BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

AUTOMATED TESTING OF SMART CARDS

AUTOMATIZOVANÉ TESTOVÁNÍ ČIPOVÝCH KARET

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — **PAVEL YADLOUSKI**
AUTOR PRÁCE

SUPERVISOR — **Ing. IVAN HOMOLIAK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2020/2021

Bachelor's Thesis Specification

Student: **Yadlouski Pavel**
Programme: Information Technology
Title: **Automated Testing of Smart Cards**
Category: Security

Assignment:

1. Get acquainted with principles of software and hardware testing as well as various testing types.
2. Study the standard PKCS#11 as well as example projects implementing it, such as OpenSC and SoftHSM.
3. Analyze the existing solutions for testing of smart cards, such as Smart Card Removinator.
4. Identify problems and limitations of existing solutions.
5. Design a new library for automated testing of smart cards, which will improve limitations of existing solutions and provide some extensions (e.g., integration to beakerlib).
6. Implement the library and demonstrate its usage by solving a real-world problem.
7. Compare the proposed library with existing solutions in terms of functionality and performance.

Recommended literature:

- PKCS#11 standard, https://www.cryptsoft.com/pkcs11doc/
- Smart Card Removinator, https://github.com/nkinder/smart-card-removinator
- Beakerlib, https://github.com/beakerlib/beakerlib
- OpenSC library, https://github.com/OpenSC/OpenSC
- SoftHSMv2 library, https://github.com/opendnssec/SoftHSMv2

Requirements for the first semester:

- Items 1 to 4.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Homoliak Ivan, Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 12, 2021
Approval date: November 11, 2020

Abstract

This bachelor thesis deals with automated testing of smart cards in Red Hat Enterprise Linux. The problem of manual testing is solved by creating a new testing library. This library is responsible for configuring the test environment and providing a way for the tester to interact with that environment in an automated way. As a result, we created a new universal library for testing the support of smart cards. The primary goal is to implement the testing library by itself, after that to transfer manual test into the code using created library and run those automated tests in Red Hat internal pipelines.

Abstrakt

Tato bakalářská práce se zabývá automatizovaným testováním podpory Smart Karet v RHEL. Problém manuálního testování je vyřešen vytvořením nové testovací knihovny. Tato knihovna je zodpovědná za konfiguraci testovacího prostředí a poskytuje testerovi rozhraní pro automatizovanou manipulaci s tímto prostředím. Jako výsledek jsme vytvořili univerzální knihovnu pro testování podpory smart karet. Primárním cílem je implementace samotné knihovny, pak následující převod existujících manuálních testů do kódu za pomoci teto knihovny a zprovozněni těchto testů ve vnitřní pipelině Red Hat.

Keywords

Smart cards, RHEL, automation, virtualization, hardware testing, testing library, removinator, virt_cacard.

Klíčová slova

Smart karty, RHEL, automatizace, virtualizace, testování hardweru, testovácí knihovna, removinator, virt_cacard.

Reference

YADLOUSKI, Pavel. *Automated Testing of Smart Cards.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ivan Homoliak, Ph.D.

Rozšířený abstrakt

Hlavní problém, který se snažíme vyřešit v tomto projektu je testovaní čipových karet. Nejvíc nás zajímá zejména automatizované integrační testovaní čipových karet v Red Hat Enterprise Linux. Do dnešního dne stav testovaní je takový, že všechno se testuje manuálně, a to je přes 200 testů.

Pro testování čipových karet potřebujeme řadu komponent, například GNOME Desktop Manager, smart card removinator, Kerberos server, Red Hat Certificate System server, a další. Však v tuto chvíli neexistuje žádný prvek, který by tyto všechny komponenty svázal a umožnil automatizaci procesu testování.

Cílem této bakalářské práce je navrhnout a implementovat takovou technologii, která by spojila všechny potřebné prvky pro testování a umožnila uživateli jednoduchou práci s těmito prvky.

Našim řešením je Python knihovna SCAutolib, do které jsou integrovány manipulační prostředky pro prvky testovacího prostředí. Kromě integrace už zmíněných systémů testovacího prostředí, pro usnadnění testování naše knihovna využívá knihovnu virt_cacard, která poskytuje virtuální čipové karty. Tato knihovna používá softwarovou implementaci Hardware Security Module – SoftHSM2. Sam o sobě SoftHSM2 neposkytuje dostačující funkcionalitu pro to, aby simuloval skutečné karty, proto je obalen pomocí knihovny lib_cacard aby přidala vlastnosti reálné karty do SoftHSM2 tokenu. Nicméně virtuální karta má svá omezení, a proto stále potřebujeme fyzické karty. Pro automatizaci s reálnými kartami využíváme speciální čítač čipových karet smart card removinator.

SCAutolib v sebe má prostředky i pro nastavení kompletního testovacího prostředí. Testovacím prostředím se myslí potřebné servisy, a to je minimálně Kerberos server, lokální Certifikační Autorita a Red Hat Certificate System server. Pro nastavení testovacího prostředí se používají skripty, napsané v Bash. Zajímavou špičkou naší knihovny bude to, že dokáže pracovat i s grafickým uživatelským rozhraním. Tato funkcionalita se plánuje integrovat s využitím openQA frameworku pro testování GUI aplikace.

Aktuálně knihovna SCAutolib má v sebe implementované scripty pro nasazení lokálního CA a následně vytvoření virtuální karty s certifikátem od lokální CA. Tyto scripty jsou volány prostřednictvím interních funkcí knihovny, aby uživatel mohl naplno využívat knihovnu přes python a nezabýval se přímým voláním Bash scriptu.

Pro manipulaci s virtuálními kartami je vytvořena třída se souvisejícími metodami (insert, remove). Pro nastavení způsobu přihlášení v RHEL se používá authselect balíček. Ten se v knihovně také představen třídou s odpovídajícími metodami (set, reset). Zmíněné třídy jsou používány v rámci kontextového manažeru. V knihovně je také přípustná funkcionalita, která přímo nepatří ani do jedné ze zmíněných skupin metod. To jsou funkce pro manipulaci se soubory (zálohování, editace konfiguračních soborů) a celkovou manipulaci s operačním systémem (restartovaní systémových servisů, přihlášení uživatelů).

Aktuální stav knihovny je dostačující na to, aby dokázali transformovat základní manuální testy na automatické. Samozřejmě, že čas potřebný na spuštění automatizovaných testů je výrazně menší v porovnání s manuální evokaci testů. I když před každým voláním automatických testů nějaký čas bude využit na nastavení systému, to stále zabere méně času a bude méně náchylné k chybám.

V planu do budoucna je rozšíření knihovny o integraci se čítačem smart card removinator. Tato integrace vyžaduje vetší prozkoumání existující knihovny na bezpečnou práci s removinatorem. Dalším bodem je integrace RCHS serveru do testovacího prostředí a propojení s Kerberos serverem. Pro podporu testovaní grafického rozhráni plánujeme integrovat i openQA framework.

Ve výsledku máme knihovnu, která poskytuje velkou sadu funkcí potřebných pro testování čipových karet. Tato knihovna může být využita nejen v interním pipelině Red Hat, ale i v jiných systémech na bázi RPM, kde je potřebný zásah čipových karet, jako například testování integrace čipových karet do desktopových aplikací jako například Firefox nebo poštovný klient.

Automated Testing of Smart Cards

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Ivan Homoliak, Ph.D.. The supplementary information was provided by Ivan Nikolchev and Jakub Jelen. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Pavel Yadlouski
May 9, 2021

Acknowledgements

My thanks go to my supervisor Ing. Ivan Homoliak, Ph.D. for the useful comments, remarks and engagement through the learning process of this term project. I would like to thank also Ivan Nikolchev and Jakub Jelen for providing technical consultations and reviews.

Contents

Chapter 1

Introduction

Our modern world requires us to be more secure than ever before. In the sphere of Information Technologies the question „how to keep private data secure?“ is still one of the most challenging. As time goes on, new technologies are invented to protect our information from being stolen and our identity from being compromised. These technologies should be reliable, secure, easy to use, and meet various standards. This bachelor thesis deals with one of the information security technologies called Smart Card. Smart cards were invented in the late '60s but they remain relevant today. The idea behind smart cards is to incorporate an integrated circuit chip onto a plastic card. Smart cards are commonly used today not only because of their technical abilities, but also because of their credit card size, so it is very easy to carry them with us anywhere.

Like any other technology, smart cards need to be thoroughly tested before deployment or release. For smart cards, this is true for both the hardware and software sides. This thesis aims to design and implement a new testing library that would be able to replace manual testing of smart cards with automated testing, and thus significantly contributes to usability and performance of testing. Since a smart card is a hardware device that is plugged into the computer via the physical smart card reader, this work also deals with virtualization technologies and software representation of smart cards.

Organization

The rest of this work is divided into 6 chapters (excluding Introduction).

The first part Chapter 2 of this thesis give necessary theoretical background knowledge about how smart cards work from the inside and some of their use cases. It also introduces the basics of technologies Section 2.5 and cryptographic standards Section 2.6 that are commonly used for security and smart cards.

The second part Chapter 3 presents an overview of advanced technologies related to smart cards. The main use case this thesis tries to deal with is using smart cards to authenticate and login into operating systems. To be able to do that we also need to get familiar with other technologies, such as Certificate Management System in the Section 3.1, Kerberos in the Section 3.4, and testing technologies in the Section 3.2.

As the main aim of the project is to design and implement a new python library for automated testing of the smart cards, the third part Chapter 4 deals with the architecture of our library and introduces it from different points of view. The next part Chapter 5 introduces how the library is implemented and discusses the most important features that

are included in the library. The limitations of the library are discussed in the Section 6.2 along with our plan for future development in the Section 6.1.

The last part Chapter 7 provides a summary of the whole work.

Chapter 2

Background

First of all, let us start with a must-have knowledge about smart cards. This chapter provides us with the necessary background knowledge for understanding smart cards. We introduce the basic definition of a smart card and talk about common use-cases and related technologies.

2.1 Smart Card

Smart Card (SC) is a hardware authorization device (token) with an embedded integrated circuit (IC) chip. Typically it is a plastic card of the credit card size. This integrated chip can store small chunks of data (for example, certificates) and perform different types of functionality (such as identification, authentication, etc.). Moreover, the smart card provides support for the strong single sign-on (SSO) authentication mechanism. All technical parameters are specified by standard ISO/IEC 7816 and its parts. In the Table 2.1 you can see parts of the ISO/IEC 7816 standard.

The motivation for using a smart card is to increase security by adding the second factor or replacing the authentication factor of knowledge (password) with a factor of possession (having a card). In the second case, you don't need to type in your password to login to unlock the required services because all the necessary information is stored on the smart card. In some cases, a PIN might also be required.

When used for identification, a certificate issued by a certificate authority is stored on the card. One possible smart card type is the Personal Identity Verification [11] card (PIV). This type of card is used in US Government, for example, and also stores certificates. In this work, we would deal also with open-source smart cards storing certificates.

The certificate is created by an authority organization using Certificate Management System (Section 3.1). For credit cards, this organization can be VISA[1] or MasterCard[2]. But any organization can create its Certificate Management System server and manage certificates by itself. In this work, for testing purposes, the Certificate Management System server is created with the open-source implementation of Certificate System – Dogtag [5].

[1] https://www.visa.com/
[2] https://www.mastercard.com/

Standard	Description
ISO/IEC 7816-1	specifies physical characteristics for cards with contacts
ISO/IEC 7816-2	specifies dimensions and location of the contacts
ISO/IEC 7816-3	specifies electrical interface and transmission protocols for asynchronous cards
ISO/IEC 7816-4	specifies organization, security and commands for interchange
ISO/IEC 7816-5	specifies registration of application providers
ISO/IEC 7816-6	specifies interindustry data elements for interchange
ISO/IEC 7816-7	specifies commands for structured card query language
ISO/IEC 7816-8	specifies commands for security operations
ISO/IEC 7816-9	specifies commands for card management
ISO/IEC 7816-10	specifies electrical interface and answer to reset for synchronous cards
ISO/IEC 7816-11	specifies personal verification through biometric methods
ISO/IEC 7816-12	specifies electrical interface and operating procedures for USB cards
ISO/IEC 7816-13	specifies commands for handling the life cycle of applications
ISO/IEC 7816-15	specifies cryptographic information application

Table 2.1: Parts of ISO/IEC 7816 standard.

2.2 Common Use Cases

ATM Withdrawal. Smart cards are used for personal identification and authentication. Different systems use smart cards on a daily basis. For example, a credit card as representative of a smart card is a trivial use case where the client needs to authenticate himself against ATM to get access to his bank account. As it was mentioned, a credit card stores personal information that is used by ATMs for user authentication. The authentication succeeds only if the certificate on the credit card is valid and the user enters a correct PIN code for a given card.

Physical Access Control. Another example is a situation where a user needs to get access to the office floor. The process of granting access can be the same as the credit card and ATM scenario (inserting the card to the card reader and then providing PIN). But some smart cards support contactless authentication. So, when an associate comes to the office door, the only thing he needs to do is to bring the card close enough to the appropriate card reader and this device will get the required certificates. After that associate may need to provide his PIN. This additional authentication check is required mostly in government organizations with a high level of security. Without a PIN requirement, the doors would open as soon as your certificates are validated.

Smart Card as the Second Factor. Nobody will be surprised with two-factor authentication (2FA) in the modern world. 2FA increases the security of the users in a relatively simple way for the users – adding one more security step to authentication. For example, the application can ask the user for a smart card PIN or just to insert the card by itself. But from the providers' side implementation of the 2FA can be problematic at least from the universality side – users don't want to use different 2FA implementations for each service.

For standardization of 2FA Universal second factor protocol (U2F) was developed by Fast IDentity Online (FIDO) Alliance[3]. The U2F protocol allows online services to augment the security of their existing password infrastructure by requiring a physical token, called an authenticator [29]. Smart card technology is the most capable of providing the highest level of security for FIDO implementations. The key to this capability is that the smart card holds the chip where FIDO implementation can be securely executed. So all cryptographic operations (such as key pair generation) can be transferred to the smart card, where those operations are optimized. Due to the secure elements inside the smart card, cryptographic operations are executed securely and efficiently.

This work deals with a use case when a person needs to login into the computer system using a smart card to get access to an organization's private services and data. Imagine your company has sensitive information and internal services, that must not be visible outside the company. The smart card provides a mechanism for the user to login into the organization's computer system. Also, the same card can be used in applications, such as the browser and email clients, for granting access to resources (private websites or servers through a web browser) or for providing digital signatures and encryption (sending encrypted and signed emails).

2.3 Types of Smart Cards

Like we mentioned before, smart cards can be used in several ways. Each use-case requires a different type of smart card.

2.3.1 Based on a Connection

Use-case with a credit card and ATM mostly requires so-called **contact** smart card[4]. That means that for using a card you also need to have a card reader where you can insert that card. Opposite to this type is a **contactless** smart card. That type does not require inserting a card into the card reader, just bringing it close enough to the corresponding type of the reader (use-case with getting access to office floor).

2.3.2 Based on the Processing Element

A different classification of the smart cards is based on the type of embedded integrated circuit inside the card. This integrated circuit can be a **memory** chip that stores a small chunk of data (up to 4KB) and can be a RAM, ROM, or EEPROM. Memory cards have no data processing power, just a simple logic for accessing memory cells.

Another type of smart card is Multifunction Card with a **microprocessor** (or micro-controller) inside with volatile memory. This microprocessor can be programmed by the user using Global Platform [8] protocol and applets written in Java, C# (.NET), or other more exotic types. For accessing objects on the card (data on the smart card is stored in the objects) PKCS #11 (Section 2.6) protocol is used. At this moment a smart card becomes a pocket-size computer with strong enough data processing capabilities right on the card. An example of such a Multifunction Card can be a debit card that is used for paying for public transport when we enter a tram or a bus and our card is connected with a ticket until we leave the transport (or until the last station). The chips of this category

[3] https://fidoalliance.org/

[4] Today contactless ATMs become more and more common also

have a variety of configurations. The chip can support Public Key Infrastructure (PKI) functions with on-board math co-processors or JavaCard[5] with virtual machine hardware blocks (for example MIFARE[6], IDEMIA[7]).

2.3.3 Tamper-resistant features

Like any other device with a processing unit, the smart card is also a target for attackers that aims to steal sensitive data from the smart card. Since around 1994, almost every type of smart card processor used in European, and later also American and Asian, pay-TV conditional-access systems have been successfully reverse-engineered. Compromised secrets have been sold in the form of illicit clone cards that decrypt TV channels without revenue for the broadcaster [14]. Here is a list of the most common techniques for breaking into smart cards:

- **Microprobing** techniques can be used to access the chip surface directly, thus we can observe, manipulate, and interfere with the integrated circuit.
- **Software** attacks use the normal communication interface of the processor and exploit security vulnerabilities found in the protocols, cryptographic algorithms, or their implementation.
- **Eavesdropping** techniques monitor, with high time resolution, the analog characteristics of all supply and interface connections and any other electromagnetic radiation produced by the processor during normal operation.
- **Fault generation** techniques use abnormal environmental conditions to generate malfunctions in the processor that provides additional access.

To overcome these vulnerabilities, design principles for tamper-resistant smart card processors were suggested [14, p8-11].

2.4 Symmetric and Asymmetric Cryptography

The base concept that needs to be clarified first of all is the types of cryptographic keys that are commonly used in general cryptography and smart cards. Symmetric and asymmetric encryption differs first of all in core idea and used algorithms for implementing this idea.

Symmetric encryption uses one key (symmetric key) both for data encryption and decryption. Symmetric encryption use algorithms such as Advanced Encryption Standard (AES) [17], International Data Encryption Algorithm (IDEA) [10], and others. The usual key length for the symmetric key is 128, 192, 256 bits (e.g for the AES algorithm). Symmetric encryption is faster than asymmetric one, but one drawback is that the secret key needs to be securely shared between the involved parties.

Asymmetric encryption on other hand uses two keys - private and public key (key pair). The idea behind those two keys is that data encrypted by one of those keys can be decrypted on with a second key, e.g message encrypted with a public key can be decrypted only with a corresponding private key. So, for establishing secure communication, all hosts should share their public keys. When sharing a public key we also need to present proof of

[5] https://www.oracle.com/java/technologies/java-card-tech.html
[6] https://www.mifare.net/en/
[7] https://www.idemia.com/smart-onecard

ownership of that public key. This proof is done by presenting a digital certificate. We talk more about digital certificates in the Section 2.5.1. Asymmetric encryption uses asymmetric key algorithms such as Rivest–Shamir–Adleman (RSA) [28], Digital Signature Algorithm (DSA) [30], elliptic curve algorithms such as Elliptic Curve Digital Signature Algorithm (ECDSA) [23], and others. In asymmetric encryption generated keys are much bigger than symmetric ones (from 1536 to 4096 bits for RSA algorithm), and an algorithm by itself is very complex. Due to the size of the key, secure operations with asymmetric keys are slower than with symmetric ones.

In reality combination of both concepts is used. Asymmetric cryptography is used to establish a secure channel which is then used to share the symmetric key for further data encryption.

2.5 Public Key Infrastructure

To unleash the full power of smart cards, we need to explain core technologies that are used in the real world with relation to smart cards. On the top we have the Public Key Infrastructure (PKI) which includes a lot of subsystems. For understanding what PKI is, we need to mention main building blocks of PKI: Digital Certificate and Certificate Authority. Besides, we emphasize that PKI should meet a few security goals [6]:

- **Accurate Registration:** The user must be unable to register an identity that he does not own.
- **Identity Retention:** The user must be unable to impersonate an identity already registered.
- **Censorship Resistance:** The user must be able to register any identity that he owns.

Traditional approaches to PKI are Certificate Authorities (CAs) and decentralized peer-to-peer networks called Webs of Trust [6].

2.5.1 Digital Certificate

A digital certificate, also known as a public key certificate or identity certificate, is a kind of electronic „passport“ for a person, organization, particular machine, or even a service. The certificate is usually represented by a file with .crt or .pem extension (e.g example.crt, exmaple.pem). A digital certificate allows users to prove their identity and the target service can check the user's identity using Public Key Infrastructure.

What makes the certificates secure? How is this security implemented? The main point of a digital certificate is that it binds the assymetric key pair to a user. User or host represents a **subject** of a certificate. Why do we trust such kind of binding? Because this binding is provided by a third party we trust - Certificate Authority (Section 2.5.2). As it was mentioned, Certificate Authority is an **issuer** of the certificate. So, if someone would use a public key from a given certificate to encrypt data, he can be sure, that only the subject of this certificate would be able to decrypt an encrypted message. The information about a **subject** and an **issuer** is stored as a distinguished name.

Obviously, a digital certificate contains more information than just a subject, public key, and issuer. X.509 is a common standard that defines the format of digital certificates. In our work, we will be working with X.509 certificates as this standard is used with smart

cards. In addition to the Distinguished Name (DN) of a subject and issuer (in ASN.1 notation), X.509 certificate contains the following fields:

- **version** – version of a certificate. Currently, three versions of X.509 certificate are defined (v1, v2, v3).
- **serialNumber** – unique serial number per certificate. It useful for auditing purposes to make sure the CA created particular certificate and not created some other
- **signature** – issuing authority's signature.
- **validity** – contains two dates: the date when the certificate becomes valid (start of certificate validity) and the data certificate is expired (certificate expiration).
- **extensions** – this field is present only in v3 of X.509 certificate. For example, here can be defined extended key usage.

X.509 standard is defined by RFC5280[4].

In practice, there are two types of digital certificates based on their signature: **self-signed** and **CA signed**.

- **self-signed** – the certificate (in our context is an X.509 certificate) is signed by the same entity for which this certificate was issued (subject and issuer fields are the same).
- **CA signed** – the certificate is signed by Certificate Authority, different from the subject for which the given certificate is issued.

2.5.2 Certificate Authority (CA)

Certificate Authority is a trusted entity that provides digital certificates to users which in turn prove the ownership of a public key. In communication between hosts (for example, user and public webserver), CA is an entity that proves the identity of a server to the user. Users can also prove their identity to the server in the same way. In other words, the digital certificate is bound to the user or host (subject of the certificate) with a public key.

To obtain the digital certificate, the user or host has to generate their asymmetric key pair (public and private key). The next step is to generate Certificate Signing Request (CSR) for the CA. This CSR contains information about the subject of a certificate and the public key of a subject. CSR is an X.509 certificate signed with a subject private key[8]. After composing the CSR, it is sent to the CA. As CSR is signed with a subject private key, CA doesn't need to know the actual value of the private key to validate the public key from the certificate. After validation of a signature and other fields in the CSR, CA signes the CSR with its private key and sends back to the user the digital certificate with a trusted signature.

2.5.3 Public Key Infrastructure (PKI)

Public Key Infrastructure (PKI) [1] is a set of roles, policies, hardware, software, and procedures needed to create, manage, distribute, use, store and revoke digital certificates and

[8]The CSR may be accompanied by other credentials or proofs of identity required by the certificate authority.

manage public-key encryption. In other words, PKI is a system that defines a way for providing secure operations using digital certificates and public keys. Minimal PKI environment requires at least a Certificate Authority. Top level overview of the PKI workflow is show on the Figure 2.1.

A PKI involves the participation of trusted third parties who verify the identity of the parties wishing to engage in secure communication through the issuing of digital certificates. A real-world analogy might involve customs and immigration. When a person arrives at an airport to board an international flight they have to pass through customs. If an arriving passenger simply verbally claims to be John Smith there is no way for the customs officer to verify his identity. It is entirely possible that he really is John Smith, but because the customs office doesn't know the person he has no way of knowing whether he is trustworthy. Instead, the customs officer relies on a trusted third party in the form of a government passport issuing office. The passport office goes through the process of confirming a person's identity before issuing a passport. The passenger then uses this passport to confirm to the customs officer that they are who they say they are. Because the person has a passport, and the customs officer trusts the passport office the person is permitted into the country. Public key infrastructures work in a very similar way. A trusted third party called a Registration Authority verifies the identity of a person or entity and instructs another body, the Certificate Authority to issue a digital certificate which also contains that entities public key. This certificate (and the public key contained therein) may subsequently be used to prove identity and enable secure transactions with other parties [32].

As we already mentioned in the Section 2.5.2, to obtain a digital certificate the user has to compose a CSR. In the PKI environment, the CSR is sent to the Registration Authority (RA). RA assists the PKI cycle by verifying that the body requesting a certificate is legitimate. Once the verification is complete, it carries out the request by allowing the request to reach the CA, who uses a certificate server to execute it.

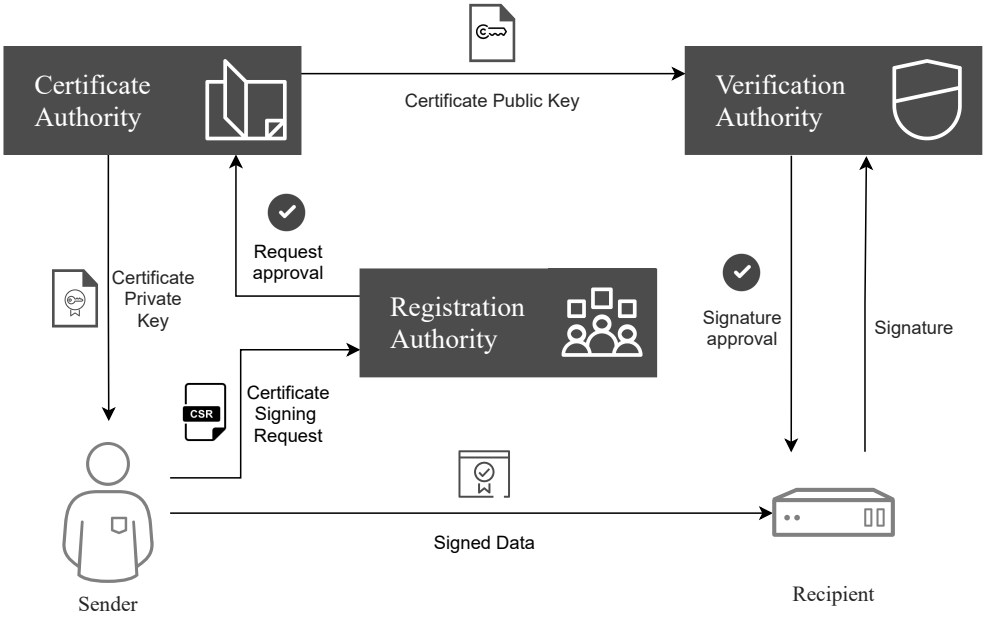

Figure 2.1: Public Key Infrastructure [3].

The public key of the user is shared with Verification Authority (VR). When someone (e.g webserver) would need to verify the certificate of the user, VR would provide such kind of verification. For user verification signature from the digital certificate would be used. This signature is made by the CA and added to the certificate. A service can take this signature from the user's certificate and send it to the VR for validation. If a signature is valid and the certificate is not in the Certificate Revocation List (CRL), a signature is approved and the service can trust the user's certificate. These operations are specified in Online Certificate Status Protocol [16].

2.6 Standard PKCS #11

PKCS #11 – is the Public-Key Cryptography standard that defines the platform-independent application programming interface (API) for hardware tokens [7]. These tokens can be Hardware Security Module (HSM) or smart cards. The API for PKCS #11 is also called Cryptoki („cryptographic token interface“ and pronounced as „crypto-key“). Cryptoki API provides common functions for cryptography and follows a simple object-based approach. The goal of Cryptoki is to provide independent API, so it can be used on any device, and to provide resource sharing (multiple applications accessing multiple devices). These two goals lead to a smart card (generally, cryptographic token) to be presented to the application in a common and logical view.

2.7 Testing of Smart Cards

Historically, smart cards were tested manually. These tests have been developed for about 10 years. Testing of smart cards was done in a way that is closest to the way how our customers are using smart cards and related software. Another reason why smart cards are tested manually during all this time is that there was no technology for automated testing that could provide smart card virtualization. This negatively influences the performance of smart card testing as well as it is prone to human errors, which in turn increase overall costs of testing.

Over the years, technologies took a big step forward and now we have virtualization technologies that provide the ability to process most test cases automatically without any or just little need of a smart card as a hardware device by itself.

Before we deep dive into the issues with smart card testing, we need to introduce the testing techniques which are used.

2.7.1 Sanity testing

This technique provides quick stability validation of the new functionality or code changes in the new build. Sanity testing determines whether the build is eligible for further rounds of testing or not. This decision is made based on whether essential functionality is working or not. If critical parts of a new build are working correctly, this particular build can be transferred to the next stage of testing.

2.7.2 Integration testing

Following the article „*Integration testing techniques*“ [21], integration testing is a logical extension of unit testing. In its simplest form, two units that have already been tested are

combined into a component and the interface between them is tested. A component, in this sense, refers to an integrated aggregate of more than one unit. In a realistic scenario, many units are combined into components, which are in turn aggregated into even larger parts of the program. The idea is to test combinations of pieces and eventually expand the process to test your modules with those of other groups. Eventually all the modules making up a process are tested together. Beyond that, if the program is composed of more than one process, they should be tested in pairs rather than all at once.

Integration testing identifies problems that occur when units are combined. By using a test plan that requires you to test each unit and ensure the viability of each before combining units, you know that any errors discovered when combining units are likely related to the interface between units. This method reduces the number of possibilities to a far simpler level of analysis.

2.7.3 Regression testing

After a new build is sanity tested and marked as ready for the next step of testing, regression tests cames to play. The regression testing technique aims to check if bug fixes are done and the code changes did not break other parts of the software. Also, regression testing checks not only internal functionality, but also other affected areas. So, regression testing is a part of integration testing. All these test scenarios have to be executed to verify bug fixes and code changes. Because the support of smart card is required in Red Hat Enterprise Linux (RHEL) distribution[9], this support should be fully tested with every new release of RHEL.

2.7.4 Black-Box testing & White-Box testing

There are a lot of vendors for smart cards and each vendor chooses his own way of developing the smart cards. Some of the implementations are open-sourced, others are proprietary. But customers don't care so much about the implementation and license for the source code of the smart card, they just use the card. At this point, we come to the following two types of testing: **black box testing** and **white box testing**. Schematically the idea behind these two types is shown on the Figure 2.2.

White box testing is a software testing technique in which internal structure, design, and coding of software are tested to verify the flow of input-output and to improve design, usability, and security. In white box testing, code is visible to testers so it is also called Clear box testing, Open box testing, Transparent box testing, Code-based testing, and Glass box testing [9]. This technique applies to open-source smart cards, so we can test each aspect we are interested in.

Black box testing [22, p.55] (sometimes referred to an functional or behavioral testing) is a testing technique in which the internal structure of the system or the code is not known. Tests in this technique are made based on the specifications or requirements of the tested system. Smart cards have an open-source implementation of applets like CoolKey[10] cards (implemented in Java) with drivers implemented in OpenSC [19] and we also have commercial implementations. Both of the implementations have to be supported. So, with the commercial implementation, we don't have another choice for testing technique than black box testing. The scope of tests that can be applied to the proprietary implementation

[9] https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux
[10] https://www.dogtagpki.org/wiki/CoolKey

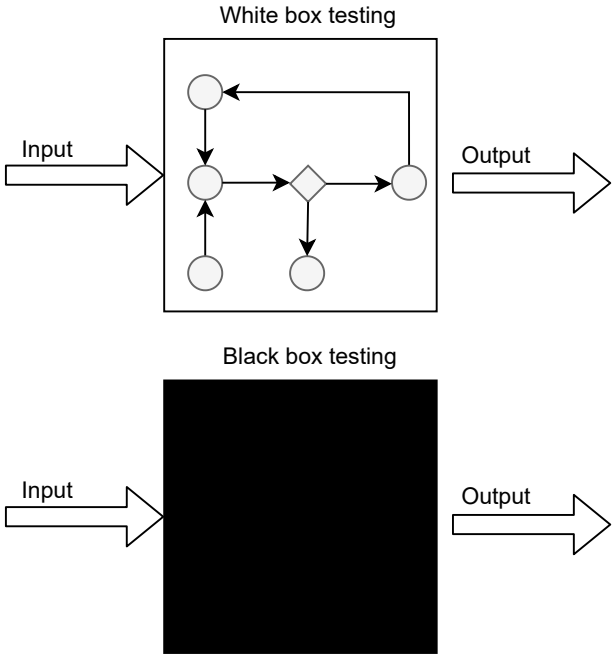

Figure 2.2: Difference between white box and black box testing.

of the smart card is limited because we don't have access to the source code that runs on the card, so we can only run tests based on technical specifications.

Chapter 3

Related Work

This chapter introduces the principle of how system authentication with smart cards works. Enterprise authentication with smart cards requires other services such as Kerberos, LDAP, and Certificate System server. For the configuration of these services, System Security Services Daemon[1] (sssd) is used[2]. This chapter also covers technologies that are used for smart card testing, such as smart card virtualization (Section 3.2.1), BeakerLib (Section 3.2.3) Smart Card Removinator (Section 3.2.2).

3.1 Certificate Management System server

As it was mentioned in previous sections, smart cards can store certificates. These certificates should be obtained from somewhere. For this purpose, there is a Certificate Authority (described in the Section 2.5.2) that issues and manages certificates. Manipulations with certificates are held on the Certificate Management System server (CMS) in Public Key Infrastructure (PKI). Because the CMS server needs to validate the user's personal information, an LDAP server is used. After validation of the necessary information, the CMS server generates certificates that are used for Kerberos authentication. Responsibilities of the CMS server are show on the Figure 3.1.

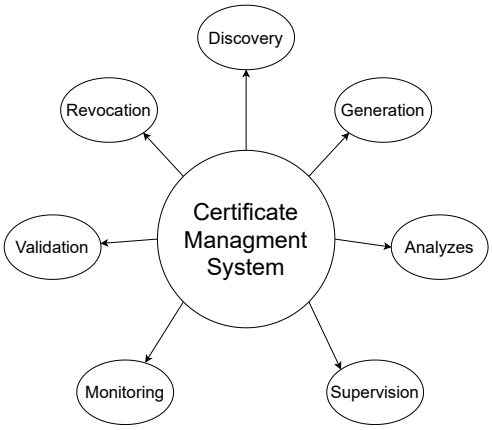

Figure 3.1: Certificate Management System responsibilities.

[1] https://sssd.io/

[2] sssd is used from RHEL 8 release. Before RHEL 8 pam_pkcs11 was used for login configuration

3.2 Testing technologies

As it was mentioned, the support of smart cards is tested manually at this moment. These tests include over 200 test cases for different scenarios that have to be executed for each new release. The problem with automated testing is that standard readers can't be used for automation due to their architecture – they have physical contact that detects the smart card insertion or removal. Moreover, some test cases can be destructive for the real cards, and if one of those tests fails, then the card could be blocked. From this side virtualization technologies are essential for the automated testing of smart cards.

3.2.1 Smart Card virtualization

Virtualization of smart cards can be implemented using *SoftHSM*[3]. This is part of the OpenDNSSEC project[4]. SoftHSM is a software implementation of the real Hardware Security Module (HSM) that can be used for cryptographic operations. SoftHSM implements a store that can be accessed with the PKCS#11 interface. But as the real HSM is an expensive hardware for small businesses, the software representation provides wide enough functionality for handling necessary cryptographic operations. We will use this module to emulate smart cards with stored certificates in our project.

Unfortunately, SoftHSM as it is can't be used directly for simulation of the smart card behavior because the SoftHSM token can't be *removed* or *inserted*, it just presents static token in the system. Moreover, using SoftHSM token as it is would abstract too many layers of the smart card testing, such as OpenSC and pcsc-lite layers. To overcome this limitation of SoftHSM **virt_cacard** [20] library is used for smart card virtualization. The architecture of the virt_cacard library is shown on the Figure 3.2 The virt_cacard acts like a glue between SoftHSM and plugin for pcscd daemon. The virt_cacard is using libcacard [15] library, that takes the SoftHSM token and wraps it with functionality that provides abstraction from SoftHSM token and makes it similar to the simple smart card. In the end, the SoftHSM token acts as a simple real card with corresponding functionality: it can be inserted and removed from the virtual smart card reader visible in the system.. To enroll in such kind of a virtual smart card, the user would need to upload a certificate

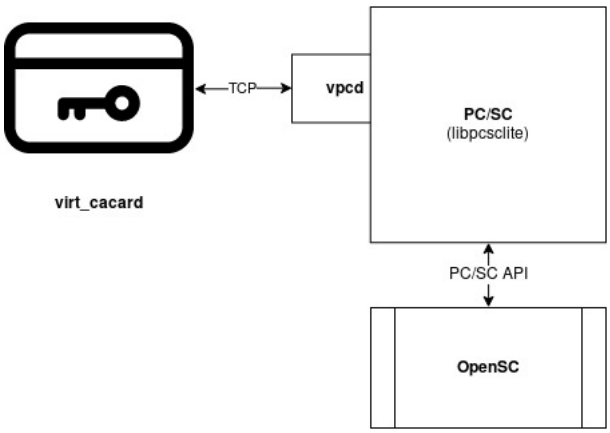

Figure 3.2: virt_cacard scheme [20].

[3] https://www.opendnssec.org/softhsm/
[4] https://www.opendnssec.org/

on the SoftHSM token and then use virt_cacard. VPCD plugin creates virtual slots so virtual cards can be plugged in and accessed through a Personal Computer/Smart Card (PC/SC) standard[5]. As soon virtual smart card is recognized on the PC/SC level, the user can manipulate with a virtual smart card same as with a real card through the OpenSC middlware.

As virtualization technologies are just a simulation of the smart card, they have some limitations as any simulator might have. One of such limitations that virtualization has is that there is no well-known way to connect the virtual smart card with the CMS server so we can dynamically request and upload new certificates to the virtual smart card. It is one of the challenges for the future testing library – how to connect these two technologies (virtual smart card and CMS) so they can cooperate automatically. Another limitation of a virtual smart card is that such kind of a smart card is very generic and can't simulate closed source smart cards. At this point, we have to return to the physical cards, but we still need to automate the testing of those cards. This problem was partially resolved with a special hardware card reader – the so-called Smart Card Removinator.

3.2.2 Smart Card Removinator

The smart card Removinator [13] is a card reader that can switch up to eight smart cards. Switching is controlled by RS-323 interface [31]. The smart card Removinator provides functionality such as simulating the removal of the smart card to trigger the corresponding signal for example for locking the PC screen. Additionally, the important thing for the test automation is that on each of eight smart cards different applets and certificates can be used. Smart cards that can't be taken out of some country can be accessed remotely with Removinator. This functionality brings more flexibility in testing support of the smart cards because those cards can be used for a variety of use cases, and each of them may require a different driver in OpenSC. You can see how smart card removinator looks like on the Figure 3.3.

With virtualization technologies and smart card removinator the risk of system damage comes into play. This problem can be solved by executing tests on the virtual machine as well as with carefully backing up and restoring configuration and system files needed for the particular test.

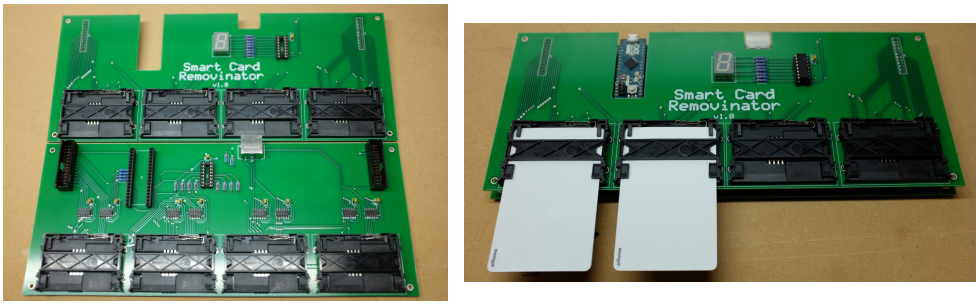

Figure 3.3: Smart Card Removinator [13].

[5] https://pcscworkgroup.com/

3.2.3 BeakerLib.

BeakerLib [24] is a shell-level integration testing library, providing convenience functions that simplify writing, running, and analysis of integration and black-box tests. Beakerlib wraps around a lot of system utilities and other tools that are used for testing and it makes it really easy to write good, easily readable tests with good structure. Other than Beakerlib we are also exploring the Avocado framework. [27]. We will decide which library fits better our needs during the implementation process for integrating tests into the existing QA pipeline.

3.3 Red Hat Certificate System (RHCS)

Red Hat Certificate System (RHCS) is a certificate management system from Red Hat. RHCS is shipped as a web server Apache Tomcat [2]. The core of RHCS is Dogtag [5] Certificate Management System (Section 3.1) which provides complete certificate lifecycle management and management of PKI (Section 2.5.3).

For our test library, we need to communicate with the RHCS server to use the PKI environment inside it. There are two main ways to choose. The first option is to use its **PKI REST API**[6] directly. PKI provides REST [34] interface to allow clients to access services on the server. The REST interface uses regular HTTP request methods:

- **GET**: Fetch data, no side effects
- **POST**: Create new entries in the namespace
- **PUT**: Update entires in the namespace.

In general, the POST request will not create active entries but will require a further PUT request to approve. One exception is when the user creates and approves certificates in one call. If we continue this approach, we will have to revise the security mechanisms around it, as currently it requires disabling nonces. All HTTP calls should have return codes defined for expected success and error cases. The Dogtag has bindings on Tomcat URL.

The second option is **PKI Client CLI**[7] calls provided by the PKI client package. CLI calls can provide the same functionality as using REST API, but with more user-friendly interface. With CLI the user can use simple command line calls to create a CSR and to obtain the certificate. All other work would be handled on the background.

The third options is to use **PKI Client Python API**[8]. This variant seems to be the most suitable from the implementation point of view, because PKI Client Python API has direct bindings on the REST interface. Also, with PKI Client Python API we wouldn't need to wrap PKI Client CLI with python code or to integrate poor REST interface into our library.

3.4 Kerberos

When using smart cards it is essential to provide a secure channel for the user's data (credentials, certificates). When the user logs into the system, sensitive data is sent via

[6] https://github.com/dogtagpki/pki/wiki/PKI-REST-API
[7] https://github.com/dogtagpki/pki/wiki/PKI-Client-CLI
[8] https://github.com/dogtagpki/pki/wiki/PKI-Client-Python-API

some kind of network to the server where users' credentials can be checked. In this client - server communication, the network is the „Achilles heel“ because of network vulnerabilities.

To minimize the influence of network weaknesses, the Kerberos[9][12] protocol is used. Kerberos is a network authentication protocol that provides strong authentication for client - server applications by using symmetric cryptography. In our work Kerberos server additionally provides a Single Sign-On (SSO) mechanism[26]. This mechanism provides a universal way to use one credential for different independent services. Credentials can be a standard pair login + password or it equally can be the certificate from the smart card. The key to this universality is that the user needs to provide credentials only while obtaining a ticket that would be automatically used for signing on to supported services. And here this universal ticket is obtained from the Key Distribution Center (KDC) on the Kerberos server.

Imagine that a user wants to login into an account as a work user and for this he needs a smart card. Before getting the certificates on the smart card we need to prepare the card by itself. As the aim of the smart card is to identify the owner of the card, we need to format the card to be sure that there are no certificates that do not belong to this particular user. After formatting, we need to enroll the smart card. Enrolment of the smart card tells the CMS server that this particular smart card now belongs to that user. This user has to be an LDAP user. The enrolment of the smart card is done when there is a need to change the user or certificates on the smart card. After enrolment, we can request the necessary certificates from the CMS server. This is made under any local user and also can be made on another PC configured with the same CMS server. The user provides his credentials and the CMS server validates them on an LDAP server. If credentials are valid, then the CMS server generates requested certificates and sends them to the user. Typically, it is not only one certificate, but a few of them. Those certificates can be for a different purposes:

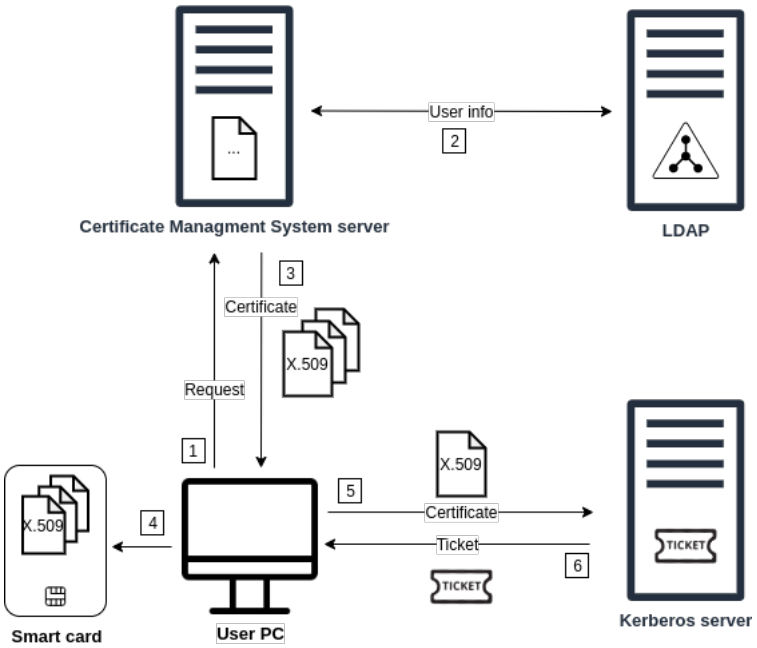

Figure 3.4: Communication between elements.

[9] https://web.mit.edu/kerberos/

- signature – provides digital signature
- authentication – used for authentication to the system/services
- encryption – used for encrypting user data
- and etc.

After receiving the certificates, they are uploaded to the smart card. If a corresponding rule for authentication exists in the system configuration (in sssd match rules [33]), then the corresponding authentication certificate would be used for authentication with the Kerberos server. If the Kerberos server accepts the certificate for this user, then the KDC issues a Ticket Granting Ticket (TGT) encrypted with a private key from Ticket Granting Service (TGS) to this particular user. The user can login with this ticket into the system and use services, that are allowed by this ticket. The whole communication schematically is shown on the Figure 3.4.

Chapter 4

Architecture

In this chapter, we will talk about the architecture and design of the library and what is the library responsible for. After that, we will discuss the current implementation in the Chapter 5. Current limitations of the library are discussed in Section 6.2.

SCAutolib is designed to be a standalone library and it does not depend on any internal Red Hat services. This makes the library available to be used by anyone who wants to test smart card functionality. The library uses a combination of Bash and Python[1] languages to provide the necessary functionality. Bash scripts are used for the deployment of the testing environment. The library is responsible for the following aspects:

- deploying the environment
  - creating local Certificate Authority (CA)
  - deploying remote Red Hat Certificate System (Section 3.3)
  - deploying Kerberos server (Section 3.4) with an LDAP instance connected
  - creating virtual smart cards
- providing API for manipulations with smart cards (virtual or real cards connected through removinator)
  - removing and inserting
  - formatting
  - uploading certificates issued from the local CA or the RHCS server on the smart card
- providing an interface for communicating with the necessary subsystems in Public Key Infrastructure (Section 2.5.3) environment
  - Certificate Authority (CA) (both on localhost and RHCS server) for issuing certificates and revoking them
  - Online Certificate Status Protocol (OCSP)

4.1 Logical diagram

Let's look at how our library fits into the test environment and how the library will manage all of the environment components. The logical diagram is shown on the Figure 4.1.

[1] Python version ≥ 3

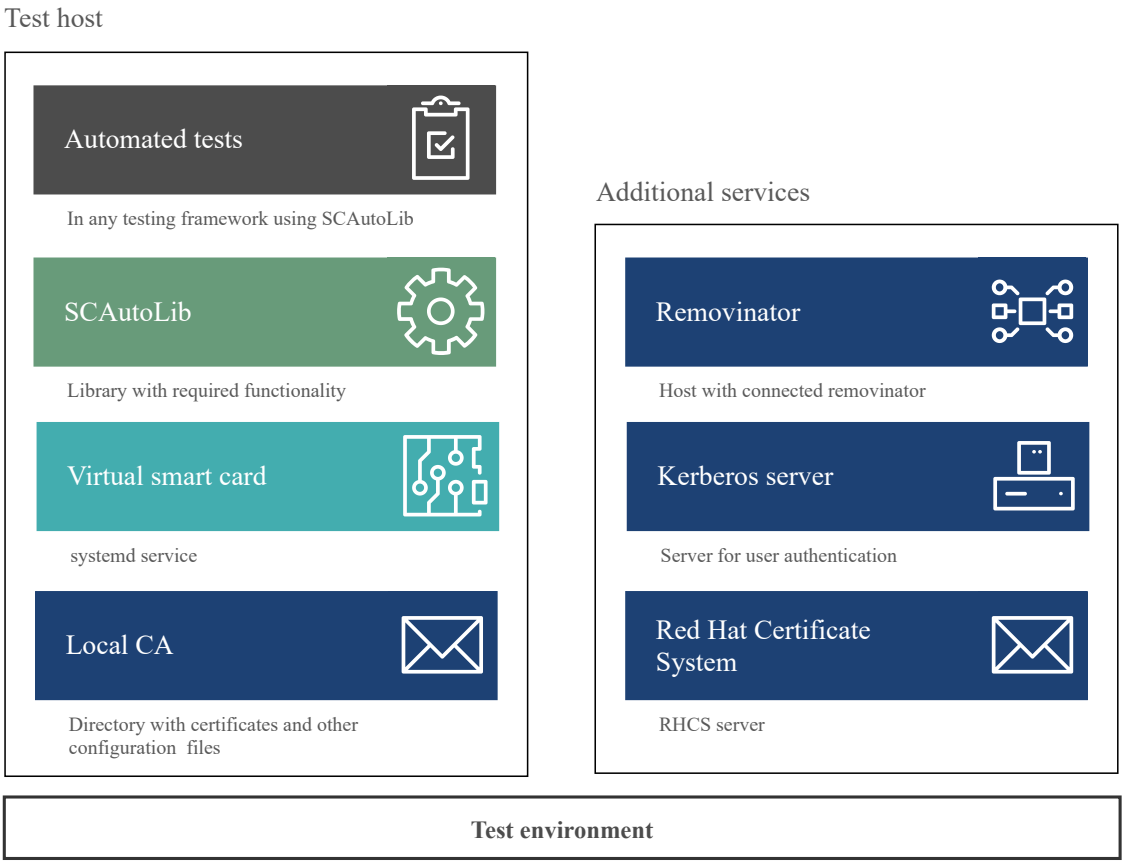

Figure 4.1: Components of a test environment.

A testing environment consists of two main parts – **tested host** and **additional services**.

Test host

Test host is a machine that we are testing. We will assume that this machine is using RHEL 8 as an operating system. This machine contains a virtual smart card which is represented as a systemd service. Starting this service simulates insertion of the card and stopping simulates removal. For testing, we would need to have two virtual cards. The first card would contain a certificate from the local CA that is created by the library before the virtual card is set and the second card would contain certificates from the RHCS server. The test host contains the automated test cases which use SCAutolib to create, configure and manage services. The test cases can be written in any framework. In our case, the tests are written in the Avocado framework.

Additional services

Additional services in our case are a set of additional technologies that can be used to more thoroughly test smart cards. It is not required to have these services on the separated machines, so for testing, purposes we can deploy them on the same host. The most important part of additional services is a Kerberos server in combination with an RHCS server. Obviously, Kerberos is an important part because it enables smart card login with non-local

users and enables Single Sign-On for such users. All information about the Kerberos users is stored in an LDAP instance. We don't do any LDAP manipulation from the library, so there is no connection point between SCAutolib and an LDAP instance. In the diagram, we assume that an LDAP instance is deployed on the same host where the Kerberos server is. Removinator is an intermediary between code and real cards. As Removinator is a physical card reader that is connected to some remote host, the real smart cards inserted into the removinator are tested on the remote host (check Section 6.2.2). The user would be able to access cards in removinator through the library.

4.2 Schematic Diagram

In this section, we will talk about how SCAutoLib communicates with each component in the test environment. The schematic diagram is presented on the Figure 4.2

First, let's look at the left side of the diagram - tested (local) host. On this side, we have a virtual card(s) and local CA along with the SCAutolib library. As we mentioned before, a virtual card is represented by a systemd service, so SCAutoLib manages the virtual cards using the system service manager (e.g `systemctl start virt_cacard.service` for inserting the card). In reality, local CA is represented by a set of directories (for generated certificates, certificate revocation list (CRL)) and files (root private key and self-signed root certificate, issued certificates). This directory structure is generated using a bash script and the certificates are generated using OpenSSL. The library con-

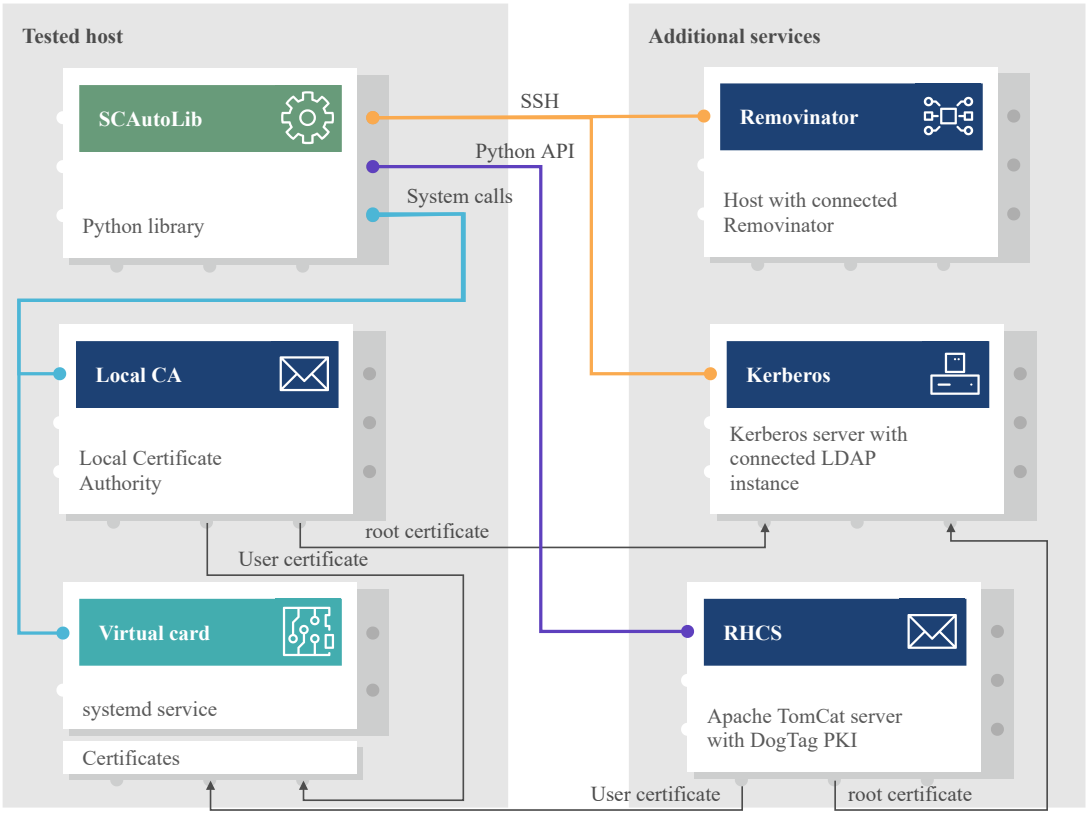

Figure 4.2: Schematic diagram of SCAutoLib library.

tains a Bash script[2] for deploying local CA on the test host. This script creates all the necessary directories and generates the certificates which we then upload on the virtual smart card using another script[3]. After all the preparation steps are done, the library can manage the local certificates using the OpenSSL command-line utilities (e.g `openssl ca -keyfile rootCA.key -in user.csr -out user.crt`)[4]

On the right side, we have additional services. Each of the hosts is deployed through the library by a Bash script. For communicating with removinator host a special library is used. But in reality, this library connects to removinator host via SSH. Removinator requires a special library because it is unique hardware that is not manufactured by any company but is handmade. So, manipulations with Removinator have to be as much secure and non-damaging as possible. This special library acts as a fuse as it provides safe wrappers to user commands.

Kerberos server is deployed with LDAP instance connected to it as we have to store user information there. For communication with the Kerberos server SCAutoLib also uses an SSH connection. There is no need for a direct communication channel between an LDAP instance and SCAutoLib because LDAP is required by the Kerberos server and not the library by itself. The library would be responsible for the deployment and configuration of an LDAP instance. Also, for enabling Kerberos login to the system, local CA and RHCS have to share their root certificates with the Kerberos server. With these certificates, the Kerberos server would be able to validate the user certificate and successfully authenticate the user.

Last but not least is the RHCS server. RHCS server is Apache TomCat web server with running instance of Dogtag PKI. The library communicates with the RHCS server via client python API that Dogtag PKI provides to the client.

As you can see, the testing environment for the smart cards contains a lot of services and hosts. But to have each service on the separated host is very unpractical from the maintainer's point of view. Moreover, the more real separated hosts are involved in the environment, the more the test environment is prone to problems that are not necessarily related to smart cards. As a result, the real problem of test failure might be shadowed by other errors. To eliminate such kind of situation, from a practical point of view it would be better to run all services on the same host. In other words, to have Kerberos server with LDAP instance and RHCS server on the same machine would be less error-prone.

[2] https://github.com/x00Pavel/SCAutoLib/blob/master/src/env/setup_ca.sh

[3] https://github.com/x00Pavel/SCAutoLib/blob/master/src/env/setup_virt_card.sh

[4] Another opportunity is to use python cryptography module (https://pypi.org/project/cryptography/) for such kind of operations.

Chapter 5

Implementation

In this chapter, we will look at how our library is implemented and how we deal with configuring and managing the main components. As we mentioned before, the main part of the library is written in Python with the addition of some Bash scripts that handle the deploying and cleanup of the test environment. We will introduce how the library handles main components, such as virtual smart cards, sssd, etc. The setup of removinator host is shown on the Figure 5.1. The whole library implementation at this moment takes around 450 lines of Python code along with 250 lines of code in Bash scripts that are used in the testing process at this moment.

5.1 Components

Each component in the library is represented as a class with corresponding methods. Tests have a common structure such as

Figure 5.1: Smart card removinator connected to the host.

1. system configuration
2. card manipulations
3. command execution
4. restore card state
5. restore system state

These steps can be implemented with `try: ... except: ...` block, but it would be a frequent repetition of the code. Python contains the solution for such kinds of tasks. The scenario of a kind **setup** ⇒ **action** ⇒ **cleanup** is implemented with a context manager (`with` block). To use the object in the context manager, the class of the object has to implement the magic methods `__enter__` and `__exit__`. Method `__enter__` is triggered on entering the context, so the setup step is implemented in this method. And `__exit__` method triggered on exiting from the context. A simple example for a class handling system configuration with `authselect` can be implemented in the following way:

```
def __enter__(self):
    self._set()
    return self

def __exit__(self, ext_type, ext_value, ext_traceback):
    if ext_type is not None:
        log.error("Exception in authselect context")
        log.error(f"Exception type: {ext_type}")
        log.error(f"Exception value: {ext_value}")
        log.error(f"Exception traceback: {ext_traceback}")
    self._reset()
```

As you can see, in `__enter__` method there is a call of a class method that configures authselect. In this method, authslect is configured to use sssd profile with a smart card. Options for the smart card such as enforcing the smart card for authentication, lock the screen on smart card removal, etc., are passed to the constructor and then are used in the `_set()` method.

In `__exit__` method we check if any exception was raised during the context usage. If any kind of exception would be raised, the corresponding parameters would be logged. In any case, at the end the system is restored to its original state. In the end, a simple test cases that checks su login with a smart card that uses local certificates can look like this:

```
def test_su_login(self):
    with Authselect(required=True, lock_on_removal=True, mk_homedir=False):
        with VirtCard(insert=True) as sc:
            sc.run_cmd(
                cmd='su - localuser -c "su - localuser -c whoami"',
                pin=True,
                passwd='123456'
                expect='localuser')
```

5.2 Test Utilities

Some test cases require specific configuration of some services, generation of additional temporary files or certificates, and other common operations with the system. Such kind

of functionality is common for the virtual card, physical card, and also can be used in Authselect class. Auxiliary functionality is implemented in the separated file `utils.py`. There are the functions for editing specific configuration files of services, that correspond to the smart cards (e.g sssd), for file backup, service restart, etc. Some of the functions are implemented as decorators that users can use to wrap the test case.

It is worth paying attention to how we edit those configuration files. To enable such kind of functionality, unique placeholders in form of comments were added to the default configuration files that are copied to the corresponding location in the system during the setup phase. Usage of such decorator is shown below

```
@utils.edit_config(service="sssd", string="some string", holder="pam", section=True)
det test(self):
    ...
```

As you can see, the first parameter is the name of the `service` for which the user wants to change the configuration file. Then we have a `string` that is the actual line we want to add to the configuration file. `holder` can be used to add the line in a specific section or it can be a substring in the file content that would be changed. The parameter `section` specifies the meaning of the holder, whether it a section or just a substring. Before editing any file, we do a backup of those files and after the test execution, we restore the original files so we don't cause any problems to the next test case that will be executed. After every update of the configuration file, the corresponding service is restarted.

5.3 Evaluation

Now we have the library with minimal functionality and we have written tests using SCAutolib. In this section, we will discuss how test cases are executed and how we can analyze the output of those tests.

We assume that we already have up and running host with RHEL distribution on it. Smart Card Support module also has to be installed on the system. In the directory `/root` we create the following directory structure:

```
/root/
├── tests/
│   ├── SCAutolib/ ..........................................Library source code
│   ├── test_certs.py
│   ├── test_login.py
│   └── test_sssd_conf.py
└── CA/
    └── conf/ ..............................Directory with default configuration files
```

Before actual test execution, we need to setup the target host. As we already mentioned, setup phase contains two steps: configuring the local CA and deploying the virtual smart card. This two steps are done with following commands:

```
python3 SCAutolib/src/env.py setup-ca --work-dir /root/CA/ --conf-dir /root/CA/conf/
python3 SCAutolib/src/env.py setup-virt-card --work-dir /root/CA/ --conf-dir /root/CA/conf
```

As you can see, we need to specify the working directory and the configuration files directory for both the local CA and the virtual card deployment. The directory with configuration files contains default configuration files that would be changed and copied to corresponding places during the deployment process. Working directory after deployment is finished contains following file structure:

```
/root/
└── CA/
    ├── conf/ ..............................Directory with default configuration files
    ├── crl/ ..............................................Certifcate Revocation List
    ├── db/ ..................................................................NSS Database
    ├── localuser1.crt
    ├── localuser1.key
    ├── rootCA.crt
    ├── rootCA.key
    └── tokens/ ..............................................SoftHSM tokens
```

You can see that the directory contains necessary files and directories for local CA (root certificate `rootCA.crt`, root private key `rootCA.key`, user certificate `localuser1.crt` and private key `localuser1.key`, Certificate Revocation List `crl/`) along with directories for SoftHSM tokes (`tokens/`) and NSS database (`db/`).

For the execution of test cases, we would show relatively to `/root/tests/` directory (test directory). As we mentioned above, tests are written in Avocado framework. In the test director, we use a command

```
avocado --show base,app run test_sssd_conf.py
```

to execute tests with library logger `base` and built-in Avocado logger `app`. The output of the test execution you can see in the Appendix A.

5.4 Test Metrics

Now we can build some expectations regarding the time that would be needed to run all test cases one by one. The original test plan contains **248 test cases** for 3 main components: authselect, OpenSC, and p11-kit. For evaluation of this test plan experienced full-time QA engineer spends around 8 working days (8 days * 8 hours = **64 hours**) without maintaining the test environment. Test results in the Appendix A shows one of the worst cases we need to count with: editing the configuration files followed by service restart. So, even if we would map 1:1 manual test cases to automated ones, then overall time to run 248 test cases would take (248 tests * 17 sec)/3600 = **1.17 hour** without counting time for the setup phase. Time consumed for the execution of automated tests is **54 times** less than the time consumed by manual test evaluation!

Unfortunately, in reality, we can't map manual test cases to automated in 1:1. The reason is that some tests require specific cards that can't be inserted into removinator (YubiKey USB token) or special card readers that also can't be automated in the pipeline. Test of this kind takes around **5%** of all test cases. Another special type of test includes manipulations with the desktop application such as mail clients and browser. These test cases are not a part of integration testing, so we can skip them. Again, tests with desktop applications involved make around **5%** of all test plan. So, we have 248 - 10% = **223 test cases** that theoretically can be implemented with mentioned technologies and library improvements. This amount of test cases will take approximately (223 tests * 17 sec)/3600 = **1 hour** without setup phase and special test cases with real hardware, which is **64 times** faster, than manual testing.

This calculations are made based on current state of the implemented test cases and the library. In future we would be able to implement more complex test cases that might

took more time for execution. But it still would be significantly faster than executing test cases manually.

Chapter 6

Discussion

In this section, we will discuss the limitations of our work as well as possible extensions and improvements followed by limitations of the library.

6.1 Future Improvements

Currently, the library is not fully equipped with all the necessary functionality. There are still a lot of things that needed to be added. In this section, we would talk about possible improvements that can be added to the current implementation of the library. Also, we would mention a possible way for the implementation of these features. Upgrading library functionality is a plan for future development.

6.1.1 Improvements

The first thing that we can improve is to get rid of the Bash scripts for deploying the environment. Bash scripts can be replaced with a more modern solution for the deployment tasks – Ansible [25]. The advantage of this solution is that Ansible is very flexible technology in sense of configuration and writing the specifications in the YAML format. For example, Ansible provides distributive-independent way to install required packages to the system[1]. Moreover, Ansible is also implemented in Python language, so it is a very easy to integrate Ansible calls into the Python code.

At this moment the library is very hardcoded in a sense of system configuration during environment deployment. That means that the user (aka tester) can't specify for example passwords and PIN that he would like to have in the system user and for the smart card. Or the configuration files are also stored in the library and the user can't specify his files and the only way how those files can be edited by the user is through the decorators as shown in the example in Section 5.2. To solve this problem, those parameters such as root password, user password, the path to configuration files, etc. can be specified in the separated file with the given format. Before environment deployment, the user would create such kind of a file and give it as a parameter for the corresponding scripts. This file can be in JSON or YAML format, as they are easily serializable. For example, such kind of a file in YAML format might look in the following way

```
variables:
    root: rootpassword
```

[1] https://docs.ansible.com/ansible/latest/collections/ansible/builtin/package_module.html

```
    users:
        - name: localuser
          passwd: userpassword
        - name: kerberos-user
          passed: kerberospassword
    smartcards:
        - name: localuser
          pin: Secret.123
          service_name: virt_card_localuser.service
        - name: kerberos-user
          pin: RealCard.321
configs:
    files:
      - dir: /home/test/other_configs/
      - sssd: /home/test/config/sssd.conf
      - kerberos: /home/test/config/krb.conf
    local_ca_path: /root/ca/
```

In the `variables` section, the user specifies root password, credentials for two users on the test host, information for smart card instances that are needed for correct usage and deployment. If given credentials are used for virtual smart cards, there is a service name for the corresponding virtual smart card. Also, this service name would be used during the deployment on the setup phase. If there is no service name, the given credential is used for the real card in removinator. In a section `configs`, the user specifies the directory where all default configuration files are located. The user can override the location of configuration files for some service if it would be needed. Another useful parameter that the user would be able to use with such kind of parameterization is the configuration of local CA deployed before test execution. For example, the user can define where the folder that is required for creating local CA would be created.

Currently, the library requires the system to have authselect for setting up the authentication method and configuring sssd. One of the possible improvements can be adding other tools for configuring authentication methods, such as authconfig. As authselect is a replacement for authconfig, but adding this support will allow users to use the library even on older operating systems, such as RHEL 7.

The way of distributing our library can be also improved. Currently, the SCAutolib library can be downloaded from the public GitHub repository [35]. The problem with such kind of distribution is that the source code of the library has to be placed near test files (or has to be placed into the system directory manually). Also, the user would need to install all required package by themselves. A possible solution for this problem is to compose a Python package based on best practices for packaging and then to upload this package to the public Python Package Index[2] (PyPI). This way of distribution would require creating `setup.cfg` file where package specification and metadata would be placed. Required packages can be included in the package metadata and would be installed during the library installation process as library dependencies.

6.1.2 New functionality

One of the required features that are not implemented yet in the library is communication with removinator host. Ability to communicate with removinator and use it is essential for smart card testing. The reason for this is a big variety of smart cards that have to be tested at least on the detection level. And the virtual smart card can't simulate all of

[2] https://pypi.org/

those cards, because some of them are read-only and customer-provided. After fixing minor problems in the existing code, we will focus on this functionality of the library. Removintor has a python module that provides API for manipulations with the smart card inserted in it. This module needs more research to be integrated into our library.

Another important feature of the library that is not implemented yet is the ability to deploy the RHCS server and communicate with the PKI environment in it. Here communication means issuing the certificates from the PKI and managing the PKI environment through the library. As it was mentioned, communication with the RHCS server can be implemented through client CLI calls. But as with removinator python module, the CLI API needs more research before integrating it into the SCAutolib.

To provide more flexibility to the user regarding the certificates on the virtual smart card, we need not only to issue certificates through our library but also have the ability to upload issued certificates on the smart card. Uploading certificates on the smart card in technical terminology is called smart card enrollment. Enrollment of the virtual smart card is processed through the SoftHSM token (Section 3.2.1).

After this functionality is added to the library, corresponding fields can be added to the configuration file. Credentials from the RHCS or LDAP root user can be stored under `root` section for example in the following way:

```
variables:
    root:
        - system: sysRootPasswd
        - rhcs: rhcsRootPasswd
        - kerberos:
            name: krbRoot
            passwd: krbRootPasswd
```

Storing credentials in this way in the production environment would cause vulnerabilities, but in our case, these are the credentials for hosts in the internal testing environment. Hosts in the testing environment don't contain any real sensitive data. Moreover, all hosts that are used during the testing would be destroyed after test execution.

There are a lot of test scenarios where the user might want to test smart cards with Graphical User Interface. For example, testing that GNOME Desktop Manager (GDM) is recognizing the inserted smart card and prompts to insert the smart card pin require such kind of GUI testing framework. This requirement leads to integrating the GUI testing framework into our library. One of the possibilities is to use OpenQA [18]. This framework is used in automated GUI testing in Fedora release validation testing process, and testing updates[3]. With using OpenQA we could be able to automate most of the test cases that require graphical interface.

6.2 Limitations

Even with possible improvements and must-have new functionality, SCAutolib might have limitations regarding the technologies the library is using.

6.2.1 Virtual smart cards

At this moment our solution has some specific limitations regarding the virtual smart cards.

[3] https://fedoraproject.org/wiki/OpenQA

- virt_cacard can't have two instances running on the same machine at the same time. The codebase of virt_cacard is not designed for such use-case at this moment[4].
- The user typically has two and more certificates on the card (at least a signing certificate and encryption certificate), but a virtual smart card is not correctly working with more than one certificate on it.

The support of two running services with virt_cacard and correct handling of two and more certificates on one virtual card depends on the library developers.

6.2.2 Removinator

Another bottleneck of our library is removinator. Smart card testing requires real hardware (real cards and card reader) for testing and verifying some specific cards, functionality, and bugs. And the fact that execution of test with removinator is provided against the host where Removinator is connected, brings more responsibilities, such as maintaining this host, maintaining Removinator by itself, and risks such as blocking the real card under unexpected circumstances, e.g wrong PIN for the smart card has been entered more than three times due to some external bug in the system. Currently, there is no way to simulate inserting the smart card into the local machine through the Removinator connected to the remote host.

[4]We already created the issue for adding this functionality https://github.com/Jakuje/virt_cacard/issues/1

Chapter 7

Conclusion

The main purpose of this work was to design and try to implement a new tool for automated testing of the smart cards. The smart card by itself is a plastic card with an integrated circuit that can store and/or process a small amount of data. In our case, the data is digital X.509 certificates. This tool is needed because there is no other solution that provides all the functionality that we need for testing. Due to the lack of clear similar technologies, this project is developed from scratch.

Our tool is a Python library with functionality for low-level operations from the library perspective, such as inserting a smart card or restarting the required system service. With this low-level functionality, more high-level features, such as editing the configuration files or checking su login, can be implemented. Most of the testing can be implemented with virtual smart cards instead of real smart cards. Virtualization of the smart cards is implemented using *libcacard* library that wraps SoftHSM token with similar functionality and properties as Common Access Card has. For testing specific cards, that can't be simulated or for testing smart card functionality that virtual smart card can simulate at this moment, a special real hardware reader (Smart Card Removinator) is used. Smart Card Removinator has its library to operate with it and this library could be used in our project. The library is also responsible for the deployment required testing environment. Deployment scripts are implemented in Bash. The testing environment contains a virtual smart card, local Certificate Authority (CA), Kerberos server, and Red Hat Certificate System (RHCS) server. Some test cases require GUI interaction(such as GDM login). Currently the library doesn't support this functionality, but we plan to integrate the OpenQA framework into our library and that would make writing such test cases possible. The library is designed to be self-contained and publicly available.

Currently, the library contains implemented scripts for the deployment of the virtual card and local CA. For manipulating the virtual cards and system service, the library already has implemented basic low-level functionality alone with higher functions such as file backup or editing the configuration files. Future work would include adding support for Removinator and integrating OpenQA framework.

Bibliography

[1] Adams, C. and Lloyd, S. Understanding PKI: Concepts, standards, and deployment considerations. 2nd ed. Boston, MA: Addison-Wesley Educational, 2002.

[2] Apache Tomcat Project. Apache Tomcat [online]. [cit. 2020-12-22]. Available at: http://tomcat.apache.org/.

[3] appviewx. What is Public Key Infrastructure (PKI)? [online]. August 2019 [cit. 202-4-27]. Available at: https://www.appviewx.com/education-center/pki/.

[4] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R. et al. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile [Internet Requests for Comments]. RFC 5280. RFC Editor, May 2008. Available at: http://www.rfc-editor.org/rfc/rfc5280.txt.

[5] Dogtag Certificate System Team. Dogtag [online]. [cit. 2020-12-22]. Available at: https://www.dogtagpki.org/wiki/PKI_Main_Page.

[6] Fromknecht, C., Velicanu, D. and Yakoubov, S. A Decentralized Public Key Infrastructure with Identity Retention. IACR Cryptology ePrint Archive. 2014, vol. 2014, p. 803.

[7] Gleeson, S. and Zimman, C. PKCS #11 Cryptographic Token Interface Base Specificatio Version 2.40 [online], 14. april 2015 [cit. 2020-12-15]. Available at: http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html.

[8] GlobalPlatform Technology, Inc.. Card Specification [online]. globalplatform.org. Available at: https://globalplatform.org/wp-content/uploads/2018/05/GPC_CardSpecification_v2.3.1_PublicRelease_CC.pdf.

[9] guru99. What is WHITE Box Testing? Techniques, Example & Types [online]. [cit. 2020-12-03]. Available at: https://www.guru99.com/white-box-testing.html.

[10] Hogffman, N. A SIMPLIFIED IDEA ALGORITHM [online]. Available at: https://www.nku.edu/~christensen/simplified%20IDEA%20algorithm.pdf.

[11] IDManagement. Personal Identity Verification Guide Introduction [online]. [cit. 2020-11-03]. Available at: https://playbooks.idmanagement.gov/piv/.

[12] Itoi, N. and Honeyman, P. Smartcard Integration with Kerberos V5. In: Smartcard. 1999.

[13] Kinder, N. Smart Card Removinator. October 2016 [cit. 2020-12-20]. Available at: https://github.com/nkinder/smart-card-removinator.

[14] Kommerling, O. and Kuhn, M. G. Design Principles for Tamper-Resistant Smartcard Processors. Available at: https://www.cl.cam.ac.uk/~mgk25/sc99-tamper.pdf.

[15] Levy, A., Relyea, R. and Jelen, J. Libcacard. [cit. 2020-12-20]. Available at: https://gitlab.freedesktop.org/spice/libcacard.

[16] Myers, M., Ankney, R., Malpani, A., Galperin, S. and Adams, C. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP [Internet Requests for Comments]. RFC 2560. RFC Editor, June 1999. Available at: http://www.rfc-editor.org/rfc/rfc2560.txt.

[17] National Institute of Standards and Technology. Specification for the ADVANCED ENCRYPTION STANDARD (AES) [online]. U.S. Department of Commerce, november 2001. Available at: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[18] openQA Team. Automated tests for operating systems. [cit. 2020-4-24]. Available at: http://open.qa/.

[19] OpenSC Community.

[20] Palant, P.-L. Virt_cacard. March 2019 [cit. 2020-12-12]. Available at: https://github.com/Jakuje/virt_cacard/.

[21] Parmar, K. Integration testing techniques. SAP AG. Available at: https://archive.sap.com/kmuuid2/c03b0790-6d3b-3210-dcb4-848320a3d9e4/Integration%20Testing%20Techniques.pdf.

[22] Patton, R. Software Testing. 2nd ed. Indianapolis, IN: Sams Publishing, 2005.

[23] Pornin, T. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA) [Internet Requests for Comments]. RFC 6979. RFC Editor, August 2013. Available at: http://www.rfc-editor.org/rfc/rfc6979.txt.

[24] Pospíšil, D. and Šplícha, P. BeakerLib. March 2017 [cit. 2020-12-21]. Available at: https://github.com/beakerlib/beakerlib.

[25] Red Hat, Inc.. Ansible Documentation [online]. [cit. 2021-04-14]. Available at: https://docs.ansible.com/ansible/latest/index.html.

[26] Red Hat, Inc.. Red Hat Enterprise Linux 6 Managing Single Sign-On and Smart Cards [online]. April 2019. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/pdf/managing_smart_cards/Red_Hat_Enterprise_Linux-6-Managing_Smart_Cards-en-US.pdf.

[27] Red Hat, Inc. and Avocado Community Contributors. Avocado Framework [online]. [cit. 2020-12-21]. Available at: https://avocado-framework.github.io/.

[28] Rivest, R., Shamir, A. and Adleman, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems [online]. Available at: https://people.csail.mit.edu/rivest/Rsapaper.pdf.

[29] Smart Card Alliance. Smart Card Technology and the FIDO Protocols [online]. Princeton Junction, NJ: securetechalliance.org [cit. 2020-11-04]. Available at: https://www.securetechalliance.org/wp-content/uploads/FIDO-and-Smart-Card-Technology-FINAL-April-2016.pdf.

[30] Standards, N. I. of and Technology. Digital Signature Standard (DSS). U.S. Department of Commerce, june 2013.

[31] Strangio, C. E. RS232 standard [online]. [cit. 2020-12-21]. Available at: https://www.camiresearch.com/Data_Com_Basics/RS232_standard.html.

[32] Techotopia. An Overview of Public Key Infrastructures (PKI) [online]. [cit. 2021-01-18]. Available at: https://www.techotopia.com/index.php/An_Overview_of_Public_Key_Infrastructures_(PKI).

[33] The SSSD upstream. Sss-certmap: SSSD certificate matching and mapping rules - Linux man pages (5) [online]. [cit. 2020-11-27]. Available at: https://www.systutorials.com/docs/linux/man/5-sss-certmap/.

[34] Thomas, F. R. Architectural Styles and the Design of Network-based Software Architectures. Irvine, California, 2000. Doctoral dissertation. University of California.

[35] Yadlouski, P. SCAutolib - Library for automation of smart card testing. April 2021 [cit. 2021-04-23]. Available at: https://github.com/x00Pavel/SCAutolib.

Appendix A

Test Output

```
[root@localhost tests]# avocado  --show base,app run test_sssd_conf.py
JOB ID     : fcb9fbd4bb150e4d1ca6eb9593b4832d576bde07
JOB LOG    : /root/avocado/job-results/job-2021-04-29T05.57-fcb9fbd/job.log
 (1/4) test_sssd_conf.py:TestSssdConf.test_su_login_p11_uri_slot_description: base: File from /etc/sssd/sssd.conf is copied to /root/sc/Sanity/basics/SCAutolib/src/tmp/backup/sssd.conf
base: Section pam in config file /etc/sssd/sssd.conf is updated
-base: Service sssd is restarted
base: SSSD is set to: authselect select sssd --backup tmp.backup with-smartcard --force
base: Backupfile: tmp.backup
base: Smart card initialized
|base: Smart card is inserted
/base: Smart card removed
base: Authselect backup file is restored
base: File from /root/sc/Sanity/basics/SCAutolib/src/tmp/backup/sssd.conf is restored to /etc/sssd/sssd.conf
-base: Service sssd is restarted
PASS (17.72 s)
 (2/4) test_sssd_conf.py:TestSssdConf.test_su_login_p11_uri_wrong_slot_description:  base: File from /etc/sssd/sssd.conf is copied to /root/sc/Sanity/basics/SCAutolib/src/tmp/backup/sssd.conf
base: Section pam in config file /etc/sssd/sssd.conf is updated
|base: Service sssd is restarted
base: SSSD is set to: authselect select sssd --backup tmp.backup with-smartcard --force
base: Backupfile: tmp.backup
base: Smart card initialized
-base: Smart card is inserted
-base: Smart card removed
base: Authselect backup file is restored
base: File from /root/sc/Sanity/basics/SCAutolib/src/tmp/backup/sssd.conf is restored to /etc/sssd/sssd.conf
\base: Service sssd is restarted
PASS (17.08 s)
 (3/4) test_sssd_conf.py:TestSssdConf.test_user_mismatch:  base: File from /etc/sssd/sssd.conf is copied to /root/sc/Sanity/basics/SCAutolib/src/tmp/backup/sssd.conf
base: Substring localuser1 in config file /etc/sssd/sssd.conf is updated
/base: Service sssd is restarted
base: SSSD is set to: authselect select sssd --backup tmp.backup with-smartcard --force
base: Backupfile: tmp.backup
base: Smart card initialized
\base: Smart card is inserted
\base: Smart card removed
```

```
base: Authselect backup file is restored
base: File from /root/sc/Sanity/basics/SCAutolib/src/tmp/backup/sssd.conf is restored to /
    etc/sssd/sssd.conf
/base: Service sssd is restarted
PASS (17.24 s)
 (4/4) test_sssd_conf.py:TestSssdConf.test_wrong_subject_in_matchrule:  base: File from /
    etc/sssd/sssd.conf is copied to /root/sc/Sanity/basics/SCAutolib/src/tmp/backup/sssd.
    conf
base: Substring CN=localuser1 in config file /etc/sssd/sssd.conf is updated
\base: Service sssd is restarted
base: SSSD is set to: authselect select sssd --backup tmp.backup with-smartcard --force
base: Backupfile: tmp.backup
base: Smart card initialized
/base: Smart card is inserted
/base: Smart card removed
base: Authselect backup file is restored
base: File from /root/sc/Sanity/basics/SCAutolib/src/tmp/backup/sssd.conf is restored to /
    etc/sssd/sssd.conf
\base: Service sssd is restarted
PASS (17.18 s)
RESULTS    : PASS 4 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 69.88
s~
```