POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Protection of Private Keys with TPM 2.0

Supervisor
prof. Antonio Lioy

Candidate
Damiano Turriziani

Academic Year 2022-2023

To my family, whose love and support made this journey possible.

Summary

In today's digital era, web servers play a critical role in ensuring data security, user trust, and compliance with regulations. They are the gatekeepers of web content, enabling encrypted communication, data integrity, and reliable content delivery. Among the leading web servers, Nginx, Apache, and Cloudflare Server top the list. Web servers are essential in the face of increasing cyber threats, providing safeguards through access control, availability, and data confidentiality and integrity. Access control is facilitated by authentication and authorisation, while availability is achieved through mechanisms like fault tolerance and scaling. Data confidentiality is upheld via encryption and Transport Layer Security for secure connections, while data integrity is preserved with the help of public key infrastructure (PKI) and digital certificates. Implementing TLS with OpenSSL in Nginx and Apache is crucial for secure, encrypted communication. This security layer ensures data protection, user trust, and regulatory compliance. It enhances website credibility and overall security, defending against cyber threats and future-proofing web services. Safeguarding cryptographic keys, a complex challenge due to the need for data decryption, can be addressed by using secure storage mechanisms, such as Hardware Security Modules (HSMs), Trusted Platform Modules (TPMs), Key vaults, or Trusted Key Management Systems (KMS). TPMs are tamper-resistant cryptoprocessors adhering to international standards, designed to enhance hardware security. They store aggregated measurements in Platform Configuration Registers (PCRs) and offer cryptographic records of the software state. Sealing/Unsealing operations ensure secure key storage and retrieval, and the solution based on Sealing objects enhances private key security. The proposed solution employs the Sealing operation to protect private keys in an Apache server, ensuring they can only be retrieved using the TPM while adhering to the conditions imposed by the PCRs. The solution, developed through source code modifications, offers a strong balance between security and resource efficiency, achieving a robust defence against cyber threats like brute force and dictionary attacks. The ability to use TPM 2.0 for private key protection can also be extended to web servers like Nginx, making this approach highly versatile. In conclusion, the presented solution leverages TPM 2.0 to enhance the security of private keys in Apache servers, using PCRs for system attestation. This solution, adaptable for Nginx and any OpenSSL-based web server, offers a robust security mechanism for safeguarding critical cryptographic assets.

Acknowledgements

This thesis is the result of many contributions, not just from me. First and foremost, I would like to thank my parents who gave me the opportunity many years ago to begin this journey, which is now concluding in the best way possible. They supported me during difficult times and never lost faith in me. I would like to express my gratitude to my family, who, despite the geographical distance, have always shown their love and encouraged me to continue and not give up.

I want to thank my lifelong friends, without whom these years would have been impossible. It is only thanks to you that I am who I am today, thanks to those moments where the only concern was to have fun and be together. I also want to express my gratitude for the new friendships formed during this long journey. With you I have suffered and laughed and that helped us to overcome all those difficulties that seemed insurmountable.

Finally, I would like to thank Prof. Lioy for giving me the opportunity to carry out this thesis, allowing me to enrich my professional knowledge. A special thanks goes to the doctoral students and fellow thesis students in Lab 7 who assisted me in completing this challenging work.

Contents

Chapter 1

Introduction

In today's digital landscape, **Web Servers** have assumed crucial importance. They serve as the keystone of online operations, ensuring data security, user trust, and compliance with regulatory standards. Web servers are the gatekeepers of web content, facilitating encrypted communication, data integrity, and reliable delivery. Their role extends beyond mere content distribution; they are the bedrock for dynamic applications, high-traffic scalability, and redundancy through load balancing. According to W3Techs' data for September 2023, the three leading web servers are **Nginx**, **Apache**, and Cloudflare Server. Also, Netcraft's statistics for August 2023, corroborate this, ranking Nginx, Apache, and OpenResty as the most widely employed web servers.

In an era of heightened **Cyber threats** and user expectations, web servers are indispensable for safeguarding sensitive information, maintaining website credibility, and securing the seamless flow of online data. Cyber threats can be mitigated by ensuring: access control, which involves authentication and authorisation; availability is vital for uninterrupted service and it is achieved through mechanisms like fault tolerance, recovery, high availability, and server scaling; and last but not least in importance confidentiality and integrity, which are crucial for data protection. Confidentiality restricts unauthorised access with mechanisms like access controls, encryption and **Transport Layer Security** (TLS) for secure connections. While integrity ensures data accuracy and reliability requiring security mechanisms like Public Key Infrastructure (PKI) and digital certificates.

Enabling TLS through **OpenSSL** for both Nginx and Apache is essential to establish secure, encrypted connections between servers and clients. This security layer is fundamental for data protection, user trust, and regulatory compliance. By implementing TLS, both web servers can provide secure communication, safeguarding sensitive data and ensuring the integrity of information transferred. Additionally, TLS enhances the credibility of websites and applications. Beyond these aspects, TLS support in Nginx and Apache is pivotal for defence against cyber threats, future-proofing, and overall web security in an increasingly interconnected digital landscape.

When using TLS with OpenSSL, it is critical to store the private key securely and protect it from being cracked or accessed by unauthorised individuals. Safeguarding cryptographic keys is a complex challenge, mainly because applications

require a certain level of access to these keys for data decryption. While it may be challenging to completely shield keys from attackers who have fully compromised the application, there are steps that can be taken to increase the difficulty of their attempts to obtain the keys. The first and fastest way is to protect the private key with a strong password. Unfortunately, this is still subject to brute force or dictionary attacks. On this subject, the NIST states that whenever possible, it is advisable to utilise one of the following secure storage mechanism: a physical or virtual Hardware Security Module (HSM), a **Trusted Platform Module** (TPM), a Key vault or a Trusted Key Management System (KMS).

The **Trusted Platform Module** (TPM) adheres to an international standard and serves as a secure cryptoprocessor with tamper-resistant features. It is specifically designed to enhance the security of hardware by implementing fundamental cryptographic functions, which serve as the foundation for more complex security features. One of its primary functions is to store aggregated measurements, which play a significant role in the process of the **Sealing/Unsealing** operation. These measurements are stored in specialised registers known as **Platform Configuration Registers** (PCRs). These registers primarily serve as cryptographic records of the software state. This is possible due to their update mechanism that ensures that the registers cannot be tampered with or forged. PCRs can be accessed to provide information about the machine's state and can also be used to create **Sealed Data Blobs** which can only be accessed if the value of the current PCRs values match those at the time of their creation.

During the thesis work, different solutions were found and studied to solve the issue related to the security of private keys. In this context, various functions offered by the TPM have been analysed, and among them, two potential candidates have been identified: **Binding** and **Sealing**. Initially, the Binding function appeared to be the simpler one to use and implement, but unfortunately, it did not provide any control over the platform's state. For this reason, the Sealing operation has been chosen as our solution to implement a secure storage mechanism for the server's private key.

The solution sought should in no way change the behaviour of the Web Server used but instead should enhance security without the server having any knowledge of the integrated functions. Additionally, the solution should be as simple as possible to enable any user to integrate it into their server solutions. It should also be as portable as possible, meaning that it could run on any machine with hardware specs similar to those of the machine used for testing.

The thesis is divided into two main sections. The first part delves into the concept of Trusted Computing and introduces the TPM 2.0, providing insights into its architecture, functionalities, and how it differs from earlier versions. This section also includes an introduction to the **TSS**, followed by a specific presentation of the **ESAPI** module, which is crucial for developing the proposed solution.

Moving on to the second part, it explores and implements two solutions, offering a broad overview of the functions used and the resulting operations. Following this, there is a detailed examination of the testing phase carried out on the final solution.

Chapter 2

Trusted Computing and TPM 2.0

2.1 Trusted Computing

Trusted computing is a general term used to refer to computing systems that use software and hardware that aims to make personal computers and computing devices more secure through the creation of systems with predictable behaviours [1]. Predictable systems are obtained using *Trusted components* that are defined by the *Trusted Computing Group (TCG)* as predictable components. The term predictable is used consciously in this context for two main reasons: the first one is that something that is predictable is simple to evaluate in fact if I can predict a component's behaviour given some stimuli as input, I can evaluate its performances; the second reason is that if something is predictable this does not change, whatever the situation. The technologies which fall under the trusted computing term are [1]:

- *Trusted Platform Module (TPM)* is a chip, usually attached to the motherboard of commercial PCs and servers, which provides cryptographic functions.
- *Self-Encrypting Driver* is a fast hardware-based solution for data protection, built into a hard drive.
- *Secure CPU Mode* is provided by both Intel and AMD and offers software measurement, secure execution and code signature checking.
- *Trusted Network-Connect* is a collection of networking protocols that integrate platform-level trusted computing into network access decision
- *Multilevel Computing* is the combination of both hardware and software to create a trustworthy system capable of securely handling information at multiple levels.

The principle of Trusted Computing was first stated in the *Trusted Computer System Evaluation Criteria (TCSEC)* also known as Orange Book, published by the Department of Defence (DoD) in 1985. This document asserts that the core of trusted computer systems is the *Trusted Computing Base (TCB)* which includes

hardware, firmware and software containing all the elements of the system needed to support the security policy and the isolation of objects which establish the base protection [2]. This collection of system resources must be designed and implemented to be as simple as possible and consistent with the functionalities it has to perform [3]. In addition, an important characteristic of the TCB is that it can not be compromised by any hardware or software that is not inside the *security perimeter*, defined as the set of all the elements of the TCB [4]. Unfortunately, the concept defined by the Orange Book was proposed in the era of mainframe systems, so it did not take hold.

In the 1990s, the computer industry understood that there was a need for increased security in Personal Computers (PCs). This was mainly caused by the widespread of PCs, which were first designed without any attention to security [5]. The possibility to connect PCs through the Internet increased the concern of IT vendors, which tried to use software solutions to increase security but soon it became clear that software-only security mechanisms are not sufficient. For this reason, a new approach was introduced for protecting systems from compromise, the use of a hardware-based embedded security solution [6].

This approach was first suggested by the *Trusted Computing Platform Alliance (TCPA)* in 1999. The TCPA started releasing specifications for the *Trusted Platform Module* until February 2002 when TPM version 1.1b was published. In 2003, TCPA became the *Trusted Computing Group (TCG)*.

2.1.1 TCG

The TCG inherited the starting goal of the TCPA but they extended it to mobile devices, servers, peripherals, the network and any device which is connected to the network. The TCG's main concern is about the security of communication, transactions and exposure of data on systems. The purpose of TCG is to define and develop specifications for trusted computing that are open and vendor-neutral. The specifications provide a guideline for hardware building blocks and software interfaces for different platforms and operating environments. In addition, the TCG specifications provide a higher trusted relationship for machine authentication, attestation and user authentication. The specifications currently created are for the Trusted Platform Module and for the Trusted Computing Software Stack, which is used as a foundation for writing applications that interact with the TPM. The benefits promoted by the TCG are:

- Personally identifiable information and digital secrets and more secure storage of files. In this way data and identity are protected against external software attacks or physical theft.
- Enhance the security of remote access by protecting the keys used in the authentication process such as VPNs, S-MIME e-mail and 802.1x.
- Multi-factor user authentication achieved with low-cost and strong trusted computing components that can be integrated with other types of authentication such as smart cards, finger-print readers, pass phrases, etc.

- Protection of networks through network access control and endpoint integrity using TCG specifications.

2.2 Trusted Platform

Trusted platforms are designed to provide a secure foundation for the operation of critical systems and the protection of sensitive data. This is achieved through a combination of hardware and software security measures [1], such as:

- *Secure Boot*;
- *Hardware-assisted Memory Isolation*;
- *Trusted Platform Modules (TPMs).*

Secure boot is a process that verifies the authenticity of the system's boot-loader and any subsequent software that is loaded during the boot process. This helps to ensure that the system is running only trusted software, and not malware or other unauthorised software.

Hardware-assisted memory isolation is a security measure that uses the hardware features of the processor to protect the memory of the system from unauthorised access. This can help to prevent malware or other threats from accessing sensitive data stored in memory.

Trusted platform modules (TPMs) are specialised chips that can be used to store keys, passwords, and other sensitive data in a secure manner. TPMs use cryptographic techniques to protect the data stored on them, and they can be used to verify the integrity of the system and its components.

An important aspect of trusted platforms is that they should be predictable, this means that it is necessary to determine their identity. The TCG specification establishes that trust in a platform is based on identifying its software and hardware components [4]. The TPM is the component chosen by the TCG to provide methods for collecting and reporting on the hardware and software.

The starting point of the TCG specification to build a trusted platform is the *Trusted Building Block (TBB)* which is a component or a collection of components that are required to instantiate a Root of Trust. Typically, the TBB is a component of the Root of Trust and it is expected to not compromise the system. Another important concept is the TCB, which is the collection of system resources that maintain the security policy of the system. The TPM is not a TCB component, indeed it allows an independent entity to determine if the TCB has been compromised and in this case, the TPM may help to prevent the system from starting.

2.2.1 Roots of Trust

The Root of Trust (RoT) is a set of components on which the trustworthiness of the system is based. An important characteristic of RoT is that it is unverifiable

because if I use a second component to verify it, the second component became the root of trust [1]. For this reason, trust in the RoT should be based on out-of-band assumptions. This is possible using certificates that assure that the RoT has been implemented in a trustworthy way. For example, a certificate can provide the Evaluated Assurance Level (EAL) and the identity of the manufacturer. In addition, a certificate may provide information about the state of the installation of the TPM and if the machine is compliant with the TCG specifications. In PC clients, according to the TCG specification, are required three RoT that are:

- *Root of trust for Measurement (RTM)*;
- *Root of trust for Storage (RTS)*;
- *Root of Trust for Reporting (RTR).*

Instead, in other non-PC scenarios such as phones and cars, the trusted computing system may contain different RoTs, for example:

- The *Root of Trust for Verification* consists in verifying an integrity measurement against a policy and it is used in some embedded devices in which the manufacturer defines some approved software;
- The *Root of Trust for Update* is mostly used for firmware updates in which the update is verified by checking an authorised signature.

Root of Trust for Measurement (RTM)

The RTM provides integrity measurements and it records them in the RTS using the *Extend operations*. An Integrity measurement is any value that represents a possible change in the trust state of the system. The measured objects are often data values, hashes of data or code, and an indication of the signer of some data or code. The digest computed on an integrity measurement is statistically unique. The RTM is outside the TPM scope, and it corresponds to the *Core Root of Trust for Measurements (CRTM)*. In case the RTM code is stored inside the BIOS, it is possible to implement the CRTM inside it. The first solution is to integrate the CRTM as a boot block and the second one is to make the BIOS a TBB.

The CRTM makes the initial measurements of the platform and then stores the result inside the PCRs. For measurements to be meaningful, the platform should be in a trust state, and this can be done in two ways[4][1].

- The first method that can be used is a power-on reset which brings the platform to a known initial state. Starting from these measurements, a chain of trust is created only once and no changes to the initial trust state are possible, for this reason, is called a *Static RTM (SRTM)*. Specifically, SRTM refers to the boot block and which parts of the BIOS are measured depends on the implementation. It may happen, that critical pieces of code are not included and in some systems, this makes SRTM measurements extremely unpredictable. One of the biggest downsides is the fact that SRTM is typically implemented by BIOS vendors. Instead, SRTM's advantages consist of usability.

- The second method requires the CPU to act as the CRTM and apply protections to portions of memory it measures. This creates a chain of trust without rebooting the platform and it allows the RTM to be re-established dynamically, for this reason is called *Dynamic RTM (DRTM)*. The DRTM avoids trust in the BIOS vendors as for the SRTM and it is very different from it. Indeed, it does not launch automatically, but a special trusted CPU mode is initiated and a region of memory is passed to be executed. The CPU secure mode pauses all running processes and enters single-tread mode. After the DRTM code is checked and executed. The DRTM code hashes the region of memory and stores it in the TPM. Thanks to the fact that the memory region is chosen by the user this provides extreme flexibility, predictability, and greater precision.

In the 2.1 is possible to see a possible scenario of SRTM in comparison with 2.2 which shows a DRTM's possible usage.

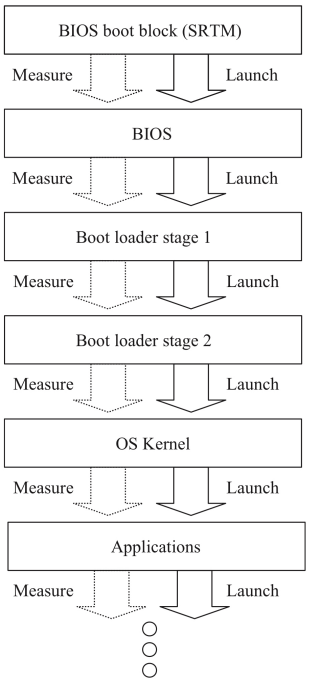


Figure 2.1. SRTM boot chain rooted in the BIOS boot block [1]

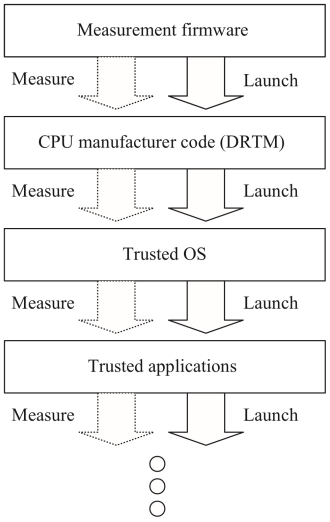


Figure 2.2. Trusted OS launch sequence using DRTM [1]

The *Chain of Trust* allows us to bootstrap from the low-level root of trust to a higher-level, by using the RoT to trust in other objects, and use the trust obtained to trust in other objects above them. In this way using the concept of Transitive Trust, defined by the TCG, the trustworthiness of the platform is incremented. This is shown in the 2.3

Root of Trust for Storage (RTS)

The RTS is a component responsible for maintaining both the secrecy and the integrity of secrets. Because the TPM memory is shielded from access by any

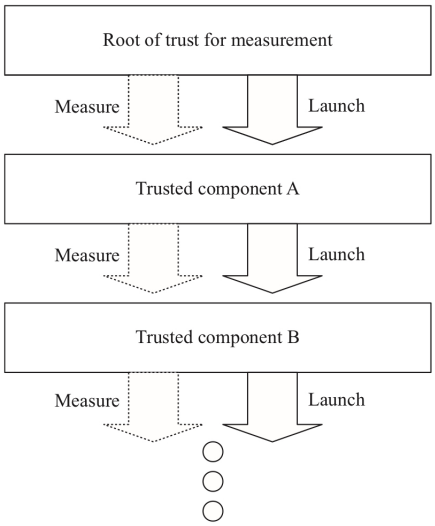


Figure 2.3. Chain of trust [1]

external entity, the TPM can act as an RTS. Typically the content of a *Shielded Location* is needed to get access to another Shielded Location and in some other cases, the content of a Shielded Location is conditioned on PCRs having specific values. TPM memory can contain :

- *non-sensitive data* which is not protected from disclosure for example the content of the PCRs;
- *sensitive data* which is protected and the TPM does not allow access to it without proper authorisation for example the private part of an asymmetric key stored in a Shielded Location.

Root of Trust for Reporting (RTR)

The RTR is a component that provides accurate reporting on the contents of the RTS. The RTR can be implemented by the TPM because as in the case of RTS, it has the cryptographic capabilities to create an RTR report. The RTR report is a digitally signed digest on some selected values within the TPM. The typically selected values are:

- evidence of the platform status using the content of PCRs;
- audit logs;
- key properties.

Since the interaction between the RTS and RTR is critical, the design and the implementation of both should mitigate tampering to prevent errors in reporting. In addition, a Platform Certificate is needed to assure the binding between the platform (the RTM) and the RTR. This is essential to assure that the PCR values, which are quoted during the creation of the RTR report, reflect the state of the platform.

2.2.2 Attestation and Authentication

In general, attestation hierarchies are used to provide a way to establish trust in a system and to ensure that only trusted entities can access certain resources or perform certain actions. This is shown in the 2.4[4].

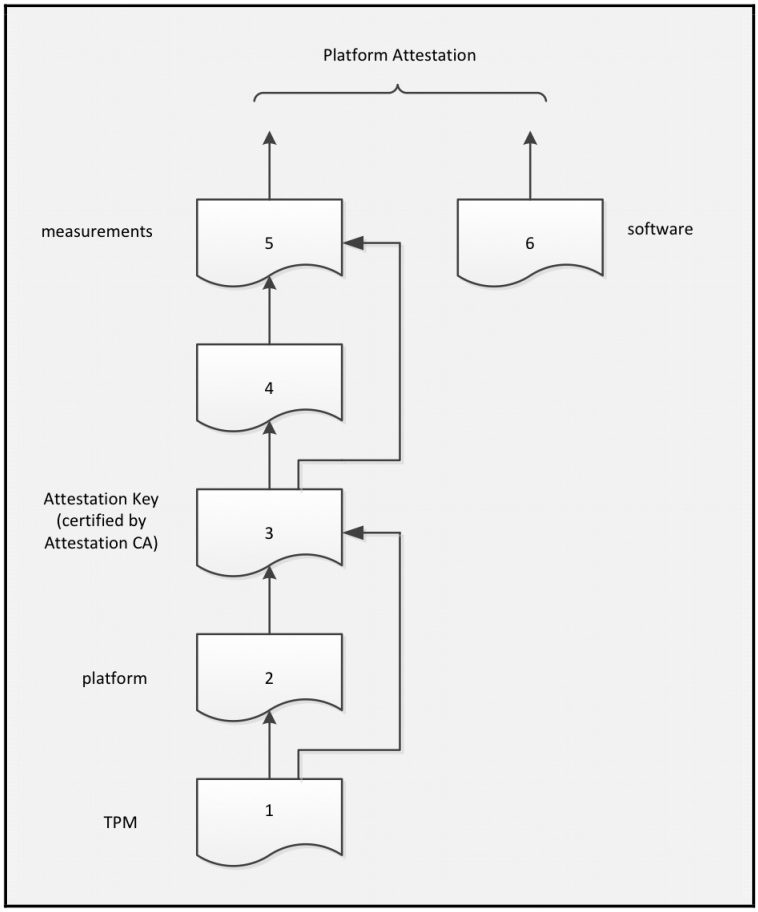


Figure 2.4. Attestation Hierarchy [4]

1. An external entity attests that the TPM is genuine and it is compliant with the TCG specifications. This attestation is defined as an asymmetric key commonly called *Endorsement Certificate* that asserts the EK to be trustworthy and could be used to identify the TPM.

2. An external entity attests that the platform contains a Root of Trust for Measurements, that the TPM installed is genuine, and that the path between the TPM and the RTM can be trusted. The attestation is commonly called *Platform Certificate*.

3. An *Attestation CA* attests that the *Attestation Key (AK)* was produced by a genuine but unidentified TPM. The AK can not be migrated and it may be used only to sign a digest created by the TPM.

4. A trusted platform certifies other key pairs creating a signature over the information that describes the key pair and its properties using the AK certified by the Attestation CA. The certification produced attests that the key pair was created by an authentic TPM and it has some properties.

5. A trusted platform generates *Quotes*, which are signed data structures computed over the hash of selected PCRs and a user-provided nonce, to attest a particular software/firmware state.

6. An external entity attests to the measurements computed by a trusted platform to vouch for the software/firmware state. The attestation is commonly called *Third-party Certification*.

2.3 Trusted Platform Module

The TPM is a specialised hardware component that is used to create and manage digital keys, certificates, and other secure data. It is designed to be tamper-resistant, but not tamper-proof. Tamper-proof means that it can not be attacked instead tamper-resistant means that it is supposed to resist various kinds of tampering attacks. Even though the TPM includes cryptographic modules, it is a rather slow cryptographic engine. An important aspect is that is certified *Common Criteria EAL4+*. The main features offered by the TPM are:

- *Hardware-based generation of random number*;
- *Secure generation of cryptographic keys* for limited uses;
- *Remote attestation* is used to verify if the system has not been compromised by a third party;
- *Binding* is used to encrypt data with a binding key and it makes it impossible to decrypt it outside the platform;
- *Sealing* is another feature, in which data is encrypted with a key internal to the TPM and to decrypt it the TPM state must be in the same state as when the data was encrypted;
- *Authentication of hardware devices* using the unique *Endorsement Key (EK)*.

The two most important versions of TPM are TPM 1.2 and TPM 2.0. The TPM 1.2 was the TCG's first attempt to solve the following major issues [5]:

- *Identification of devices* was a major problem because device identification was mainly performed using MAC or IP addresses, which could have been changed. Instead, the TPM introduced asymmetric keys, which can not be migrated to identify devices;
- *Secure generation of keys* was possible by integrating a Random Number Generator (RNG) inside the TPM, allowing the creation of more secure keys;

- *Secure Storage of keys* was achieved by the TPM in two different ways. The first one is to use Shielded Locations in which only a limited number of keys can be stored and these keys can be used only after granting the authorisation. The second technique is storing encrypted keys in Protected Locations inside the hard disk of the platform, this allows having a huge amount of keys that could not be possible if using the first technique because the storage inside the TPM is limited. The two techniques ensure that the keys are not disclosed but only those stored inside the TPM are also tamper-resistant;
- *NVRAM storage* was introduced in the TPM to store the EK because the hard disk of the platform was not the solution. Since, IT organisations when acquiring a new device, typically wipe the content of the hard disk and rewrite it.
- *Device health attestation* was achieved using the TPM to report on the state of the platform that attests if the system has been compromised instead, software solutions used before the introduction of the TPM might report the system to be healthy even though it was compromised.

The TPM 1.2 standard was used in billions of PCs, servers, network gear, embedded systems and other devices but the evolving of the Internet of Things and the huge demand for non-PC environment security led TCG to develop a new TPM specification to expand the TPM 1.2 legacy. The TCG developed TPM 2.0 using a library approach, which in 2015 become an international standard. This approach allows users to choose the features and the level of security or assurance required for their platform [7].

In the next sections, we are going to discuss the newest specification of TPM, exploring the architecture, the implementation, the difference between version 2.0 and version 1.2, and the possible use cases.

2.3.1 TPM 2.0 Architecture

The internal architecture of the TPM 2.0 is shown in 2.5.

The *Input/Output buffer* is the area in which the communication between the TPM and the host system takes place. The buffer is divided into the Input area which receives command data from the host system and the Output area which contains the response data produced by the TPM. There is no restriction that forces the I/O buffer to be physically isolated from other parts of the system. However, the only requirement that the specification imposes is that when the process of executing command data begins, the TPM must ensure it is using the correct value. This requires data to be in a Shielded Location before it is validated and modified.

The *Cryptography Subsystem* is the module that implements the TPM's cryptographic functions. It is composed of the Asymmetric Engine, Symmetric Engine, Hash Engine, RNG module and Key Generation Module. Furthermore, it is typically called by the Authorisation Subsystem, the Command Parsing module or the Command Execution module. The conventional cryptographic operations implemented are:

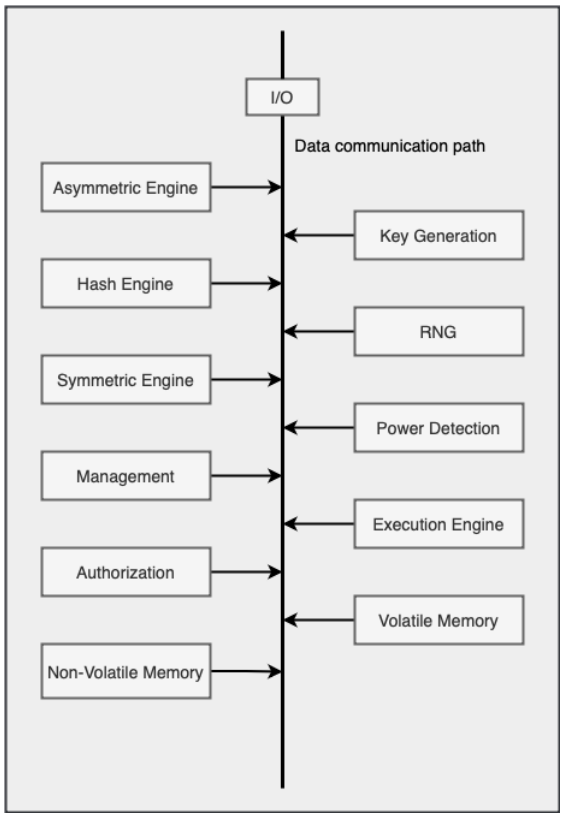


Figure 2.5. TPM 2.0 Architecture

- *Hash Functions*;
- *Symmetric and Asymmetric Encryption and Decryption*;
- *Symmetric and Asymmetric Signing (HMAC) and Signature Verification*;
- *Key Generation.*

Starting from the *Hash functions* , this can be used intrinsically by the TPM when performing other operations such as PCR Extend or directly by external applications that require the TPM to process some data. The TPM uses hash functions to implement Key Derivation Functions (KDF), to provide integrity checking and authentication. The hash function is denoted by $H_{algorithm}()$ in which the subscript refers to the hash algorithm name or identifier if missing is determined by context.

The TPM implements the *Hash Message Authentication Code (HMAC)* , which is a form of symmetric signature, it assures data to not be modified and identifies an entity with a shared key value. Attestation, identification and secret sharing are implemented using asymmetric algorithms. The algorithm used must be selected from the list of algorithms recognised by the TCG.

Signature operations can be implemented using either symmetric or asymmetric algorithms. For symmetric signatures, the TCG specifications defined the HMAC

signing schema to be the only method instead for the asymmetric algorithm, it depends on the algorithm (RSA or ECC). In addition, the key used to sign data may also have restrictions on the content that can be signed. It is possible to distinguish between restricted signing keys which must have a signing schema specified in the key definition and unrestricted keys which may have a signing scheme, or the signing schema is defined when the key is used. The signature verification requires a handle of a public key, a digest and the block that contains the signature. The TPM checks if the signing schema is compatible with the key selected and if the signature is valid, it produces a ticket that is an HMAC signature.

To encrypt *Protected Objects* stored outside the TPM and command parameters the TPM uses *Symmetric Encryption*. Every symmetric block cipher recognised by the TCG may be used for parameter encryption. In addition, XOR obfuscation should be supported because it can be used for confidential parameter passing.

The *Extend operation* is one of the most important and is used to make incremental updates to a digest value. Furthermore, it is used to update PCR, audit, and construct policies. The operation consists of using a hash function to combine new data with an existing digest. The notation is:

$$digest_{new} := H_{hashAlg}(digest_{old}||data_{new})$$

Where:

$H_{hashAlg}$ is the hash function associated with a specific bank of PCR

$digest_{new}$ is the new value of the digest after the Extend operation

$digest_{old}$ is the value of the previous digest

$data_{new}$ is the data to be hashed with the $digest_{old}$ value

Key generation is another important operation, which is used to produce two different types of keys. The first type is a key produced using the RNG to seed the computation (the seed is persistently stored on the TPM), the secret key obtained is then stored in the Shielded Location. The second type of key is a *Primary Key*, which is derived from a specific seed value based on the usage of an approved KDF. The TPM implements the KDF using two different schemes: ECDH KDF and the Counter mode KDF.

The *Random Number Generator (RNG)* is the source of randomness in the TPM and it is used for nonces, for randomness in signatures, and in key generation. The RNG is constituted of an entropy source, that is internal and should be at least more than one such as noise, clock variation, or other types of events. The entropy collector is the process that collects the entropy from the entropy source and removes bias. The result of the process is used to update the state register and it is later handled by the mixing function to produce random values. The mixing function is typically an approved hash function.

The *Authorisation Subsystem* is called by the *Command Dispatch* when a command is executed to check if the authorisations to use Shielded Locations have been provided. Depending on the command access to Shielded Locations may require no authorisation, single-factor authorisation or complex authorisation policy.

The *Random Access Memory (RAM)* is the area in which transient data is kept in the TPM, this means that data is lost when the power state changes to the TPM OFF state. There are three types of data stored inside the RAM:

- *Authorisation Session*
- *Objects* such as Keys and Data blob
- Data about the state of the system stored in the *PCRs*

Most of the values are stored in Shielded Locations except for a portion of memory reserved to the I/O buffer.

The *Platform Configuration Registers (PCRs)* are a set of registers that are automatically zeroed on reboot and contain measurements computed during the boot phase or the platform execution. The contents of the PCRs are set to know values on reboot and data can be added only using a specific command. This permits to have hash chains of measurements that make it impossible for a malicious client to forge good measurements. It is possible to store every measurement log inside a single PCR, but this is discouraged because it makes evaluating the different stages of platform evolution difficult.
The two main PCR operations are *Extend* and *Reset*. The Extend operation was extensively discussed previously. Instead, the Reset operation takes in input a PCR index, erases all previous contents of the register at that index, and sets it to the default value depending on the register implementation. The reset operation is automatically executed on boot for all registers, and for registers 17-22 when the DRTM is launched [1].
The TPM contains multiple PCRs and the number of registers available depends on the type of platform. The PC client platform requires 24 PCRs and each register is expected to store specific content, this is shown in the table 2.1 [1].

However, TPM 2.0 allows the creation inside the Non-Volatile Memory user-defined extend indexes which are essentially PCRs [5]. They can have distinct access policies and their own authorisation values. Moreover, TPMs 2.0 introduce PCR banks, which are a group of PCRs supporting a single hash algorithm.

Typically, with the term *Objects*, we refer to keys and data that are loaded into the TPM. The key data structure can be generalised for data blobs and the same can be done with the access properties. A data blob, which requires some PCRs to have defined values or an authorisation value for access is called Sealed Data Object and it is managed in the same way as keys are managed. The TPM2_Create is the command used to create the object, which is not automatically loaded but it must be done explicitly with TPM2_Load, to load both the private and public portions

Table 2.1. PC client expectation of PCR contents

PCR	*TCG defined usage*	*Other uses*
0	SRTM, BIOS, Host Platform Extensions	
1	Host Platform Configuration	
2	Option ROM Code	
3	Option ROM Configuration and Data	
4	IPL Code and Boot Attempts	Master Boot Record and Trusted GRUB
5	IPL Code Configuration and Data	
6	State Transitions and Wake Events	
7	Host Platform Manufacturer Specific	
8	Static Operating System	Trusted GRUB
9	Static Operating System	Trusted GRUB
10	Static Operating System	Application measurement register (Linux IMA)
11	Static Operating System	
12	Static Operating System	Command Line arguments for boot loader
13	Static Operating System	Measurements of user files
14	Static Operating System	Loaded files, kernel, modules, etc
15	Static Operating System	
16	Debug	
17	DRTM component measurements	
18	Measurements of signing authorities	
19	Measurements of DLME, or measurements of signing authorities	
20-22	Reserved for use of DLME	
23	Application Support	

otherwise TPM2_LoadExternal to load the public portion.

Sessions are used to control a sequence of operations such as audit actions, provide authorisations for actions or encrypt command parameters. A session is created using specific TPM commands and a handle is assigned when available. It is possible to create a session inside the same RAM shared with the object store or if isolation is needed they can be stored separately in dedicated memory areas.

The *Non-Volatile Memory (NV)* stores TPM persistent state and it contains Shielded Locations that can be accessed only with Protected Capabilities. The NV memory can be used for allocation and used by the platform and authorised entities, in detail it is used to hold:

- *NV Index Values*;

- *Objects made persistent* by the TPM2_EvictControl command;
- *TPM power states* saved by TPM2_Shutdown;
- *Persistent NV data*

The *NV Index* is a special NV data structure that may be defined by the OS or the platform in order to store persistent data values.

The *Power Detection Module* manages the TPM power states, which are only ON and OFF, and it is related to the platform states. The specification of the binding between the TPM and the platform establishes that the TPM must be notified of all power state changes, such that:

- Any power transition requiring the RTM reset also causes the TPM to reset
- Any power transition requiring the TPM reset also causes the RTM to reset

2.3.2 Implementations

Nowadays, the most popular types of TPM are five. These offer different trade-offs between security, features, and cost, this is shown in the 2.6. The TPM implementations in order of security, according to the TCG are [7]:

1. The Discrete TPM is a dedicated chip attached directly to the motherboard using a low-performance interface. The chip integrates a cryptographic processor, RAM, ROM and Flash Memory. The discrete TPM provides the highest level of security and it is used for the most critical scenarios. To obtain this level of security, it is designed, built, and evaluated to resist to tampering attacks and other sophisticated attacks.

2. The Integrated TPM is still a hardware TPM but it is embedded in another chip. In this case, it is not required to resist tampering instead, since it is still hardware base it is resistant to software bugs. The security level is one step down to the Discrete TPM.

3. The Firmware TPM is the first t software-based TPM type. It is implemented in protected software and because there is no dedicated chip the code runs on the main CPU. The code is executed and protected by the Trusted Execution Environment which is an execution environment that is separated from the rest of the programs running on the CPU. Since it is software-based it is not tamper-resistant and it is dependent on the TEE operating system, bugs in the code that runs in the TEE, etc. The security level offered is lower that the previous implementation.

4. The Virtual TPM, or Hypervisor TPM, is a software implementation of the TPM. The TPM functionalities are provided by the hypervisor to be used in isolated execution environments such as Virtual Machines. The security level is comparable to the Firmware TPM.

5. The Software TPM is typically implemented as a software emulator of the TPM. However, it has many vulnerabilities, not only tampering but because it is executed in user space it depends on the bugs in the operating system and in the running applications. For this reason, it is useful for development purposes.

TRUST ELEMENT	SECURITY LEVEL	SECURITY FEATURES	RELATIVE COST	TYPICAL APPLICATION
DISCRETE TPM	HIGHEST	TAMPER RESISTANT HARDWARE	$$$	CRITICAL SYSTEMS
INTEGRATED TPM	HIGHER	HARDWARE	$$	GATEWAYS
FIRMWARE TPM	HIGH	TEE	$	ENTERTAINMENT SYSTEMS
SOFTWARE TPM	NA	NA	¢¢	TESTING & PROTOTYPING
VIRTUAL TPM	HIGH	HYPERVISOR	¢	CLOUD ENVIRONMENT

Figure 2.6. TPM implementations [7]

2.3.3 TPM 2.0 vs TPM 1.2

The TPM 2.0 was designed by the TCG to extend the specifications of TPM 1.2 and to overcome some of its limitations. The most important was the SHA-1 algorithm, on which 1.2 version was mainly based and which was subject to cryptographic attacks. The new TPM 2.0 capabilities are [5]:

- *Algorithm Agility*;
- *Enhanced authorisation*;
- *Quick key loading*;
- *Non-brittle PCRs*;
- *Flexible management*;
- *Identifying resources by name.*

Algorithm agility

The TPM 1.2 specification supported only SHA-1 for hashing and RSA as an asymmetric algorithm. The TPM 2.0 specification, introduces flexibility by allowing new algorithms. Instead of having SHA-1 which was deprecated by NIST and was not accepted after 2014, SHA-256 and any algorithm recognised by the TCG can be used. Symmetric algorithms were introduced allowing keys to be stored off the chip encrypted with symmetric encryption like *Advanced Encryption Standard (AES)*. In addition to RSA, new asymmetric algorithms were introduced such as *Elliptic Curve Cryptography (ECC)*. The new specification was thought to be upgradable this means that algorithms can be removed or added.

Enhanced authorisation

The TPM 1.2 had a very complicated specification regarding the authentication mechanism. The TPM could be managed using the owner authorisation or the physical presence instead, the EK could be used only by gating the owner authorisation. Keys had two authorisations: one to make duplicates of the key and one to use the key. The same happens for PCRs and particular localities which had two different authorisations, one for reading and one for writing. Some owner-authorised commands and keys made things even more complicated. The TPM 2.0 instead, has only one unified authorisation model called *Enhanced Authorisation (EA)*, used for keys, data blobs, NV indexes, and other objects. It is based on the same specification as TPM 1.2 but it adds new methods of authorisation and simplifies the overall schema. Some of the following kinds of authorisation are extended from TPM 1.2, others are introduced:

- *Password in the clear*: is used mainly for the BIOS, where there is no need to use the HMAC key;
- *HMAC key*: is preferred when the OS is not trusted but the software talking with the TPM is trusted;
- *Signature* and *Signature with additional data*: the first one can be used to implement a smart card while the second one can be used for fingerprinting where additional data is needed;
- *PCR values*: can be used to prevent access to data stored in the TPM if the system has been compromised;
- *Locality*: can be used to indicate where a particular command comes from;
- *Time*: is used to define policies to limit access to a key to certain times;
- *Internal Counter*: is used to limit access to objects when the internal counter has certain values;
- *Value in an NV index*: is useful for revoking access to a key;
- *Physical presence*: requires that the TPM is physically accessed by the user.

Additionally, it is possible to create more complicated policies by combining some of the previous with logical AND or OR operations.

Quick key loading

In version 1.2, when a key was loaded it had to be decrypted using the parent's private key. This process was time-consuming and for this reason, if this operation was done multiple times in a session, to make the process faster, the key was encrypted with a symmetric key and loaded in the cache. The problem was that after TPM was turned off, the cache was erased, and the process had to start from the beginning. The TPM 2.0 solves this problem using symmetric encryption for keys stored on external memory.

Non-brittle PCRs

The fragility of PCRs was one of the many problems of TPM 1.2. An example of PCR fragility can be found in the BIOS upgrade scenario, in which before the BIOS upgrade the TPM 1.2 had to unseal all the secrets locked to PCR 0, representing the BIOS, and then resealed after the upgrade. Instead in TPM 2.0, it is possible to seal things to a PCR value that was approved by a particular authority, this means that a secret can be released only if the PCR is in the state approved by a particular signer.

Flexible management

In TPM 1.2 specification, only two authentications existed: the *Storage Root Key (SRK)* and the *owner authorisation*. The owner authorisation was used for many purposes and giving the same authorisation to so many roles made it very difficult to manage these roles independently. For these reasons and many others in the TPM 2.0 family, the roles are separated in the specification itself. This is possible by assigning different authorisations and policies to each role, and by defining four different hierarchies, which are:

- *Storage hierarchy*: replicates the TPM 1.0 and TPM 1.2 family specification for SRK;
- *Platform hierarchy*: is mainly used by the BIOS and not by the end user;
- *Endorsement hierarchy*: prevents the TPM to be used for attestation without any approval;
- *Null hierarchy*: differs from the previous one because it does not require any password or authorisation policy. The TPM is used as a simple cryptographic co-processor.

Identifying resources by name

In version 1.2, resources were identified by the handle. This means that if two resources, A and B, had the same authorisation, the software can trick the user into authorising a different action on resource B instead of performing it on resource A. Version 2.0, resolve this possible attack by identifying resources by their name, which is cryptographically bound to them. In addition, the name can be signed using a TPM key to provide evidence that the name is correct.

2.4 TPM Software Stack 2.0

The *TPM Software Stack 2.0 (TSS 2.0)* is a software specification designed by the TCG to provide a standard API to programmers for accessing the functions of the TPM 2.0 [8]. This is possible thanks to the abstraction of the hardware provided by the TSS, which permits isolating application developers from the low-level details of interfacing with the TPM. In addition hardware abstraction permits to write applications that will work independently of the hardware, OS or environment used. The TSS consists of multiple software layers, that allow TSS implementations to scale from resource-constrained low-end systems to high-end systems. Each layer is intended to use resources and functionalities from the layer below and provide its functionalities to the layer above. The higher the layer, the lower the knowledge of the low-level details. The overall architecture is shown in Figure 2.7

TPM Device Driver

The *TPM Device Driver* is the first layer that sits on top of the TPM interface. Its role is to manage the communications with the TPM by receiving a buffer stream of bytes and performing the operations needed before sending it to the TPM and by returning the response up to the stack.

Resource Manager (RM)

Similar to a virtual memory manager, the Resource Manager (RM) manages the TPM context. It efficiently swaps objects, sessions, and sequences in and out of the TPM memory, which is limited in capacity, as required. This functionality remains mostly concealed from the upper layers of the TSS and is not mandatory. Nonetheless, if the RM is not implemented, the upper layers will assume responsibility for managing the TPM context.
The RM intercepts the command byte stream, checks which resources need to be loaded, determines how many resources need to be swapped out to load the newest and finally loads the resources needed. In case the resources to load are objects and sequences, the RM needs to create virtual handles for these resources before returning them.
Typically the RM is combined with the TAB into one component per TPM. They both are transparent to the upper layers of the stack and this means that it is not

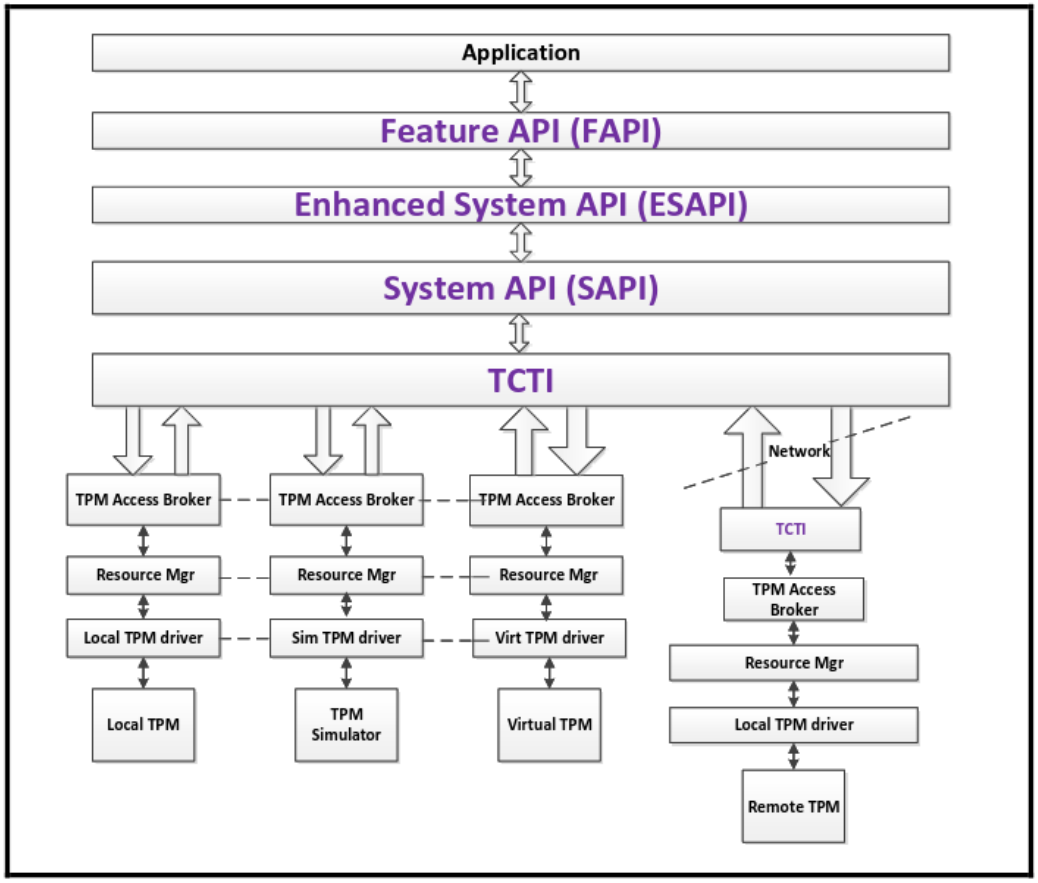


Figure 2.7. TSS Architecture [8]

mandatory to implement them. However, if they are not implemented the upper layers need to include those functionalities, especially for applications running in a multi-threaded environment.
Application transparency is accomplished by enabling applications to seamlessly operate with a Resource Manager (RM) without necessitating any modifications or recompilation. Applications that originally operate without an RM can smoothly transition to using an RM without any code changes. However, it is crucial for these applications to be aware of and support the supplementary error codes that arise when an RM is present [9].

TPM Access Broker (TAB)

The TAB has two responsibilities. The first one is to control and synchronise access to the TPM. Since the TAB is one per TPM, it is typically shared between multiple processes. The TAB ensures that a process accessing the TPM can carry out a TPM command without any disruptions caused by other concurrent processes. The second responsibility is to prevent processes from accessing resources they don't own such as objects, sequences, or TPM sessions[9].
To provide a more detailed explanation, the TAB layer offers multi-user support for a single TPM device. Scheduling occurs on a per-command basis, where a

single caller command may correspond to multiple TPM commands. The TAB ensures that the TPM executes all commands associated with a caller command before moving on to serve the next user in its scheduler. The TAB maintains a wait queue of caller commands awaiting scheduling. The TAB is authorised to optimise a command cancellation request received from the caller. If a command has not yet been transmitted to the TPM, the TAB can promptly respond with a TPM_RC_CANCELED status and remove the command from the queue.
The expected implementation of the TAB/RM functional properties is through a daemon, which can potentially incorporate additional features like access control restrictions. The specific internal interfaces between this daemon and the TAB/RM module are determined by the implementation. A vendor offering such a daemon is anticipated to provide a complementary set comprising the daemon itself and a TCTI Client-Provider. The interface between the TCTI and the RM is not specified by TCG.

TPM Access Broker and Resource Manager

As previously mentioned the TAB and RM are typically combined together. The most relevant example is the tpm2-abrmd, which is a user-space daemon, developed according to the TCG specifications, that works in conjunction with the kernel's TPM Device Driver to provide resource management and access control for TPM operations [10].
During the boot process of the operating system, the tpm2-abrmd daemon should be initiated. Communication between the daemon and clients that interact with the TPM occurs through a combination of DBus and Unix pipes. DBus facilitates tasks such as discovery, session management, and certain API calls. On the other hand, pipes are utilised to transmit TPM commands and receive corresponding responses between the client and server components.
Another solution to the tpm2-abrmd, in Linux, is the in-kernel RM, which was introduced starting from kernel version 4.12 . It is a kernel-space daemon and it offers the same functionalities as the tpm2-abrmd. However, the biggest problem is that there are TPM 2.0 users who are still running older kernel versions, for this reason it is recommended that developers utilise the tpm2-abrmd as the default option to ensure the broadest support and stability. It is possible by organising the code properly to switch between different TPM Command Transmission Interface (TCTI) modules, making the migration to the in-kernel RM a smooth process.

TPM Command Transmission Interface (TCTI)

The TCTI is the layer in charge of handling the communication to and from the lower layers of the stack using two interfaces to communicate with the TPM: the *legacy TIS interface* and the *Command/Response buffer (CRB)*. Depending on the TPM implementation, different interfaces are needed (e.g. for local hardware TPMs, firmware TPMs, virtual TPMs, remote TPMs and software TPM simulators) [11]. The TCTI API is intended for utilisation across a wide spectrum of computing devices, spanning from deeply embedded systems to server operating systems. To utilise the TCTI, one must possess knowledge of standard device

driver interfaces. The purpose of the TCTI is to transmit marshalled commands to the TPM and receive marshalled responses in return. It is built to enable the switching of modules during run-time. The TCTI incorporates conventions and aids that facilitate run-time loading, while still permitting compile-time linking without conflicts in namespaces. The initialisation process of the TCTI interface is carried out in a manner specific to each driver. The TCTI context structure is the component that is created when an upper layer wants to communicate with the TPM. In addition, the TCTI context structure is created as one per-process and per-TPM structure.

Marshalling/Unmarshalling (MUAPI)

The MUAPI is required by both SAPI and ESAPI. It has two functionalities: marshalling is the operation of building the TPM command byte stream; the unmarshalling is the operation of decomposing the TPM response byte streams.

System API (SAPI)

The SAPI is the layer that provides access to all the functionality of a TPM 2.0 implementation. It is mainly used wherever low-level calls to the TPM are made, such as firmware, BIOS, OS, etc. The goals of the SAPI specifications are:

- provide access to all functionality;
- be usable independently from memory-constrained environments to multiprocessor servers;
- make programmer's job easier;
- support synchronous and asynchronous calls to TPM;
- SAPI implementations do not need to allocate any memory

Enhanced System API (ESAPI)

The ESAPI is an interface designed to sit above the System API. The goals of the ESAPI are similar to the previous ones listed for the SAPI but there are some differences [12]:

- provide cryptographic operations on data passed to or received from the TPM;
- provide enhanced session management functionality on top of SAPI functionality;
- provide 100% TPM functionality.

Even though ESAPI is less complex than SAPI, it still requires a good knowledge of the interface of TPM 2.0. In addition, SAPI provides a better implementation for those memory-constrained environments than ESAPI because it uses less RAM and it doesn't include some cryptographic operations.

Feature API (FAPI)

The FAPI is the highest layer of the TSS architecture and it sits above the ESAPI [13]. The main goal are:

- provide the most-used functionalities of TPM 2.0 to programmers. Therefore, the Feature API does not encompass all the corner cases and advanced functionalities that the TPM 2.0 is capable of supporting;
- reduce the number of calls that have to be done and the number of parameters that have to be defined;
- introduce profile files to create default selections of keys attributes (e.g., algorithm, key size, signing schema and crypto mode) written in JSON;
- provide wherever applicable salted HMAC sessions and parameter encryption for all communications with the TPM;

A comparison of code sizes between TSS interfaces is shown in Figure 2.8:

When considering code size, the Feature API tends to have a smaller footprint, making it more lightweight compared to other interfaces. It focuses on providing a simplified set of functionalities, which contributes to its reduced code size.

On the other hand, the Extended System API (ESAPI) and System API (SAPI) typically have larger code sizes compared to the Feature API. This is because ESAPI and SAPI encompass a broader range of functionalities, offering more comprehensive coverage of the TPM's capabilities. The additional features and flexibility provided by these interfaces contribute to their larger code size.

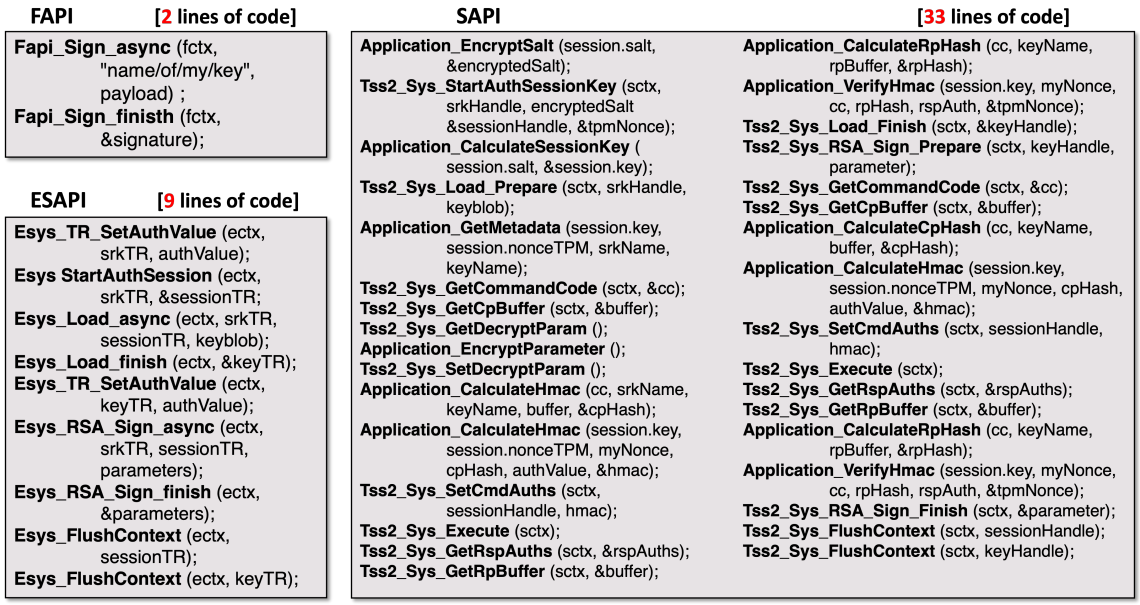


Figure 2.8. Code size comparison between TSS Interfaces [14]

Chapter 3

ESAPI

In order to cover all the functionalities provided by the TPM 2.0, the ESAPI interface was chosen for the scope of this thesis, despite the FAPI offering a simpler approach for application development based on TPM 2.0. The decision was made because the ESAPI interface provides more comprehensive coverage of the TPM's capabilities.

3.1 Overview

The Enhanced System API (ESAPI) is designed to function as an intermediate layer between the System API and applications utilising TPM functionalities [12]. Its main purpose is to simplify the programming complexity for applications that require both system-level TPM calls and cryptographic operations on the data exchanged with the TPM.

ESAPI is particularly beneficial for applications that need to utilise secure sessions for operations such as HMAC (Hash-based Message Authentication Code), parameter encryption, parameter decryption, TPM command auditing, TPM policy operations, and context/object management.

While the ESAPI offers a less complex alternative to the System API for cryptographic interactions with the TPM, it still requires a deep understanding of the TPM 2.0 interface. Therefore, it is expected that only experts developing applications will make use of the ESAPI, while typical applications may prefer higher-level interfaces like the Feature API.

The distinctions between ESAPI and SAPI are [15]:

- ESAPI provides simpler handling, from the application's perspective, of certain functions compared to SAPI. These functions include starting and salting sessions, calculating HMACs, and encrypting/decrypting sessions;
- Implementing an application using the SAPI interface offers advantages for the following reasons: SAPI implementations have a smaller footprint primarily because they exclude cryptographic functions. SAPI is suitable for use in environments with limited heap space. Additionally, SAPI allows saving RAM by allocating less memory for data structures.

3.1.1 ESAPI Key features

The ESAPI API exhibits the following key characteristics [12]:

1. **Written in C99**: The ESAPI is implemented in C99 to ensure compatibility across a wide range of operating systems and simplify the creation of language bindings for other programming languages (such as Java, Python, etc.);

2. **Fine-grained Control over Input Parameters**: Applications using the ESAPI have control over all input parameters and data elements included in TPM commands, except for HMAC session calculation;

3. **Simplified TPM Command Handling**: The ESAPI is designed to enable applications to send commands to the TPM with a single or minimal number of function calls, particularly when utilising sessions. This streamlines the process compared to the SAPI;

4. **Abstraction of Formatting and Crypto Handling**: The ESAPI abstracts the application from the problems related to formatting and cryptographic operations, performing these tasks within the ESAPI layer or lower layers;

5. **Session Management**: The ESAPI handles various aspects of session management, including session key calculation, session salt creation and encryption, HMAC calculation, resource name tracking, and session binding;

6. **Additional Functionality**: The ESAPI provides additional capabilities, such as HMAC calculation/verification with an *authValue* input, encryption/decryption of the first parameter with a key input, support for audit sessions, and cryptographic flexibility/agility;

7. **State Management and Contexts**: ESAPI implementations do not require maintaining state in global variables between function calls. Instead, state information is stored in contexts provided to the application. Multiple ESAPI contexts can be accessed simultaneously, but two threads can not access the same ESAPI context simultaneously;

8. **Synchronous and Asynchronous Call Support**: The ESAPI supports both synchronous and asynchronous call models, accommodating different application requirements;

9. **Alignment with TPM 2.0 Specification**: The ESAPI closely follows the command schematics outlined in the TPM 2.0 Command Specification defined by the TCG. This design choice aids programmers in understanding the code and facilitates correlation with the specification. Function input and output parameters are ordered to match this specification, and names are kept as similar as possible. Additionally, the ESAPI includes functionality to automatically set up and tear down sessions as part of larger operations;

10. **Memory Handling**: The ESAPI implementation provides memory for ESAPI output parameters, whereas memory for ESAPI input parameters is provided by the caller;

11. **Simplification for the Caller**: The ESAPI implementation handles tasks that System API implementations typically require but with improved ease of use;

12. **Cryptographic Operations and Data Management**: The Enhanced System API implementation handles a significant portion of the cryptographic operations and data management associated with TPM 2.0 sessions;

13. **Handling of Object**: For objects with a `nameAlg` digest, the Enhanced System API implementation performs hashing of `TPM2B_AUTH` values with the `nameAlg` of the object when the length of the `TPM2B_AUTH` value exceeds the length of the `nameAlg` digest size. This approach ensures that authentication values larger than the `nameAlg` hash length do not trigger a `TPM2_RC_AUTH_FAIL` error during object creation.

3.1.2 Top-Level usage

The application follows a series of steps, alongside the ESAPI, to establish contexts and sessions:

1. Initialisation of ESAPI Context:
   (a) The application initialises an ESAPI context;
   (b) The ESAPI dynamically allocates memory on the heap to track the necessary metadata for each TPM resource;

2. Invocation of TPM Commands and Resource Handling:
   (a) The application calls TPM commands, with resource handling partially automated by the ESAPI;
   (b) If a command creates a new resource, the ESAPI generates a new `ESYS_TR` object to track the resource and its metadata;
   (c) For static TPM resources such as NV and persistent keys, `ESYS_TR` objects can be serialised/deserialised from disk or directly created from the TPM;

3. Flushing or Closing `ESYS_TR` Objects:
   (a) The application flushes or closes `ESYS_TR` objects through ESAPI calls, allowing the ESAPI to free allocated memory;
   (b) If a command destroys a TPM resource, the ESAPI automatically releases the memory used by the object metadata and marks the `ESYS_TR` object as invalid;

(c) The application may close the `ESYS_TR` to release the memory utilised by the ESAPI without affecting the resource within the TPM;

4. Closing of ESAPI Context: The application closes the ESAPI context when it has finished its operations.

Throughout these steps, the application and ESAPI collaborate to initialise contexts, start sessions, create/load objects, and manage memory allocation. The ESAPI's role includes dynamically allocating memory for metadata, automating resource handling, managing session metadata and calculations, and freeing memory as needed.

3.2 Structures

3.2.1 ESYS_CONTEXT

The `ESYS_CONTEXT`, which serves as the ESAPI context, is a concealed structure that encapsulates all the necessary information for the ESAPI module to store between calls, eliminating the need for global state variables. It comprises the following components:

- Information for TPM communication, including the System API command context and the TPM Command Transmission Interface (TCTI) context;
- Metadata associated with each `ESYS_TR` object;
- State information for internally used libraries, such as cryptographic libraries, and cached TPM capabilities.

Memory allocation for the `ESYS_CONTEXT` is handled by the ESAPI. The life cycle of an `ESYS_CONTEXT` can be summarised as follows:

- Create an `ESYS_CONTEXT` by utilising the `Esys_Initialize()` function;
- Create or deserialise metadata from disk storage for the session and resource information structures;
- Utilise sessions and resources in TPM commands;
- Serialise resource information structures back to disk;
- Conclude the usage of the `ESYS_CONTEXT` by invoking `Esys_Finalize()` to close it.

In this way, the `ESYS_CONTEXT` provides a centralised storage mechanism for essential information, allowing the ESAPI module to operate without relying on global state variables.

3.2.2 ESYS_TR

The metadata associated with TPM resources is stored in an opaque structure that is linked to an ESYS_CONTEXT and referenced by an ESYS_TR handle. An ESYS_TR object created within a specific ESYS_CONTEXT can only be utilised within that particular ESYS_CONTEXT.

The metadata for an ESYS_TR object encompasses various information, including:

- The TPM handle for the resource, such as PCR, NV area, TPM objects (keys, data), or hierarchies;
- The authValue of the resource, if applicable;
- The public area of the resource represented by TPM2B_PUBLIC or TPM2B_NV_PUBLIC;
- The resource name, which can be computed from the aforementioned details.

The life cycle of an ESAPI resource can be summarised as follows:

1. Create an ESYS_TR object:
   (a) For transient TPM objects, the appropriate load function (e.g., Esys_Load(), Esys_LoadExternal()) is called, and the ESAPI module generates a new ESYS_TR object;
   (b) For persistent resources (e.g., evict keys, NV areas):
      - Create an ESYS_TR object using Esys_TR_FromTPMPublic();
      - Alternatively, deserialise metadata from disk using Esys_TR_Deserialize();
   (c) For TPM meta-objects (e.g., hierarchies and PCRs), each ESYS_CONTEXT contains a preexisting singleton ESYS_TR object that can be directly used;
   (d) Set the authValue for the resource using the Esys_TR_SetAuth() function;
2. Perform TPM calls that reference the resource within the handle area using the Esys_commandName() template;
3. Serialise metadata to disk using the Esys_TR_Serialise() function;
4. Destroy an ESYS_TR object:
   (a) Flush or remove the resource from the TPM when no longer needed, utilising functions like Esys_FlushContext(), Esys_NV_UndefineSpace(), Esys_NV_UndefineSpaceSpecial(), Esys_EvictControl();
   (b) Close the object using Esys_TR_Close() to release the associated metadata while keeping the resource intact within the TPM.

In this way, the ESYS_TR objects serve as handles to TPM resources, allowing for the creation, utilisation, serialisation, and destruction of resources within the context of an ESYS_CONTEXT.

TPM resource data is used by the ESAPI module to associate a TPM resource handle with relevant information maintained by the ESAPI. This resource data typically includes the public information associated with the resource, as well as the corresponding authorisation value, the authValue, that grants authorisation for the resource's usage.

3.2.3 ESYS_SESSION

ESAPI session data is stored within an opaque session information structure in an ESYS_CONTEXT and referenced by an ESYS_TR. The session information structure encompasses various types of data, including:

- TPM handle for the session;
- Information regarding session attributes to be used in the subsequent command;
- Details related to the bound state and encrypted salt of the session;
- Information associated with the generation and storage of the session key;
- Session nonce;
- Policy session-specific information.

The life cycle of an ESAPI session can be summarised as follows:

1. Start the session using the Esys_StartAuthSession() function;
2. Perform TPM calls that reference the session using the Esys_commandName() function template;
3. Close the session using the Esys_FlushContext() function to immediately evict the session, or set the continueSession bit to false using Esys_TRSess_SetAttributes() to evict the session the next time it is utilised.

By following this life cycle, the ESAPI manages the session data within the opaque session information structure, allowing for the initiation, utilisation, and closure of sessions within the ESYS_CONTEXT.

TPM session data is essential for the ESAPI module to manage the state of a specific TPM session and facilitate secure communication with the TPM. This data includes crucial information such as the session key, session attributes, and session defaults. Some of this information, like the session key, is also stored within the TPM itself. However, certain details, such as session default attributes, are specifically designed to simplify application requirements and are not shared with the TPM.

3.3 Functions

Previously, during the definition of the ESAPI key features, we have said that the ESAPI tries to mimic the TCG specification for the TPM 2.0 commands to help programmers easily develop and understand the code.

In the following paragraphs, we are going to discuss and provide a general overview of the ESAPI functions used during the development of this thesis. This will provide a comprehensive understanding of the input and output parameters involved in these functions where possible, without repeating the one already discussed.

3.3.1 ESAPI Exclusive Command

Initialise

The `Esys_Initialize()` function initialises an `ESYS_CONTEXT` context which will be used for subsequent ESAPI functions that require an open ESAPI context. Definition:

```
TSS2_RC_Esys_Initialize(ESYS_CONTEXT **esysContext,
   TSS2_TCTI_CONTEXT *tcti, TSS2_ABI_VERSION *abiVersion);
```

The *esysContext* is a pointer to the opaque `ESYS_CONTEXT` blob, it is allocated by the ESAPI and by definition it must not be null. The *tcti* is a reference pointer to a `TCTI_CONTEXT` used by the ESAPI to communicate with the TPM. If TCTI is NULL it will open a connection to the local TPM. The *abiVersion* is a reference to a pointer to the `ABI_VERSION` that the application requests and this must match the ESAPI's `ABI_VERSION` otherwise an error is returned.

Finalise

The `Esys_Finalize()` function closes an `ESAPI_CONTEXT` that was previously created with `Esys_Initialize()`. The `ESAPI_CONTEXT` is cleared but it is best practice to flush all `ESYS_TR` objects before invoking it otherwise the TPM becomes crowded with objects that are supposed to be flushed. Definition:

```
void Esys_Finalize(ESYS_CONTEXT **esysContext);
```

Free

Many ESAPI functions allocate memory for output parameters and the `Esys_Free()` function is used to free the allocated memory. This function has a similar role of `free()` for `malloc/calloc()` in C or `delete()` for `new()` in C++. Definition:

```
void Esys_Free(void *ptr);
```

The *ptr* is a pointer to memory to be freed.

Set Authentication

This function set the authentication value associated with an ESYS_TR object. The authentication value is used if the ESYS_TR object is used in a command that requires authValue based authentication, such as password-authentication, HMAC-authentication, PolicySecret or PolicyPassword-enabled policy sessions. This value is stored in the ESYS_CONTEXT and if we want to delete the value we need to overwrite it with NULL. This can be done by passing NULL inside the function for the authValue field. Because the authValue is not part of the serialisation blob for the ESYS_TR object, it needs to be set after any serialisation/deserialisation.

Definition:

```
TSS2_RC Esys_TR_SetAuth(ESYS_CONTEXT *esysContext, ESYS_TR
   handle, TPM2B_AUTH const *authValue);
```

The esysContext is a pointer to the opaque context blob used and must not be null.
The handle is the identifier of the ESYS_TR object for which the authValue is set.
The authValue is a pointer to the TPM2B_AUTH structure containing the password. The ESAPI module will copy this value into its internal state. It is possible to reset the authorisation value associated with the object by passing NULL, or an authValue of length zero.
The TPM2B_AUTH is a structure used for an authorisation value and limits an authValue to being no larger than the largest digest produced by a TPM. In case the authValue is greater than the length of the nameAlg digest associated with the object, the ESAPI shall hash the contents of the authValue buffer using that object's nameAlg algorithm and set the size field to the length of the digest in bytes in the ESYS_CONTEXT state.

Set Authentication Parameters

This function is used to set or clear session attributes, such as continueSession, encrypt or decrypt. The new session attributes are computed in the following way:

$$attributes = (attributes \quad \wedge \neg mask) \mid (flags \wedge mask)$$

If the function is never called the default values for the session flags are: continueSession bit is set and all other bits are clear. If the continueSession value is CLEAR(0) the ESYS_TR object for the session is invalidated after the next successful TPM response. After which, the ESAPI must set the internal type/status of the ESYS_TR sequenceHandle to invalid so that it cannot be used. In this case, the ESYS_TR does not need to be flushed with Esys_FlushContext() or closed with Esys_TR_Close().
Definition:

```
TSS2_RC Esys_TRSess_SetAttributes(ESYS_CONTEXT *esysContext,
   ESYS_TR session, TPMA_SESSION flags, TPMA_SESSION mask);
```

The esysContext is a pointer to the opaque context blob on which we are operating. The session parameter refers to the session, for which we are setting the session attributes. The flags parameter indicates which bits are modified, any bits that are clear in the mask will not be modified. If we want to set all bits simultaneously we can use 0xFF otherwise. The mask parameter of type TPMA_SESSION, is an octet used to identify the session type. In the following Table 3.1 is possible to check the parameters and the possible usages [8].

3.3.2 ESAPI Command Template

The ESAPI specification outlines a collection of functions that follow a template format. For each command defined in the TCG specification for the TPM 2.0 commands [16], the ESAPI includes three corresponding functions: Esys_commandName(), Esys_commandName_Async(), and Esys_commandName_Finish(). These functions offer both synchronous and asynchronous invocation of TPM commands.

When using the synchronous approach, the Esys_commandName() function is called directly. It causes the ESAPI to execute the TPM command and blocks until the execution is complete. Within the function's scope, Esys_commandName_Async() is invoked, setting the timeout to indefinite. Subsequently, the Esys_commandName_Finish() function is called, and the Esys_commandName() function remains blocked until the return code is TSS2_BASE_RC_TRY_AGAIN.

On the other hand, the Esys_commandName_Async() function is used for asynchronous execution of TPM commands. It does not block while the TPM executes the command. To retrieve the TPM's response parameters, the Esys_commandName_Finish() function must be called.

The Esys_commandName_Finish() function is responsible for obtaining the results of an asynchronous TPM command. Its blocking behaviour is determined by the timeout set using Esys_SetTimeout(), which is non-blocking by default. When the TPM returns an object name, the ESAPI records these parameters within the associated ESYS_TR object instead of returning them to the caller. Therefore, the output parameter's name is not present in certain functions such as Esys_Load(), Esys_CreatePrimary(), Esys_LoadExternal(), and Esys_CreateLoaded(). However, this rule does not apply to Esys_ReadPublic(), which can be used to retrieve the output parameter's name, as well as Esys_NV_ReadPublic().

Create

The TPM2_Create() command is utilised to generate an object that can be subsequently loaded into a TPM using the Esys_Load() function. Upon successful completion of the command, the TPM will create the new object and provide the caller with its creation data (creationData), public area (outPublic), and encrypted sensitive area (outPrivate). The responsibility of preserving the returned data lays with the caller. Prior to usage, the object needs to be loaded. The sole distinction between the inPublic and outPublic lays in the unique field.

Table 3.1. TPMA_SESSION Parameters

Bit	*Parameter name*	*Description*
0	continueSession	SET(1): session remain active after successful completion of the command. CLEAR(0): TPM should close the session and flush any related context when the command completes successfully.
1	auditExclusive	SET(1): this setting is only allowed if the audit attribute is SET. In a command this setting indicates that the command should only be executed if the session is exclusive. In a response, it indicates that the session is exclusive. CLEAR(0): the opposite of the SET. In addition, if the audit bit is CLEAR also the auditExclusive must be CLEAR both in the command and the response.
2	auditReset	SET(1): allowed only if the audit attribute is SET. In a command, this setting indicates that the audit digest of the session should be initialised and the exclusive status of the session SET. CLEAR(0): indicate that the audit digest should not be initialised. This bit is always set to CLEAR in a response. If the audit bit is CLEAR also the auditReset must be CLEAR in both command and response.
4:3	reserved	shall be CLEAR.
5	decrypt	SET(1): in a command indicates that the first parameter in the command is symmetrically encrypted. The TPM will decrypt the parameter after performing any HMAC computation and before unmarshalling the parameter. This attribute may only be SET in one session per command. CLEAR(0): session not used for encryption. For password auth this attribute will be CLEAR in both command and response.
6	encrypt	SET(1): in a command this setting indicates that the TPM should use this session to encrypt the first parameter in the response. CLEAR(0): session not used for encryption. For password auth, this attribute will be CLEAR in both the command and the response.
7	audit	SET(1): both in a command and in a response it indicates that the session is for audit and that the auditExclusive and auditReset have meaning. The decrypt and encrypt fields can be SET or CLEAR. CLEAR(0): session is not used for audit.

The TPM2B_PUBLIC (inPublic) encompasses all the necessary fields to define the properties of the newly created object. The parentHandle parameter must reference a loaded decryption key that has both the public and sensitive area loaded.

During the object definition, the caller supplies a structure for the object within a TPM2B_PUBLIC structure (inPublic), an initial value for the authValue field

(inSensitive.userAuth), and, if the object is symmetric, an optional initial data value (inSensitive.data). The TPM validates this parameter based on the Creation rules in TPMA_OBJECT as specified.

The inSensitive parameter can be encrypted using parameter encryption. When a value is indicated as being TPM-generated, it is populated with bits from the Random Number Generator for TPM2_Create() or with values from the Key Derivation Function for TPM2_CreatePrimary().

The TPM performs specific operations based on the type of object being created. Lets examine the three cases:

- **Symmetric key**:
  - If the input data, inSensitive.sensitive.data, is an EmptyBuffer, a TPM-generated key value is generated and stored in the new object's TPM_SENSITIVE.sensitive.sym field. The size of the key is determined by the parameters specified in inPublic.publicArea .
  - If the input data, inSensitive.sensitive.data, is not an EmptyBuffer, the TPM validates that its size does not exceed the key size specified in the inPublic template. If the condition is met, the input data is copied to TPMT_SENSITIVE.sensitive.sym of the new object.
  - A TPM-generated obfuscation value, TPMT_SENSITIVE.sensitive.seedValue, is generated. Its size is determined by the digest computed by the nameAlg specified in inPublic. This value prevents leakage of sensitive information through the public unique value.
  - The TPMT_PUBLIC.unique.sym value for the new object is computed by hashing the key and obfuscation values in TPMT_SENSITIVE with the nameAlg of the object, this can be seen in the following equation 3.1.
- **Asymmetric key**:
  - If inSensitive.sensitive.data is not an EmptyBuffer, the TPM returns TPM_RC_VALUE, indicating that a parameter does not have one of its allowed values.
  - A TPM-generated private key value is created with a size determined by the parameters in inPublic.publicArea.
  - If the key is a Storage Key, a TPM-generated TPM_SENSITIVE.seedValue is created; otherwise, TPMT_SENSITIVE.seedValue.size is set to zero. An object that is not a storage key does not require a symmetric key as it has no child object to encrypt.
  - The public unique value is computed from the private key according to the methods specific to the key type.
  - If the key is an ECC key and the required scheme defined by curveId differs from the scheme in the public area of the template, the TPM returns TPM_RC_SCHEME.

  - If the key is an ECC key and the required KDF defined by `curveId` differs from the KDF in the public area of the template, the TPM returns `TPM_RC_KDF`. The caller cannot specify the KDF to be used with an ECC decryption key in the current implementation. Hence, the KDF parameter in the template must be set to `TPM_ALG_NULL`.

- **Keyed hash**:

  - If `inSensitive.sensitive.data` is an `EmptyBuffer` and both the sign and decrypt attributes in `inPublic` are `CLEAR`, the TPM returns `TPM_RC_ATTRIBUTES`. This indicates a data object with no actual data.
  - If both sign and decrypt are either `CLEAR` or `SET`, and the scheme in the public area of the template is not `TPM_ALG_NULL`, the TPM returns `TPM_RC_SCHEME`.
  - If `inSensitive.sensitive.data` is not an `EmptyBuffer`, it is copied to `TPMT_SENSITIVE.sensitive.bits` of the new object.
  - If `inSensitive.sensitive.data` is an `EmptyBuffer`, a TPM-generated key value, with a size equal to the digest produced by the `nameAlg` specified in `inPublic`, is stored in `TPMT_SENSITIVE.sensitive.bits` .
  - A TPM-generated obfuscation value, with a size equal to the digest produced by the `nameAlg` specified in `inPublic`, is stored in `TPMT_SENSITIVE.seedValue` .
  - The `TPMT_PUBLIC.unique.keyedHash` value for the new object is generated using the same equation as mentioned before.

$$unique := H_{nameAlg}(key\ (\ TPM{-}generated\ or\ provided\ by\ the\ caller\ )\ ||\ obfuscation.values\ ) \tag{3.1}$$

For `TPM2_Load()`, the TPM applies standard symmetric protections to the `TPMT_SENSITIVE` to create `outPublic`. The encryption key is derived from the symmetric seed in the sensitive area of the parent.

The ESAPI implementation of `TPM2_Create()` provides three functions: `Esys_Create()`, `Esys_Create_Async()`, and `Esys_Create_Finish()`, following the general template.

CreatePrimary

The `TPM2_CreatePrimary()` command is utilised to create a Primary Object under one of the Primary Seeds or a Temporary Object under `TPM_RH_NULL`. This command takes a `TPM2B_PUBLIC` as a template for the object being created. The size of the unique field is not validated for consistency with other object parameters. The

command creates and loads a Primary Object, but the sensitive area is not returned. As a result, the key cannot be reloaded and must either be made persistent or recreated.

For interoperability purposes, it is recommended that the unique field does not exceed the size allowed by the object parameters. This ensures that unmarshalling does not fail. An empty buffer can be used as the value for the unique field.

The `TPM2_CreatePrimary()` command can create any type of object with any combination of attributes allowed by `TPM2_Create()`. The constraints on templates and parameters are the same, except that a Primary Storage Key and a Temporary Storage Key are not required to use the algorithms of their parents.

When setting the attributes of the created object, `fixedParent`, `fixedTPM`, `decrypt`, and `restricted` are implied to be set in the parent (a `Permanent Handle`). The remaining attributes are implied to be `CLEAR`.

The TPM derives the object from the Primary Seed indicated in `primaryHandle` using an approved Key Derivation Function (KDF).

If this command is called multiple times with the same `inPublic` parameter, `inSensitive.data`, and `Primary Seed`, the TPM will produce the same `Primary Object`. However, if the Primary Seed is changed, the generated Primary Objects with the new seed will be statistically unique, even if the call parameters remain the same.

Authorisation is required for this command. For a Primary Object attached to the Platform Primary Seed, authorisation can be provided by `platformAuth` or `platformPolicy`. For a Primary Key attached to the Endorsement Primary Seed, authorisation can be provided by `endorsementAuth` or `endorsementPolicy`.

In the ESAPI implementation, these actions are performed using the functions `Esys_CreatePrimary()`, `Esys_CreatePrimary_Async()`, and `Esys_CreatePrimary_Finish()` .

Generation Primary Object

During the key generation, two types of keys are produced. The first type is an ordinary key, which is generated by using the Random Number Generator to seed the computation. The resulting secret key value is stored in a secure location. The second type is a Primary Key, which is derived from a seed value rather than directly from the RNG. Typically, the seed is generated by the RNG and persistently stored on the TPM. The generation of a Primary Key from a seed is based on the use of an approved Key Derivation Function, with the specification widely employing the KDF defined in `SP800-108`.

The TPM utilises two different schemes for the Key Derivation Function, depending on whether it is used for Elliptic Curve Diffie-Hellman (ECDH) or for other purposes. The ECDH KDF follows the specifications outlined in `SP800-56A`. On the other hand, the Counter mode KDF, referred to as `KDFa`, is used for all other instances and employs HMAC as the pseudo-random function, as described in `SP800-108`.

CreateLoaded

The TPM2_CreateLoaded() command performs two actions: it creates an object and then loads it into the TPM. The type of object created (Primary, Ordinary, or Derived) depends on the type of parentHandle specified. If parentHandle is associated with a Primary Seed, a Primary Object is created. If the parentHandle is linked to a Storage Parent, an Ordinary Object is created. And if the parentHandle is connected to a Derivation parent, a Derived Object is generated. The input validation process is the same as TPM2_Create() and TPM2_CreatePrimary(), except when parentHandle references a DerivationParent. In that case, sensitiveDataOrigin in inPublic must be set to CLEAR.

When the parentHandle refers to a DerivationParent or a Primary Seed, the outPrivate will be an EmptyBuffer. Unlike TPM2_Create() and TPM2_CreatePrimary(), this command does not provide creation data as a return value. If creation data is required, the previous commands should be used.

In the ESAPI implementation, these actions are performed using the functions Esys_CreateLoaded(), Esys_CreateLoaded_Async(), and Esys_CreateLoaded_Finish().

Load

The TPM2_Load() command is utilised to load objects into the TPM. This command is specifically used when both the TPM2B_PUBLIC and the TPM2B_PRIVATE components need to be loaded. If only the TPM2B_PUBLIC component is to be loaded, the TPM2_LoadExternal() command is used instead. It is important to note that loading an object is different from restoring a saved object context.

When using the TPM2_Load() command, the TPMA_OBJECT attributes of the objects are checked based on the specifications'rules. If the object is not a keyedHash object and both the sign and encrypt attributes are not set, the TPM will return TPM_RC_ATTRIBUTES.

Objects that are loaded using this command will have a Name, which is determined by concatenating the nameAlg and the digest of the public area. The nameAlg is a parameter found in the public area of the inPublic structure.

If the inPrivate.size is zero, the load operation will fail. Following that, the inPrivate.buffer is decrypted using the parent's symmetric key. The integrity value is checked before the sensitive area is used or unmarshalled. Checking the integrity before utilising the data helps prevent attacks on the sensitive area by analysing response code differences caused by fuzzing the data.

The command returns a handle for the loaded object and the Name that the TPM computed for inPublic.public. The returned handle is associated with the object until it is flushed.

In the ESAPI implementation, the TPM2_Load() command provides three functions following a general template: Esys_Load(), Esys_Load_Async(), and Esys_Load_Finish().

PCR Operations

PCR Initialisation

When a TPM is reset or restarted, the PCRs (Platform Configuration Registers) are set back to their default initial state. However, certain PCRs may be designated to preserve certain values during a TPM resume operation. These preserved PCRs are restored to the state they had at the last `TPM2_Shutdown(state)` operation. Any PCRs that are not designated as preserved are restored to their default initial state when the `TPM2_Startup()` operation successfully concludes.

In addition to TPM resets and restarts, a PCR can also be reset using `TPM2_PCR_Reset()` or by a Dynamic Root of Trust, if allowed by its attributes. The attributes of PCRs are specified in the platform-specific specification. These attributes determine the reset value of a PCR and the localities required to perform the reset. For all PCRs except for `PCR[0]`, the default initial condition is either all bits cleared or all bits set.

PCR Integrity Collection

In TPM 1.2, an Event was hashed using the `SHA1` algorithm, and the resulting digest was extended to a PCR using the `TPM_Extend()` function. However, TPM 2.0 introduced some enhancements. It allows the use of multiple PCRs at a specific Index, with each PCR able to use a different hash algorithm. Furthermore, instead of requiring external software to generate multiple hashes of the Event, TPM 2.0 enables the Event data to be sent directly to the TPM for hashing. This ensures that the chosen hash algorithms for the PCRs are accurately reflected, even if the calling software cannot implement the hash algorithm itself.

Changing the value of a PCR requires authorisation, which can be in the form of an authorisation value or an authorisation policy. The determination of which PCRs can be controlled by a policy is specified in the platform-specific specifications. All other PCRs are controlled by authorisation using an authorisation value.

If a PCR is associated with a policy, the algorithm ID of that policy determines whether the policy should be applied. If the algorithm ID is not `TPM_ALG_NULL`, then the policy digest associated with the PCR must match the `policyDigest` in a policy session. However, if the algorithm ID is `TPM_ALG_NULL`, it means that no policy is present, and authorisation can be granted with an `EmptyAuth`.

If a platform-specific specification indicates that PCRs are grouped, it means that all the PCRs in that group share the same authorisation policy or authorisation value.

PCR Extend

The `TPM2_PCR_Extend()` command is utilised to update a specific PCR. The digests parameter contains one or more tagged digest values, each identified by an algorithm ID. Each digest value is then extended into the PCR associated with the `pcrHandle`, in the corresponding bank specified by the tag `hashAlg` such as `SHA-1` or `SHA-256`.

For each entry in the list, the TPM checks if the `pcrNum` is implemented for that algorithm. If it is, the following operation is performed 3.2:

$$PCR.digest[\ pcrNum\ ][\ hashAlg\ ] := H_{hashAlg}(\ PCR.digest_{old}[\ pcrNum\ ][\ hashAlg\ ]\ \ ||\ \ data[\ hashAlg\ ].buffer\ ) \tag{3.2}$$

Here, $H_{hashAlg}$ represents the hash function using the `hash algorithm` associated with the PCR instance, `PCR.digest` refers to the digest value in a PCR, `pcrNum` is the numeric selector for the PCR (`pcrHandle`), `hashAlg` is the algorithm selector for the digest, and `data[alg].buffer` represents the bank-specific data to be extended.

If no digest value is specified for a particular bank, the corresponding PCR in that bank remains unchanged. If a digest value is specified but the selected bank is not implemented, the digest value is not utilised. For instance, if the caller includes digests for algorithms that are not implemented, the TPM will fail the call, resulting in a `TPM_RC_HASH` error.

The `pcrHandle` parameter is allowed to reference `TPM_RH_NULL`. In such cases, the input parameters are processed, but the TPM does not take any action. The result of the command in this case is the same as `TPM2_GetCapability()`.

PCR Event

The `TPM2_PCR_Event()` command is used to update a specified PCR. The `eventData` is hashed using the hash algorithm associated with each bank where the indicated PCR has been allocated. After hashing the data, the resulting digests list is returned. If the `pcrHandle` references an implemented PCR (not `TPM_RH_NULL`), the digests list is processed similarly to `TPM2_PCR_Extend()`.

A TPM must support an `Event.size` ranging from zero to 1024. An `Event.size` of zero indicates that there is no actual data, but the indicated operations will still take place.

Upon successful completion of the command, the digests list will contain the tagged digests of the `eventData` that were computed to prepare for extending the data into the PCR. The list may include a digest for each bank, or it may only contain a digest for each bank where the `pcrHandle` exists. If the `pcrHandle` is `TPM_RH_NULL`, the TPM may return either an empty list or a digest for each bank.

PCR Read

The `TPM2_PCR_Read()` command is used to retrieve the current values of the specified PCR. The TPM processes the list of `TPMS_PCR_SELECTION` in `pcrSelectionIn` sequentially. Within each `TPMS_PCR_SELECTION`, the TPM examines the bits in the `pcrSelect` array, following the ascending order of the PCR numbers. If a bit is `SET` and the corresponding PCR is present, the TPM includes the digest of that PCR in the list of values returned in `pcrValues`. The TPM continues processing bits until all of them have been processed or until adding additional values to `pcrValues`

would exceed the capacity of the output buffer. If none of the selected PCR are implemented, the returned list may be empty. Reading a PCR does not require any authorisation, and any implemented PCR can be read from any locality.

`TPM2_PCR_Read()` enables the selection of multiple PCR across different banks. Other commands that allow the caller to choose PCR from multiple banks are `TPM2_Quote()` and `TPM2_PolicyPCR()`. When a command supports the selection of multiple PCR, a list of selectors is used. Each entry in the list includes an algorithm ID followed by a bit array. Each bit in the array corresponds to a specific PCR. If a bit is `SET`, it indicates that the corresponding PCR in the bank associated with the algorithm ID is selected. The correspondence between the bits and PCR is such that the bit corresponding to `PCR[n]` is the **(n mod 8) bit in the [n/8] octet of the array**.

PCR Reset

The `TPM2_PCR_Reset()` command is used to reset the current value of a PCR, provided that proper authorisation is provided and the PCR's attributes, as defined in the platform-specific specification, allow for the reset operation. The attributes of the PCR may impose restrictions on the locality from which the reset operation can be performed. If the `pcrHandle` references a PCR that is not allowed to be reset, the TPM will return `TPM_RC_LOCALITY`.

PCR Policy

The `TPM2_PolicyPCR()` command is utilised to apply a policy conditionally based on the state of PCR. Together with `TPM2_PolicyOR()`, this command allows for different sets of authorisations to be applied depending on the state of the PCR. The TPM will modify the PCRs parameter by clearing the bits that correspond to unimplemented PCR. If the `policySession` is not a trial policy session, the TPM will use the modified PCRs value to select PCR values for hashing. The hash algorithm of the policy session is used to compute a digest (`digestTPM`) of the selected PCR. If `pcrDigest` has a non-zero length, it is compared to `digestTPM`. If the values do not match, the TPM will return `TPM_RC_VALUE` without making any changes to `policySession.policyDigest`. If the values match or if `pcrDigest` has a length of zero, `policySession.policyDigest` is extended as follows 3.3:

$$policyDigest_{new} := H_{policyAlg}(\ policyDigest_{old}\ ||\ TPM\_CC\_PolicyPCR\ ||\ pcrs\ ||\ digestTPM\ ) \tag{3.3}$$

The `pcrs` parameter corresponds to a bit representation of the PCRs implemented, using the `0` to mark those PCRs not implemented. The `digestTPM` corresponds to the hash of the selected PCRs using the hash algorithm of the policy session.

If the caller provides the expected PCR value, the policy evaluation will stop at that point if the PCR values do not match. If the caller does not provide the

expected PCR value, the validity of the settings will not be determined until an attempt is made to use the policy for authorisation. If the policy session is used for authorisation and the PCR values are known to be incorrect, the TPM will return `TPM_RC_PCR_CHANGED`.

The TPM utilises the `pcrUpdateCounter` parameter, which is incremented each time PCR are updated unless the PCR being changed is specified not to cause a change to this parameter. The value of this counter is stored in the policy session context when this command is executed. When the policy is used for authorisation, the current value of the counter is compared to the value in the policy session context, and the authorisation will fail if the values are different.

Since the `pcrUpdateCounter` is updated whenever any PCR is extended (except for those ignored), this means that the command will fail even if a PCR not specified in the policy is updated.

If this command is used for a trial policy session, the `policyDigest` will be updated using the values provided in the command rather than the values from a digest of the TPM PCR. If the caller does not provide PCR settings (`pcrDigest` has a digest of zero length), the TPM may use the current TPM PCR settings (`digestTPM`) in the calculation for the new `policyDigest`.

Quote

The `TPM2_Quote()` command is employed to obtain a quote of PCR values. The TPM will generate a hash of the selected PCR values, based on the hash algorithm specified in the chosen signing scheme. If the selected signing scheme or the scheme's hash algorithm is `TPM_ALG_NULL`, the TPM will return `TPM_RC_SCHEME`. The digest is computed as the hash of the combined digest values of the selected PCR.

Start Authorisation Session

The command to start an authorisation session is made of 3 functions according to the general function template previously described. Those are used to start an authorisation session using alternative methods of establishing the session key (`sessionKey`).
The `sesssionKey` is then used to derive values used for authorisation and for encryption parameters. This command allows to inject a secret into the TPM using either asymmetric or symmetric encryption.
The type of `tpmKey` determines how the value in `encryptedSalt` is encrypted. The decrypted secret value is used to compute the `sessionKey`. If `tpmKey` is `TPM_RH_NULL`, then `encryptedSalt` is required to be an `EmptyBuffer`. The TPM generates the `sessionKey` from the recovered secret value.
No authorisation is required for `tpmKey` or bind. This is possible because the result of using the key is not available to the caller, except indirectly through the `sessionKey`. This does not represent a point of attack because if the caller attempts to use the session without knowing the `sessionKey` value, the effect is an authorisation failure and it will trigger the dictionary attack logic.
The entity referenced with the bind parameter contributes an authorisation value

to the sessionKey generation process. If both tpmKey and bind are TPM_RH_NULL, the sessionKey is set to the EmptyBuffer. If tpmKey is not TPM_RH_NULL then the *ecnryptedSalt* is used in the computation of the session key. Instead if the bind is not TPM_RH_NULL the authValue of bind is used in the sessionKey computation. If symmetric is specified the TPM_ALG_CFB is the only allowed value for the mode field in the symmetric parameter (TPM_RC_MODE).

Depending on the value of the tpmKey, two situations can happen:

- NULL: no special action needs to be performed, since no salt value is transferred to the TPM;
- non-NULL:
  - ESAPI checks if the tpmKey is suitable for encrypting salts. If not it returns TSS2_ESYS_RC_BAD_TR;
  - ESAPI generates a salt value using the internal RNG implementation, which is then encrypted using the tpmKey public key. The encrypted salt is transferred to the TPM and the plain salt value is used for session key calculation during.

On completion of this function the result is different:

- Successful TPM response:
  1. creation of an Esys_TR object for sessionHandle of type session;
  2. setting sessionType in Esys_TR sessionHandle to the esysContext sessionType;
  3. setting authHash in Esys_TR sessionHandle to the esysContext authHash, unencryptedtSalt, the bindAuthValue, the nonceTPM and nonceCaller;
  4. storing the sessionKey in the Esys_TR sessionHandle;
  5. storing the response parameter tpmNonce in Esys_TR sessionHandle;
  6. setting bindName in Esys_TR sessionHandle to the esysContext bindName if provided during Esys_StartAuthSession_Async();
- Unsuccessful TPM response: deletion of values saved to esysContext (e.g., unencryptedSalt, sessionType, symmetric, authHash, bindName, bindAuth, nonceCaller).

ESAPI implementation includes functions such as Esys_StartAuthSession(), Esys_StartAuthSession_Async(), and Esys_StartAuthSession_Finish().

Unseal

The TPM2_Unseal() command retrieves the data stored within a loaded **Sealed Data Object**. A Sealed Data Object, generated by the TPM using TPM2_Create() or TPM2_CreatePrimary(), can be randomly created based on a template. In TPM

1.2, PCR authorisation is hard coded, while in TPM 2.0, PCR authorisation requires a policy to be implemented. The returned value may be encrypted using encryption provided by an authorisation session. ESAPI implementation includes functions such as `Esys_Unseal()`, `Esys_Unseal_Async()`, and `Esys_Unseal_Finish()`.

ReadPublic

The `TPM2_ReadPublic()` command provides access to the public area of a loaded object. Authorisation is not required to use the *objectHandle* parameter. As the caller may not be aware of the public area associated with the `objectHandle`, it is not feasible to include the Name associated with `objectHandle` in the hash computation. ESAPI implementation includes functions such as `Esys_ReadPublic()`, `Esys_ReadPublic_Async()`, and `Esys_ReadPublic_Finish()`.

Chapter 4

Sealing and possible usages

In the context of Trusted Platform Module (TPM) technology, *Sealing* refers to a cryptographic process of binding sensitive data to a specific state of the platform and enforcing a set of conditions under which the data can be accessed. The process ensures the confidentiality, integrity, and controlled access to sensitive data.

During the thesis work, two potential solutions were identified. The first solution involved creating an RSA key using the TPM and then using it in an Nginx server by appropriately modifying the `tpm2-openssl` library. On the other hand, the second solution fully utilised the concept of Sealing, particularly the commands provided by the `tss2-esys` library. The last solution will be extensively discussed in the next chapter as it is the official solution.

To implement the first solution, we first explored various uses of a `Sealed RSA key` and in this chapter, we are going to analyse these useful scenarios on how to `Encrypt/Decrypt` and `Sign` some data using a Sealed key created using the TPM.

4.1 Overview

Sealing involves using the TPM to bind a specific piece of data, often referred to as *Sealed Data*, to the current state of the TPM and the platform it resides on. While the specifics might vary slightly based on the TPM manufacturer and the software environment, here's a general overview of how sealing can be performed:

1. **Data Preparation**: In this stage, the data that needs to be protected is identified and prepared for sealing. This data could be a cryptographic key, password, or sensitive information requiring secure storage. The data is usually in plain text form at this point and it is ready to be bound to the TPM's state.

2. **TPM Initialisation**: Ensure that the TPM is properly initialised and activated on the platform. This involves:

   (a) *Power-On Self-Test (POST)*: When a computer system is powered on or reset, the TPM goes through a POST phase. During this phase, the

TPM checks its internal components and functionality to ensure that it is operational.

(b) *Clearing and Enabling*: The TPM might need to be cleared of any previous data, settings, or keys from its memory to ensure a clean slate.

(c) *Initialisation of Key Hierarchies*: The TPM supports various key hierarchies that are used to create and manage cryptographic keys. During initialisation, the primary root keys for these hierarchies are created.

(d) *Configuration and Seeding*: The TPM's internal settings and parameters are configured during this phase. This might include setting security policies, enabling specific features, and configuring cryptographic algorithms. Additionally, the TPM's random number generator might be seeded with additional entropy to enhance the randomness of its generated keys.

3. **Policy Definition**: The policy essentially dictates the requirements that must be met to allow unsealing of the data. The policy is a set of rules or conditions that the owner of the sealed data defines. These rules are established at the time of sealing the data. The policy could involve various aspects of the system's state, the presence of certain measurements, or even the time that has passed since the sealing occurred.

4. **TPM Binding**: The binding process involves the following steps:

   (a) *State Measurement*: The TPM measures the current state of the platform, which includes various hardware and software components. These measurements create a unique representation of the platform's configuration, which can be thought of as a digital fingerprint of the system's current status. The state measurement captures information like the BIOS version, hardware components, firmware details, and more.

   (b) *Creating Sealed Object*: The TPM generates a sealed object by combining the following:

   - the measured state of the platform;
   - the sensitive data that needs to be protected (the data prepared in the first stage);
   - a set of policy settings that define when the sealed data can be unsealed.

   (c) *Sealing Process*: The sealed object is created by applying cryptographic operations that incorporate the measured state, the sensitive data, and the policy settings. This process essentially binds the sensitive data to the measured state of the platform and the specified policies. As a result, the sealed data becomes inaccessible unless the platform is in the exact same state as the one it was in when the data was sealed.

5. **Store Sealed Object**: Store the sealed object within the TPM's secure storage area. This storage is isolated from the rest of the system and is protected from unauthorised access.

6. **Unsealing**: When the sealed data needs to be accessed, the unsealing process occurs:

   (a) *State Verification*: Before attempting to unseal the data, the TPM measures the current state of the platform again. This new measurement is compared to the state that was recorded during the sealing process. If the states match, it indicates that the platform is in a legitimate and trusted state.

   (b) *Unsealing Process*: If the state verification is successful, the TPM uses its internal cryptographic functions to verify the integrity of the sealed object. If the integrity is confirmed, the TPM decrypts the sealed data, making it available for use by authorised applications or processes.

   (c) *Policy Enforcement*: During the unsealing process, the TPM enforces the defined policy settings (if defined during the Sealing process). This might involve additional authentication checks, validation of system components, or other conditions specified in the policies.

Overall, the sealing process combines cryptographic principles, hardware-based measurements, and policy enforcement to create a secure environment for sensitive data. This process ensures that the data remains confidential, its integrity is maintained, and its access is controlled based on the platform's state. Sealed data can only be accessed in a trusted environment, preventing unauthorised access even if an attacker has physical access to the storage medium. The process enhances the overall security posture of systems and applications that rely on TPM technology.

4.2 Analysis

The implementation was carried out starting from the `tpm2-tools` library, proceeding to analyse the code related to terminal commands, and subsequently consulting the manuals for their proper usage. The study was initiated by selecting the necessary commands and examining them individually.

The first aspect to focus on was **Unsealing**, which allowed us to follow a reverse path, starting from the end result and progressing back to the starting point through reverse engineering. This command allows for the extraction of a data blob from an object stored in the TPM. The analysis of this function was crucial to understand which further actions were required and in what sequence to execute them. These included creating a primary key, initialising a session, and creating a policy based on Platform Configuration Registers.

In particular, by examining the `PolicyPCR` command, a detailed guide was obtained on how to use a `Session Policy` to create a sealed data blob. The following Figure represents a modified sequence of commands taken from the `tpm2-tools` manual 4.1.

In this example 4.1, it is necessary to make the appropriate changes because in our case, it is not required to create a data blob by passing data but rather to create a Sealed key that depends on the state of the PCRs. This can be done by

```
$ tpm2_createprimary -C e -g sha256 -G rsa -c primary.ctx
$ tpm2_startauthsession -S session.dat
$ tpm2_policypcr -S session.dat -l "sha256:23" -L policy.dat
$ tpm2_flushcontext session.dat
$ tpm2_create -u key.pub -r key.priv -C primary.ctx -L
   policy.dat -i- <<< "data␣blob"
$ tpm2_load -C primary.ctx -u key.pub -r key.priv -n
   unseal.key.name -c unseal.key.ctx
$ tpm2_startauthsession --policy-session -S session.dat
$ tpm2_policypcr -S session.dat -l "sha256:23" -L policy.dat
$ tpm2_unseal -p session:session.dat -c unseal.key.ctx -o
   unsealed.dat
$ tpm2_flushcontext session.dat
```

Figure 4.1. Creation of a Sealed Object

changing the previous `tpm2_create` command in the following way represented in the Figure 4.2.

```
$ tpm2_create -Q -u key.pub -r key.priv -C primary.ctx -L
   policy.dat
```

Figure 4.2. tpm2_create command changed

By modifying the command in this way, it is possible to create a key that depends on the values of the PCRs. To test its proper functionality, which means it should only be used when the state of the PCRs matches the state defined during its creation, two implementations were made: one regarding the Encryption and Decryption operation 4.3 and the other one regarding the Signing operation.

To better understand the commands in the previous Figures 4.1 4.3, here is a small description of what each command does and the options used:

- `tpm2_createprimary`: this command is used to create and load inside the TPM a primary key under one of the hierarchies: Owner, Platform, Endorsement, NULL. Options:
  - `C`: the hierarchy under which the object is created;
  - `g`: the hash algorithm to use for generating the object name;
  - `G`: the algorithm type for the generated primary key;
  - `c`: file path to save the object context of the generated primary key.
- `tpm2_startauthsession`: starts a session with the TPM and saves the policy session data to a file, so it can then be used for authorisation or policy events. Options:
  - `S`: the name of the policy session file, required;

```
$ tpm2_createprimary -C e -g sha256 -G rsa -c primary.ctx $
$ tpm2_startauthsession -S session.dat
$ tpm2_policypcr -S session.dat -l "sha256:23" -L policy.dat
$ tpm2_flushcontext session.dat
$ tpm2_create -u key.pub -r key.priv -C primary.ctx -L policy.dat
$ tpm2_load -C primary.ctx -u key.pub -r key.priv -n
   unseal.key.name -c unseal.key.ctx
$ tpm2_rsaencrypt -c key.ctx -o msg.enc msg.dat
$ tpm2_startauthsession --policy-session -S session.dat
$ tpm2_policypcr -S session.dat -l "sha256:23" -L policy.dat
$ tpm2_rsadecrypt -c key.ctx -p session:session.dat -o msg.ptext
   msg.enc
$ tpm2_flushcontext session.dat
```

Figure 4.3. Encryption and Decryption using a Sealed RSA Key

  - `policy-session`: to start a policy session of type `TPM_SE_POLICY`.

- `tpm2_policypcr`: creates a PCR policy event using the TPM, establishing a policy linked to specific PCR values.
  - `S`: policy session file generated by `tpm2_startauthsession`;
  - `l`: the list of selected PCRs;
  - `L`: file to save the policy digest.

- `tpm2_flushcontext`: remove a designated handle or all contexts linked to a transient object, loaded session, or saved session from the TPM.

- `tpm2_create`: create a child object that can either be a key or a sealing object. A sealing object allows sealing user data to the TPM, with a maximum size of 128 bytes. Options:
  - `u`: the output file which contains the public portion of the object;
  - `r`: the output file which contains the sensitive portion of the object;
  - `C`: the parent handle of the object to be created;
  - `L`: the input policy file or a hex string, optional;
  - `i`: the data file to be sealed.

- `tpm2_load`: load both the private and public portions of an object into the TPM and returns the object context. Options:
  - `C`: the handle of the parent key;
  - `u`: the file containing the public portion of the object;
  - `r`: the file containing the sensitive portion of the object;
  - `n`: the output file to store the name structure of the object;
  - `c`: the file name of the saved object context, required.

- `tpm2_rsaencrypt`: executes RSA encryption on the input data. Options:
  - `c`: the context to the key to be used;
  - `o`: the output file containing the encrypted data.
- `tpm2_rsadecrypt`: performes RSA decryption on the encrypted data received in input. Options:
  - `c`: the context to the key to be used;
  - `p`: the file containing the session to use for authorisation;
  - `o`: the output file containing the decrypted data.
- `tpm2_unseal`:
  - `p`: the file containing the session to use for authorisation;
  - `c`: the context to the key to be used;
  - `o`: the output file containing the sensitive data unsealed.

The initial issues arose when attempting Encryption and Decryption operations. This was because when calling the `Esys_RSAEncrypt()` function, it returned the error `Esys error 0x00090006`, which using the integrated decoder for errors it returned `mu: A buffer isn't large enough`. Since the error seemed to be related to marshalling and unmarshalling due to incorrect data being passed, to better understand the problem and try to solve it `LTRACE` was used.

4.2.1 LTRACE

LTRACE is a debugging and diagnostic tool for Linux-based operating systems. It is used to intercept and trace the library calls made by a running process. This tool is particularly useful for understanding how a program interacts with shared libraries, as well as for diagnosing and debugging issues in software.

Here are the ltrace key points:

- **Interception**: When ltrace is executed followed by a command or the name of a program, ltrace attaches to the specified process or program and monitors its library calls.
- **Output**: ltrace then logs the library calls made by the process, along with the arguments passed to those calls, to the standard output or a file, depending on how it was configured.
- **Analysis**: This output can be analysed to understand how the program is using the libraries, which can be helpful in diagnosing issues, profiling performance, or gaining insights into the program's behaviour.

```
$ ltrace -x 'Esys*' -L <<command to execute>>
```

Figure 4.4. General LTRACE command executed for tests

```
$ ltrace -x 'Esys*' -L tpm2 create -C primary.ctx -u key.pub -r
   key.priv -L policy.dat
```

Figure 4.5. Test `tpm2_create` command with authentication

For example, if a program is misbehaving or it is crashing, ltrace can be used to identify which library calls it is making just before the problem occurs, which might lead to the root cause of the issue. It is a valuable tool in a developer's toolkit for troubleshooting and understanding program behaviour at the library call level.

In our case, it was used to analyse each command starting from the creation of the primary key up to the decryption of the input data.

In the command represented in the Figure 4.4, the `-x` option is used to define the filter, which in this case includes every function that starts with Esys. This is because the command-line TOOL relies entirely on function calls using ESAPI. In addition, the `-L` option allows us to trace all library calls made not only by the main process but also by child processes.

Two different scenarios were considered to observe the behaviour of the commands and functions called, in order to identify the cause of the error. In both cases, we took the example of the Encryption and Decryption of a file through the creation of an RSA key, as previously shown in the Figure 4.3. The two test cases are:

- First case: the RSA key is sealed, meaning it is dependent on the values of the PCRs and requires the definition of a session policy;
- Second case: the RSA key is not dependent on sessions or permissions.

In particular, the following commands were analysed: `tpm2_create`, `tpm2_load`, `tpm2_rsaencrypt` and `tpm2_rsadecrypt`. Additionally, for the first case, the `tpm2_policypcr` command was also examined.

Create

The command to test the creation of a key with authentication is represented in the following Figure 4.5.

The command to test the creation of a key without authentication is represented in the following Figure 4.6.

The first command we will analyse and compare the results is the `tpm2_create` command. In both situations, the same function calls were obtained, with the only

```
$ ltrace -x 'Esys*' -L tpm2 create -C primary.ctx -u key.pub
   -r key.priv
```

Figure 4.6. Test `tpm2_create` command without authentication

```
Esys_Initialize@libtss2-esys.so.0(0x7fffb1f308c8,
   0x565134213e90, 0, 59) = 0
<...>
Esys_StartAuthSession@libtss2-esys.so.0(0x5651342145e0, 4095,
   4095, 4095 <unfinished ...>
Esys_StartAuthSession_Async@libtss2-esys.so.0(0x5651342145e0,
   4095, 4095, 4095) = 0
Esys_StartAuthSession_Finish@libtss2-esys.so.0(0x5651342145e0,
   0x565134217078, 0x7601, 0x3b0000000180) = 0
<... Esys_StartAuthSession resumed> ) = 0
<...>
Esys_TRSess_GetAttributes@libtss2-esys.so.0(0x5651342145e0,
   0x40418487, 0x7fffb1f30807, 0) = 0
Esys_TR_SetAuth@libtss2-esys.so.0(0x5651342145e0, 0x40418488,
   0x565134216bc6, 0x7fffb1f30510) = 0
Esys_Create@libtss2-esys.so.0(0x5651342145e0, 0x40418488,
   0x40418487, 4095 <unfinished ...>
Esys_Create_Async@libtss2-esys.so.0(0x5651342145e0, 0x40418488,
   0x40418487, 4095) = 0
Esys_Create_Finish@libtss2-esys.so.0(0x5651342145e0,
   0x565133c233a0, 0x565133c23388, 0x565133c233b0) = 0
<... Esys_Create resumed> ) = 0
<...>
Esys_Finalize@libtss2-esys.so.0(0x565133c41bb0, 0x7fffb1f30810,
   1, 4) = 0
+++ exited (status 0) +++
```

Figure 4.7. Ltrace output for `tpm2_create` command with authentication

differences being the session identifiers and other parameters whose meaning in this context is meaningless. The result of the ltrace command is represented in the following Figure 4.7.

The only real difference was found when analysing the code in the `tpm2-tools` library. In the `tpm2_create.c` file, which defines how the creation of a sealed key or data blob occurs. During the input data initialisation phase, there is a call to the `tpm2_auth_util_from_optarg()` function, shown in the following Figure 4.8. This function is responsible for managing the session restore if it has been passed as input; otherwise, it creates a new session without authorisation, setting the required password for authorisation to an Empty string.

In the first case, the session in which we had defined the policy will be restored,

and this is the moment when the key created will be dependent on the PCR values. It is important to note that this function allows managing authorisation to create or use an object in four possible ways:

- *session*: in the case of using a policy session to authorise the creation or usage of an object. In this case, the file obtained from the previous call to `tpm2_startauthsession` is passed as input;
- *file*: in this case, the file contains the password that needs to be read;
- *pcr*: used to satisfy a policy based on PCRs values;
- *password*: authorisation password, by default interpreted as strings, but it is possible to also use a hex-string based password.

.

On the other hand, in the case of creating the child object without authorisation, a new session with an empty password will be created to allow the key's creation.

Load

The command executed to test both cases (with and without authorisation) is represented in the following Figure 4.9.

The Load function also yields the same result in both cases, as in the previous case of the Create function, this can be seen in the following Figure 4.10. This is because, once again, the differentiating factor is the Session Restore present in the authorisation scenario, while the creation of a new session occurs in the second case.

In this case as well, it is possible to verify what was stated by analysing the `tpm2_load.c` file, where, during the call to the function that manages the input data, it is possible to observe that in the `tpm2_util_object_load2.c` function 4.11, we find several calls to the previously illustrated function 4.8, which handles the Session restoration.

RSAEncrypt

The command executed to test both cases (with and without authorisation) is represented in the following Figure 4.12.

The other command analysed is the RSAEncrypt, and as for the results obtained for the two scenarios, they overlap. In fact, since no authorisation is required to encrypt data, there is no discrepancy between the first and the second case. The result of ltrace command can be seen in the following Figure 4.13.

```
1 tool_rc tpm2_auth_util_from_optarg(ESYS_CONTEXT *ectx, const char
      *password,
2         tpm2_session **session, bool is_restricted) {
3     password = password ? password : "";
4     /* starts with session: */
5     bool is_session = !strncmp(password, SESSION_PREFIX,
     SESSION_PREFIX_LEN);
6     if (is_session) {
7         if (is_restricted) {
8             LOG_ERR("Cannot specify password type \"session:\"");
9             return tool_rc_general_error;
10        }
11        return handle_session(ectx, password, session);
12    }
13    /* starts with "file:" */
14    bool is_file = !strncmp(password, FILE_PREFIX,
     FILE_PREFIX_LEN);
15    if (is_file) {
16        return handle_file(ectx, password, session);
17    }
18    /* starts with pcr: */
19    bool is_pcr = !strncmp(password, PCR_PREFIX, PCR_PREFIX_LEN);
20    if (is_pcr) {
21        if (is_restricted) {
22            LOG_ERR("Cannot specify password type \"pcr:\"");
23            return tool_rc_general_error;
24        }
25        return handle_pcr(ectx, password, session);
26    }
27    /* must be a password */
28    return handle_password_session(ectx, password, session);
29 }
```

Figure 4.8. `tpm2_auth_util_from_optarg` function

```
$ ltrace -x 'Esys*' -L tpm2 load -C primary.ctx -u key.pub -r
   key.priv -c key.ctx
```

Figure 4.9. Test `tpm2_load` command

RSADecrypt

The command to test the decryption with the RSA key with authentication is represented in the following Figure 4.14.

```
Esys_Initialize@libtss2-esys.so.0(0x7ffc45708b58,
   0x555f76f6ae00, 0, 29) = 0
<...>
Esys_StartAuthSession@libtss2-esys.so.0(0x555f76f6b550, 4095,
   4095, 4095 <unfinished ...>
Esys_StartAuthSession_Async@libtss2-esys.so.0(0x555f76f6b550,
   4095, 4095, 4095) = 0
Esys_StartAuthSession_Finish@libtss2-esys.so.0(0x555f76f6b550,
   0x555f76f69858, 0x7601, 0x3b0000000180) = 0
<... Esys_StartAuthSession resumed> ) = 0
<...>
Esys_TRSess_GetAttributes@libtss2-esys.so.0(0x555f76f6b550,
   0x40418487, 0x7ffc45708a77, 0) = 0
Esys_TR_SetAuth@libtss2-esys.so.0(0x555f76f6b550, 0x40418488,
   0x555f76f63f96, 0x7ffc45708a54) = 0
Esys_Load@libtss2-esys.so.0(0x555f76f6b550, 0x40418488,
   0x40418487, 4095 <unfinished ...>
Esys_Load_Async@libtss2-esys.so.0(0x555f76f6b550, 0x40418488,
   0x40418487, 4095) = 0
Esys_Load_Finish@libtss2-esys.so.0(0x555f76f6b550,
   0x555f76bdaed0, 0x555f76bda650, 0x555f76bda630) = 0
<... Esys_Load resumed> ) = 0
Esys_TR_GetName@libtss2-esys.so.0(0x555f76f6b550, 0x40418489,
   0x7ffc45708ad8, 0x555f76bda650) = 0
name: 000b359a1ed7507f59adfb014c28f3e0cc44495a99c0b5d14
     30324ea90e4ddd3d10b
<...>
Esys_Finalize@libtss2-esys.so.0(0x555f76bd8bb0, 0x7ffc45708aa0,
   1, 4) = 0
+++ exited (status 0) +++
```

Figure 4.10. Ltrace output for `tpm2_load` command

Instead, the command to test the decryption with the RSA key without authentication is represented in the following Figure 4.15.

On the contrary, a command that invokes different functions and, therefore, does not provide the same comparable behaviour as seen in the previous cases is the RSADecrypt command.

Regarding the first case in which we used a Sealed key, thus protected by a Session Policy dependent on PCR values, the call to the `Esys_TR_SetAuth()` function is made to restore the previous session passed as a parameter. The following Figure 4.16 shows the results of the ltrace command executed for the first case.

Conversely, in the case where a non-Sealed key is used, the authorisation to access the TPM is required to decrypt the data by creating a new session with null parameters, through an empty password. This can be observed from the use of

```
1 static tool_rc tpm2_util_object_load2(ESYS_CONTEXT *ctx, const
    char *objectstr,
2         const char *auth, bool do_auth, tpm2_loaded_object *
    outobject,
3         bool is_restricted_pswd_session, tpm2_handle_flags flags)
     {
4
5     tool_rc rc = tool_rc_success;
6     if (do_auth) {
7         ESYS_CONTEXT *tmp_ctx = is_restricted_pswd_session ? NULL
     : ctx;
8         tpm2_session *s = NULL;
9         rc = tpm2_auth_util_from_optarg(tmp_ctx, auth, &s,
10                 is_restricted_pswd_session);
11        if (rc != tool_rc_success) {
12            return rc;
13        }
14        outobject->session = s;
15    }
16
17    /*...*/
18
19    return rc;
20 }
```

Figure 4.11. `tpm2_util_object_load2` function

```
$ ltrace -x 'Esys*' -L tpm2 rsaencrypt -c key.ctx -o
   file.enc file.txt
```

Figure 4.12. Test `tpm2_rsaencrypt` command

`Esys_StartAuthSession` and the subsequent call to the `Esys_TR_SetAuth` function, as depicted in the following Figure 4.17.

Policy PCR

The command to test the creation of a policy session depending to the values to the selected PCRs is represented in the following Figure 4.18.

Finally, the PolicyPCR command was analyzed, which generates a PCR policy event with the TPM 4.19. A PCR policy event establishes a policy that is associated with particular PCR values and can be employed within more extensive policies created using policyor and policyauthorize events. It is possible to define the PCR data used in the policy in one of three ways:

```
Esys_Initialize@libtss2-esys.so.0(0x7ffc3c849728,
   0x564863cfaea0, 0, 23) = 0
<...>
Esys_ReadPublic@libtss2-esys.so.0(0x564863cfb5f0, 0x40418487,
   4095, 4095 <unfinished ...>
Esys_ReadPublic_Async@libtss2-esys.so.0(0x564863cfb5f0,
   0x40418487, 4095, 4095) = 0
Esys_ReadPublic_Finish@libtss2-esys.so.0(0x564863cfb5f0,
   0x7ffc3c8496c0, 0, 0) = 0
<... Esys_ReadPublic resumed> ) = 0
Esys_Free@libtss2-esys.so.0(0x564863cfdb80, 0x564863cfdb80, 3,
   0) = 0
Esys_RSA_Encrypt@libtss2-esys.so.0(0x564863cfb5f0, 0x40418487,
   4095, 4095 <unfinished ...>
Esys_RSA_Encrypt_Async@libtss2-esys.so.0(0x564863cfb5f0,
   0x40418487, 4095, 4095) = 0
Esys_RSA_Encrypt_Finish@libtss2-esys.so.0(0x564863cfb5f0,
   0x7ffc3c8496c0, 0x56486285c878, 0x56486285c888) = 0
<... Esys_RSA_Encrypt resumed> ) = 0
<...>
Esys_Finalize@libtss2-esys.so.0(0x56486285ebb0, 0x7ffc3c849670,
   1, 4) = 0
+++ exited (status 0) +++
```

Figure 4.13. Ltrace output for `tpm2_rsaencrypt` command

```
$ ltrace -x 'Esys*' -L tpm2 rsadecrypt -c key.ctx -o
   file.dec file.enc -p session:session.dat
```

Figure 4.14. Test `tpm2_rsadecrypt` command with authentication

- using a file that contains a concatenated list of PCR values, similar to the output from tpm2_pcrread;
- requiring the TPM to read the PCR values without specifying a PCR file input;
- specifying the digest of all the PCR values directly as an argument.

In our case, it was specified that we wanted to use only PCR number 23 to create the session policy. As we can verify from the results obtained from LTRACE and through a careful analysis of the `tpm2_policypcr.c` file 4.20, it is possible to observe that a restoration of the previously created session occurs through the `tpm2_session_restore()` function, which takes the previous session in input. Subsequently, the selected PCRs provided as input are chosen, and finally, the Policy Digest is calculated, defining the Session Policy by calling the `tpm2_policy_build_pcr()` function.

```
$ ltrace -x 'Esys_' -L tpm2_rsadecrypt -c dup.ctx -o
   data.ptext data.encrypted
```

Figure 4.15. Test `tpm2_rsadecrypt` command without authentication

```
Esys_Initialize@libtss2-esys.so.0(0x7ffc4e851268,
   0x556f7ef0a3e0, 0, 27) = 0
<...>
Esys_TR_SetAuth@libtss2-esys.so.0(0x556f7ef0c410, 0x40418488,
   0x556f7ef0aa36, 0x7ffc4e851174) = 0
Esys_RSA_Decrypt@libtss2-esys.so.0(0x556f7ef0c410, 0x40418488,
   0x40418487, 4095 <unfinished ...>
Esys_RSA_Decrypt_Async@libtss2-esys.so.0(0x556f7ef0c410,
   0x40418488, 0x40418487, 4095) = 0
Esys_RSA_Decrypt_Finish@libtss2-esys.so.0(0x556f7ef0c410,
   0x556f7d41e5d0, 0x556f7d41e5b8, 0x556f7d41e368) = 0
<... Esys_RSA_Decrypt resumed> ) = 0
<...>
Esys_Finalize@libtss2-esys.so.0(0x556f7d420bb0, 0x7ffc4e8511b0,
   1, 4) = 0
+++ exited (status 0) +++
```

Figure 4.16. Ltrace output for `tpm2_rsadecrypt` with authentication

4.2.2 Analysis conclusions

Thanks to the exhaustive analysis conducted using the LTRACE tool and a thorough study of the tpm2-tools library, we were able to successfully implement the encryption and decryption examples and the signing one.

The primary issue identified was the improper usage of sessions, particularly when HMAC sessions were required in contrast to policy sessions. Initially, in the early implementation attempts, a new HMAC session was created each time authorisation was needed, without proper consideration. This led to difficulties in tracking all the sessions created, and if they were not flushed, they resulted in the 'maximum sessions error'.

Upon rectifying this misuse of HMAC sessions, we streamlined the process by creating a single HMAC session at the beginning of the code, following TPM initialisation, and reusing it as needed. Another challenge we encountered was determining the precise moments and locations to employ HMAC and policy sessions. Only after several attempts and careful analysis of the LTRACE results, we were able to identify the optimal placement and timing for their usage. Further details on this are provided in the next section.

```
Esys_Initialize@libtss2-esys.so.0(0x7fffbfe57428,
   0x55f1f5a443b0, 0, 27) = 0
<...>
Esys_StartAuthSession@libtss2-esys.so.0(0x55f1f5a44ac0, 4095,
   4095, 4095 <unfinished ...>
Esys_StartAuthSession_Async@libtss2-esys.so.0(0x55f1f5a44ac0,
   4095, 4095, 4095) = 0
Esys_StartAuthSession_Finish@libtss2-esys.so.0(0x55f1f5a44ac0,
   0x55f1f5a42f08, 0x7601, 0x3b0000000180) = 0
<... Esys_StartAuthSession resumed> ) = 0
<...>
Esys_TRSess_GetAttributes@libtss2-esys.so.0(0x55f1f5a44ac0,
   0x40418487, 0x7fffbfe57357, 0) = 0
Esys_Free@libtss2-esys.so.0(0x55f1f5a61980, 0x40418487, 0, 0) = 0
Esys_TR_SetAuth@libtss2-esys.so.0(0x55f1f5a44ac0, 0x40418488,
   0x55f1f5a470a6, 0x7fffbfe57334) = 0
Esys_RSA_Decrypt@libtss2-esys.so.0(0x55f1f5a44ac0, 0x40418488,
   0x40418487, 4095 <unfinished ...>
Esys_RSA_Decrypt_Async@libtss2-esys.so.0(0x55f1f5a44ac0,
   0x40418488, 0x40418487, 4095) = 0
Esys_RSA_Decrypt_Finish@libtss2-esys.so.0(0x55f1f5a44ac0,
   0x55f1f55785d0, 0x55f1f55785b8, 0x55f1f5578368) = 0
<... Esys_RSA_Decrypt resumed> ) = 0
<...>
Esys_Finalize@libtss2-esys.so.0(0x55f1f557abb0, 0x7fffbfe57370,
   1, 4) = 0
+++ exited (status 0) +++
```

Figure 4.17. Ltrace output for `tpm2_rsadecrypt` without authentication

```
$ ltrace -x 'Esys*' -L $ tpm2_policypcr -S session.dat -l
   "sha256:23" -L policy.dat
```

Figure 4.18. Test `tpm2_policypcr` command

4.3 Use Cases Implementation

To implement the few use cases previously introduced, we used the tests provided within the `tpm2-tss` library, which tests the basic functionalities. In addition to these tests, we also used the `tpm2-js` project, which implements some functionalities of the TPM through the browser. While in the first case, most of the examples were written using ESAPI, in the second case, it used function calls present in SAPI. Therefore, it primarily served as a foundation for understanding the overall functioning.

Unfortunately, in the various tests encountered and analysed, none of them used

```
Esys_Initialize@libtss2-esys.so.0(0x7ffff412ba08,
   0x559f30bf5ea0, 0, 23) = 0
<...>
Esys_TRSess_GetAttributes@libtss2-esys.so.0(0x559f30bf65f0,
   0x40418487, 0x7ffff412b955, 0x559f30bf4720) = 0
Esys_PCR_Read@libtss2-esys.so.0(0x559f30bf65f0, 4095, 4095, 4095
   <unfinished ...>
Esys_PCR_Read_Async@libtss2-esys.so.0(0x559f30bf65f0, 4095,
   4095, 4095) = 0
Esys_PCR_Read_Finish@libtss2-esys.so.0(0x559f30bf65f0,
   0x7ffff412b6f4, 0, 0x7ffff412b6f8) = 0
<... Esys_PCR_Read resumed> ) = 0
Esys_PolicyPCR@libtss2-esys.so.0(0x559f30bf65f0, 0x40418487,
   4095, 4095 <unfinished ...>
Esys_PolicyPCR_Async@libtss2-esys.so.0(0x559f30bf65f0,
   0x40418487, 4095, 4095) = 0
Esys_PolicyPCR_Finish@libtss2-esys.so.0(0x559f30bf65f0,
   0x559f30ba9010, 0x559f305e9ab0, 0x7ffff412b700) = 0
<... Esys_PolicyPCR resumed> ) = 0
Esys_PolicyGetDigest@libtss2-esys.so.0(0x559f30bf65f0,
   0x40418487, 4095, 4095 <unfinished ...>
Esys_PolicyGetDigest_Async@libtss2-esys.so.0(0x559f30bf65f0,
   0x40418487, 4095, 4095) = 0
Esys_PolicyGetDigest_Finish@libtss2-esys.so.0(0x559f30bf65f0,
   0x7ffff412b9b0, 0x8901, 0xe0000000180) = 0
<... Esys_PolicyGetDigest resumed> ) = 0
<...>
Esys_Finalize@libtss2-esys.so.0(0x559f305e2bb0, 0x7ffff412b950,
   1, 4) = 0
+++ exited (status 0) +++
```

Figure 4.19. Ltrace output for `tpm2_policypcr` command

a key that was dependent on PCRs. Consequently, there is no example of how to proceed with Sealing a key. On the contrary, there is an example of creating sealed data, but this is not our case. For this reason, we proceeded by analysing the functioning of various functions, as previously extensively demonstrated, as there is no definitive guide on how to proceed.

The fundamental steps to implement encryption/decryption and signing can be summarised in the following seven stages, followed by another four stages, two for encryption/decryption and two for signing.

These are the steps in common for both scenarios:

1. initialise TCTI_TABRMD context;
2. initialise ESYS context;

```
1 static tool_rc tpm2_tool_onrun(ESYS_CONTEXT *ectx,
    tpm2_option_flags flags) {
2
3     /*...*/
4
5     tool_rc rc = tpm2_session_restore(ectx, ctx.session_path,
    false,
6             &ctx.session);
7     if (rc != tool_rc_success) {
8         return rc;
9     }
10
11    rc = tpm2_policy_build_pcr(ectx, ctx.session, ctx.
    raw_pcrs_file,
12            &ctx.pcr_selection, ctx.raw_pcr_digest, &ctx.forwards
    );
13    if (rc != tool_rc_success) {
14        LOG_ERR("Could not build pcr policy");
15        return rc;
16    }
17
18    return tpm2_policy_tool_finish(ectx, ctx.session, ctx.
    policy_out_path);
19 }
```

Figure 4.20. `tpm2_tool_onrun` function

3. create a temporary HMAC session;
4. create a primary key;
5. create a session policy and extract the policy digest;
6. create the RSA key;
7. load the RSA key in the TPM.

These are the steps needed for the encryption and decryption scenario:

8. encrypt a file with the RSA key;
9. decrypt the data blob previously obtained with the RSA key.

These are the steps needed for the signing scenario:

8. signing a data blob with the RSA key;
9. verify the signature;

In the following paragraphs, we will analyse the functions created, explaining their operation, the parameters they receive in input, and their return values. Some of these functions will be of great importance as they will be extensively used for the solution proposed in the next chapter, and for this reason, they will not be discussed again and will be considered as given.

4.3.1 Initialise TCTI_TABRMD Context

The function `Init_Tcti_Tabrmd_Context()` is used to initialise a TCTI context, enabling communication with the `tpm2-abrmd`. The `tpm2-abrmd` serves as a daemon responsible for implementing the TPM access broker and resource management. This daemon employs the DBus system bus and various pipes for client communication.

Function definition:

```
void Init_Tcti_Tabrmd_Context();
```

The implemented function takes no input parameters and attempts to initialise the `tcti_context` with a size equal to the defined configuration. Since the caller must allocate the `tcti_context`, it is possible to determine the context size by calling the allocation function once with a NULL context, allowing the caller to retrieve the required size. By calling the initialisation function again and passing a non-NULL context of the obtained size, it is possible to initialise the `tcti_context` with the default parameters defined for the `tpm2-abrmd`.

The function has no return parameters because the `tcti_context` is globally allocated if created.

4.3.2 Initialise ESYS Context

Initialize an `ESYS_CONTEXT` that holds the metadata and the state information during an interaction with the TPM.

Function definition:

```
void Init_Esys_Context();
```

The function takes no input parameters because it directly accesses the `tcti_` context variable that was just created and uses the currently used `ABI version` extracted from the globally defined variable `TSS2_ABI_VERSION_CURRENT`.

It has no return parameters because once the `esys_context` is created, it is globally allocated.

4.3.3 Create HMAC Session

The first function that needs to be called to create the primary key and for all the following operations in which the HMAC Session is required is the `StartAuthSession()`.

For our purposes, a general function was created in order to create a specific session based on the parameters passed.

Function definition:

```
StartAuthSessionResult StartAuthSession(int session_type,
   bool is_symmetric, uint32_t handle);
```

The function accepts three input parameters: the type of the session; a flag to set the algorithm and key size for parameter encryption; and the session handle. The function enables the creation of three possible session types: HMAC, Policy and Trial.

Within the function, other parameters are defined to create the desired session, primarily concerning the session's attributes. By default, there are three attributes: `TPMA_SESSION_DECRYPT`, `TPMA_SESSION_ENCRYPT` and `TPMA_SESSION_CONTINUESESSION`. Another critically important parameter is the definition of the hash algorithm to be used.

Once the session is created, the defined attributes for the newly created session are set.

If the session creation is successful, what is returned is a data structure called `StartAuthSessionResult`, which consists of: the return code; the session handle for use in subsequent operations, and the `nonce_tpm`. By default, the `nonce_tpm` is passed empty at the time of session creation and is subsequently modified.

4.3.4 Create Primary Key

The function `CreatePrimary()` is responsible for creating the Primary Key. In practice, it involves creating the primary key from the seed within the specified hierarchy: `OWNER`, `PLATFORM`, `ENDORSEMENT` and `NULL`). The Primary Key is of extreme importance because it is necessary to create the child key we are interested in.

Function definition:

```
CreatePrimaryResult CreatePrimary(int hierarchy, int type,
   int restricted, int decrypt, int sign, const std::string
   &unique, const std::string &user_auth, const std::string
   &sensitive_data, const std::vector<uint8_t> &auth_policy,
    int session_handle);
```

The function accepts ten input parameters: the primary key hierarchy; the primary key type (`RSA`, `ECC`, `SYMCIPHER` and `KEYEDHASH`); three flags (restricted, decryption and signature); the unique field; the user authentication passphrase; the sensitive data; the policy digest, and the session handle.

The defined function was created to allow the passing of various parameters for maximum flexibility. In fact, for each desired key type, there is a corresponding function through which it is possible to define the key's public parameters.

If the creation is successful, a data structure called `CreatePrimaryResult` is generated, which includes various parameters. The most important ones are: the

return code; the primary key handle and depending on the type of the key created some fields to keep this information.

4.3.5 Create Session Policy and Extract Policy Digest

The function `PolicyPCR()` is necessary for creating the policy digest by specifying which PCR you want to use. This function requires the creation of a policy session before being called.

Function definition:

```
TPM2B_DIGEST* PolicyPCR(ESYS_TR pcrHandle, ESYS_TR
    sessionHandle, const std::vector<uint8_t> &digest);
```

The defined function accepts three input parameters: the identifier of the PCR (usually, up to 5 different PCRs can be defined); the policy session handle and a buffer which by default is empty but it can be used to extend the value of the PCR selected.

Within the function, the desired PCR is selected by both its numerical identifier and its PCR group (the group corresponding to SHA256 is selected by default). The return value is of type `TPM2B_DIGEST` and contains the policy digest.

4.3.6 Create RSA Key

The function `Create()` allows the creation of the RSA key that underlies our use cases. This function is similar to the function for creating the primary key, except for the fact that it accepts other parameters.

Function definition:

```
CreateResult Create(uint32_t parent_handle, int type, int
    restricted, int decrypt, int sign, const std::string
    &user_auth, const std::string &sensitive_data, const
    std::vector<uint8_t> &auth_policy, int session_handle);
```

The function takes nine input parameters, which correspond to those already present in `CreatePrimary`, except for the unique field. Unlike the previous function, what changes here is that we pass the handle of the primary key, and the `auth_policy` field, in this case, is not empty but instead contains the previously extracted `policy digest`.

The operation is the same as `CreatePrimary`, but in this case, the policy digest is provided during the definition of the public parameters. In this way, we ensure that the key can only be used if the selected PCR is still in the key's creation state.

The return value in this case is the data structure `CreateResult`. This has two essential fields to use the key just created, namely `tpm2b_private` and `tpm2b_public`. In contrast, the other parameters are the same as `CreatePrimaryResult`, except for the handle field.

4.3.7 Load RSA Key

The function `Load()` is necessary to use the newly created key because, unlike the primary key that is already inside the TPM after creation, the child key must be loaded explicitly for use. This is because the creation function does not return the key handle. There is the option to use `CreateLoad` function, which allows for the direct creation and loading of the key, but in our case, this function is not supported by the TPM 2.0 used.

Function definition:

```
LoadResult Load(ESYS_TR parent_handle, const
    std::vector<uint8_t> &tpm2b_private, const
    std::vector<uint8_t> &tpm2b_public, int session_handle);
```

The function takes four parameters as input: the handle of the primary key; the private part of the key; the public part of the key and the handle of the HMAC session created at the beginning.

If the key has been successfully loaded, the return parameter is a data structure called `LoadResult`, which contains: the return code; the child key handle and the primary key's name.

4.3.8 Encrypt Data Blob

The function `Load()` is responsible for encrypting a data blob of appropriate size to be encrypted using an RSA key. This function has been extensively studied and tested, as explained in the analysis phase because, depending on the session passed among the parameters, this function either returns successfully or returns an error related to incorrect handling of the necessary permissions.

Function definition:

```
std::vector<uint8_t> Load(uint32_t key_handle,
    const std::vector<uint8_t> &message, uint32_t
       session_handle);
```

The function takes three fields as input: the handle of the previously loaded key; the data blob to be encrypted and the session handle of the HMAC session created in the initial phase.

The function returns the encrypted data blob in the case of successful encryption.

4.3.9 Decrypt Data Blob

The function `DecryptRSAWithSession` is responsible for decrypting the encrypted data blob. In this case, as well, this function has been extensively studied and analysed, and it requires an input session policy that depends on the previously selected PCR to perform decryption.

Function definition:

```
std::vector<uint8_t> DecryptRSAWithSession(uint32_t
    key_handle, const std::vector<uint8_t> &message,
    uint32_t session_handle);
```

The function receives three parameters as input: the handle of the loaded key; the encrypted data blob and the policy session.

The function returns the decrypted data blob in case of success.

4.3.10 Sign Data Blob

The function `Sign()` is responsible for signing a data blob using the created key. The difficulties encountered in this case are the same as those encountered for the decrypt function.

Function definition:

```
SignResult Sign(uint32_t key_handle, int type, const
    std::string &str, uint32_t session_handle);
```

The function takes four parameters as input: the handle of the loaded key; the type of key used for the signing operation; the data blob to be signed and the policy session handle.

The function returns the `SignResult` data structure, which contains, in case of success: the return code; the algorithm used for signing and the signature.

4.3.11 Verify Signature

The function `VerifySignature()` is responsible for verifying the signature created through the Sign function. Like the encrypt function, this one requires an HMAC session and not a Policy session.

Function definition:

```
TPM2_RC VerifySignature(uint32 t key_handle, const
    std::string &str, const SignResult &in_signature,
    uint32_t session_handle);
```

The function takes four parameters as input: the handle of the loaded key; the original data blob used to compute the signature; the generated signature and the HMAC session handle.

The function returns the return code of the verification function, returning `TPM2_RC_SUCCESS` only if the verification was successful.

4.4 TPM2-OpenSSL Changes

The `tpm2-openssl` is an open-source project which implements a provider to integrate the TPM operations to `OpenSSL 3.x`. It relies entirely on ESAPI and uses

the `tpm2-tss` software stack implementation. The provider allows the usage of some TPM 2.0 features via the command line using `openssl` or via the `libcrypto API`.

By analysing the code and studying the various examples provided by the library itself, it was possible to see how to create a TLS server using the commands provided by OpenSSL and utilising the provider. The first fundamental step was to create an RSA key using these commands, and upon examining the code, it was observed that this occurred in two distinct steps. The first step involved creating a primary key, while the second step referred to the creation of the key we desired, using the previously created primary key.

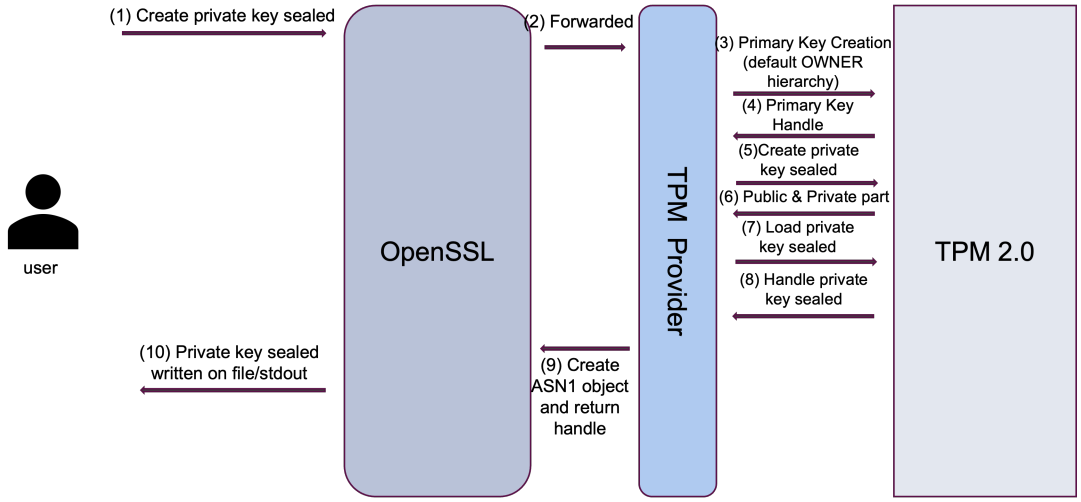


Figure 4.21. Sealed private key creation

Based on this knowledge, the key creation process was modified to force the creation of a private key dependent on the values of the PCRs. The changes made primarily affected these two phases. The first modification involved creating an HMAC Session to ensure that it was unique for both the primary key and the child key creation. The second modification involved creating the child key to ensure its dependency on the PCR values. This was achieved by creating a Policy Session to extract the policy digest created by selecting specific PCRs. The final code modification pertained to the PCR state check, which was included in the code section responsible for loading the private part of the newly created RSA key. The execution flow can be seen in the following Figure 4.21.

The overall process is depicted in the following Figure 4.22, emphasising the steps required for a Web Server implemented either through Nginx or created using OpenSSL to leverage a previously generated Sealed private key. In summary, the Web Servers seek access to the key through OpenSSL, which, in turn, knows it must request it from the TPM provider. Following the initialisation of essential data structures, the provider loads the sealed key into the TPM. Upon receiving key information from the TPM, the provider generates an ASN1 object, subsequently handed over to OpenSSL. OpenSSL then retrieves the private key from this object.

Unfortunately, this solution was discarded due to the following problems that arose during the testing phase:

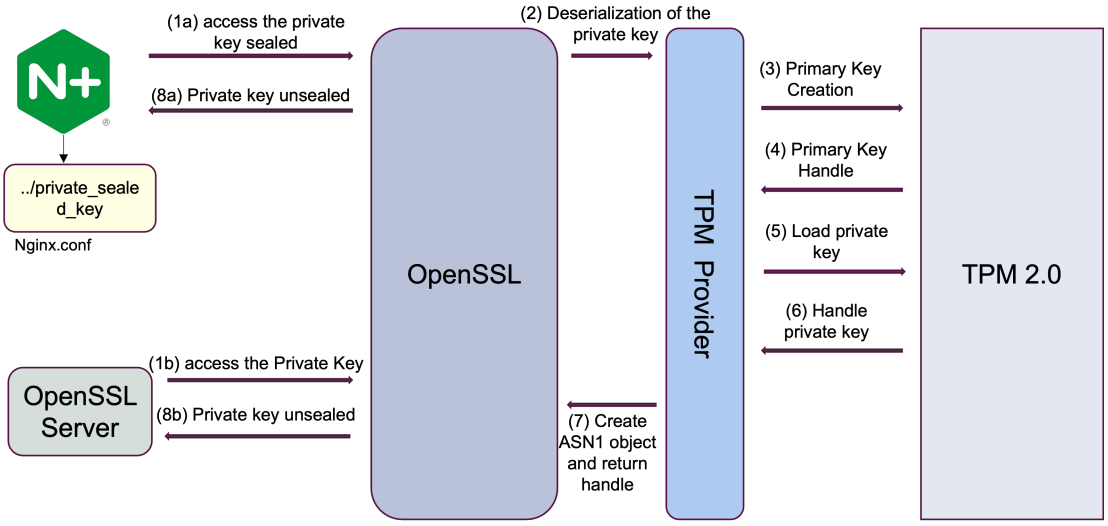


Figure 4.22. Web server initialisation

- the private key needs to be created using the tpm2 provider and can not be imported;
- the list of algorithms supported by the tpm2 provider may not contain the algorithm needed;
- problem found when the client tried to connect to the server without the tpm2 provider. This issue was identified long before modifying the source code by running tests provided by the library itself using the hardware TPM instead of the software TPM, as specified in the library.
- PCR state check should be automatically managed by the TPM during key usage, but this did not occur;
- the global configuration of OpenSSL needs to be modified to integrate the new provider.

For this reason, an alternative solution and implementation were pursued, which will be discussed in the next chapter.

Chapter 5

Implementation

In this chapter, we will analyse the final and official solution of this thesis. Special attention will be given to explaining the motivations that led to choosing this solution and how it was implemented.

The proposed solution is divided into two phases: in the first phase, the user needs to create the Sealed key, while in the second phase, the key is Unsealed to be used by Apache. In particular, the solution is based on Apache, and specifically, the mod-ssl module, which is responsible for creating Transport Layer Security to protect the data.

The purpose of this chapter is to provide a high-level description of the operations carried out, while the practical and implementation-level analysis will be discussed in the User's developer chapter.

5.1 Motivations

The implemented solution involves the usage of the Apache server. Apache was chosen among the various possible web servers because it already provides some functionalities essential for the sought-after solution. The main functionalities that Apache provides are:

- Popularity: Apache popularity means that there is a large community of users, extensive documentation making it a reliable choice for many projects;
- Module Ecosystem: Apache's modular architecture allows the addition of various modules to extend its functionality. The mod-ssl module specifically adds support for TLS encryption. If TLS support is a critical requirement for a project, mod-ssl provides a well-established and widely used solution;
- Configuration Flexibility: Apache provides extensive configuration options, allowing fine-grained control over server behaviour. The mod-ssl module integrates seamlessly into Apache's configuration system, making it easier to manage TLS settings.

Once the web server was chosen, efforts were made to integrate the necessary functionalities for using the TPM in the easiest and most efficient way possible. In this context, two possible solutions were identified: creating a new module to handle TPM initialisation and the related functions for implementing Unsealing; or extending the mod-ssl module to ensure it supports the desired new functionalities. Between these two options, the chosen solution was to extend the mod-ssl module for the following reasons:

- Efficiency and Optimisation: the modifications needed were relatively minor and aligned with the existing module's functionality, so it was more efficient to enhance the existing code rather than create an entirely new module. This approach helped avoid code duplication and reduced the complexity of managing multiple interdependent modules;
- Integration with Existing Features: modifying an existing module allowed for seamless integration with the features and functionalities already present in that module. This was beneficial because the modifications were intended to enhance the behaviour of the existing module in a way that it maintained the compatibility with its original design.
- Code simplicity: extending the module source code was easier than creating a new module from scratch and this led to faster development.

5.2 Creation of the Sealed Key

The first phase of the implementation involves the creation of the Sealed key, this is shown in the following Figure 5.1. This process differs from what was seen in the previously presented solution. In fact, while in the previous solution, we saw how to create a private key that depended on the values of PCR, in this case, the private key is not created by the TPM but is imported and processed later. The Sealing process is carried out in full compliance with the TCG specification, using the Create function to create Sealed Objects and the Unseal function for data retrieval.

In particular, the function used to create Sealed objects does not allow using data larger than 128 bytes. Initially, this posed a problem as it prevented the creation of a single Sealed object containing the private key. It was decided to circumvent the issue by breaking down the private key into many data chunks, each with a maximum size of 128 bytes, enabling the creation of multiple Sealed objects. Each Sealed object created in this manner is dependent on the value of the PCR at that moment.

To retrieve the sealed data, it is necessary to load these into the TPM. Therefore, during the creation of Sealed objects, information was stored in two separate files. This information, in the form of data buffers, corresponds to the public and private parts created for each Sealed object. The public part contains the necessary information about how the object was created, while the private part contains details about the sensitive data that has been sealed. While the public part can

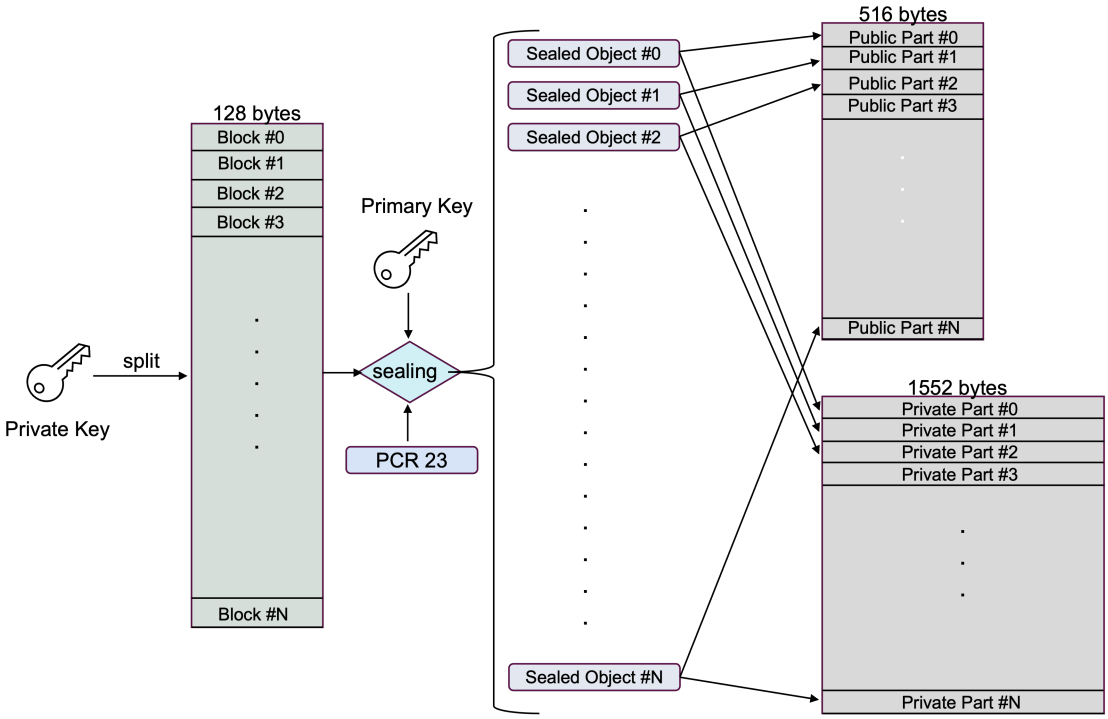


Figure 5.1. Sealing process overview

be read using commands provided by tpm2-tools, the private part is encrypted and unreadable without using the primary key used during the creation phase. Additionally, the policy on the PCR state must be adhered to.

In the proposed solution, PCR 23 of the sha256 group was used for two reasons: it was not used beforehand by other processes, and it was easy to use for testing. In a real-world scenario, other PCRs should be chosen based on what needs to be monitored.

These two files containing this information can be saved on the disk without posing any security risk because, to the eyes of a generic user or an attacker, they appear as encrypted files. Furthermore, as a result of concatenating byte buffers, it is impossible to extract and use the Sealed Objects without knowledge of the exact size of each data chunk.

The success of this operation is primarily based on the fact that the public and private parts have different but fixed sizes, independent of the current sealed data. The sizes are 516 bytes for the public part and 1552 bytes for the private part. This facilitated an easy implementation for the Unsealing phase, where no other data structure was associated to maintain information regarding the actual dimensions of each private/public buffer.

5.3 Unsealing integration in Apache

The following Figure 5.2 details the various phases related to the configuration of Apache when using a Sealed private key, as previously depicted in the Figure

5.1. We can group the various operations into 3 phases: Apache configuration, the unsealing process, and the private key recovery.

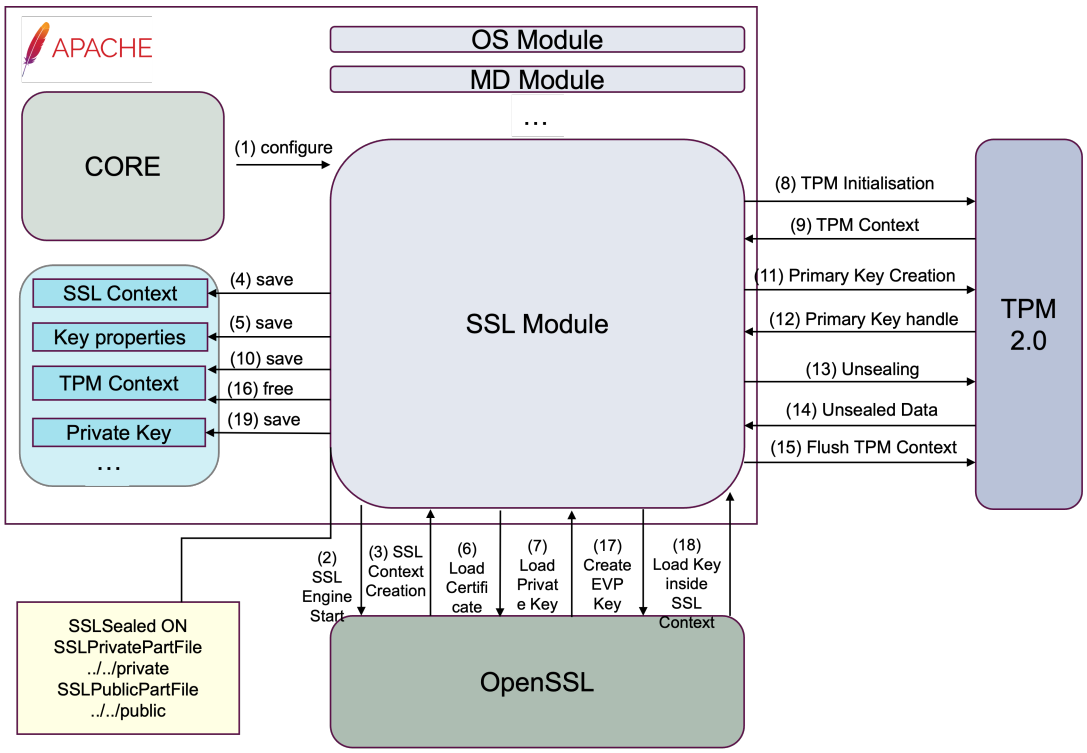


Figure 5.2. Apache configuration

5.3.1 Apache configuration

The added directives allow Apache, during the initial configuration phase, to use the TPM only when necessary. This is because every effort has been made to integrate code that minimally alters the standard operation of Apache in case the user wants to use the standard version. By doing so, the user can decide whether or not to use the TPM, depending on whether they want to use Sealed keys or regular keys saved in a file on the PC.

During the configuration phase with the modified version, what happens is analogous to what would occur in the standard case, except for the configuration of the parameters related to the three new directives introduced. The new directives introduced are:

- **SSLSealed**: used as a selector to inform the server that a sealed key is being used, requiring appropriate precautions and functions;
- **SSLSealedPublicFile**, corresponding to the path of the file containing the public part;
- **SSLSealedPrivateFile** (corresponding to the path of the file containing the private part.

Once the various necessary data structures have been initialised, the module requests the associated certificate and private key.

5.3.2 Unsealing process

In the second phase, the Unsealing process takes place to recover the private key 5.3. After reading data from the two files corresponding to the two previous directives, the public and private parts for each sealed object are retrieved. This allows us, after initialising the necessary resources for TPM usage, to proceed with loading the public part and private part data into the TPM to perform the Unsealing function on them. The result of this process is a byte buffer containing the unsealed private key.

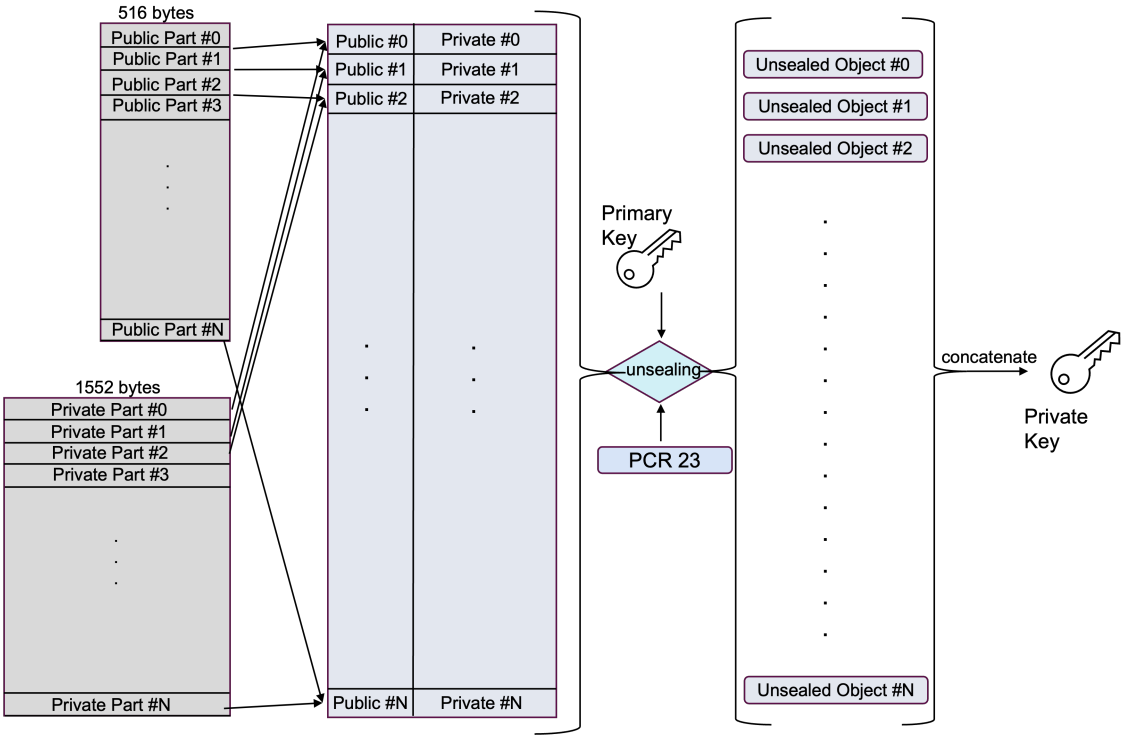


Figure 5.3. Unsealing process overview

5.3.3 Private key recovery

The final phase involves using the functionalities provided by OpenSSL to recover the private key from a byte buffer. This way, the private key is loaded into the SSL context created during the module configuration and can be used to perform all the necessary operations during the execution of TLS.

5.4 Reseults achieved

The solution developed offers the following benefits:

- No additional modules or external providers are needed. Indeed, the only external elements needed are the library to use the TPM.
- Better flexibility with respect to key creation. Keys can be created externally using any algorithm that may not be implemented by the TPM and supported by the web server. In addition, it offers the possibility to integrate new key types for future specifications.
- Possibility to use existing Private keys. No need to create a private key if one already exists because the Sealing operation can be done on any file containing any key or data blob.
- No need to drastically change the functioning of the SSL module. The functionalities of the module were not changed but instead were enhanced, permitting the usage of the TPM for the Unsealing operation. This is a huge accomplishment with respect to the solution developed for the tpm2-openssl library, in which the operation was disrupted, forcing the creation of private keys depending on PCRs. Instead, with this solution, the user itself can decide to use a Sealed key if available.
- Private keys are protected from being exploited or stolen. Once the Sealed private key is created, the original private key can be safely deleted, forcing anyone who wants to access it to use the TPM.

From a functional perspective, the TPM has added a new layer of protection to safeguard the private keys used by Apache. Specifically, being a hardware component, the TPM ensures that the Primary key used to create our Sealed key is tamper-resistant because the seed from which it was created is stored in a Shielded location inaccessible to the user without authorisation. This means it is resistant to various physical attacks and resilient against attacks that target solutions implemented through software.

Unfortunately, the Sealed key is not tamper-resistant but possesses the following characteristics:

- the Sealed private key cannot be directly accessed because there is no file containing the actual data, but it can only be recovered by loading the private part inside the TPM after being authorised;
- even though the private part is stolen from the machine and it is given in input to another machine equipped with a TPM 2.0, the loading operation will fail because the parent key will be different;
- in the event of multiple unsuccessful authorisation attempts during the Unsealing operation, triggered by a compromised platform, a preventive measure similar to a dictionary attack prevention mechanism is enacted. This mechanism imposes a time delay, compelling the attacker to wait before making additional attempts to access the resource;

- the public and private files containing the public and private parts appear as encrypted files. In case an attacker has no knowledge about the Sealing operation and how it is performed, there is no way he can decrypt and use this information to retrieve the private key.

The level of protection can be enhanced by combining various types of authorisation with the existing one based on the system's state. Some possible implementations include the use of passwords or external tools such as smart cards, USB drives, or other hardware components to implement multi-factor authentication.

Moreover, through the utilisation of highly specialised techniques that demand a pre-existing understanding of TPM, knowledge typically lacking in a standard attacker, we've effectively minimised the potential threats. In essence, besides a grasp of the TPM's general functionality, an attacker must possess comprehensive knowledge of the data structures and functions employed in the Sealing operation.

Chapter 6

Testing

The chapter presents the results of the tests performed on the final solution using the Apache version patched with the implemented solution. In particular, performance tests were performed to evaluate both latency times and resources consumption. In addition, functional tests were performed to check if the solution behaved correctly.

6.1 Testbed

The evaluation tests were performed using the following testbed: a machine having an Intel i7-7600U, 32GB of RAM, running Ubuntu 22.10 with kernel version 5.19 and a discrete Infineon SLB9670 TPM 2.0. The installation and configuration of the testbed can be found in the Appendix A.

6.2 Functional tests

Functional tests developed are designed to assess the operation of the proposed solution. Specifically, two cases were analyzed: in the first case, the machine's state was compromised, while in the second case, the machine was in a predefined state.

During the development of these tests, an issue related to the TABRM Daemon was identified. The error retrieved from the Apache logs is this one:

```
Error number :  0xa0008:  failed to allocate dbus proxy object:
Timeout was reached.
```

After some research, it was found that this issue was already known in some implementations of Apache with the TPM. Analysing the execution flow it was seen that Apache spawns child processes using the `fork` command and each child calls the `C_Initialize API`, which initialises the TABRM daemon and the `DBus` connection. The problem is with the `C_Finalize` function, which does not clear all the resources allocated for the DBus connection. So the problem appears to be related to the `glib`

library that should clear everything about DBus connection when finalise functions are executed. To solve this problem one solution is to remove glib from the client library. That's a pretty heavy lift which has limited long-term value given that the goal is to have the kernel take over this role anyway. The other solution found and implemented involves using the resource manager integrated within the kernel. By deactivating the TABRM and specifying in the TCTI module initialisation phase that the `tpmrm0` should be used, which corresponds to the in-kernel resource manager, the problem encountered with DBus allocation no longer persists. In case, the in-kernel RM is not available and the TABRM daemon is the only component accessible, it is still possible to use this solution, but Apache must be launched in `debug mode`, or the maximum number of clients must be specified as 1.

The PCR value during the creation of the sealed key is represented in the following Figure 6.1:

```
23:0x0000000000000000000000000000000000000000000000000000000000000000
```

Figure 6.1. Initial value of PCR 23

6.2.1 Compromised machine

The first step is to modify the PCR 23 to a different value than the one used during the creation of the sealed key. This can be done using the `prcextend` command provided by the `tpm2-tools`.

```
23:0x44F12027AB81DFB6E096018F5A9F19645F988D45529CDED3427159DC0032D921
```

Figure 6.2. Value of PCR 23 after the extend operation

After changing the value of PCR 23 to a new value, shown in the Figure 6.2, the next step is to launch Apache. The server configuration will occur, and the data structures necessary for the Unsealing process will be created and initialised. However, during the first Unsealing operation, the automatic check by the TPM on the state of PCR 23 will fail, resulting in the failure of the Unsealing process and the server's startup. The following errors can be read in the error_log file, an example is shown in the following Figure 6.3.

6.2.2 No compromised machine

After confirming that the value of PCR 23 is in the same state as when the sealed key was created Figure 6.1, Apache is started. During this time frame, the necessary data structures to save the unsealed private key will be created, and the TPM will be initialised. If PCR 23 remains in the same state throughout the Unsealing phase,

```
Error: Unsealing process failed.
WARNING:esys:src/tss2-esys/api/Esys_Unseal.c:295:
    Esys_Unseal_Finish() Received TPM Error
ERROR:esys:src/tss2-esys/api/Esys_Unseal.c:98:Esys_Unseal()
    Esys Finish ErrorCode (0x0000099d)
```

Figure 6.3. Errors from the log file

it will conclude successfully, and the private key will be formatted to be loaded into the SSL context. If the process concludes successfully no errors will be written in the error_log file but only messages containing meaningful information about the correct execution of the TPM functions. This can be seen in the following Figure 6.4:

```
Initialization of the TCTI context successfull
Initialization of the ESYS context successfull
HMAC Session Created
Primary Key Created
    ...
    Policy Session Created
    Policy Digest Created
    Unmarshalling of the Private Part
    Unmarshalling of the Public Part
    Loading sealed object successful
    Unsealed object successful
    Loaded Data Blob Flushed
    Policy Session Flushed
    ...
Primary Key Flushed
HMAC Session Flushed
```

Figure 6.4. Messages from the Unsealing execution process

6.3 Performance tests

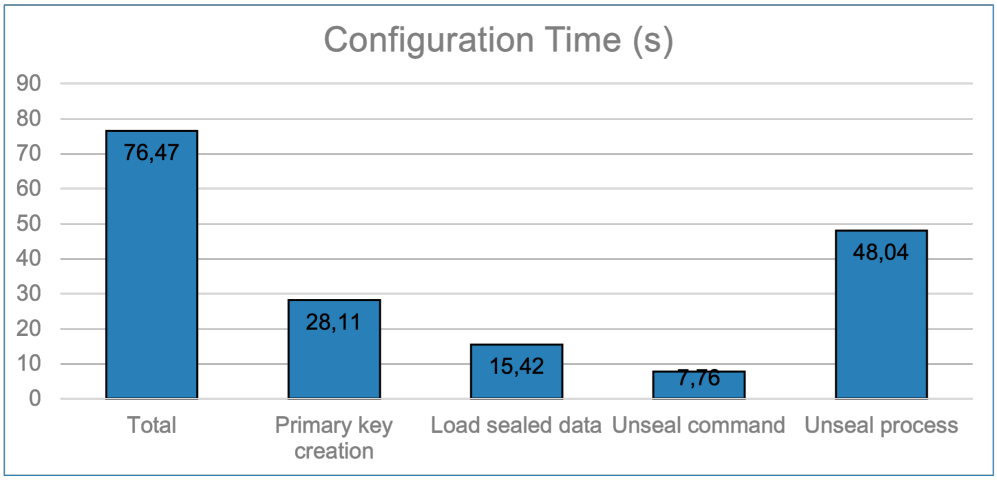
Performance evaluation is based on measuring the time required to configure the Apache server using the proposed solution. Specifically, the time needed to configure an Apache server providing a standard private key was measured first, and then the time was compared with the proposed solution. Observations revealed that the configuration time in the standard case averages 0.06 seconds, while the configuration time using a sealed key averages 76.50 seconds.

The significant time difference is primarily due to the execution of certain operations by the TPM, which inherently is a slow component in performing cryptographic actions.

To better understand which functions were responsible for the execution slowness, the timings of some functions fundamental to the Unsealing process were measured.

The measured functions are as follows: `CreatePrimary`, `Load`, `Unseal`, and the set of functions necessary to proceed with the Unsealing operation, iterated for the number of sealed objects.

As seen in the Figure 6.3, the block of functions needed to retrieve the private key takes the longest time to complete. This is because it requires executing the following functions for the number of sealed objects, which in the test corresponds to 14 times: `StartAuthSession`, `Load`, `Unseal`, and `Flush` of temporary variables allocated in the TPM.



Next is the `CreatePrimary` function used to recreate the primary key from the Seed stored in a `Shielded Location`.

Then follow, in order, the `Load` functions to load the current sealed data into the TPM to then call the `Unseal` function.

Unfortunately, performance improvement is not possible because the bottleneck is the TPM itself, which takes a long time to execute these operations.

Tests were also conducted to measure the resources used regarding CPU usage and RAM consumption, but the obtained data is comparable to the standard ones since all functions are executed by the TPM, and the CPU is responsible for formatting the private key from the received buffer, as would happen in the standard case where the key is read from a file and loaded. RAM consumption is also comparable to the standard case.

Finally, tests were conducted to check the correct functioning of the server by making multiple requests, but the resources used in this case are also comparable to the standard case. This is because once the Unsealed key is loaded into the SSL context, it is treated as if it were a normal key read from a file.

Chapter 7

Conclusions and future work

The objectives of this thesis were to test various TPM functionalities and leverage some of them to protect the private keys used by web servers for secure client-server connections. These goals were achieved by exploring the basic functionalities of the TPM and using some of them to provide an implementation that offers a trade-off between security and performance.

Initially, efforts were made to develop examples demonstrating the use of various TPM functionalities. This allowed me to understand how the TPM functions based on the commands used and the input values provided. Using the developed examples and acquired knowledge, it was possible to extend the tpm2-openssl library. This library allows users to utilise some TPM functionalities by invoking commands provided by OpenSSL or libcrypto. The primary goal behind the modifications made to the tpm2 provider's operation was to enable users to create and use private keys that depended on the state of the machine. The creation of these keys involved retrieving the machine's state through the use of PCR and creating a security policy that governed access to the key. In this way, a web server wanting to use a key created in this manner had to use the tpm2-provider to request it from the TPM. This solution allowed any web server to use the private key created and protected by the TPM, with the only requirement being support for OpenSSL 3.x. Unfortunately, this solution was discarded after careful analysis because, in addition to not fully complying with the TCG specification, it was not user-friendly and flexible for those who did not have access to a TPM. For these reasons, a new solution was developed, this time selecting Apache as the starting web server.

The proposed solution involves using the mod-ssl module already included among the various modules offered by Apache. This module is responsible for creating a secure connection between the client and server using TLS. Since this module already provides all the basic functionalities needed to create an SSL context, it was decided to extend it to support Unsealing functionality. The Sealing and Unsealing operation, managed by the TPM, allows the creation of data protected by a primary key internal to the TPM and dependent on the state of the PCR, providing a measure of the machine's state. The Sealing operation is handled through code designed to receive the private key as input, which can be either created on the spot by the user or imported if already exists. The changes made to Apache involve the introduction of new directives for configuring the mod-ssl module and

the integration of some functions necessary to implement the Unsealing function. The general operation of Apache and the module remains unchanged since the goal was to achieve an implementation that was as non-invasive and flexible as possible.

The developed solution allows the use of sealed private keys, achieving a high level of security at the expense of performance. Indeed, the sealed keys created are resistant to attacks based on having access to the private key for brute force, long-exponent, or dictionary attacks, among others. However, this comes at the cost of increased latency for Apache configuration, as the standard version requires less than 1 second, while the modified version needs about 76 seconds. Unfortunately, it is not possible to reduce this time since the TPM itself is the bottleneck. It is recognised that the TPM takes a significant amount of time to perform the required cryptographic operations. Fortunately, tests have shown that the only negative measurement is related to latency, while CPU usage and RAM consumption remain unchanged both during the configuration phase and in the server usage phase, as once the key is retrieved, it is treated like a standard private key.

The proposed solution is a good starting point for integrating TPM functionalities into Apache and beyond. In fact, by making similar modifications to other web servers that use OpenSSL for secure connection creation, it is possible to enhance the security of the private keys used. Considering the solution suggested for Apache, it can be further improved by expanding the supported types of private keys and allowing the use of passphrase-protected keys. If the Unsealing operation in the tpm2-tss library is enhanced to support the creation of Sealed objects of various sizes, this would significantly reduce the time required for server configuration, estimated at around 30%. Moreover, since the proposed solution for the tpm2 provider heavily relies on the current version and considering it is a very active project, it is possible that in the near future, the Unsealing operation will be natively implemented, overcoming the encountered issues. The work carried out in this thesis aims to address challenges related to the use of web servers by providing a hardware solution to problems typically managed through software. This approach seeks to mitigate threats targeting software, offering a more robust and secure perspective.

Bibliography

[1] A. Segall, "Trusted Platform Modules - Why, When and How to Use Them", The Institution of Engineering and Technology, December 2016, ISBN: 978-1-84919-893-6

[2] P. S. Tasker, "Trusted computer systems", 1981 IEEE Symposium on Security and Privacy, Oakland (California), April 27-29, 1981, pp. 99–99, DOI 10.1109/SP.1981.10020

[3] J. Teo, "Features and Benefits of Trusted Computing", 2009 Information Security Curriculum Development Conference, Kennesaw (Georgia), September 25, 2009, pp. 67–71, DOI 10.1145/1940976.1940990

[4] T. C. Group, "Trusted Platform Module Library. Part 1: Architecture", September 2016, https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.38.pdf

[5] K. G. Will Arthur, David Challener, "Practical Guide to TPM 2.0: Using the New Trusted Platform Module in the New Age of Security", Springer Nature, January 2015, ISBN: 978-1-4302-6583-2

[6] B. Berger, "Trusted computing group history", Information Security Technical Report, vol. 10, August 2005, pp. 59–62, DOI 10.1016/j.istr.2005.05.007

[7] T. C. Group, "Trusted Platform Module 2.0: A Brief Introduction", August 2015, https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-A-Brief-Introduction.pdf

[8] T. C. Group, "TCG TSS 2.0 Overview and Common Structures Specification", September 2021, https://trustedcomputinggroup.org/wp-content/uploads/TSS_Overview_Common_v1_r10_pub09232021.pdf

[9] T. C. Group, "TCG TSS 2.0 TAB and Resource Manager Specification", April 2019, https://trustedcomputinggroup.org/wp-content/uploads/TSS_2p0_TAB_ResourceManager_v1p0_r18_04082019_pub.pdf

[10] T. tpm2-software project, https://github.com/tpm2-software/tpm2-abrmd

[11] T. C. Group, "TCG TSS 2.0 TPM Command Transmission Interface (TCTI) API Specification", January 2020, https://trustedcomputinggroup.org/wp-content/uploads/TCG_TSS_TCTI_v1p0_r18_pub.pdf

[12] T. C. Group, "TCG TSS 2.0 Enhanced System API (ESAPI) Specification", September 2021, https://trustedcomputinggroup.org/wp-content/uploads/TSS_Overview_Common_v1_r10_pub09232021.pdf

[13] T. C. Group, "TCG TSS 2.0 Feature API (FAPI) Specification", September 2019, https://trustedcomputinggroup.org/wp-content/uploads/TSS_FAPI_v0.94_r04_pubrev.pdf

[14] A. Fuchs, "Enabling the TPM2.0 Ecosystem in Linux", Presentation at Open Source Summit + Embedded Linux Conference Europe, Lyon (France), October 25, 2018. https://static.sched.com/hosted_files/osseu19/35/OSSEU2019.pdf

[15] T. C. Group, "TCG TSS 2.0 System Level API (SAPI) Specification", August 2019, https://trustedcomputinggroup.org/wp-content/uploads/TSS_SAPI_v1p1_r29_pub_20190806.pdf

[16] T. C. Group, "Trusted Platform Module Library Part 3: Commands", November 2019, https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part3_Commands_pub.pdf

Appendix A

User's manual

In this appendix, we will describe the necessary steps to replicate the solutions developed during the thesis. In particular, all information about the device on which the solutions were developed will be provided, including the libraries required for proper functionality.

A.1 Requirements

The solutions were developed on a machine equipped with TPM 2.0. The operating system used is `Ubuntu 22.10`, which can be downloaded from https://ubuntu.com/download/desktop.

The following information regarding the TPM 2.0 used was retrieved using the command provided by `tpm2-tools`

```
$ tpm2 getcap properties-fixed
```

Listing A.1. TPM 2.0 Information

```
TPM2_PT_FAMILY_INDICATOR:
  raw: 0x322E3000
  value: "2.0"
TPM2_PT_LEVEL:
  raw: 0
TPM2_PT_REVISION:
  raw: 0x74
  value: 1.16
TPM2_PT_DAY_OF_YEAR:
  raw: 0xF
TPM2_PT_YEAR:
  raw: 0x7E0
TPM2_PT_MANUFACTURER:
  raw: 0x49465800
  value: "IFX"
TPM2_PT_VENDOR_STRING_1:
  raw: 0x534C4239
```

```
  value: "SLB9"
TPM2_PT_VENDOR_STRING_2:
  raw: 0x36373000
  value: "670"
...
```

This command allows you to obtain all the necessary information for the correct use of the integrated TPM in the machine being used. In addition to providing important information such as the TPM manufacturer. For readability purposes, only a limited set of information were displayed; additional details can be found in the main folder under the name of `tpm2_information.txt`.

As mentioned earlier, the following libraries are required for the proper use of TPM:

- **tpm2-tss**;
- **tpm2-tools**;
- **tpm2-abrmd**;
- **tpm2-openssl**.

In addition to the previously listed libraries, two Web servers were used for the proposed solutions:

- **Nginx**;
- **Apache2**.

A.1.1 tpm2-tss

This repository contains the source code that implements the TPM2 Software Stack as defined by the Trusted Computing Group (TCG). In order to build and install the `tpm2-tss` software, specific software packages are needed. Often, these dependencies are tailored to the platform in use, for GNU/Linux:

- GNU Autoconf;
- GNU Autoconf Archive, version $\geq$ 2019.01.06;
- GNU Automake;
- GNU Libtool;
- C compiler;
- C library development libraries and header files;
- pkg-config;

- doxygen;
- OpenSSL development libraries and header files, version $\geq$ 1.1.0;
- libcurl development libraries;
- Access Control List utility;
- JSON C Development library;
- Package libusb-1.0-0-dev.

Once all the dependencies are installed to set up the `tpm2-tss` source code for configuration and install it, the following code can be used:

```
$ git clone https://github.com/tpm2-software/tpm2-tss.git
$ cd tpm2-tss
$ ./bootstrap
$ ./configure --prefix=/usr
$ make -j5
$ sudo make install
```

A.1.2 tpm2-abrmd

The `tpm2-abrmd` repository implements a system daemon following the specifications for the TPM2 Access Broker (TAB) and Resource Manager (RM) from the Trusted Computing Group (TCG). The daemon, is developed using Glib and the GObject system. The dependencies for GNU/Linux are:

- GNU Autoconf;
- GNU Autoconf archive;
- GNU Automake;
- GNU Libtool;
- C compiler;
- C Library Development Libraries and Header Files (for pthreads headers);
- pkg-config;
- glib and gio 2.0 libraries and development files;
- libtss2-sys, libtss2-mu and TCTI libraries from https://github.com/tpm2-software/tpm2-tss dbus;

To build the source code and install it from the repository the following code can be used:

```
$ git clone https://github.com/tpm2-software/tpm2-abrmd.git
$ cd tpm2-abrmd
$ ./bootstrap
$ ./configure --with-dbuspolicydir=/etc/dbus-1/system.d
--with-udevrulesdir=/usr/lib/udev/rules.d
--with-systemdsystemunitdir=/usr/lib/systemd/system
--libdir=/usr/lib64 --prefix=/usr
$ make -j5
$ sudo make install
```

A.1.3 tpm2-tools

The `tpm2-tools` project offers a comprehensive set of commands, both at a low-level and aggregate level, to access the functionalities provided by the TPM 2.0 device. The primary purpose of this project was to gain an understanding of and analyse the commands required for implementing the developed solutions. The dependencies needed for GNU/Linux are:

- GNU Autoconf (version $\geq$ 2019.01.06);
- GNU Automake;
- GNU Libtool;
- pkg-config;
- C compiler;
- C Library Development Libraries and Header Files (for pthreads headers);
- ESAPI - TPM2.0 TSS ESAPI library (tss2-esys) and header files;
- OpenSSL libcrypto library and header files (version $\geq$ 1.1.0);
- Curl library and header files.

The following code is needed to build and install the tpm2-tools project:

```
$ git clone https://github.com/tpm2-software/tpm2-tools.git
$ cd tpm2-tools
$ ./bootstrap
$ ./configure --prefix=/usr
$ make -j5
$ sudo make install
```

A.1.4 tpm2-openssl

The `tpm2-openssl` library enables accessibility to TPM 2.0 through the standard OpenSSL API and command-line tools, making it possible to incorporate TPM support into nearly any OpenSSL 3.x based application. Adheres to the new OpenSSL provider API and strictly avoids using any legacy API. As a result, this implementation:

- Maintains compatibility with OpenSSL 3.x and, ideally, future OpenSSL versions;
- Is not compatible with any previous versions, including the current OpenSSL 1.1.
- Involves a substantial restructuring of the `tpm2-tss-engine`, with the code still present but extensively reorganised to align with the new OpenSSL API. Consequently, this implementation:
  - Retains nearly all functions of the tpm2-tss-engine, despite changes in the command-line interface and the API.
  - Preserves the format of the TSS2 PRIVATE KEY file, ensuring that keys created by the previous version remain functional.

For more complex scenarios such as SSL or X.509 operations, the in-kernel resource manager is insufficient. Instead, the `tpm2-abrmd` must be employed.

The dependencies for GNU/Linux are:

- pkg-config;
- GNU Autotools (Autoconf, Automake, Libtool);
- GNU Autoconf Archive, version $\geq$ 2017.03.21;
- C compiler and C library;
- TPM2.0 TSS ESAPI library (libtss2-esys) $\geq$ 3.2.0 with header files;
- OpenSSL $\geq$ 3.0.0 with header files;

The following code is needed to build and install the tpm2-openssl project:

```
$ git clone https://github.com/tpm2-software/tpm2-openssl.git
$ cd tpm2-openssl
$ ./bootstrap
$ ./configure --prefix=/usr
$ make
$ sudo make install
```

A.1.5 nginx

Nginx is a popular open-source web server and reverse proxy server software that is known for its high performance, efficiency, and scalability. It is designed to handle a wide range of web-serving tasks, including serving static and dynamic content, load balancing, and acting as a reverse proxy for other web servers.

Nginx is known for its event-driven, non-blocking architecture, which allows it to efficiently handle a large number of concurrent connections without consuming excessive system resources. It's often used in scenarios where high concurrency and low latency are essential, making it a popular choice for serving web applications and content in high-traffic environments.

This server software supports various features and modules that enable users to customise and extend its functionality. It is highly configurable through its configuration files, allowing administrators to define server behaviour, set up virtual hosts, and configure security settings.

The following code can be used to download, install Nginx and enable the `ssl_module` which is responsible for creating the SSL/TLS session.

```
wget https://nginx.org/download/nginx-1.25.1.tar.gz
tar zxf nginx-1.25.1.tar.gz
cd nginx-1.25.1
./configure
--sbin-path=/usr/local/nginx/nginx
--conf-path=/usr/local/nginx/nginx.conf
--pid-path=/usr/local/nginx/nginx.pid
--with-http_ssl_module
--with-stream
--with-pcre=../pcre2-10.42
--with-zlib=../zlib-1.2.13
--without-http_empty_gif_module
make
sudo make install
```

A.1.6 apache2

Apache2 is a widely used open-source web server software that is renowned for its flexibility, reliability, and scalability. It is designed to serve web pages, applications, and content to clients, typically over the HTTP or HTTPS protocols. Apache2 is an integral component of the LAMP (Linux, Apache, MySQL, PHP/Perl/Python) stack and is compatible with various operating systems, making it a versatile choice for hosting websites and web applications.

This server software is highly configurable and extensible, allowing administrators to customise its behaviour to suit their specific needs. Apache2 employs a modular architecture, with each module responsible for a specific aspect of server functionality, such as authentication, security, and performance optimisation.

The following code can be used to download and install the Apache Server using the default configuration.

```
wget https://dlcdn.apache.org/httpd/httpd-2.4.58.tar.gz
tar zxf httpd-2.4.58.tar.gz
cd httpd-2.4.58.tar.gz
$ ./configure
$ make
$ make install
```

SSL/TLS can be enabled in two different ways:

- executing the following command during the build of the source code:

```
./configure --enable-ssl
```

- modifying the configuration file found in the default directory in which Apache was installed, /usr/local/apache2/conf/httpd.conf, and removing the comments for the following lines:

```
LoadModule ssl_module modules/mod_ssl.so

Include "conf/extra/httpd-ssl.conf"

<IfModule ssl_module>
    SSLRandomSeed startup builtin
    SSLRandomSeed connect builtin
</IfModule>
```

It is possible to debug Apache using gdb. Prior to using gdb, ensure that the server has been compiled with the -g option in CFLAGS to include symbol information in the object files. This can be done in the following way:

```
.configure CFLAGS="-g"
```

The only somewhat challenging aspect of running gdb with Apache is ensuring that the server operates in a single-process mode so that the parent process, which handles requests, becomes the target for debugging rather than spawning child processes. To address this, the -X option needs to be used. However, certain modules may not cooperate with the -X option but can operate smoothly if you restrict the server to a single child process using `MaxClients 1`.

A.2 Use cases

The source code to test the use cases previously defined can be found in the main directory `use_cases`. Inside it can be found six files:

- main.cpp

- tpm.cpp
- test.cpp
- tpm.h
- structures.h
- test.h

The language used to implement the use cases is C++ instead of C. The main reason for using C++ is due to the tpm2-openssl software which is entirely written in the same language.

The following code can be used to execute the functionality to test by selecting the corresponding use case. The possible use cases are Encryption/Decryption and Signing.

The code options are:

- -v or –verbose: to print on standard output information about the execution;
- -in or –input: to pass the file containing the data to encrypt/sign;
- -out or –output: to write the returned data of the encryption/signing;
- -enc or –encryption: to test the encryption operation;
- -sig or –signing: to test the signing operation;

```
cmake
.\test [-enc/-sig] -v <enable_printing> [-in <input_file>]
   [-out <output_file>]
```

A.3 Tpm2-OpenSSL solution

The following section defines the changes made to the tpm2-openssl library in order to enable the creation of an RSA key protected with PCRs values.

The code can be found in the main directory tpm2-openssl. The file changed are:

- src/tpm2-provider-keymgmt-rsa.c
- src/tpm2-provider-decoder-tss2.c

and the new file created are:

- src/tpm2-provider-sessionmgmt.c
- src/tpm2-provider-sessionmgmt.h

The following code can be used to build the code:

```
cd tpm2-openssl
./bootstrap
./configure --enable-debug
make
make install
```

The following code represented in the following Figure A.1 can be used to test it by creating a Server with OpenSSL:

```
# create the primary key
tpm2_createprimary -C o -G rsa -c primary.ctx
# create the session
tpm2_startauthsession -S session.dat
# extend the session selecting the PCR 23
tpm2_policypcr -S session.dat -l "sha256:23" -L policy.dat
# flush the session created
tpm2_flushcontext session.dat
# create the private key
tpm2_create -u key.pub -r key.priv -C primary.ctx -L policy.dat
# load the private key
tpm2_load -C primary.ctx -u key.pub -r key.priv -n
   unseal.key.name -c unseal.key.ctx
# load the private key to persistent handle
HANDLE=$(tpm2_evictcontrol -c unseal.key.ctx | cut -d '␣' -f 2 |
   head -n 1)
# use the private key to generate a self-signed certificate
openssl req -provider tpm2 -provider default -propquery
   '?provider=tpm2' -x509 -config testcert.conf -key
   handle:${HANDLE} -out testcert.pem
# start SSL server with RSA-PSS-PSS signing, port 4432
openssl s_server -provider tpm2 -provider default -propquery
   '?provider=tpm2' -accept 4432 -www -key handle:${HANDLE}
   -cert testcert.pem &
```

Figure A.1. Server creation using OpenSSL and the TPM2 provider

To enable Nginx to use the private key created using the TPM 2.0 a new entry in the `openssl.cnf` file is needed A.2 in this way we force the usage of the tpm when it is needed. This can be done by including these lines:

A.4 Apache Solution

In the following section, we are going to discuss how to use the solution created with Apache. The code can be found in the `httpd-2.4.57` directory, but all the changes made can be found in `httpd-2.4.57/modules/ssl` directory.

```
1 [provider_sect]
2 default = default_sect
3 + tpm2 = tpm2_sect
4 [default_sect]
5 activate = 1
6 + [tpm2_sect]
7 + activate = 1
```

Figure A.2. Global configuration of the TPM2 provider

The following code Figure A.3 can be used to build the solution, in particular to install the module `mod-ssl` with the patch.

```
1 cd httpd-2.4.57
2 CFLAGS="-g" ./configure --enable-ssl
3 make
4 make install
5 make clean
6 apxs -c -i -I modules/md -ltss2-esys -ltss2-mu
     -ltss2-tcti-tabrmd -ltss2-tcti-device -lssl modules/ssl/*.c
```

Figure A.3. Build of Apache with the patched module

The first step to proceed with the test of the main solution is to create the Sealed Key. This can be done by firstly creating an RSA key using the commands provided by Openssl Figure A.4.

Once the RSA Key has been created or if already been created, the next step consists on passing it to the actual piece of code responsible for creating the Sealed Key by accepting in input the key and providing in output the two files needed for the Unsealing process. The actual code can be found in the main directory `sealing`.

The code options are:

- -in or –input: path to the private key to be sealed;
- -prv or –private: path to the file that will contain the private part of the sealed key;
- -pub or –public: path to the file that will contain the public part of the sealed key;
- -v or –verbose: print on standard output information about the execution.

The following code is an example of an actual scenario:

```
1 openssl genrsa -out private_key.pem 2048
2 openssl req -key private_key.pem -new -out
     certificate_request.csr
3 openssl x509 -signkey private_key.pem -in
     certificate_request.csr -req -days 365 -out certificate.crt
```

Figure A.4. Build of Apache with the patched module

```
.\sealing -in private_rsa.key -prv
   \etc\ssl\private_sealed_key.dat -pub
   \etc\ssl\public_sealed_key.dat -v
```

The RSA key can be safely destroyed after this operation, by executing:

```
rm private_rsa.key
```

The second step is to change the configuration file of the ssl module httpd-ssl.conf found in `usr/local/apache2/conf/extra` by adding the following Directives:

- **SSLSealed** ON
- **SSLSealedPrivateFile** /etc/ssl/private_sealed_key.dat
- **SSLSealedPublicFile** /etc/ssl/public_sealed_key.dat

Apache can be executed using the default command: `sudo apachectl start`. If we want to launch it in debugging mode this command can be used:
`sudo apachectl -X`.

Appendix B

Developer's manual

B.1 Use Cases and Tpm2-OpenSSL solutions

In this section, we are going to present and explain the code developed for both the use cases and `tpm2-openssl` library. Because they were both written in C++ and used some common functions and structures, for readability purposes they were merged.

B.1.1 Structures

The following Figures B.1, B.2, B.3, B.4, B.5 represent the structures defined for both the implementations of the use cases and the changes made to the `tpm2-openssl` project. For each field, a comment is inserted.

B.1.2 Code

The following functions are used both in the use cases and TPM2-OpenSSL scenarios but also in the final solution developed with Apache. For this reason they

```
1  struct StartAuthSessionResult {
2      // Returning code to check if different from
3      // TPM2_RC_SUCCESS
4      int rc;
5      // Session handle
6      ESYS_TR handle;
7      // Nonce created after the creation of the Session
8      std::vector<uint8_t> nonce_tpm;
9  };
10
```

Figure B.1. Structure to store meaningful information of Sessions

```
struct LoadResult {
    // Returning code to check if different from
    // TPM2_RC_SUCCESS
    int rc;
    // Loaded object handle
    uint32_t handle;
    // Hash corresponding to the object name extracted
    // with the Esys_ReadPublic() function
    std::vector<uint8_t> name;
};

```

Figure B.2. Structure to store meaningful information of Keys/Sealed Objects

```
struct CreatePrimaryResult {
    // Returning code to check if different from
    // TPM2_RC_SUCCESS
    int rc;
    // Primary Key handle
    uint32_t handle;
    // RSA public key material
    std::vector<uint8_t> rsa_public_n;
    // ECC public key material
    std::vector<uint8_t> ecc_public_x;
    std::vector<uint8_t> ecc_public_y;
    int ecc_curve_id;
    // Symmetric key information
    std::vector<uint8_t> sym_cipher_buffer;
    // Hash corresponding to the name of the Primary Key
    std::vector<uint8_t> name;
    // Parent information
    std::vector<uint8_t> parent_name;
    std::vector<uint8_t> parent_qualified_name;
};

```

Figure B.3. Structure to store meaningful information of Primary Keys

will not be further discussed in the final section of this chapter. The only difference with the Apache functions is that they are written in `C` and not in `C++` but the structure and execution are equal.

The first pair of function that we are going to discuss are the TCTI Initialisation function Figure B.6 and the ESYS Initialisation function Figure B.7. Both of them

```
struct CreateResult {
    // Returning code to check if different from
    // TPM2_RC_SUCCESS
    int rc;
    // Private part of Key/Sealed Object to be
    // used with Load() function
    std::vector<uint8_t> tpm2b_private;
    // Public part of Key/Sealed Object to be
    // used with Load() function
    std::vector<uint8_t> tpm2b_public;
    // RSA public key material
    std::vector<uint8_t> rsa_public_n;
    // ECC public key material
    std::vector<uint8_t> ecc_public_x;
    std::vector<uint8_t> ecc_public_y;
    // Curve identifier
    int ecc_curve_id;
    // Parent information
    std::vector<uint8_t> parent_name;
    std::vector<uint8_t> parent_qualified_name;
};

```

Figure B.4. Structure to store meaningful information of Child Keys/Sealed Objects

```
struct SignResult {
    // Returning code to check if different from
    // TPM2_RC_SUCCESS
    int rc;
    // Signing algorithm used
    int sign_algo;
    // Hash algorithm used
    int hash_algo;
    // RSA signature
    std::vector<uint8_t> rsa_ssa_sig;
    // ECDSA signature
    std::vector<uint8_t> ecdsa_r;
    std::vector<uint8_t> ecdsa_s;
};

```

Figure B.5. Structure to store meaningful information of Signing Data

are used to initialise the context needed to send commands to the TPM and to use its functionalities. The contexts will be flushed once the program ends using the Finalisation functions Figures B.8 and B.9.

```
void TPM::Init_Tcti_Device_Context() {
    /* Variable configuration */
    ...

    /* Function call need to obtain the right size for the
   tcti_context */
    rc = Tss2_Tcti_Device_Init(nullptr, &context_size, NULL);
    /* Check returned value */

    /* Memory allocation for the tcti_context */
    tcti_context = (TSS2_TCTI_CONTEXT *) calloc(1,context_size);
    /* Check returned value */

    /* Initialisation of the tcti using the default configuration
    */
    rc = Tss2_Tcti_Device_Init(tcti_context,&context_size, NULL);
    /* Check returned value */
}

```

Figure B.6. TCTI initialisation using the in-kernel Resource Manager

```
void TPM::Finalize_Tcti_Device_Context() {
    /* Clear the tcti_context */
    Tss2_Tcti_Finalize(tcti_context);
}

```

Figure B.7. TCTI finalisation

The next function that has a key role in the solutions developed is the function responsible for the Session creation Figure B.10. This function initialise all the data structures and the properties needed to create a Session. It supports the creation of the three types of sessions: HMAC, Policy and trial.

The next two functions are used to create the Primary key. They were developed to be as flexible as possible, allowing the user to select different properties for the primary key. For this reason the first function Figure B.11 is used to create the data structures needed and to set the properties of the private key, while the second function Figure B.12 creates the Primary key.

The following function Figure B.13 is used to extract the policy digest from the Policy Session. This value is later used to create Sealed Data/Keys protected with

```
void TPM::Init_Esys_Context() {
    /* Initialization of the ESYS context */
    rc = Esys_Initialize(&esys_context,tcti_context,&abi_version)
    ;
    /* Check returned value */
}

```

Figure B.8. ESYS Context initialisation

```
void TPM::Finalize_Esys_Context() {
  Esys_Finalize(&esys_context);
}

```

Figure B.9. ESYS Context Finalisation

the PCRs.

The creation of the Sealed Data/Key is done using the following function Figure B.14. From the functional point of view, it is similar to the pair of functions for the primary key creation. The only difference with them is that some of the input parameters are different. The key point of this function is the usage of the policy digest to create an Object which is protected by the PCRs.

The next function Figure B.15 is needed to load inside the TPM the Objects created with the Create function. It accepts in input both the public and private data created during the creation of the Object. Then the operation of Unmarshalling is carried out on those data and then the Load operation is done. An handle to the object loaded inside the TPM is returned.

The following four functions are only used inside the Use cases. The first function Figure B.16 is used to encrypt a data blob received in input using a RSA private key protected with the PCRs. The second function Figure B.17 instead, is used to decrypt the encrypted data from the previous one using a Policy Session for the authorisation. The policy session is used to check if the machine is in the correct state. The next two functions implement the Sign Figure B.18 and Signature verification Figure B.19. The key point is that the Sign function as the Decrypt function must use a Policy session to be authorised to use the private key.

The final function is the one used for the Unsealing process and it is used only in the Apache solution. It receives in input an handle to the Sealed data loaded inside the TPM. The Unsealing operation consist on retrieving the sensitive data stored inside the sealed data only if the platform is in the correct state. In our case the check is done on the value of the PCR selected.

```
1 StartAuthSessionResult TPM::StartAuthSession(int session_type,
    bool is_symmetric, uint32_t handle) {
2
3     /* TPM resources allocation and Session parameter
    defininition */
4     ...
5
6     /* Function that implements the Session creation */
7     result.rc = Esys_StartAuthSession(esys_context, handle,
    ESYS_TR_NONE,
8                                       ESYS_TR_NONE, ESYS_TR_NONE,
     ESYS_TR_NONE, &nonce_caller,
9                                       sessionType, &symmetric,
    authHash, &session);
10
11    /* Check Session Creation */
12    ...
13
14    /* Function to set up the attributes the Session must have */
15    result.rc = Esys_TRSess_SetAttributes(esys_context, session,
    sessionAttributes, 0xff);
16    return result;
17 }
18
```

Figure B.10. Session creation

B.1.3 TPM2-OpenSSL code changes

The main changes to the `tpm2-openssl` code were made in the `tpm2-provider-keymgmt-rsa.c` file. The first step Figure B.21 is to define the variables needed for the creation of the Private Key which will be dependent on the 23rd PCR value. Those variables are mainly used to save the current handlers of the sessions created. The second step is to modify the function responsible for the Primary Key creation that in our case will contain also a reference to the HMAC session handler in this way we can later pass it to the function responsible for creating the Private Key.

Once the Primary Key is created, the next step Figure B.22 is to start the Policy Session responsible for creating the policy digest. After the extraction of the policy digest, we need to modify the sensible data that will be encrypted during the key creation passing the value just computed. In this way, the Private Key created will be dependent on the current value of the PCR 23.

```
CreatePrimaryResult TPM::CreatePrimary(int hierarchy, int type,
    int restricted, int decrypt,
                                       int sign, const std::
    string &unique,
                                       const std::string &
    user_auth,
                                       const std::string &
    sensitive_data,
                                       const std::vector<uint8_t>
     &auth_policy,
                                       int session_handle) {
    /* Check on the key hierarchy selected */
    ...
    /* Check if the type of key selected is supported */
    ...
    /* Creation of the public data containing key properties */
    ...
    /* Creation of the sensitive data containing authorisation
    properties */
    ...
    return CreatePrimaryFromTemplate(hierarchy, in_sensitive,
    in_public, session_handle);
}

```

Figure B.11. Creation of the data structures for the Primary key creation

B.2 Apache solution

The solution developed for the Apache web server mainly consists of using the mod_ssl module, which is already provided by Apache, to integrate the operations needed to use TPM 2.0. An external procedure has been developed for the Sealing operation.

B.2.1 Sealing procedure

Structures

The data structures defined are already described in the first section of this chapter so no other analyses will be made, except the following one Figure B.23. The `SealedData` structure has been introduced to store information about the data read from file and the Sealed Private key just created.

The code developed to handle the Sealing of the Private Key is symmetrical to what will be analysed in the next section. As previously explained in the User manual, this procedure takes as input parameters the paths to the following files: private key, public part, and private part.

```
CreatePrimaryResult
TPM::CreatePrimaryFromTemplate(ESYS_TR hierarchy,
                               const TPM2B_SENSITIVE_CREATE &
    in_sensitive,
                               const TPM2B_PUBLIC &in_public,
                               int session_handle) {
    ...

    /* Creation fo the primary key */
    result.rc = Esys_CreatePrimary(esys_context, hierarchy,
    session_handle, ESYS_TR_NONE, ESYS_TR_NONE, &in_sensitive, &
    in_public, &outside_info,
                                   &creation_pcr, &result.handle,
     &out_public, &creation_data, &creation_hash, &creation_ticket
    );

    /* Store the primary key information inside the
    CreatePrimaryResult data structure */
    ...

    return result;
}

```

Figure B.12. Creation of the Primary key

The first phase involves reading the private key from a file and dividing it into blocks of data of 128 bytes each Figure B.24. This process is strictly necessary because the library used allows the creation of Sealed Objects with a maximum size of 128 bytes.Subsequently, the necessary variables are initialised to manage this process.

Sealing code

The first operation consists on creating an HMAC Session Figure B.25 to be used to generate the primary key. Another session, of type Policy, is then created to extract the policy digest that need to be associated to the Sealed Key, which is dependent to the PCR value defined.

In the process of creating a Sealed Object, the primary key's role is to protect the object by encrypting the sensitive information that needs to be saved in the file containing the private part Figure B.26.

The following block of code Figure B.27 manages the creation of `N`-different Sealed Objects, using the primary key handle, the policy digest previously computed and the HMAC session.

Once the Sealing process is finished Figure B.28, for each Sealed object we

```
1 TPM2B_DIGEST* TPM::PolicyPCR(ESYS_TR pcr_handle, ESYS_TR
    session_handle, const std::vector<uint8_t> &digest){
2     /* Selection of the right PCR */
3
4     /* Function to extend the session policy and bind it to the
    PCR selected */
5     TSS2_RC rc =  Esys_PolicyPCR(esys_context, session_handle,
    ESYS_TR_NONE, ESYS_TR_NONE, ESYS_TR_NONE,
6                                  &pcr_digest_zero, &pcrSelection)
    ;
7     /* Check returned value */
8     ...
9     /* Extraction of the policy digest */
10    r = Esys_PolicyGetDigest(esys_context,
11                             session_handle,
12                             ESYS_TR_NONE,
13                             ESYS_TR_NONE, ESYS_TR_NONE, &
    policyDigest);
14    /* Check returned value */
15    ...
16    return policyDigest;
17 }
18
```

Figure B.13. Creation of the policy digest

extract the values of `tpm2b_public` and `tpm2b_private` and concatenate these values to create two files containing the public and private parts.

In the last phase Figure B.29, the values are written to the file that has been provided in input and the flush of both sessions happens.

B.2.2 mod_ssl code

Structures

The first changes made to the code involved defining new variables and data structures in the `ssl_private.h` file. The most important one is related to the enum variable, ssl_sealed_t Figure B.30, which distinguishes whether the server will use a Sealed key or a key independent of the TPM context. In addition, some global variables were defined for the Unsealing process Figure B.31.

This was made possible by defining a new data structure called tpm2_context_t Figure B.31, containing the necessary fields to manage instances of both the TCTI context and the ESYS context. This structure was added to the `SSLSrvConfigRec` Figure B.33structure already defined, which contains all the meaningful information on the current server's configuration.

```
CreateResult TPM::Create(uint32_t parent_handle, int type, int restricted,
                         int decrypt, int sign, const std::string &user_auth,
                         const std::string &sensitive_data,
                         const std::vector<uint8_t> &auth_policy,
                         int session_handle) {
    /* Creation of the data structures needed to create the Sealed Data/Key */
    ...
    /* Sealed Data/Key creation dependent to the PCR value */
    result.rc = Esys_Create(esys_context, parent_handle, session_handle,
                            ESYS_TR_NONE, ESYS_TR_NONE, &in_sensitive,
                            &in_public, &outside_info, &creation_pcr,
                            &out_private, &out_public, &creation_data,
                            &creation_hash, &creation_ticket);
    /* Store meaningful information about the key inside the CreateResult data structure */
    ...
    return result;
}

```

Figure B.14. Creation of the Sealed Data/Key

Lastly, three new Directives were defined in the `mod_ssl.c` file. These new entries defined in the `command_rec` structure Figure B.34, containing all the directives defined by mod_ssl, contain the callback to the function for that precise Directive.

Functions

The first step is to allocate the necessary memory for the previously defined data structures. In this context, the default method apr_palloc() defined by Apache was used Figure B.35 and the variables were initiated to their default values.

In addition, the merge function used to copy the configuration of the current server instance to the new one was extended to support the new structure defined for TPM purposes Figure B.36.

As previously mentioned, three new functions were defined to correctly manage the directives created. The first function Figure B.37 manages the `SSLSealed` directive, it receives in input two possible values `On` or `Off` and based on the value received the `Sealed` process is enabled.

```
LoadResult TPM::Load(ESYS_TR parent_handle, const std::vector<
    uint8_t> &tpm2b_private, const std::vector<uint8_t> &
    tpm2b_public, int session_handle) {
    /* Unmarshalling of the public and private part of Sealed
    Data/Key */
    ...
    /* Load the Sealed Data/Key inside the TPM */
    result.rc = Esys_Load(esys_context, parent_handle,
    session_handle, ESYS_TR_NONE, ESYS_TR_NONE, &in_private, &
    in_public, &result.handle);
    /*Check returned value */
    ...
    return result;
}

```

Figure B.15. Load the Sealed Data/Key inside the TPM

```
std::vector<uint8_t> TPM::EncryptRSAWithSession(uint32_t
    key_handle,
                                                const std::vector
    <uint8_t> &message,
                                                uint32_t
    session_handle) {
    /* Data structure initialisation */
    ...
    /* Encryption of the data received using the private key */
    rc = Esys_RSA_Encrypt(esys_context, key_handle,
    session_handle, ESYS_TR_NONE,
                          ESYS_TR_NONE, &inData, &scheme, &
    outsideInfo, &data_out);
    /* Check returned value */
    ...
    return std::vector<uint8_t>(data_out->buffer, data_out->
    buffer + data_out->size);
}

```

Figure B.16. Encryption of the data blob with the Private key

The second function Figure B.38 manages the `SSLSealedPublicFile` directive, it receives in input the path to the file containing the public parts of the Sealed Private Key.

The third function Figure B.39 manages the `SSLSealedPrivateFile` directive, it receives in input the path to the file containing the private parts of the Sealed

```
std::vector<uint8_t> TPM::DecryptRSAWithSession(uint32_t
    key_handle,
                                                const std::vector
    <uint8_t> &message,
                                                uint32_t
    session_handle) {
     /* Initialisation of the data structure needed to store the
    decrypted data */
     ...
     /* Decryption of the encrypted data using the policy session
    for authentication */
     rc = Esys_RSA_Decrypt(esys_context, key_handle,
    session_handle, ESYS_TR_NONE,
                           ESYS_TR_NONE, &inData, &scheme, &
    outsideInfo, &data_out);
     /* Check returned code */
     ...
     return std::vector<uint8_t>(data_out->buffer, data_out->
    buffer + data_out->size);
}

```

Figure B.17. Decryption of the encrypted data

```
SignResult TPM::Sign(uint32_t key_handle, int type, const std::
    string &str, uint32_t session_handle) {
     /* Check if the algorithm selected is supported and
    initialisation of the data structures */
     ...
     /* Signing of the data received */
     result.rc = Esys_Sign(esys_context, key_handle,
    session_handle,
                           ESYS_TR_NONE, ESYS_TR_NONE, &message, &
    scheme, &validation, &signature);
     /* Check returned value and store the signature */
     return result;
}

```

Figure B.18. Signing the data blob

Private Key.

The following function Figure B.40 manages the initialisation of the SSL Context for the current server. It retrieves the Certificate from the file defined in the configuration, extracts the Private Key associated to the certificate and checks it

```
1 TPM2_RC TPM::VerifySignature(uint32_t key_handle, const std::
    string &str, const SignResult &in_signature, uint32_t
    session_handle) {
2     /* Check if the algorithm selected is supported and
    initialisation of the data structures */
3     ...
4     /* Verification of the signature */
5     return Esys_VerifySignature(esys_context, key_handle,
    ESYS_TR_NONE, ESYS_TR_NONE, ESYS_TR_NONE, &message, &signature
    , &validation);
6 }
7
```

Figure B.19. Verification of the signature

```
1 std::vector<char> TPM::Unseal(uint32_t keyHandle, uint32_t
    sessionHandle){
2     /* Unsealing of the Sealed data */
3     TSS2_RC r = Esys_Unseal(esys_context, keyHandle,
    sessionHandle,
4                             ESYS_TR_NONE, ESYS_TR_NONE, &outData)
    ;
5     /* Store the unsealed data into a buffer of bytes */
6     return sensitive_data;
7 }
8
```

Figure B.20. Unsealing sealed data

authenticity against the key. The main changes are made during in the extraction stage in which, based on the `sealed` flag previously set, the Unsealing operation is executed instead of the extraction from file.

The function that manages all the operation to Unseal the private key starting from initialisation to the finalisation of the TPM is the `ssl_unsealing_tpm2()`. For readability purposes the function code has been divided in many parts and the code responsible for error checking has been removed. The functions in these code snippets have already been addressed in the first section of this chapter and so they will not be further analysed. The only difference is that to be compliant with the Apache source code, the code has been converted from `C++` to `C`.

The first block of code Figure B.41 represent the definition and initialisation of the variables used in the function. As it can be seen, the majority of the defined variables are specified in the *tss2* library, to adhere to the standard as much as possible without creating new data structures.

The next code snippet Figure B.42 represent the process of retrieving the public and private part of the Sealed Key. This two values are later used to Unseal the

```
1  static void *tpm2_rsa_keymgmt_gen(void *ctx, OSSL_CALLBACK *cb,
2      void *cbarg){
3      ...
4      ESYS_TR hmac_handle;
5      ESYS_TR policy_handle;
6      ...
7      if (gen->parentHandle && gen->parentHandle != TPM2_RH_OWNER)
       {
8          ...
9      } else {
10         DBG("RSA GEN parent: primary 0x%x\n", TPM2_RH_OWNER);
11         if (!tpm2_build_primary(pkey->core, pkey->esys_ctx,
12                 pkey->capability.algorithms, ESYS_TR_RH_OWNER,
13                 &gen->parentAuth, &parent, &hmac_handle))
14             goto error;
15     }
16
```

Figure B.21. Caption

Private Key. The first call to `readFileToVectorUnseal()` extract the public part by reading blocks of 616 bytes. The second call extract the private part by reading blocks of 1552 bytes.

The next phase is to initialise the TPM Figure B.43. In this stage both the TCTI and Esys context are initialised.

In the following stage Figure B.44, the HMAC session is created and it is kept until the Unsealing process is finished, because it is used for both the Primary Key creation and later on for the Load operation. As mentioned, the Primary Key creation is one of the steps needed to Load and Unseal the Private key.

The core of this function is the following code snippet Figure B.45. As it is possible to see, the process of Unsealing consist on iterating on the actual number of sealed chucks.

The operations can be divided in four phases:

1. First phase: we have to start a Policy Session in order to be able to retrieve the current policy digest which will be checked against the one computed during the Sealing process.

2. Second phase: consists on Loading the public chuck and the private chuck of the sealed object. This is necessary to be able to Unseal the current sealed object.

3. Third phase: the Unsealing function is called passing the handle to the loaded object and the handle to the policy session, if the PCR value is different from

```
    /* Start a new session of type POLICY this time */
    r = tpm2_start_auth_session(2, gen->esys_ctx, 1,
            &policy_handle);
    /* Check on Policy Session Creation */
    ...

    /* Create the policy session depending on the value of the
    PCR-23 */
    TPM2B_DIGEST policy_digest = {
        .size = 0,
        .buffer = {}
    };

    r = tpm2_create_policy_digest(23, gen->esys_ctx,
            policy_handle, &policy_digest);

    /* Change the inSensitive field, in order to set the value of
    the authentication policy equal to the policy digest */

    gen->inPublic.publicArea.authPolicy.size =
        policy_digest.size;
    memcpy(gen->inPublic.publicArea.authPolicy.buffer,
        policy_digest.buffer, policy_digest.size);

    ...
}

```

Figure B.22. Caption

```
struct SealedData{
    // Used to store the Private Key read from file
    map<int, vector<uint8_t>> data_blobs;
    // Used to store informations about the Sealed Objects
    // It contains both information about public and private
    CreateResult *sealed_objects;
    // Number of Sealed blobs created
    int blobs;
};

```

Figure B.23. Caption

```
void Sealing(TPM *tpm, string inputPath, string privateKeyPath,
                string publicKeyPath, bool verbose) {

    SealedData sealed_data;
    vector<uint8_t> public_part;
    vector<uint8_t> private_part;
    int current_chunk= 0;

    /* Read the file and store the content in a vector */
    readFileToVectorKey(inputPath, &sealed_data);

```

Figure B.24. Caption

```
/* Start HMAC Session */
StartAuthSessionResult temporarySession =
    tpm->StartAuthSession(3, true, ESYS_TR_NONE);

/* Check HMAC Session Creation */
...

/* Start Policy Session */
StartAuthSessionResult policySession =
    tpm->StartAuthSession(2, true, ESYS_TR_NONE);

/* Check Policy Session Creation */
...

/* Policy Digest extraction */
TPM2B_DIGEST *policyDigest = tpm->PolicyPCR(23,
    policySession.handle, vector<uint8_t>());
vector<uint8_t> digest = vector<uint8_t>(
    policyDigest->buffer,
    policyDigest->buffer+policyDigest->size);

```

Figure B.25. Caption

the one during the Sealing operation an error is returned in this stage and the Unsealing operation is aborted.

4. Last phase: it is mandatory to Flush the loaded object temporary stored in the TPM and the current policy session. This is needed to be able to continue

```
/* Create Primary Key */
CreatePrimaryResult pk = tpm->CreatePrimary(ESYS_TR_RH_OWNER,
    TPM2_ALG_RSA, 1, 1, 0, "", "","", vector<uint8_t>(),
    temporarySession.handle);

/* Check Primary Key Creation */
...

/* Allocate the memory to store the Sealed data objects */
sealed_data.sealed_objects = (CreateResult *) calloc(
    sealed_data.blobs, sizeof (CreateResult));

```

Figure B.26. Caption

```
    while (current_chunk < sealed_data.blobs){
        /* Sealed Object Creation */
        CreateResult sealed_object = tpm->Create(pk.handle,
            TPM2_ALG_KEYEDHASH, 0, 0, 0, "",
    string(sealed_data.data_blobs[current_chunk].begin(),
            sealed_data.data_blobs[current_chunk].end()), digest,
            temporarySession.handle);

        /* Check Sealed Object Creation */
        ...

        sealed_data.sealed_objects[current_chunk++] =
    sealed_object;
    }

```

Figure B.27. Caption

with the Unsealing process. In practice if we use the same policy session what happens is that the policy digest is computed each. This causes the Unsealing process to fail after the first iteration.

The following code Figure B.46 is used to align the unsealed data size to its actual size in this way we avoid problems when serialising the Private Key from the buffer to the EVP_PKEY data structure. It follows the flush process for both the Primary Key and the HMAC session and finally the finalisation of both the ESYS and TCTI context.

```
1     current_chunk = 0;
2     /* Fill the public_part and private_part with the
      informations needed */
3     while(current_chunk < sealed_data.blobs){
4         public_part.insert(public_part.begin(),
5             sealed_data.sealed_objects[current_chunk].
      tpm2b_public.begin(), sealed_data.sealed_objects[current_chunk
      ].tpm2b_public.end());
6
7         private_part.insert(private_part.begin(),
8             sealed_data.sealed_objects[current_chunk].
      tpm2b_private.begin(), sealed_data.sealed_objects[
      current_chunk].tpm2b_private.end());
9         current_chunk++;
10    }
11
```

Figure B.28. Caption

```
1     /* Write the Public and Private part of the Sealed Private
      Key to file */
2     writeFileInBinary(publicKeyPath, public_part);
3     writeFileInBinary(privateKeyPath, private_part);
4
5     /* Flush both the HMAC Session and the Policy Session
6     tpm->FlushContext(temporarySession.handle);
7     tpm->FlushContext(policySession.handle);
8
```

Figure B.29. Caption

The last phase Figure B.47 is to actually create the key from the buffer containing the Unsealed Private Key. This process is done using the OpenSSL library, using the BIO data type that is an Input/Output abstraction to hide I/O details from the application. Once the BIO memory buffer is created it is possible to proceed to create the EVP_PKEY, which will be later associated to the current SSL_CTX of the server instance.

```
/**
 * Define the Sealed enabled state
    - UNSET = Default value if not defined in the
              configuration file
    - 0     = Private Key Not-Sealed
    - 1     = Private Key Sealed
*/
typedef enum{
    SSL_SEALED_UNSET = UNSET,
    SSL_SEALED_FALSE = 0,
    SSL_SEALED_TRUE = 1
} ssl_sealed_t;

```

Figure B.30. Directives Variables

```
/**
 * TPM2 Global Variables
    - PUB_SIZE  = Default value of the public part of the
                  Sealed Object
    - PRI_SIZE  = Default value of the private part of
                  the Sealed Object
    - BLOB_SIZE = Default value of the MAX size of a
                  Sealed Object
    - DEBUG_TPM2= Used to print meaningful informations

*/
#define TPM2_PUB_SIZE 616
#define TPM2_PRI_SIZE 1552
#define BLOB_SIZE 128
#define DEBUG_TPM2

```

Figure B.31. Global Variables

```
typedef struct{
    \\ Used to save the current TCTI context, once the
    \\ TPM is Initiated
    TSS2_TCTI_CONTEXT *tcti_context;
    \\ Used to save the current ESYS context
    ESYS_CONTEXT      *esys_context;
    \\ Used to the save the current path to the file
    \\ containing the public parts of the Sealed Object
    const char      *public_path;
    \\ Used to the save the current path to the file
    \\ containing the private parts of the Sealed Object
    const char      *private_path;
} tpm2_context_t;

```

Figure B.32. tpm2_context_t structure

```
struct SSLSrvConfigRec {
    ...
    tpm2_context_t  *tpm2;
    ssl_sealed_t     sealed;
    ...
};

```

Figure B.33. SSLSrvConfigRec structure

```
static const command_rec ssl_config_cmds[] = {
    ...
    SSL_CMD_SRV(Sealed, TAKE1,
            "Enable Sealed mode ('on', 'off')")
    SSL_CMD_SRV(SealedPublicFile, TAKE1,
            "TPM2 Public Part Sealed Private Key ")
    SSL_CMD_SRV(SealedPrivateFile, TAKE1,
            "TPM2 Private Part Sealed Private Key ")
        ...
}

```

Figure B.34. Definition of the functions corresponding to the directives created

```
1 static void modssl_ctx_init_tpm2(SSLSrvConfigRec *sc,
2                                  apr_pool_t *p)
3 {
4     tpm2_context_t *tpm2ctx;
5     tpm2ctx = sc->tpm2 = apr_palloc(p,
6                                  sizeof(*sc->tpm2));
7     tpm2ctx->tcti_context = NULL;
8     tpm2ctx->esys_context = NULL;
9     tpm2ctx->public_path = NULL;
10    tpm2ctx->private_path = NULL;
11 }
12
```

Figure B.35. Initialisation of the data structure created

```
1 static void modssl_ctx_cfg_merge_tpm2(apr_pool_t *p,
2         tpm2_context_t *base, tpm2_context_t *add,
3         tpm2_context_t *mrg){
4     cfgMerge(tcti_context, NULL);
5     cfgMerge(esys_context, NULL);
6     cfgMergeString(public_path);
7     cfgMergeString(private_path);
8 }
9
```

Figure B.36. Merge of the data structures from the previous configuration to the new one

```
const char *ssl_cmd_SSLSealed(cmd_parms *cmd, void *dcfg,
                                const char *arg){

    SSLSrvConfigRec *sc = mySrvConfig(cmd->server);
    if(!strcasecmp(arg, "On")){
        sc->sealed = SSL_SEALED_TRUE;
        return NULL;
    }
    else if(!strcasecmp(arg, "Off")){
        sc->sealed = SSL_SEALED_FALSE;
        return NULL;
    }
    return "Argument must be On or Off";
}

```

Figure B.37. Selector for enabling the Unsealing process

```
const char *ssl_cmd_SSLSealedPublicFile(cmd_parms *cmd,
                                void *dcfg,
                                const char *arg)
{
    SSLSrvConfigRec *sc = mySrvConfig(cmd->server);
    const char *err;

    if (arg!=NULL){
        sc->tpm2->public_path = arg;
    }else{
        return apr_pstrcat(cmd->pool, cmd->cmd->name,
                        ": file '", *arg,
                        "' does not exist or is empty", NULL);
    }

    return NULL;
}

```

Figure B.38. Store the path to the public part of the sealed key

```
const char *ssl_cmd_SSLSealedPrivateFile(cmd_parms *cmd,
                                 void *dcfg,
                                 const char *arg)
{
    SSLSrvConfigRec *sc = mySrvConfig(cmd->server);
    const char *err;

    if (arg!=NULL){
        sc->tpm2->private_path = arg;
    }else{
        return apr_pstrcat(cmd->pool, cmd->cmd->name,
                        ": file '", *arg,
                        "' does not exist or is empty", NULL);
    }

    return NULL;
}

```

Figure B.39. Store the path to the private part of the sealed key

```
static apr_status_t ssl_init_server_certs(server_rec *s,
        apr_pool_t *p, apr_pool_t *ptemp, modssl_ctx_t *mctx,
        apr_array_header_t *pphrases, tpm2_context_t *tpm2ctx,
        ssl_sealed_t sealed)
{
    ...
        else if ((sealed == SSL_SEALED_FALSE &&
                    SSL_CTX_use_PrivateKey_file(mctx->ssl_ctx,
                        keyfile, SSL_FILETYPE_PEM) < 1) &&
                    CHECK_PRIVKEY_ERROR(ERR_peek_last_error())) {
                ...
            }
            else if ((sealed == SSL_SEALED_TRUE
                        && tpm2ctx->public_path != NULL
                        && tpm2ctx->private_path != NULL)){

                pkey = ssl_unsealing_tpm2(tpm2ctx);

                if (SSL_CTX_use_PrivateKey(mctx->ssl_ctx,
                                           pkey) != 1){
                    fprintf(stderr, "Couldn't use the Private
                                Key\n");
                    return APR_EGENERAL;
                }else{
                    fprintf(stdout, "Everything fine\n");
                }
            }
        }
    ...
}

```

Figure B.40. Loading the key inside the SSL_CTX using the TPM

```
EVP_PKEY * ssl_unsealing_tpm2(tpm2_context_t *tpm2ctx){
    // Used to save temporarly the TCTI Context
    TSS2_TCTI_CONTEXT *tcti_context;
    // Used to save temporarly the ESYS Context
    ESYS_CONTEXT *esys_context;
    // By default is Empty but can be used as an entropy value
    TPM2B_DIGEST unique = {};
    // Used to set a password for key usage
    TPM2B_DIGEST user_auth = {};
    // Used to define the data to protect as in the Sealing
    // process
    TPM2B_SENSITIVE_DATA sensitive_data = {};
    // Used to save the authentication policy if defined
    TPM2B_DIGEST auth_policy = {};
    // Used as an empty policy digest
    TPM2B_DIGEST empty_digest = {};
    TPM2B_DIGEST *policy_digest;
    // Store the Public Part of the Sealed Object
    struct SealedData public_part;
    // Store the Private Part of the Sealed Object
    struct SealedData private_part;
    int current_chunk = 0;
    int current_buffer_position = 0;

    uint8_t *unsealed_data = (uint8_t *) calloc(private_part.
    blobs, BLOB_SIZE);

```

Figure B.41. Variables definition

```
/* Read Public Part */
public_part = readFileToVectorUnseal(tpm2ctx->public_path, TRUE);

/* Read Private Part */
private_part = readFileToVectorUnseal(tpm2ctx->private_path,
    FALSE);

/* Check on file read */
...

current_chunk = private_part.blobs-1;
```

Figure B.42. Read of the files containing the public and private part

```
tpm2ctx->tcti_context = Init_Tcti_Tabrmd_Context();

tpm2ctx->esys_context = Init_Esys_Context();

```

Figure B.43. Initialisation of the TCTI and ESYS context

```
/* Start HMAC Session */
struct StartAuthSessionResult hmac_session = StartAuthSession(3,
    TRUE, ESYS_TR_NONE, tpm2ctx->esys_context);

/* Check Session Creation */
...

/* Create Primary Key */
struct CreatePrimaryResult primaryKey = CreatePrimary(
    ESYS_TR_RH_OWNER, TPM2_ALG_RSA, TRUE, TRUE, FALSE, "",
    user_auth, sensitive_data, auth_policy, hmac_session.handle,
    tpm2ctx->esys_context);

/* Check Primary Key Creation */
...

```

Figure B.44. Creation of the HMAC session and Primary key

```
/* Unsealing process */
while(current_chunk >= 0){
    /* Start Policy Session */
    struct StartAuthSessionResult policy_session_temp =
        StartAuthSession(2, TRUE, ESYS_TR_NONE,
            tpm2ctx->esys_context);

    /* Check Policy Session Creation */
    ...

    /* Policy Digest Creation */
    policy_digest = PolicyPCR(23, policy_session_temp.handle,
        empty_digest, tpm2ctx->esys_context);

    /* Check Policy Digest Creation */
    ...

    /* Load current Sealed Object */
    struct LoadResult loaded = Load(primaryKey.handle,
        private_part.tpm2b_part[current_chunk],
        public_part.tpm2b_part[current_chunk],
        hmac_session.handle, tpm2ctx->esys_context);

    /* Unseal current Loaded Object */
    struct UnsealedDataTmp tmp_unsealed_data = Unseal(
        loaded.handle, policy_session_temp.handle,
        tpm2ctx->esys_context);

    for (int i=0; i<tmp_unsealed_data.size; i++){
        unsealed_data[current_buffer_position++] =
            tmp_unsealed_data.unsealed_data_tmp[i];
    }

    current_chunk--;

    /* Flush both the Loaded Object and the current Policy
   Session */
    ...
}

```

Figure B.45. Unsealing process

```
/* Fix the Unsealed data to the right size */
uint8_t *final_unsealed_data = (uint8_t *) realloc(unsealed_data,
    current_buffer_position);

/* Flush both the primary key and the HMAC session */
...

/* Finalise both the TCTI and the ESYS contexts */
...

```

Figure B.46. Flush of the TPM resources

```
    /* Creation of the EVP_PKEY */
    BIO *bio;
    EVP_PKEY *pkey = NULL;

    bio = BIO_new_mem_buf(final_unsealed_data,
                          current_buffer_position);

    /* Check on BIO Creation */
    ...

    pkey = PEM_read_bio_PrivateKey(bio, &pkey, NULL, NULL);

    /* Check on EVP_PKEY Creation */
    ...

    BIO_free(bio);
    free(final_unsealed_data);
    return pkey;
}

```

Figure B.47. Creation of the EVP_PKEY from the unsealed data buffer