CryptokiX: A cryptographic software token with security fixes [*]

M. Bortolozzo
Università Ca' Foscari, Venezia
mbortolo@dsi.unive.it

M. Centenaro
Università Ca' Foscari, Venezia
centenaro@dsi.unive.it

R. Focardi
Università Ca' Foscari, Venezia
focardi@dsi.unive.it

G. Steel
LSV, CNRS & ENS de Cachan, France
graham.steel@lsv.ens-cachan.fr

Abstract

PKCS#11 defines a widely adopted API for cryptographic tokens, it provides access to cryptographic functionalities and should preserve certain security properties. For example, PKCS#11 is intended to protect its sensitive cryptographic keys even when connected to a compromised host, however it is known to be vulnerable to various attacks that break this property. This paper presents a software token prototype implementing different security patches to PKCS#11 which prevent the leakage of sensitive keys. This offers a proof-of-concept that secure, fully fledged token could be realized in practice and might constitute a reference and test framework for hardware producers.

1 Introduction

PKCS#11 defines a widely adopted API for cryptographic tokens [6]. It provides access to cryptographic functionalities and should preserve certain security properties, e.g. the values of a *sensitive* key stored on a device should never become known 'in the clear'. PKCS#11 is intended to protect its sensitive cryptographic keys even when connected to a compromised host. However, it is known to be vulnerable to various attacks that break this property [5, 3].

In a PKCS#11-based API, applications initiate a *session* with the cryptographic token, by supplying a PIN. Once a session is initiated, the application may access the *objects* stored on the token, such as keys and certificates. However, acces to the objects is controlled. Objects are referenced in the API via *handles*, which can be thought of as pointers to or names for the objects. In general, the value of the handle, e.g. for a secret key, does not reveal any information about the actual value of the key. Objects have *attributes*, which may be a bitstring such as the value of a key, or a Boolean flag signalling a property of the object, e.g. whether the key may be used for encryption, or for encrypting other keys. New objects can be created by calling a key generation command, or by 'unwrapping' an encrypted key packet. In both cases a fresh handle is returned. When a function in the token's API is called with a reference to a particular object, the token first checks that the attributes of the object allow it to be used for that function.

One of the main weaknesses of PKCS#11 is in the way it defines the operations for exporting and importing sensitive keys. More precisely these commands are implemented respectively by the `WrapKey` and `UnwrapKey` API functions, the former performing the encryption of a key under another one and the latter performing the corresponding decrypt and import. The standard does not clearly separate roles for keys so that is possible to use the same key for 'conflicting' purposes: for example, a key could be have its `decrypt` and `wrap` attributes set, enabling an easy wrap-decrypt attack described below.

We have experimented that most of the commercially available tokens are either insecure or drastically cut in their functionalities, e.g. by completely disabling wrap and unwrap [2]. Intermediate approaches are possible: the standard can be patched without necessarily removing the wrapping functionality [4]. To this aim, we have developed Cryptoki-fiXed (CryptokiX, for short), a software prototype of a PKCS#11 token, whose security is highly configurable by selectively enabling different patches. Our starting point is openCryptoki [1], an open-source PKCS#11 implementation for Linux including a software token for testing. The analysis of the openCryptoki software token has revealed that it is subject to

[*]Work partially supported by Miur'07 Project SOFT: "*Security Oriented Formal Techniques*"

attacks revealing the value of sensitive keys [2]. This is in a sense expected, as it implements the standard 'as is', i.e., with no security patches.

CryptokiX offers a proof-of-concept that secure, fully fledged token could be realized in practice and might constitute a reference and test framework for hardware producers.

2 CryptokiX

CryptokiX extends openCryptoki by providing the following patches:

Conflicting attributes. It is insecure to allow the setting of some configurations of attributes on the same key. In our token we can configure the set of conflicting attributes.

Sticky attributes. Some attributes once set should never been unset while some others once unset must not be enabled again. For example a sensitive key should never be set non-sensitive. This is also useful when combined with the 'conflicting attributes' patch above: if two attributes are conflicting we certainly want to avoid that they are separately set and unset.

Wrapping formats. It is known from previous work that it in not sufficient to specify a non-conflicting attribute policy. A wrapping format must also be used that correctly binds key attributes to the key. This prevents attacks where the key is unwrapped twice with conflicting attributes [5, 3]. Some devices are known to already include such wrapping formats, such as the Eracom ProtectServer [4].

Secure templates. We limit the set of admissible attribute combinations for keys so to avoid that keys ever assume conflicting roles at runtime. This is configurable at the level of the specific PKCS#11 operation. For example, we can define different secure templates for key generation and unwrapping.

CryptokiX will be soon available for download at `http://secgroup.ext.dsi.unive.it/cryptokix` together with the examples showing the effectiveness of the different patches. The four distinct security fixes are now presented in details.

Conflicting attributes. The first patch adds a check to every functions setting the attributes of an object in order to verify that they respect the conflicting-attribute policy. For the moment, this policy can only be configured in the source code of the prototype, but we are currently working on a configuration run-time tool. The hard-coded policy, available on the downloadable software token, define conflicts between `wrap` and `decrypt` attributes and `unwrap` and `encrypt` ones.

Enabling this patch alone, keys having conflicting roles cannot be created, but it will be possible to turn the corresponding conflicting attributes on and off to bypass the policy as in the following example. Recall that PKCS#11 offers handle to objects stored inside a token. In all the examples, variables whose name starts by '`h_`' are handle to an object.

```
h_myKey = GenerateKey();
SetAttribyte(h_myKey, {decrypt => false});
SetAttribute(h_myKey, {wrap => true});
wrapped = WrapKey(h_myKey, h_mySuperSecretKey);
SetAttribyte(h_mykey, {wrap => false});
SetAttribyte(h_mykey, {decrypt => true});
yourSuperSecretKey = Decrypt(h_myKey, wrapped);
print "oops: " + yourSuperSecretKey;
```

Sticky attributes. This patch sets a sticky policy for a key attributes. An attribute could be *sticky_on*, meaning that once set it cannot be subsequently unset, *sticky_off*, avoiding an attribute to be set after it has being unset, or read-only. The implemented token enforce a policy where `sensitive`, `wrap`, `unwrap`, `encrypt` and `decrypt` are sticky_on attributes, while `extractable` is a sticky_off one and the remaining attributes are read-only.

This patch alone does not provide any substantial security, but it is indeed useful to prevent the attack shown above when also the conflicting attributes patch is enabled. Notice, however, that is still possible to retrieve the value of a sensitive key by creating two different keys inside the token whose values are the same and which are used for conflicting roles:

```
wrapped = <<generate a random bytestream>>;
h_decKey = UnwrapKey(h_unwrapKey, wrapped, {decrypt => true, wrap => false});
h_wrapKey = UnwrapKey(h_unwrapKey, wrapped, {decrypt => false, wrap => true});
```

Wrapping formats. To prevent the last kind of attack the `WrapKey` function must keep track, in the produced ciphertext, of the original values of wrapped key's attributes so that the `UnwrapKey` can restore the original ones. This is being implemented by adding a message authentication code of the wrapped attributes inside the ciphertext returned by the wrapping operation. The unwrap function will then ignore the attributes given in its third input parameter and use instead the ones obtained by decrypting the wrapped key.

This patch alone is not enough to obtain a secure token, indeed for example it will be possible to generate a key with both `decrypt` and `wrap` attributes set. Only if all the three patches are enabled together we are guaranteed to have a secure device [4].

Secure templates. This patch limit the set of admissible attribute combinations for keys so to avoid that they ever assume conflicting roles at run-time. This is configurable at the level of the specific PKCS#11 operation. For example, it is possible to define different secure templates for key generation and unwrapping. The idea of secure templates is general and can be instantiated with different sets of admissible templates. The interesting attributes are wrap, unwrap, encrypt and decrypt.

We consider the following configuration for symmetric keys:

Key generation: we allow three possible templates:

1. wrap and unwrap, for exporting/importing other keys;
2. encrypt and decrypt, for cryptographic operations;
3. none of the four attributes, the default template if none of the above is specified.

Key import: we allow a single template with unwrap and encrypt set and wrap and decrypt unset. Any key which is imported either with `CreateObject` or with `Unwrap` is given this restricted set of attributes.

Moreover all keys have modifiable unset and sensitive set. Templates for key generation are rather intuitive and correspond to a clear separation of key roles, which is the base of any meaningful patch. The single template for key import, instead, is less obvious and might appear too restrictive. The idea is to fully allow wrapping and unwrapping of keys while 'halving' the functionality of imported/unwrapped keys: an unwrapped key can only be used to unwrap other keys or to encrypt data, wrapping and decrypting with such a key are forbidden. This, in a sense, offers an asymmetric usage of imported keys: to achieve full-duplex encrypted communication two devices will have to wrap and send a freshly generated key to the other device. Once the keys are unwrapped and imported in the device they can be used to encrypt outgoing data in the two directions. Notice that imported keys can never be used to wrap sensitive keys, which is clearly dangerous and typically a source of attacks. This solution has been proved to be secure by model-checking [2].

3 Conclusion

CryptokiX will be soon available for download at `http://secgroup.ext.dsi.unive.it/cryptokix` to anyone how want to experiment with it. Source code for some of the known attacks to PKCS#11 (containing all the ones presented here) can also be fetched there in order to easy the testing operation.

We believe this prototype is useful for educational purpose and also as a proof-of-concept of the fact that secure, full-fledged tokens can be obtained in practice. The source code of our prototype could be useful to anyone interested in learning how to patch an existing token or producing a new one.

Future Work We plan to let all the patches be highly configurable at run-time instead of requiring you to re-build the software tokens every time you want to change a policy.

References

[1] openCryptoki. `http://sourceforge.net/projects/opencryptoki/`.

[2] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. How Smart is Your Smartcard? (Attacking and Fixing PKCS#11 Security Tokens). Submitted for publication.

[3] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 331–344, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.

[4] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 2009. To appear.

[5] J. Clulow. On the security of PKCS#11. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, pages 411–425, 2003.

[6] RSA Security Inc., v2.20. *PKCS #11: Cryptographic Token Interface Standard.*, June 2004.