On the Security of PKCS #11

Jolyon Clulow

University of Natal, Department of Mathematical and Statistical Sciences, Durban, South Africa clulow@icon.co.za

Abstract. Public Key Cryptography Standards (PKCS) #11 has gained wide acceptance within the cryptographic security device community and has become the interface of choice for many applications. The high esteem in which PKCS #11 is held is evidenced by the fact that it has been selected by a large number of companies as the API for their own devices. In this paper we analyse the security of the PKCS #11 standard as an interface (e.g. an application-programming interface (API)) for a security device. We show that PKCS #11 is vulnerable to a number of known and new API attacks and exhibits a number of design weaknesses that raise questions as to its suitability for this role. Finally we present some design solutions.

1 An Introduction to PKCS #11

The Public Key Cryptography Standards (PKCS) were developed by RSA Security Inc. "in cooperation with representatives of industry, academia and government to provide a standard to allow interoperability and compatibility between vendor devices and implementations." [1] A significant factor in the success of these standards can be attributed to this co-operative approach. The standards cover a variety of aspects of Public Key cryptography including PKCS #1: RSA Encryption Standard, PKCS #11: Cryptographic Token Interface Standard [18] and PKCS #8: Private-Key Information Syntax Standard. Many significant APIs and protocols have been built upon PKCS #11 (e.g. SSL). Notable products with PKCS #11 support include Mozilla (the open source browser upon which the Netscape browser is based) and SSL hardware accelerators from companies such as nCipher, IBM, Thales, Rainbow and AEP amongst others. Indeed, this research was prompted by the question of the suitability of the PKCS #11 API as an interface to a hardware security module (or crypto coprocessor).

The designers of PKCS #11 described the design goals as follows: to "provide a standard interface between applications and (portable) cryptographic devices" and at the same time to "allow resource sharing" (a many-to-many relationship between applications and devices). It was not intended to be a general interface to cryptographic operations or security services. Rather it could be used to build such services, operations or suitable APIs.

[1] Unless indicated otherwise, all quotations and figures are reproduced with permission from [18].

© Springer-Verlag Berlin Heidelberg 2003

In PKCS #11 terminology, a token is a device that stores objects (e.g. Keys, Data and Certificates) and performs cryptographic operations. This is a logical rather than a physical characterization; where one device may have several, distinct logical tokens (e.g. akin to the concept of distinct domains). When intending to make use of a token (or to communicate with it), one must first establish a session with the token, which requires the user to 'login' and to be authenticated to the device. Thereafter, the user may make use of the functionality provided by the token by making calls through the interface or API. Objects are characterized as either token objects or session objects. Token objects are non-volatile in nature and exist (i.e., are stored) on a token. In addition, they possess the property that they are visible to all applications connected to the token. In contrast, session objects are volatile, existing only for the duration of the session between an application and a token. They only have scope within that session (i.e., are only visible to the application which created them).

Each object has a set of properties that describes the object and controls its use. For example, every key possesses the Key Type property which identifies it either as a public, private or secret key. Private and secret keys are recognised by the standard for the requirement to protect the secrecy thereof, and possess the properties sensitive, extractable, always sensitive and never extractable. "Sensitive keys cannot be revealed in plaintext off the token, and unextractable keys cannot be revealed off the token even when encrypted (though they can still be used as keys)."

PKCS #11 describes two types of users: security officers (SO) and normal users (users). The security officer is responsible for administering the users and for performing such operations as initially setting and changing passwords. Unlike normal users they cannot perform cryptographic operations. All users must 'login' (i.e., be authenticated to the token) before they can access the objects or capabilities of a token. This is achieved through the use of a personal identification number (PIN), which acts essentially as a password. The standard allows for this mechanism to be augmented with or replaced by an alternative, custom mechanisms in any given implementation (e.g. PIN entry via PINpad or the use of smarts cards). This does not, however, prevent access to other users' token objects although this could be made another implementation feature.

The Security of PKCS #11

The standard has the following stated security targets.

1. "Access to private objects on the token, . . . , requires a PIN. Thus, possessing the cryptographic device that implements the token may not be sufficient to use it; the PIN may also be needed."
2. "Additional protection can be given to private keys and secret keys by marking them as 'sensitive' or 'unextractable'. Sensitive keys cannot be revealed in plaintext off the token, and unextractable keys cannot be revealed off the token even when encrypted (though they can still be used as keys)."

Implied within these statements is the intention that by marking objects as 'sensitive' and 'unextractable', another user is prevented from recovering the secret values thereof. It does not appear to be the intention to prevent one user from using another user's private objects.

The designers discuss several areas of concern including operating system security, the actions of rogue applications and the threat posed by Trojan linked libraries or device drivers that may subvert security, perhaps by stealing the password. Similar concerns related to the 'sniffing' of communication lines to the cryptographic device exist(eavesdropping). This leads to several possible compromises such as PIN recovery, unauthorized access to a session (and the ability to insert, modify or delete commands) and the impersonation of a token or device. However, the standard claims that "...none of the attacks just described can compromise keys marked 'sensitive,' since a key that is sensitive will always remain sensitive. Similarly, a key that is 'unextractable' cannot be modified to be extractable." Thus, in addition to examining the API for vulnerabilities, we are particularly interested in this claimed property.

A cryptographic device that supports a PKCS #11 faces the following potential threat models:

– a malicious security officer who abuses the authority of his position and his access to the device and user management functions,
– a cheating or malicious user who exploits his authorized access to the token, and
– a malicious third party who gains access to the token through some means.

Essentially, these threats resolve into either gaining access to a session, or gaining access to a device during a session (e.g. by injecting messages into communications lines) or having knowledge of a password.

There exist some obvious, well-known attacks that are, generally speaking, implementation dependant as opposed to weaknesses in the API itself. We briefly describe them for completeness. The `C_Login` function is potentially vulnerable to an exhaustive PIN (password) search since a user can try all possible passwords. One typical defence is to keep a count of the number of failed login attempts and 'lock' the card after a certain threshold of fails has been reached. Ideally, the counter should be incremented prior to testing the PIN and decremented thereafter only if successful. This can lead to a denial of service attack where a malicious party tries to prevent a valid user from being able to use the token. The attacker repeatedly and intentionally masquerades as the user and attempts to login with an incorrect PIN. An alternative approach is to make use of time delays during start up and between login attempts.

```
CK_DEFINE_FUNCTION(CK_RV, C_Login)
(
    CK_SESSION_HANDLE hSession,
    CK_USER_TYPE userType,
    CK_CHAR_PTR pPin,
    CK_ULONG ulPinLen
);
```

A malicious security officer could use the `C_InitPIN` function to change a given user's PIN to a known value, hence gaining security access to the token. Since all users have access to all objects on the token, another less detectable approach would be to make a new user with a known PIN. This new user would be able to gain access to the token objects. While the power inherently held by a security officer in a given system is understood, PKCS #11 fails to specify directly the use of dual control mechanisms, which would defeat a single malicious security officer, although not a conspiracy of security officers.

```
CK_DEFINE_FUNCTION(CK_RV, C_InitPIN)
(
    CK_SESSION_HANDLE hSession,
    CK_CHAR_PTR pPin,
    CK_ULONG ulPinLen
);
```

Key Management Functions

PKCS #11 provides a typical set of key management functionality including:

- `C_GenerateKey` that generates a secret key,
- `C_GenerateKeyPair` that generates a public/private key pair,
- `C_WrapKey` that wraps (i.e., encrypts) a private or secret key,
- `C_UnwrapKey` that unwraps (i.e. decrypts) a wrapped key, and
- `C_DeriveKey` that derives a key from a base key.

Let us consider the `C_WrapKey` function further. It has the following prototype:

```
CK_DEFINE_FUNCTION(CK_RV, C_WrapKey)
(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hWrappingKey,
    CK_OBJECT_HANDLE hKey,
    CK_BYTE_PTR pWrappedKey,
    CK_ULONG_PTR pulWrappedKeyLen
);
```

`hSession` is the session's handle; `pMechanism` points to the wrapping mechanism; `hWrappingKey` is the handle of the wrapping key; `hKey` is the handle of the key to be wrapped; `pWrappedKey` points to the location that receives the wrapped key; and `pulWrappedKeyLen` points to the location that receives the length of the wrapped key.

`C_WrapKey` can be used in the following situations:

- To wrap any secret key with an RSA public key.
- To wrap any secret key with any other secret key.
- To wrap an RSA, Diffie-Hellman, or DSA private key with any secret key.

2 Symmetric Key API Attacks

A wrapped key or external encrypted key is commonly referred to as an encrypted key token (T). Keys are typically wrapped (or encrypted) under a key encrypting key (KEK) for exchange or under a master key (MK) for storage external to the device. Initially we shall consider the wrapping of a secret key with another secret key. The mechanism describes the method of the wrapping operation and follows a naming convention of the form `CKM_<NAME>_<MODE>`. For example, `CKM_DES_ECB`, `CKM_DES_CBC`, `CKM_DES_CBC_PAD`, `CKM_DES3_ECB`, `CKM_DES3_CBC` and `CKM_DES3_CBC_PAD` are the mechanisms that make use of either single DES or triple DES. Other ciphers are possible including RC2, RC4, RC5, CAST, IDEA, etc.

2.1 Key Conjuring

Key conjuring is any technique that leads to the unauthorized generation of keys in the device. It is so named owing to the fact that the keys are 'conjured' (magically created or appearing seemingly out of nowhere). Bond in [6] first identified key conjuring as a security risk. This is for two reasons. First, it defeats any access control that was placed on the official key generation function by providing an alternative and unauthorized mechanism to perform effectively the same operation. Secondly, a key conjuring mechanism can be exploited to build a large set of keys, which can then be attacked by a parallel search, as described in Section 2.6.

Bond observed that crypto coprocessor designs, which stored keys outside the tamper-proof device, were vulnerable to unauthorized key generation. For instance, a random 8 bytes submitted as an external encrypted DES key will be decrypted and used as key. For example, using random data (R), a user creates a token $T_{random} = R$ which is then supplied to the `C_UnWrapKey` function call to the device. The device decrypts T_{random} as $d_{MK}(T_{random})$, yielding a new key $k_{random} = d_{MK}(T_{random})$. If parity checking is enforced, then there is a 1 in 2^8 chance that this new 'key' will have the correct parity. By repeating this process on average 2^8 times, an attacker can expect to conjure successfully a new key into the system in this manner. In fact this method is available in some older devices as a key generation function. Instead of merely testing for parity, the function will correctly set the parity in the process.

Key conjuring can be defeated through the associated use of a MAC or hash. This has the property of authenticating the clear value of the key as valid.

2.2 Key Binding (Integrity)

We observe that the choice of mode for the `C_WrapKey` is left to the caller (the user). In addition, there is no enforced use of a MAC or other technique to ensure data authenticity. There is also no restriction on the use of keys with repeated halves. As a result of the lack of cryptographic binding, one can attack each half of a key independently in the following way:

1. Export the target double length key (under any key encrypting key and in any mode). We denote the double length key as the ordered pair $K = \langle K_1, K_2 \rangle$ and note that each half is encrypted independently to form the encrypted key token (T);

$$\begin{aligned} T &= e_{KEK}(\langle K_1, K_2 \rangle) \\ &= \langle e_{KEK}(K_1), e_{KEK}(K_2) \rangle \\ &= \langle T_1, T_2 \rangle . \end{aligned}$$

2. Re-import the first half of the exported key as a single length key encrypted in ECB mode (using the same key encrypting key); $d_{KEK}(T_1) = d_{KEK}(e_{KEK}(K_1)) = K_1$.
3. Re-import the second half as a single length key encrypted in ECB mode (using the same key encrypting key); $d_{KEK}(T_2) = d_{KEK}(e_{KEK}(K_2)) = K_2$.
4. Perform a key search attack against each single length key (K_1, K_2) individually.

Algorithm 1: Typical Key Binding Attack

The key binding issue for double (and triple) length DES keys is well known, having been documented in [6] and exploited by [7], [9] and [11]. Indeed, this flaw has prompted a warning from the ANSI X9 Financial Services Committee [3] and is the subject of several revised proposals [1] and [2].

The API should not allow an exported key to be modified (especially the 'cut and paste' action on key components). Ideally, it should prevent the importation of such a modified or 'Trojan' key by employing some technique to verify that it is a genuine and authentic key. A typical solution is the use of a MAC on the exported key.

2.3 Key Separation

The secret key objects of PKCS #11 do allow for the specification of the use of the key for the operations of encrypting, decrypting, signing (MAC generation), verifying (MAC verification), key wrapping and key unwrapping. This is done through the use of the following attributes:

Attribute	*Value*	*Meaning*
`CKA_ENCRYPT`	`CK_BBOOL`	`TRUE` if key supports encryption
`CKA_DECRYPT`	`CK_BBOOL`	`TRUE` if key supports decryption
`CKA_SIGN`	`CK_BBOOL`	`TRUE` if key supports signatures (i.e.,authentication codes)
`CKA_VERIFY`	`CK_BBOOL`	`TRUE` if key supports verification (i.e., of authentication codes)
`CKA_WRAP`	`CK_BBOOL`	`TRUE` if key supports wrapping
`CKA_UNWRAP`	`CK_BBOOL`	`TRUE` if key supports unwrapping

Unfortunately, the API allows the specification of conflicting properties in that these attributes can be independently specified. This leads to a typical separation attack:

1. Start with the key (K) having the ability to wrap keys (i.e., act as a key encrypting key) and decrypt data.

2. Export the target key (K_{target}) under any key encrypting key (K) using the `C_WrapKey` function yielding the token $T = e_K(K_{target})$.
3. Decrypt the resultant token using the `C_Decrypt` function with K (the key wrapping key) as a data decryption key. This returns $d_K(T) = d_K(e_K(K_{target})) = K_{target}$ (i.e., the clear value of the target key).

Algorithm 2: Typical Key Separation Attack

Since the values of the attributes may be modified using the `C_SetAttributeValue` call or in the process of copying an object using the `C_CopyObject` function, it is possible for an adversary to manipulate existing keys. The PKCS #11 documentation does note that a particular implementation or token may choose to " ... permit modification of the attribute, or may not permit modification of the attribute during the course of a `C_CopyObject` call".

The problem is exacerbated in the key export/import process since an exported (or wrapped) key contains no such separation information bound to the token. As a result, any given exported key could be imported twice with different attributes. For example, the key could be imported as a key wrapping key the first time, and then as a data decrypting key the second time, thus facilitating the attack.

Clearly, greater consideration must be paid to key separation issues in the API. Ideally, the choice of attribute combination must be restrictive in order to prevent such attacks. Furthermore, such information must be cryptographically bound to the wrapped key as in [1].

2.4 Weaker Key/Algorithm

The PKCS #11 specification allows for the wrapping of a key by a second key of shorter length. Thus one need only attack the weaker key in order to recover the original key.

1. Export the target double length DES key ($K_{target} = \langle K_1, K_2 \rangle$) under a single length key (KEK) as

$$\begin{aligned} T &= e_{KEK}(K_{target}) \\ &= e_{KEK}(\langle K_1, K_2 \rangle) \\ &= \langle e_{KEK}(K_1), e_{KEK}(K_2) \rangle \\ &= \langle T_1, T_2 \rangle . \end{aligned}$$

2. Export the single length key (KEK) under itself yielding $T_{KEK} = e_{KEK}(KEK)$.
3. Attack the single length key by performing an exhaustive search.
4. Once the single length key has been recovered, one can trivially recover the original double length key.

Algorithm 3: Example Weaker Key Attack

PKCS #11 supports keys with particularly small key sizes (e.g. RC2), making the search feasible. It should not be possible to downgrade the security, by protecting a longer key with a shorter key. Similarly, it should not be possible to use a weaker algorithm when exporting keys.

We note that the previous attacks do not contradict the security claim that 'sensitive' and 'unextractable' keys cannot be compromised, since they require that the target key be exportable. What about other attacks? We focus our attention on the `C_DeriveKey` function, which has the following prototype:

```
CK_DEFINE_FUNCTION(CK_RV, C_DeriveKey)
(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hBaseKey,
    CK_ATTRIBUTE_PTR pTemplate,
    CK_ULONG ulAttributeCount,
    CK_OBJECT_HANDLE_PTR phKey
);
```

The `C_DeriveKey` supports the following mechanisms:

– `CKM_CONCATENATE_BASE_AND_KEY`, which derives a secret key from the concatenation of two existing secret keys,
– `CKM_CONCATENATE_BASE_AND_DATA`, which derives a secret key by concatenating data onto the end of a specified secret key,
– `CKM_CONCATENATE_DATA_AND_BASE`, which derives a secret key by prepending data to the start of a specified secret key,
– `CKM_XOR_BASE_AND_DATA`, which is a mechanism that provides the capability for deriving a secret key by performing the exclusive-oring of a key pointed to by a base key handle and some data, and finally
– `CKM_EXTRACT_KEY_FROM_KEY` that provides the capability of creating one secret key from the bits of another secret key.

2.5 Reduced Key Space

Using the `CKM_EXTRACT_KEY_FROM_KEY` mechanism, one can extract a subset of the bits from a given key to create a shorter key. The can be used to reduce the key space required to be searched. For example, one could extract 40 bits from a DES key to create a 40-bit RC2 key, which can then be searched by exhaustive means. The actual key space may be smaller owing to the existence of parity bits in the DES key. The remaining 24 bits (less 3 parity bits) of the original DES key can then be searched for independently. This potentially dangerous mechanism relies on the 'unextractable' flag in the key token to prevent misuse. It does not prevent an attacker from using this method to obtain a known key in the system or from compromising extractable keys.

2.6 Parallel Search

The CKM_XOR_BASE_AND_DATA provides an easy method with which to exclusive-or known patterns onto a key. This can be used to reduce the key space required to be searched by generating a large number of (known) related keys as per the method suggested by [12] and [19] and exploited by [9].

1. Generate a set of 2^{16} known related keys of original target key $\{K_i | K_i = K_{target} \bigoplus \Delta_i, i = 1, ..., 2^{16}\}$ where $\Delta_i \neq \Delta_j$ for $i, j \leq 2^{16}$, $i \neq j$ and Δ_i is a non-zero known value.
2. Using each key, encrypt a known pattern (P) and store the result in searchable database $\{C_i | C_i = e_{K_i}(P), i = 1, ..., 2^{16}\}$.
3. Search for a key by iteratively performing trial encryptions of the known pattern (P) and compare result to entries in database.
4. After 2^{39} trial encryptions on average, we expect to find a match (i.e., we find a key K_i which produces an encrypted output in the database).
5. Recover the original target key K_{target} as $K_{target} = K_i \bigoplus \Delta_i$.

Algorithm 4 : Parallel Key Search Using Related Keys

Since we know how this key is related to all the others, we known all the 2^{16} keys including the original one. This clearly demonstrates the danger of being able to modify a key as well as the true threat posed by the seemingly benign key conjuring vulnerability. Knowledge of the modification makes the attack easier but is not a requisite for the attack.

2.7 Related Key Attack

Using the CKM_XOR_BASE_AND_DATA mechanism, one can create a set of related keys with which to perform a related key attack [5], [14], [15]. This can be used to reduce 3-key 3DES to only slightly stronger than single DES (reducing the key space search to 2^{56} operations to isolate a key component). The attack is elegantly simple and easily explained. Using the related key pair $K1 =< k1, k2, k3 >$, $K2 =< k1 \bigoplus \Delta, k2, k3 >$, encrypt a plaintext P with $K1$, and then decrypt the ciphertext with $K2$ yielding P'. Then $C = e_{K1}(P)$, $P' = d_{K2}(C)$, and hence $P' = d_{K2}(e_{K1}(P))$. Using 3DES in EDE mode (the mode itself doesn't matter):

$$\begin{aligned} P' &= d_{k1 \bigoplus \Delta}(e_{k2}(d_{k3}(e_{k3}(d_{k2}(e_{k1}(P)) \\ &= d_{k1 \bigoplus \Delta}(e_{k1}(P)) \,. \end{aligned}$$

Thus, $k1$ has been successfully isolated and can be recovered independently of $k2$ and $k3$, typically by exhaustive key search. The work required on average to effect the search is 2^{56} single DES operations. Hence the cipher in triple mode has been reduced to only slightly greater than the strength of the cipher in single mode. This attack can be further enhanced by combining it with parallel key search techniques. For example, using a set of related key pairs $\{(< k_1 \bigoplus \Delta_i, k_2, k_3 >, < k_1 \bigoplus \Delta_i \Delta, k_2, k_3 > | i = 1, \ldots, 2_{16}\}$ would reduce the average search effort to 2^{40} DES operations.

The 2-key 3DES version of the attack described in [11] is not practically feasible. However, there exists a more efficient attack by first 'converting' a double length DES key into a triple length DES key using the `CKM_CONCATENATE_BASE_AND_DATA` mechanism. Following this, the three-key related key attack can be used as is.

Analysis and Implications

We return to the security claim made by the designers. Both the parallel search attack and the related key attack *contradict* the claims of the API designers. This has several implications for individual users who are reliant on the security of a PKCS #11 token. Any user with read and write access to the token has the ability to recover all token key objects. In addition, an adversary with the ability to gain access to a session (perhaps by injecting raw messages into the physical communications lines) likewise has the ability to recover keys from the token. To thwart the attack, one must prevent all unauthorized access to token objects. This intensifies the security concerns already listed by the designers and previously referred to.

We now consider a means to expand the scope of the attack to include sessions with read only access to token objects. The `C_CopyObject` provides a method to copy a read only token object and to produce as output a session object. However, since all session objects have read/write access to that session, the attacker successfully obtains a duplicate of the key object with write access. He can thus attack the session object using the methods previously described, despite only having read access to the original target object. Therefore, it is advisable to reconsider the functionality of the `C_CopyObject` call particularly with respect to the preservation of properties such as write access.

Finally, it is worth noting the work done in [9] as it directly reflects on the feasibility and speed of performing these attacks in practice. Bond and Clayton devised a parallel exhaustive key search machine using an 'off the shelf' FPGA evaluation card costing approximately $1000, which was capable of performing a 2^{39} search in 22 hours.

3 Public Key API Attacks

We now extend our focus to consider attacks involving the use of (or against) Public Key API functionality. We start by revisiting the `C_WrapKey` function and consider first the wrapping of private RSA keys by symmetric keys.

Wrapping/Unwrapping of Private Keys Using Symmetric Keys

In PKCS #11, a private key can only be exported (and imported) if it contains not only the private exponent and modulus, but also the public exponent and

CRT info. This information is BER-encoded according to PKCS #1's RSAPrivateKey ASN.1 type. The resulting string of bytes is encrypted with a secret key in CBC mode and with PKCS padding.

Attribute	*Data Type*	*Meaning*
`CKA_MODULUS`	Big integer	Modulus n
`CKA_PUBLIC_EXPONENT`	Big integer	Public exponent e
`CKA_PRIVATE_EXPONENT`	Big integer	Private exponent d
`CKA_PRIME_1`	Big integer	Prime p
`CKA_PRIME_2`	Big integer	Prime q
`CKA_EXPONENT_1`	Big integer	Private exponent d modulo $p-1$
`CKA_EXPONENT_2`	Big integer	Private exponent d modulo $q-1$
`CKA_COEFFICIENT`	Big integer	CRT coefficient $q-1 \bmod p$

The CBC-encrypted ciphertext is decrypted, and the PKCS padding is removed. The data thereby obtained are parsed as a PrivateKeyInfo type, and the wrapped key is produced. An error will result if the original wrapped key does not decrypt properly, or if the decrypted unpadded data does not parse properly, or its type does not match the key type specified in the template for the new key. The unwrapping mechanism contributes only those attributes specified in the PrivateKeyInfo type to the newly-unwrapped key; other attributes must be specified in the template, or will take their default values.

3.1 Weaker Key/Algorithm

Following this description we are immediately concerned with the choice of symmetric key algorithm (and key length) used to protect the RSA private key leading to equivalent attacks described in Section 2.4.

3.2 Private Key Modification

Consider the effect of replacing one block of the ciphertext (i.e., the wrapped key) with a different value. When the key is unwrapped, this will cause the corresponding block of plaintext as well as the following block to have different values. The rest of the key remains intact. The length of the BER encoded big number data types depends upon the size of the big numbers (typically 512, 1024 or 2048 bit numbers). In any event, they consist of at least a number of blocks. Thus an attacker can modify one of the big numbers independently of the other data in the wrapped private key (including the padding at the end). If the various key components (e.g. n, p, q, e, d, $d \bmod p-1$, $d \bmod q-1$ and $q-1 \bmod p$) are not explicitly tested for consistency, the attacker gains access to a modified 'Trojan' key in the system. This can be used to effect the Fault Analysis attacks of [8], [4] and [13]. A similar attack against PGP private keys is described in [16] and, more generally, against public key APIs in [17] and [10].

A possible solution is that encrypted private keys have a strong cryptographic method to ensure integrity of the key (e.g. MAC, hash or signature). In addition, the integrity of the key must be confirmed using simple arithmetic checks (for example, is $d_p \equiv d \bmod p$ and $n = p \cdot q$).

Wrapping/Unwrapping of Symmetric Keys Using Public Keys Techniques

PKCS #11 supports two mechanisms for wrapping symmetric keys using Public Key techniques, namely:

- `CKM_RSA_PKCS` (PKCS #1 RSA), and
- `CKM_RSA_X_509` (X.509 Raw RSA).

The `CKM_RSA_X_509` mechanism performs no padding or manipulation of data prior to encryption. It merely "…encrypts a byte string by converting it to an integer, most-significant byte first, applying 'raw' RSA exponentiation, and converting the result to a byte string, most significant byte first." The encrypted token is $T = k^e \bmod n$ where e is public exponent, k the key being exported and n the modulus. This simple method results in exported keys being vulnerable when encrypted under small public exponents.

3.3 Small Public Exponent with No Padding

The clear key is right justified in the field provided, and the field padded to the left with zeroes up to the size of the RSA encryption block (e.g. for 128-bit key $k = k_1k_2 \ldots k_{128}$ is prepended with zero bits $0_1 0_2 \ldots 0_{l-128} k_1 k_2 \ldots k_{128}$, where l is the length of the modulus). The resultant field is encrypted yielding $T = k^e \bmod n$. If $k^e < n$ (i.e., $e < \frac{log_2(n)}{log_2(k)} \leq \frac{log_2(n)}{128}$), then $T = k^e$. Thus k can be recovered as $k = T^{\frac{1}{e}}$.

Due to the speed advantages of having a small exponent with low Hamming weight, it is common for public keys to have exponents of 3 and $2^{16} + 1$. It is not uncommon to be able to specify this as an option in many APIs when generating a public key. It is thus possible that a suitable public key will exist in the system. In any event, the public keys in PKCS #11 are clear tokens and thus one can easily 'conjure' or create a public key with an exponent of 3. This weakness exists in a number of APIs [10].

3.4 Trojan Public Key

As previously mentioned, the public keys in the PKCS #11 API are clear tokens with no additional authentication checks. Thus it is possible to use any clear public key as input to the `C_WrapKey` function. This allows an attacker to use a 'Trojan' public key for which he knows the corresponding private key (typically the attacker will probably generate the key pair himself). He requests the PKCS #11 token exports the target key k under his supplied public key obtaining the response $T = k^e \bmod n$. Since the attacker knows the corresponding private exponent d, he can easily recover the key as $T^d \bmod n = (k^e)^d = k$. This simple method can be used to recover all exportable keys regardless of whether they are symmetric or private keys. It is thus clear that a public key needs to be authenticated before use to verify that it indeed has the authority to export a given key.

3.5 Trojan Wrapped Key

Similarly to the unauthenticated use of public keys, there is no method to verify that a wrapped key token is indeed authentic. Thus given a PKCS #11 device containing a private key ($< d, n >$), and knowledge of the value of the public key ($< e, n >$), the attacker proceeds as follows. He chooses an arbitrary key k, which he then 'wraps' under the known public key obtaining $T = k^e \bmod n$. He then calls the `C_UnWrapKey` function supplying this 'Trojan' wrapped key T and referencing the handle of the private key inside the device. The PKCS #11 token calculates $T^d \bmod n = (k^e)^d = k$ and imports the known k as a new key into the system. The attacker can then use k to export other keys from the device, which he can then decrypt and recover. Thus there exists a requirement to provide a means to verify the authenticity and origin of the wrapped key.

3.6 Key Separation

A symmetric key wrapped by a public key contains no separation information and can be exploited as described previously in Section 2.3.

4 Solutions

Some of these security issues can be easily addressed in the implementation of a PKCS #11 API. The more concerning issues unfortunately require a design change to the PKCS #11 standard. With the latter come the dual concerns of backwards compatibility and interoperability with other systems. A lack of backwards compatibility may be the price for a previously flawed design and a commitment to security.

The Key Conjuring and Key Binding attacks are perhaps best addressed through a change in the external key token format, particularly for wrapped keys. There exist proposals such as [1] and [2] and one can expect a decision and guidance from such influential bodies as ANSI Financial Services Committee, which will largely address the interoperability issues. Key Separation can be partially addressed by a given implementation that does not permit the conflicting use of key attributes (e.g. `CKA_WRAP` and `CKA_DECRYPT`). However, the fact that the wrapped key contains no separation information is a fundamental design flaw and like the Key Conjuring and Key Binding attacks must be addressed through a new external key token format. The Weaker Key/Algorithm attack can be prevented by a given implementation by understanding and obeying the principle that a key should not be protected by a weaker key or algorithm. The 'unextractable' and 'never extractable' flags do offer protection against the

Reduced Key Search attack. Regardless, the author is not convinced that the `CKM_EXTRACT_KEY_FROM_KEY` mechanism deserves consideration in the API. Similarly, the `CKM_XOR_BASE_AND_DATA` mechanism creates the opportunity for both the Parallel Search and Related Key attacks. Again one may question the need for such a function, particularly in its present form.

Prevention of the Private Key Modification attack requires either the use of a consistency check to confirm the integrity of the key components, which could be implementation specific, or else a revision of the encrypted RSA key token that ensures integrity through some cryptographic means, such as an encrypted hash or MAC over the token. The Small Public Exponent with No Padding attack highlights the dangers of providing raw RSA functionality. The most sensible solution is to enforce the use of a recognised padding scheme. The only concern here would be backwards compatibility. Interoperability should not be an issue since any device that uses this method to export a key is obviously vulnerable to the attack. The Trojan Public Key and Trojan Wrapped Key attacks exploit a lack of authentication of public keys used for export and wrapped keys being imported. This requires a significant change to the standard to achieve these goals.

5 Conclusions

This paper has shown the susceptibility of PKCS #11 used as an API to a number of attacks. The attacks are efficient, computationally trivial and easy to implement. Some possible solutions are presented to defend against the attacks.

Acknowledgements. I would like to express deep gratitude to my supervisor Henda Swart whose guidance and support were crucial to the successful completion of this project. I am also indebted to Alex Dent and Christine Swart for their rigorous reading of an earlier version of this paper and for their comments and suggestions, which shaped this work. I would also like to thank the current PKCS #11 editor, Simon McMahon, as well as Burt Kaliski and Magnus Nystrom for the positive and proactive manner with which they considered this research and for their helpful comments and continued efforts relating to the security of PKCS. The author is supported by the Cecil Renaud Scholarship.

References

1. ACI Worldwide, HP Atalla, Diebold, Thales e-Security, and VeriFone Inc. Global interoperable secure key exchange key block specification, 2002.
2. ACI Worldwide, HP Atalla, Diebold, Thales e-Security, and VeriFone Inc. Newly-formed payment consortium moves ahead with endorsement of secure 3DES implementation specification: Industry leaders align on new proposed key management standard, 2002.
3. American National Standards Institute (ANSI) Accredited Standards Committee (ASC) X9 - Financial Services (X9-F). Notice regarding TDES key wrapping techniques, 2002.

4. Feng Bao, Robert H. Deng, Yongfei Han, Albert B. Jeng, A. Desai Narasimhalu, and Teow-Hin Ngair. Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. In *Security Protocols Workshop*, volume 1361, pages 115–124, 1997.
5. Eli Biham. New types of cryptanalytic attacks using related keys. In *Advances in Cryptology EUROCRYPT '93*, volume 675, pages 398–409, 1994.
6. Mike Bond. Attacks on cryptoprocessor transaction sets. In *Cryptographic Hardware and Embedded System – CHES 2001 Third International Workshop*, volume 2162, pages 220–234, 2001.
7. Mike Bond and Ross J. Anderson. API-level attacks on embedded systems. *Computer*, 34(10):67–75, 2001.
8. Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology EUROCRYPT '97*, volume 1233, pages 37–51, 1997.
9. Richard Clayton and Mike Bond. Experience using a low-cost FPGA design to crack DES keys. In *Cryptographic Hardware and Embedded System - CHES 2002*, volume 2523, pages 579–592, 2003.
10. Jolyon Clulow. The design and security of public key crypto APIs, 2001.
11. Jolyon Clulow. The design and analysis of cryptographic application programming interfaces for devices. Master's thesis, University of Natal, Durban, 2003.
12. Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics, and Chip Design*. O'Reilly, Sebastopol, 1998.
13. M. Joye, A. K. Lenstra, and J.-J. Quisquater. Chinese remaindering based cryptosystems in the presence of faults. *Journal of Cryptology*, 12(4):241–246, 1999.
14. John Kelsey, Bruce Schneier, and David Wagner. Key-schedule cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES. In *Advances in Cryptology – CRYPTO '96*, volume 1109, pages 237–251, 1996.
15. John Kelsey, Bruce Schneier, and David Wagner. Related-key cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X NewDES, RC2, and TEA. In *1997 International Conference on Information and Communications Security, Beijing*, volume 1334, pages 233–246, 1997.
16. Vlastimil Klíma and Tomas Rosa. Attack on private signature keys of the OpenPGP format, PGPTM programs and other applications compatible with OpenPGP. *Cryptology ePrint Archive*, 2002.
17. Vlastimil Klíma and Tomas Rosa. Further results and considerations on side channel attacks on rsa. In *Cryptographic Hardware and Embedded System – CHES 2002*, volume 2523, pages 244–259, 2003.
18. RSA Security Inc. PKCS #11: Cryptographic Token Interface Standard. An RSA Laboratories Technical Note, Version 2.01, December 22, 1997.
19. F. Hoornaert Y. Desmedt and J. J. Quisquater. Several exhaustive key search machines and DES. In *EUROCRYPT '86*, pages 17–19, 1986.