Run-time analysis of PKCS#11 attacks

Gianluca Caiazza, Riccardo Focardi and Marco Squarcina

Dipartimento di Scienze Ambientali, Informatica e Statistica,
Università Ca' Foscari, Venezia, Italy

Abstract

The goal of this paper is to report on the development of a tool aimed at the automatic detection of attacks against PKCS#11 devices. Instead of modifying or configuring the API, we propose a stateful run-time monitor which is able to track key usage over time, for the identification of operations that might result in the leakage of sensitive keys. We briefly report on the components developed for implementing the monitor and discuss new challenges and open issues.

1 Introduction

Cryptography is one of the dominant technologies to provide security in various settings and cryptographic hardware and services are becoming more and more pervasive in everyday applications. The interfaces to cryptographic devices and services are implemented as *Security APIs* that allow untrusted code to access resources in a secure way. These APIs provide various functionalities such as: the creation or deletion of keys; the encryption, decryption, signing and verification of data under some key; the import and export of *sensitive* keys, i.e., keys that should never be revealed as plaintext outside a smart card or hardware security modules (HSMs).

In the last decade a multitude of attacks has been found both on HSMs and smart cards (see, e.g., [1, 3, 4, 8]). Some attacks are related to the key wrapping operation: For example, attacks on the IBM CCA interface are related to the improper bound, provided by the XOR function, between the attributes of a wrapping key and the usage rules [3], and attacks on the PKCS#11 security tokens can be mounted by assigning particular sets of attributes to the keys, and by performing particular sequences of (legal) API calls [4]. Other attacks, e.g., the ones on PIN processing APIs, are based on formats used for message encryption [10], or on the lack of integrity of user data [6].

In the literature, there are many results about the analysis of existing security APIs and the development of new, secure APIs (see, e.g. [5, 7, 11, 12, 14, 15]). In this paper we explore a different approach. Instead of fixing the API or proposing a new secure one, we study how to detect (and possibly prevent) attacks using a *stateful* run-time monitor. The idea is to track how keys are used so to detect possible conflicting operations that might lead to the leakage of a sensitive key value. For example, by keeping the information that a certain key has been used to wrap other sensitive keys we are able to detect if this key will be successively used to decrypt ciphertexts, detecting the so-called *wrap/decrypt* attack [9].

A stateful monitor presents many challenges. In particular it should (*i*) track the usage of a key without exposing its value; (*ii*) never break applications unless it is certain that the detected sequence is an attack; (*iii*) work across many devices to prevent attacks where “conflicting” operations are performed on different devices sharing sensitive keys, (*iv*) be hard to bypass, even when the host where the application is running is compromised. We have tackled some of the above challenges and implemented a proof of concept in the form of a library wrapper with an analyzer: the wrapper intercepts all PKCS#11 calls and logs them without leaking key values; interestingly, the analyzer is able to detect all attacks on PKCS#11 key management reported in [12, 13] on the produced logs.

Our approach is very closely related to Caml Crush [2], a PKCS#11 Filtering Proxy that can be configured to prevent dangerous PKCS#11 commands and mechanisms. Caml Crush does not currently implement any stateful filtering [1] which is instead our main focus. Notice that without a stateful filter the only way to prevent wrap/decrypt attacks from an external monitor or proxy would be drastically reduce the API functionality, for example by forbidding key wrapping.

2 Run-time Analysis

In order to perform the run-time detection of attacks against PKCS#11 devices, we propose a software layer that wraps the existing PKCS#11 library interface. The wrapper provides an hooking facility for analysing selected API calls typically used by attackers to perform malicious actions, e.g., leaking the values of sensitive keys stored in the device. This tool records the actions accounted by the underlying library and provides an abstraction over the API calls to the run-time component responsible of identifying security risks.

2.1 PKCS#11 Wrapper

We have implemented a wrapper of the full PKCS#11 API. However, since our tool is in an early stage of development, we only keep track of a subset of PKCS#11 functions. In fact, despite of the small number of functions considered, all the attacks introduced in [9] can be carried out by exploiting the following API calls: `C_GenerateKey`, `C_SetAttributeValue`, `C_WrapKey`, `C_UnwrapKey`, `C_Encrypt` and `C_Decrypt`. For the remaining functions, the wrapper transparently performs the corresponding call.

Additionally, we keep track of meaningful key attributes which define the capabilities of a given key and its security level: `TOKEN`, `SENSITIVE`, `ENCRYPT`, `DECRYPT`, `WRAP` and `UNWRAP`. If a key has the `TOKEN` attribute set to *true*, then it is stored inside the token. A `SENSITIVE` key disallows the key value to be read directly. All the other considered attributes are used to enable or disable specific functions using the key to which they are associated.

The wrapper produces at run-time a log file listing, for each of the aforementioned PKCS#11 calls, an entry amenable to further analysis. Log entries contain information about the function name, the identifier of each key involved in the operation and, when supported by the current call, the attribute template. To track keys over time, we compute key identifiers by encrypting a fixed value under the inspected key, using a predefined encryption mechanism compatible with the specific key. The result of the operation is unique for each key, in practice, since the probability of obtaining the same result given two different keys is negligible.

An example of a log entry associated with a `C_SetAttributeValue` call is the following:

```
SetAttributeValue F3C7719288E3CFC53A8355E6B06F16D4 110110,
```

where the first field is the function name and the second one is the key identifier obtained by encrypting a fixed message with the actual key. The last field represents the attribute template used in the current operation, in this case the attributes `TOKEN`, `SENSITIVE`, `DECRYPT` and `WRAP` are set, whilst `ENCRYPT` and `UNWRAP` are unset.

[1] See `https://github.com/ANSSI-FR/caml-crush/blob/master/doc/FILTER.md` in section "Filtering PKCS#11 mechanisms options"

2.2 Attack Detection

Our stateful analyser parses the log file generated by the wrapper in order to account for the run-time identification of attacks. Specifically, the tool keeps track of all the keys stored within the token and their templates. It also records the wrapping operations and the encryptions occurred during the logger lifespan. Keys which are deleted in the token at a certain point in time are not discarded by the analyser and are still considered meaningful. This captures attacks where a key is deleted and then reimported in the device.

To briefly illustrate the operating principle of the tool, we consider a simple wrap/decrypt attack starting from the relevant part of the log file as presented in Listing 1. For ease of reading, in this example we make use of mnemonic identifiers for the keys involved in the attack. The first entry of the log file accounts for the creation of key A. After parsing this entry, the analyser immediately rises a warning since key A is generated with a well known dangerous combination of attributes `WRAP` and `DECRYPT` set to *true*. B is the sensitive key that we suppose to be already stored inside the token. The attack proceeds by performing a wrap operation on key B using the session key A. The result of the wrapping operation is the hex string `DBB2A24842E91C5D`, as summarized in the second log entry. The last step of the attack (see log entries three and four) consists of decrypting the value of the wrapped key B with key A, finally obtaining the plain text value of the sensitive key B (`75C3234CE14954D5`).

Listing 1: Log file produced by a wrap/decrypt attack

```
1  GenerateKey   KeyIdA              000110
2  WrapKey       KeyIdA              KeyIdB              DBB2A24842E91C5D
3  DecryptInit   KeyIdA
4  Decrypt       DBB2A24842E91C5D    75C3234CE14954D5
```

Our analyser is able to immediately identify the wrap/decrypt attack since the input value of the decrypt operation performed with key A has been previously obtained as a result of a wrap operation with the same key A. Similar strategies are applied to detect (and potentially prevent) other attacks such as wrap/unwrap in which a sensitive key is wrapped and then unwrapped with the `SENSITIVE` attribute disabled in order to access the key value and classical encrypt/unwrap and re-import attacks as described in [12, 13].

3 Discussion

Our approach hinges on the ability of tracking the usage of a key without exposing its value. To do so, we identify each key with the result of the encryption of a fixed value with the actual key. It follows that all the keys should provide the capability of performing encrypt operations. This is not the case for keys with the `ENCRYPT` attribute disabled, since encryption for these keys is forbidden. We take on this issue by temporary altering the attribute in order to allow the encrypt operation needed for key identification. Even if for a short amount of time, we may enable a malicious user to perform an encrypt/unwrap attack using a previously safe key, i.e., a key without the conflicting attributes `ENCRYPT` and `UNWRAP` set to *true*. The checks implemented in our analyser allow the detection of such attack, but we are fully aware that tampering with the keys is typically unwanted.

Another limitation of our solution is the overhead due to the multiple encrypt calls needed to track keys. Indeed, our first prototype requires keys to be identified by the mean of an encryption call each time they are used. We are still investigating possible optimizations, but we may be able to drastically lower the number of encrypt calls by pairing the key identifier with the ID attribute assigned to the key by the token. We also plan to develop a formal framework for the detection of run-time attacks: we argue that a sound approximation of the dynamic evolution of the keys in PKCS#11 devices would be of practical interest and would make it possible to improve the performance of our tool.

References

[1] R. Anderson. The correctness of crypto transaction sets (discussion). In *Revised Papers from the 8th International Workshop on Security Protocols*, pages 128–141, London, UK, 2001. Springer Verlag.

[2] Ryad Benadjila, Thomas Calderon, and Marion Daubignard. Caml Crush: A PKCS#11 Filtering Proxy. In *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, pages 173–192, 2014.

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

[4] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proc. of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 260–269. ACM, 2010.

[5] C. Cachin and J. Camenisch. Encrypting keys securely. *IEEE Security & Privacy*, 8(4):66–69, 2010. IEEE Computer Society.

[6] M. Centenaro, R. Focardi, F.L. Luccio, and G. Steel. Type-Based Analysis of PIN Processing APIs. In *Proc. of the 14th European Symposium on Research in Computer Security (ESORICS'09)*, volume 5789 of *LNCS*, pages 53–68. Springer Verlag, 2009.

[7] Matteo Centenaro, Riccardo Focardi, and Flaminia L. Luccio. Type-based analysis of key management in PKCS#11 cryptographic devices. *Journal of Computer Security*, 21(6):971–1007, 2013.

[8] R. Clayton and M. Bond. Experience using a low-cost FPGA design to crack DES keys. In *Cryptographic Hardware and Embedded System (CHES'02)*, volume 2523 of *LNCS*, pages 579–592. Springer Verlag, 2003.

[9] J. Clulow. On the security of PKCS#11. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, volume 2779 of *LNCS*, pages 411–425. Springer Verlag, 2003.

[10] J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.

[11] V. Cortier and G. Steel. A generic security API for symmetric key management on cryptographic devices. In *Proc. of the 14th European Symposium on Research in Computer Security (ESORICS'09)*, LNCS, pages 605–620. Springer Verlag, 2009.

[12] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, November 2010. IOS Press.

[13] R. Focardi, F.L. Luccio, and G. Steel. An introduction to security api analysis. In *FOSAD*, pages 35–65, 2010.

[14] S.B. Fröschle and G. Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In *Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security, (ARSPA-WITS'09)*, volume 5511 of *LNCS*, pages 92–106, York, UK, 2009. Springer Verlag.

[15] S. Kremer, G. Steel, and B. Warinschi. Security for key management interfaces. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF'11)*, pages 266–280. IEEE Computer Society Press, June 2011.