn.microsoft.com/en-us/library/windows/desktop/aa380252(v=vs.85).aspx.

[11] RSA-laboratories. Pkcs#11: the cryptographic token interface standard. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf, 2004.

[12] SecGroup. Cryptokix: a software cryptoki (fixed) token based on opencryptoki. http://secgroup.ext.dsi.unive.it/projects/security-apis/cryptokix/, 2010.

[13] Graham Steel. Cryptographic key management apis. http://www.lsv.ens-cachan.fr/\~steel/teaching/pkcs11/FMSS-crypto.pdf, 2013.

[14] Douglas Stinson. *Cryptography: Theory and Practice,Second Edition*. CRC/C&H, 3nd edition, 2006.