The worst Hardware-Security-Model vulnerabilities.

And how you can overcome them with these clever steps.

Cognizant Mobility

1 Introduction

HSMs are cryptographic processors with built-in hardware security features and, in some instances, with cryptographic accelerators capable of fast modular exponentiation, elliptic-curve scalar multiplication, and other crypto-specific arithmetic operations. The main purpose of such a device is the secure, tamper-resistant storage and management of cryptographic keys and allowing applications to use these keys without necessitating their extraction. HSMs range from smartcards with basic cryptographic functionalities to full-fledged LAN-based rack units, capable of performing vast numbers of concurrent operations. In common parlance, and throughout the rest of this work, the term HSM refers to the latter.

Traditionally, HSMs were used for banking applications because of various compliance requirements as well as the need for high concurrency. Essentially, HSMs are relevant for all scenarios where storing and using long-term cryptographic keys in a secure manner (i.e. protected against both logical and physical attacks) are requirements. More recently, other areas where HSMs have found application are Public Key Infrastructures (PKIs), e-commerce systems, and DNS Security Extensions (DNSSEC). In keeping with other areas of application development, cloud computing solutions for HSMs have become more ubiquitous and accessible for the consumer. HSMs in the cloud have found several applications, ranging from the traditional (e.g. as a root of trust) to more cloud-oriented approaches, most notably:

- Scaling: collaborative, concurrent utilization of multiple HSMs to handle large amounts of traffic[1].
- Key management service (KMS): provides access to a shared pool of keys to multiple instances of the same or different applications. KMSs often exclusively offer key distribution and/or generation, but no cryptographic functionalities, e.g. Azure KeyVault, AWS KMS.
- Cloud of Secure Elements: aiming to provide trusted computing resources to mobile and cloud applications[2].
- Data deduplication: ensuring that data persisted by cloud providers is not stored multiple times unnecessarily, even if uploaded by different users. Implies nontrivial processing if the data are to be stored in an encrypted manner[3].

[1] J. Han, S. Kim, T. Kim, and D. Han. "Toward scaling hardware security module for emerging cloud services." In: Proceedings of the 4th Workshop on System Software for Trusted Execution. 2019, pp. 1–6.

[2] P. Urien. "Cloud of secure elements perspectives for mobile and cloud applications security." In: 2013 IEEE Conference on Communications and Network Security (CNS). IEEE. 2013, pp. 371–372.

[3] P. Puzio, R. Molva, M. Önen, and S. Loureiro. "ClouDedup: Secure deduplication with encrypted data for cloud storage." In: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science. Vol. 1. IEEE. 2013, pp. 363–370.

As security threats from parties ranging from individuals to state-backed actors grow more common-place, so too do security-first approaches to application development. This leads to hardware-based security solutions such as HSMs finding application in more contexts where such approaches were traditionally thought of as overly complex. As we will see in the next chapter, the complexity of integrating HSMs is, in fact, significant. At the same time, in many situations, discounting this integration due to its difficulty is not a luxury that can be afforded. In this work, we propose several solutions meant to mitigate these effects and enable a more straight-forward and secure integration of HSMs.

2 Common pitfalls of HSM integration

Integrating HSM functionality into an application often proves a complex task. Issues range from vulnerabilities enabled by HSM standard specifications to deviations from the standards introduced by various manufacturers, and the need for specialized knowledge. This can cause the development process to slow significantly, at the same time resulting in software vulnerable to targeted attacks against the underlying HSM specification itself. In this chapter, we discuss four common pitfalls:

Problem	Effect	Solution
API vulnerabilities	Extraction of sensitive data is made possible.	Chapter 3.1: Automatic vulnerability detection and prevention.
Different standard implementations	Vendor lock-in, as migration to new device implies significant code adjustments.	Chapter 3.2: Automatic handling of implementation differences.
High requirement for technical expertise	Slow development, as engineers must focus on cryptographic and integrative rather than business functionality.	Chapter 3.3: Simplified access to cryptographic functionality.
Difficult choice of HSM flavour	Choosing an HSM which does not satisfy one's requirements can lead to unsatisfactory performance or unnecessary administration overhead.	Chapter 3.4: Comparison of various devices' strengths and weaknesses.

2.1 Vulnerabilities to API-level exploits

The PKCS #11 specification[4], nicknamed `Cryptoki`, which handles the communication with hardware cryptographic modules, has stood the test of time, having first been proposed in 1994, and has since become the de-facto standard for integrating HSMs with software. While most manufacturers also offer libraries to access the device via various other interfaces (such as JCA/JCE or Microsoft CAPI), these libraries are often mere abstraction layers, mapping cryptographic requests to relevant PKCS #11 calls. In short, for most devices, there is no avoiding the Cryptoki API.

Despite its popularity and staying power, the PKCS #11 standard is not without its faults, nor are many of its implementations. Vulnerabilities fall into one of three categories:

1. API design: faults in the API specification that allow the extraction of sensitive information in specific circumstances

2. Non-compliant implementations: the standard is not followed, opening a specific device to various attack vectors.

3. Usage errors: the cryptographic mechanisms are used incorrectly, which leads to a weakening of the underlying algorithms.

Various studies analysing the standard's security as well as that of many popular HSMs have been conducted[5], showing that many well-known vulnerabilities can be exploited even in modern devices. In this work, we focus on nine vulnerabilities:

[4] http://docs.oasis-open.org/pkcs11/pkcs11-curr/v2.40/os/pkcs11-curr-v2.40-os.html

[5] e.g.: J. Clulow. "On the security of PKCS# 11.", S. Delaune, S. Kremer, and G. Steel. "Formal analysis of PKCS# 11.", M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. "Attacking and fixing PKCS# 11 security tokens."

ID	Type	Description
W1	API design	Key extraction by inadequate key separation: allowing keys to be exported in plaintext by wrapping[6] them with keys which can also be used to decrypt data.
W2	API design	Trojan wrapped key: allow the extraction of keys by wrapping them using a trojan key created by unwrapping and granting it permissions to wrap other keys.
W3	API design	Trojan public key: allow the extraction of keys by wrapping them using an imported public key, for which an attacker holds the corresponding private key.
W4	API design	Weaker key wrapping: wrap a secure key using an insecure one (i.e. wrapping an AES key with a DES key), thus reducing its bit-security.
W5	Non-compliant impl.	Neglection of security attributes: device ignores attributes marking keys as sensitive, allowing them to be exposed in plaintext.
W6	Non-compliant impl.	Unauthorised attribute change: device allows modification of immutable attributes, possibly exposing sensitive information.
W7	Non-compliant impl.	Incorrect security attribute carryover: attributes are not set correctly when copying or deriving a key.
W8	Usage errors	Outdated cryptographic mechanisms: using algorithms which can be broken in feasible time.
W9	Usage errors	Insecure mechanism usage: incorrect initialisation of cryptographic mechanism, by e.g. not setting an initialisation vector for the AES-CBC encryption scheme, enabling attacks capable of gaining access to the plaintext.

Whether maliciously or accidentally, performing operations that exploit these vulnerabilities can lead to exposing cryptographic keys which should otherwise remain securely within the protection capabilities of the HSM, or to decoding ciphertexts without having access to the corresponding keys, thus severely undermining the security expected of HSMs.

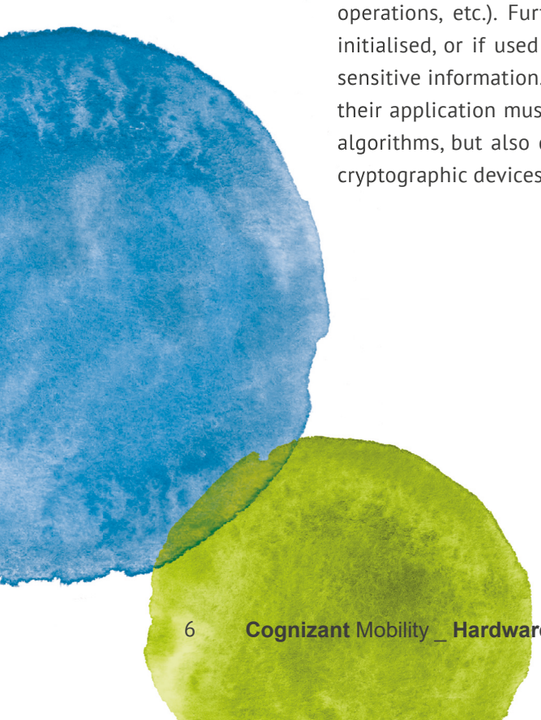
[6] Wrapping is the process of encrypting a key using another key. Unwrapping is decrypting a ciphertext corresponding to a cryptographic key and using the result as a key.

2.2 Differences in standard implementations across manufacturers

While the Cryptoki specification aims to standardise the communication with cryptographic devices, it also opens the possibility for manufacturers to introduce custom mechanism definitions, called vendor-defined mechanisms, even for underlying cryptographic algorithms already covered by standard ones. Furthermore, other variables in the usage of HSMs (such as maximum buffer sizes) vary from one manufacturer to another. Finally, the error handling mechanisms also vary across vendors, resulting in differing error codes for the same issues. These differences, unless explicitly handled, slow down development even for experienced engineers, since each manufacturer ships devices with own idiosyncrasies which must be examined individually before proceeding with the implementation. They also result in codebases which are virtually vendor locked-in and cannot be easily migrated to other devices.

2.3 Requirement of specialised expertise

Finally, using the cryptographic functionality offered by HSMs requires extensive knowledge about the used API, the types and acceptable values for parameters required for mechanism initialisation as well as various other workflows defined by the standard (such as authentication, customization of parallelism, multi-part operations, etc.). Furthermore, many cryptographic mechanisms, if improperly initialised, or if used under unfavourable conditions, may cause the leaking of sensitive information. Thus, a software developer looking to integrate an HSM in their application must not only possess expertise with regards to cryptographic algorithms, but also detailed knowledge of the mode of operation of the used cryptographic devices.



2.4 Deciding between various devices based on requirements

While there is a multitude of HSM devices and equivalent cloud services on the market, their precise technical capabilities (such as maximum throughput for various cryptographic algorithms or storage capacity) are often non-transparent. Even a question as fundamental as the choice between a cloud service and an on-premise solution is plagued by unknowns. Most relevant information, such as the complexity of administrative responsibilities, only comes to light during the initial setup, while aspects like security concerns, such as third-party access to stored keys or vulnerability to API-level weaknesses, remain hidden even beyond this phase. Like with most choices surrounding computational infrastructure, there is not a "catch-all" solution, ideal regardless of requirements. Unlike many such choices, there are few data points regarding HSMs publicly available, making a choice that optimally suits one's requirements difficult.

3 A more secure, versatile and easy-to-use approach

Since much of the complexity surrounding the integration of HSMs is independent of higher-level use-cases, we separated this integration into its own library which aims to avoid pitfalls commonly found in these scenarios, from the prevention of vulnerabilities to facilitating the use of cryptographic mechanisms via developer-friendly interfaces. Presently, four devices are supported: nCipher Connect XC Base, AWS CloudHSM, Azure Dedicated HSM (using a Gemalto SafeNet Luna 7 model A790[7]) and the Ultimaco CryptoServer.

[7] https://azure.microsoft.com/en-us/services/azure-dedicated-hsm/

3.1 Improving upon the standard – preventing vulnerabilities

While the Cryptoki standard is structured in a way meant to avoid leaking sensitive information (such as private keys), it still leaves open many avenues of attack. Many of these are handled by the security community through principles of best practice (such as the principle of key separation, i.e. not using the same key for different purposes) and are thus left at the developer's discretion. Other vulnerabilities relying on inexact implementations of the standard or on usage errors can also be exploited to extract sensitive material.

This led us to the decision to implement a mechanism capable of automatically detecting and preventing vulnerabilities. While the Cryptoki API itself is stateful, most of the exploits aiming to extract sensitive information rely on individual calls rather than the state of the device, making their detection relatively straightforward and inexpensive. This type of detection implies comparing calls to HSMs to pre-defined patterns and raising alarms in case any matches are found. In select cases, more complex processing operations are required, such as finding a corresponding private key and examining its attributes when working with a public key. In order to strike a balance between security, development flexibility, and performance, four security levels are defined, which ensure varying degrees of protection:

- Level 0: No protection.
- Level 1: Enforcement of the Cryptoki standard, i.e. preventing vulnerabilities caused by non-compliant implementations. Contains preventions for W5, W6, W7.
- Level 2: Enforcement of best practices. In addition, contains preventions for W1, W8, W9.
- Level 3: Maximum protection, which further restricts some operations, such as granting certain privileges to imported keys. In addition, contains preventions for W2, W3, W4.

The various preventions are particularly meant to enforce software quality standards, and thus prevent the accidental execution of cryptographic requests that would result in the disclosure of information intended to remain within the physical confines of the HSM. This mechanism would also prevent the execution of exploits using only interfaces offered by the application (e.g. REST APIs), it would not, however, offer additional protection against attackers with the means of using the PKCS #11 API to the HSM directly. For this purpose, the protections that are part of this library would have to be implemented directly onto the devices themselves (or at least in the dynamic library used to transport PKCS #11 requests to the device). Some manufacturers (e.g. nCipher or Thales) do, in fact, ship their devices with various similar preventions, which may be a deciding factor when choosing a device.

Finally, we also implemented a tool that automatically detects if a device is vulnerable by executing the corresponding exploits and verifying that the extracted information is correct. These results should be considered before an HSM choice is made.

3.2 Consolidating divergent standard implementations under a single API

As described previously, there are several differences in the ways various manufacturers implement the PKCS #11 standard. One example is the generation of keys to be used when creating Message Authentication Codes, where nCipher devices require the use of vendor-defined mechanisms rather than the ones set by the standard. Other devices, such as the ones used by AWS CloudHSM set a maximum buffer size of 16KB, necessitating multiple operations to encrypt, hash or sign data blocks larger than this limit. Error handling for this HSM also deviates from the norm, some error codes being returned in circumstances not covered by the standard. Many other similar deviations can be found in the devices covered by this work, and we expect the list to grow with the addition of each new HSM.

If left unhandled, these types of differences would cause significant complications when trying to migrate an application from one device to another. The basic underlying functionality can be performed by each device, albeit with slightly differently structured calls, so if these differences were handled explicitly and individually by a layer of abstraction functioning between the application's logic and the device, migrating to a different HSM would have no effect on the codebase. Our library offers a vendor-agnostic API, handling the translation of high-level cryptographic requests to vendor-specific implementations of the PKCS #11 standard, and thus allowing developers to focus on functionality rather than hardware integration.

3.3 Developer-friendly interface to HSM functionality

In addition to the intricacies of the PKCS #11 standard and the complications introduced by its divergent implementations, most plain cryptographic APIs, like the one offered by JCA, require the user to have complex knowledge of the underlying algorithms in question. Possible values for the various mechanism parameters (mode, padding, key size, used hash or mask generation functions, definitions of elliptic curves, etc.) must be known beforehand. Our library's API takes advantage of IDEs' code completion features in the initialization of mechanisms, while also simplifying high-level cryptographic requests (like encryption or decryption) to only necessitate a single call. These differences are illustrated in the following:

AES-CBC encryption with JCA/JCE

```
//preamble
byte[] payload = Util.randomBytes(100000);

//key generation
var generator = KeyGenerator.getInstance("AES");
generator.init(256);
var key = generator.generateKey();

//mechanism initialization
var cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
IvParameterSpec iv = new IvParameterSpec(Util.randomBytes(16));
cipher.init(Cipher.ENCRYPT_MODE, key, iv);

//encryption
var cipherText = new byte[cipher.getOutputSize(payload.length)];
int encryptedLength = cipher.update(payload, 0, payload.length,
cipherText, 0);
encryptedLength += cipher.doFinal(cipherText, encryptedLength);

//decryption
cipher.init(Cipher.DECRYPT_MODE, key, iv);
byte[] plainText = new byte[cipher.getOutputSize(encryptedLength)];
int decryptedLength = cipher.update(cipherText, 0, encryptedLength,
plainText, 0);
cipher.doFinal(plainText, decryptedLength);
```

AES-CBC encryption with this work's library

```
//preamble
var facade = new CryptoFacade();
var payload = Util.randomBytes(100000);

//key generation
var template = KeyInventory.SymmetricKey.AES();
template.setLength(AesKeyDTO.Length.LENGTH_256);
var aesKey = facade.generateKey(template);

//mechanism initialization
var mechanism = MechanismInventory.SymmetricEncryption.AES_CBC();
mechanism.setIV(Util.randomBytes(16));
mechanism.setPadding(AesCbcMechanismDTO.Padding.PKCS5);

//encryption
var cipherText = facade.encrypt(payload, aesKey, mechanism);

//decryption
var plainText = facade.decrypt(cipherText, aesKey, mechanism);
```

The total amount of code necessary is reduced slightly, but our library's API's real advantage is given by the fact that there is a lower pre-requirement of knowledge to implement the same functionality. In the background, the relevant calls are translated either to PKCS #11 heading to the integrated HSM, or to JCA calls executed by the Bouncy Castle provider, depending on configuration. Thus, simplified testing of business logic is made possible, one which does not require connection to a physical device. In tandem, these features streamline development and testing.

3.4 Considering the pros and cons of various devices

As described in Chapter 2.4, deciding between different approaches to HSM infrastructure based on requirements is non-trivial and must often rely on scant information. For this purpose, we conducted an analysis of the four devices supported by our library in terms of various criteria:

- Performance with regards to their supported algorithms.
- Security (third party access, authentication mechanisms, vulnerability to exploits, additional protection mechanisms, etc.).
- Scalability and resilience.
- Storage capacity.
- Supported algorithms and APIs.
- Maintenance (backups, updates, monitoring, migration of keys, etc.).
- Partitioning (dividing keys into partitions to be used by different applications).

Depending on the exact requirements, some criteria may be more relevant than others, so the superiority of one device over another may change with requirements. Some of these comparisons

4 Conclusion

As we have seen throughout this paper, HSMs, while oftentimes indispensable, are also frequently very complex systems. From the choice of vendor, to the initial setup and regular maintenance activities, to the integration and testing of security applications, these devices require significant, wide-ranging expertise. Our knowledge in this field and approach to HSM management and usage result in faster, more efficient and more secure development processes.