Cybersecurity MSc: Applied Cryptography 2023-24

Practical Exercises:
Digital Signatures with the Portuguese Citizen Card

September 14, 2023 Due date: no date

Changelog

- v1.0 - Initial version.

1 Introduction

In order to elaborate this laboratory guide it is required to install the Java Development Environment (JDK), Python 3 or the C compiler and development environment.

The examples provided will use both Java, Python and C. For Python you need to install the `cryptography` and the `pykcs11` modules. For C you need to install the packages `libopencryptoki0` and `libopencryptoki-dev`.

For practicing with digital signatures we will make use of the capabilities of the Portuguese Citizen Card (Cartão de Cidadão, CC). The middleware required to use the CC is distributed together with a support application and is available here.

The interaction between applications and smartcards is mediated by a system daemon. In Windows systems it is usually running by default. In Linux system it is called `pcscd` and it may not be present or running, so you may need to install it.

2 Relevant CC middleware

The CC middleware that is relevant for this work is composed by two main components:

- A PKCS #11 library.

  In Linux systems, this file has as base name `libpteidpkcs11` and is located in `/usr/local/lib`. There is only a dynamic version of this library (extension `.so`).

  In MacOS systems, the path should be the same, but the file extension of the shared library is `.dylib`.

  In Windows systems, this file has the name `pteidpkcs11.dll` and is located in the subdirectory `System32` of the Windows root directory (usually `C:\Windows`);

- A set of public key certificates that help to create the topmost certification chain from the certificates contained in a CC up to a particular root CA (MULTICERT Root CertPathValidator Authority 01);

  In Linux and MacOS ,these files are stored in `/usr/local/share/certs`.

  In Windows system, these files are stored in `eidstore\certs` under the directory `C:\Programa Files\Portugal Identity Card\eidstore\certs`.

- Several root certification authorities:
  - Baltimore CyberTrust Root (Linux only);
  - COMODO RSA Certification Authority (Linux only);
  - Global Chambersign Root - 2008 (Linux only); and
  - MULTICERT Root Certification Authority 01.
  - EC Raiz do Estado 002 (for certificates issued after July 10, 2020).

  The location of these files is the same of the other certificates above referred.

3 Exploitation of a PKCS #11 device

3.1 With Java

There are basically two ways to explore a PKCS #11 device from a Java program:

1. Using the native Sun PKCS #11 provider functionalities of Java. We will explore this one in this guide. With this provider, we can abstract the fact of using a CC as a PKCS #11 token, we can use its objects (keys) as any other keys that may exist in the program memory (accept having access to the value of private keys).
2. Using a PKCS #11 wrapper package (e.g. the IAIK wrapper). In this case we can explore the PKCS #11 interface to explore and interact with the CC.

A PKCS #11 provider is a piece of software that provides cryptographic functionalities through a PKCS #11 API. This is a standard API that was designed to normalize the access to cryptographic functionalities provided by hardware devices (or tokens) to applications. For each device, one should have a PKCS #11 library with a subset of the relevant API functions for the device.

The definition of a PKCS #11 provider follows the general rules of the definition of another security provider: it has to be added to the list of security providers recognized by the JVM. This can be done statically, through JVM configuration files, or programmatically by an application. To make it programmatically, one has to associate a configuration file for the provider to the ones already used by providers of the same type. In our case, since we want to explore a PKCS #11 provider, if `CitizenCard.cfg` is the name of the configuration file for exploring a CC, we would do (in Java 9 and beyond):

```
import java.security.Provider;
import java.security.Security;

Provider p = Security.getProvider( "SunPKCS11" );
p = p.configure( "CitizenCard.cfg" );
Security.addProvider( p );
```

For Java versions below 9, the code is slightly different:

```
import java.security.Provider;
import java.security.Security;

Provider p = new sun.security.pkcs11.SunPKCS11( "CitizenCard.cfg" );
Security.addProvider(p);
```

The configuration file has to contain enough information to allow the Sun PKCS #11 provider to explore another provider using the same API. This information is provided in textual `key = value` pairs, where the keys `name` and `library` are mandatory (see section 2.2 of this Web page for other keys that can be used) with Java 8.

The configuration presented below associates the name `PTeID` (an arbitrary name) to the path of the CC's PKCS #11 shared library path. The name is useful to refer to this particular PKCS #11 provider (there may be many). The library path will loaded and used by the JVM to access the CC through a PKCS #11 API.

```
name = PTeID
library = /usr/local/lib/libpteidpkcs11.so
```

Note: on MacOS X systems, use the extension `.dylib` instead of `.so`.
Note: on Windows systems use `\\` as path separator. The library path should be `c:\\Windows\\System32\\pteidpkcs11.dll`.

Since cryptographic operations require some kind of key, in Java a PKCS #11 provider is first of all viewed as a java.security.**KeyStore**, i.e., as a piece of software capable of providing cryptographic keys. However, this is a special case of KeyStore where its contents are not copied into memory (and this is why the parameters of the `load` method below are both null:

```
import java.security.KeyStore;

KeyStore ks = KeyStore.getInstance( "PKCS11", "SunPKCS11-PTeID" );
ks.load( null, null );
```

Another option is to use the class java.security.**KeyStore.Builder**. This is almost equivalent to the former but allows the application to handle callbacks from the PKCS #11 provider (e.g. for asking for a PIN to authorize a particular signing operation).

```
import java.security.KeyStore;
import java.security.KeyStore.*;

KeyStore.CallbackHandlerProtection func =
    new KeyStore.CallbackHandlerProtection( new MyCallbackHandler() );

KeyStore.Builder builder = KeyStore.Builder.newInstance( "PKCS11", "SunPKCS11-PTeID", func );
KeyStore ks = builder.getKeyStore();
```

3.2 With Python

With Python, and using `pykcs11`, we create an object to access a PKCS #11 token using a particular middleware library as shown below (this example lists the available slots, were a slot is a logical reader that potentially contains a token):

```
from PyKCS11 import *
from PyKCS11.LowLevel import *

lib = ’/usr/local/lib/libpteidpkcs11.so’

pkcs11 = PyKCS11.PyKCS11Lib()
pkcs11.load( lib )
slots = pkcs11.getSlotList()

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

3.3 With C

In C with need to use the standard PKCS #11 interface provided by the`cryptoki` package and the dynamic objects' handling library to load and resolve symbols of the PKCS #11 library.

```c
#include <opencryptoki/apiclient.h>

void * handle;
CK_RV (*func_list)( CK_FUNCTION_LIST_PTR_PTR );
CK_FUNCTION_LIST_PTR pkcs11_funcs;
CK_RV ret;
CK_ULONG slots;
CK_SLOT_ID * slotIds, slot;
CK_SLOT_INFO slotInfo;
CK_TOKEN_INFO tokenInfo;

handle = dlopen( library_path, RTLD_LAZY);
if (handle == 0) {
    // error
}

func_list = dlsym( handle,  "C_GetFunctionList" );
if (func_list == 0) {
    // error
}

ret = func_list( &pkcs11_funcs );
if (ret != CKR_OK) {
    // error
}

ret = pkcs11_funcs->C_Initialize ( 0 );
if (ret != CKR_OK) {
    // error
}

slots = 0;
ret = pkcs11_funcs->C_GetSlotList ( FALSE, 0, &slots );
if (ret != CKR_OK) {
    // error
}

slotIds = (CK_SLOT_ID *) alloca ( slots * sizeof(CK_SLOT_ID) );
ret = pkcs11_funcs->C_GetSlotList ( FALSE, slotIds, &slots );
if (ret != CKR_OK) {
    // error
}

for (int i = 0; i < slots; i++) {
    ret = pkcs11_funcs->C_GetSlotInfo ( slotIds[i], &slotInfo );
    if (ret != CKR_OK) {
        // error
    }
    if (slotInfo.flags & CKF_TOKEN_PRESENT) {
        ret = pkcs11_funcs->C_GetTokenInfo ( slotIds[i], &tokenInfo );
        if (ret != CKR_OK) {
            // error
        }
        printf( "Found token %s\n", tokenInfo.label );
    }
}

pkcs11_funcs->C_Finalize ( 0 );
```

Alternatively, you can use directly the PKCS #11 functions but you have to link the application directly with the target PKCS #11 library.

4 PKCS #11 objects

A PKCS #11 provider (usually implemented by a library that knows how to manage a given type of crypto tokens) manages a set of objects, usually keys and certificates.

With the following Java code we can list the names of all the objects inside a CC (note that objects of different type can have the same name, but only one is displayed):

```
import java.util.Enumeration;

Enumeration<String> aliases = ks.aliases();
while (aliases.hasMoreElements()) {
    System.out.println( aliases.nextElement() );
}
```

For making signatures with the CC the objects of interest are the two private keys with the following names:

- CITIZEN AUTHENTICATION CERTIFICATE
- CITIZEN SIGNATURE CERTIFICATE

For validating those signatures, the objects of interest are the two public key certificates with exactly the same names.

The following code does the same in Python:

```
from  PyKCS11 import *
from PyKCS11.LowLevel import *

lib = '/usr/local/lib/libpteidpkcs11.so'

pkcs11 = PyKCS11.PyKCS11Lib()
pkcs11.load( lib )
slots = pkcs11.getSlotList()

classes = {
    CKO_PRIVATE_KEY : 'private key',
    CKO_PUBLIC_KEY : 'public key',
    CKO_CERTIFICATE : 'certificate'
}

for slot in slots:
    if 'CARTAO DE CIDADAO' in pkcs11.getTokenInfo( slot ).label:
        session = pkcs11.openSession( slot )
        objects = session.findObjects()
        for obj in objects:
           l = session.getAttributeValue( obj, [CKA_LABEL] )[0]
           c = session.getAttributeValue( obj, [CKA_CLASS] )[0]
           print( 'Object with label ' + l + ', of class ' + classes[c] )
```

The following code does the same in C:

```
#include <opencryptoki/apiclient.h>

void * handle;
CK_RV (*func_list)( CK_FUNCTION_LIST_PTR_PTR );
CK_FUNCTION_LIST_PTR pkcs11_funcs;
CK_RV ret;
CK_ULONG slots;
CK_SLOT_ID * slotIds, slot;
CK_SLOT_INFO slotInfo;
CK_TOKEN_INFO tokenInfo;

// Load and initialize library
```

```
slots = 0;
ret = pkcs11_funcs->C_GetSlotList ( FALSE, 0, &slots );
if (ret != CKR_OK) {
    // error
}

slotIds = (CK_SLOT_ID *) alloca ( slots * sizeof(CK_SLOT_ID) );
ret = pkcs11_funcs->C_GetSlotList ( FALSE, slotIds, &slots );
if (ret != CKR_OK) {
    // error
}

for (int i = 0; i < slots; i++) {
    ret = pkcs11_funcs->C_GetSlotInfo ( slotIds[i], &slotInfo );
    if (ret != CKR_OK) {
        // error
    }
    if (slotInfo.flags & CKF_TOKEN_PRESENT) {
        ret = pkcs11_funcs->C_GetTokenInfo ( slotIds[i], &tokenInfo );
        if (ret != CKR_OK) {
            // error
        }
        printf( "Found token %s\n", tokenInfo.label );

        if (strncmp( tokenInfo.label, "CARTAO DE CIDADAO", 17 ) == 0) {
            CK_SESSION_HANDLE session;

            ret = pkcs11_funcs->C_OpenSession( slotIds[i], CKF_SERIAL_SESSION, 0, 0, &session
                );
            if (ret != CKR_OK) {
                // error
            }

            pkcs11_funcs->C_FindObjectsInit( session, 0, 0 );
            for (;;) {
                unsigned long nr_objects;
                CK_OBJECT_HANDLE object;
                ret = pkcs11_funcs->C_FindObjects( session, &object, 1, &nr_objects );
                if (ret != CKR_OK) {
                   // error
                }
                if (nr_objects == 0) break;

                char label[1024];
                CK_OBJECT_CLASS class;
                CK_ATTRIBUTE attributes[] = {
                    {CKA_LABEL, label, sizeof(label) -1},
                    {CKA_CLASS, &class, sizeof(class)}
                };
                pkcs11_funcs->C_GetAttributeValue( session, object, attributes, 2 );
                if (class == CKO_PRIVATE_KEY) {
                    printf( "Private key, label %s\n", label );
                }
                else if (class == CKO_PUBLIC_KEY) {
                    printf( "Public key, label %s\n", label );
                }
                else if (class == CKO_CERTIFICATE) {
                    printf( "Certificate, label %s\n", label );
                }
                else {
                    printf( "Object of class %ld, label %s\n", class, label );
                }
            }
            pkcs11_funcs->C_FindObjectsFinal( session );
            pkcs11_funcs->C_CloseSession( session );
        }
    }
}

pkcs11_funcs->C_Finalize ( 0 );
```

5 Digital signatures with the CC

Use the class java.security.**Signature** to create a digital signature of a data buffer using the methods `initSign`, `update` and `sign`. The CC supports `SHA1withRSA` and `SHA256withRSA` signatures (this last one may not be available to older cards).

Try the signatures with both the private keys of the CC. These can be “obtained” with the java.security.**KeyStore** method `getKey`. The PIN will be asked through a graphical interface.

Verify the signature with the same class, but now using the methods `verifyInit`, `update` and `verify`. The certificates required for the validation can be obtained with the java.security.**KeyStore** method `getCertificate`.

With Python you can generate a signature using the authentication private key of the Citizen Card as follows:

```
from PyKCS11 import *
from PyKCS11.LowLevel import *

lib = ’/usr/local/lib/libpteidpkcs11.so’

pkcs11 = PyKCS11.PyKCS11Lib()
pkcs11.load( lib )
slots = pkcs11.getSlotList()

for slot in slots:
    if ’CARTAO DE CIDADAO’ in pkcs11.getTokenInfo( slot ).label:
        data = bytes( ’data to be signed’, ’utf-8’ )

        session = pkcs11.openSession( slot )

        privKey = session.findObjects( [(CKA_CLASS, CKO_PRIVATE_KEY),
                                       ( CKA_LABEL, ’CITIZEN AUTHENTICATION KEY’ )] )[0]
        signature = bytes(session.sign( privKey, data, Mechanism( CKM_SHA1_RSA_PKCS ) ))

        session.closeSession
```

For validating the signature we do it differently, since we do not need to use a PKCS #11 provider, since this is an operation that does not need to be performed by a crypto token.

The following code completes the one presented before to verify the signature generated.

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.serialization import load_der_public_key
from cryptography.hazmat.primitives.asymmetric import (
    padding, rsa, utils
)

session = pkcs11.openSession( slot )

pubKeyHandle = session.findObjects([(CKA_CLASS, CKO_PUBLIC_KEY),(CKA_LABEL, ’CITIZEN
    AUTHENTICATION KEY’)])[0]
pubKeyDer = session.getAttributeValue( pubKeyHandle, [CKA_VALUE], True )[0]

session.closeSession

pubKey = load_der_public_key( bytes( pubKeyDer ), default_backend() )

try :
    pubKey.verify( signature, data, padding.PKCS1v15(), hashes.SHA1() )
    print( ’Verification succeeded’ )
except:
    print( ’Verification failed’ )
```

6 Validation of certification chains

Usually digital signatures are accompanied by a set of certificates. This set includes two different types of certificates, all of them part of a single certification chain:

- The personal certificate of the signer (usually a person);
- The certificate of the intermediate Certification Authority the issued the personal certificate, and the successive Certification Authorities thereafter until the last one, immediately under a root Certification Authority.

The validation of a certification chain must be performed for a particular date, namely the date where a particular operation (e.g. a signature) was performed with the private key corresponding to the (personal) certified public key. The validation can also use downloaded CRL lists or query OCSP services to check possible certificate revocations at the referred date.

To validate a certification path we need first to define it. A path is no more than a set of certificates, where one certifies the public key that signs the other, from a trusted root (trust anchor) until the certificate that we want ultimately to validate.

For a signature performed with a CC, and in the absence of any certification chain provided along with the signature, we can use the following process:

- Fetch all certificates used by CC's intermediate CAs (check this Web page) and build a keystore with them for facilitating their usage. Add to the keystore the certificates distributed with the CC middleware (there may be some repeated ones). This Makefile shows how this can be done.
- From a Java program, open the keystore, and go through all the certificates, separating them in two Java java.util.**Collection**s: one for the trusted anchors (formed only by self-certified certificates), another for intermediate certificates. Self-certified certificates can be easily detected because their signature can be validated with their own public key.

```
import java.security.PublicKey;

PublicKey key = cert.getPublicKey();
cert.verify( key );
```

- Define a set of parameters (with a java.security.cert.**PKIXBuilderParameters** object) for guiding the search of a certification path from a target certificate until some anchor. These parameters should include:
  - a selector, which is a java.security.cert.**X509CertSelector** object with the certificate to validate;
  - a java.util.**Set** of acceptable anchors (a subset of all known self-certified certificates);
  - rules to validate or not revocations of certificates; and
  - a java.security.cert.**CertStore** containing all known intermediate certificates.

```
X509CertSelector selector = new X509CertSelector();
selector.setCertificate( cert );
PKIXBuilderParameters pkixParams = new PKIXBuilderParameters( anchors, selector );
pkixParams.setRevocationEnabled(false); // No CRL checking
pkixParams.addCertStore( intermediateCertStore );
```

- One having all these parameters set, we create a PKIX java.security.cert.**CertPathBuilder** object and we build a java.security.cert.**PKIXCertPathBuilderResult** using those parameters.

```
CertPathBuilder builder = CertPathBuilder.getInstance( "PKIX" );
PKIXCertPathBuilderResult path = (PKIXCertPathBuilderResult) builder.build( pkixParams );
```

Once having a certification path to validate, we can use the PKIX (Public Key Infrastructure for X.509 certificates) certification validation rules as follows. First we create a java.security.cert.**CertPathValidator** for implementing a PKIX validation policy. Then we define the parameters for the validation (e.g. date), and we ask for a validation.

```
CertPathValidator cpv = CertPathValidator.getInstance( "PKIX" );
PKIXParameters validationParams = new PKIXParameters( anchors );
validationParams.setRevocationEnabled( true );
validationParams.setDate( date );
cpv.validate( path.getCertPath(), validationParams );
```

As an exercise, validate the certification chain for both certificates in a CC using different dates (e.g. one before their issuing, another with the current date, and yet another after their expiration date). Check also what happens when CRL validation is used.

The following Python program uses the `pyOpenSSL` module to validate certificate chains. The validation uses a `X509Store` object to keep all the elements that are key to the validation: trusted root certificates, validation date and CRLs. The validation is performed by an object of class `X509StoreContext`, which is parameterized with the certificate to validate and a list of potentially useful intermediate certificates. The validation is performed in two steps:

- In the first one we do not consider any revocation information. We do this because we do not have any cached CRL information (we could do otherwise if we had it). Therefore, we will mainly check if we can build a chain up to a trusted root certificate and if the date provided falls in the validity period of all the chain certificates.
- In the second one, we use the validation chain previously identified and we load the base and delta CRLs identified in its certificates. Then, we repeat the certificate validation, but now requesting CRL validations.

To simplify, the program can read a single PEM file with all the certificates that may be used to validate a given certificate. Trusted root certificates are automatically identified as the ones that are self-certified.

```
#!/usr/bin/python3

import os
import sys
import OpenSSL.crypto as osc
import cryptography.x509 as x509
from datetime import datetime
from asn1crypto import pem
import requests

def load_from_url( url ):
    response = requests.get( url )
    if response.status_code == 200:
        return response.content

    return None

def validate_ocsp( cert ):
    print( '\tOCSP validation not yet implemented' )

def add_crl( root, cert, base=False, delta=False ):
    c = x509.load_der_x509_certificate( osc.dump_certificate( osc.FILETYPE_ASN1, cert ) )
    if base:
        cdp = c.extensions.get_extension_for_oid(
            x509.oid.ExtensionOID.CRL_DISTRIBUTION_POINTS )
    elif delta:
        cdp = c.extensions.get_extension_for_oid( x509.oid.ExtensionOID.FRESHEST_CRL )
    else:
        return

    for dpoint in cdp.value:
        for url in dpoint.full_name:
            print( ’\t%s’% (url.value) )
            crl = osc.load_crl( osc.FILETYPE_ASN1, load_from_url ( url.value ) )
            root.add_crl( crl )

def main():
    if len(sys.argv) < 6:
        print( ’Usage: %s certificate year month day chain_certificate [chain_certificate]\n’
            % sys.argv[0], file=sys.stderr )
        sys.exit( 1 )

    # Create validation context (a X509Store object)

    root = osc.X509Store()

    # Set validation flags

    root.set_flags( osc.X509StoreFlags.X509_STRICT )

    # Set validation date

    d = datetime(year=int(sys.argv[2]), month=int(sys.argv[3]), day=int(sys.argv[4]) )
    root.set_time( d )

    intermediate = []

    # Load certificates into validation context

    for i in range(5, len(sys.argv)):
        with open( sys.argv[i], ’rb’ ) as cf:
            cf_bytes = cf.read()

            if pem.detect(cf_bytes):
                for _, _, der_bytes in pem.unarmor( cf_bytes, multiple=True ):
                    cert = osc.load_certificate( osc.FILETYPE_ASN1, der_bytes )
                    if cert.get_subject() == cert.get_issuer():
                        root.add_cert( cert )
                    else:
                        intermediate.append( cert )
            else:
                cert = osc.load_certificate( osc.FILETYPE_ASN1, der_bytes )
                if cert.subject == cert.issuer:
                    root.add_cert( cert )
                else:
                    intermediate.append( cert )

    # Load certificate to validate

    with open( sys.argv[1], ’rb’ ) as cf:
        cf_bytes = cf.read()

        if pem.detect(cf_bytes):
            cert = osc.load_certificate( osc.FILETYPE_PEM, cf_bytes )
        else:
            cert = osc.load_certificate( osc.FILETYPE_ASN1, cf_bytes )

    validator = osc.X509StoreContext( root, cert, intermediate )

    try:
        validator.verify_certificate()

        # Chain was build, let’s do OCSP/CRL some checking

        print( ’chain:’ )
        chain = validator.get_verified_chain()

        # See what we have in the chain certificates

        for cert in chain:
            # print CN field of certificate’s subject

            for component in cert.get_subject().get_components():
                if component[0] == b'CN':
                    print( 'CN: %s' % (str(component[1], 'utf8')) )

            # Look in extension for OCSP/CRL information

            for i in range(0, cert.get_extension_count()):
                # print( '\textension: %s' % (cert.get_extension( i ).get_short_name()) )
                if cert.get_extension( i ).get_short_name() == b'authorityInfoAccess':
                    validate_ocsp( cert )
                elif cert.get_extension( i ).get_short_name() == b'crlDistributionPoints':
                    add_crl( root, cert, base=True )
                elif cert.get_extension( i ).get_short_name() == b'freshestCRL':
                    add_crl( root, cert, delta=True )

        # Set validation flags to include CRL validations

        root.set_flags( osc.X509StoreFlags.CRL_CHECK_ALL )

        validator.verify_certificate()
        print( 'YES' )

    except Exception as e:
        print( 'NO: %s' % (e) )


if __name__ == '__main__':
    main()
```

References

- Portuguese Citizen Card web site: https://www.cartaodecidadao.pt
- Java PKCS #11 Reference Guide: https://docs.oracle.com/javase/9/security/pkcs11-reference-guide1.htm#JSSEC-GUID-30E98B63-4910-40A1-A6DD-663EAF466991
- Welcome to pyca/cryptography: https://cryptography.io/en/latest
- PyKCS11 1.5.2 documentation: https://pkcs11wrap.sourceforge.io/api/index.html
- Welcome to pyOpenSSL's documentation: https://pypi.org/project/pyOpenSSL