SIO Lab
Smartcards and the PTeID

version 1.0

João Paulo Barraca, Hélder Gomes, André Zúquete

Version log:

- 1.0: Initial version

1 Introduction

In this guide you will develop Python programs using the Python PKCS11 module[1] that interacts with PKCS#11 tokens to produce and validate digital signatures.

For that, you need to have installed the following software:

- pcscd: `apt install pcscd`
- pkcs11-tool: `apt install opensc`
- PyKCS11: `apt install python3-pykcs11`
- PTeID Software: available at https://www.autenticacao.gov.pt/cc-aplicacao, check instructions below.
- (Optional) Opensc: `apt install opensc-pkcs11`

2 PTeID software installation

We are going to make to software installation, one to have an application, another to have the necessary libraries.

2.1 PTeID libraries installation

Download the latest Ubuntu distribution (actually, v22.04) and install it as follows:

```
$ sudo dpkg -i ~/Downloads/pteid-mw_ubuntu22_amd64.deb
```

This package as a long list of dependencies, many of which may not be met. If that is the case, run

```
$ sudo apt --fix-broken install
```

The software of interest was installed on directory '/usr/local/lib'. In this directory you can find shared libraries (or shared objects, that's the original name that justifies the '.so' extension).

2.2 Application installation with flatpack

If you are using a test PTeID, jump this section, as this application is of no use for those cards.

First, install the **flatpak package**:

[1] https://github.com/LudovicRousseau/PyKCS11

```
$ sudo apt install flatpak
```

Then configure your flatpack installation with a **runtime repository**:

```
$ sudo flatpak remote-add flathub https://flathub.org/repo/flathub.flatpakrepo
```

Install the **PTeID flatpak package**:

```
$ sudo flatpak install pteid-mw-linux.x86_64.flatpak
```

Finally, you can run the application that allows you to explore the PTeID:

```
$ flatpak run pt.gov.autenticacao
```

Note: this application cannot handle test cards, since these are not legit; they are meant to be used for testing purposes by other applications.

Flatpak packages are installed under directory 'var/lib/flatpak'. The PTeID software is installed under the directory 'pt.gov.autenticacao'.

3 A note on shared libraries

Shared libraries are files that contain executable code, including global variables and functions. This executable code does not run alone, it is used by many applications, possibly at the same time. It is said that applications *load* shared shared libraries, and those libraries complement the functionality of the applications where they are loaded.

3.1 Dynamically linked applications

Most applications use shared libraries. This has three immediate advantages: their binary file is smaller, less RAM is occupied when running several simultaneous applications (less redundancy) and applications are automatically updated when their libraries are updated.

The shared libraries used by a binary can be loaded in different occasions. Applications the were created as binaries that use shared libraries are called *dynamically linked*. You can see that with the command **file**:

```
$ file /bin/bash
```

that will output something like:

```
/bin/bash: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
     /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=0b6b11360e339f231f17484da2c87d0d78554e31, for GNU/Linux 3.2.0, stripped
```

Applications like this start with a bootstrap component called *dynamic linker/loader*, which is **ld.so** . This identifies the dynamic libraries the application needs, where they are in the file system, loads them in memory and links unresolved symbols. An *unresolved symbol* is the name of a variable or function that one binary needs but does not know where it is. *Resolving a symbol* means providing the required location of that variable or function. The set of libraries automatically loaded by an application, on its launching, is called its *shared object dependencies*. They can be observed with the **ldd** command:

```
$ ldd /bin/bash
```

which shows something like this:

```
linux-vdso.so.1 (0x00007fffb4e85000)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007f5f1eb18000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5f1e937000)
/lib64/ld-linux-x86-64.so.2 (0x00007f5f1ec9f000)
```

You can see the dynamic libraries needed (e.g. 'libc.so.6', the C language runtime library, version 6) and their location location in the file system (e.g. '/lib/x86_64-linux-gnu/libc.so.6').

The locations where dynamic loaders look for libraries can be customised by tools and configuration files. But they can also be adjusted by environment variables, and these affect only the applications that have access to those variables.

When applications are executed in Linux, they receive a series or arguments and an environment list. This list contains name-value pairs, which are known as *environment variables*. Environment variables are typically defined in command interpreters (e.g. bash) and sent by command interpreters to every application they launches.

The variable **LD_LIBRARY_PATH** is particularly interesting in this case, because it allows an application to use search directories for shared libraries beyond those configured in the system.

Fortunately, the directory '/usr/local/lib' is among the ones that are used by default to look for shared libraries required by applications.

3.2 Ad hoc loading of shared libraries

Applications can load shared libraries at any time, in order to use their functionality. Loading, in fact, is a two-step operation:

1. Map it in virtual memory (place the dynamic library in a specif, free virtual memory area); and
2. Find symbols in the loaded library for using them (e.g. find a function by its name, get its address, and use it to call the function).

This is what we are going to do to manipulate the PTeID smartcard. We will use a shared library, named **libpteidpkcs11.so**, which implements part of the PKCS#11 interface (there is no obligation to implement the whole of it), and this library is going to be loaded by our application specifically, and not automatically by the operating system.

4 The PTeID libpteidpkcs11 dynamic library

Listing directory '/usr/local/lib' for files with a name starting by 'libpteidpkcs11.so'

```
$ ls -la /usr/local/lib/libpteidpkcs11*
```

you can find several names of shared libraries:

```
lrwxrwxrwx 1 root root     38 Nov 26 04:40 libpteidpkcs11.so -> /usr/local/lib/libpteidpkcs11.so.2.0.0
lrwxrwxrwx 1 root root     38 Nov 26 04:40 libpteidpkcs11.so.2 -> /usr/local/lib/libpteidpkcs11.so.2.0.0
lrwxrwxrwx 1 root root     38 Nov 26 04:40 libpteidpkcs11.so.2.0 -> /usr/local/lib/libpteidpkcs11.so.2.0.0
-rw-r--r-- 1 root root 142880 Sep 16  2022 libpteidpkcs11.so.2.0.0
```

In fact, there is only one shared library, the remaining names are just symbolic links to hide versioning.

4.1 Dependencies

A dynamic library can be itself an binary object that requires other shared objects (that depends on other shared libraries). This is the case of our library of interest:

```
$ ldd libpteidpkcs11.so
```

and these are the expected dependencies:

```
linux-vdso.so.1 (0x00007ffd334ff000)
libpteidcommon.so.2 => /usr/local/lib/libpteidcommon.so.2 (0x00007f1b248e0000)
libpteidcardlayer.so.2 => /usr/local/lib/libpteidcardlayer.so.2 (0x00007f1b248a3000)
libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f1b24600000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f1b24883000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1b2441f000)
libpteiddialogsQT.so.2 => /usr/local/lib/libpteiddialogsQT.so.2 (0x00007f1b2485f000)
libpcsclite.so.1 => /lib/x86_64-linux-gnu/libpcsclite.so.1 (0x00007f1b24852000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f1b24340000)
/lib64/ld-linux-x86-64.so.2 (0x00007f1b24980000)
```

As you can see, we have C and C++ dependencies (libstdc++, libgcc_s, libc), mathematical dependencies (libm) and dependencies from other libraries that are also related with the PTeID token (the ones that start by 'libpteid').

4.2 Functions exported

You can inspect the functions provided, or exported, by this file with the command **nm** (you may have to install the **binutils**:

```
$ nm -D libpteidpkcs11.so | grep " T "
```

The list of functions includes several that are not part of PKCS#11. Just to see the functions of the PKCS#11 interface that the library implements, run the same command the same command with another filter:

```
$ nm -D libpteidpkcs11.so | grep " T " | grep C_
```

4.3 Test of the library with its target smartcard

There are generic tools that can query and manipulate PKCS#11 tokens. One of them is **pkcs11-tool**.

Execute the following command:

```
$ pkcs11-tool --module /usr/local/lib/libpteidpkcs11.so -L
```

You will be warned that there are no *slots* (a term that designates a *PKCS#11 token reader*).

Connect a smartcard reader to your system (do not forget to make it available to the virtual machine) and repeat the command. This time, you see that there is a slot, but it is empty (there is no token on it).

Insert a PTeID card (Cartão de Cidadão) in the reader and check the result. You should get something as follows:

```
token label        : CARTAO DE CIDADAO
token manufacturer : Portuguese Government
token model        : Portuguese eID N
token flags        : PIN pad present, token initialized, PIN initialized
hardware version   : 1.0
firmware version   : 1.0
serial num         : 0000049117895653
pin min/max        : 4/8
```

possibly with a different serial number.

A token, together with its companion PKCS#11 library, can perform cryptographic operations with a set of *PKCS#11 mechanisms*. Use the following command to list the available mechanisms:

```
$ pkcs11-tool --module libpteidpkcs11.so -M
```

The result is something that may look as follows:

```
Supported mechanisms:
  SHA-1, keySize={160,160}, digest
  SHA256, keySize={256,256}, digest
  SHA384, keySize={384,384}, digest
  SHA512, keySize={512,512}, digest
  RSA-PKCS, keySize={1024,3072}, hw, sign
  SHA1-RSA-PKCS, keySize={1024,3072}, hw, sign
  SHA256-RSA-PKCS, keySize={1024,3072}, hw, sign
  SHA384-RSA-PKCS, keySize={1024,3072}, hw, sign
  SHA512-RSA-PKCS, keySize={1024,3072}, hw, sign
  RSA-PKCS-PSS, keySize={1024,3072}, hw, sign
  SHA256-RSA-PKCS-PSS, keySize={1024,3072}, hw, sign
  SHA384-RSA-PKCS-PSS, keySize={1024,3072}, hw, sign
  SHA512-RSA-PKCS-PSS, keySize={1024,3072}, hw, sign
```

You can see that with this token you can compute digests and RSA signatures (both with PKCS and PSS paddings). RSA signatures are a special kind of RSA encryptions with the private key. Note that you cannot decrypt with a private key. This means that you cannot use your own PTeID private keys to decrypt messages encrypted with the corresponding public keys. This limitation was on purpose.

Note: the test PTeID cards are very old, the list of mechanisms should be very different from the one presented above. But it should be similar to the one presented by the actual PTeID cards.

PKCS#11 tokens provide objects, which in our case are private and public RSA keys and X.509v3 certificates. You can list them as follows:

```
$ pkcs11-tool --module libpteidpkcs11.so -O
```

You can see that there are objects that have the same label (e.g. CITIZEN AUTHENTICATION KEY), which are each of the elements of key pair (private and public key). You can see that a key pair and the certificate of its public key are bound by the same ID. These elements are crucial to the right objects a token: first, we get one by type and label; then, we get a related one by type and ID, using the ID of the first.

5 Using the PKCS#11 standard to manage the PTeID

The PKCS#11 standard (Cryptographic Token Interface Standard) is produced by RSA Security and defines native programming interfaces to cryptographic tokens, such as hardware cryptographic accelerators and smartcards, as is the case of Portuguese PTeID, known as Cartão de Cidadão (Citizen Card).

Python PyKCS11 includes a provider that, in contrast to most other providers, does not implement cryptographic algorithms itself. Instead, it exposes functions that are executed in the hardware tokens, and allows to obtain the objects stored in a smartcard. When using the Python Cryptography module, this is the role of the Backends (providing access to hardware tokens). Unfortunately, no public (and stable, and ubiquitous) backend supports PKCS#11, making it required to use a standalone module (PyKCS#11).

5.1 Loading the card interface module

Interacting with the PTeID requires the use of a module that translated PKCS#11 calls into some internal format adequate to the smartcard hardware. This is independent of the programming language used, and it is why we need to install the PTeID software: besides a visualization and exploitation tool, it also contains libraries that allow accessing the PTeID (namely, 'libpteidpkcs11.so')

Using the PyKCS11 module, the following snippet will try to list all slots, and present information about the tokens contained:

```
import PyKCS11
import binascii

lib = 'libpteidpkcs11.so'
pkcs11 = PyKCS11.PyKCS11Lib()
```

```
pkcs11.load( lib )
slots = pkcs11.getSlotList()

for slot in slots:
    print( pkcs11.getTokenInfo( slot ) )
```

Task: List all tokens and print the details of each token (revision, manufacturer, model, serial, etc...).

5.2 Contents of the PTeID

After the token is available, it is possible to list the objects it contains. Some objects have public visibility and are accessed through a public session, while others are private, and the user must explicitly create a private session. The different between sessions lies in the existence of omission of a login with a PIN code when opening the session.

You can list the contents of the PTeID (i.e., its PKCS#11 objects), with code based on the following snippet:

```
# Attributes are key-value pairs, get all the possible keys
all_attr = list( PyKCS11.CKA.keys() )
#Filter attributes
all_attr = [e for e in all_attr if isinstance( e, int )]

session = pkcs11.openSession( slot )
for obj in session.findObjects():
    # Get object attributes
    attr = session.getAttributeValue( obj, all_attr )
    # Create dictionary with attributes
    attr = dict( zip( map( PyKCS11.CKA.get, all_attr ), attr ) )
    # Print the object label
    print( 'Label: %s, Class: %d' % (attr['CKA_LABEL'], attr['CKA_CLASS']) )

print( 'Class: %d means private key, %d means public key, %d means certificate'
       % (PyKCS11.CKO_PRIVATE_KEY, PyKCS11.CKO_PUBLIC_KEY, PyKCS11.CKO_CERTIFICATE) )
```

Note: The prefix 'CKA_' is used to refer to names of object attributes (e.g. 'CKA_CLASS' is the class of an object), whilst 'CKO_' is used to refer to a predefined value of an object attribute (e.g. 'CKO_CERTIFICATE' is the class value of a certificate object).

The information you get in the `CKA_LABEL` attribute is the human-understandable name of the object included in the PTeID. Through these identifiers we can select which certificate, or private key, we intend to use for each cryptographic operation. The attribute `CKA_CLASS` identifies the class of the object.

You can individually inspect all attributes of the objects. If you require access to the certificate content, convert the tuple contained in the `CKA_VALUE` to bytes (`bytes(attributes['CKA_VALUE'])`) and load it as a DER certificate (`x509.load_der_x509_certificate`).

Task: Print the label of all objects, as well as the issuer and subject of all certificates.

Hint: Once loaded as a X.509 certificate, you can reuse the code from the last class to print the issuer and subject.

5.3 Digital signature

Task: Create a program capable of generating a digital signature of a document, using the PTeID, and storing it on a given file. Furthermore, complement it for writing on a file the public key certificate corresponding to the private key used for signing the document.

For this purpose consider the CITIZEN AUTHENTICATION CERTIFICATE and the corresponding CITIZEN AUTHENTICATION KEY (the private key), as many citizens to not have the CITIZEN SIGNATURE CERTIFICATE activated.

Because the private key is not extractable (check the 'CKA_EXTRACTABLE' attributes), you cannot load this key as an RSA key to your Python program. Instead, use the session object to sign the text:

```python
# The function returns a list, that's why we need the [0] at the end
private_key = session.findObjects( [ (PyKCS11.CKA_CLASS, PyKCS11.CKO_PRIVATE_KEY),
                                     (PyKCS11.CKA_LABEL, 'CITIZEN AUTHENTICATION KEY') ] )[0]

mechanism = PyKCS11.Mechanism( PyKCS11.CKM_SHA1_RSA_PKCS, None )

text = b'text to sign'
signature = bytes( session.sign( private_key, text, mechanism) )
```

Alternatively, you can use a signature method without referring to a hashing function (e.g. CKM_RSA_PKCS) and provide the sign function with a computed digest instead of the original data to sign. The sign function will infer the hashing function used from the number of bytes provided in the digest. This inference is required to properly pad the hash (internally) with an ASN.1 header referring the OID (Object Identifier) of the hashing function used.

```python
import hashlib

mechanism = PyKCS11.Mechanism( PyKCS11.CKM_RSA_PKCS, None )

h = hashlib.sha1()
h.update( text.encode( 'UTF-8') )
signature = bytes( session.sign( private_key, h.digest(), mechanism) )
```

This approach is interesting when is preferable for you to compute the message digest, instead of leaving that to the PKCS#11 library function.

5.4 Signature validation

Task: Create a program capable of checking a digital signature on a document. The program should use as inputs a file with the digital signature, a file with the signer's public key certificate and a file with the signed contents. You should also display the identity of the entity that produced the signature (the subject). This is important as the user should know the identity of the user that created the signature.

Hint: Validating the signature does not use the PTeID and reuses the validation processes from the last laboratory guide. To verify the signature, use `padding.PKCS1v15()` as padding and `hashes.SHA1()` as the hash mechanism. Do not forget to validate the certification chain, dates and purposes.

6 Bibliography

- Welcome to pyca/cryptography
- PyKCS11 - PKCS#11 Wrapper for Python
- PyKCS11 1.5.12 documentation