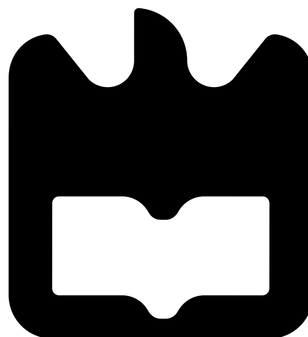
Universidade de Aveiro
2021

Manuel Augusto Capão Estrela Santos

SmartCMD: acesso à Chave Móvel Digital através de um smartcard virtual

SmartCMD: access to the Digital Mobile Key via a virtual smartcard



Universidade de Aveiro
2021

Manuel Augusto Capão Estrela Santos

SmartCMD: acesso à Chave Móvel Digital através de um smartcard virtual

SmartCMD: access to the Digital Mobile Key via a virtual smartcard

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica de André Ventura da Cruz Marnôto Zúquete, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e de João Paulo Silva Barraca, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente / president	**Professor Doutor José Luis Guimarães Oliveira** Professor Catedrático da Universidade de Aveiro (por delegação do Reitor da Universidade de Aveiro)
vogais / examiners committee	**Professor Doutor Manuel Eduardo Carvalho Duarte Correia** Professor Associado, Universidade do Porto - Faculdade de Ciências
	Professor Doutor André Ventura da Cruz Marnoto Zúquete Professor Auxiliar, Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Agradeço primeiramente ao Prof. André Zúquete e Prof. João Paulo Barraca pela oportunidade, suporte, e disponibilidade no desenvolvimento desta dissertação.

Gostava de agradecer especialmente ao Prof. André Zúquete que no decorrer deste desafio superou qualquer expectativa que pudesse ser imaginável. Fico muito grato por todos os ensinamentos e mentalidades transmitidas. Tomou um papel essencial durante toda a extensão deste ano, sempre com uma palavra amiga e incentivadora.

À minha familia, especialmente ao meu pai e à minha mãe. Sem o apoio e ajuda deles ao longo destes anos nada disto teria sido possível.

Quero deixar também uma agradecimento especial à minha namorada Rafaela Patinha pela paciência que teve comigo, pelos momentos de companheirismo, pela motivação incondicional e por todo o suporte me deu durante todo este processo nunca desistindo de mim, assim como a todos aqueles que de alguma maneira influenciaram positivamente este resultado.

Palavras Chave

CSP, Chave Móvel Digital, SmartCard, Assinatura digital, Chaves privadas, PKCS#11, SOAP, Interface Standard, API, Algoritmos de hash, Sistemas cloud

Resumo

Hoje em dia, as soluções de assinatura baseadas em cloud estão em crescimento, de modo a oferecer às pessoas uma maneira mais prática de assinar os seus documentos do dia-a-dia. O Estado português disponibiliza aos seus cidadãos um serviço chamado CMD, que pode ser usado para criar assinaturas digitais sem o uso nem do Cartão de Cidadão nem de um leitor de cartões.
A CMD tem inúmeras vantagens e constitui um enorme avanço a nível de usabilidade de operações criptográficas, mas actualmente o uso da CMD esta extremamente limitado visto que apenas pode ser usada por software providenciado ou certificado pela AMA como por exemplo a aplicação "autenticação.GOV" para desktop ou a correspondente extensão para o browser ou então, no caso de um sistema Windows, pode ser usado através da Microsoft CAPI recorrendo ao CSP providenciado. No caso de um sistema Unix como é o caso do Linux ou do Mac OS, ou aplicações não nativas de um sistema Windows, não é possível utilizar a CMD uma vez que a AMA não providencia nenhuma interface standard dedicada à integração da mesma. Tal limita a escolha dos consumidores, que muitas vezes preferem utilizar ferramentas alternativas, como é o caso do Adobe Acrobat Reader, PDF Studio, Mozilla Thunderbird (para assinar emails usando uma assinatura digital), entre outros.

Nesta tese, e como solução para este problema, irá ser apresentada uma prova de conceito com o objectivo principal de desenvolver e validar em Linux um módulo PKCS#11 capaz de usar a CMD para executar assinaturas digitais compatíveis com aplicações usadas globalmente e desta forma aumentar o numero de opções oferecidas aos utilizadores desta ferramenta. Adicionalmente, o modulo PKCS#11 desenvolvido permite explorar em conjunto as funcionalidades da CMD e do CC.

Keywords CSP, Digital Mobile Key (CMD), SmartCard, PKCS#11, Standard Interface, API, Cloud-based, Digital Signatures, SOAP, Asymmetric Cryptography, Hash Algorithms

Abstract Nowadays, cloud-based signing solutions are on the rise to offer people a more practical way to sign their everyday documents. The Portuguese state provides its citizens with a service called CMD, which can be used to create digital signatures without using either a citizen card or a card reader.

CMD has numerous advantages and is a huge advance in the usability of cryptographic operations, but currently the use of CMD is extremely limited as it can only be used by software provided or certified by AMA such as the "authentication.GOV" desktop application or the corresponding browser extension or, in the case of a Windows system, it can be used via Microsoft CAPI using the provided CSP. In the case of a Unix system such as Linux or Mac OS, or applications not native to a Windows system, it is not possible to use CMD as the AMA does not provide any standard interface dedicated to its integration. This limits the consumers choice, who often prefer to use alternative tools, such as Adobe Acrobat Reader, PDF Studio, Mozilla Thunderbird (to sign emails using a digital signature), among others.

In this thesis, a solution was designed following the lines of a proof of concept that will be presented in the following chapters, with the main objective of developing and validating in Linux a PKCS#11 module capable of using CMD in order to execute digital signatures compatible with applications used globally, and thus increase the number of options offered to end users. Additionally, the developed PKCS#11 module allows the use of both CMD and CC functionalities.

Contents

Contents i

List of Figures iii

List of Tables v

Acronyms vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem . . . 2
- 1.3 Contribution . . . 2
- 1.4 Dissertation Structure . . . 3

2 Context 5
- 2.1 Asymmetric Cryptography & Cryptographic Tokens . . . 5
- 2.2 PKCS#11 Standard API . . . 6
- 2.3 Portuguese cryptographic devices and middleware . . . 10
  - 2.3.1 Citizen card (CC) . . . 10
  - 2.3.2 Chave Móvel Digital (CMD) . . . 11
  - 2.3.3 Middleware and drivers . . . 12
- 2.4 Library wrapping . . . 14

3 Background & Studies 17
- 3.1 Cloud-based Digital Signatures . . . 17
- 3.2 Digital Signature APIs . . . 18
  - 3.2.1 PKCS#11 API . . . 18
  - 3.2.2 Microsoft Cryptography API (CAPI) . . . 19
  - 3.2.3 Cryptographic Service Provider (CSP) . . . 19
  - 3.2.4 Microsoft Cryptography API: Next Generation(CNG) . . . 20
  - 3.2.5 Key Storage Provider (KSP) . . . 21
- 3.3 Cloud-based product solutions . . . 22
  - 3.3.1 Austrian Mobile Phone Signature . . . 22
  - 3.3.2 Crypthomatic Signer and Crypto Service Gateway . . . 23

4 Smart CMD **27**
4.1 Possible approaches . . . 27
4.2 Architecture . . . 27
4.3 Implementation . . . 33
4.3.1 Installation and configurations . . . 33
4.3.2 PKCS#11 module . . . 33
4.3.3 Python module . . . 37
4.3.4 IPC (Inter-process communication) . . . 38
4.3.5 GUI (Graphical user interface) . . . 38

5 Tests & results **41**
5.1 OpenSC pkcs11-tool . . . 42
5.2 Autenticação.GOV application . . . 43
5.3 MyPDFSigner . . . 44
5.4 PDFStudio . . . 45

6 Conclusions **49**

Bibliography **51**

A CMD: Especificação dos serviços de Assinatura **55**

B PKCS#11 Wrapper API **72**

List of Figures

2.1 new PKCS#11 application overview [1]. 7
2.2 "Autenticação.gov" UI, Card menu. 13
2.3 "Autenticação.gov" UI, Signature menu. 13
2.4 "Autenticação.gov" UI, Security menu. 14
2.5 Wrapper architecture. 15

3.1 Cloud digital signature operation using a PKCS#11 driver and virtual smart card [2] 19
3.2 Cryptography API: Next Generation architecture [3] 21
3.3 Signature operation using a KSP with CNG [3]. 22
3.4 Citizen take-up of Austrian Mobile Phone Signature versus smartcard 'e-card' [4] 23
3.5 Cloud signing vs. Smartcard signing. [5] 23

4.1 Smart CMD architecture design 29
4.2 Interaction between applications and the CMD service through PKCS#11. . 29
4.3 Graphical User Interface for SmartCMD. 39
4.4 Detail signature operation using the developed software. 39

5.1 PDFStudio upload PKCS#11 library page. 46
5.2 Signature configuration window. 47
5.3 Result of a valid signature using CMD on PDFStudio app. 48

List of Tables

2.1 AMA available endpoints for CMD services. 12

4.1 PKCS#11 API implemented functions. 31
4.2 Attributes implemented for CMD objects. 32
4.3 Mechanism supported by the CMD token. 36
4.4 Prefixes used supported by the developed python module and PKCS#11 library 38
4.5 Pipe message structure. 38

5.1 Mechanism supported by the CMD token. 42
5.2 PKCS#11 functions tested using pkcs11-tool. 43

Acronyms

2FA Two factor authentication. 18

AMA Agência para a Modernização Administrativa. v, 1–3, 11, 12, 20, 27, 28, 37, 38, 41

API Application Programming Interface. i, iii, 2, 3, 6, 8, 11–14, 17–24, 27, 28, 32, 33, 37, 41, 43, 45, 49, 50

BER Basic Encoding Rules. 32

CAdES CMS Advanced Electronic Signatures. 24

CAPI Cryptographic Application Programming Interface. i, 1, 2, 13, 17–21, 24

CC Citizen Card. i, 1–3, 10, 11, 13, 14, 20, 27, 28, 33–36, 43, 44, 48, 49, 72, 81

CKMS Cryptographic Key Management System. 23

CMD Cháve Móvel Digital. i–iii, v, 1–4, 11–14, 18, 20, 22, 23, 27–29, 32–39, 41–50, 55

cms Connection Manager Service. 24

CNG Cryptography Next Generation. i, iii, 18, 20–22, 24

CSC Cloud Signature Consortium. 18

CSG Cryptographic Service Gateway. 23, 24

CSP Cryptographic Service Provider. i, 1, 2, 13, 17, 19–21, 24

DER Distinguished Encoding Rules. 32

DLL Dynamic-link library. 7, 20

eIDAS Electronic Identification, Authentication and trust Services. 17, 44

GUI Graphical User Interface. iii, 4, 27, 28, 33, 38, 39, 45, 50

HSM Hardware Security Module. 1, 6, 18, 24

HTTP Hypertext Transfer Protocol. 3, 12, 22, 27, 28, 33, 37, 50

IPC Inter Process Communication. 36, 38, 50

KSP Key Storage Provider. i, iii, 21, 22

NIF Número de identificação fiscal. 11

OTP One Time Password. 3, 11, 12, 19, 28, 37, 38

PAdES PDF Advanced Electronic Signatures. 24

PDF Portable Document Format. 1, 7, 22, 24, 41, 44–48

PKCS Public Key Cryptography Standards. i–iii, v, 1–8, 10, 13, 14, 17–19, 22–24, 27–29, 33, 34, 36–38, 41–46, 48–50

RIPEMD RIPE Message Digest. 6, 43

RSA Rivest-Shamir-Adleman. 5, 6, 9, 10, 12, 32, 35, 45

SHA Secure Hash Algorithm. 6, 43, 44, 46, 48

SOAP Simple Object Access Protocol. 3, 12, 37

SRP Secure Remote Password protocol. 24

SSCD Secure Signature Creation Device. 7

TLS Transport Layer Security. 7, 24

TSP Trust Service Provider. 7, 17, 18, 20

UI User Interface. iii, 13, 14, 37

USB Universal Serial Bus. 1, 6, 17, 44

XAdES XML Advanced Electronic Signature. 24

XML Extensible Markup Language. 24, 37

Chapter 1

Introduction

1.1 Motivation

Qualified digital signatures and authentication through cryptographic devices plays a very important role in the world of cyber security and their demand has been growing exponentially in recent years. Over the last decade, people have come to understand the power of digital security mechanisms in comparison to more traditional methods like a paper signature or emails and password-based authentication, that most of the time are extremely easy to discover.

Today, there are two large groups of accessible cryptographic services, personal physical devices and cloud-based services,the respective service provider's Hardware Security Modules (HSM), leading to all the operations being performed on the server side. Regarding personal physical devices, such as smart cards or USB crypto tokens, these are seen by most people as safer and more reliable mechanisms, as people feel that by having them in their possession, they are, somehow, better protected when compared to cloud services. However these physical equipment bring several limitations for mobile products such as mobile phones or tablets, as they usually cannot interact with them. Nowadays, and due to advances in the area of cyber security, the cloud services are starting to being seen as reliable, more convenient and easily accessible services for all people around the world, and the population are becoming to surrender to its practicality.

In Portugal, the government and the institutions responsible for the digital transformation, such as the Administrative Modernization Agency (AMA), have been providing and improving both services based on physical devices and fully cloud-based services. The Citizen Card (CC) is the personal physical device offered to citizens, it has a smart card with the functionality to store deployed cryptographic secrets, such as private keys used for digital signatures and authentications. On the cloud-based service side exits the "Chave Móvel Digital" (CMD), created in 2015 with the aim of exploring digital signatures without CC for both document signing and personal authentication , and so CMD marks a major step forward in providing a cloud-based mechanism that is completely free of physical equipment, capable of meeting the demands of Portuguese citizens and companies, providing a more practical solution over the CC.

1.2 Problem

On these present days there is a wide variety of cryptographic tokens that can be used by numerous applications and softwares, creating the need to develop a standard integration solution, independent of the operating system and thus create a global model of how an application can interact with a cryptographic token. Therefore was created the PKCS#11, a standard API capable of being internally new to allow the use of any cryptographic device or service without changing the way applications use these APIs. Therefore, applications that are familiar with the use of a PKCS#11 module are able to dynamically associate it as long as it is developed according to the standard interface.

There exists although another way to integrate these devices through the operating system itself. If a given system provides a service capable of using modules developed particularly by a cryptographic device, then it is possible to access the crypto tokens and their functionalities through a high-level interface. One of the most used solutions that offers a wide range of supported tokens is CAPI, when using a Windows system. This service presents a standard interface thus facilitating the development of specific software for a given token, being known as Cryptographic Service Provider (CSP), allowing the use of any cryptographic device on a Windows system through Microsoft CAPI, as long as it recognizes the presence of its own CSP for the desired token.

This solution, however, only allows the use of cryptographic operations in native operating system applications, and so it is still necessary to use a PKCS#11 library for all third-party applications. It is also relevant to point out that in the case of Unix systems, such as Linux or Mac OS, doesn't exist any solution at the operating system level, being thus limited to the use of standard libraries such as PKCS#11.

Regarding CC, several integration solutions are provided, including both a PKCS#11 library and a CSP itself, to allow the use of the token through Microsoft CAPI in Windows for native applications of the operating system and allow also the use of it in third-party applications, such as Thunderbird, or in Unix systems, using the respective PKCS#11 module.

However, the same does not happen with CMD, for which only a CSP is provided, missing a standard library for using the tool outside of a Windows system. This creates a limitation on its integration in systems and applications that exclusively depends on the use of these libraries. AMA also provides its own application, "authentication.GOV", totally independent of the system, allowing the user to digitally sign documents using both CMD and CC. In summary, CMD's big problem is focus on the fact that its usability is limited to applications on native Windows, through Microsoft CAPI, and the application "authenticação.GOV" so, this thesis focus on the develop of a standard solution in order to allow the use of this cloud-based token in all applications prepared to interact with PKCS#11 libraries.

1.3 Contribution

As discussed earlier, CMD's main problem is its lack of support for Unix systems, so in order to solve this problem, a proof of concept will be presented and described using as an interface between CMD's applications and services, a PKCS#11 API. Therefore, as a first step, it was made a detailed study of the entire standard interface and the way in which applications interact with this library, in order to have the perception of how it will be possible to change its behavior in order to change the normal interaction with a physical device with

a full cloud-based service.

Taking into account all the possible approaches, there were found two possible options with enormous potential, the first being the development of a virtual smart card, with a similar behavior to CC, simulated by a device driver, and the second one the building of a PKCS#11 interface capable of emulating the presence of a smart card in the system (as will be described later, the presence of a physical card is not mandatory).

After some study and discussion concerning both approaches, it was decided to follow the path of the PKCS#11 module, as the design of the virtual smart card would be a much more time-consuming and painful process, as there is no information or documentation regarding the internal functioning of the CC smart card or how the PKCS#11 interacts with it. Based on the work developed by Professor João Paulo Barraca in [6], where the objective is to virtualize the CC, giving the possibility to use in an academic environment this virtual card over the physical one, this investigation came to the conclusion that this approach would not be a better solution when compared to a simple new PKCS#11 module.

Having then as the main objective a new library, the first step would be to list all the necessary methods of the original library, having at the same time a perception of how the interaction between an application and a device is made during a digital signature operation. To get around this problem, it was used the PKCS#11 module provided for the CC as a foundation. From this point it was developed a *wrapper*, in order to have access not only to all the interface methods implemented by AMA but also in order to have a solid perception of the entire sequence of calls made by an application to this library. This *wrapper* will, in the final solution, still allow the selection and forwarding of requests to different PKCS#11 interfaces, making it possible to perform digital signatures not only with the CMD but also with the CC.

Now, having already achieve the concept of how to develop an interface between the applications and the services intended, the next step would be to access all objects and cryptographic operations of the CMD service. In this way, AMA provides an API containing all the operations necessary to access the cryptographic information and perform digital signatures, with the aim of developers and programmers being able to implement these services in third-party solutions and applications. Therefore, this API will be accessed through a *python module* launched by our PKCS#11 library, using *HTTP* requests with a *SOAP* protocol [7]. This program would also be responsible for encrypting the user's credentials to be included in the header of the *HTTP* request using the public key of the CMD services provided by AMA. Another one of its responsibilities would also concern the interaction with users through a simple graphic interface where it will be possible to enter their credentials and the OTP codes received during the signature process.

It was then intended as a final product, a PKCS#11 module capable of interacting with various applications correctly and in a standard way, providing the possibility of carrying out signature operations with both the CMD and the CC, thus allowing users a fluid and transparent use of CMD's cloud services.

1.4 Dissertation Structure

Before creating a proper and robust PKCS#11 module, some research and debate were needed, and so the next chapter will present some of the essential knowledge about asymmetric encryption and crypto tokens, Portuguese Citizen card structure and its use cases, and finally

library wrapping technology. chapter 3 will describe the state-of-the-art made to find existing functional cloud-based signature mechanisms or well-structured concepts. chapter 4, as the name implies, will detail all the work done to develop standard support for CMD, including the new PKCS#11 module, a python module and some basic GUI. Next, in chapter 5, results and validations are presented, where the PKCS#11 module and all software are tested against known third-party applications. Finally, the problems faced and possible future work are inferred at the end of the document.

Chapter 2

Context

Digital signatures are in many aspects equivalent to handwritten signatures, but when implemented properly they become difficult to forge or tamper. They were vastly adopted in different professional workflows such as for financial transactions, contract management software, and every other case when it is important to provide a solid proof of the signer's identity.

And so, a digital signature consists of a mathematical scheme for verifying the authenticity of digital messages or documents, and to became valid, it needs to pass on certain authentication and integrity prerequisites, so that the recipient will be confident that the message was created by a well-known and certified sender and the message content was preserved in transit. In their basics they rely on asymmetric architecture, so are used a RSA key pair where the sender can sign the content using the private key and the receiver can verify the signature using the corresponding public key.

In the following sections will be described all the concepts behind digital signatures and, more specifically, signatures using Portuguese cryptographic services over standard integration interfaces, and so will be covered as well the architecture of PKCS#11 interface and the middleware provided for the Portuguese cryptographic devices and services.

2.1 Asymmetric Cryptography & Cryptographic Tokens

An important concept to understand the process behind a digital signature is asymmetric cryptography. It is a cryptographic system that uses pairs of keys, where each pair consists of a public key and private key. The first one can be known by others and is used for encryption and signature verification operations, while the private key must be always kept secret, since it is used to decrypt information and generate digital signatures. For key generation, it can be used a vast number of asymmetric algorithms, depending on the software support and security level needed.

This system was created to resolve key management on an symmetric system, where both sender and recipient have the same key and it is used both for encryption and decryption. This generates a exponential growth of the number of keys in circulation, because for each new member on the network is necessary to generate n new keys, where n is the actual number of individuals. In contrast, for an asymmetric architecture, the public key just have validation and verification capabilities, so it can be used by all the other members in the network. Therefore, when adding a new member, it just needs to create a new key pair,

where the private key stays in the owner's possession and the public key will be distributed between all the other individuals.

There are still some more benefits in asymmetric cryptography, for example, it allows for tampering detection and non-repudiation so the sender cannot deny a sent message, but this advantage come with a certain setback in performance, so the architecture most adopted is a hybrid system between asymmetric and symmetric cryptography.

In the case of digital signatures, these also use asymmetric cryptography, where the message is signed with the sender's private key and signature can be verified by anyone who has access to the corresponding public key. This verification proves that the person sending the signed message had access to the private key, so there is a high possibility that the key actually belongs to the person who owns the used key pair. As seen before, this verification can also prove that the message information has not been altered since the signature generation, because it is almost exclusive to the original information.

Regarding the entity responsible for ensuring the authenticity of a public key and, thus, ensuring that the key is correct and belongs to the claiming entity, exists Public Key Infrastructures (PKI) composed of one or more entities that certify the owner of the key pair, known as Certification Authorities (CA). Each of these entities is responsible for issuing digital certificates containing the ownership of a certain public key, ensuring that any signature created with the private key can be correctly verified by the corresponding public key.

Regarding asymmetric cryptography, it is relevant to talk about the RSA cryptographic system, emphasizing its impact on digital signatures. As one of the first asymmetric cryptography systems, it is nowadays the most used system for digital signature operations due to its high degree of security.

To implement a signature with RSA, the signer entity will sign the message content using the following mathematical expression, $s = h^d \mod n$, and then the recipient uses, $h = s^e \mod n$ where $h = hash(m)$, to verify the signature made [8]. A particular step commonly used regarding the signed information, is "digesting" the message using a hash algorithm such as SHA-1 [9] or RIPEMD-160 [10], making the size of the actual data less variable because the length of a certain digest hash, created with a specific algorithm, is always the same.

For key pair storage are used security tokens (or Cryptographic tokens) present in, for example, smart cards, Universal Serial Bus key and a mobile device or using cryptographic hardware security modules (HSMs). These devices are physical and need to be kept private and protected, and are with them that a user can access their cryptographic credentials (as keys and certificates) using a valid password or Pin. After, these credentials can be used via standard programming interfaces, such as PKCS#11, implemented by modules (as libraries) specifically for a certain device. Regarding cloud-based systems, credentials are stored in the service provider on-premises HSMs, and they are responsible to maintain data integrity and ensure that the cryptographic private information are only access by the respective owner.

2.2 PKCS#11 Standard API

The PKCS#11 is one of the Public-Key Cryptography Standards [11] designed to be an interface between applications and cryptographic devices such as smartcards, HSMs and USB key tokens. It defines the 'Cryptoki' API allowing that applications may access cryptographic tokens and their functionalities, and due to its robustness and customization possibilities, it has been widely adopted in the cryptography industry, promoting diversification across

multiple platforms. Using a well-developed, PKCS#11 driver offers third-party applications a mechanism of using security functions without the need of changing neither the application source code nor the way the user interacts with it.

In an operating system environment, PKCS#11 drivers come in a form of a dynamic-link library (DLL) in Windows systems or Shared Objects (SO) in Unix-based operating systems. Applications that work with PKCS#11 offer the user the possibility to load the corresponding SO/DLL file of a certain crypto token, enabling cryptographic operations with the objects stored on that device. Some examples of popular applications that use PKCS#11 libraries are Adobe Acrobat Reader and Mozilla Firefox, allowing users to digitally sign PDF files (in case of Adobe Reader) with their private keys or performing SSL/TLS client certificate-based authentication (Mozilla Firefox).

Concerning cloud-based digital signatures, the development of a PKCS#11 module capable of using such services is a popular discussion in the last years, because of the increased demand for cloud and mobile services. Figure 2.1 presents a possible approach using a new PKCS#11 library to communicate and make requests to a cloud-based cryptographic service[1], switching the normal interaction with a physical device with a cloud trust service provider (TSP) that contains a SSCD.

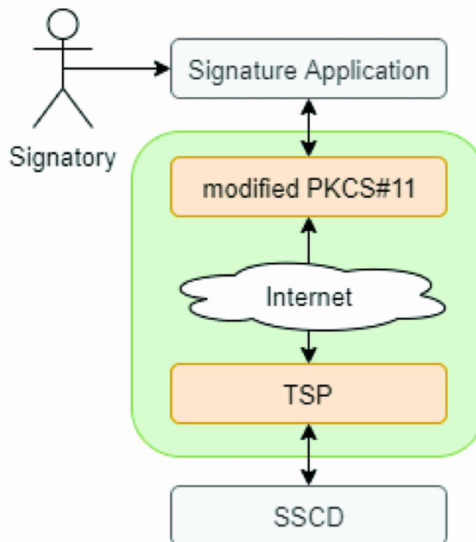


Figure 2.1: new PKCS#11 application overview [1].

Taking in consideration the internal architecture in a PKCS#11 module, each cryptographic device is represented by a *slot*, with which applications can interact by establishing one or more *sessions*. After a session has been established, the user authenticate their self with a token using a valid matching PIN, this way gaining access to protected objects within the token, such as keys, used to perform the cryptographic operations allowed by that specific cryptographic device [12](authenticated users are the only entity with access to private objects). Those objects can take many forms and have a vast number of different attributes and properties. In a *session*, each object is referenced and managed via a *handle* that do not reveal any information about a certain object, being the *handle* just a volatile reference to the object.

Regarding the PKCS#11 interface, a complete digital signature operation is composed of five main steps: initialization of communications between the application and the Cryptoki API (PKCS#11 API), establishment of sessions between the application and the token, extraction of all attribute informations needed from objects, such as private keys or public certificates, performing a signing operation with a private key, and finally closing all sessions and finalizing all processes. Following, a normal function sequence between an application and a Cryptoki library will be described regarding a digital signature operation using a crypto token.

1. Initialize Cryptoki

   In this interaction between the application and the PKCS#11 API it will be exchanging some trivial information such as Cryptoki general information or list of functions supported by this PKCS#11 module, as well as initialize the desired library.

   1.1 *C_Initialize*, where Cryptoki library is initialized and are passed as arguments information on how the library should deal with multi-threaded access.

   1.2 *C_GetInfo*, returns general information about Cryptoki library.

   1.3 *C_GetFuntionList*, application asks for all Cryptoki functions supported by the module´s API.

2. Establish sessions with a valid Slot

   Here, the application finds the wanted token present in a slot and starts a connection using Cryptoki sessions.

   2.1 *C_GetSlotList*, used for the application to find slots available in the system, is returned a list of Slot Ids.

   2.2 *C_GetSlotInfo*, returns information about a particular slot, passing as argument the slot identifier (Slot Id). With this information, applications can verify if the slot has a token or read the slot description, manufacturer identifier and some more static information.

   2.3 *C_GetTokenInfo*, have the similar behavior as the method *C_GetSlotInfo* but focused just on the token information, such as flags, representing all the cryptographic operations supported by the token (if the token supports digital signing operations, CKF_SIGN flag must be up), PIN lengths, serial number and other descriptive information.

   2.4 *C_GetMechanismList*, applications use this function to find what cryptographic algorithms are supported by the token. For example, for digital signatures, the application will be focus on CKM_SHA256_RSA_PKCS or CKM_SHA512_RSA_PKCS mechanism types.

   2.5 *C_GetMechanismInfo*, returns information regarding mechanisms capabilities (how exactly they can be used) and key lengths. Are passed as arguments the slot ID and the mechanism identifier.

   2.6 *C_OpenSession*, used to open a session with a particular slot. Opening a session with a slot is mandatory to perform cryptographic operations with the present token. Are passed as arguments flags to indicate the type of session needed, an

application-defined pointer to be passed to the notification callback, the address of the notification callback function and the slot identifier. One slot can have multiple open sessions at the same time with multiple applications. Are returned the session handle.

2.7 *C_Login*, if CKF_LOGIN_REQUIRED flag is active in the token information variable, this function must be called for the user to perform the authentication process. Arguments passed are user type, PIN for token authentication and its length. Regarding the crypto tokens tested, they use their own middleware for authentication, so this function was not called by the applications.

3. Extract Objects information

   After established the session with a particular token and the user perform the authentication process required, the application needs to acquire more specific information, particularlyRSA private key attributes. As already discussed in section 2.1, to create a digital signature we need a private key from a validRSA key pair.

3.1 *C_FindObjectsInit*, first application starts an object search operation, where it specifies what attributes it is looking for, using a CK_ATTRIBUTE data structure template.

3.2 *C_FindObjects*, after starting a object search (*C_FindObjectsInit* must be called before), this function is called to return the actual handles of the objects that match the desired template (passed as argument in *C_FindObjectsInit*). Arguments are the session handle and the maximum number of objects to be returned. The method returns a list of object handles.

3.3 *C_GetAttributeValue*, now that we have the object handles, we can call this function to actually obtain the object attributes needed to perform the signing operation. For some attributes, the application needs to call this function twice, the first call to get the attribute length for memory allocation and the second one to acquire the actual value of the attribute.

3.4 *C_FindObjectsFinal*, this method must be called to stop the search cycle started with *C_FindObjectsInit* in this session.

4. Signing operation

   After finding the desired private key and public key certificate to use in the signing operation, the application will initiate a signature process. Based on the information collected above, the application has all the information it needs to start a signature operation with the corresponding crypto token signing software.

4.1 *C_SignInit*, starts a signing process in a session previously open with a slot. Are passed as arguments, the signature key handle and the signing mechanism. The application after receiving a CKR_OK from *C_SignInit*, can choose to do a single-part signature using just *C_Sign* method or a multi-part signature, calling *C_SignUpdate* one time for each block of data and calling *C_SignFinal* to finalize the operation and receive the signature value (in a single part signature operation, *C_SignFinal* does not need to be called).

4.2 *C_Sign*, this method will receive the total data to be signed and will return the corresponding signature. Are passed as arguments, the session *handle* and the data to be signed.

4.3 *C_SignUpdate*, starts a multi-part signature operation where this method will be called multiple times. Are passed as arguments, the session *handle* and a data block.

4.4 *C_SignFinal*, ends a multi-part signature operation. Returns the signature from all the data received along the multiple *C_SignUpdate* calls.

5. Finalize and close sessions

   5.1 *C_CloseSession* or *C_CloseAllSessions*, application closes a session/sessions with a particular token. Calling this method will shut down all pending operations between the application and the token (in the specified session).

   5.2 *C_Finalize*, is called to indicate that an application is finished with the Cryptoki library. It should be the last Cryptoki call made by the application

This is a typical interaction between an application and a PKCS#11 module that makes use of a crypto token for signing operations. In chapter 4 and chapter 5 will be discussed and presented some examples of multiple applications assembling a digital signature using the process described above.

2.3 Portuguese cryptographic devices and middleware

2.3.1 Citizen card (CC)

The Portuguese Citizen Card (Cartão de Cidadão) is a personal device that stores relevant information in order to identify the owner citizen, as well as cryptographic objects such as authentication and signature digital certificates and asymmetric key pairs. Although in provided documentation there is no clear information about the presence of private keys [13], these keys are mandatory to perform operations such as authentications and digital signatures.

There is a set of well-delimited storage capacities of the smart card present in the CC, such as: store personal information used for validation of the holder's entity and this information is made of descriptive elements (biometric references) of user fingerprint; store sensitive and private information that just can be used by the owner of the card and it should never be revealed, including one symmetric authentication key, a private key from a RSA asymmetric key pair used for user authentication and a private key from a RSA asymmetric key pair used for creating digital signatures; store address information; save public information, more specifically personal photograph holders and public digital certificates in X.509v3 format used to authenticate the user and validate digital signatures; and finally store visible information like the personal photograph, name, birth date, all identity numbers, and CC expiration date [14].

Some operations need previous user authentication, made with the assist of a secret PIN. Following this idea, the Portuguese CC comes with 3 PINs, each one of them with different objectives, being the first one to access the home address, the second one for user authentication and the last one to perform digital signatures. These PINs guarantee a responsible and individual use of the card, for example in case of loss the person that founds the card

cannot use its capabilities. In more extreme cases, for example, in a theft situation, there is an 8 digit code for disabling all card functionalities.

Regarding the Citizen Card utilization, it can only be used in desktop environment using a smart card reader and the provided software used to enabling interaction between applications and the cryptographic token present in the card.

2.3.2 Chave Móvel Digital (CMD)

CMD was implemented in 2015 as a full cloud-based cryptographic technology that allows authentication and generation of certified digital signatures by the Portuguese Government. It allows a citizen (not just Portuguese citizens) to associate their identification number (passport or card of residence in case of foreign citizens) to a cell phone number or email address [15].

The user can activate the CMD functionalities using their CC, a smart card reader and the authentication PIN given with the CC, or they can do it through the Portuguese finances with the NIF and access code. If needed, users can activate CMD in person. For digital signatures, the user needs to activate personally the Qualified digital signature service present in the CC, this applies for making digital signatures both with the CC and CMD.

Regarding digital signatures with CMD, authentication and signing operations are secured via a two-factor authentication process, where the user needs their cell phone number and the corresponding PIN, next they will receive an OTP via text message to the associated number and then the user should provide the received code in order to complete the requested operation.

All operations concerning CMD, from emission and key generation until usage or revocation, are controlled by a Trustworthy System Supporting Server Signing (TW4S) called CMD Service Provider [16]. This means that the signing operations and private key storage are done in the server-side and not in a local physical device, and so, exits a public institution called "Agência para a Modernização Administrativa" (AMA) responsible for ensure a secure storage and access of the criptographic information as well as maintain a high availability of the provided services.

AMA provides several integration components for CMD, including the integrated signature process within the "autenticação.GOV" application and an API for external systems, giving programmers and thrid-party software developers access to CMD cryptographic operations.

This API provides digital signature operations and has in its core the following services:

- *GetCertificate*: called when a user wants to access their certificate files;
- *SCMDSign*: should be utilized when we wants to sign a single document using the CMD service;
- *SCMDlMultipleSign*: has the same functionality as the operation described above but for multiple documents;
- *GetCertificateWithPin*: used for acquiring user certificates with the associated CMD PIN;
- *ForceSMS*: forces to be send a SMS with a new OTP associated to an operation in progress;

- *ValidateOtp*: after receive an OTP code via SMS, this operation is used for validation. Is returned the signature of the document, a list of signatures or the citizen certificate, following the previous initialized operation;

This service can be used to get user's certificates or to sign documents. It can be accessed via HTTP communication with basic-authentication (credentials provided by the service entity) and SOAP messages. Regarding the credentials used to access this service, they need to be given by AMA.

All the operations described, must use asymmetric cryptography within the HTTP requests, to secure all user credentials (phone number, signature PIN and OTP) and private data exchanged. The public key used for encrypting this information is given by CMD services' X.509 certificate. On the other side, CMD services will decrypt the information using their private key from the RSA key pair. AMA provides the following endpoints, including a test endpoint for all the developing iterations (DEV), a pre-production endpoint (PPR) and a production endpoint (PRD):

DEV	Service: https://dev.cmd.autenticacao.gov.pt/Ama.Authentication.Frontend/SCMDService.svc (just accessible inside AMA' network) Certificate: to be defined
PPR	https://preprod.cmd.autenticacao.gov.pt/Ama.Authentication.Frontend/SCMDService.svc Certificate: to be defined
PRD	https:// cmd.autenticacao.gov.pt/Ama.Authentication.Frontend/SCMDService.svc Certificate: to be defined

Table 2.1: AMA available endpoints for CMD services.

To use this service, it is first needed to contact the AMA resource service to ask for credentials to access their services (Application ID and a password) and a public key to encrypt data when making requests through an insecure channel. Following, it is required to establish a valid connection to one of their endpoints using HTTP with basic authentication and SOAP message format. In chapter 4, when presenting our solution, we will discuss some of the requests used to perform a qualified digital signature using this service. More information regarding AMA API for CMD services can be found in [7].

2.3.3 Middleware and drivers

To give access to the functionalities provided by the Portuguese cryptographic systems, users can use a desktop application, "Autenticação.gov". The application is available for a variety of operating systems, including Microsoft Windows, Linux and Mac OS. Concerning application interface, its composed by three menu options: card, signature and security.

In the Card page, the users can have access to their personal information stored on the card, such as name, age, birth date and address. This last one is PIN protected. The Signature menu allows to digitally sign documents with both CC and CMD. The last menu, Security, allows the user to check information regarding digital certificates, such as the chain of trust and also download them as files. "Autenticação.gov" application does not have any signature verification mechanism, what is major disadvantage compared to other third-party applications.

To extended usability, are also provided PKCS#11 modules to use the Portuguese CC with third-party applications that make use of that standard API. However, no PKCS#11 module is provided to use CMD services The only way of signing documents with CMD is through "Autenticação.gov" application or using the provided CSP for Microsoft Cryptographic API (Microsoft CAPI) in a Windows operation system.

Notably, the "Autenticação.Gov" application does not make use of the CC'sPKCS#11 module, which is somewhat strange.

Figure 2.2: "Autenticação.gov" UI, Card menu.

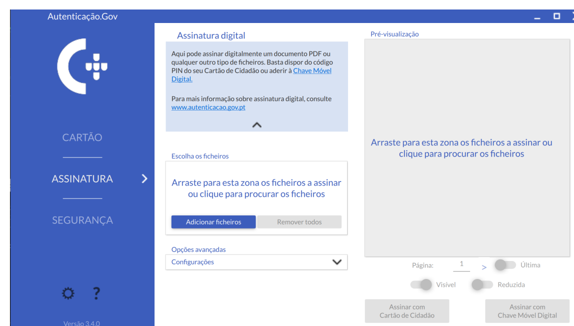


Figure 2.3: "Autenticação.gov" UI, Signature menu.

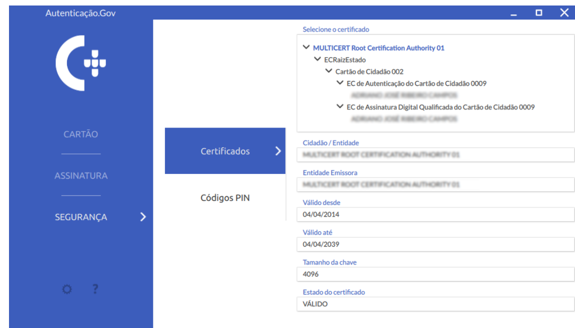


Figure 2.4: "Autenticação.gov" UI, Security menu.

2.4 Library wrapping

Sometimes standard APIs do not have enough flexibility, or Libraries are too big to be changed for a hand full of alterations. To face this problem can be use wrapping technologies. Library wrappers consist of a layer of new code implemented between applications and the wrapped libraries. They allow a flexible way to, for example, refine a poorly designed interface, upgrade compatibility or enable cross-language and/or runtime interoperability [17].

Wrapping technologies bring programmers tools capable of suit their needs of scalability without too much effort. For example, a certain library is important for a given number of applications, but there are some methods, or parts of the code, that are outdated and need some intervention. Therefore, using a *wrapper* there is no need to know the code behind the already compiled Shared Object (in Linux systems), we just need to implement a layer between the applications and the original library with the following logic: if some changed method is called, the request goes to the new library; if not, the request is forwarded to the old library (it is a very simple example, in practice the work is sometimes more complex). Wrappers can also be used for simple tasks, as debuggers or logging systems, because it is an easy way of catching requests to a certain library interface without changing the way applications use that library.

This technology was first used in the project to in understand the communication between different applications and the CC PKCS#11 module, next, we used the *wrapper* as a thin layer of code to suit as a "front-end" API that communicate with applications, forwarding the requests to the appropriated library (CMD developed PKCS#11 library or CC provided PKCS#11 module).

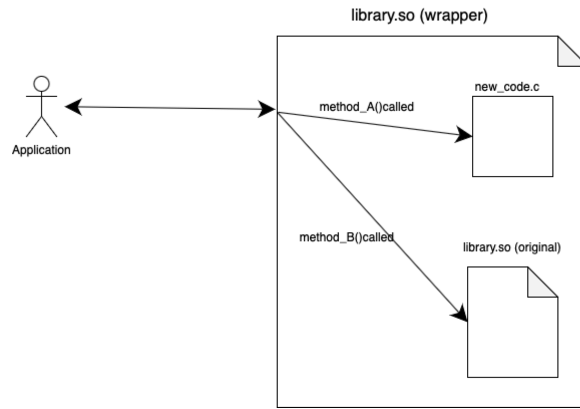


Figure 2.5: Wrapper architecture.

Chapter 3

Background & Studies

During the last decade, the migration from physical or on-premises to cloud services is increasing in a extreme rate given its vast advantages in cost, scalability and availability. Concerning digital signatures, the gains are mainly in usability and convenience for the end user.

In July first of 2016, cloud qualified digital signatures were made available for the global consumer, since the eIDAS Regulation [18] was approved. With this regulation, its not mandatory that the signer own the secure digital signature device, and so this devices can also be managed by the authority that issues the cloud qualified digital certificates, giving that the trust service provider (TSP) can also assure the sole control of the signer over the signature creation data (namely the private keys).

In order to create a grounded solution, first it is necessary to analyze some existing solutions or theories, so that can be followed the footsteps of many experienced organizations in this service transformation, since remote signatures solutions are available and already in use by well-known software companies and EU Governments [19].

Therefore, it is important to understand the differences between cloud-based signatures, where services and certificates are stored server-side, and digital signatures through physical devices that contain the respective certificates and keys, and where the cryptographic operations are made using local middleware. Regarding on how it is possible to offer applications the possibility to use cryptographic services in the cloud, it is relevant to mention and describe the way the integration of these services can be done through cryptographic APIs such as PKCS#11 or the adaptation of CSPs for its use through Microsoft CAPI.

3.1 Cloud-based Digital Signatures

A cloud signature or "remote signature" is a type of certificate-based digital signature that uses standard protocols to generate an e-signature using digital identity certificates that are provided as-a-service in the cloud from an certified TSP. Compared to cryptographic methods based on physical devices such as smart cards or USB tokens, a cloud signature service allows significantly more flexibility and availability, as they can be used easily across desktop applications, web browsers, and mobile devices.

The main principle behind cloud-based digital signatures is that the private key is hosted server-side, and not in a physical device. This implies that the actual signing operations need to be done in the server-side, but at the same time maintaining its use under the main control

of the user with his private key [20, 21]. Regarding the smart cards and other physical devices, they are not mandatory to perform a qualified digital signature, as described and proved in [22].

Concerning cryptographic information, namely private keys, they are stored and managed in the service provider HSMs and can only be used by the respective owner. In this way, and as a way to ensure a permissive use of such information, entities that provide cloud digital signature services, adopt 2FA (two factor authentication) as the default security scheme.

3.2 Digital Signature APIs

The following section presents the most used cryptographic APIs and how they can integrate digital signatures for cloud providers: PKCS#11 (platform independent), Microsoft Cryptography API (CAPI) and Microsoft Cryptography API: Next Generation (CNG) (only for Microsoft Windows operating systems).

Since this thesis focus on integration of a cloud-based digital signature provider (CMD) throught a PKCS#11 API, will be more intensely described the solutions regarding the ones that use PKCS#11 as cryptographic API, but will be also briefly mention solutions that can integrate cloud providers in a restrict environment (namely Microsoft CAPI and CNG for Windows operating systems).

3.2.1 PKCS#11 API

In order to offer a wide range of compatibility applications with digital signatures in the cloud, a good initiative would be the development of a PKCS#11 module, changing the use of direct communication with a physical device with the connection to a TSP service to carry out the cryptographic operations.

Regarding private keys and digital certificates stored on the server side, these are securely generated and managed through the HSMs of the TSP, in this way a PKCS#11 module can offer transparently to applications that do not know how to work with signatures on the server-side with a mechanism capable of providing these cryptographic operations. When using such a PKCS#11 module, there is no need for either the applications to change the source code or how the user interacts with these services.

Likewise, the HSMs producers of each TSP would be responsible for providing their own PKCS#11 module, enabling their clients to use the cryptographic information issued in the cloud with any application that can integrate the PKCS#11 module.

In [2] is described a possible way of implementing a cloud PKCS#11 driver using a software token (virtual smart card) to store all objects and perform cryptographic operations. More specifically, the PKCS#11 driver communicates with a generic signing web service exposed by the server-side signature provider, preferably one running according to the Cloud Signature Consortium Standard (CSC) [23].

Regarding sessions, virtual slots, users and objects, they are managed using specified implemented classes. Regarding the signature operation, the application that integrates the PKCS#11 driver starts a signature operation calling the function *C_SignInit* passing as arguments the session handle, the algorithm used to perform the message *digest* and the signature key handle (private key), obtained after a object search operation. Next, it is called the function *C_Sign* twice, the first call to get the signature length computed using the private key modulus, and the second call to receive the signature, for that the driver sends a request

to the signing service provider containing the signature password, the OTP and the message digest.

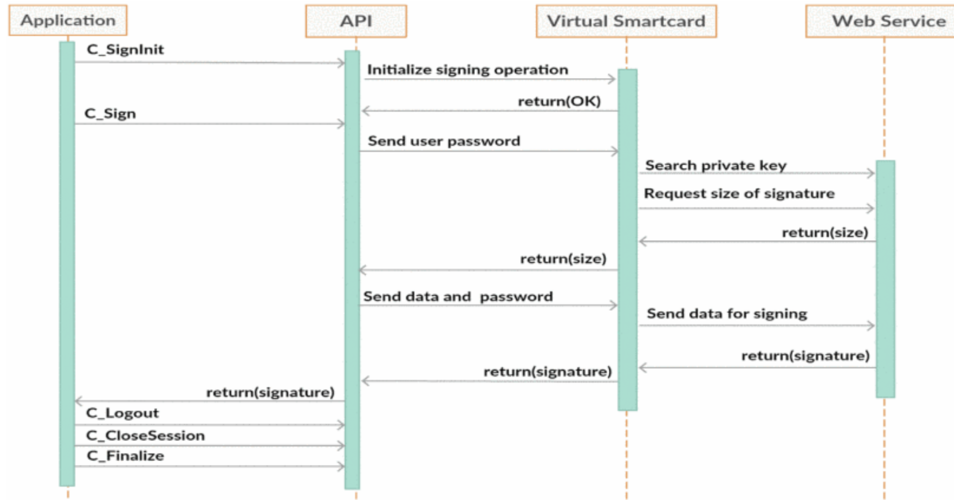


Figure 3.1: Cloud digital signature operation using a PKCS#11 driver and virtual smart card [2]

3.2.2 Microsoft Cryptography API (CAPI)

The Cryptographic Application Programming Interface (CAPI), also known as Microsoft CryptoAPI, is a cryptographic API native to the Windows operating system, that offers to applications directly provided with the system an interface to access cryptographic operations of various security devices and services.

Microsoft CryptoAPI is not dependent of any algorithm or operation since specific cryptographic functions and algorithms are not directly attached to the API [24]. They are instead performed by independent modules known as Cryptographic Service Providers (CSP). The cryptographic functions, instead, rely in the different CSPs to provide the necessary cryptographic algorithms and secure storage for any cryptographic session or keys that may be generated. All application-to-CSP communications must be done via the exposed cryptographic functions in the CryptoAPI, and each of these cryptographic functions, that communicate with a CSP, has a parameter that describes which CSP will be used.

CSPs modules need to be properly installed in the system, and these can be specifically developed not only for physical devices but also for a cloud-based digital signature service. Therefore, Microsoft CAPI can be a viable alternative to PKCS#11 regarding enabling cloud-based digital signatures, only in Windows systems, limiting its use to native operating system applications made available by Microsoft. This API does not solve the problem described in this thesis, and so the search for a solution proceed.

3.2.3 Cryptographic Service Provider (CSP)

In Microsoft Windows, a Cryptographic Service Provider (CSP) is a software library that is design based on the Microsoft CryptoAPI (CAPI). An CSP abstracts from the applications the entire internal process used by a cryptographic operation, thus offering a completely

transparent support solution. Each CSP module is independent and can be used by any application prepared for its integration.

Each CSP provides a different implementation of the cryptographic operations that it provides to the CryptoAPI and may be implemented in hardware with a software interface library or exclusively in software. Microsoft is always encouraging independent developers to write CSPs, this way providing support to other algorithms.

Regarding CSP, consists of a set of functions made available to CryptoAPI attached to a library file. As this library is available at runtime to applications through the CryptoAPI, the library is provided in the form a dynamic-link library (DLL). Some CSPs may, in specific situations, communicate with users directly, such as when a digital signatures are performed using the user's signature private key. In order to ensure that the DLL is not tampered during the time it is in use by the application, the CSP includes itself a digital signature in a signature file. This signature is checked in regular intervals by the CryptoAPI while the CSP is in use. Currently only Microsoft is able to generate valid signatures for each CSP. As Microsoft will only sign CSPs in the United States, these would have to be shipped to the US first to be signed.

The CryptoAPI programming model takes its philosophy from traditional multi-user operating system design, where access to a hardware device by a user's application is provided through well defined operating system calls. Just as well-behaved applications are not allowed to communicate directly with device drivers and hardware, well-behaved applications cannot directly access the CSPs and cryptographic hardware.

Regarding cloud-based digital signatures, a remote CSP would integrate the call to the cloud-based digital signature service using a TSP, offering the cloud service cryptographic functionalities to the applications. Generally, any existing application that is compatible with Microsoft CryptoAPI would thus work with cloud digital certificates and so is not required to perform any changes. An example of a cloud-based digital signature tool, besides Portuguese CMD, that provides a CSP, is Cryptomatic Signer [19].

AMA provides a CSP both for the CC and CMD within the middleware package available, in this way consumers can access the Portuguese cryptographic services through Microsoft CAPI. In order to enable the CMD service, the user needs to download the corresponding digital certificate using the "Autenticação.gov" application and store it along side the remaining user certificates (in Microsoft Windows exists a special directory exclusively to store them). When using a CC, this certificates are extracted directly from the smart card and stored in the user certificates directory, and when the card is removed, the certificates are also deleted from the directory. Finally, when using a signature application that supports Microsoft CAPI, it will be present a option to use CMD and CC to digitally sign documents using the provided CSP, and this CSP will use the digital certificate, previously downloaded or extracted (for the CC), to access CMD cloud service or the CC cryptographic service.

3.2.4 Microsoft Cryptography API: Next Generation(CNG)

This tool was developed in order to solve Microsoft CAPI flexibility problems regarding the implementation of new or modified algorithms and cryptographic operations. Having this in mind, Microsoft decided to create CNG, as a cryptographic solution prepared to support changes in existing functions or even creating new ones.

When programmers need to support a new cryptography algorithm or function, they only need to implement these new functions and add the result to the existing library. This design

concept is called the plug-in model [25], which has as its main advantage the possibility of reusing common parts and thus alleviating the complexity of integrating new interfaces.

Even though CAPI has a similar component, a CSP, developers need to implement individual CSPs and the results require a certification of signature to verify its feasibility after implementing the new CSPs, which causes lack of agility. CNG uses KSP instead of CSP. As Microsoft CAPI, CNG could be implemented for cloud-based digital signatures [3], however is not a viable approach for our solution because it only works in a Windows machine as well (same as CAPI).

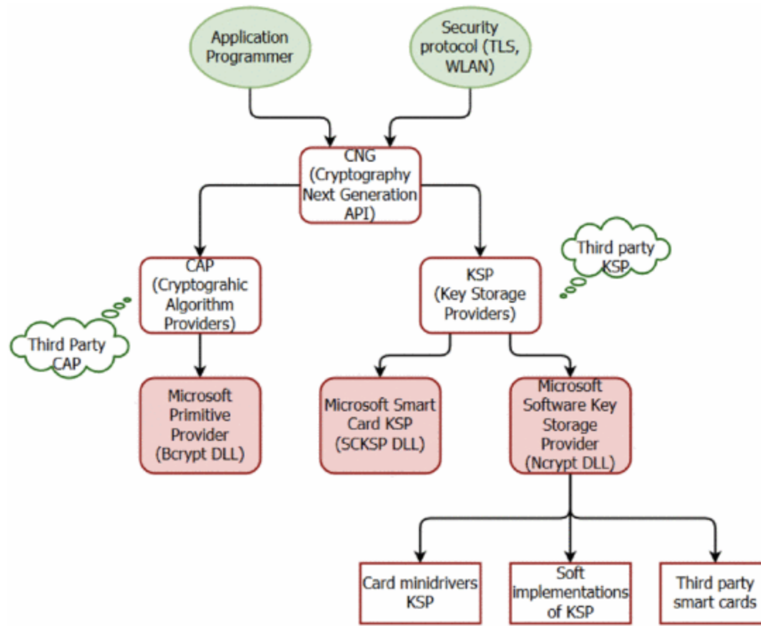


Figure 3.2: Cryptography API: Next Generation architecture [3]

3.2.5 Key Storage Provider (KSP)

KSPs are used in CNG and are equivalent to CSPs in Microsoft CryptoAPI. The main role of the key storage provider is to handle the private keys storage (including private keys encryption), but it has also a role in many cryptographic operations that include the use of the private key, like digital signatures. Therefore, as CSPs, KSPs can be used for digital signatures using a cloud-based solution, without the need of any alteration in comparison with physical cryptographic solutions.

In Figure 3.3 is described the complete process regarding a digital signature using a KSP that implements a cloud service, where the user starts with a login in the respective web application from the cloud service enabling the digital signature capabilities for that user. Then, using a signing application (for example, Adobe Acrobat) that supports CNG, the user will start a document signature operation. In this way, the application will use the respective KSP (developed for that cloud service) to request the signature, sending the corresponding document hash. Once the KSP have all the information necessary, it will send it to the cloud service where it will be generated the signature value. Finally, the KSP receive the signature and send it back to the application.

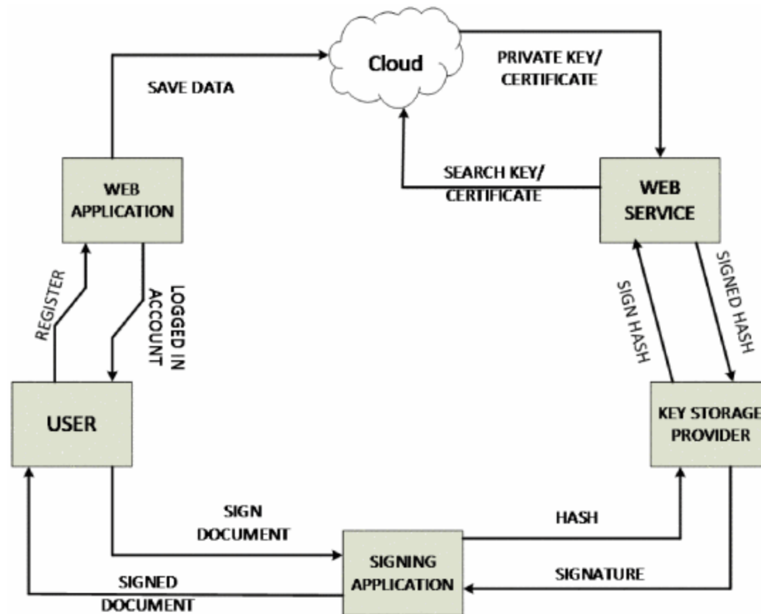


Figure 3.3: Signature operation using a KSP with CNG [3].

3.3 Cloud-based product solutions

Regarding cloud-based solutions already used and commercialized, there were found several solutions including government authentication tools and commercial frameworks. After all this analysis, and after all the aggregated solutions, it was concluded that only two of them have a similar architecture and behavior to CMD. In this way, only those two are going to be addressed in the elapse of this thesis. The first one was implemented by the Austrian Government and presents the same limitations of the Portuguese CMD and a second with a vast number of integration options, such as standard APIs developed by Crypthomatic Software.

3.3.1 Austrian Mobile Phone Signature

The Austrian Mobile Phone Signature service provides users with a digital signature service based on Austria's official eID. This service meets the requirements for the creation of qualified electronic signatures, listed in the EU Signature Directive [19].

This service can be used by any web application since it is not necessary to have any other software installed in the client devices. To start a signature operation, the web application makes a signing request to a Security Layer, and in this request holds the document to be signed and some other information needed. After, the signature server presents a web page where the user can select the key he wants to use, then the user authenticates itself in order to authorizes the signing process. Being a service designed just for web services or web applications, the user cannot use this service to sign something on, for example, a desktop application, since it is not provided a integration mechanism such as a PKCS#11 module, which represents a lack of versatility and usability. As for CMD, the Austrian Government offers a pdf-signer tool for creating PDF signatures using this remote digital signature service.

Additionally, the PDF signature tool web application can be accessed using a HTTP interface. Through this, third-party applications can send files to be signed by the Austrian

service [26].

Regarding the Austrian signature service, it is an easy-to-use signing tool for Austrian citizens but with serious integration limitations because, to use it, are only available a web-application interface and pdf-signer tool (mobile and desktop application); there is no interface developed to integrate this remote service with industry-standard APIs. In conclusion, this service has the same limitations as Portuguese CMD since it cannot be used in third-party applications that use, for instance, PKCS#11 as cryptographic API.

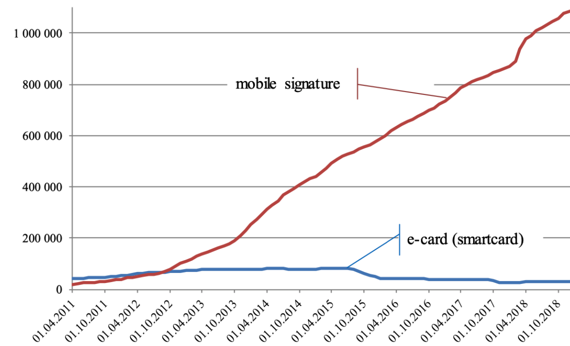


Figure 3.4: Citizen take-up of Austrian Mobile Phone Signature versus smartcard 'e-card' [4]

3.3.2 Crypthomatic Signer and Crypto Service Gateway

Crypthomatic company is a global provider of server solutions including cryptography services. They focus on delivering a well secure and flexible product to their customers, offering cloud cryptographic solutions for cloud-based signing and authentication operations like Crypthomatic Signer and CSG. In [5] are shown advantages and scalability of cloud-based digital signatures mechanisms over physical smartcard devices.

	Signer	Chipcard
CARD READERS REQ	no	yes
MOBILITY	yes	no
WYSIWYS	yes	no
LOGGING POSSIBLE	yes	no
INSTANT REVOCATION	yes	no
BLACKLISTS REQUIRED	no	yes
IND. TIMESTMP REQ	no	yes
COST PER USER	very low	very high

Figure 3.5: Cloud signing vs. Smartcard signing. [5]

First, they offer a Cryptographic Key Management System (CKMS) which is used to assists its users in handling their keys. Next for signing operations, they provide a signer

service enabling web-based signing capabilitie. Finally, they provide a crypto service gateway service to give the possibility to use a managed HSM.

Regarding the CSG, it is a platform that simplifies application integration while ensuring HSMs availability and utilization, acting as an abstraction layer between applications and the HSMs. CSG needs to guarantee a high-performance service with minimal latency and any interruption, ensuring a quality user experience. So, in sum, CSG provides the following benefits:

- Reduces costs through shared infrastructure, and so increasing HSM utilization;
- Uses a centralized policy and control over all cryptographic operations and key management, assigning this to the security team and this way facilitating crypto-agility;
- Enables complete central management and monitoring over the entire HSM infrastructure;
- Provides proof of compliance with easy-to-read audit logs;
- Offers simple-to-use APIs for increased software readiness and consequently reduced time to market.

Communication between applications and the cryptographic services is secured by standard methods, such as the use of the Secure Remote Password protocol (SRP) [27], TLS [28] and others. Concerning the formats supported by the provided services, are offered integration with the more common ones, such as XAdES [29], PAdES [30], and CAdES [31] for digital signatures [32].

Cryptomathics Signer, as well as their Crypto Service Gateway (CSG), offer well-known and standard cryptographic methods and primitives. For instance, in advanced Electronic Signature are available schemes for XML, PDF and cms formats regarding the Signer product. For the CSG is offered a wide range of primitives and methods as well, by using high quality HSMs (like AEP, SafeNet, Thales, and Ultimaco).

For authentication, Cryptomathic products also offer a great range of methods. For instance, the Signer product offers multi-factor authentication methods as default, and so the service can be considered as well-protected and secured. In the other hand, Cryptomathics CSG offers a smaller set of authentication methods, since are just available Username/password as well as LDAP- and RADIUS-based authentication methods, missing the option of multi-factor authentication. As for key storage, Cryptomathic uses third-party HSMs build by AEP, SafeNet, Thales, and Ultimaco among others [33]. Therefore the physical protection level depends on the HSM in use.

Cryptomathics provides a variety of integration options for CSG and Signer both in terms of applications interfaces and HSM to applications and services. Developers can use a set of well-known APIs to send requests to CSG [33] such as, a PKCS#11 interface, a JCE, a Microsoft CAPI CSP and a CNG.

Doing a complete review of the services offered by Cryptomathic, they can be reasonably secure and easy-to-use. They offer a vast range of authentication methods, as well as a wide selection of integration interfaces, and feature professional HSMs to ensure the key protection. The Signer service offers all major signature formats and therefore seems to meet a wide range of use cases. Even so, if a special use case cannot be met with the Signer product, Cryptomathic also offers its Crypto Service Gateway.

This way, Cryptomathic Signer is a strong solution for large organizations concerning their digitalization strategies.According to [32], ”It is the only zero-footprint signing technology that can offer an appropriate security assurance level while being compatible with all types of devices”.

Chapter 4

Smart CMD

4.1 Possible approaches

Here, the main objective was to enable CMD capabilities for applications that use PKCS#11 as a cryptographic API. To solve this issue and after studying some possibilities, it was found two promising approaches. The first approach consists on the development of a virtual smart card capable of replicating the Portuguese CC functionalities and cryptographic operations with capabilities of working with the already provided CC middleware while exchanging the default communication of the physical smart card with the CMD cloud service. The second approach concerns the development of a new PKCS#11 module capable of emulate a presence of a virtual CMD token in the system that could be detected by applications and use an external HTTP client to make requests to the CMD service API.

After some research and discussion about both approaches, the conclusion was that implementing a virtual smart card would not be a superior solution when compared with the development of a new PKCS#11 module, with the aggravation of having a greater implementation effort, since the lack of information regarding the Portuguese CC internal architecture and software leads to a thorough process in understanding how the CC works internally and how it interacts with its cryptographic services.

In this way, the method chosen was the development of a new PKCS#11 module extending the Portuguese CC module, adding all the software needed to establish a connection to the CMD cloud service, as well as to implement a GUI for authentication inputs.

4.2 Architecture

The Smart CMD uses a set of developed software to properly generate a qualified digital signature using a cloud-based service. The architecture was design with the objective of offering users the capabilities of access both the CMD and the CC cryptographic operations. In this way, was created a *wrapper* to serve as a front-end shared object with the objective of redirect the application requests to the appropriated library.

Going a step below the front-end layer, was developed two workflows, the first one to handle CMD requests and another one to support the CC capabilities using the middleware already provided by AMA. For the CMD service was created a new PKCS#11 library containing all the functions necessary to perform a digital signature, a *python module* to communicate with the cloud CMD service provider and lastly, a GUI designed to interact with the user

when authentication is necessary (for credentials) and to show some operations feedback. Additionally, is used a pre-defined directory and a folder containing all the software, that will be used for the PKCS#11 library to confirm the presence of a virtual CMD token and offering it as a valid slot when *C_GetSlotList* is called by applications. Since was used a wrapper to offer the possibility two work with both the Portuguese cryptographic services, when applications call *C_GetSlotList*, and if the user connect a CC via smartcard reader, the PKCS#11 module will return a list with not just the CMD slot but also the one with the CC.

After implementing the *wrapper*, we could list all the implemented PKCS#11 functions for the CC, in this way was possible to filter just the necessary functions to perform a actual signature operation. In Table 4.1 is listed all the implemented functions for both the CMD and CC libraries. In addition, Table 4.2 describe all the cryptoki attributes implemented for CMD used in a signature operation. This attributes will be retrieved to applications when they call *C_GetAttributeValue*.

Regarding the communication process, was chosen a *python module* because it can be launch just when needed and run exclusively in background, this way giving the perfect solution to interact with the CMD cloud service. This process has three main functionalities, that is, to make HTTP requests to CMD API based on the application requests to the new PKCS#11 API, to receive user credentials over a Qt design GUI and finally to encrypt credentials using AMA provided public key.

The interaction between the python module and the CMD service is done using a set of HTTP requests to the corresponding API. In this way, after receive a signature request from the new PKCS#11 library, the python module will created a HTTP client and call the *CCMovelSign* service, sending the data to be sign (and some more information that will be described in the following sections). After receive a valid response, the user will receive a text message containing the OTP value, and with which the python module will validate and request the signature value, calling the *ValidateOTP* service. The response from this last service, contains the signature value that the python module will then forward to the PKCS#11 library. Concerning the *python module* life cycle, after sending the signature value, this process is closed, together with the Linux pipes previously created.

In the CC workflow, was used the already provided middleware, and so, if the crypto token in use matches a Portuguese CC, the front-end layer will forward the application calls to the correct PKCS#11 library.

Concerning applications library integration, the new module is used in the exact same way as it was working with a common smart card PKCS#11 API, since was guaranteed a transparent and seamless interaction. More details about every component can be found more ahead as they will be discussed in section 4.3. Figure 4.1 provides a visual representation where the implemented solution was based on. Additionally, in Figure 4.2, is visually described a full interaction between the application and the CMD, when performed a digital signature operation, through the new PKCS#11 module.

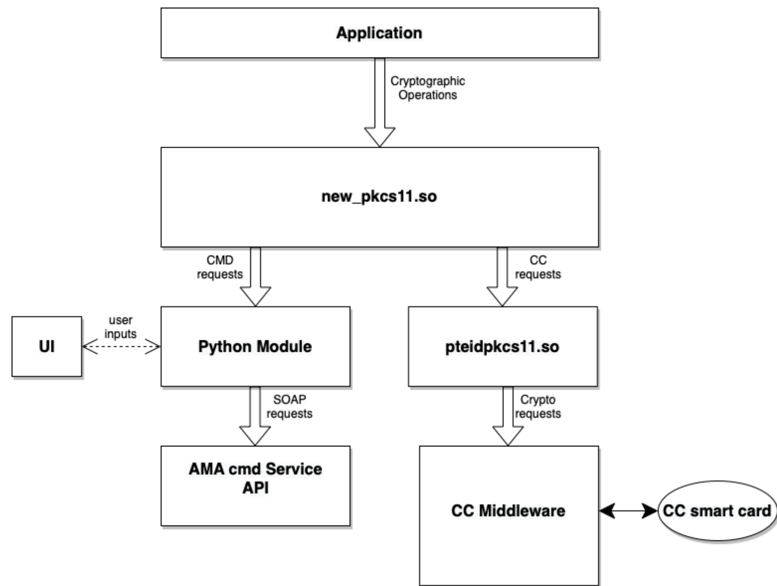


Figure 4.1: Smart CMD architecture design

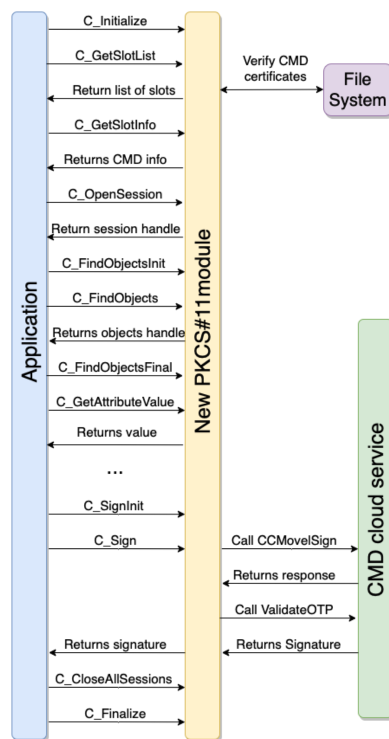


Figure 4.2: Interaction between applications and the CMD service through PKCS#11.

PKCS#11 implemented API Functions	Chave Móvel Digital	Cartão de Cidadão
C_CancelFunction	✖	✔
C_CloseAllSessions	✔	✔
C_CloseSession	✔	✔
C_CopyObject	✖	✔
C_CreateObject	✖	✔
C_Decrypt	✖	✔
C_DecryptDigestUpdate	✖	✔
C_DecryptFinal	✖	✔
C_DecryptInit	✖	✔
C_DecryptUpdate	✖	✔
C_DecryptVerifyUpdate	✖	✔
C_DeriveKey	✖	✔
C_DestroyObject	✖	✔
C_Digest	✖	✔
C_DigestEncryptUpdate	✖	✔
C_DigestFinal	✖	✔
C_DigestInit	✖	✔
C_DigestKey	✖	✔
C_DigestUpdate	✖	✔
C_Encrypt	✖	✔
C_EncryptFinal	✖	✔
C_EncryptInit	✖	✔
C_EncryptUpdate	✖	✔
C_Finalize	✔	✔
C_FindObjects	✔	✔
C_FindObjectsFinal	✔	✔
C_FindObjectsInit	✔	✔
C_GenerateKey	✖	✔
C_GenerateKeyPair	✖	✔
C_GenerateRandom	✖	✔
C_GetAttributeValue	✔	✔
C_GetFunctionList	✔	✔
C_GetFunctionStatus	✖	✔
C_GetInfo	✔	✔

PKCS#11 implemented API Functions	Chave Móvel Digital	Cartão de Cidadão
C_GetMechanismInfo	✔	✔
C_GetMechanismList	✔	✔
C_GetObjectSize	✖	✔
C_GetOperationState	✖	✔
C_GetSessionInfo	✔	✔
C_GetSlotInfo	✔	✔
C_GetSlotList	✔	✔
C_GetTokenInfo;	✔	✔
C_Initialize	✔	✔
C_InitPIN	✖	✔
C_InitToken	✖	✔
C_Login	✖	✔
C_Logout	✖	✔
C_OpenSession	✔	✔
C_SeedRandom	✖	✔
C_SetAttributeValue	✖	✔
C_SetOperationState	✖	✔
C_SetPIN	✖	✔
C_Sign	✔	✔
C_SignEncryptUpdate	✖	✔
C_SignFinal	✔	✔
C_SignInit	✔	✔
C_SignRecover	✖	✔
C_SignRecoverInit	✖	✔
C_SignUpdate	✔	✔
C_UnwrapKey	✖	✔
C_Verify	✖	✔
C_VerifyFinal	✖	✔
C_VerifyInit	✖	✔
C_VerifyRecover	✖	✔
C_VerifyRecoverInit	✖	✔
C_VerifyUpdate	✖	✔

Table 4.1: PKCS#11 API implemented functions.

Group	Attribute	Value	Description
Global attributes	CKA_CLASS	CKO_CERTIFICATE, CKO_PUBLIC_KEY or CKO_PRIVATE_KEY	Identifies the object class, was used CKO_CERTIFICATE value for the X509 certificates and CKO_PUBLIC_KEY and CKO_PRIVATE_KEY for the public and private key
	CKA_SUBJECT	Value presented in the user CMD certificate	Includes the certificate subject name, encoded in DER format;
	CKA_TRUSTED	CK_TRUE	Flag indicating that an object can be trusted by the applications;
	CKA_ISSUER	Value presented in the user CMD certificate	Describes the issuer entity, encoded in DER format;
	CKA_LABEL	e.g. "Signature Private key"	Brief object description;
	CKA_ID	1221 (could be any number)	Identifier value for a certain key. Considering a certificate object, this field corresponds to the key pair ID associated with the certificate. Within a key pair, both the public and private key should have the same CKA_ID value;
	CKA_TOKEN	CK_TRUE	indicates that a certain object is a token object;
Certificate attributes	CKA_CERTIFICATE_TYPE	CKC_X_509	In a PKCS#11 API exists different certificate types, and so this value is used to identifify them. In the PKCS#11 module developed, the CMD certificates have this attribute with CKC_X_509 value, indicating that they correspond to a X.509 public key certificates;
	CKA_VALUE	Complete value of the certificate in a byte-array	The complete value of the certificate, encoded in BER format;
	CKA_SERIAL_NUMBER	Serial number present in the user certificate	Certificate serial number (present in the digital certificate), encoded in DER format;
Keys attributes	CKA_KEY_TYPE	CKK_RSA	Specifies the key type. Was used CKK_RSA value for the key objects available for signature operations;
	CKA_MODULUS_BITS	Number of bits of the key modulus	Number of bits of the RSA private key modulus;
	CKA_MODULUS	Modulus value of the key, could be extracted from the certificate	Modulus n of the RSA private key;
	CKA_PUBLIC_EXPONENT	Public exponent value of the key, could be extracted from the certificate	value of the public exponent e of the RSA keypair;
	CKA_VERIFY	CK_TRUE	Specific public key flag for signature verification. Takes the value *true* for the CMD public key object;
	CKA_SENSITIVE	CK_TRUE	Flag specifying if the key is sensitive. Sensitive means that the actual value of the key is not exposed and so applications cannot extract the key value using its correponding CKA_VALUE attribute. This value is *true* for the private key object used;
	CKA_SIGN	CK_TRUE	Flag indicating that a certain key can be used to signature operations. This value is *true* for the CMD private key object used;
	CKA_EXTRACTABLE	CK_FALSE	Basically is used for the same purpose of the CKA_SENSITIVE attribute, is *false* the key value could not be extracted. This value is *false* for the CMD private key object;
	CKA_ALWAYS_AUTHENTICATE	CK_FALSE	Attribute used to force user authentication. This attribute was set to *false* in the key objects used, since we used an external authentication mechanism and so, the function *C_Login* do not need to be used;
	CKA_DERIVE	CK_FALSE	CK_TRUE if key supports key derivation
	CKA_LOCAL	CK_FALSE	CK_TRUE only if key was either: 1. generated locally (i.e., on the token) with a C_GenerateKey or C_GenerateKeyPair call, or 2. created with a C_CopyObject call as a copy of a key which had its CKA_LOCAL attribute set to CK_TRUE
	CKA_ENCRYPT	CK_FALSE	CMD private key was not used to encrypt data
	CKA_VERIFY_RECOVER	CK_FALSE	CK_TRUE if key supports verification where the data is recovered from the signature
	CKA_WRAP	CK_FALSE	CK_TRUE if key supports wrapping (i.e., can be used to wrap other keys), is not the case of CMD key
	CKA_SIGN_RECOVER	CK_FALSE	CK_TRUE if key supports signatures where the data can be recovered from the signature
	CKA_UNWRAP	CK_FALSE	CK_TRUE if key supports unwrapping (i.e., can be used to unwrap other keys)
	CKA_ALWAYS_SENSITIVE	CK_TRUE	The attribute CKA_SENSITIVE from the private key object, is always the value CK_TRUE
	CKA_NEVER_EXTRACTABLE	CK_TRUE	The private key content cannot be extracted, at any time
	CKA_WRAP_WITH_TRUSTED	CK_FALSE	CK_TRUE if the key can only be wrapped with a wrapping key that has CKA_TRUSTED set to CK_TRUE. Even with trusted value true, the key cannot be used to wrap operations

Table 4.2: Attributes implemented for CMD objects.

4.3 Implementation

4.3.1 Installation and configurations

Regarding the detection of a CMD token by the PKCS#11 library, some pre-installations were mandatory. First, the user needs to download and install the packages that includes the new PKCS#11 library, a configuration python module and all the software developed used to enable the communication with the CMD signature service. After this, it is necessary to run the configuration script that will be used to download the user certificates to a default directory in the Linux file system. For this, the configuration file will create a HTTP client and call the *GetCertificateWithPin* service from the CMD API.

Additionally, if the user do not run the configuration file and the directory used to store certificates is empty (users can download the certificates them self's and put them manually in the directory), when the new PKCS#11 validates a presence of a CMD token (when applications call *C_GetSlotList*), it will be opened the GUI for credentials input in order to give the user the possibility to enable the CMD capabilities, downloading the corresponding user certificates.

4.3.2 PKCS#11 module

The first challenge in the development of this robust and well-structured PKCS#11 library was to find what functions of that API had been implemented and used by the CC PKCS#11 module. To face this problem it was developed a script capable of generate a clone API of the *libpteidpkcs11.so* library functions, this way creating a wrapper for the CC library. Additionally, wrapping enabled the possibility of maintaining all the previous functionalities to work with the CC in parallel with CMD.

In the presented solution, the focus was to find all the PKCS#11 methods used specifically for digital signatures operations and do the proper changes to enable the CMD services. Taking into consideration the methods implemented by the final library, every one are available when is used a CC smart card, but only a restricted list works with the CMD. This validation mechanism is done internally by the wrapper, so when the application calls a method that is not implemented for the CMD services, the value CKR_FUNCTION_NOT _SUPPORTED will be returned.

The library keeps a record of all sessions created with each slot as well as their actual state, used as validation when certain asserts are mandatory. For example, in order to guarantee that only the token supported operations are used and all the operations are properly initialized (using, for example, *C_Initialize*, *C_FindObjectsInit* or *C_SignInit* in the correct order).

For a complete step-by-step validation, it was keeped a set of structures storing all open sessions for each token, a list of all CMD cryptoki objects handles and a list with all cryptographic mechanisms allowed. Each session object, regarding the structure used in the PKCS#11 module, is composed by a *handle* used to identify a certain open session, a session *state* describing the last function where this session was used, and specifically in a signature operation it will also have a *hash_value* storing the *digest* value of the information to be signed, the hash mechanism to be used and lastly the signature value updated in the end of a signature process. Regarding all the *handles* used in the PKCS#11 library, they are unique identification numbers and both the application and the library will use them to refer to, for example, token objects or cryptoki sessions.

In chapter 2 was discussed all the PKCS#11 methods called in a normal digital signature operation using a smart card. Next, it will be given an in-depth description of all the differences in each one of them to ensure a full support for CMD cloud services. Additionally for debugging purposes, it was made use of a log file directly updated within the PKCS#11 module functions, providing important information about the complete interaction between an application and the PKCS#11 module regarding a signature operation.

1. **Initialize Cryptoki**
   This step of the process has the same behavior as the one described in the chapter 3 section 2.2. All the information about the Cryptoki library used and the list of functions implemented can be collected using the default CC library since at this stage there is no verification needed for a presence of a smart card in the system, so it can be used the wrapper to forward the request to the default PKCS#11 module.

   The initializing process include *C_Initialize*, *C_GetInfo* and *C_GetFuntionList*.

2. **Establish sessions with CMD slot**
   At this time, assuming that the user has already performed all the installations and configurations needed, will be given a slot option with the CMD token. The token and slot information is then stored along side the digital certificates as well as the signing mechanisms supported by the CMD (but not including the private information). The new PKCS#11 module will also show all the Portuguese CC smart cards connected via a smart card reader. In the following section will be described the functions used to list the available slots and establish sessions with them.

   2.1 *C_GetSlotList*: when called, the library will validate the presence of the CMD directory created, containing all certificates and general information needed such as slot description, and present the CMD slot as a option. After, the wrapper will forward the request to the CC default PKCS#11 module for smart card detection. The list with all available slot identifiers will be returned to the application.

   2.2 *C_GetSlotInfo*: method used to get slot information, and if the provided handle, within the arguments, corresponds to the CMD slot, the information from the general file will be returned to the application. In case that the handle coincide with a smart card slot, the request is forwarded to the CC PKCS#11 library.

   2.3 *C_GetMechanismList*: this function will return the list of signing mechanisms supported by CMD services (there is no official list of mechanisms in the manual provided or in any online documentation, so we tested some of the most used ones). The list in Table 4.3 contains the supported mechanisms.

   2.4 *C_OpenSession*: used to open sessions between a slot and an application. Applications normally use one session for an entire interaction with a certain token or use one session for each operation needed (e.g. one session for searching objects, one session for the signing process and another one for signature verification). The session handle will be saved in the corresponding token structure with an *open* state.

   2.5 *C_Login*, called to authenticate the user, but since that was created an external authentication process, this function does nothing.

3. **Objects search**

   When is detected a CMD token, the new library software will create 5 *cryptoki* objects based on the certificate information (stored in the directory described above), providing the necessary data for signing and signature verification operations. The number of objects created is based on information presented in the CC smart card for digital signatures. In this way, are present three digital *X.509 Certificates* (a user certificate, a CA certificate and a ROOT certificate), an RSA public key and an RSA private key. Usually, in a signing process, the application starts with a certificate chain verification (using the three certificates provided) and then tries to validate the RSA key pair provided before initiate any signature operation, to guarantee that the public key provided to the application matches the correspondent private key used in the signature operation.

   3.1 *C_FindObjectsInit*: when this function is called by applications, the library validates the object search template and compares it to all the CMD objects created within the CMD slot. If there is a match, is returned a CKR_OK value. The session *search_type* value is updated with the corresponding object type that the application is looking for.

   3.2 *C_FindObjects*: when called, returns the object *handles* for the corresponding objects that match the template used in *C_FindObjectsInit*.

   3.3 *C_FindObjectsFinal*: used to end an object search operation for the session within the function argument.

4. **Extract attribute information from objects**

   The application will start a attribute search in order to extract all the information necessary to perform a digital signature, including attributes from both the certificates and the RSA keypair.

   Regarding the attributes of this private object, the application can only access attributes provided by the public key and the provided certificates, as all the other ones are unavailable. All attributes necessary for signing operations returned by the consecutive *C_GetAttributeValue* calls are according to the user certificates retrieved from the CMD service, every other attribute search (not used for signatures) returns a CK_UNAVAILABLE_INFORMATION value. In Table 4.2 is described all attributes implemented for CMD objects regarding the necessary ones to perform a digital signature (detail attribute description in [11]).

5. **Signing operation**

   After requesting all CMD private key object attributes needed to perform a signature operation, the application will then call the *C_SignInit* method in order to start the signing process. As previously described, the application can choose a single or multipart signing processes, using *C_Sign* or *C_SignUpdate* respectively.

   5.1 *C_SignInit*: function used to start a signature operation, where the application provide, as arguments, the session handle, the private key object handle and the

signing mechanism to be used. In order to properly initialize a signature process, the object handle provided need to match one of the private key objects handle, either the CMD private key or, after forwarding the request, the private key present in the CC smartcard. Additionally, is initialized a digest variable accordingly with the mechanism passed in the function arguments (using OpenSSL library functions). All this information is stored within the respective session structure.

5.2 *C_Sign*: after initializing a signature operation, the application will call this method, where it receives the data to sign. First, is generated the data hash using digest functions from the OpenSSL library (https: //www.openssl.org), next the python module will be launched and are created two Linux pipes for Inter Process Communication. Now with the python module running, are passed the signature request including the hashed data, the hash length and the hashing mechanism used. The python module will return the signature value after receiving it from the CMD cloud service. The signature and corresponding length are returned to the application. Concerning the *python module* life cycle, after sending the signature value, this process is closed, together with the Linux pipes previously created.

5.3 *C_SignUpdate*, starts or continues a multi-part signature operation where this method will be called multiple times. Are passed as arguments a data block and its size. Are updated the hash value in every iteration until *C_SignFinal* is called.

5.4 *C_SignFinal*, ends a multi-part signature operation and launch the *python module* (equal process as the one described in *C_Sign*). Returns the matching signature from all the data received in the multiple *C_SignUpdate* calls. Regarding the *python module* life cycle, after sending the signature value, this process is closed, together with the Linux pipes previously created.

6. **Finalize and close sessions**

6.1 *C_CloseSession* or *C_CloseAllSessions*: called when application want to close a session/sessions with the CMD token. Calling this method will shut down all pending operations between the application and the token (in the specified session) and all the cryptoki objects created. Sessions are removed from the stored session handle list created by the new library.

6.2 *C_Finalize*: the default CC library is used for this function, since was also used *C_Initialize* for the cryptoki initialization.

Mechanisms
CKM_SHA1_RSA_PKCS
CKM_SHA256_RSA_PKCS
CKM_SHA384_RSA_PKCS
CKM_SHA512_RSA_PKCS
CKM_RIPEMD160_RSA_PKCS

Table 4.3: Mechanism supported by the CMD token.

4.3.3 Python module

For the signature assemble step and in order to communicate with the CMD services, it was develop a mechanism capable of connect the local PKCS#11 module with a cloud API, and for that, it was chosen a python module, capable of doing secure requests to CMD test API using SOAP messages through a SOAP client. This protocol is used in web services to specify the message structure as a XML document, this way SOAP allows clients to invoke web services and receive responses independent of languages and platforms, and it can operate over all the communication protocols used.

In this way, this script can be launched by the new PKCS#11 module using a *execl(3)* command (Linux command used to launch a process), when any CMD service is needed, being responsible for all HTTP requests, data encryption and UI manipulation.

When launched, the python module starts listening to the created Linux pipes waiting for any requests from the PKCS#11 library. And so, after receiving one request, the information is divided, creating a dictionary composed by the message digest, message digest length and signature mechanism. If any of these three parameters are missing from the original message read from the pipe, the operation is canceled since every part is mandatory.

After the request validation, the python module will open a Qt UI window for the user to submit his CMD credentials (phone number and signature PIN)and the document name, and encrypt them using the provided AMA public key. After gathering all information fields necessary, it is opened a SOAP client session with HTTP basic authentication using a user-name and password supplied by AMA for testing purposes. Before moving on, the hash type prefix is added to the received document hash from the PKCS#11 module. More information regarding the hash prefix is described in Table 4.4.

Following that, the client will call *CCMovelSign* service from the pre-production URl [1], passing in the SOAP message structure, the Application Id (also provided by AMA, unique for this project), the document name (acquired with user input), the document hash with hash type prefix, encrypted user phone number and encrypted CMD signature PIN. Is returned a response from the CMD service with the process id, a status code (with value 200 if valid) and a description message. As a following step, the user will receive a text message (to the phone number passed in the signing request) with the OTP and the document name for the initialized signature operation. Than, it will be open a new Qt UI window where the user should input the OTP value received and then the python module will call *ValidateOtp* service passing the application id, the process id (received in the response from *CCMovelSign* service) and the OTP value encrypted with AMA public key. The response from this service will be a status message validating the signature operation and the actual signature value. More information regarding CMD services responses and requests structure is described in [7].

In the last step, our python module will send the signature back to the PKCS#11 module using the Linux pipes previously created, along with the status value (*SUCCESS* or *ERROR*) and the signature length.

[1] *https://preprod.cmd.autenticacao.gov.pt/Ama.Authentication.Frontend/SCMDServicesvc?wsdl*

Hash Algorithm	Hash Prefix
SHA-1	0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2b, 0x0E, 0x03, 0x02, 0x1A, 0x05, 0x00, 0x04, 0x14
SHA-256	0x30, 0x31, 0x30, 0x0d, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01, 0x05, 0x00, 0x04, 0x20
SHA-384	0x30, 0x41, 0x30, 0x0d, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x02, 0x05, 0x00, 0x04, 0x30
SHA-512	0x30, 0x51, 0x30, 0x0d, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x03, 0x05, 0x00, 0x04, 0x40
RIPEMD-160	0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2B, 0x24, 0x03, 0x02, 0x01, 0x05, 0x00, 0x04, 0x14

Table 4.4: Prefixes used supported by the developed python module and PKCS#11 library

4.3.4 IPC (Inter-process communication)

In order to establish a communication channel between the PKCS#11 module and the python module process an IPC infrastructure was implemented.

The IPC chosen for this project was Linux Pipes for its simplicity and easy integration in multiple programming languages, and with Pipes we can guarantee a private communication channel between the two process that can be opened or closed at any instance, since they depend exclusively from the processes that we control. Regarding the definition of an Linux Pipe, it is an unidirectional communication channel between two or more programs used in Linux systems.

To create a bidirectional IPC we created two pipes within the PKCS#11 library just before launch the python module process. One starting at the PKCS#11 module and ending in the python module, and another one in the opposite direction. The first one is used to send signing requests to the python module and the second one sends back the signature value from the python module. In table 4.5 is described the message format used in each one of the implemented Pipes.

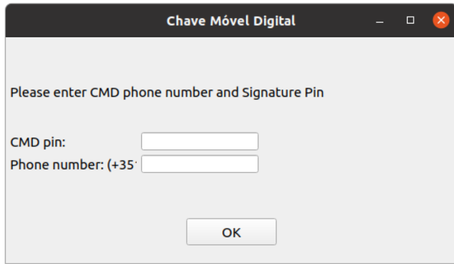
Pipe	Direction	Message format
1	PKCS#11 ->python module	Format: [request] | [data hash] | [hash length] | [hash algorithm] e.g. "SignatureRequest | uqphfquwiefbeqwp | 20 | SHA1".
2	python module ->PKCS#11	Format: [status] | [signature] | [signature length] e.g. "SUCCESS | qeroghrçiugbqrçigubef... | 384".

Table 4.5: Pipe message structure.

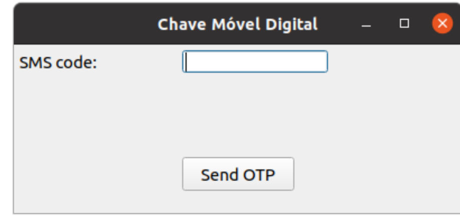
4.3.5 GUI (Graphical user interface)

Qt for python was the programming tool used to develop the Graphical User Interface for user credential inputs. Since the interface was not an important feature of the presented project, was prioritized to find an easy and rapid way [34] to design a basic set of windows capable of receiving user inputs and give a brief description of the operation actual status.

The approach regarding the GUI design, was chosen following the solution implemented for the middleware provided by AMA. In this way, the developed GUI is formed by two windows, the first one with a phone number, a CMD PIN and document name submission fields and the other one to the user submit the OTP received by text message after a valid signature request.



(a) PIN and phone number input window.



(b) OTP input window.

Figure 4.3: Graphical User Interface for SmartCMD.

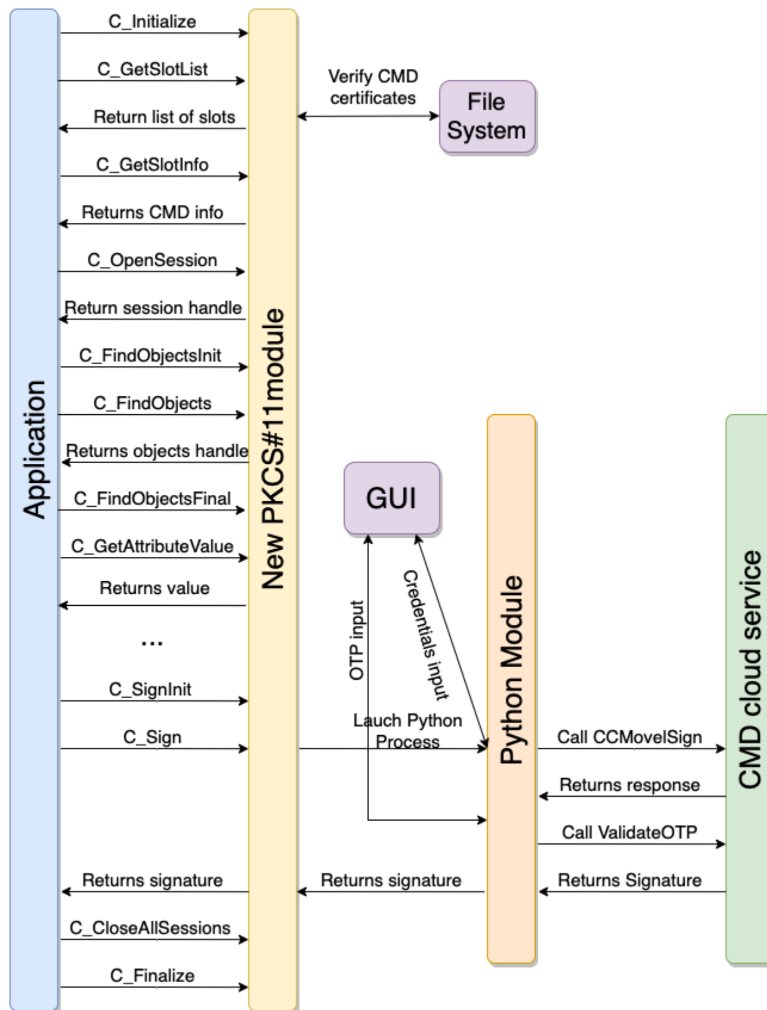


Figure 4.4: Detail signature operation using the developed software.

Chapter 5

Tests & results

In order to test the implemented modules, including the PKCS#11 library and the python module, was picked a set of tools and applications that can work with a PKCS#11 API and with which we could perform digital signatures. Before start the PKCS#11 module tests, we need to guarantee that the python module was well implemented and all of the calls made to the CMD service API have the expected result. Additionally, and in order to assemble a list of algorithms supported by CMD, we performed a set of tests making signature requests with numerous distinct hash algorithms. The final algorithm list is described in Table 5.1.

After conclusion of the python module tests, we chosen the *pkcs11-tool* application to start testing the PKCS#11, since it provides the perfect set of individual tests. In this way, offering us the possibility to ensure that every implemented function was well developed and can handle a hand full of different workflows.

Once we had a complete implementation of the API, we tried to find some test suites capable of do a overall run including all the developed functions. But since we could not find any suitable script and even AMA could not provide us any testing tool, we decided to move directly to tests using another applications that could provide us a more complete and demanding interaction.

And so, as the next step, we take the solution forward and test it against more rigid and commercial applications. For this, the first option was the Portuguese "autenticação.gov" application, but unfortunately, it does not use a PKCS#11 module for cryptographic operations, and so we could not test the implemented solution.

The research for an appropriate application continues, and after some search, we encountered two applications that could use PKCS#11 and had digital signature capabilities. In one hand, MyPDFSigner as command-line tool that can actually sign PDF documents, and in the other hand, in the first instance, it was picked the Adobe Acrobat Reader application to test our module, but due to the lack of support for Linux (the last update was in 2013), it cannot perform a viable signature using the native application, therefore it was exchanged by PDFStudio, developed by Qoppa Software [35], that is a very similar high-end application also used worldwide and with excellent support for Linux systems. With this last one, since the proof of concept was achieve and the solution was well tested, we decide to end the test section for this project. In the following sections will be present a extensive description of all the tests performed.

Hash functions	CMD service
SHA 1	✔
SHA 224	✔
SHA 256	✔
SHA 384	✔
SHA 512	✔
MD5	✔
RIPEMD 160	✔

Table 5.1: Mechanism supported by the CMD token.

5.1 OpenSC pkcs11-tool

The PKCS#11 module was initially tested using pkcs11-tool from OpenSC [36]. This tool is a command-line application used to manage and use PKCS#11 security tokens, having a built-in capability to perform tests on tokens and reports if the operation was carried out successfully. Table 5.2 depicts a list of tested PKCS#11 functionalities.

With pkcs11-tool the new module functions can be tested almost individually outputting the perfect control over the tests performed. These PKCS#11 tests were used to validate the functionality of CMD services and all software developed. Additionally this tool contains a predefined test suite (script with multiple individual operation tests) containing the expected outputs for each cryptographic operation.

Tests can be run through PKCS#11 interface using CMD for cryptographic operations. Running pkcs11-tool command against the implemented new PKCS#11 library uses the exact same function interfaces as an high-end application would use.

Starting with the individual tests, it was used the pkcs11-tool to test all the functions within a signature operation, this includes not just the actual signature process (functions *C_SignInit*, *C_Sign*, *C_SignUpdate* and *C_SignFinal*) but also operations such as acquiring information regarding the slot and respective token, opening the cryptoki *sessions*, object search and fetching of object attributes.

It is relevant to take in consideration that this tool performs just basic validations over the implemented functions, and if the module had been design just with this small step of asserts, it would certainly rise future problems when testing it with other applications. In this way, and as previously specified, this tool acts only as an entry point for testing the library before moving forward to a more restrict and demanding application.

After testing all the operations individually with unitary tests, it was used the predefined pkcs11-tool test suite to perform an overall run over the implemented operations, giving a more overall view of the module capabilities and its robustness. As can be seen in Table 5.2, the function *C_Login* was implemented and tested, but this function is not used in a normal signature operation since we guarantee that the flag *CKF_LOGIN_REQUIRED* from the

token is set to false. However, when using the pkcs11-tool test suite, this function is used and so we implemented a empty function that returns CKR_OK, just to surpass this iteration.

As expected, and after some changes, all unitary tests perform perfectly with the new module giving the pretended outputs, and the test suite was completed with no errors (take in account that many operations may not work since they were not implemented for the CMD token). These results gave the confidence needed to take the developed solution to a more advanced and complex environment.

Additionally, *pkcs11-tool* offered us the possibility to test the python module individually, this way running a hand full of hashing algorithms tests to gather a reasonably among of them, in order to provide some algorithms diversification on the developed software. In total, were tested the following algorithms: SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, and RIPEMD-160. As all of them work using the CMD service API, we offered every one as a hash algorithm possibility in the new PKCS#11 module.

PKCS#11 implemented functions	Interface tests
C_Initialize	✔
C_Finalize	✔
C_GetInfo	✔
C_GetFunctionsList	✔
C_GetSlotList	✔
C_GetSlotInfo	✔
C_GetTokenInfo	✔
C_GetMechanismList	✔
C_GetMechanismInfo	✔
C_OpenSession	✔
C_CloseSession	✔
C_CloseAllSessions	✔
C_Login	✔
C_GetSessionInfo	✔
C_GetAttributeValue	✔
C_FindObjectsInit	✔
C_FindObjects	✔
C_FindObjectsFinal	✔
C_SignInit	✔
C_SignUpdate	✔
C_Sign	✔
S_SignFinal	✔

Table 5.2: PKCS#11 functions tested using pkcs11-tool.

5.2 Autenticação.GOV application

After completing the development of the PKCS#11 module and performing all the requested tests with the pkcs11-tool, the first choice as a high-level application for library integration was the "authentication.Gov", as it integrates correctly all the digital signature functionality of both CC and CMD. In this way, we replaced the original PKCS#11 module of CC by the new library, in order to also integrate the CMD services through PKCS#11.

However, after some verifications on the libraries used by the application, it was concluded that it does not use directly any PKCS#11 module to perform cryptographic operations. Additionally, and in order to ensure that the application does not actually use the CC PKCS#11 module, as it could dynamically load the library, this module was completely removed from the original directory. Even so, the application worked normally, in this way can be affirmed that it does not use a PKCS#11 module at all to perform the digital signatures with the CC. Consequently, it cannot use CMD through our PKCS#11 module.

5.3 MyPDFSigner

MyPDFSigner [1] is a command-line tool to digitally sign PDF documents using tokens stored in a security device, such as a smart card, like a citizen card issued by many governments in Europe, or a USB token.

This tool has also an option to use a PKCS#12 file (a *.pfx or *.p12 file) and supports Time Stamping (RFC 3161), encryption, bulk signing and signature verification. Additionally, it supports cloud digital signature providers, since it works with providers certified by eIDAS (EU regulation 910/2014). Regarding the visualization of signed PDF documents, is currently available using, for example, Adobe Acrobat application.

To use a PKCS#11 security device in Linux, we need to configure a file in a specific directory within the application installation folder [2] containing the following parameters [37]:

- *signerpem*, being the path to a file with the certificate associated with the token private key in PEM format. We used the certificates previously downloaded when the Smart CMD tool was installed.
- *capem*, being the path to a file containing the certificates of the chain of trust from the Root CA certificate down to the certificate of the authority that issued the signer certificate. The certificates should be in PEM format again.
- *engine*, being the path to the PKCS#11 engine, the module that allows OpenSSL to interact with the token through its own PKCS#11 driver (OpenSSL ↔ Engine ↔ Driver ↔ Token). In Linux it is generally in /usr/lib/engines/engine_pkcs11.so or /usr/lib64/openssl/engines/pkcs11.so.
- *p11url*, being the URL of the token private key. Can be obtained using the *p11tool* command, “p11tool –provider /path/to/pkcs11-module –list-keys”;
- *library*, being the path to the PKCS#11 module associated with the token.
- *hashalgo*, being the algorithm to be used. By default MyPDFSigner uses SHA256 but that can be replaced by SHA1, SHA224, SHA384 or SHA512, with this entry.
- *embedcrl*, being used in situations where an OCSP status is not available, either because the OCSP end point is not present in the signer certificate or for any other reason (like being unable to reach the OCSP server). There is the option of embedding a CRL file, if available.

[1] https://www.kryptokoder.com/download.html
[2] /usr/local/mypdfsigner/tests/testconfiguration.conf

The application starts then with the cryptoki API initialization (calling *C_Initialize*), gets all implemented functions (C_GetFunctionsList) and checks the available slot with a valid token(C_GetSlotList and C_GetSlotInto to verify the CKF_TOKEN_PRESENT flag within the token information).

In the following step, using the configuration file information, the application will perform a certificate chain validation and save all the information needed to sign the document using the provided token private key. Usually, the applications make several *C_GetAttributeValue* calls to acquire all the information, but MyPDFSigner gets this information by reading the certificates directly. C_OpenSession will be called to establish a new session between the application and the CMD token, being this session used to get a valid private key object handle.

In this way, the application starts the object search using C_FindObjectInit and and a template with private key attributes, following a call to C_FindObjects to return the private key handle and finally ends the search operation using C_FindObjectFinal. As previously described, applications can use just one session for the entire set of operations or use multiple ones, MyPDFSigner uses the second option since the session used to perform the object search is closed and a new one will be open to perform the signature operation.

Starting the signature operation, the C_SignInit is called with the private key handle and the signature algorithm to be used, this application uses a multi-part signature mechanism (as described in the chapter 2, PKCS#11 module section). The python module will be called after calling the C_SignFinal function to communicate with the CMD signature service and acquire the actual signature value.

After receiving the signature, MyPDFSigner will try to use the OCSP URL present in the certificate to verify the certificates revocation status. Since the CMD user certificates provided by the developer do not have OCSP information, so the *embedcrl* option was used in the configuration file. In the last step, the application will apply the signature to the pretended PDF document. Even with the problems with the OCSP link, all the tests performed using the MyPDFSigner tool were positive and well accomplished.

After some research and discussion, it was concluded that this application was not a very good example of a default interaction with a PKCS#11 module because of the active use of the configuration file and the direct access to the certificate files. In this way it was decided to take a step forward and test this module with the Adobe Acrobat Reader direct concurrent, the PDFStudio desktop application.

5.4 PDFStudio

A lot of challenges were faced regarding all the numerous interactions between the PDF-Studio application and the PKCS#11 new module, in comparison with the applications previously used. The first one was the debugging operations because now it would be used a graphical application instead of a command-line tool. To face this was created various log files to record all the functions called as well as all the information exchanged. In second place, this application, in comparison with the ones previously tested, perform a considerable number of object validations, including for all the certificates and RSA keys, leading to the addition of new attributes (not implemented until the tests performed with this application).

The initialization process is similar to the ones previously described but using the application GUI where, using a ”config.cfg” file, can be chosen the PKCS#11 library and, optionally,

the slot ID that we want to use. In the following step, and after cryptoki initialization, it is opened a session with the present CMD token, and then the application starts an object search operation. In Figure 5.1 it can be visualized the detection of the private key object created for the CMD token.

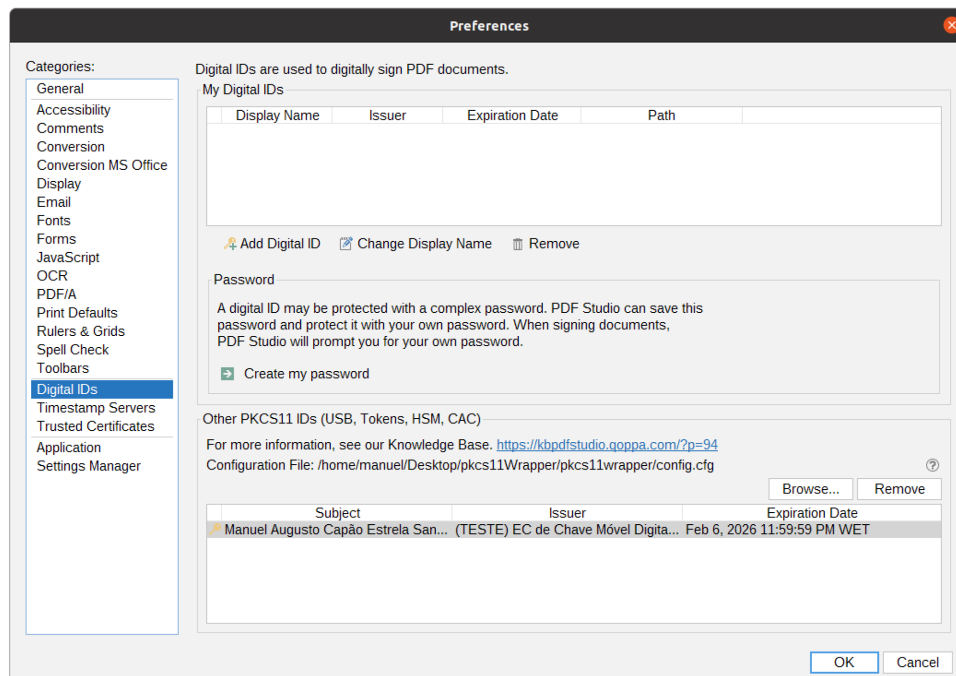


Figure 5.1: PDFStudio upload PKCS#11 library page.

Regarding the objects that the application looks for, we can assume that it tries to validate the certificate chain, since it starts to search certificates issuer and subject multiple times. At this point, and after the applications gather all the information it needs, regarding the different object attributes, it is ready to start a signature operation. For that, it is necessary to choose the pretended PDF document to be signed and the signature configurations as presented in Figure 5.2. On this page can be select the signature type, the information showed in the signature field and finally review the certificate issuer information.

Regarding the signature customization, do not exist the possibility to choose the pretended hash algorithm to use in the signing operation and there is no accessible documentation regarding this information. In this way, after contacting the PDFStudio support team about this topic, we were told that, by default PDFStudio selects the most secure hash algorithm supported accordingly with the PDF version (the application tries to use the most secure one first) of the uploaded document, so for versions below 1.6 just can work with SHA1, for version 1.6, can be used SHA256 and SHA1 and for version 1.7, SHA512, SHA384, SHA256, and SHA1 are supported.

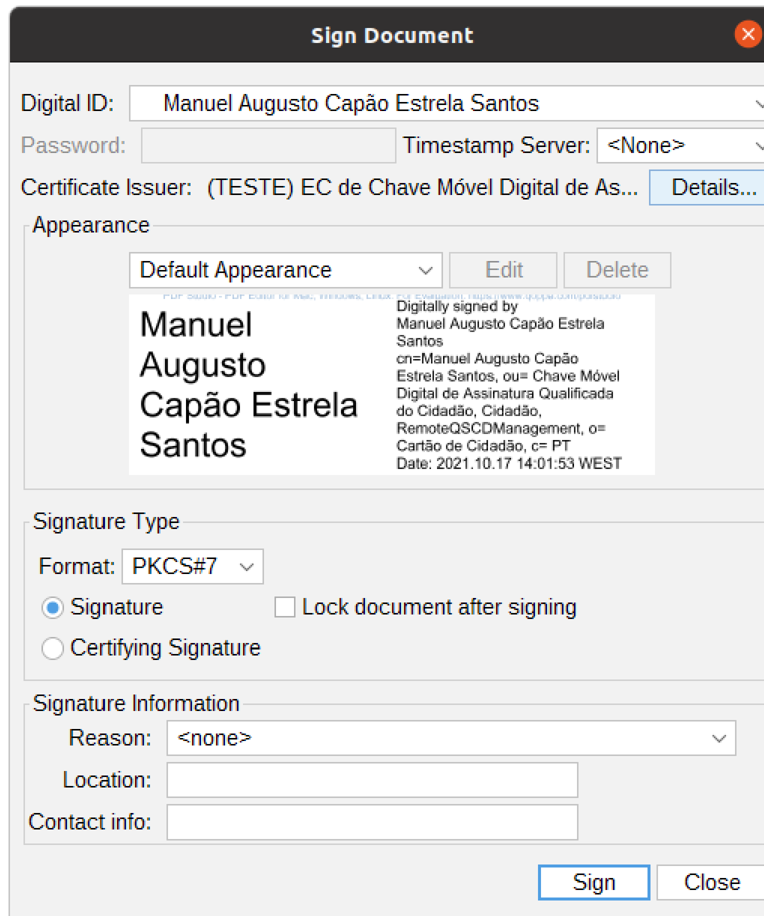


Figure 5.2: Signature configuration window.

After confirming the signing operation (clicking in the *Sign* button), the application will start a single part signing operation calling *C_SignInit*, passing as arguments the private key handle and the most secure algorithm supported (both by the document version and by the CMD token) and next calling *C_Sign*. The PDFStudio application, after receiving the signature value, will verify and add it to the desired PDF document, as displayed in Figure 5.3.

All the tests done with this application were successful in every part of the process, starting with object detection and certificate information, all the way down to the actual signing operation and signature verification.

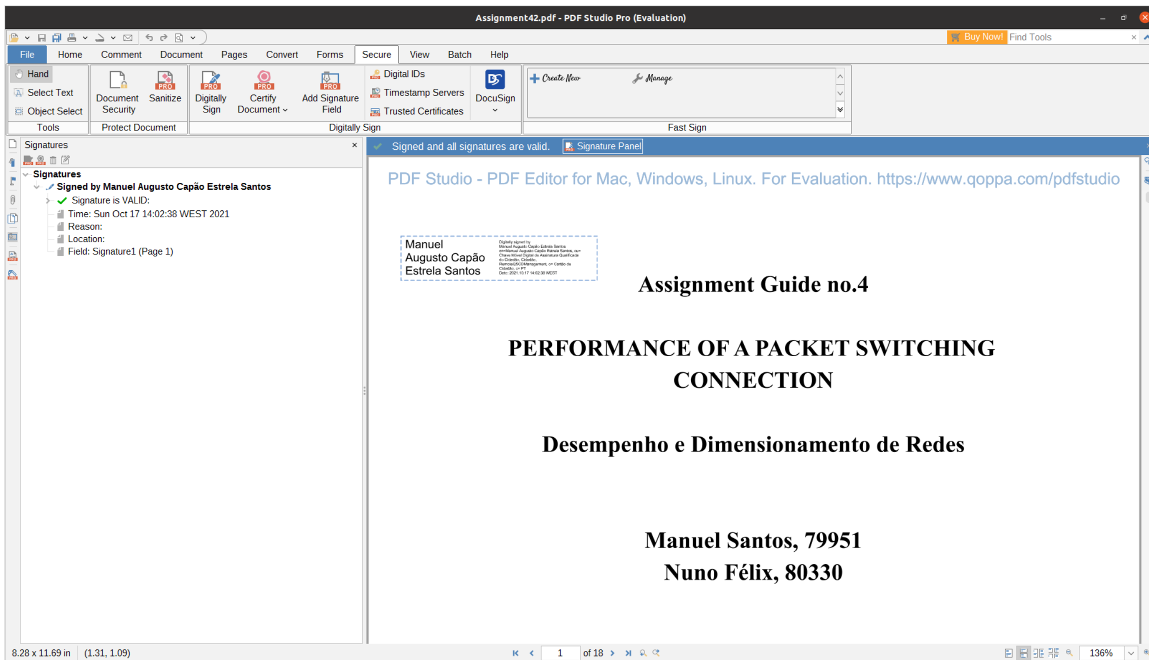


Figure 5.3: Result of a valid signature using CMD on PDFStudio app.

We also test the signature operations using the CC and everything worked just fine until we tried to sign a PDF with version 1.7. As referenced above, PDFStudio tries to use the most secure algorithm supported both by the PDF version and the token, and for a 1.7 PDF, SHA 512 is the algorithm used by default. But the Portuguese CC smartcard do not support SHA512 (this hash algorithm does not appear in the supported algorithms list for the smartcard), so the application should use SHA 384 instead.

However, this is not the actual application behavior, since in the list of supported mechanisms for the CC smartcard is present the CKM_RSA_PKCS mechanism (this mechanism is used when the application sends a already digested and prefixed information to be sign), and the application will tried to use this mechanism to sign the document using SHA 512 algorithm.

Unfortunately, the CC smartcard throws a "Device Error" message and the application cancels the signature operation. With this we can conclude that the CC PKCS#11 module is coherent since it neither support the SHA 512 algorithm internally nor support that algorithm via an already digested information (external digest done by the application). This test could also found an error in the PDFStudio application, because it should have a fallback behavior if anything wrong happens in a signature operation, as the situation previously described.

Chapter 6

Conclusions

In this thesis, we pretended to achieve a viable solution in order to use the Portuguese CMD services with an industry-standard cryptographic API, filling the lack of support for Linux applications. In this way, we developed a mechanism that offers the possibility to generate a valid digital signature using a cloud-based service through a PKCS#11 library. Regarding library versatility, we performed numerous validations across all the developed software, in order to ensure a proper integration with existing third-party applications that can work with a PKCS#11 library, and make sure that all the code and scripts are redundant and generalized in order to fulfill all use cases within multiple *Linux* devices (and not just in the tested system).

The new PKCS#11 module, as well as all the external software, has passed all proposed tests, where we used both command-line tools and with different Linux commercial applications with a graphic interface targeted to document manipulation. Therefore it can be concluded that this proof of concept has a high versatility regarding its integration with different Linux applications. With this project the main objective was accomplished, that is, to assemble a robust API with a behavior similar to the default CC provided library, changing the traditional smart-card cryptographic operations with a cloud-based service.

The developed solution can be used side-by-side with the CC library offering the full package for the Portuguese cryptographic signature services. And so, it can be used with applications that already support CC PKCS#11 module, like the ones described in chapter 5.

Some setbacks were encountered during the project implementation regarding all the documentation provided for the CC and CMD cryptographic services. There is no detailed information about the inside work developed for the CC PKCS#11 library, and since the "autenticação.gov" application does not use this library, makes all the perception process much harder and nearly impossible. This information, if available, could provide us the necessary help in developing the PKCS#11 module, more specifically regarding how the CC module handles all the necessary validations (e.g. validate the operations initialization) or how it deals with applications requests, such as in a attribute search operation. In this way, the lack of information, lead us to develop a wrapper in order to, in a first instance, study the CC PKCS#11 library behavior when interacting with applications.

Regarding CMD and its API services, it exists some setbacks mainly in availability which is derived from the nonexistence of a public and accessible manual about the integration process. Consequently, if some information is needed, the only solution is contacting the CMD support team, which delays the project considerably. Additionally, in the provided manual exists a

lack of information about the supported hash algorithms as well as the possible signature customization.

Regarding the tremendous quantity of real-world applications that this project enables, there is still a lot of features and use cases that can be added in the future, regarding fault-tolerance and interoperability across different operating systems.

In order to take this project to the next level, will be necessary some work scaling the PKCS#11 module usability. The first step would be to extend integration across different operating systems, not just for the PKCS#11 module but for all the developed complementary software (python module, IPC and GUI). The second update would be a fault-tolerance mechanism both in the API module and on the python module, allowing the possibility to recover from bad request errors or HTTP connection problems. The final step, and the most ambitious, would be decoupling the different developed modules, granting that the final PKCS#11 API could work not just with the CMD services but with any cloud-based signature service by attaching a custom drive for a cloud signature service capable of communicating with the pretended cloud service provider. This solution would open a vast number of possibilities for exploring cryptographic cloud-based signature services all around the world.

Bibliography

[1] Elena-Cristina Ruica, Mihai-Lica Pura, and Iulian Aciobanitei. Implementing Cloud Qualified Electronic Signatures for Documents using Available Cryptographic Libraries: A Survey. In *2020 13th International Conference on Communications (COMM)*, pages 113–118, 2020.

[2] Iulian Aciobanitei, Lorena Leahu, and Mihai Pura. A PKCS#11 Driver for Cryptography in the Cloud. In *2018 10th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–4, 2018.

[3] Iulian Aciobanitei, Paul Danut Urian, and Mihai Pura. A Cryptography API: Next Generation Key Storage Provider for Cryptography in the Cloud. In *2018 10th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–4, 2018.

[4] Leitold Herbert and Daniel Konrad. Qualified remote signatures – solutions, its certification, and use. In *Smartcard Workshop Tagungsband*, pages 219–231, feb 2019.

[5] Cryptomatic. Cloud Signing vs. Smartcard Signing. In *[online] Available: https://www.cryptomathic.com/hubfs/Documents/White_Papers. Accessed: 2021-09-29*, 2015.

[6] João Paulo Barraca. WIP: Support for Portuguese Electronic ID Cards . In *[online] Available: https://github.com/frankmorgner/vsmartcard/pull/208. Accessed: 2021-10-16*, 2021.

[7] Filipe Leitão, Jorge Basílio, Adriano Pires, André Vasconcelos, Bruno Teixeira, and Ricardo Conceição. Chave Móvel Digital - Especificação dos serviços de Assinatura. Manual de utilização dos serviços CMD, 3 2020.

[8] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. 26, 1983.

[9] D. Eastlake and P. Jones. RFC3174: US Secure Hash Algorithm 1 (SHA1), 2001.

[10] Bart Preneel, Antoon Bosselaers, and Hans Dobbertin. The cryptographic hash function RIPEMD-160, 1997.

[11] OASIS Standard. PKCS#11 Cryptographic Token Interface Profiles Version 3.0. In *https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.0/pkcs11-profiles-v3.0.html*, 2020.

[12] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal Analysis of PKCS#11. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 331–344, 2008.

[13] Agência para a Modernização Administrativa. Guia prático de utilização do cartão de cidadão. In *[online] Available: https://www. autenticacao.gov.pt/web/guest/documentos*, 2007.

[14] André Zúquete. *Segurança em Redes Informáticas.* FCA, 4ª edition, 2013.

[15] Agência para a Modernização Administrativa. Autenticação.gov , 2007.

[16] Maria Teresa Queiroz Machado Urbano Ferreira. Portuguese Citizen Card and Digital Mobile Key: the trust required of the citizen. 2021.

[17] C. Landauer and K.L. Bellman. Wrappings for software development. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 3, pages 420–429 vol.3, 1998.

[18] European Union. Regulation (EU) No 910/2014 of the European Parliament and of the Council of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC. In *eIDAS Regulation*, 2014.

[19] Florian Reimair. Cloud-based signature solutions: a survey. In *Technical report Secure Information Technology Center Austria*, 2014.

[20] Christof Rath, Simon Roth, Harald Bratko, and Thomas Zefferer. Encryption-Based Second Authentication Factor Solutions for Qualified Server-Side Signature Creation. In Andrea Kő and Enrico Francesconi, editors, *Electronic Government and the Information Systems Perspective*, pages 71–85, Cham, 2015. Springer International Publishing.

[21] Christof Rath, Simon Roth, Manuel Schallar, and Thomas Zefferer. A Secure and Flexible Server-Based Mobile eID and e-Signature Solution. In *2014 The Eighth International Conference on Digital Society IARIA*, 2014.

[22] Clemens Orthacker, Martin Centner, and Christian Kittl. A Qualified Mobile Server Signature. In Kai Rannenberg, Vijay Varadharajan, and Christian Weber, editors, *Security and Privacy – Silver Linings in the Cloud*, pages 103–111, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[23] Cloud Signature Consortium. Cloud Signature Consortium Standard Architectures Protocols and API Specifications for Remote Signature Applications public pre-release version 0.1.7.9. In *[online] Available: http://www.cloudsignatureconsortium.org/.*, pages 1–4, 2017.

[24] Andreas Fuchsberger. Microsoft CryptoAPI. *Information Security Technical Report*, 2(2):74–77, 1997.

[25] Kyungroul Lee, Youngjun Lee, Junyoung Park, Kangbin Yim, and Ilsun You. Security Issues on the CNG Cryptography Library (Cryptography API: Next Generation). In *2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 709–713, 2013.

[26] Thomas Zefferer, Arne Tauber, Bernd Zwattendorfer, and Klaus Stranacher. *Qualified PDF signatures on mobile phones.* 2012.

[27] Amir Sayegh and Mahmoud El-Hadidi. A Modified Secure Remote Password (SRP) Protocol for Key Initialization and Exchange in Bluetooth Systems. volume 2005, pages 261–269, 01 2005.

[28] Ahmed Alqattaa and Andreas Aßmuth. Analysis of the Internet Security Protocol TLS Version 1.3. 01 2019.

[29] Martin Centner. XML Advanced Electronic Signatures (XAdES). *Dipl.-Ing. Dr.techn. Reinhard Posch Supervisor: Ass.-Prof. Dipl.-Ing. Dr.techn. Peter Lipp*, 10 2003.

[30] Hrvoje Brzica, Boris Herceg, and Hrvoje Stančić. Long-term Preservation of Validity of Electronically Signed Records. 11 2013.

[31] Mehran Alidoost Nia, Ali Sajedi, and Aryo Jamshidpey. An Introduction to Digital Signature Schemes. 04 2014.

[32] Cryptomatic. eIDAS Compliant Remote eSigning. In *[online] Available: https://www.cryptomathic.com/hubfs/Documents/White_Papers. Accessed: 2021-09-29*, 2017.

[33] Cryptomatic. Achieving Real-World Crypto-Agility: a Guide for Banks and Financial Institutions . In *[online] Available: https://www.cryptomathic.com/hubfs/Documents/White_Papers. Accessed: 2021-09-29*, 2021.

[34] Mark Summerfield. Rapid GUI programming with Python and Qt: the definitive guide to PyQt programming. 01 2007.

[35] Qoppa Software. PDFStudio application page . In *[online] Available: https://www.qoppa.com/pdfstudio/. Accessed: 2021-10-16*, 2021.

[36] Olaf Kirch. pkcs11-tool(1) - Linux man page . In *[online] Available: https://linux.die.net/man/1/pkcs11-tool. Accessed: 2021-10-16*, 2021.

[37] KryptoKoder. Mypdfsigner use manual. In *[online] Available: https://www.kryptokoder.com/manual.html. Accessed: 2021-10-16*, 2021.

Appendix A

CMD: Especificação dos serviços de Assinatura

The CMD service can be used through a API described in the following pages. In this project the API was accessed using a SOAP client in a python program.

Take in considerations that the documentation provided may not include specific descriptions, such as the signature algorithms supported by the CMD service, an so in order to get this information, is mandatory to contact directly the responsible entity (AMA).

Chave Móvel Digital

Especificação dos serviços de Assinatura

Versão <V1.12>

Agência para a Modernização Administrativa I.P.

Referências a outros Documentos

Ref.	Descrição	Autor

Registo de Revisões

Data	Versão	Descrição	Autor
10-05-2016	V0.1	Versão Inicial	Filipe Leitão
11-11-2016	V0.2	Versão Inicial	Jorge Basílio
28-03-2017	V0.3	Versão Inicial	Jorge Basílio
09-11-2017	V0.4	Atualização dos Serviços	Adriano Pires
13-04-2018	V0.9	Revisão e atualização do WSDL Adição de informação de autenticação	Adriano Pires
16-04-2018	V1.0	Revisão documento	André Vasconcelos
08-05-2018	V1.1	Alteração da especificação de assinatura de múltiplos documentos Atualização do WSDL	Bruno Teixeira
08-05-2018	V1.2	Esclarecimento, na introdução, da ordem de invocação das operações do serviço	Bruno Teixeira
24-05-2018	V1.3	Atualização de lista de erros	Adriano Pires
05-06-2018	V1.4	Atualização Serviço CCMovelMultipleSign Atualização ficheiro wsdl	Adriano Pires

Data	Versão	Descrição	Autor
24-07-2008	V1.5	Alteração do tipo do campo Pin	Bruno Teixeira
29-09-2008	V1.6	Alteração da ordem dos campos no tipo SignRequest, de forma estarem coerentes com o WSDL	Bruno Teixeira
20-12-2018	V1.7	Novo serviço para receber os valores do PIN, UserId e OTP encriptados	Filipe Leitão
26-03-2019	V1.8	Atualização ficheiro WSDL Alteração do tipo dos campos cifrados	Ricardo Conceição
17-09-2019	V1.9	Esclarecimentos sobre a geração de hash	Bruno Teixeira
23-09-2019	V1.10	Melhoramento códigos erro	Ricardo Conceição
11-12-2019	V1.11	Atualização ficheiro WSDL Atualização de métodos existentes Introdução de novos métodos	Ricardo Conceição
06-03-2020	V1.12	Adicionada secção informações úteis	Ricardo Conceição

Lista de Distribuição

Nome	Organização	email
André Vasconcelos	AMA I.P.	andre.vasconcelos@ama.pt

Índice

1 INTRODUÇÃO ... 6

2 SERVIÇOS ... 7

2.1 SCMDSERVICE - SERVIÇO DE ASSINATURA QUALIFICADA ... 8

2.1.1 SCMDSign - Assinatura de Hash ... 8

2.1.1.1 SignRequest ... 8

2.1.1.2 SignStatus ... 9

2.1.2 GetCertificate – Obtém certificado do cidadão ... 10

2.1.3 ValidateOtp – Validação de código enviado para o cidadão ... 10

2.1.3.1 SignResponse ... 10

2.1.3.2 HashStructure ... 11

2.1.4 SCMDMultipleSign - Assinatura de múltiplos Hash ... 11

2.1.4.1 MultipleSignRequest ... 11

2.1.4.2 HashStructure ... 12

2.1.4.3 SignStatus ... 12

2.1.5 GetCertificateWithPin – Obtém certificado do cidadão com o PIN de assinatura ... 12

2.1.6 ForceSMS – Reenvia um SMS com o código de validação para o utilizador ... 13

3 GERAÇÃO DO HASH DO DOCUMENTO ... 14

4 INFORMAÇÕES ÚTEIS ... 15

5 ANEXOS ... 16

5.1 CÓDIGOS DE ERRO ... 16

5.2 WSDL 16

1 Introdução

A Chave Móvel Digital vem massificar o processo de autenticação e assinatura eletrónica qualificada do Cidadão.

O presente documento visa especificar o serviço para integração de sistemas externos para realização de assinaturas qualificadas através da Chave Móvel Digital (CMD). Este serviço irá disponibilizar as seguintes operações, que devem ser invocados pela ordem indicada abaixo:

1. GetCertificate: obtém certificado do cidadão;
2. SCMDSign: quando se pretende assinar um único documento, deve ser utilizada esta operação que recebe o hash do documento a assinar;
2. SCMDlMultipleSign: quando se pretende assinar vários documentos, deve ser utilizada esta operação que recebe a lista dos hash dos documentos a assinar;
3. GetCertificateWithPin: obtém o certificado do cidadão com o pin de assinatura associado.
4. ForceSMS: força o envio de um SMS com um novo código OTP, associada a uma operação em processo.
5. ValidateOtp: último passo do processo, que valida o código de segurança enviado e devolve o hash assinado, uma lista de hash assinados ou o certificado do cidadão, conforme ter sido invocada anteriormente a operação SCMDSign, SCMDMultipleSign ou GetCertificateWithPin.

2 Serviços

Nesta secção é descrita a especificação do serviço SCMDService. Os sistemas externos que desejem efetuar a integração com este serviço devem implementar os seguintes protocolos para a comunicação:

- Comunicação HTTPS com basic authentication;
- Mensagem SOAP;

As credenciais para a autenticação serão disponibilizadas pela AMA aquando dos trabalhos de integração.

As operações descritas, irão implementar criptografia assimétrica para encriptação e desencriptação dos dados sensíveis introduzidos pelo Cidadão (nº de Telemóvel, PIN de Assinatura e OTP). A CMD irá disponibilizar a chave pública através de um certificado X.509 para que os sistemas externos efetuem a encriptação da informação, com RSA, que será desencriptada pela CMD com a chave privada.

Endpoints

DEV	Serviço: https://dev.cmd.autenticacao.gov.pt/Ama.Authentication.Frontend/SCMDService.svc Certificado: a definir (apenas acessível dentro da rede da AMA)
PPR	https://preprod.cmd.autenticacao.gov.pt/Ama.Authentication.Frontend/SCMDService.svc Certificado: a definir
PRD	https:// cmd.autenticacao.gov.pt/Ama.Authentication.Frontend/SCMDService.svc Certificado: a definir

2.1 *SCMDService - Serviço de Assinatura Qualificada*

O serviço *SCMDService* irá disponibilizar 6 operações, a operação de assinatura de hash de 1 documento, a operação de assinatura de várias hash, a operação de validação de código de segurança OTP (*One Time Password*), a operação de obter o certificado do cidadão, a operação de obter o certificado do cidadão com recurso a PIN, e a operação de forçar um SMS para um processo em curso.

Para mais informações sobre os dados do serviço consulte a secção 4 deste documento.

2.1.1 SCMDSign - Assinatura de Hash

Operação	SCMDSign
Parâmetro de Entrada	SignRequest (2.1.1.1)
Parâmetro de Saída	SignStatus (0)

2.1.1.1 *SignRequest*

Parâmetros	Tipo	Obrigatório?	Descrição
ApplicationId	byte[]	Sim	Identificador da aplicação que efetua o *request*
DocName	string	Não	Nome do Documento ou identificador para permitir ao Cidadão identificar o ato que vai originar a assiantura
Hash	byte[]	Sim	Hash da informação sobre a qual vai ser gerada a assinatura
Pin	base64String(cifrado)	Sim	Código PIN do utilizador
UserId	base64String(cifrado)	Sim	Indentificador da conta do utilizador (ex.: Nº de Telemóvel: "+351 966666666")

2.1.1.2 *SignStatus*

Parâmetros	Tipo
ProcessId	string
Code	string
Message	string
Field	string
FieldValue	string

2.1.2 GetCertificate – Obtém certificado do cidadão

Operação	GetCertificate
Parâmetro de Entrada	byte[] applicationId base64String userId (cifrado)
Parâmetro de Saída	string certificate

2.1.3 ValidateOtp – Validação de código enviado para o cidadão

Operação	ValidateOtp
Parâmetro de Entrada	base64String code (cifrado) string processId byte[] applicationId
Parâmetro de Saída	SignResponse (2.1.3.1)

2.1.3.1 *SignResponse*

Parâmetros	Tipo
Signature	Byte[]
ArrayOfHashStructure	List<HashStructure>
SignStatus	Objecto referido no ponto 2.1.1.2
certificate	String

2.1.3.2 *HashStructure*

Parâmetros	Tipo	Obrigatório?	Descrição
Signature	byte[]	Sim	Assinatura do documento
Name	string	Sim	Nome do documento
id	string	Sim	Indentificador do documento

2.1.4 SCMDMultipleSign - Assinatura de múltiplos Hash

Operação	CCMovelMultipleSign
Parâmetro de Entrada	MultipleSignRequest List<HashStructure>
Parâmetro de Saída	SignStatus (0)

2.1.4.1 *MultipleSignRequest*

Parâmetros	Tipo	Obrigatório?	Descrição
ApplicationId	byte[]	Sim	Identificador da aplicação que efetua o *request*
Pin	base64String (cifrado)	Sim	Código PIN do utilizador
UserId	base64String (cifrado)	Sim	Indentificador da conta do utilizador (Nº Telemóvel: "+351 966666666")

2.1.4.2 *HashStructure*

Parâmetros	Tipo	Obrigatório?	Descrição
Hash	byte[]	Sim	Hash da informação sobre a qual vai ser gerada a assinatura
Name	String	Sim	Nome do documento
id	String	Sim	Indentificador do documento

2.1.4.3 *SignStatus*

Parâmetros	Tipo
`ProcessId`	string
`Code`	string
`Message`	string
`Field`	string
`FieldValue`	string

2.1.5 GetCertificateWithPin – Obtém certificado do cidadão com o PIN de assinatura

Operação	GetCertificateWithPin
Parâmetro de Entrada	byte[] applicationId base64String userId (cifrado) Base64String pin (cifrado)
Parâmetro de Saída	SignStatus (0)

2.1.6 ForceSMS – Reenvia um SMS com o código de validação para o utilizador

Operação	ForceSMS
Parâmetro de Entrada	byte[] applicationId base64String userId (cifrado) String processId
Parâmetro de Saída	SignStatus (0)

3 Geração do Hash do documento

A geração do hash deve ser feita conforme o "PKCS #1: RSA Cryptography Specifications Version 2.2", ponto 9.2 até ao passo 2:

- https://tools.ietf.org/html/rfc8017#page-45

A título de exemplo, o AlgorithmIdentifier para um hash SHA-256 seria o seguinte:

```
unsigned char[] sha256SigPrefix =
  { 0x30, 0x31, 0x30, 0x0d, 0x06, 0x09,
    0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01,
    0x05, 0x00, 0x04, 0x20};
```

O hash enviado para os serviços deve ser o prefixo seguido do hash do documento.

4 Informações úteis

Esta secção serve para proceder ao esclarecimento sobre os tipos de dados do serviço de assinatura qualificada.

Base64String (cifrado) – Cifrar informação com a chave pública do certificado disponibilizado pela AMA e converter o resultado dessa cifra numa string base64.

Byte[] (Byte Array) – Informação convertida em Byte Array.

5 Anexos

5.1 *Códigos de Erro*

Código	Descrição
200	OK
500's	Erros de sistema
800's	Parâmetros inválidos
801	Pins não correspondem
802	OTP inválido
817	Ocorre quando o serviço de assinatura do SCMD está inativo. O cidadão não recebe o OTP.
900	Erro genérico

5.2 *WSDL*

SCMDService.wsdl

Appendix B

PKCS#11 Wrapper API

As described in the previous sections, was used a PKCS#11 wrapper for the CC library provided, in order to, in a first instance use it as a study tool to understand the interaction between the applications and the PKCS#11 API.

In this appendix chapter is presented the code for the PKCS#11 wrapper used in a early stage of the project.

```
1  #include "bootstrap.c"
2  #include "logger.h"
3
4  void
5  resolve()
6  {
7    _funcs.C_CancelFunction = C_CancelFunction;
8    _funcs.C_CloseAllSessions = C_CloseAllSessions;
9    _funcs.C_CloseSession = C_CloseSession;
10   _funcs.C_CopyObject = C_CopyObject;
11   _funcs.C_CreateObject = C_CreateObject;
12   _funcs.C_Decrypt = C_Decrypt;
13   _funcs.C_DecryptDigestUpdate = C_DecryptDigestUpdate;
14   _funcs.C_DecryptFinal = C_DecryptFinal;
15   _funcs.C_DecryptInit = C_DecryptInit;
16   _funcs.C_DecryptUpdate = C_DecryptUpdate;
17   _funcs.C_DecryptVerifyUpdate = C_DecryptVerifyUpdate;
18   _funcs.C_DeriveKey = C_DeriveKey;
19   _funcs.C_DestroyObject = C_DestroyObject;
20   _funcs.C_Digest = C_Digest;
21   _funcs.C_DigestEncryptUpdate = C_DigestEncryptUpdate;
22   _funcs.C_DigestFinal = C_DigestFinal;
23   _funcs.C_DigestInit = C_DigestInit;
24   _funcs.C_DigestKey = C_DigestKey;
25   _funcs.C_DigestUpdate = C_DigestUpdate;
26   _funcs.C_Encrypt = C_Encrypt;
27   _funcs.C_EncryptFinal = C_EncryptFinal;
28   _funcs.C_EncryptInit = C_EncryptInit;
29   _funcs.C_EncryptUpdate = C_EncryptUpdate;
30   _funcs.C_Finalize = C_Finalize;
31   _funcs.C_FindObjects = C_FindObjects;
32   _funcs.C_FindObjectsFinal = C_FindObjectsFinal;
33   _funcs.C_FindObjectsInit = C_FindObjectsInit;
34   _funcs.C_GenerateKey = C_GenerateKey;
35   _funcs.C_GenerateKeyPair = C_GenerateKeyPair;
```

```
36   _funcs.C_GenerateRandom = C_GenerateRandom;
37   _funcs.C_GetAttributeValue = C_GetAttributeValue;
38   _funcs.C_GetFunctionList = C_GetFunctionList;
39   _funcs.C_GetFunctionStatus = C_GetFunctionStatus;
40   _funcs.C_GetInfo = C_GetInfo;
41   _funcs.C_GetMechanismInfo = C_GetMechanismInfo;
42   _funcs.C_GetMechanismList = C_GetMechanismList;
43   _funcs.C_GetObjectSize = C_GetObjectSize;
44   _funcs.C_GetOperationState = C_GetOperationState;
45   _funcs.C_GetSessionInfo = C_GetSessionInfo;
46   _funcs.C_GetSlotInfo = C_GetSlotInfo;
47   _funcs.C_GetSlotList = C_GetSlotList;
48   _funcs.C_GetTokenInfo = C_GetTokenInfo;
49   _funcs.C_Initialize = C_Initialize;
50   _funcs.C_InitPIN = C_InitPIN;
51   _funcs.C_InitToken = C_InitToken;
52   _funcs.C_Login = C_Login;
53   _funcs.C_Logout = C_Logout;
54   _funcs.C_OpenSession = C_OpenSession;
55   _funcs.C_SeedRandom = C_SeedRandom;
56   _funcs.C_SetAttributeValue = C_SetAttributeValue;
57   _funcs.C_SetOperationState = C_SetOperationState;
58   _funcs.C_SetPIN = C_SetPIN;
59   _funcs.C_Sign = C_Sign;
60   _funcs.C_SignEncryptUpdate = C_SignEncryptUpdate;
61   _funcs.C_SignFinal = C_SignFinal;
62   _funcs.C_SignInit = C_SignInit;
63   _funcs.C_SignRecover = C_SignRecover;
64   _funcs.C_SignRecoverInit = C_SignRecoverInit;
65   _funcs.C_SignUpdate = C_SignUpdate;
66   _funcs.C_UnwrapKey = C_UnwrapKey;
67   _funcs.C_Verify = C_Verify;
68   _funcs.C_VerifyFinal = C_VerifyFinal;
69   _funcs.C_VerifyInit = C_VerifyInit;
70   _funcs.C_VerifyRecover = C_VerifyRecover;
71   _funcs.C_VerifyRecoverInit = C_VerifyRecoverInit;
72   _funcs.C_VerifyUpdate = C_VerifyUpdate;
73   _funcs.C_WaitForSlotEvent = C_WaitForSlotEvent;
74   _funcs.C_WrapKey = C_WrapKey;
75 }
76
77
78 CK_RV C_CancelFunction ( CK_SESSION_HANDLE p1 )
79 {
80   bootstrap();
81   LOG(C_CancelFunction);
82   return funcs->C_CancelFunction(p1);
83 }
84 CK_RV C_CloseAllSessions ( CK_SLOT_ID p1 )
85 {
86   bootstrap();
87   LOG(C_CloseAllSessions);
88   return funcs->C_CloseAllSessions(p1);
89 }
90 CK_RV C_CloseSession ( CK_SESSION_HANDLE p1 )
91 {
92   bootstrap();
```

```
93   LOG(C_CloseSession);
94   return funcs->C_CloseSession(p1);
95 }
96 CK_RV C_CopyObject ( CK_SESSION_HANDLE p1, CK_OBJECT_HANDLE p2,
     CK_ATTRIBUTE_PTR p3, CK_ULONG p4, CK_OBJECT_HANDLE_PTR p5 )
97 {
98   bootstrap();
99   LOG(C_CopyObject);
100  return funcs->C_CopyObject(p1, p2,  p3,  p4,  p5 );
101 }
102 CK_RV C_CreateObject ( CK_SESSION_HANDLE p1, CK_ATTRIBUTE_PTR p2, CK_ULONG p3,
      CK_OBJECT_HANDLE_PTR p4 )
103 {
104   bootstrap();
105   LOG(C_CreateObject);
106   return funcs->C_CreateObject(p1, p2,  p3,  p4 );
107 }
108 CK_RV C_Decrypt ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3,
     CK_BYTE_PTR p4, CK_ULONG_PTR p5 )
109 {
110   bootstrap();
111   LOG(C_Decrypt);
112   return funcs->C_Decrypt(p1, p2,  p3,  p4,  p5 );
113 }
114 CK_RV C_DecryptDigestUpdate ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG
     p3, CK_BYTE_PTR p4, CK_ULONG_PTR p5 )
115 {
116   bootstrap();
117   LOG(C_DecryptDigestUpdate);
118   return funcs->C_DecryptDigestUpdate(p1, p2,  p3,  p4,  p5 );
119 }
120 CK_RV C_DecryptFinal ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG_PTR p3 )
121 {
122   bootstrap();
123   LOG(C_DecryptFinal);
124   return funcs->C_DecryptFinal(p1, p2,  p3 );
125 }
126 CK_RV C_DecryptInit ( CK_SESSION_HANDLE p1, CK_MECHANISM_PTR p2,
     CK_OBJECT_HANDLE p3 )
127 {
128   bootstrap();
129   LOG(C_DecryptInit);
130   return funcs->C_DecryptInit(p1, p2,  p3 );
131 }
132 CK_RV C_DecryptUpdate ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3,
     CK_BYTE_PTR p4, CK_ULONG_PTR p5 )
133 {
134   bootstrap();
135   LOG(C_DecryptUpdate);
136   return funcs->C_DecryptUpdate(p1, p2,  p3,  p4,  p5 );
137 }
138 CK_RV C_DecryptVerifyUpdate ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG
     p3, CK_BYTE_PTR p4, CK_ULONG_PTR p5 )
139 {
140   bootstrap();
141   LOG(C_DecryptVerifyUpdate);
142   return funcs->C_DecryptVerifyUpdate(p1, p2,  p3,  p4,  p5 );
```

```
143 }
144 CK_RV C_DeriveKey ( CK_SESSION_HANDLE p1, CK_MECHANISM_PTR p2,
      CK_OBJECT_HANDLE p3, CK_ATTRIBUTE_PTR p4, CK_ULONG p5,
      CK_OBJECT_HANDLE_PTR p6 )
145 {
146   bootstrap();
147   LOG(C_DeriveKey);
148   return funcs->C_DeriveKey(p1, p2, p3, p4, p5, p6 );
149 }
150 CK_RV C_DestroyObject ( CK_SESSION_HANDLE p1, CK_OBJECT_HANDLE p2 )
151 {
152   bootstrap();
153   LOG(C_DestroyObject);
154   return funcs->C_DestroyObject(p1, p2 );
155 }
156 CK_RV C_Digest ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3,
      CK_BYTE_PTR p4, CK_ULONG_PTR p5 )
157 {
158   bootstrap();
159   LOG(C_Digest);
160   return funcs->C_Digest(p1, p2, p3, p4, p5 );
161 }
162 CK_RV C_DigestEncryptUpdate ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG
      p3, CK_BYTE_PTR p4, CK_ULONG_PTR p5 )
163 {
164   bootstrap();
165   LOG(C_DigestEncryptUpdate);
166   return funcs->C_DigestEncryptUpdate(p1, p2, p3, p4, p5 );
167 }
168 CK_RV C_DigestFinal ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG_PTR p3 )
169 {
170   bootstrap();
171   LOG(C_DigestFinal);
172   return funcs->C_DigestFinal(p1, p2, p3 );
173 }
174 CK_RV C_DigestInit ( CK_SESSION_HANDLE p1, CK_MECHANISM_PTR p2 )
175 {
176   bootstrap();
177   LOG(C_DigestInit);
178   return funcs->C_DigestInit(p1, p2 );
179 }
180 CK_RV C_DigestKey ( CK_SESSION_HANDLE p1, CK_OBJECT_HANDLE p2 )
181 {
182   bootstrap();
183   LOG(C_DigestKey);
184   return funcs->C_DigestKey(p1, p2 );
185 }
186 CK_RV C_DigestUpdate ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3 )
187 {
188   bootstrap();
189   LOG(C_DigestUpdate);
190   return funcs->C_DigestUpdate(p1, p2, p3 );
191 }
192 CK_RV C_Encrypt ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3,
      CK_BYTE_PTR p4, CK_ULONG_PTR p5 )
193 {
194   bootstrap();
```

```
195   LOG(C_Encrypt);
196   return funcs->C_Encrypt(p1, p2,  p3,  p4,  p5 );
197 }
198 CK_RV C_EncryptFinal ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG_PTR p3 )
199 {
200   bootstrap();
201   LOG(C_EncryptFinal);
202   return funcs->C_EncryptFinal(p1, p2,  p3 );
203 }
204 CK_RV C_EncryptInit ( CK_SESSION_HANDLE p1, CK_MECHANISM_PTR p2,
      CK_OBJECT_HANDLE p3 )
205 {
206   bootstrap();
207   LOG(C_EncryptInit);
208   return funcs->C_EncryptInit(p1, p2,  p3 );
209 }
210 CK_RV C_EncryptUpdate ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3,
      CK_BYTE_PTR p4, CK_ULONG_PTR p5 )
211 {
212   bootstrap();
213   LOG(C_EncryptUpdate);
214   return funcs->C_EncryptUpdate(p1, p2,  p3,  p4,  p5 );
215 }
216 CK_RV C_Finalize ( CK_VOID_PTR p1 )
217 {
218   bootstrap();
219   LOG(C_Finalize);
220   return funcs->C_Finalize(p1);
221 }
222 CK_RV C_FindObjects ( CK_SESSION_HANDLE p1, CK_OBJECT_HANDLE_PTR p2, CK_ULONG
      p3, CK_ULONG_PTR p4 )
223 {
224   bootstrap();
225   LOG(C_FindObjects);
226   return funcs->C_FindObjects(p1, p2,  p3, p4 );
227 }
228 CK_RV C_FindObjectsFinal ( CK_SESSION_HANDLE p1 )
229 {
230   bootstrap();
231   LOG(C_FindObjectsFinal);
232   return funcs->C_FindObjectsFinal(p1);
233 }
234 CK_RV C_FindObjectsInit ( CK_SESSION_HANDLE p1, CK_ATTRIBUTE_PTR p2, CK_ULONG
      p3 )
235 {
236   bootstrap();
237   LOG(C_FindObjectsInit);
238   return funcs->C_FindObjectsInit(p1, p2,  p3 );
239 }
240 CK_RV C_GenerateKey ( CK_SESSION_HANDLE p1, CK_MECHANISM_PTR p2,
      CK_ATTRIBUTE_PTR p3, CK_ULONG p4, CK_OBJECT_HANDLE_PTR p5 )
241 {
242   bootstrap();
243   LOG(C_GenerateKey);
244   return funcs->C_GenerateKey(p1, p2,  p3,  p4,  p5 );
245 }
246 CK_RV C_GenerateKeyPair ( CK_SESSION_HANDLE p1, CK_MECHANISM_PTR p2,
```

```
    CK_ATTRIBUTE_PTR p3, CK_ULONG p4, CK_ATTRIBUTE_PTR p5, CK_ULONG p6,
    CK_OBJECT_HANDLE_PTR p7, CK_OBJECT_HANDLE_PTR p8 )
247 {
248   bootstrap();
249   LOG(C_GenerateKeyPair);
250   return funcs->C_GenerateKeyPair(p1, p2,  p3,  p4,  p5,  p6,  p7,  p8 );
251 }
252 CK_RV C_GenerateRandom ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3 )
253 {
254   bootstrap();
255   LOG(C_GenerateRandom);
256   return funcs->C_GenerateRandom(p1, p2,  p3 );
257 }
258 CK_RV C_GetAttributeValue ( CK_SESSION_HANDLE p1, CK_OBJECT_HANDLE p2,
    CK_ATTRIBUTE_PTR p3, CK_ULONG p4 )
259 {
260   bootstrap();
261   LOG(C_GetAttributeValue);
262   return funcs->C_GetAttributeValue(p1, p2,  p3, p4 );
263 }
264 CK_RV C_GetFunctionList ( CK_FUNCTION_LIST_PTR_PTR p1 )
265 {
266   bootstrap();
267   LOG(C_GetFunctionList);
268   return funcs->C_GetFunctionList(p1);
269 }
270 CK_RV C_GetFunctionStatus ( CK_SESSION_HANDLE p1 )
271 {
272   bootstrap();
273   LOG(C_GetFunctionStatus);
274   return funcs->C_GetFunctionStatus(p1);
275 }
276 CK_RV C_GetInfo ( CK_INFO_PTR p1 )
277 {
278   bootstrap();
279   LOG(C_GetInfo);
280   return funcs->C_GetInfo(p1);
281 }
282 CK_RV C_GetMechanismInfo ( CK_SLOT_ID p1, CK_MECHANISM_TYPE p2,
    CK_MECHANISM_INFO_PTR p3 )
283 {
284   bootstrap();
285   LOG(C_GetMechanismInfo);
286   return funcs->C_GetMechanismInfo(p1, p2,  p3 );
287 }
288 CK_RV C_GetMechanismList ( CK_SLOT_ID p1, CK_MECHANISM_TYPE_PTR p2,
    CK_ULONG_PTR p3 )
289 {
290   // ao criar o slot, especificar quais os mecanismos que a CMD implementa
291   bootstrap();
292   LOG(C_GetMechanismList);
293     return funcs->C_GetMechanismList(p1, p2,  p3 );
294 }
295 CK_RV C_GetObjectSize ( CK_SESSION_HANDLE p1, CK_OBJECT_HANDLE p2,
    CK_ULONG_PTR p3 )
296 {
297   bootstrap();
```

```
298   LOG(C_GetObjectSize);
299   return funcs->C_GetObjectSize(p1, p2,  p3 );
300 }
301 CK_RV C_GetOperationState ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG_PTR
      p3 )
302 {
303   bootstrap();
304   LOG(C_GetOperationState);
305   return funcs->C_GetOperationState(p1, p2,  p3 );
306 }
307 CK_RV C_GetSessionInfo ( CK_SESSION_HANDLE p1, CK_SESSION_INFO_PTR p2 )
308 {
309   bootstrap();
310   LOG(C_GetSessionInfo);
311   return funcs->C_GetSessionInfo(p1, p2 );
312 }
313 CK_RV C_GetSlotInfo ( CK_SLOT_ID p1, CK_SLOT_INFO_PTR p2 )
314 {
315   bootstrap();
316   LOG(C_GetSlotInfo);
317   return funcs->C_GetSlotInfo(p1, p2 );
318
319 }
320 CK_RV C_GetSlotList ( CK_BBOOL p1, CK_SLOT_ID_PTR p2, CK_ULONG_PTR p3 )
321 {
322   bootstrap();
323   LOG(C_GetSlotList);
324   return funcs->C_GetSlotList(p1, p2,  p3 );
325
326 }
327 CK_RV C_GetTokenInfo ( CK_SLOT_ID p1, CK_TOKEN_INFO_PTR p2 )
328 {
329   bootstrap();
330   LOG(C_GetTokenInfo);
331   return _C_GetTokenInfo(p1,p2);
332 }
333 CK_RV C_Initialize ( CK_VOID_PTR p1 )
334 {
335   bootstrap();
336   LOG(C_Initialize);
337   return funcs->C_Initialize(p1);
338
339
340 }
341 CK_RV C_InitPIN ( CK_SESSION_HANDLE p1, CK_CHAR_PTR p2, CK_ULONG p3 )
342 {
343   bootstrap();
344   LOG(C_InitPIN);
345   return funcs->C_InitPIN(p1, p2,  p3 );
346 }
347 CK_RV C_InitToken ( CK_SLOT_ID p1, CK_CHAR_PTR p2, CK_ULONG p3, CK_CHAR_PTR p4
      )
348 {
349   bootstrap();
350   LOG(C_InitToken);
351   return funcs->C_InitToken(p1, p2,  p3,  p4 );
352 }
```

```
353 CK_RV C_Login ( CK_SESSION_HANDLE p1, CK_USER_TYPE p2, CK_CHAR_PTR p3,
      CK_ULONG p4 )
354 {
355   bootstrap();
356   LOG(C_Login);
357   return funcs->C_Login(p1, p2,  p3,  p4 );
358 }
359 CK_RV C_Logout ( CK_SESSION_HANDLE p1 )
360 {
361   bootstrap();
362   LOG(C_Logout);
363   return funcs->C_Logout(p1);
364 }
365 CK_RV C_OpenSession ( CK_SLOT_ID p1, CK_FLAGS p2, CK_VOID_PTR p3, CK_NOTIFY p4
      , CK_SESSION_HANDLE_PTR p5 )
366 {
367   bootstrap();
368     LOG(C_OpenSession);
369     return funcs->C_OpenSession(p1,p2,p3,p4,p5);
370 }
371 CK_RV C_SeedRandom ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3 )
372 {
373   bootstrap();
374   LOG(C_SeedRandom);
375   return funcs->C_SeedRandom(p1, p2,  p3 );
376 }
377 CK_RV C_SetAttributeValue ( CK_SESSION_HANDLE p1, CK_OBJECT_HANDLE p2,
      CK_ATTRIBUTE_PTR p3, CK_ULONG p4 )
378 {
379   bootstrap();
380   LOG(C_SetAttributeValue);
381   return funcs->C_SetAttributeValue(p1, p2,  p3,  p4 );
382 }
383 CK_RV C_SetOperationState ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3,
       CK_OBJECT_HANDLE p4, CK_OBJECT_HANDLE p5 )
384 {
385   bootstrap();
386   LOG(C_SetOperationState);
387   return funcs->C_SetOperationState(p1, p2,  p3,  p4,  p5 );
388 }
389 CK_RV C_SetPIN ( CK_SESSION_HANDLE p1, CK_CHAR_PTR p2, CK_ULONG p3,
      CK_CHAR_PTR p4, CK_ULONG p5 )
390 {
391   bootstrap();
392   LOG(C_SetPIN);
393   return funcs->C_SetPIN(p1, p2,  p3,  p4,  p5 );
394 }
395 CK_RV C_Sign ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3, CK_BYTE_PTR
      p4, CK_ULONG_PTR p5 )
396 {
397   bootstrap();
398   LOG(C_Sign);
399   return _C_Sign(p1, p2,  p3,  p4,  p5 );
400   //return funcs->C_Sign(p1, p2,  p3,  p4,  p5 );
401 }
402 CK_RV C_SignEncryptUpdate ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3,
       CK_BYTE_PTR p4, CK_ULONG_PTR p5 )
```

```
403 {
404   bootstrap();
405   LOG(C_SignEncryptUpdate);
406   return funcs->C_SignEncryptUpdate(p1, p2,  p3,  p4,  p5 );
407 }
408 CK_RV C_SignFinal ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG_PTR p3 )
409 {
410   bootstrap();
411   LOG(C_SignFinal);
412   return funcs->C_SignFinal(p1, p2,  p3 );
413 }
414 CK_RV C_SignInit ( CK_SESSION_HANDLE p1, CK_MECHANISM_PTR p2, CK_OBJECT_HANDLE
      p3 )
415 {
416   bootstrap();
417   LOG(C_SignInit);
418   return funcs->C_SignInit(p1, p2,  p3 );
419 }
420 CK_RV C_SignRecover ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3,
     CK_BYTE_PTR p4, CK_ULONG_PTR p5 )
421 {
422   bootstrap();
423   LOG(C_SignRecover);
424   return funcs->C_SignRecover(p1, p2,  p3,  p4,  p5 );
425 }
426 CK_RV C_SignRecoverInit ( CK_SESSION_HANDLE p1, CK_MECHANISM_PTR p2,
     CK_OBJECT_HANDLE p3 )
427 {
428   bootstrap();
429   LOG(C_SignRecoverInit);
430   return funcs->C_SignRecoverInit(p1, p2,  p3 );
431 }
432 CK_RV C_SignUpdate ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3 )
433 {
434   bootstrap();
435   LOG(C_SignUpdate);
436   return funcs->C_SignUpdate(p1, p2,  p3 );
437 }
438 CK_RV C_UnwrapKey ( CK_SESSION_HANDLE p1, CK_MECHANISM_PTR p2,
     CK_OBJECT_HANDLE p3, CK_BYTE_PTR p4, CK_ULONG p5, CK_ATTRIBUTE_PTR p6,
     CK_ULONG p7, CK_OBJECT_HANDLE_PTR p8 )
439 {
440   bootstrap();
441   LOG(C_UnwrapKey);
442   return funcs->C_UnwrapKey(p1, p2,  p3,  p4,  p5,  p6,  p7,  p8 );
443 }
444 CK_RV C_Verify ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3,
     CK_BYTE_PTR p4, CK_ULONG p5 )
445 {
446   bootstrap();
447   LOG(C_Verify);
448   return funcs->C_Verify(p1, p2,  p3,  p4,  p5 );
449 }
450 CK_RV C_VerifyFinal ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3 )
451 {
452   bootstrap();
453   LOG(C_VerifyFinal);
```

```
454   return funcs->C_VerifyFinal(p1, p2,  p3 );
455 }
456 CK_RV C_VerifyInit ( CK_SESSION_HANDLE p1, CK_MECHANISM_PTR p2,
       CK_OBJECT_HANDLE p3 )
457 {
458   bootstrap();
459   LOG(C_VerifyInit);
460   return funcs->C_VerifyInit(p1, p2,  p3 );
461 }
462 CK_RV C_VerifyRecover ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3,
       CK_BYTE_PTR p4, CK_ULONG_PTR p5 )
463 {
464   bootstrap();
465   LOG(C_VerifyRecover);
466   return funcs->C_VerifyRecover(p1, p2,  p3,  p4,  p5 );
467 }
468 CK_RV C_VerifyRecoverInit ( CK_SESSION_HANDLE p1, CK_MECHANISM_PTR p2,
       CK_OBJECT_HANDLE p3 )
469 {
470   bootstrap();
471   LOG(C_VerifyRecoverInit);
472   return funcs->C_VerifyRecoverInit(p1, p2,  p3 );
473 }
474 CK_RV C_VerifyUpdate ( CK_SESSION_HANDLE p1, CK_BYTE_PTR p2, CK_ULONG p3 )
475 {
476   bootstrap();
477   LOG(C_VerifyUpdate);
478   return funcs->C_VerifyUpdate(p1, p2,  p3 );
479 }
480 CK_RV C_WaitForSlotEvent ( CK_FLAGS p1, CK_SLOT_ID_PTR p2, CK_VOID_PTR p3 )
481 {
482   bootstrap();
483   LOG(C_WaitForSlotEvent);
484   return funcs->C_WaitForSlotEvent(p1, p2,  p3 );
485 }
486 CK_RV C_WrapKey ( CK_SESSION_HANDLE p1, CK_MECHANISM_PTR p2, CK_OBJECT_HANDLE
       p3, CK_OBJECT_HANDLE p4, CK_BYTE_PTR p5, CK_ULONG_PTR p6 )
487 {
488   bootstrap();
489   LOG(C_WrapKey);
490   return funcs->C_WrapKey(p1, p2,  p3,  p4,  p5,  p6 );
491 }
```

To complete the wrapping functionality was used the function *bootstrap()* described below to load the functions from the default CC PKCS#11 API.

```
1  #define ENV_LIB_PATH "PTEIDPKCS11_WRAPPER"
2  #define DEFAULT_LIB_PATH "/usr/local/lib/libpteidpkcs11.so"
3
4  static void
5  bootstrap()
6  {
7      void * handle;
8      char * libname;
9
10     CK_RV (*gfl) ( CK_FUNCTION_LIST_PTR_PTR );
11
12     if (funcs) return;
```

```
13
14
15      libname = getenv( ENV_LIB_PATH );
16      if (libname == 0) {
17          libname = DEFAULT_LIB_PATH;
18      }
19      handle = dlopen( libname, RTLD_NOW );
20      gfl = dlsym( handle, "C_GetFunctionList" );
21      gfl( &funcs );
22
23      resolve();
24      funcs->C_GetFunctionList = _C_GetFunctionList;
25
26      printf( "Bootstrap done!\n" );
27  }
```