Fixing PKCS#11 by key-diversification

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

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

Abstract

PKCS#11 is a security API for cryptographic tokens. It is known to be vulnerable to attacks which can directly extract, as cleartext, the value of sensitive keys. In particular, the API does not impose any limitation on the different roles a key can assume and this gives the possibility to perform conflicting operations such as asking the token to wrap a key with another one and then to decrypt it. The typical solution, in the literature, is to impose policies restricting key roles. Here we take a different perspective and we propose a 'fix' based on *key diversification*. The idea is to prevent conflicting roles by always deriving different keys for different roles. This greatly simplifies the delicate task of exporting/importing keys. In a technical paper, submitted for publication, we prove via type-checking that the fix discussed here preserves the secrecy of sensitive keys.

Introduction. PKCS#11 (also known as Cryptoki) defines a widely adopted API for cryptographic tokens [10]. It provides access to cryptographic functionalities while, in principle, providing some security properties. More specifically, the value of keys stored on a PKCS#11 device and tagged as *sensitive* should never become known 'in the clear' out of the token, even when connected to a compromised host. Unfortunately, PKCS#11 is known to be vulnerable to various attacks that break this property [2, 5, 7].

An application initiate a *session* with a PKCS#11 compliant device by supplying a PIN. It then may access the functionalities provided by the token. There may be various *objects* stored in the token, such as cryptographic keys and certificates. Objects are referenced via *handles* to permit, e.g., that a cryptographic key is used without necessarily knowing its value: we can ask a token to encrypt some data just providing a handle to the encryption key. The value of a key is one of the *attributes* of the enclosing object. There are other attributes to specify the various roles a key can assume: each different API call can, in fact, require a different role. For example, decryption keys are required to have attribute `CKA_DECRYPT` set, while key-encrypting keys, i.e., keys used to encrypt other keys, must have attribute `CKA_WRAP` set.

The attacks on PKCS#11 we consider here [2, 5, 7] are at the lavel of the API [1, 4, 8], i.e., the attacker is assumed to control the host on which the token is connected and can perform any sequence of (legal) API calls. The crucial functionalities of PKCS#11 are the ones for exporting and importing sensitive keys, called `C_WrapKey` and `C_UnwrapKey`. The former performs the encryption of a key under another one, giving as output the resulting ciphertext, and the latter performs the corresponding decrypt and import in the token. They allows for exporting and reimporting keys, in an encrypted form. As already mentioned, having a wrapping key (`CKA_WRAP`) which can also be used for decryption (`CKA_DECRYPT`) is dangerous and leads to the following simple 'wrap-decrypt' API-level attack:

```
h_myKey = C_GenerateKey({CKA_DECRYPT, CKA_WRAP});
wrapped = C_WrapKey(h_sensitiveKey, h_myKey);
leak = C_Decrypt(wrapped, h_myKey);
```

First, we ask the token to generate a new key with attributes `CKA_DECRYPT`, `CKA_WRAP` set. Then, we use this key to wrap an existing sensitive key referenced by `h_sensitiveKey`. Finally, we ask

the token to decrypt the resulting ciphertext using again the freshly generated key. Since it is the same key used for wrapping, we obtain the value of the sensitive key in the clear.

In a recent work with other authors [2], we have shown that the state of the art in PKCS#11 security tokens is rather poor: many existing commercially available devices are vulnerable to attacks similar to the above one; the secured ones, instead, prevent the attacks by completely removing wrapping functionalities. However, it has been shown that the API can be 'patched' without necessarily cutting down so much its functionalities [2, 7]: this can be done by (*i*) imposing a policy on the attributes so that a key cannot be used for conflicting operations; (*ii*) limiting the way attributes can be changed so to avoid that conflicting attributes are set at two different instants; (*iii*) either adding a wrapping format which binds attributes to wrapped keys [7] or limiting very carefully the usage of imported keys to a subset of non-critical functions [2].

In this work, we propose a new fix to PKCS#11 based on key-diversification, a standard cryptographic technique to derive a new key from a known one. Key-diversification is already implemented in PKCS#11 but it is up to the application whether to adopt it or not. We have seen, in fact, that the API allows applications to generate a new key with certain attributes set such as `CKA_DECRYPT` or `CKA_WRAP`, representing possible roles. Our idea, instead, is to explicitly require that keys are always diversified when used in a certain role so that it becomes impossible any confusion between different usages: keys for different roles will always be different.

Key-diversification is, of course, not a new idea but it is the first time, to the best of our knowledge, that it is proposed in the setting of cryptographic tokens, as a systematic mechanism to secure key management. There are numerous advantages: (*i*) we obtain a clear separation of roles with no need of imposing a, possibly complicate, conflicting attribute policy; (*ii*) we can make relevant attributes read-only and require that they can only be 'set' when diversifying a key, gaining great control on roles assumed by keys during their life-cycle; (*iii*) we can safely import a new key by simply giving no role to it and, later on, derive from it keys with specific, useful roles: we do no need any wrapping format for attributes and we do not limit in any way the possible future usages of the (derived) keys; (*iv*) existing application should still work on our fixed API since the modification is only 'internal': the API still looks the same, but key are diversified on-the-fly when needed for a certain role. Notice that this is not necessarily the case when using wrapping formats as discussed in [7] since the length of wrapped keys changes due to the presence of (a MAC of) the attributes.

The proposed fix. In our variant of PKCS#11, devices only store special keys that we call *seeds*, i.e., keys used to obtain actual keys by diversification. In a sense, the token will not base its security on the usual attributes `CKA_ENCRYPT`, `CKA_DECRYPT`, `CKA_WRAP` and `CKA_UNWRAP` since it will always internally diversify keys for the different roles. In particular, the key used for encryption and decryption will be different from the one used for wrapping and unwrapping, even if the handle specified by the application is the same (pointing to the same seed).

In order to keep track ok key integrity we use the attributes `CKA_ALWAYS_SENSITIVE` and `CKA_TRUSTED`. The former attribute means that the relative key has always been sensitive in its entire life; it cannot be set when importing a key (see [10], Table 15 footnotes 4 and 6) and it is automatically set by the token when generating sensitive keys. The latter attribute means that the key has been loaded into the token by the *Security Officer*, a token administrator who is supposed to operate in an isolate and safe environment.

We then require that wrapping keys are always derived by seeds with `CKA_ALWAYS_SENSITIVE` or `CKA_TRUSTED` set. In this way it will not be possible for an attacker to wrap a sensitive key under a known one. Following the standard (version 2.20), it is like all keys have the

CKA_WRAP_WITH_TRUSTED attribute set, i.e., they can only wrapped under trusted key, extended to always sensitive wrapping keys. We will discuss more on this issue in the conclusion.

Diversified keys will be calculated on-the-fly at need and then thrown away. Key diversification may be implemented in many different ways that are out of the scope of this work. We assume the existence of a function `diversify` taking as input a *role tag* `t` and a key `k` and returning the diversification of `k` suitable for the roles identified by `t`. To illustrate, we show the code for `C_Decrypt` and `C_Wrap` commands. We use tags `__DATA__` and `__KEYS__` to respectively refer to roles operating on data, such an encryption and decryption, and to roles operating on keys, such as wrapping and unwrapping. We also let `getKeyValue` be a function retrieving the value of a key from its handle. It is used internally for the implementation and, of course, is not part of the API.

```
C_Wrap(h_key, h_wrap) {
  if (!AlwaysSensitive(h_wrap) && !Trusted(h_wrap))
    return CKR_OPERATION_NOT_PERMITTED;
  w1 = getKeyValue(h_wrap);
  w2 = diversify(__KEYS__, w1);
  k = getKeyValue(h_key);
  return encrypt(k, w2);  }
```

```
C_Decrypt(data, h_key) {
  k1 = getKeyValue(h_key);
  k2 = diversify(__DATA__, k1);
  return decrypt(data, k2);
}
```

The attack shown in the introduction is no more effective since the decrypt and wrap commands use different keys: the wrap-decrypt sequence would, in this case, give the decryption with `k2` of a key wrapped under `w2`.

Notice that the proposed solution is completely transparent to the user: it automatically ensures that the same key is never used for encrypting and decrypting both data and keys, since different role tags are used to derive the respective keys. It must be noted that this breaks the compatibility with other devices. Indeed, a key wrapped by a token implementing this patch cannot be correctly imported by a device compliant to the standard (and vice versa), the same holds for encrypted data.

Notice that, for two devices to communicate (exchanging encrypted data and keys) a common master key has to be shared. This can be installed by the *Security Officer* using the above mentioned CKA_TRUSTED attribute. Additionally, it is possible to write a version of our patch where ALWAYS_SENSITIVE keys can be wrapped and imported on another device, making it possible to share new trusted keys among different tokens. This extends the pretty static use-case of trusted keys. The important point is to have in mind what kind of seeds can be wrapped/unwrapped. If they are required to be always sensitive, we have to check that the attribute is set before wrapping and then set it after unwrapping. For lack of space, we omit the detail.

Concluding remarks and type-based analysis. We have presented a new patch of PKCS#11 based on key-diversification. It gives to the token the (crucial) responsibility of separating roles for each key, greatly simplifying the policy on attributes necessary to prevent known API-level attacks. In fact, it seems more appropriate to let the devices implement the basic principle of key-separation, so to guarantee the security of sensitive keys stored on PKCS#11 tokens. Giving the user the possibility of freely setting the CKA_ENCRYPT, CKA_DECRYPT, CKA_WRAP and CKA_UNWRAP attributes has proved to be a really bad design choice causing a series of attacks [5, 7].

Starting from version 2.20, RSA has introduced a new attribute to the standard, called CKA_WRAP_WITH_TRUSTED. The idea is that only the security officer is able to import a trusted key, i.e., with CKA_TRUSTED set, into a token and any sensitive key, which one would like to

protect, has to have its `CKA_WRAP_WITH_TRUSTED` attribute set, meaning that it can only be wrapped under a trusted key.

The idea of trusted key is really helpful and we exploit it in our patch but we fail to understand the sense of `CKA_WRAP_WITH_TRUSTED`. In fact, in our patch we implicitly assume that any sensitive key should be wrapped with a trusted one (or with an always sensitive one, that we in fact consider as trusted). Again, having the user setting what should or should not be wrapped with a trusted key is an excessive flexibility that might lead to new API level attacks: one such key could be in fact be wrapped and then unwrapped on another device with `CKA_WRAP_WITH_TRUSTED` unset. At this point the discussed wrap-decrypt attack might be performed on the key. (It is, by the way, rather irritating that RSA still ignores the well-know problem of conflicting roles and never mentions API-level attacks in the new versions of the standard.)

In a work submitted for publication we have developed a type system suitable to check the security of PKCS#11 APIs. It is a tool that could help developers and hardware producers to better understand the crucial bugs affecting the design and implementation of this standard and to prove the correctness of possible patches. In fact, we have been able to show that the key diversification fix presented here is indeed sound (i.e., it preserves sensitive keys from being leaked). In our type-based analysis we also generalize the idea of role tags so to obtain, by diversification, keys which are able to wrap keys of various type. For example, we could have keys for wrapping sensitive encryption/decryption keys and keys for wrapping other trusted or always sensitive seeds. It is enough to have different tags for different kind of wrapping keys.

As a future work, we intend to implement the key diversification patch on a software token. As already done for other fixes [2, 3, 6] the starting point will be be the open-source project openCryptoki [9].

References

[1] M. Bond. Attacks on cryptoprocessor transaction sets. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, volume 2162 of *LNCS*, pages 220–234, Paris, France, 2001. Springer.

[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)*, pages 260–269, Chicago, Illinois, USA, October 2010. ACM.

[3] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. CryptokiX: a cryptographic software token with security fixes. In *Proceedings of the 4th International Workshop on Analysis of Security APIs (ASA)*, Edinburgh, UK, July 2010.

[4] R. Clayton and M. Bond. Experience using a low-cost FPGA design to crack DES keys. In *Cryptographic Hardware and Embedded System - CHES 2002*, pages 579–592, 2002.

[5] Jolyon Clulow. On the security of PKCS#11. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, volume 2779 of *Lecture Notes in Computer Science*, pages 411–425, Cologne, Germany, September 2003. Springer.

[6] cryptokiX. `http://secgroup.ext.dsi.unive.it/cryptokiX`.

[7] 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, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.

[8] D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.

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

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