Secure your PKCS#11 token against *API attacks!*[*]

M. Bortolozzo, G. Marchetto, R. Focardi
Università di Venezia, Italy
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 devices such as USB crypto-tokens and smartcards. Despite its widespread adoption, PKCS#11 is known to be vulnerable to various attacks that enable the extraction (as plaintext) of sensitive keys stored on the device. These attacks have been formalized and analyzed via model-checking, so as to automatically find flaws on specific subsets of PKCS#11 and on given device configurations. The analyses, however, are performed on an abstract model of the standard and the 'theoretical' attacks have to be tried by hand on real devices. In this paper we shortly describe a new tool, named *API attacks!*, that aims at automatically performing the above mentioned analyses on real PKCS#11 devices. We believe this tool might be helpful both to hardware developers, willing to improve the security of their existing and new devices, and to system administrators that might want to check their device is configured is a secure way before distributing it to end-users.

1 Introduction

PKCS#11 defines a widely adopted API for cryptographic devices such as USB crypto-tokens and smartcards. As well as providing access to cryptographic functionality, the interface is supposed to preserve certain security properties, e.g. no matter what sequence of commands is called by the application, the values of keys stored on the device and marked as *sensitive* should never become known 'in the clear'. However, PKCS#11 is known to be vulnerable to various attacks that compromise this property.

PKCS#11 has been formalized and analyzed via model-checking, allowing the automatic detection of attacks on specific subsets of the API and on given device configurations [4]. The analyses, however, are performed on an abstract model of the standard and the 'theoretical' attacks have to be tried by hand on real devices. Particular PKCS#11 compatible devices may implement the standard in subtly different ways to try to prevent the attacks. What is needed is a way to link the abstract model checking analysis to the API as implemented on real devices.

In this paper we describe a new tool, named *API attacks!* [1], that aims at automatically performing the above mentioned analyses on real PKCS#11 devices. In summary, the tool (i) attempts a set of known attacks reporting the results; (ii) reads the actual subset of PKCS#11 implemented on the token; (iii) generates a model of the specific subset of the standard and gives it as an input to the model checkers NuSMV and SATMC; (iv) parses the results of the model checkers and tries to mount the attacks on the real token;[1] (v) performs custom attacks and static checks on the actual token configuration, pointing out potential sources of flaws.

We believe this tool might be helpful to hardware developers, willing to improve the security of their existing and new devices. In order to circumvent the flaws on PKCS#11, hardware developers usually modify or patch the standard via proprietary extensions. *API attacks!* could be used to try all the existing known attacks on new extensions, reporting where the attack possibly fails, and, more interestingly, could incorporate a model of a new proposed extension so to analyse it via model checking looking for possible new flaws or variants of existing ones. It might be the case, in fact, that a new patch blocks all

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

[1]This last feature is currently under development: at the present state the tool just reports the results of (iii) to the user.

the known attacks but can be circumvented by non trivial variations of the attack sequences. This could be detected automatically via model-checking, once the proprietary extension is modelled in the tool.

The tool might also be interesting for system administrators, who want to configure tokens in a secure way before distributing them to end-users. Attacks, in fact, are sometime based on 'weak' token configurations where, e.g., keys can be used with different, conflicting, roles (see section 2). We thus believe that having the possibility of automatically checking a specific configuration against known attacks (and variants of those, via model-checking) could be extremely valuable.

2 Attacks on PKCS#11

In this section, we illustrate some of the attacks checked by the tool. They are all sequences of legal API invocations that lead to the leakage of *sensitive* keys, i.e., keys intended to be securely stored in the token and never extracted, unless encrypted under other suitable keys called *key encryption keys*. This latter event is useful, e.g., when we need to share a new key between two devices that already share a key encryption key *kek*: the new key k is *wrapped* under *kek* obtaining a ciphertext $\{k\}_{kek}$. This wrapped key is exported from the first device and imported in the second one. Only when the ciphertext is inside the second device, the *unwrap* occurs: k is security stored and, from now on, it can be used for encrypted communication between the two devices.

This simple wrap/unwrap mechanism is often source of attacks, if we mix the role of the keys. The standard, in fact, does not forbid having keys with attributes that specify different uses like, e.g., *wrap* and *decrypt*. Unfortunately this leads to simple attack sequences as the following one. From now on, we write $\&k$ to denote the *handle* of key k stored in the device.

1. `wrap(`$\&k$`,`$\&k$`)` gives $\{k\}_k$
2. `decrypt(`$\{k\}_k$`,`$\&k$`)` gives k

This attack is a variant with just one key of the key separation attack presented in [2]. Intuitively, key k is wrapped under itself, via a call to `wrap`, and the obtained ciphertext is decrypted with k, by calling `decrypt`, obtaining k as plaintext. Notice that this decryption occurs in the token with no knowledge of k (only the handle $\&k$ is needed).

It seems thus important to forbid this double role on the same key. This can be done, e.g. by 'patching' the API so that attributes *wrap* and *decrypt* are sticky, i.e., once set they cannot be unset, and can never be set together. However a subtler variant of the attack might be performed as follows. Let k_e be a key generated by the attacker and k_u a key, stored in the device, that can be used both to *unwrap* and *encrypt*:

1. `encrypt`(k_e, $\&k_u$) gives $\{k_e\}_{k_u}$
2. `unwrap`($\{k_e\}_{k_u}$, $\&k_u$) imports k_e in the device returning $\&k_e$
3. `set_wrap`($\&k_e$) sets the *wrap* attribute for k_e
4. `wrap`(k, $\&k_e$) gives $\{k\}_{k_e}$
5. the intruder decrypts $\{k\}_{k_e}$ obtaining k

This attack has been discovered in [4], via model-checking. Intuitively, the intruder encrypts his key under k_u (1). This allows him to import the key via an `unwrap` call (2). Once the key is in the device, he sets the *wrap* attribute (3) and just wraps the sensitive key k with k_e (4). The intruder can now decrypt $\{k\}_{k_e}$ with k_e, which he knows, so this last event is performed outside the device.

Even subtler attacks can be mounted by, e.g., unwrapping twice the same key so to obtain two different instances of the very same key. This allows the intruder to set two conflicting attributes on the same key by just setting one attribute to each identical copy. This makes it even more difficult

to circumvent the above attacks since the conflicting attribute policy should extend to all the identical copies of the very same key. The interested reader is referred to [3, 2, 4] for more detail.

3 The *API attacks!* tool

The *API attacks!* tool aims to allow hardware developers and administrators to check the security of their device and configurations against the attacks described in the previous section. We have, first of all, tried to give an easy and intuitive interface. Moreover, the tool has been designed to be as portable as possible. To this aim we have decided to implement it in Java, via the IAIK PKCS#11 wrapper [5], which supports many existing devices.[2]

The attack window Figure 1 shows the attack windows of the tool where, using the buttons on the left, the user can perform six pre-existent attacks, plus custom attacks as described below. When an attack is successful, i.e., a sensitive key is extracted, the tool double-checks that the value of the extracted key coincides with the value of the key stored in the device: it encrypts the plaintext 'PKCS#11' under the stored key and then decrypts it with the extracted one, checking that the obtained plaintext is, again, 'PKCS#11'. The first two buttons on the right can be used to look for conflicting attributes on the keys stored in the device, and for performing a preliminary (static) test on the feasibility of the known attacks on such real keys. The third button, 'Attack test', tries to perform the six above mentioned attacks on the device real keys. The six buttons on the left, in fact, create ad-hoc keys for the specific attacks; while this is interesting for hardware developers, an administrator might only be interested in discovering weaknesses on the actual device configuration and keys and not just potential attacks on the subset of the standard the token implements. Button 'Find attack' produces the input for the two model checkers NuSMV and SATMC, via variations of the perl scripts described in [4, 6]: in order to increase the performance of the analysis, only the actual subset of PKCS#11 implemented on the token is analyzed.

The custom attack window Figure 2 shows the interface for custom attacks. This window allows users to perform all the typical operations performed during attacks: users can create keys, convert keys to byte streams and vice-versa, perform wrap/unwrap and encrypt/decrypt plus extra functions on a different windows, test extracted keys against stored ones. When creating new keys (see the windows on the right), it is possible to try to set their attributes. Since proprietary extensions of the standard restrict the setting of some attributes, the tool reports whether the requested setting has been successfully stored in the device.

Current and future work We are currently testing the tool on a number of commercial devices. So far, we have have found that many devices are indeed vulnerable to some or all of these attacks. We will publish more details after due notice has been given to the manufacturers in question. Meanwhile, the tool is still under development, and is partially supported by the MIUR Italian project SOFT: "*Security Oriented Formal Techniques*. In particular, we are making attack sequences more adaptable to the specific device; an attack may fail because one needed cryptographic primitive is not supported by the device, but a slight variant of the attack might still be possible using a different, supported, functionality. As a trivial example, think of using AES instead of DES. We are studying a graphical interface for custom attacks, so that attack logic can be more intuitively represented in terms of information flows from/to the device. We are extending the model-checking functionality so that the actual configuration, referring

[2] Compatibility has been reported with Giesecke & Devrient, Utimaco, Oberthur, SeTec, Orga, IBM, Safenet, Schlumberger, Gemplus, Dallas, Rainbow, ActivCard, A-Trust, A-Sign, Eracom, Aladdin, Mozilla, Eutron, TeleSec, nCipher.

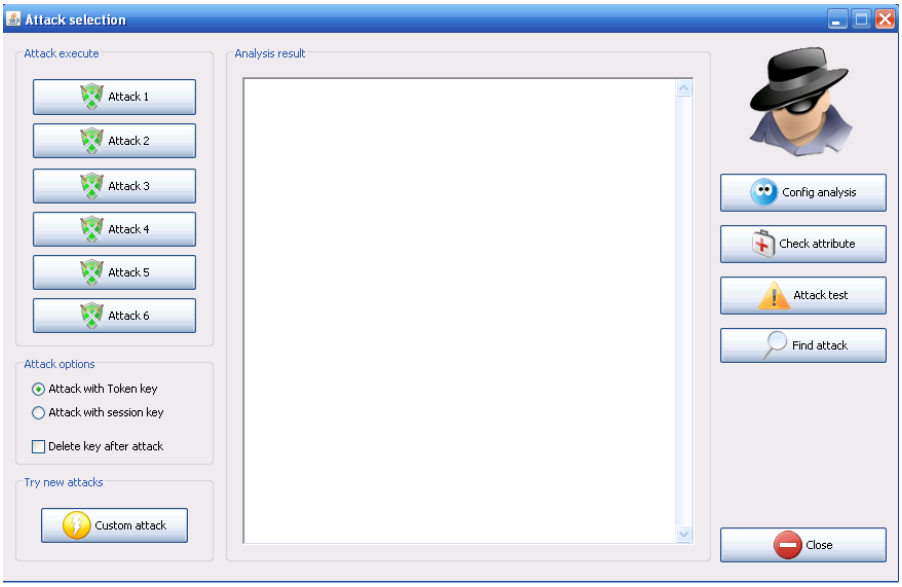

Figure 1: The attack window

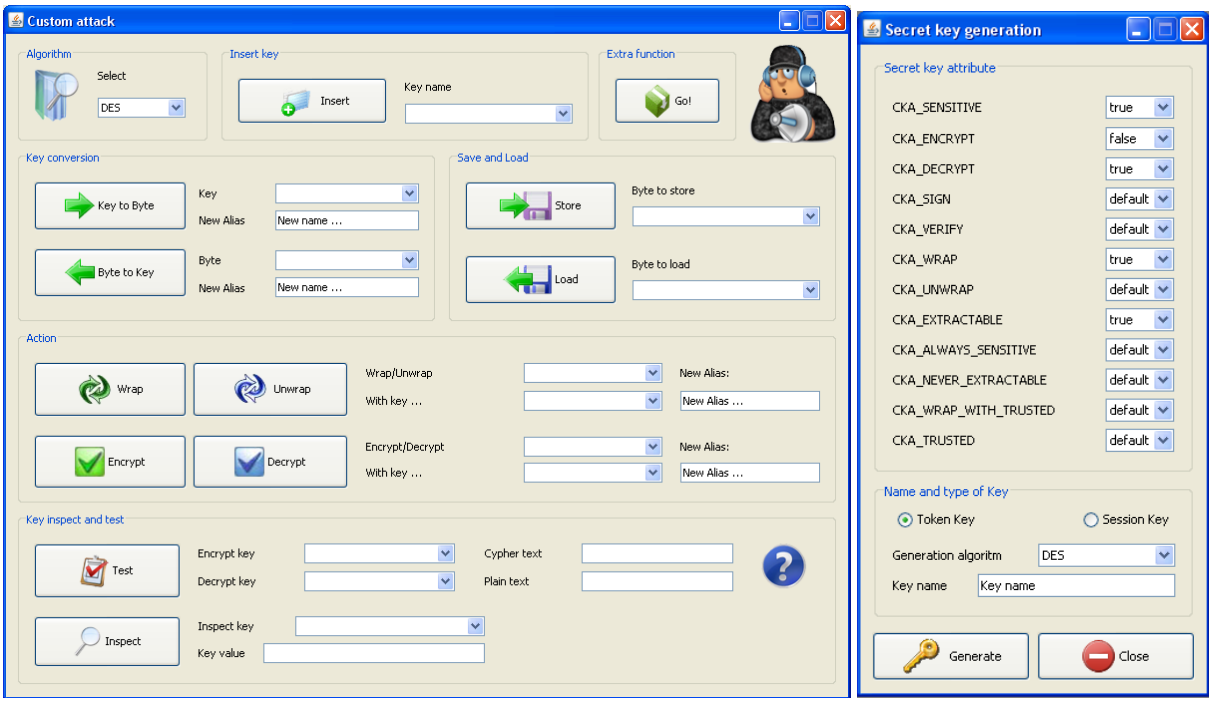

Figure 2: The custom attack window with key generation

to the real keys stored in the device, is included in the model. This might be useful to model-check a specific configuration of the device, instead of assuming the presence of weakly configured keys. We are also writing a parser for the output of model-checkers so that the theoretical attacks can be directly tested on the real devices. Finally, we intend to formalize the static analyses we already perform on the key attributes, to see if they can be used to statically validate specific device and configurations. This might complement the model checking analysis, by providing an additional tool for the static validation of real PKCS#11 devices.

References

[1] M. Bortolozzo and G. Marchetto. Vulnerabilità dello standard PKCS#11: dalla teoria alla pratica. Master's thesis, University of Venice, Italy, 2009.

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

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

[4] 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.

[5] Institute for Applied Information Processing and Communication (IAIK) of the Graz University of Technology. The IAIK Provider for the Java Cryptography Extension (IAIK-JCE) . `http://jce.iaik.tugraz.at/`.

[6] S. 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)*, 2009.