HSM SECURITY

Securing PKCS#11 Interfaces

Whitepaper v1.8

June 2019

cryptosense

Contents

Cryptosense SA, 231 Rue Saint-Honoré, 75001 Paris France

cryptosense.com

1. Introduction

Modern applications that use cryptography usually access that functionality via an application program interface (API) to a software or hardware cryptographic provider. Security-critical applications often make use of Hardware Security Modules (HSMs): special purpose computers that provide high-speed cryptographic services whilst keeping key material inside a tamper-sensitive enclosure. Together with smart cards or similar chip-based tokens, they form the backbone of many modern cryptographic applications in diverse sectors from banking to automotive.

HSMs and smart cards typically undergo a security certification procedure such as FIPS 140 or Common Criteria that examines tamper resistance and susceptibility to side channel attacks amongst other things. However, neither examines the security of the interface of the device. In fact, it would be hard to categorically state that the interface is secure, since each user will typically choose different configuration options that may affect security, and use of the interface by applications must also be taken into account.

The most commonly used API standard is RSA PKCS#11 [4], also known as Cryptoki. While PKCS#11 has important benefits for interoperability since it is so widely supported, it is a large and open standard supporting a wide range of use cases. This makes it rather tricky to implement correctly, and difficult to configure and use in a secure way.

In this white paper, we discuss attacks on systems using the PKCS#11 API. We consider what it means for an interface to be secure, and we discuss how to audit applications' use of the API. There will be plenty of concrete examples of attacks. Finally, we explain the Cryptosense methodology for securing PKCS#11 using our automated analysis tools, which consists of testing a provider's implementation of the spec for compliance and vulnerability, tracing application calls to the interface using the App Tracer, and then combining the results in Cryptosense Analyzer to optimize the configuration. Going forward, our Cryptosense Monitor tool can perform regular checks to make sure devices stay in the right configuration.

2. PKCS#11 Security

In this section we discuss the notion of an attack on a PKCS#11 stack. To detect vulnerabilities, we can look down the stack, i.e. consider the possible breaches that might result from an attacker calling commands in the PKCS#11 interface, and look up the stack, i.e. see what can go wrong if the application misuses the interface. In both cases we give several examples.

2.1. Attack Surfaces in the PKCS#11 Stack

In many application scenarios the command interface of an HSM is kept physically separate from external-facing networks. Even so, there are several scenarios in which an attacker may get access to the interface. One is the case where an attacker obtains legitimate credentials for the internal network, and is hence able to call the HSM API just like a legitimate application. Real-world instances of such attacks include the RBS Worldpay compromise [1] and DigiNotar [2]. In such cases, the fact that session login to the HSM requires a PIN code provides little protection, since the attacker can tamper with the device drivers to bypass the need for a PIN or even retrieve the

PIN from a configuration file. Another scenario is indirect access, where an application accepts some input from an insecure network and as a result calls the HSM to execute some command. This "proxy" may only offer access to a single command, but doesn't require the PIN at all. As we will see, even access to a single command can in some cases give rise to a vulnerability. Smartcard PKCS#11 APIs typically offer less functionality and fewer configuration options than an HSM API. They too are typically protected by a PIN code, but if the card is plugged in to a compromised machine, again the PKCS#11 API becomes an attack surface. This is why it is recognized best practice that cryptography must remain secure even if access controls fail (see e.g. [7]).

If the attacker has gained access to the PKCS#11 interface, we already expect some degree of security compromise. For example, he may be able to create new keys on the device without proper authorisation to do so. However, if the interface is properly configured, it should be impossible for him to obtain the value of keys that have been marked sensitive or unextractable [3, Section 7]. Unfortunately, many PKCS#11 interfaces in their default configuration do not adequately protect these keys, allowing them to be compromised by unanticipated sequences of commands. This can mean a catastrophic breach of security rather than a minor incident.

In the rest of this section we examine some of the vulnerabilities commonly found in PKCS#11 stacks. Refer to the standard [3] for explanations of technical terms. Note that concrete attacks on a PKCS#11 installation will often rely on a combination of several of these vulnerabilities.

2.2. Non-Compliance Vulnerabilities

The PKCS#11 Standard contains a number of important footnotes and implementation details that are critical for the security of key-storage. However, perhaps as a result of the size of the standard, some implementations omit these details, which results in simple single-command attacks on the API. For example, in Section 7 of the standard [3] it is specified that `GetAttribute` should not return the value of a key it is has either the attribute `CKA_SENSITIVE` set to TRUE or the attribute `CKA_EXTRACTABLE` set to FALSE. Unfortunately, real implementations sometimes miss one or other of these protections (see e.g. CVE-2010-3321), leaving an open door for attackers. Another typical mistake is to allow these protection attributes to be turned off, something that the standard forbids (see Section 10.9 and 10.10), but several implementations allow (see [6]).

In addition to key-storage, the security of a cryptographic API relies on the secure implementation of the cryptographic algorithms it provides. However, sometimes these implementations contain faults. Examples include random number generation, which can be affected by glitches or insufficient entropy, and timing attacks on private key operations, for which protection may be turned off by default. Finally, a cryptographic interface must respect good general coding practices such as validating inputs correctly, otherwise vulnerabilities may result that allow standard (non-cryptographic) attacks to compromise the HSM [8].

2.3. Compound PKCS#11 Vulnerabilities

Unfortunately, compliance to the PKCS#11 standard does not imply security. This is because the standard leaves many implementation choices open, and the security consequences depend on

the use of keys by applications. Here we give a sample of the kinds of vulnerabilities that can be found in compliant implementations when combinations of several commands are considered.

2.3.1. Conflicting Key Roles

It is often possible to give keys more than one role via their `CK_ATTRIBUTE` settings, in a way that leads to insecure key management. For a simple example, suppose a key `k1` stored on the device with the `CKA_WRAP` attribute set to TRUE, allowing it to be used by the `C_WrapKey` command to encrypt another key. Suppose it also has the `CKA_DECRYPT` attribute set. In that case a simple attack is possible on any sensitive, extractable key: the attacker simply calls wrap and then decrypt as shown below (we write `{x}y` to denote `x` encrypted by key `y`, and `&k` to denote a handle pointing to a key `k`. We give only the pertinent parameters in the PKCS#11 command calls).

Target of the attack: A symmetric key `k`, present on the device with handle `&k`, and with attributes `CKA_SENSITIVE, CKA_EXTRACTABLE` set.

1. `C_GenerateKey({CKA_WRAP,CKA_DECRYPT})`
   (generates a new key `k1` with WRAP, DECRYPT set)
2. `C_WrapKey(&k1 ,&k)`
   (gives `{k}k1`)
3. `C_Decrypt({k}k1, &k1)`
   (gives `k` as plaintext)

Note that even if key generation commands prevent conflicts, the second role could be given to a copy of the key or could be applied by changing attributes. There are also similar attacks exploiting `C_Encrypt`/`C_UnwrapKey` conflicts.

2.3.2. Insecure Key Export

It can be possible to encrypt a sensitive key with a weaker key by calling the `C_WrapKey` command. Here's a simple example using the `C_CreateKey` command. Target of the attack: Symmetric key `k`, present on the device with handle `&k`, and with attributes `CKA_SENSITIVE, CKA_EXTRACTABLE` set.

1. `C_CreateKey(k1 ,{CKA WRAP})`
   (generates a new handle for known cleartext key `k1` with WRAP set)
2. `C_WrapKey(&k1 ,&k)`
   (gives `{k}k1`)
3. `Offline Decrypt {k}k1 using k1`
   (gives `k` as plaintext)

In other variations, the weaker wrapping key might be non-sensitive (i.e. with the attribute `CKA_SENSITIVE` set to FALSE, and hence readable), or sensitive but already known by the attacker (a so-called Trojan key), or it may be a shorter key than the wrapped key.

2.3.3. Insecure Key Import

It can be possible to obtain the value of an encrypted key presented for import at the PKCS#11 interface by manipulating the `C_UnwrapKey` command. Well-known examples include the padding oracle attacks on RSA PKCS#1v1.5 and CBC_PAD [4].

Other examples include using `C_UnwrapKey` to create a non-sensitive copy of a sensitive key, and private key modification attacks [5, 6].

2.3.4. Insecure Attribute Change

PKCS#11 gives guidance about which `CK_ATTRIBUTES` can be modified and in what sense (i.e. from FALSE to TRUE or from TRUE to FALSE). Unfortunately this guidance is not always followed, and as mentioned in section 2.2, it can be possible to downgrade the security of a key using `C_SetAttributes`. Even if the guidance is followed, an attacker can sometimes upgrade key permissions e.g. to create a conflict by exporting and importing keys.

2.3.5. Insecure Key Derivation

It is possible to abuse the facilities provided by PKCS#11 for key derivation to obtain either the master key or the derived key. For a simple example, suppose we are using `C_DeriveKey` to obtain sensitive keys from a master key by encryption. If the master key also has the attribute `CKA_ENCRYPT` set, the derived keys can easily be obtained using `C_Encrypt`. There are also attacks based on constructing a reduced key space using extracted bits, and parallel key search attacks [5].

2.3.6. Other Attacks

Further examples of attacks can be found in the literature [4, 5, 6].

2.4. Vulnerabilities Caused by Misuse of the PKCS#11 API

A second source of vulnerabilities in a PKCS#11 stack is the application itself. Even if the device interface has been implemented and set up correctly to avoid all the vulnerabilities mentioned above, it is still possible for things to go wrong. Here we give some examples.

2.4.1. Insecure Crypto Mechanisms

PKCS#11 contains a large number of "legacy" algorithms that, thanks to advances in computing power and cryptanalysis, are now considered insecure, like MD5, and single DES. There are also mechanisms that use insecure padding methods like RSA PKCS#1v1.5 (`CKM_RSA_PKCS`) or "raw" unpadded encryption (`CKM_RSA_X_509`). Version 2.40 of PKCS#11, due to emerge in early 2015, will retire some of these mechanisms to a "historical" mechanism section, but this won't completely remove the problem. A new whitepaper from Cryptosense on Algorithm Choice and Key Lengths in PKCS#11 will cover this.

2.4.2. Misuse of Crypto

Sometimes the problem comes from a misunderstanding between the application and the provider over the properties of a certain crypto mechanism. A classic example is the use of the block cipher mode CBC (cipher block chaining), available in PKCS#11 for all block ciphers (`CKM_AES_CBC`, `CKM_3DES_CBC`, etc.). To encrypt, the application must supply an initialization

vector. A classic mistake, still seen widely, is to always supply 0. This results in identical plaintexts giving identical ciphertexts. Worse, plaintexts that are identical just in their first few blocks will have identical prefixes. In fact with CBC, in order to achieve chosen-plaintext security, the IV must be random (even a value just used once is not good enough). Finally, even when used with a random IV, CBC on its own only ensures confidentiality, not integrity. In practice this means that chosen-ciphertext attacks are often possible, in particular padding oracle attacks against modes like `CKM_AES_CBC_PAD` are common in PKCS#11 implementations [4].

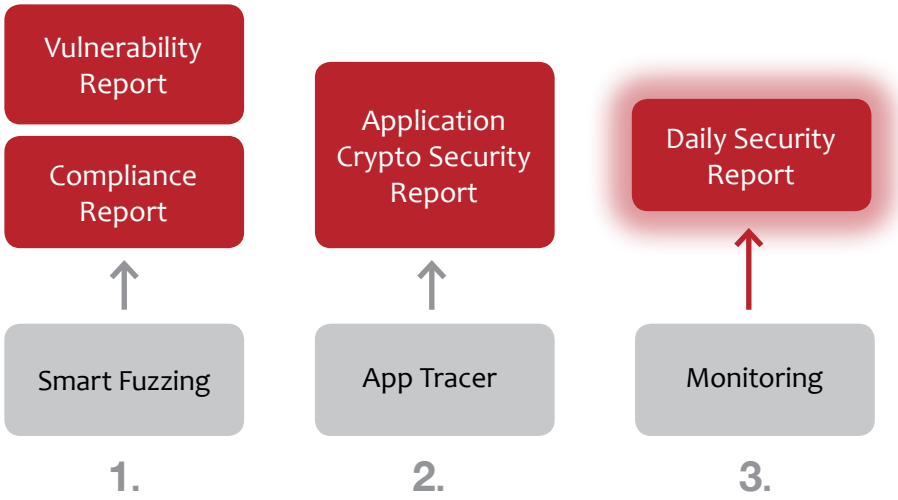
2.4.3. Key Management Failures

As we saw above, finding the right setup for PKCS#11 key management commands in terms of restrictions based on attribute settings is no easy business. However, even once a good configuration has been found, it is still possible for an application error to destroy security. An easy example is if the application fails to generate keys with the correct settings for the `CKA_SENSITIVE` or `CKA_EXTRACTABLE` attributes. Sometimes this is because the application code has been tested in a development environment where the default values assigned to non-specified attributes were different from those in the production environment.

3. The Cryptosense Approach

We now describe how Cryptosense Software can help users of PKCS#11 audit their devices and applications to detect and remove these vulnerabilities.

The Cryptosense approach works in three stages that correspond to the three categories of vulnerability laid out in section 2. We first learn a model of the implementation from testing using our smart fuzzing algorithm. Then, we test the results of the fuzzing, first for compliance bugs, then for vulnerabilities resulting from insecure combinations of operations. In practice, this fuzzing often causes memory corruption in the target HSM, which may be indicative of exploitable buffer overflow type vulnerabilities [8]. The fuzzer contains specific features to detect inconsistent responses from the HSM in order to detect memory corruption.

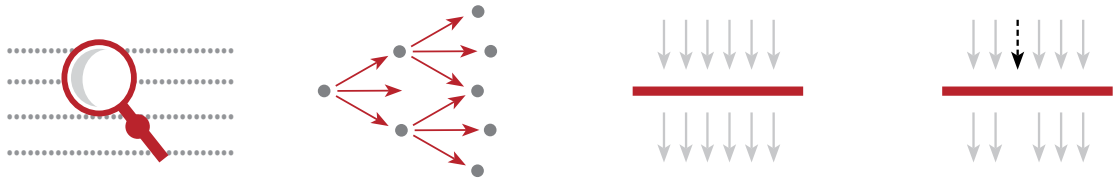


We then trace the application to look for vulnerabilities in the way the interface is used. Finally we can iterate on the configuration options of the HSM to minimise vulnerabilities while still allowing the required functionality. This allows us to proceed to a gold-standard configuration that we can monitor.

3.1. Compliance Testing of the Implementation

Our algorithm for testing implementations employs "well-typed smart fuzzing". The user can trade off testing time against coverage and precision of the fuzzing.

Having learned the behavior of the implementation, we apply a set of more then 120 compliance criteria extracted from the specification to the results. The resulting compliance report details passes and fails for each of the criteria. This includes eliminating the simple vulnerabilities described in section 2.2, and also other behavior that might cause confusion for application developers, such as incorrect default values for attributes.



3.2. Testing Applications with Cryptosense App Tracer

A secure PKCS#11 configuration is only satisfactory if it not only protects keys but also allows applications to get access to the key management and cryptographic functions required. Furthermore, application errors may result in keys being managed in an insecure way, for example if they are not created with the right attributes. Cryptosense App Tracer can trace calls made to the PKCS#11 interface by the application. This trace is processed by Cryptosense Analyzer to reveal cryptographic and key-management vulnerabilities.

Most modern HSMs now include features that allow the configuration of the PKCS#11 interface to be changed. For example, some commands can be disabled or their use restricted to certain patterns of parameters. While these options are welcome and indeed necessary, it is not always clear what the consequences are of each combination of options. Indeed, the range of different options can be baffling. The right configuration will depend on a number of requirements including support for specific applications, internal security policy and legacy issues.

Cryptosense Analyzer helps to find secure configurations in two ways. First, it presents all the attacks found in a simple report. Secondly, after changes have been made, one can simply re-run the testing phase to assess the security of the new configuration.

3.3. Installing Cryptosense HSM Monitor

Having fixed on the "Gold Standard" configuration for a PKCS#11 interface, we can install Cryptosense Monitor onto a machine with access to the live HSM cluster. We set the Monitor to make regular configuration tests on all the HSMs with a frequency chosen by the customer (once per week is a popular choice) and compare the results to the desired configuration. Full reports are produced detailing the testing that can be used for internal and external audit. Additionally, the Monitor can be set to send an alert immediately if an HSM configuration is found to be non-conforming.

4. About Cryptosense

Based in Paris, France, Cryptosense SA creates software to help users of cryptography stay secure. A spin-off of the French institute for computer science research (INRIA) and a collaborator of the University of Venice Ca' Foscari, Cryptosense's founders combine more than 40 years experience in research and industry.

Cryptosense software is used to test the cryptographic systems used to secure over half of daily interbank messages, over half of daily forex trades, and the world's largest financial transaction database. To find out more got to cryptosense.com.

To request a quote for an audit of your PKCS#11 infrastructure, email pkcs11@cryptosense.com.

5. Bibliography

[1] Grand Jury Indictment of Pleschuck et al, http://cryptosense.com/wp-content/uploads/2014/11/rbs-supercediing-indictment-worldpay.pdf

[2] Black Tulip Report, http://cryptosense.com/wp-content/uploads/2014/11/black-tulip-update.pdf

[3] RSA PKCS#11 v2.20, http://cryptosense.com/wp-content/uploads/2014/11/pkcs-11v2-20.pdf

[4] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel and J.-K. Tsay. Efficient Padding Oracle Attacks on Cryptographic Hardware. In CRYPTO'12, LNCS 7417, pages 608-625. Springer, 2012.

[5] Clulow, J.S. "On the Security of PKCS#11", CHES 2003, LNCS 2779, 2003

[6] M. Bortolozzo, M. Centenaro, R. Focardi and G. Steel. Attacking and Fixing PKCS#11 Security Tokens. In CCS'10, pages 260-269. ACM Press, 2010.

[7] OWASP Cryptographic Storage Cheat Sheet, July 2014, https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet#Rule_-_Ensure_that_the_cryptographic_protection_remains_secure_even_if_access_controls_fail

[8] How Ledger Hacked an HSM https://cryptosense.com/blog/how-ledger-hacked-an-hsm/