Masaryk University
Faculty of Informatics

An automated testing of smartcards in OpenSC project

Master's Thesis

Bc. Martin Strhársky

Brno, Spring 2016

Replace this page with a copy of the official signed thesis assignment and the copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Martin Strhársky

Advisor: RNDr. Petr Švenda, Ph.D.
Consultant: dr. Nikos Mavrogiannopoulos

Acknowledgement

I would like to thank my supervisor RNDr. Petr Švenda, Ph.D., for his guidance while writing the thesis. I would also like to thank dr. Nikos Mavrogiannopoulos, for helping me with introduction to smart cards community, valuable comments and his time invested in consultation throughout the work.

Abstract

The thesis investigates the common use of smart cards in the Fedora distribution of the operating system Linux, in order to create a unit testing application for OpenSC project. At first, the smart card usage data are collected and analysed, in order to prepare a set of tests for selected PKCS#11 drivers. Subsequently, the unit testing application for OpenSC project is developed. The objective of the application is to execute a test suite against selected PKCS#11 drivers. The selected drivers for testing are the PIV driver and the smart card driver for Cryptoflex card.

Keywords

OpenSC, unit testing, smart cards, PKCS#11, YubiKey Neo, Cryptoflex

Contents

1 Introduction

The rapid growth in the development of information technologies during the last few years resulted in the necessity to secure user's sensitive data. Many protection methods have been already developed. One of the popular options for increasing security of user's personal data nowadays is the use of smart cards.

The advantage of smart cards over other protection methods is increased security. Furthermore, their use is convenient, mainly because, they are small and compact, so a user can carry and use them at any time. The cards are used for authentication, but they can also carry sensitive information like fingerprints, face or iris images. Furthermore, the operating systems such as Microsoft Windows, Linux and Mac OS provide direct support for the smart card usage.

There are various applications, which allow the user to use smart cards in the Linux operating system. One of them is OpenSC. It provides many utility programs that are used for smart card manipulation. Moreover, the OpenSC supplies the system with smart card drivers, which enable communication with the card.

However, there still is a possibility of security issues during the implementation of smart card drivers. In order to lower the number of issues, the code should be covered by tests. For developers, the benefit of such test suite is the possibility to verify that their code did not break any functionality. Moreover, the test suite can also be used by the end user to verify that the obtained version of code is working with his/her smart card without any problems.

The goal of this master thesis is to develop a unit testing application for OpenSC project. At first, a common smart card usage in the Fedora distribution of the Linux operating system, is explored. Data are collected and analysed from several selected use cases. This analysis is further used in the creation of test suite for selected smart card drivers - PIV driver and driver for Cryptoflex card.

The theoretical part of the thesis introduces the reader to the basic concepts and definitions. The second chapter contains basic information about smart cards and smart card operating systems. Furthermore, this chapter describes the various types of smart cards.

The third chapter briefly introduces Public-Key Cryptography Standards in order for reader to get overview of the functionality, which is tested by developed application. The PKCS#11 and the PKCS#15 standards are selected and explained. The PKCS#11 standard is discussed in more detail. The description includes possible vulnerabilities, for instance various attacks through API. In addition, the Microsoft Cryptographic API is outlined as alternative to PKCS#11.

The fourth chapter deals with data collection and analysis. In order to work with smart cards, the Fedora configuration is explained thoroughly. This chapter also contains in depth data analysis obtained using the OpenSC's pkcs11-spy utility module.

The applied technologies, such as CMocka and PKCS#11 API, are described in the implementation chapter 5. This chapter also explain the created test suite. All test cases are divided into various groups according to set of PKCS#11 functions they have used.

At the end of the thesis are attached thesis appendices. The Appendix A shows the example output of the `pcsc_scan` program. The `pkcs11-tool` utility program usage with various command line arguments is described in the Appendix B. The Appendix C contains the example output of the `pkcs15-tool -D` command. This command displays all objects stored on the card and it is used to verify, which objects are created on the card after the certificate is imported using Mozilla Firefox.

The output of the practical part of the master thesis is the application written in C programming language. This application implements and provides all tests cases described in chapter 5. The application is part of the OpenSC test coverage. The output of all tests can be found in the Appendix D.

2 Smart cards

Smart card (chip card or integrated circuit card (ICC)) is a plastic card with a microprocessor and memory embedded in it. These cards are available in various sizes and different types.

There are some cards that have only non-programmable memory. These cards are read-only and the information stored on them cannot be changed or manipulated. The other group of cards containing a microprocessor, have various functionalities. Smart cards can be designed to be inserted into a slot and read by a special reader or to be read from a distance.

Smart cards are used for personal identification, authentication, data storage and application processing. Furthermore, they provide strong security authentication for single sign-on (SSO).

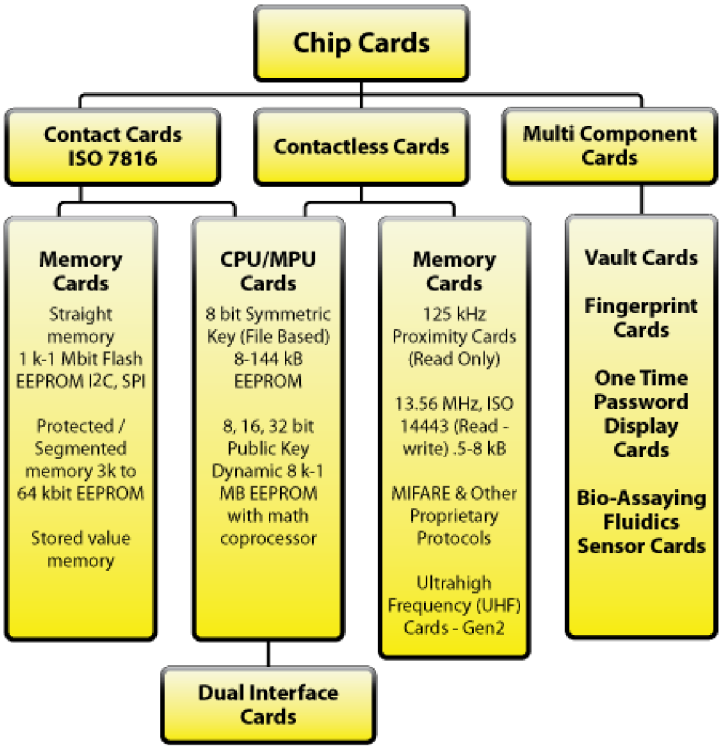
Operating systems such as Microsoft Windows or all new versions of Linux have built-in software hooks to deploy smart cards as a replacement for user names and passwords. Microsoft has built a complete credential platform around the `Scard DLL` and `Crypto Service Provider` (CSP). Business-to-business Intranets and Virtual Private Networks (VPNs) are enhanced by the use of smart cards. Based on a smart card, a user can be authenticated and authorized to have access to specific information.

2.1 Types of smart cards

There are two criteria, for smart cards classification:

- functionality of smart cards
  - microprocessor based cards
  - memory based cards
- communication with the reader (image 2.1-1)
  - contact cards
  - contactless cards

– combination cards
– hybrid cards



Img. 2.1-1. Types of smart cards [1]

2.1.1 Microprocessor based cards

Microprocessor based cards have on-card dynamic data processing capabilities. The card contains a microprocessor or micro-controller chip. This chip manages memory allocation and file access via card operating system.

Unlike other operating systems, this software controls access to on-card user memory. This capability permits different applications to reside on the card.

There are many configurations of chips in this category, for example:

- chips that support cryptographic PKI[1] functions with on-board math co-processors

- JavaCard with virtual machine hardware blocks.

2.1.2 Memory based cards

Cards in this category are used for applications in which the function of the card is fixed. They can perform only following operations: store, read and write data to a particular memory location. These data cannot be manipulated or processed.

Memory based cards contain a non-volatile memory EEPROM[2] and need a card reader to manipulate the data on the card. They communicate with the reader via synchronous protocols. Memory based smart cards have no processing power and cannot manage the data stored in them.

Due to the incapability of cryptography, this type of cards is used in storing telephone credits, electronic cash or transportation tickets.

2.1.3 Contact cards

Contact smart cards have embedded microprocessors. They contain golden plates (contact pads) in one corner of the card. Proper function of a smart card is contingent upon energy supply. Furthermore, a card needs some mechanism for communication, receiving and sending data.

The golden plates are used to supply the necessary energy and to communicate via direct electrical contact with the reader. Readers for contact smart cards are generally a separate devices plugged into serial or USB ports.

Contact cards are the most common type of smart cards. They have the size of credit cards and are used for a network security, accessing control, e-commerce and electronic cash.

---

1. Public Key Infrastructure
2. Electrically Erasable Programmable Read-Only Memory

2.1.4 Contactless cards

Another type of a smart card is a contactless card. Many of these cards are considered to be CPU/MPU microprocessor cards. In order for a contactless card to communicate with a smart card reader, the radio identification (RFID) technology is used. Furthermore, these cards have embedded antenna, which is used as an inductor for supplying of energy to the card.

Smart card readers are usually connected to the computer via USB or serial port. As the contactless cards do not need to be inserted into the reader, they are usually composed only of a serial interface for the computer and an antenna connected to the card. Readers for contactless smart cards may or may not have a slot.

A special type of contactless cards is a proximity card. Proximity cards are read-only cards and the information on these cards cannot be changed or manipulated. Such cards also use radio frequency identification (RFID) technology.

Contactless smart cards can be credit-card or token sized. They are used for electronic passports, student identification, electronic toll collection, vehicle parking and identification purposes.

2.1.5 Combination cards

These cards are combination of contact and contactless smart cards. They are read or written with or without any contact with the smart card reader. In order to manipulate the data, an antenna or contact pads are used.

Combination cards are used in mass transit, network security and for vending purposes.

2.2 Smart card operating system

The smart card's Chip Operating System (frequently referred to as COS or the Mask) is a sequence of instructions permanently embedded in the ROM[3] of the smart card. It provides basic functionality such as secure access to on-card storage, authentication and encryp-

3. Read Only Memory

tion. The operating system instructions are not dependent on any specific application, but are frequently used by most of applications [2].

Chip operating systems are divided into two families [3]:

- the general purpose COS – it has a generic command set in which the various sequences cover most of applications.
- the dedicated COS – commands are designed for specific applications and can contain the application itself.

The smart card operating system provides the baseline functions that are common across all smart card products. It is responsible for file and data management held in memory and communication between the card and the card reader. Additional responsibilities are access control to information and functions (e.g. select file, read, write and update data), management of card security and the cryptographic algorithm procedures.

Chip operating systems that support multiple applications on smart cards, are:

- JavaCard OS
- MultOS (Multi-application Operating System)

3 Public-Key Cryptography Standards

The Public-Key Cryptography Standards (PKCS) are a set of standards for public-key cryptography developed by RSA Laboratories in cooperation with an informal consortium, originally including Sun Apple, Microsoft, DEC, Lotus and MIT [4].

The PKCS are designed for binary and ASCII data and they are also compatible with the ITU-T[1] X.509 standard.

The PKCS include two types of standards: algorithm-specific and algorithm-independent. Many algorithms are supported, including RSA and Diffie-Hellman key exchange, however, only the latter two are specifically detailed [4]. The PKCS also define an algorithm-independent syntax for digital signatures, envelopes and also for extended certificates. This enables the implementation of any cryptographic algorithm to conform to a standard syntax and achieve interoperability.

The following is the list of Public-Key Cryptography Standards (PKCS) [4]:

- PKCS#1 – defines mechanisms for encrypting and signing data using a RSA public-key cryptosystem.
- PKCS#3 – defines a Diffie-Hellman key agreement protocol.
- PKCS#5 – describes a method for encrypting a string with a secret key derived from a password.
- PKCS#6 – is being phased out in favor of version 3 of X.509.
- PKCS#7 – defines a general syntax for messages that include some cryptographic enhancements such as digital signatures and encryption.
- PKCS#8 – describes a format for private key information.
- PKCS#9 – defines selected attribute types for use in the other PKCS standards.

1. Telecommunication Standardization Sector of the International Telecommunications Union

- PKCS#10 – describes syntax for certification requests.
- PKCS#11 – defines an independent programming interface, called Cryptoki, for cryptographic devices such as smart cards and PCMCIA[2] cards.
- PKCS#12 – specifies a portable format for storing or transporting a user's private keys, certificates, miscellaneous secrets, etc.
- PKCS#13 – defines mechanisms for encrypting and signing data using Elliptic Curve Cryptography.
- PKCS#14 – covers pseudo-random number generation, but it is currently in development phase.
- PKCS#15 – is a complement to PKCS#11, giving a standard for the cryptographic credentials format stored on cryptographic tokens.

3.1 PKCS#11

The PKCS#11 is a cryptographic token interface standard, that specifies an API[3], called Cryptoki. With this API, it is possible to implement and to perform cryptographic functions for cryptographic tokens.

The Cryptoki follows simple object-based approach, addressing the goals of technology independence and resource sharing [5]. Consequently, any kind of device thus can be used and multiple applications can access multiple devices. Furthermore, it presents a logical view of the device, called a cryptographic token, to the application. For instance, devices such as hardware security modules (HSM) and smart cards belong to cryptographic token category.

The API defines the most commonly used cryptographic object types (`RSA` keys, `X.509` certificates, `DES` and `Triple DES` keys, etc.) [6]. Additionally, the API specifies all necessary functions to use, create or

2. Personal Computer Memory Card International
3. Application Programming Interface

generate, modify and delete those objects. Complete list of functions defined by Cryptoki is described in chapter 5.

When the cryptographic token is connected to the computer (e.g. inserted to the reader), the appropriate slot ID defined by the Cryptoki is assigned. In order for application to use a particular cryptographic token, the appropriate slot ID has to be specified.

3.1.1 Microsoft CryptoAPI

One of the alternatives to PKCS#11 is the Microsoft Cryptographic Application Programming Interface (also known as CryptoAPI or MSCAPI). The CryptoAPI together with Microsoft Cryptographic Service Providers (CSPs) are included within the Microsoft Windows operating systems. Furthermore, they provide services enabling developers to secure Windows-based applications by means of cryptography.

The CryptoAPI supports both public-key and symmetric-key cryptography. It also contains functions that allow applications to encrypt or digitally sign data. All cryptographic operations are performed by independent modules known as cryptographic service providers (CSPs).

The Cryptographic Service Provider provides an implementation of a Cryptography API layer. Some provide stronger cryptographic algorithms, while others contain hardware components such as smart cards [7]. CPS is built and delivered as a `.dll` library on Windows. There is no concept of a CSP on other platforms than Microsoft Windows.

The advantage of the CryptoAPI over PKCS#11 is that smart card vendors can write card mini-drivers. These drives present consistent interface to the Microsoft Smart Card Base Cryptographic Service Provider or Crypto Next Generation Key Storage Provider and to the Smart Card Management Interface. These card mini-drivers plug in to Windows operating system code [8].

3.1.2 PKCS#11 vulnerabilities

Although the PKCS#11 standard is widely used, it contains some vulnerabilities. According to "*Attacking and Fixing PKCS#11 Security Tokens*"[4] paper there is a chance for attacks on PKCS#11 compliant tokens. The feasible attacks have been detected by tool called **Tookan**.

The Tookan (TOOl for cryptoKi ANalysis) is an automated tool which reverse-engineers a real PKCS#11 token to deduce its functionality. It constructs a model of its API and then executes any attack trace found by the model checker directly on the token. The Tookan is able to extract sensitive cryptographic keys from a variety of cryptographic security tokens, exploiting vulnerabilities in their PKCS#11 based APIs [9].

Attacks

One common way to derive a device-specific symmetric key is to take a master key and some public data unique to the device, like a serial number, and encrypt the data under the key.

In PKCS#11, this is done by calling the `C_DeriveKey` function. In order to keep a key value as a secret, the `CKA_SENSITIVE` attribute is set to `TRUE`. The `C_DeriveKey` is implemented as specified in the standard, deriving the key from a key which is `CKA_SENSITIVE` and produces another key which is `CKA_SENSITIVE`.

However, when the master key has the attribute `CKA_DERIVE` and also `CKA_ENCRYPT` set to `TRUE`, an attacker can execute `C_Encrypt` using the master key and recover the value of the derived secret keys from the serial numbers.

Another attack is possible via a flaw in the PKCS#11 implementation. The value of sensitive keys is explicitly required to never be revealed outside of the token. In theory, when the token is asked for the value of a sensitive key, the "*value is sensitive*" error code is returned. In spite of that, some of the analysed devices just returned the plain key value, ignoring this basic policy [10].

Another violation of the PKCS#11 security policy is the possibility

4. `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/BCFS-ccs10.pdf`

of changing sensitive and unextractable keys respectively into non-sensitive and extractable ones [10].

3.2 PKCS#15

The PKCS#15 defines a standard that allows users of cryptographic tokens to identify themselves to multiple applications, regardless of the application's Cryptoki (or other token interface) provider. The IC-card-related parts of this standard are specified in ISO/IEC 7816-15 standard [11].

The "object-oriented" approach (treating keys, certificates and other data as objects with attributes and values) selected for PKCS#11, has been adopted for PKCS#15 as well [12].

4 Analysis and design

Smart cards usage during everyday tasks was analysed in order to obtain various test scenarios. The analysis was mostly concerned with the usage of smart cards in Fedora distribution of operating system Linux. The obtained data were used to design test suite that covers majority commonly used functions from PKCS#11 API. This way, all the common use cases of smart card usage are covered by tests.

There are many applications using smart cards for various objectives. They can be divided into two groups:

- applications using an already initialized smart card via PKCS#11 high-level functions. (e.g. Mozilla Firefox or TrueCrypt)
- applications, that work directly with data structures on a card. This category includes utility programs from the **OpenSC** project like **pkcs15-init** or **pkcs11-tool**, whose role is to initialize the card and prepare it for high level programs usage.

Each application works with a given smart card in different ways, but there are some common use cases:

- authentication – logging into Fedora via smart card belongs to this category. Linux authentication modules (**Linux-PAM**[1]) are responsible for the authentication to the Fedora system via a smart card.
  - a special example is the authentication to the remote server via ssh. The **OpenSSH** library is used for this purpose and it supports authentication via a smart card.
- disk encryption – representative of this category is **TrueCrypt**. It can create a virtual encrypted disk within a file or encrypt a partition or the entire storage device.

1. Pluggable Authentication Modules

- smart card initialization – utility programs like **pkcs11-tool** or **pkcs15-init** from the **OpenSC** project serve for a card initialization. Their tasks is either generation of RSA key pair on a card or the creation of PKCS#15 structure on a card.
- managing certificates – main operation used with certificates is reading. However, a new certificate can be also stored on or erased from a smart card. Certificates are mostly used for authentication for example to some web pages or to a remote server. Applications that are managing certificates are **Mozilla Firefox**, **pkcs11-tool** and others.
- data storage – smart cards can be used as data storage for sensitive data like finger prints, face or iris images

Particular information about smart card usage from applications were obtained the by utility module **pkcs11-spy** from the **OpenSC** project. Data for the analysis were obtained from these applications:

- OpenSSH
- Linux-PAM
- Mozilla Firefox
- pkcs11-tool

4.1 Usage of pkcs11-spy

The **pkcs11-spy** is a special PKCS#11 module that sits between the application and the real PKCS#11 module. It creates a log file with all functions calls from the application and returns values from the real PKCS#11 module [13].

The **pkcs11-spy** does not change the communication in any way, it only serves for logging sequence of PKCS#11 functions that are used by the application. These log files are security sensitive, since all information is logged, including PIN, PUK, signatures etc.[13]. This is the reason, why the pkcs11-spy should be only used for debugging and preferably only with test keys.

On Linux the PKCS#11 Spy is used with environment variables, which specify where the output is logged. `stderr` is used for logging by default, but the environment variable `PKCS11SPY_OUTPUT` can be set to a file name and all logging output is appended to that file [13]. The environment variable `PKCS11SPY` needs to be set to the real PKCS#11 module path such as **opensc-pkcs11.so** (absolute path should be used) in order to use the PKCS#11 module.

All outputs for data analysis were obtained by utility **pkcs11-spy**. Selected applications for data collection were OpenSSH, Linux-PAM, Firefox and utility pkcs11-tool.

4.1.1 Environment configuration

In order to work correctly with smart cards and with OpenSC utility programs, the Fedora environment has to be configured.

First, the middle-ware has to be installed in order to access a smart card using the SCard API (PC/SC) and a PKCS#11 standard interface for smart cards connected to a PC/SC compliant reader [14].

The **pcsclite** project provides the middle-ware layer. It is split into a few packages. These packages have to be installed:

```
$ dnf install pcsc-lite pcsc-lite-devel pcsc-lite-ccid
$ dnf install perl-pcsc pcsc-tools
```

When a user connects a smart card reader to a computer, it is recognized by the `pcsc_scan` program. The smart card is also recognized when it is inserted into the reader. The output example of the `pcsc_scan` is included in Appendix A.

The computer is now able to recognize the connected smart card. The OpenSC has to be installed in order to work with the smart card.

The OpenSC provides a set of libraries and utilities to work with smart cards. Its main focus is on cards that support cryptographic operations and facilitate their use in security applications such as authentication, mail encryption and digital signatures. The OpenSC implements the PKCS#11 API so applications supporting this API (such as Mozilla Firefox and Thunderbird) can use it. On the card, the OpenSC implements the PKCS#15 standard and aims to be com-

patible with every software and card that does so as well [15].

For the purpose of this thesis was used the latest OpenSC version cloned from GitHub. Before the installation of the OpenSC, two utility programs and the OpenSSL have to be installed:

```
$ dnf install autoconf automake
$ dnf install openssl openssl-devel engine_pkcs11
```

The OpenSSL has to be installed before the OpenSC, so the OpenSC can use some OpenSSL functions for reading certificates, working with public or private key, etc. In addition, the OpenSSL is used to check whether outputs from testing applications were correct. At this point, the OpenSC project can be cloned from GitHub and installed:

```
$ cd ~/Downloads
$ git clone https://github.com/OpenSC/OpenSC.git
$ cd OpenSC/
$ autoreconf --install --verbose
$ make
$ sudo make install
```

The last package required by the test application is **CMocka**[2]. The version 1.0.1 of the CMocka package was used in this thesis and it was downloaded from url `https://cmocka.org/files/1.0/`.

```
$ cd ~/Downloads
$ xz -d cmocka-1.0.1.tar.xz
$ tar vxf cmocka-1.0.1.tar
$ cd cmocka-1.0.1
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ..
$ make
$ sudo make install
```

After this configuration, the environment is ready for work with the smart card and the test application. All examples with the pkcs11-spy module assume, that the `Cryptoflex 32k e-gate` smart card is used.

2. `https://cmocka.org/`

4.1.2 Pkcs11-spy with OpenSSH

In order to get the output of the used PKCS#11 API functions with their parameters from the OpenSSH, the card needs to be initialized with correct objects. At the beginning, the `Cryptoflex` card has to be erased and initialized with a user PIN. This step ensures, that there are not any objects which are needed by the OpenSSH.

```
$ pkcs15-init -ET
$ pkcs15-init -CT --no-so-pin
$ pkcs15-init --store-pin --auth-id 01 --label "
    My test label" --pin 12345 --puk 54321 -T
```

The `-T` parameter for the `pkcs15-init` command means, that the default transport key is being used. If the transport key has been already changed this command fails. In this case, the command should be running without the `-T` option and a user would be asked for the transport key.

The next step is to generate a RSA key pair and store the certificate to the smart card. There are two possible options:

1. keys are generated on the smart card and the certificate will be imported to the card. It includes 3 steps:

   (a) generate RSA key pair on the smart card

   ```
   $ pkcs11-tool --login --pin 12345
   --keypairgen --key-type rsa:1024 --id a1
   --label "My generated key"
   ```

   (b) create the certificate from a public key, that was generated on the card. (Note during the creation of X.509 certificate inputs like "Common name", "Country name" have to be filled in)

   ```
   $ openssl
   OpenSSL> engine dynamic -pre SO_PATH:/usr/
       lib64/openssl/engines/libpkcs11.so -pre ID:
       pkcs11 -pre LIST_ADD:1 -pre LOAD -pre
       MODULE_PATH:/usr/lib64/opensc-pkcs11.
       so
   ```

```
OpenSSL> req -engine pkcs11 -new -key id_a1
    -keyform engine -x509 -out cert.der -
    outform DER
OpenSSL> quit
```

(c) store/import the certificate to the card

```
$ pkcs11-tool -l --pin 12345 --write-object
    cert.der --type cert --id a1
```

2. the keys are generated by the OpenSSL and imported to the card. These steps must be followed:

(a) create a 2048 bit RSA private key in DER format

```
$ openssl genpkey -algorithm RSA -out
    private_key.key -outform DER -pkeyopt
    rsa_keygen_bits:2048
```

(b) from the obtained private key generate a public key in DER format

```
$ openssl rsa -pubout -in private_key.key -
    inform DER -out public_key.key -outform
    DER
```

(c) create the X.509 certificate

```
$ openssl req -out cert.der -key private_key.key
    -keyform DER -new -outform DER -x509 -
    days 365
```

(d) import all created objects to the smart card

```
$ pkcs11-tool -l --pin 12345 --label "My
    Private Key" --type privkey --write-object
    private_key.key --id a1 --usage-sign --
    usage-decrypt
$ pkcs11-tool -l --pin 12345 --label "My
    Public Key" --type pubkey --write-object
    public_key.key --id a1 --usage-sign --
    usage-decrypt
```

$ pkcs11−tool −l −−pin 12345 −−write−object cert.der −−type cert −−id a1 −−label "My certificate"

At this moment, there are 4 objects on the card: the user PIN, the private key, the public key and the certificate. In order to list all objects from the card the command `pkcs15-tool -D` is used. Output is as follows:

```
PKCS#15 Card [OpenSC Card]:
Version        : 0
Serial number  : 0000D909FFFF0200
Manufacturer ID: OpenSC Project
Last update    : 20160221162253Z
Flags          : EID compliant

PIN [My test label]
Object Flags   : [0x3], private, modifiable
ID             : 01
Flags          : [0x32], local, initialized,
                         needs-padding
Length         : min_len:4, max_len:8, stored_len:8
Pad char       : 0x00
Reference      : 1 (0x01)
Type           : ascii-numeric
Path           : 3f0050154b01

Private RSA Key [My Private Key]
Object Flags   : [0x3], private, modifiable
Usage          : [0x2E], decrypt, sign,
                         signRecover, unwrap
Access Flags   : [0x0]
ModLength      : 2048
Key ref        : 0 (0x0)
Native         : yes
Path           : 3f0050154b0130000012
Auth ID        : 01
ID             : a1
```

```
MD:guid         : {4a6c27d7-802a-5be2-b97d-88aab52f5238}

Public RSA Key [My Public Key]
Object Flags    : [0x2], modifiable
Usage           : [0xD1], encrypt, wrap,
                          verifyRecover, verify
Access Flags    : [0x0]
ModLength       : 2048
Key ref         : 0 (0x0)
Native          : no
Path            : 3f0050154800
ID              : a1

X.509 Certificate [Certificate]
Object Flags    : [0x2], modifiable
Authority       : no
Path            : 3f0050154500
ID              : a1
Encoded serial : 02 09 00C45607B26EE39A4A
```

An ssh key has to be created to use the OpenSSH with the generated certificate. (Note if the OpenSSL was not installed before the OpenSC, there would be no option `read-ssh-key` for `pkcs15-tool` utility)

$ pkcs15−tool −−read−ssh−key a1

The obtained key should be copied to a `~/.ssh/authorized_keys` file on a remote server.

aisa:/home/xusername>$ cat ~/.ssh/authorized_keys
ssh−rsa ${obtained_key}

After all these steps, everything is set for the use of the OpenSSH with the smart card to login via ssh to the remote server. The path to the PKCS#11 module has to be specified as a parameter to ssh command. When the path to the pkcs11-spy module is set as a parameter, all PKCS#11 functions used by the OpenSSH are logged to the file.

$ export PKCS11SPY=/lib64/opensc−pkcs11.so
$ export PKCS11SPY_OUTPUT=/path/to/output.log

```
$ ssh -I /usr/lib64/pkcs11/pkcs11-spy.so xstrhars@fi.
    muni.cz
```

A user is asked to provide a PIN for the smart card. When the PIN is correct, the user is logged in to the remote server.

4.1.3 Pkcs11-spy with pkcs11-tool

A prerequisite for using the **pkcs11-tool** utility is a reinitialized card without any unnecessary objects (e.g like in chapter 4.1.2).

The **pkcs11-tool** is universal and can be used with most operations supported by smart cards. According to the documentation, a subset of functions was selected to get the pkcs11-spy output.

Note that before using any of these commands, the smart card has to be reinitialized and pkcs11-spy environment variables have to be exported. All examples of pkcs11-tool commands are included in Appendix B.

```
$ export PKCS11SPY=/lib64/opensc-pkcs11.so
$ export PKCS11SPY_OUTPUT=/path/to/output.log
```

Selected commands to analyse were:

- **test** – to run basic tests on the smart card e.g. random data generation tests, tests of signatures, verification, message digest, decryption and key unwrap.
- **list slots** – to list all available slots (physical and virtual) and display their information (e.g. card insertion, virtual or physical slot). Note that it is not necessary to have a user PIN on the card.
- **list mechanisms** – to list all mechanisms supported by the card. Every mechanism has its name and is flagged for which function it can be used for. Possible function flags are `digest`, `sign`, `verify`, `decrypt` and `generate_key_pair`. Some mechanisms have a supported size of keys, that can be used (`keySize={512,2048}`). Note, that it is not necessary to have a user PIN on the card.

- **list objects** – to list all objects stored on the card. If this option is used without logging in to the card, it will list only public available objects such as certificates or public keys. When the login and pin options are provided, private objects are also shown.

- **init pin** – to initialize user's PIN. Only the Security Officer (SO) has privileges to initialize the user PIN, therefore he must be logged in. A default SO PIN for the `Cryptoflex` smart card is `00000000`. Note, that for this command, the card must not be already initialized with the user PIN.

- **change pin** – to change an old user PIN to a new one. To change the PIN, a user has to log in to the smart card.

- **test ec** – to test whether elliptic curves are supported. This test checks whether the mechanism `CKM_EC_KEY_PAIR_GEN` is supported by the card.

- **key pair gen** – to generate key pair on the card. It is possible to choose between two key pair types – `rsa:{keySize}` or `EC:prime256v1`. Mostly only the `rsa` is supported. The card must support the mechanism for generating a key pair (it has a flag `generate_key_pair`). A key size must also be specified before the generation of the key pair.

- **read object** – to read object stored on a card. Some objects, like a private key cannot be read from the card, therefore this command fails. Furthermore, an id of the object and its type must be provided.

- **write object** – to write/import object to the card. Four types of objects can be imported to the card – certificate (`cert`), private key (`privkey`), public key (`pubkey`) and data (`data`)

- **delete object** – to delete object from the card. This operation usually fails because deletion is not supported by the majority of smart cards.

- **sign** – to sign message with a private key. A prerequisite is that a private key has to exist on the card. Verification is not the part

of the pkcs11-tool, because it uses a public key which is rarely present on the smart cards.

- **decrypt** – to decrypt an encrypted message. There is no support for encryption due to the same reason as the verification. Moreover, most cards do not have a mechanism with the `encrypt` flag.

- **hash** – to use a message digest algorithm to create a hash of an input message. There can be more than one message digest algorithm supported by the card. Commonly used are `SHA-1`, `SHA256`, `SHA512`, `MD5`, `RIPEMD160`, `GOSTR3411`. It is possible to specify which message digest mechanism is used to create the hash of the message.

Most of the data for analysis were obtained by means of the **pkcs11-spy** utility module with the **pkcs11-tool**.

4.1.4 Pkcs11-spy with Mozilla Firefox

A prerequisite for using program **Mozilla Firefox** with the pkcs11-spy is a reinitialized card without any unnecessary objects (as in chapter 4.1.2).

There are three ways how Mozilla Firefox works with certificates:

1. to read a certificate

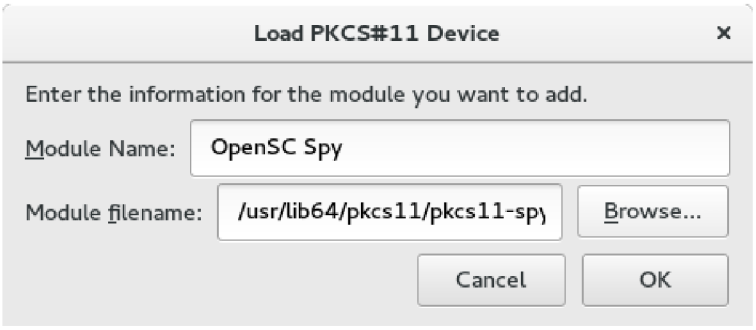
2. to store/import a certificate to a card

3. to delete a certificate – on a `Cryptoflex` card only the public key is deleted from the card, but the certificate and the private key remained on the smart card

In order to make Mozilla Firefox work with the smart card a new Security Device has to be added to Firefox. Firstly, the pkcs11-spy environment variables are exported. Subsequently, Firefox is executed from the command line, so it is possible to add the pkcs11-spy security device.

```
$ export PKCS11SPY=/lib64/opensc-pkcs11.so
$ export PKCS11SPY_OUTPUT=/path/to/firefox.log
$ firefox
```

The path to the configuration of security devices is as following: `Menu - Preferences - Advanced - Security Devices`. Afterwards `Load` is pressed and the `Load PKCS#11 Device` dialogue window is opened. The name of the module and path to the pkcs11-spy utility module is filled as shown in the picture 4.1-1



Img. 4.1-1. Load PKCS#11 Device dialogue

Importing of a certificate

Firefox supports certificate importing only in PKCS#12 format. In order to create the importing certificate, these steps have to be followed:

1. using the OpenSSL generate a private key

   ```
   $ openssl genpkey -algorithm RSA -out private_key.
       key -pkeyopt rsa_keygen_bits:2048
   ```

2. from the obtained private key create the certificate in PEM[3] format

   ```
   $ openssl req -out cert.pem -key private_key.key -
       new -x509 -days 365
   ```

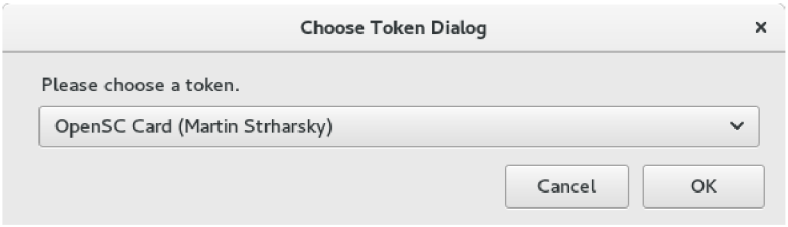
3. Privacy Enhanced Mail

3. create the certificate in PKCS#12 format using the private key and the certificate in PEM format. A user is asked to provide a password, which will be subsequently used during the certificate importing from Firefox.

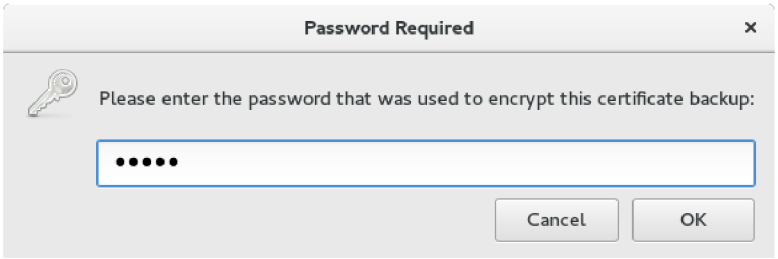
   $ openssl pkcs12 −export −inkey private_key.key −in cert.pem −out cert.p12

The user has to navigate to `Menu - Preferences - Advanced - Security Devices` and press `View Certificates` in order to import the newly created certificate. The password required dialogue is displayed and the user has to fill in the PIN to the smart card

After `Import` is pressed, the user is asked to choose, the certificate location (image 4.1-2), the password dialogue for token PIN and the certificate password (image 4.1-3).



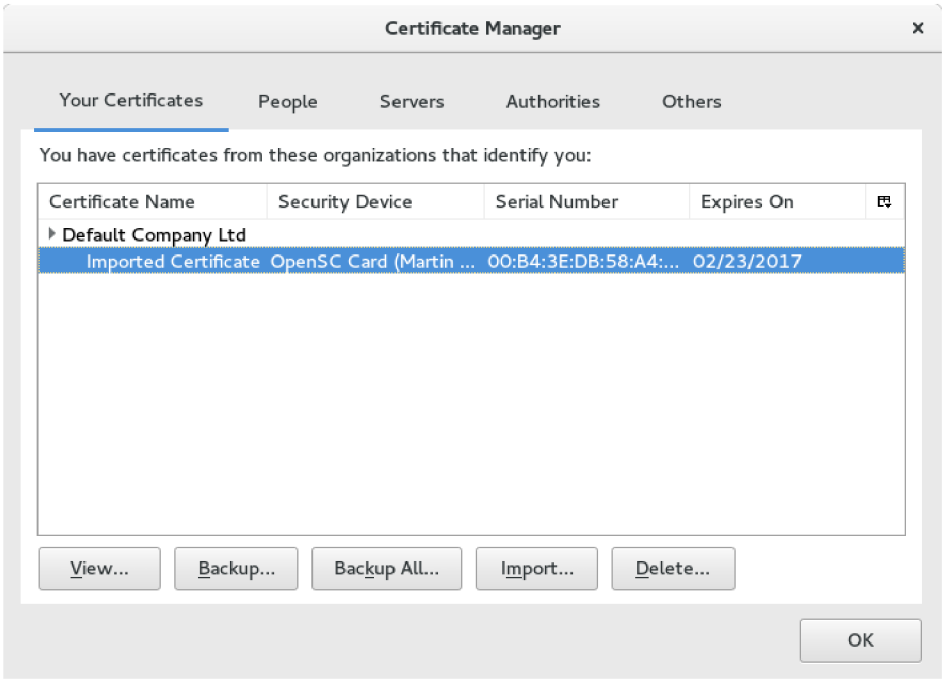
Img. 4.1-2. Import certificate dialogue



Img. 4.1-3. Certificate password

At this moment, the certificate is successfully imported to the smart card. It is shown in `View Certificates` dialogue (image 4.1-4)

The imported certificate, the private and the public key can are shown via "`pkcs15-tool -D`" command. The output example is included in Appendix C.



Img. 4.1-4. View imported certificate

The pkcs11-spy output log is used to verify, that the certificate has been in fact imported to the smart card. In the log file, the function `C_CreateObject` is used at least 3 times to subsequently create the private key, the public key and the certificate. The log file includes all input parameters of the created objects such as a modulus, a private exponent, prime numbers and others.

4.1.5 Pkcs11-spy with Linux-PAM

Linux-PAM (Pluggable Authentication Modules) is an authentication framework that uses modules to authenticate users by a wide variety of methods [16]. A PKCS#11 PAM module enables smart cards to authenticate against any service that uses PAM. The most obvious usage of PAM is in system logins, either console or graphical. However, a lot of other services, for example sudo, use it as well.

The PKCS#11 PAM module is found in the `pam_pkcs11` package in the repositories.

Several methods are used to "map" a smart card to a user. One of the methods is **pwent**. The **pwent** checks the CN (Common Name) field of the X.509 certificate associated with a key, and grant access only if it matches either the login name or the real name of the user. Alternative mappers are [17]:

- **Common Name (CN) mapper** - assumes the CN field on the certificate to be the login name.
- **Subject mapper** - extracts the Certificate Subject and assumes it as login.
- **LDAP**[4] **mapper** - uses an LDAP server to retrieves the user name. An additional file informs the module about the mapping between Cert fields and LDAP entries.
- **OpenSC library mapper** - searches for the certificate in `${HOME}/.eid/autorized_certificates` in a similar way as the OpenSC does. When it is used as a login finder, it returns the user, who owns the `${HOME}` directory, to the certificate location.
- **OpenSSH library mapper** - searches for the certificate public key in `${HOME}/.ssh/authorized_keys` in a similar way as the OpenSSH does. The openssh mapper uses Naming Service Switch (NSS) via `getpwent()` to get the list of users and home directories.
- **Email Cert to login mapper** - is an email mapper that extracts an e-mail from the certificate. When the `mapfile` option is set and the file is provided, the module maps the email field from the certificate to a user (or an alternate email). When the `mapfile` is not set, only an email address from the certificate is used to perform find/match.
- **Microsoft Universal Principal Name mapper** - finds and uses a Microsoft Universal Principal Name (UPN) extension to get the login name. The Microsoft UPN is an ASN1-encoded UTF8 string with the syntax `login@ADS_Domain`. When a UPN is found, the mapper extracts login part as the login user.

4. Lightweight Directory Access Protocol

- **Kerberos mapper** - finds and uses Kerberos Principal Name (KPN) as the login name. When the `mapfile` is specified, it maps KPN into the login.
- **Unique ID to login mapper** - uses the Unique ID (UID) field as the login name. It is similar to the CN mapper, but uses the UID as the field to find/match.
- **Certificate Digest to login mapper** - evaluates the certificate digest and maps the result into a login by using a `mapfile`.
- **Generic mapper** - groups several mappers into one. A user selects which certificate content is used to deduce/match login and whether a file mapping is wanted. If it is desired, it consults mapped string to NSS services to get the final user login.
- **Null mapper** - is a blind access/deny mapper.

After the `pam_pkcs11` package installation, some additional manual configuration has to be performed. Firstly, some additional directories are created in `/etc/pam_pkcs11`.

The directory `/etc/pam_pkcs11/cacerts` stores the certificates of all trusted certificate authorities – a certificate is accepted only if it has been signed by one of those certificate authorities. On the other hand, the directory `/etc/pam_pkcs11/crls` stores the Certificate Revocation Lists sent by the certificate authorities, so the PAM module knows which certificates are no longer valid (revoked).

The private key and the X.509 certificate for the key is generated and stored on the card. Note that the Common Name (CN) field is used to match the certificate to the account. Therefore either the login or the real name has to be put in the field.

```
$ openssl genpkey -algorithm RSA -out private_key.key -
    pkeyopt rsa_keygen_bits:2048 -outform DER
$ openssl req -out certificate.pem -key private_key.key -
    keyform DER -new -x509 -days 365
$ openssl x509 -in certificate.pem -out certificate.der -
    outform der
```

```
$ pkcs11-tool -l --pin 12345 --write-object private_key
    .key --type privkey --label "My Private Key"
$ pkcs11-tool -l --pin 12345 --write-object certificate.
    der --type cert --label "My certificate"
```

The created certificate is copied in PEM format to the directory `/etc/pam_pkcs11/cacerts` (self-sign certificate is used) and the list of CA certificates is rehashed with:

```
$ cd /etc/pam_pkcs11/cacerts
$ sudo pkcs11_make_hash_link
```

Linux-PAM works correctly with the smart card, only when the used certificate is registered as "trusted" in NSS[5]. The first step is to create a folder for NSS database and register it:

```
$ sudo mkdir -p /etc/pam_pkcs11/nssdb
$ sudo chmod 700 /etc/pam_pkcs11/nssdb
$ certutil -d /etc/pam_pkcs11/nssdb -N
```

The next step is to import the created certificate to the certificate database. It has to be set as trusted, so that it can be used during authorization via the smart card.

```
$ sudo certutil -A -n "Certificate nickname" -t "CT,C,
    C" -a -d /etc/pam_pkcs11/nssdb -i /path/to/
    created/certificate
```

By default, Linux-PAM is not set to use the `OpenSC` module. Hence, there have to be changes in the configuration. These variables are changed in configuration file `/etc/pam_pkcs11/pam_pkcs11.conf`:

- the **debug** property for the root element `pam_pkcs11` is set to be true, when the verbose debug output is necessary

```
pam_pkcs11 {
    # Allow empty passwords
    nullok = true;
```

5. Name Service Switch

```
    # Enable debugging support.
    debug = true;
}
```

- the PKCS#11 module, has to be specified in the `use_pkcs11_module` property

```
pam_pkcs11 {
    # Filename of the PKCS #11 module.
    use_pkcs11_module = opensc;
}
```

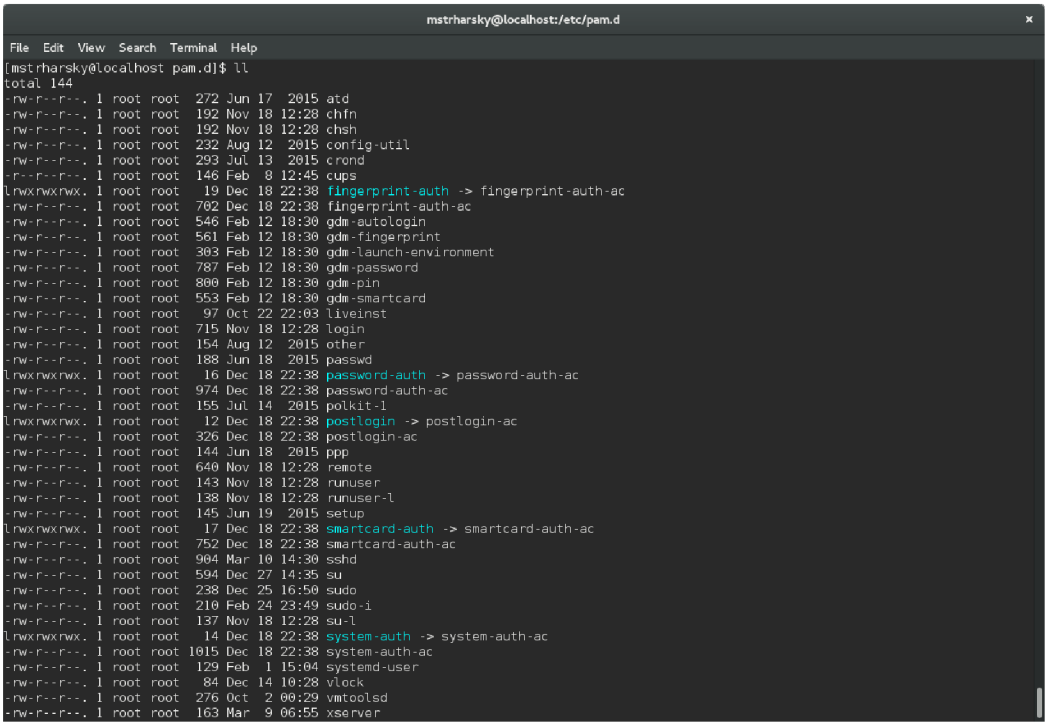
- the OpenSC PKCS#11 module, where the path to the `pkcs11-spy.so` library and correct paths to CA and CRL directories must be set

```
pkcs11_module opensc {
    module = /usr/lib64/pkcs11/pkcs11−spy.so;
    description = "OpenSC PKCS#11 module";

    slot_num = 0;

    ca_dir = /etc/pam_pkcs11/cacerts;
    crl_dir = /etc/pam_pkcs11/crls;

    # set the certificate policy
    cert_policy=ca, signature;
}
```

- `use_mappers` specifies mappers, which are used during the smart card authentication. It is sufficient if there is only the `pwent` mapper, which checks only the Common Name field of the X.509 certificate

```
pam_pkcs11 {
        use_mappers = pwent;
}
```

The last step is to choose which services are using the smart card authentication. All available services are listed in `/etc/pam.d` directory as shown in the picture 4.1-5.



Img. 4.1-5. Linix-PAM services

The `su` service from `/etc/pam.d/` is used with the **pkcs11-spy**, because services like `login` or `sudo` clean up the environment before they start. As a result, the used environment variables `PKCS11SPY` and `PKCS11SPY_OUTPUT` are not set and there is no output for the **pkcs11-spy**.

In order to use `su` with the smart card the `pam_pkcs11.so` has to be registered in the `/etc/pam.d/su` configuration file:

```
#%PAM-1.0
auth   sufficient  pam_pkcs11.so
auth   sufficient  pam_rootok.so
```

Finally, everything is set for using Linux-PAM authentication with the pkcs11-spy module.

```
$ export PKCS11SPY=/lib64/opensc-pkcs11.so
$ export PKCS11SPY_OUTPUT=/path/to/output.log
$ su -l mstrharsky
```

5 Implementation

The OpenSC project provides a PKCS#11 library (opensc-pkcs11.so), which is responsible for a communication with smart cards. In order to communicate with a card the specific card driver is used. There are smart card drivers for PIV (Personal Identity Verification) cards, PKCS#15 cards and for other types of cards.

The goal of the unit testing application, is to run a set of tests against specified card drivers. Tested smart card drivers were:

- driver for PIV cards (tested with `YubiKey Neo`)
- driver for PKCS#15 cards (tested with `Cryptoflex 32k`)

All tests are part of the forked OpenSC repository[1] and can be executed during the OpenSC installation. In order to run the test suite, these steps have to be followed:

- cloning the forked OpenSC repository

  ```
  $ git clone https://github.com/strho/OpenSC.git
  ```

- changing directory to the forked OpenSC and run the build

  ```
  $ cd OpenSC
  $ autoreconf -fvi
  $ ./configure
  $ make
  ```

- running the tests (Note that the token has to be connected to the PC, otherwise tests are skipped)

  ```
  $ make check
  ```

At the end of the command "`make check`", the test suite summary is reported. It reports number of passed, skipped or failed tests. Two tests are run:

1. `https://github.com/strho/OpenSC`

- `yubico_test.sh` – tests PIV driver
- `cryptoflex_test.sh` – tests Cryptoflex driver

When the PIV card is connected to the PC (e.g. `YubiKey`), the test for the Cryptoflex card is skipped. Likewise, when there is no card connected, both tests are skipped. This is the output of the "`make check`" command:

```
FAIL: yubico_test.sh
============================================
Testsuite summary for OpenSC 0.15.0
============================================
# TOTAL: 2
# PASS:  0
# SKIP:  1
# XFAIL: 0
# FAIL:  1
# XPASS: 0
# ERROR: 0
============================================
See tests/test-suite.log
============================================
```

The log file `yubico_test.sh.log` contains single results of the individual tests. At the end of the file, there is a summary displaying the number of launched and failed tests. The example of log file with failed tests is included in Appendix D.

5.1 Technologies

The smart card test application was implemented in the C programming language. In order to work with a smart card, the standard PKCS#11 API (called Cryptoki) is used. The implementation of the Cryptoki is provided by the **opensc-pkcs11.so** library. The **CMocka** is selected as the unit testing framework. The **CLion IDE**[2] by JetBrains[3] was used during the implementation.

2. Integrated Development Environment
3. `https://www.jetbrains.com/`

5.1.1 CMocka

The CMocka is a unit testing framework for C. One of the principles of the CMocka is that a test application requires only the standard C library and the CMocka itself [18]. This minimizes conflicts with standard C library headers, especially on a variety of different platforms. The CMocka supports [19]:

- mock objects – mock objects are simulation objects that mimic the real implementation of actual objects. They are used to simulate dependencies of an interface in order to help testing the interface in isolation.
- several output formats – by default, the test output is printed to the `stderr`. It is possible to configure several other output formats. The configuration is performed by the environment variable `CMOCKA_MESSAGE_OUTPUT`. The supported values are:
  - `STDOUT` – for the default standard output printer.
  - `SUBUNIT` – for subunit output.
  - `TAP` – for Test Anything Protocol (TAP) output.
  - `XML` – for xUnit XML format.
- test fixtures – test fixtures are **setup** and **teardown** functions that can be shared across multiple tests. They provide common functions to prepare and destroy the test environment.

The CMocka is released under the Apache License Version 2.0.

5.2 Test cases

The Cryptoki offers many cryptographic functions, which are used for working with tokens. According to PKCS#11 documentation[4] released by RSA Laboratories in January 2004, functions are organized into the following categories [20]:

4. `ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf`

- general-purpose functions (4 functions)

Function	Description
C_Initialize*	initializes the Cryptoki
C_Finalize*	cleans up miscellaneous resources associated with the Cryptoki
C_GetInfo*	obtains general information about the Cryptoki
C_GetFunctionList*	obtains entry points of the Cryptoki library functions

- slot and token management functions (9 functions)

Function	Description
C_GetSlotList*	obtains a list of slots in the system
C_GetSlotInfo*	obtains information about a particular slot
C_GetTokenInfo*	obtains information about a particular token
C_WaitForSlotEvent	waits for a slot event (token insertion, removal, etc.) to occur
C_GetMechanismList*	obtains a list of mechanisms supported by a token
C_GetMechanismInfo*	obtains information about a particular mechanism
C_InitToken	initializes a token
C_InitPIN*	initializes a normal user's PIN
C_SetPIN*	modifies a PIN of a current user

- parallel function management functions (2 functions)

Function	Description
C_GetFunctionStatus	legacy function which always returns `CKR_FUNCTION_NOT_PARALLEL`
C_CancelFunction	legacy function which always returns `CKR_FUNCTION_NOT_PARALLEL`

- session management functions (8 functions)

Function	Description
C_OpenSession*	opens a connection between an application and a particular token or sets up an application callback for a token insertion
C_CloseSession*	closes a session
C_CloseAllSessions*	closes all sessions with a token
C_GetSessionInfo	obtains information about a session
C_GetOperationState	obtains a cryptographic operation state of a session
C_SetOperationState	sets a cryptographic operation state of a session
C_Login*	logs into a token
C_Logout*	logs out from a token

- object management functions (9 functions)

Function	Description
C_CreateObject*	creates an object
C_CopyObject	creates a copy of an object
C_DestroyObject*	destroys an object
C_GetObjectSize	obtains the size of an object in bytes
C_GetAttributeValue*	obtains an attribute value of an object
C_SetAttributeValue	modifies an attribute value of an object
C_FindObjectsInit*	initializes an object search operation
C_FindObjects*	continues an object search operation
C_FindObjectsFinal*	finishes an object search operation

- encryption functions (4 functions)

Function	Description
C_EncryptInit	initializes an encryption operation
C_Encrypt	encrypts single-part data
C_EncryptUpdate	continues a multiple-part encryption operation
C_EncryptFinal	finishes a multiple-part encryption operation

- decryption functions (4 functions)

Function	Description
C_DecryptInit*	initializes a decryption operation
C_Decrypt*	decrypts single-part encrypted data
C_DecryptUpdate	continues a multiple-part decryption operation
C_DecryptFinal	finishes a multiple-part decryption operation

- message digesting functions (5 functions)

Function	Description
C_DigestInit*	initializes a message-digesting operation
C_Digest*	digests single-part data
C_DigestUpdate*	continues a multiple-part digesting operation
C_DigestKey	digests a key
C_DigestFinal*	finishes a multiple-part digesting operation

- signing and MACing[5] functions (6 functions)

Function	Description
C_SignInit*	initializes a signature operation
C_Sign*	signs single-part data
C_SignUpdate	continues a multiple-part signature operation
C_SignFinal	finishes a multiple-part signature operation
C_SignRecoverInit	initializes a signature operation, where the data can be recovered from the signature
C_SignRecover	signs single-part data, where the data can be recovered from a signature

5. Message Authentication Code

- functions for verifying signatures and MACs (6 functions)

Function	Description
C_VerifyInit*	initializes a verification operation
C_Verify*	verifies a signature on single-part data
C_VerifyUpdate*	continues a multiple-part verification operation
C_VerifyFina*	finishes a multiple-part verification operation
C_VerifyRecoverInit	initializes a verification operation that recovers data from a signature
C_VerifyRecover	verifies a signature on single-part data, where data are recovered from the signature

- dual-purpose cryptographic functions (4 functions)

Function	Description
C_DigestEncryptUpdate	continues simultaneous multiple-part digesting and encryption operations
C_DecryptDigestUpdate	continues simultaneous multiple-part decryption and digesting operations
C_SignEncryptUpdate	continues simultaneous multiple-part signature and encryption operations
C_DecryptVerifyUpdate	continues simultaneous multiple-part decryption and verification operations

- key management functions (5 functions)

Function	Description
C_GenerateKey	generates a secret key
C_GenerateKeyPair*	generates a public-key/private-key pair
C_WrapKey	wraps (encrypts) a key
C_UnwrapKey	unwraps (decrypts) a key
C_DeriveKey	derives a key from a base key

- random number generation functions (2 functions)

Function	Description
C_SeedRandom*	mixes in additional seed material to the random number generator
C_GenerateRandom*	generates random data

All data obtained by the utility module **pkcs11-spy** were processed and various test cases are created. The test cases cover majority of commonly used cryptographic functions offered by the Cryptoki (functions marked with asterisk) and are divided into the following categories, which are elaborated on the next section:

- General tests
- User PIN tests
- Message digest tests
- Key generation tests
- Sign and verify tests
- Encrypt and decrypt tests
- Find objects tests
- Generate random data tests
- Create and delete objects tests

5.2.1 General tests

Test name: Get all mechanisms test

- **Called function**: `get_all_mechanisms_test`
- **Description**: This test checks whether there are any supported mechanisms on the card
- **Tested Cryptoki functions**: `C_GetMechanismList` and `C_GetMechanismInfo`

- **Fail condition**: The card does not have any supported mechanisms

- **Side effect**: The `supported` property of `token_info_t` structure is set. It states, whether there are any sufficient mechanisms for e.g. encryption, decryption, message digest, etc. or not. When there is not any mechanisms for currently running test, the test is skipped.

Test name: Elliptic curves test

- **Called function**: `is_ec_supported_test`

- **Description**: Tests whether elliptic curves are supported by the card. It controls if the mechanism with the `CKM_EC_KEY_PAIR_GEN` flag exists on the card

- **Fail condition**: The test cannot fail, but it is skipped when there is no mechanism with the `CKM_EC_KEY_PAIR_GEN` flag

5.2.2 User PIN tests

A precondition for tests in this group is that a card is initialized with the user PIN. The PIV cards (tested with `YubiKey Neo`) are already initialized with the PIN, but for the PKCS#15 card, a new user PIN has to be created.

Test name: Initialize token with user PIN test

- **Called function**: `initialize_token_with_user_pin_test`

- **Description**: At first the test logs into the card with incorrect PIN and the `CKR_PIN_INCORRECT` return value is expected. Subsequently the correct PIN is used to log into the card and the `CKR_OK` is returned.

- **Tested Cryptoki functions**: `C_Login`, `C_InitPIN` for PKCS#15 cards

- **Fail condition**: The test fails, when an incorrect PIN is used and the `CKR_PIN_INCORRECT` value is not returned from the `C_Login` function. When the correct PIN is used, the `CKR_OK` is expected.

Test name: Change user PIN test

- **Called function**: `change_user_pin_test`
- **Description**: An existing user PIN is changed. The old PIN is entered as a password and the `CKR_PIN_INCORRECT` value is returned from the `C_Login` function. Afterwards the new PIN is used and the login is successful.
- **Tested Cryptoki functions**: `C_SetPIN`
- **Fail condition**: The function `C_Login` returns other value than the `CKR_PIN_INCORRECT`. It is not possible to log into the card with the newly created user PIN.

5.2.3 Message digest tests

Only two message digest algorithms were tested: `MD5` and `SHA1`. If either one of these mechanisms is not supported by the card, the test for a given algorithm is skipped. There are two test variants for both algorithms:

1. digest a short message (use `C_DigestInit` and `C_Digest` functions)
2. digest a message, that is read from a file (use `C_DigestInit`, `C_DigestUpdate` and `C_DigestFinal` functions)

The output hash of the message, created by the card, is compared to the hash obtained by the OpenSSL on the same message.

Test name: Tests of MD5 digest algorithm

- **Called function**: `create_hash_md5_short_message_test` and `create_hash_md5_long_message_test`

- **Description**: At first, the `CKM_MD5` mechanism is used to create a hash of the short string message. The functions `C_DigestInit` and `C_Digest` are used.

  Next, the test reads a message from an enclosed file and uses `CKM_MD5` mechanisms to obtain the message digest. The digest functions `C_DigestInit`, `C_DigestUpdate` and `C_DigestFinal` are called.

  Output hashes from both tests are compared to hashes obtained from OpenSSL on the same messages.

- **Tested Cryptoki functions**: `C_DigestInit`, `C_Digest` and functions `C_DigestUpdate`, `C_DigestFinal`

- **Fail condition**: The test fails, when any Cryptoki functions returns other value than `CKR_OK`. Furthermore, when the output hash obtained by the card is different from the hash, created with the OpenSSL, the test fails.

Test name: Tests of SHA1 digest algorithm

- **Called function**: `create_hash_sha1_short_message_test` and `create_hash_sha1_long_message_test`

- **Description**: The test is similar to the *MD5 digest algorithm* test. However, instead of the `CKM_MD5` it uses the `CKM_SHA1` mechanism.

- **Tested Cryptoki functions**: `C_DigestInit`, `C_Digest` and functions `C_DigestUpdate`, `C_DigestFinal`

- **Fail condition**: Fail conditions are the same as for the *Tests of MD5 digest algorithm* test.

5.2.4 Key generation tests

When a card does not support any mechanism for key pair generation flagged with `CKF_GENERATE_KEY_PAIR`, tests are skipped.

Test name: Generate a RSA key pair on a token with a wrong template

- **Called function**: `generate_rsa_key_pair_no_key_generated_test`
- **Description**: The test tries to generate a RSA key pair with an incorrect template. After the `C_GenerateKeyPair` function is used, the private key and the public key are not generated and stored on the card
- **Tested Cryptoki functions**: `C_GenerateKeyPair`
- **Fail condition**: When either one of the keys is generated on the card, the test fails.

Test name: Generate a RSA key pair on a token

- **Called function**: `generate_rsa_key_pair_test`
- **Description**: The test generates a RSA key pair on the token, with given properties (id, label, exponent and modulus for public key, etc.). After the successful key generation, the test searches for the objects with given id and object class (`CKO_PUBLIC_KEY` and `CKO_PRIVATE_KEY`).

  The `CK_OBJECT_HANDLE` values that are obtained by the functions `C_FindObjects` and `C_GenerateKeyPair` are identical.
- **Tested Cryptoki functions**: `C_GenerateKeyPair`
- **Fail condition**: When object handles obtained by `C_FindObjects` and `C_GenerateKeyPair` functions are not the same, the test fails. Moreover, the value of the modulus of the stored public key must be the same as the modulus used during the key generation.

5.2.5 Sign and verify tests

The test RSA key pair is pre-generated using the OpenSSL in order to properly test signing and verification. A private and a public key are imported to the token before all tests.

Test name: Sign a message and compare the signature

- **Called function**: `sign_message_test`
- **Description**: The test searches for a private key with specific properties on the card. Afterwards, it uses the obtained key to create a signature for the test message. The output signature is compared to the signature created with the same private key, by the OpenSSL. The comparison of signatures is made on a byte level.
- **Tested Cryptoki functions**: `C_SignInit`, `C_Sign`
- **Fail condition**: The test fails when:
  - the imported private key is not found
  - signing the Cryptoki functions return other value than `CKR_OK`
  - the obtained signature is different from the signature created by the OpenSSL.

Test name: Verify a signed message

- **Called function**: `verify_signed_message_test`
- **Description**: Firstly, the imported public key is looked for. Then the signature created by the OpenSSL is read from the file. The next step is to use `C_VerifyInit` and `C_Verify` functions in order to determine whether the verified signature is the same in the test message that has been signed.
- **Tested Cryptoki functions**: `C_VerifyInit`, `C_Verify`
- **Fail condition**: The test fails when:
  - a public key is not found on the card
  - the verification of the signature returns other value than `CKR_OK`

5.2.6 Encrypt and decrypt tests

A prerequisite for tests in this group is the same as in section 5.2.5. A key pair has to be generated and imported to the token.

The encryption test is not provided, because the majority of cards does not support encryption and does not contain any mechanisms flagged with `CKF_ENCRYPT`.

Test name: Decrypt an encrypted message

- **Called function**: `decrypt_encrypted_message_test`
- **Description**: The message is encrypted using the pre-generated public key and stored to the file. The test finds the private key on the card and uses it to decrypt the encrypted message. Subsequently the decrypted message is compared to the test message.
- **Tested Cryptoki functions**: `C_DecryptInit`, `C_Decrypt`
- **Fail condition**: The test fails when:
  - a private key is not found
  - decryption functions do not return `CKR_OK` values
  - an output message is different from the test message, that is encrypted

5.2.7 Find objects tests

Similarly to previous chapters the same RSA key pair is imported to the card. Additionally, the `X.509` certificate is created from the public key and also imported to the card.

Test name: Find all imported objects

- **Called function**: `find_all_objects_test`
- **Description**: The certificate, the public key and the private key are imported to the token. The `C_FindObjects` function is used to find all objects on the card, The object class found is one of these: `CKO_PRIVATE_KEY`, `CKO_PUBLIC_KEY` or `CKO_CERTIFICATE`.

At the end of the test, there are exactly three (imported) objects found that respect the criteria.

- **Disclaimer**: The Cryptoki `C_FindObjects` function cannot be used to find all objects on the token. This is due to the fact that some tokens (e.g. `YubiKey Neo`) have predefined objects, which are present on the card as default.
- **Tested Cryptoki functions**: `C_FindObjectsInit`, `C_FindObjects` and `C_FindObjectsFinal`
- **Fail condition**: The test fails when:
  - find functions do not return the `CKR_OK` value
  - the number of found objects is not exactly three

Test name: Find an object according to a template

- **Called function**: `find_object_according_to_template_test`
- **Description**: A specific template is used to find a specific object on the card. The `C_FindObjects` function returns the imported certificate.
- **Tested Cryptoki functions**: `C_FindObjectsInit`, `C_FindObjects` and `C_FindObjectsFinal`
- **Fail condition**: The test fails when:
  - find functions do not return the `CKR_OK` value
  - the certificate is not found

Test name: Find object and read attributes

- **Called function**: `find_object_and_read_attributes_test`
- **Description**: An imported certificate is found on the card. Then an attribute template is used to read the certificate attributes. Finally, all read attributes are checked, whether they have a correct value. The read attributes of the certificate, are:

  - certificate type – `CKC_X_509` value
  - label – the `"Certificate"` value
  - subject, issuer and serial number – all obtained values are checked on a byte level, since they are returned in `DER`[6] format, which is binary encoded

- **Tested Cryptoki functions**: `C_FindObjectsInit`, `C_FindObjects`, `C_FindObjectsFinal` and `C_GetAttributeValue`
- **Fail condition**: The test fails when:
  - a certificate is not found
  - any of the acquired properties (certificate type, label, subject, issuer and serial number) does not have an expected value

5.2.8 Generate random data tests

Test name: Generate random data test

- **Called function**: `generate_random_data_test`
- **Description**: The test seeds a random data generator and then it generates 64 bytes of random data. In the end, the random data are compared to the array containing 64 `0x00` bytes.
- **Tested Cryptoki functions**: `C_SeedRandom`, `C_GenerateRandom`
- **Fail condition**: The random data generation is supported, but the data are not generated.

5.2.9 Create and delete objects tests

There is no prerequisite for creating object tests. However, in order to destroy object tests there has to exist at least one object, which is being destroyed.

Test name: Create object on a token

6. Distinguished Encoding Rules

- **Called function**: `create_object_test`
- **Description**: A data object (`CKO_DATA` object class) is created on the card with the given attributes template. A returned object handle is a valid handle.
- **Tested Cryptoki functions**: `C_CreateObject`
- **Fail condition**: The test is skipped when the `C_CreateObject` function is not supported. Furthermore, the test fails when the value of the returned object handle to the created data object is `CK_INVALID_HANDLE`.

Test name: Delete an object from a token

- **Called function**: `destroy_object_test`
- **Description**: The test finds an imported certificate object and calls the `C_DestroyObject` function. After the certificate deletion, it finds the same certificate.
- **Tested Cryptoki functions**: `C_DestroyObject`
- **Fail condition**: The test is skipped when the `C_DestroyObject` function is not supported. Additionally, the test also fails when the `C_DestroyObject` returned value is not `CKR_OK` or when the certificate is not deleted from the card.

6 Conclusion

The goal of the master thesis was to create smart cards unit testing applications for the OpenSC project. Foremost, the current situation of smart card usage in Fedora distribution of Linux operating system was mapped. Subsequently, data about smart card usage were collected and analysed and a set of tests was created.

This goal was achieved and the output is documented in the previous chapters. In addition the smart card test application for two selected smart card drivers (PIV driver and driver for Cryptoflex card), was developed. The application was created as a part of test coverage of the OpenSC project. However, it is only part of the forked branch. In order to integrate the application to the master branch, there will be some cooperation with the OpenSC developers.

The main advantage of the developed applications is the possibility for developers to verify that changes in existing functions did not break any functionality.

Although the required functionality was implemented, there are still many ways how to improve the smart card test application. The drawback of the current version is, that only two smart card drivers were tested. In order for the application to cover larger set of drivers, the selected drivers have to be researched as each driver slightly differs from the others.

Another drawback of the application is the necessity to connect smart card reader and inserted card. The solution is the usage of a virtual smart card. One tested possibility was to use the **Virtual Smart Card**[1] project from Frank Morgner and Dominik Oepen. However, the problem with this option was, that the provided virtual Cryptoflex card is incomplete, hence it is not applicable.

The next tested option was to use a JavaCard instead of hardware smart card. The JavaCard is also able to run in virtual environment using the **jCardSim**[2] simulator. However, the drawback of this solution is, that there is no existing suitable and working PIV applet for JavaCards.

---

1. https://frankmorgner.github.io/vsmartcard/virtualsmartcard/README.html
2. https://jcardsim.org/

Bibliography

[1] CardLogix Corporation. *Types of smart cards*. 2010. URL: http://www.smartcardbasics.com/smart_card_images/types-of-smart-cards.gif (visited on 04/12/2016).

[2] Inc. Jacquinot Consulting. *Smart Card Operating System*. 2015. URL: http://www.cardwerk.com/smartcards/smartcard_operatingsystems.aspx (visited on 04/12/2016).

[3] *Smart Cards and their Operating Systems*. Heng Guo HUT, Telecommunications Software and Multimedia Laboratory, May 3, 2015. URL: http://www.tml.tkk.fi/Studies/Tik-111.590/2001s/papers/heng_guo.pdf (visited on 04/12/2016).

[4] RSA Laboratories. *WHAT IS PKCS?* URL: http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs.htm (visited on 04/20/2016).

[5] RSA Laboratories. *PKCS #11: CRYPTOGRAPHIC TOKEN INTERFACE STANDARD*. URL: http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm (visited on 04/20/2016).

[6] Wikipedia. *PKCS #11*. URL: https://en.wikipedia.org/wiki/PKCS_11 (visited on 04/20/2016).

[7] Microsoft. *The Cryptography API, or How to Keep a Secret*. URL: https://msdn.microsoft.com/en-us/library/ms867086.aspx (visited on 04/22/2016).

[8] Microsoft. *Smart Card Minidrivers*. URL: https://msdn.microsoft.com/en-us/library/windows/hardware/dn468773(v=vs.85).aspx (visited on 04/22/2016).

[9] Secgroup Ca' Foscari. *Tookan*. URL: https://secgroup.dais.unive.it/projects/tookan/ (visited on 04/22/2016).

[10] R. Behrends et al. *Attacking and Fixing PKCS#11 Security Tokens*. Tech. rep. Venezia, Italy: Università Ca' Foscari, Oct. 2010, p. 10. URL: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/BCFS-ccs10.pdf (visited on 04/22/2016).

[11] RSA Laboratories. *PKCS #15: CRYPTOGRAPHIC TOKEN INFORMATION FORMAT STANDARD*. URL: http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-15-

cryptographic-token-information-format.htm (visited on 04/20/2016).

[12] Magnus Nyström. *PKCS #15 – A Cryptographic Token Information Format Standard*. Tech. rep. Bedford MA 01730, USA: RSA Laboratories, 1999, p. 9. URL: https://www.usenix.org/legacy/events/smartcard99/full_papers/nystrom/nystrom.pdf (visited on 04/20/2016).

[13] OpenSC. *Using OpenSC*. URL: https://github.com/OpenSC/OpenSC/wiki/Using-OpenSC (visited on 02/12/2016).

[14] Ubuntu documentation. *CommonAccessCard*. URL: https://help.ubuntu.com/community/CommonAccessCard (visited on 02/12/2016).

[15] OpenSC. *OpenSC – tools and libraries for smart cards*. URL: https://github.com/OpenSC/OpenSC/wiki (visited on 02/12/2016).

[16] *PAM Authentication*. URL: http://ubuntuforums.org/showthread.php?t=1557180#2 (visited on 02/12/2016).

[17] Juan Antonio Martinez et al. *PAM-PKCS11 User Manual*. Sept. 2005. URL: https://opensc.github.io/pam_pkcs11/doc/pam_pkcs11.html#mappers (visited on 02/27/2016).

[18] Inc. Eklektix. *Unit testing with mock objects in C*. 2013. URL: https://lwn.net/Articles/558106/ (visited on 02/27/2016).

[19] Andreas Schneider. *cmocka*. 2013. URL: https://cmocka.org/ (visited on 02/27/2016).

[20] RSA Laboratories. *PKCS #11 v2.20: Cryptographic Token Interface Standard*. Tech. rep. June 28, 2004, p. 407. URL: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf (visited on 02/27/2016).

A Pcsc_scan output

The example output of `pcsc_scan`, when `Gemalto Twin Reader` card reader and `Cryptoflex 32k e-gate` smart card are used, looks as follows:

```
$ pcsc_scan
PC/SC device scanner
V 1.4.23 (c) 2001-2011, Ludovic Rousseau
Compiled with PC/SC lite version: 1.8.13
Using reader plug'n play mechanism
Scanning present readers...
0: Gemalto PC Twin Reader 00 00

Sun Feb 21 15:29:43 2016
Reader 0: Gemalto PC Twin Reader 00 00
  Card state: Card removed,

Sun Feb 21 15:29:47 2016
Reader 0: Gemalto PC Twin Reader 00 00
  Card state: Card inserted,
  ATR: 3B 95 18 40 FF 62 04 01 01 05

ATR: 3B 95 18 40 FF 62 04 01 01 05
+ TS = 3B --> Direct Convention
+ T0 = 95, Y(1): 1001, K: 5 (historical bytes)
  TA(1) = 18 --> Fi=372, Di=12, 31 cycles/ETU
  TD(1) = 40 --> Y(i+1) = 0100, Protocol T = 0
-----
  TC(2) = FF
+ Historical bytes: 62 04 01 01 05
  Category indicator byte: 62 (proprietary format)

Possibly identified card:
3B 95 18 40 FF 62 04 01 01 05
Schlumberger CryptoFlex 32Ko V1
```

B Pkcs11-tool utility

Mostly every shown **pkcs11-tool** command is used with arguments `"--module /path/to/pkcs11-spy.so -l --pin 12345"`, so for simplification these arguments are omitted.

- **test**

  $ pkcs11−tool −−test

- **list slots** (no need of logging in)

  $ pkcs11−tool −−list−slots

- **list mechanisms** (no need of logging in)

  $ pkcs11−tool −−list−mechanisms

- **list objects** (without logging in the private objects are not listed)

  $ pkcs11−tool −−list−objects
  $ pkcs11−tool −−list−objects −−login −−pin 12345

- **init pin**

  $ pkcs11−tool −−login −−login−type so −−so−pin
  00000000 −−init−pin −−new−pin 12345

- **change pin**

  $ pkcs11−tool −−change−pin −−new−pin 12345

- **test ec**

  $ pkcs11−tool −−test−ec

- **key pair gen**

  $ pkcs11−tool −−keypairgen −−key−type rsa:2048
  −−label "My generated key"

- **read object**

  $ pkcs11−tool −−read−object −−id a1 −−type pubkey −−output−file /path/to/output.file

- **write object**

  $ pkcs11−tool −−write−object /path/to/cert.der −−type cert
  $ pkcs11−tool −−write−object /path/to/data −−type data −−label "My data" −−application−id 1.10.0.0

- **delete object**

  $ pkcs11−tool −−delete−object −−type pubkey −−id a1

- **sign**

  $ pkcs11−tool −−sign −−id a1 −−input /path/to/message_to_sign −−output /path/to/message_to_sign.signature

- **decrypt**

  $ pkcs11−tool −−decrypt −−id a1 −−input /path/to/message.encrypted −−output /path/to/message.decrypted

- **hash**

  $ pkcs11−tool −−hash −−id a1 −m MD5 −−input /path/to/message_to_hash −−output /path/to/message_to_hash.hash

C Smart card with Mozilla Firefox

The `pkcs15-tool -D` command is used to dump all objects from the smart card. Moreover, it also shows the imported certificate, the private and the public key.

```
Private RSA Key [Imported Certificate]
Object Flags   : [0x3], private, modifiable
Usage          : [0x2E], decrypt, sign,
                         signRecover, unwrap
Access Flags   : [0x0]
ModLength      : 2048
Key ref        : 0 (0x0)
Native         : yes
Path           : 3f0050154b0130000012
Auth ID        : 01
ID             : a1

Public RSA Key [Public Key]
Object Flags   : [0x2], modifiable
Usage          : [0x51], encrypt, wrap, verify
Access Flags   : [0x0]
ModLength      : 2048
Key ref        : 0 (0x0)
Native         : no
Path           : 3f0050154b014800
Auth ID        : 01
ID             : a1

X.509 Certificate [Imported Certificate]
Object Flags   : [0x2], modifiable
Authority      : no
Path           : 3f0050154500
ID             : a1
Encoded serial : 02 09 00B43EDB58A4E758E7
```

D YubiKey OpenSC tests output

The example of `yubico_test.sh.log` log file with failed tests looks as follow:

```
Testing PKCS#11 implementation on Yubico

Card type: PIV
Module for testing is:
    ../src/pkcs11/.libs/opensc-pkcs11.so

[==========] Running 19 test(s).
[ RUN      ] get_all_mechanisms_test
[       OK ] get_all_mechanisms_test
[ RUN      ] is_ec_supported_test
[  SKIPPED ] is_ec_supported_test
[ RUN      ] initialize_token_with_user_pin_test
ERROR: Expected CKR_PIN_INCORRECT was not returned
[  FAILED  ] initialize_token_with_user_pin_test
[ RUN      ] change_user_pin_test
ERROR: User PIN was not correctly changed
[  FAILED  ] change_user_pin_test
[ RUN      ] create_hash_md5_short_message_test
[       OK ] create_hash_md5_short_message_test
[ RUN      ] create_hash_md5_long_message_test
[       OK ] create_hash_md5_long_message_test
[ RUN      ] create_hash_sha1_short_message_test
[       OK ] create_hash_sha1_short_message_test
[ RUN      ] create_hash_sha1_long_message_test
[       OK ] create_hash_sha1_long_message_test
[ RUN      ]
      generate_rsa_key_pair_no_key_generated_test
[  SKIPPED ]
      generate_rsa_key_pair_no_key_generated_test
[ RUN      ] generate_rsa_key_pair_test
[  SKIPPED ] generate_rsa_key_pair_test
[ RUN      ] sign_message_test
[       OK ] sign_message_test
```

```
[ RUN      ] verify_signed_message_test
[  FAILED  ] verify_signed_message_test
[ RUN      ] decrypt_encrypted_message_test
[       OK ] decrypt_encrypted_message_test
[ RUN      ] find_all_objects_test
[       OK ] find_all_objects_test
[ RUN      ] find_object_according_to_template_test
[       OK ] find_object_according_to_template_test
[ RUN      ] find_object_and_read_attributes_test
[       OK ] find_object_and_read_attributes_test
[ RUN      ] generate_random_data_test
Seed method is not supported.
[       OK ] generate_random_data_test
[ RUN      ] create_object_test
Function C_CreateObject is not supported!
[  SKIPPED ] create_object_test
[ RUN      ] destroy_object_test
Function C_DestroyObject is not supported!
[  SKIPPED ] destroy_object_test
[==========] 19 test(s) run.
[  PASSED  ] 11 test(s).
[  FAILED  ] 3 test(s), listed below:
[  FAILED  ] initialize_token_with_user_pin_test
[  FAILED  ] change_user_pin_test
[  FAILED  ] verify_signed_message_test

 3 FAILED TEST(S)
```