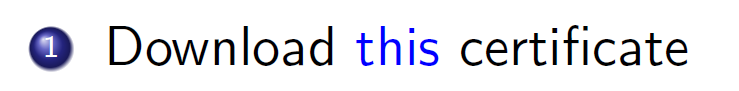
Premises

Lab Activity – Liviu-Ioan ZECHERU

All assignments will be structured as snapshot of the assignment then snapshot of my result.

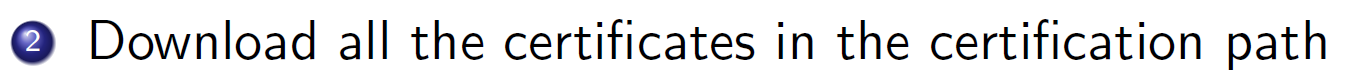
Certificates



(link behind - <https://drive.google.com/file/d/16KXExKkzw4b35OedxrJ7hAA6YTLbQk72/view?usp=share_link>)

O imagine care conține text, captură de ecran, software, Pictogramă computer

Descriere generată automat



O imagine care conține text, captură de ecran, software, afișaj

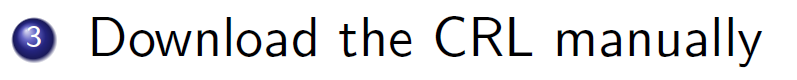
Descriere generată automat

The certificates in the certification path are: certSIGN Qualified CA and certSIGN ROOT CA G2. Here they are:

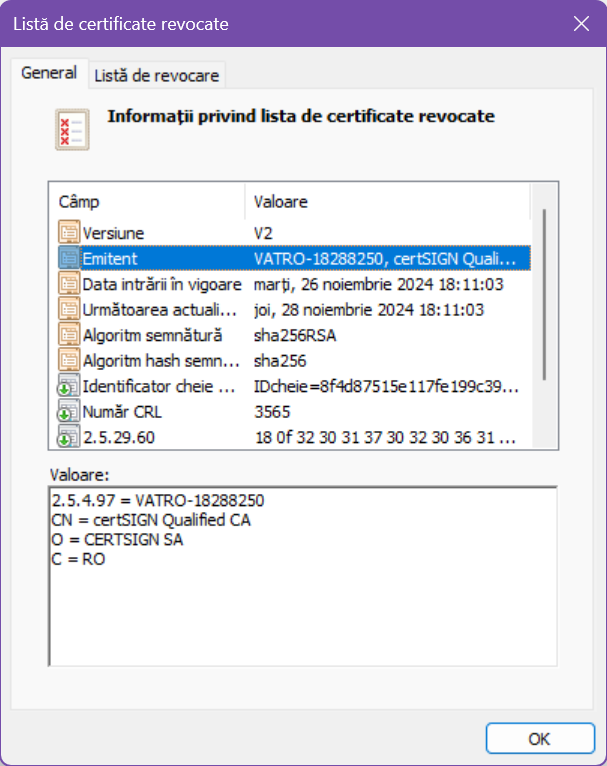
O imagine care conține text, captură de ecran, software, Pictogramă computer

Descriere generată automat O imagine care conține text, captură de ecran, software, număr

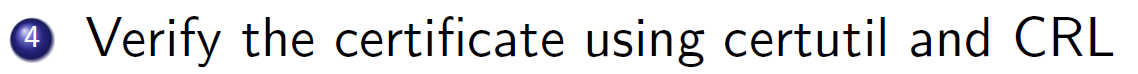
Descriere generată automat



According to the details of the certificate, the CRL is located here: <http://crl.certsign.ro/certsign-qualifiedca.crl>. Accessing the link will give us this file:

 O imagine care conține text, captură de ecran, software, număr

Descriere generată automat



We open a terminal in the path of the certificate and use the command **certutil -url cert.cer** (because that’s the name of the certificate). A window like this will open:

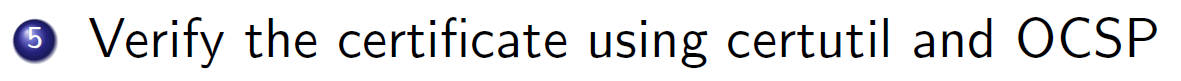
O imagine care conține text, captură de ecran, software, număr

Descriere generată automat

To verify the certificate using the CRL we select the **CRLs (from CDP)** as retrieval method then click on **Retrieve** button. Then we can see the status in the view area.

O imagine care conține text, captură de ecran, software, număr

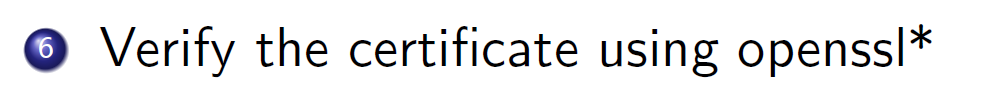
Descriere generată automat



Similarly, we select the **OCSP (from AIA)** as retrieval method then click on **Retrieve** button. Then we can see the status in the view area.

O imagine care conține text, captură de ecran, număr, software

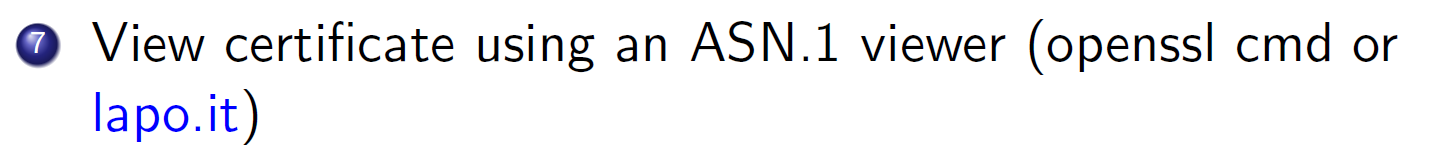
Descriere generată automat



After we install the **OpenSSL** package, we open up a terminal in the path of the certificate and use the given command: **openssl ocsp -issuer ca\_cert.cer -cert cert.cer -no\_nonce -url ”ocsp url” -CAfile root.cer** adapted to our naming. In my case it will be **openssl ocsp -issuer ".\certSIGN Qualified CA.cer" -cert cert.cer -no\_nonce -url "http://ocsp.certsign.ro" -CAfile ".\certSIGN ROOT CA G2.cer"**. Here is the result:

O imagine care conține electronice, text, captură de ecran, calculator

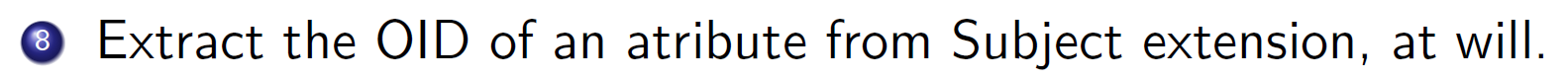
Descriere generată automat



I choose to view it using the **OpenSSL** package directly in the command prompt. The command I use is (from the location of the certificate): **openssl asn1parse -inform DER -in .\cert.cer**. The resulted output is:

O imagine care conține text, captură de ecran

Descriere generată automat



The Subject field in an X.509 certificate contains a series of attributes such as:

Common Name (CN)

Organization (O)

Organizational Unit (OU)

Country (C)

State or Province Name (ST)

Locality Name (L)

Each attribute is identified by an OID (e.g., CN = 2.5.4.3). Let’s try to extract the OID of the Country. We first need to get the offset of the zone where **OpenSSL** interprets the raw binary data as **countryName** by its internal mapping. We use the command **openssl asn1parse -inform DER -in .\cert.cer | Select-String ":countryName" -Context 1,0**. The first part displays the previous output and the command after the pipe searches for the attribute we need using the PowerShell cmdlet Select-String. The result is the following:

O imagine care conține electronice, text, captură de ecran, software

Descriere generată automat

Now we see that it appears in two places. We can analyze any of those two. Let’s take, for instance, the second one. We will dump the binary data from the offset 173, as that is the starting point, using the command **openssl asn1parse -inform DER -in .\cert.cer -strparse 173 -out oid\_raw.bin**. Now we analyze the bytes in a hex viewer like HxD:

O imagine care conține text, captură de ecran, software, afișaj

Descriere generată automat

We know that OID for the countryName is **2.5.4.6**. Considering the ASN1 encoding, the first byte encodes the first two components by the formula 40 \* X + Y. So, in our case it will be 40 \* **2** + **5** = 85 which is **55** in hexadecimal format. The subsequent components are encoded as a series of variable-length integers. Each integer is encoded using a base-128 system, where the high bit indicates whether the encoding continues to the next byte. We see that we have the sequence **55 04 06**, so we found our OID for the country name attribute!

O imagine care conține text, Font, captură de ecran, alb

Descriere generată automat

I did this part in the live lab at our first meeting. I will just provide the proof of the pages I had to go through and the imported certificate into the Windows Store.

Here is the end entity creation page where, me, as CA, had to issue a new certificate for an end user:

O imagine care conține text, captură de ecran, software, Pictogramă computer

Descriere generată automat

Here, as the end user, on the public page, I logged on with the created credentials and generated my PKCS#12 file.

O imagine care conține text, captură de ecran, software, Pictogramă computer

Descriere generată automat

Then, I imported the certificate into Windows Store:

O imagine care conține text, captură de ecran, software, afișaj

Descriere generată automat O imagine care conține text, captură de ecran, software, afișaj

Descriere generată automat O imagine care conține text, software, Pictogramă computer, Pagină web

Descriere generată automat

PDF Signatures

O imagine care conține Font, text, Grafică, tipografie

Descriere generată automat

The PDF document can be signed as shown on the live lab. I will choose a dummy [PDF](https://www.w3.org/WAI/ER/tests/xhtml/testfiles/resources/pdf/dummy.pdf) found online from official W3C website.

O imagine care conține text, captură de ecran, software, calculator

Descriere generată automat

I am signing with the certificate generated from the ejbca CA and imported to Windows Store. This is the result.

O imagine care conține text, captură de ecran, software, Software multimedia

Descriere generată automat

O imagine care conține Font, text, Grafică, tipografie

Descriere generată automat

To validate the document, we go to the validation panel. I port-forwarded the required port for OCSP and the signature is validated. The only problem is that the revocation status of the MasterCA cannot be established due to no specification of a CLR or another method for the CA. If we add MasterCA to the Trusted CAs, everything goes smoothly and the chain is fully validated.

O imagine care conține text, captură de ecran, software, Pagină web

Descriere generată automat

O imagine care conține text, Font, captură de ecran, linie

Descriere generată automat

Disclaimer: I did not succeed the timestamping, but I will document the process as discussed in the live lab. First, we will have to choose a TSA server, like **freetsa.org** or **timestamp.acs.microsoft.com**. Add it to Adobe Acrobat timestamping settings and set it as default:

O imagine care conține text, captură de ecran, afișaj, software

Descriere generată automat O imagine care conține text, captură de ecran, software, Pagină web

Descriere generată automat

After that use the **Apply a document timestamp** from the **Use a certificate** sub-menu. The result will be a timestamped signed document.

XML Signatures

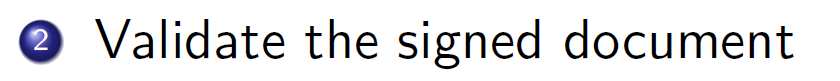
O imagine care conține text, Font, Grafică, siglă

Descriere generată automat

We use the XML Signer tool to do that on a dummy XML from the official Microsoft [site](https://learn.microsoft.com/en-us/previous-versions/windows/desktop/ms762271(v=vs.85)).

O imagine care conține text, captură de ecran, software, Pagină web

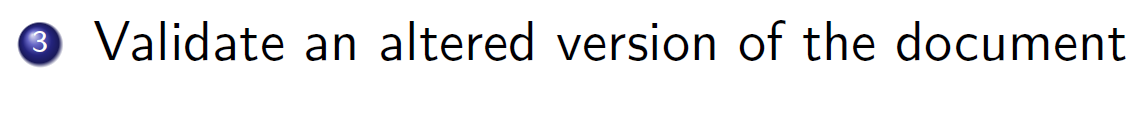
Descriere generată automat



To validate the signed XML, I will use a Linux tool called **xmlsec1**. After I added the full chain of certificates to the Kali’s trust store, I executed the command **xmlsec1 --verify --trusted-pem full\_chain.pem dummy\[signed\].xml** from the directory of the signed XML. The result:

O imagine care conține text, captură de ecran, Font

Descriere generată automat



I will edit the price of a book. Let’s say **Midnight Rain by Kim Ralls**. The new price will be **5,96**. Only the last digit will be modified! The signature is now invalid because hash of the document has changed.

O imagine care conține text, captură de ecran, Font

Descriere generată automat

PKCS#7 Signatures

O imagine care conține text, Font, captură de ecran, linie

Descriere generată automat

To obtain an attached signature of a picture, we will use P7S with the following settings:

O imagine care conține text, captură de ecran, software, număr

Descriere generată automat

O imagine care conține Font, Grafică, siglă, text

Descriere generată automat

To validate the signature we use this command **openssl smime -verify -inform DER -in dummy.jpg.p7s -content dummy.jpg -CAfile full\_chain.pem > /dev/null**. The result:

O imagine care conține text, captură de ecran, software, Font

Descriere generată automat

O imagine care conține text, Font, captură de ecran

Descriere generată automat

To obtain a detached signature we just check the box to save the signature on a separate file from the P7S Signer.

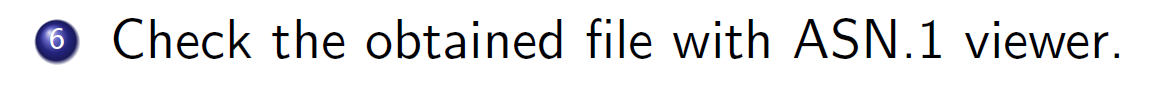
O imagine care conține text, captură de ecran, software, număr

Descriere generată automat

In order to validate the detached signature, we use the command **openssl smime -verify -inform DER -in dummy\_detached.jpg.p7s -content dummy.jpg -CAfile full\_chain.pem > /dev/null**. The result:

O imagine care conține text, captură de ecran, Font

Descriere generată automat



I will use the standard command from **OpenSSL**: **openssl asn1parse -inform DER -in dummy\_detached.jpg.p7s**. The output (a snapshot of it) will be:

O imagine care conține text, captură de ecran, software, Software multimedia

Descriere generată automat

Microsoft Word Signatures

O imagine care conține text, Font, captură de ecran, tipografie

Descriere generată automat

I did follow the steps from [here](https://www.idmanagement.gov/playbooks/signword/). The result:  
O imagine care conține text, captură de ecran, software, Pictogramă computer

Descriere generată automat

Stores

O imagine care conține text, captură de ecran, Font, linie

Descriere generată automat

We will start with Chrome. Chrome uses the Windows Trust store. We can see that in the settings:

O imagine care conține captură de ecran, text, software, proiectare

Descriere generată automat

Here we can see our certificate:

O imagine care conține text, captură de ecran, software, Pictogramă computer

Descriere generată automat

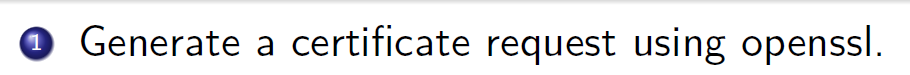
Now let’s head to Firefox.

O imagine care conține text, captură de ecran, software, Software multimedia

Descriere generată automat

Here we can see Mozilla’s own trusted authorities. We can also see our certificate on the **Own certificates** tab, but MasterCA authority (which is the authority that emitted our certificate) is not on the list of trusted authorities here.

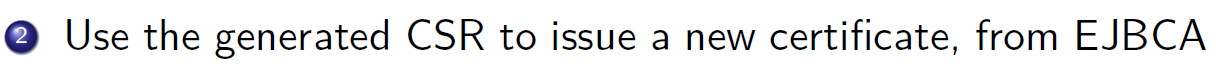
Certificate Requests



We use the provided command (adapted to our use case) **openssl req -newkey rsa:2048 -keyout private.key -out openssl.csr** and fill in the required fields. Here is the result:

O imagine care conține text, captură de ecran, afișaj

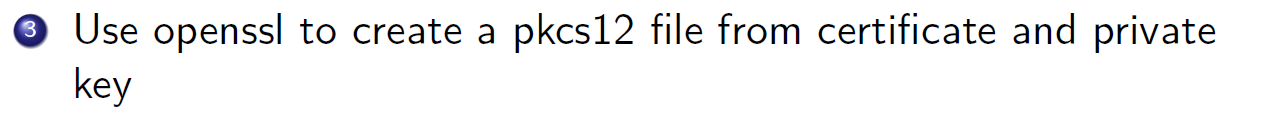
Descriere generată automat



We first must create an end entity from the admin interface. Then we navigate to the user interface and fill in the fields for enrollment via CSR. We will get a .pem file which will be our certificate.

O imagine care conține text, captură de ecran, Font, software

Descriere generată automat



Again we use the given command (adapted to our use case) **openssl pkcs12 -export -in encryption.crt -inkey enc\_private.key -out encryption.p12**. Here is the result:

O imagine care conține text, captură de ecran, Font

Descriere generată automat

And the PKCS#12 output file:

O imagine care conține text, electronice, captură de ecran, software

Descriere generată automat

O imagine care conține text, Font, captură de ecran, linie

Descriere generată automat

We import the certificate as the previous one, by following the wizard steps we followed in the live lab.

To sign, we use the steps from the live lab. I will use a copy of the **dummy.pdf** I used earlier. Here is the result:

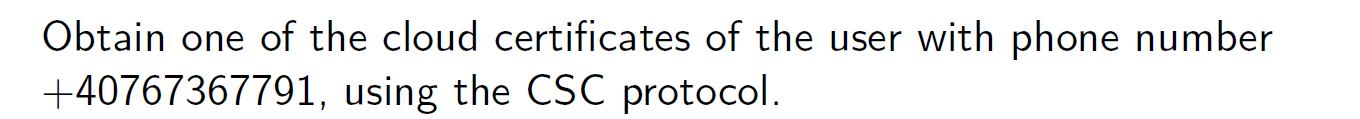
O imagine care conține text, captură de ecran, software, calculator

Descriere generată automat

O imagine care conține text, captură de ecran, software, Software multimedia

Descriere generată automat

CSC Protocol – ‘Hello world’ level



Let’s head to a REST API Client like **Thunder Client** extension from VS Code. Do the following POST to retrieve credentials IDs.

O imagine care conține text, captură de ecran, afișaj, software

Descriere generată automat

Now, to get the cloud certificate we have to do a POST like this:

O imagine care conține text, electronice, captură de ecran, software

Descriere generată automat

And here we have our cloud certificate.

Bonus - Python Certificate Requests

The Python source code:

from cryptography.hazmat.primitives.asymmetric import rsa

from cryptography.hazmat.primitives import serialization, hashes

from cryptography.hazmat.primitives.asymmetric import padding

from cryptography.x509.oid import NameOID

import cryptography.x509 as x509

*# Generate an RSA private key*

def generate\_private\_key():

    private\_key = rsa.generate\_private\_key(

        public\_exponent=65537,

        key\_size=2048,

    )

    return private\_key

*# Generate a Certificate Signing Request (CSR)*

def generate\_csr(private\_key, country, state, locality, org, org\_unit, common\_name, email):

*# Subject details*

    subject = x509.Name([

        x509.NameAttribute(NameOID.COUNTRY\_NAME, country),

        x509.NameAttribute(NameOID.STATE\_OR\_PROVINCE\_NAME, state),

        x509.NameAttribute(NameOID.LOCALITY\_NAME, locality),

        x509.NameAttribute(NameOID.ORGANIZATION\_NAME, org),

        x509.NameAttribute(NameOID.ORGANIZATIONAL\_UNIT\_NAME, org\_unit),

        x509.NameAttribute(NameOID.COMMON\_NAME, common\_name),

        x509.NameAttribute(NameOID.EMAIL\_ADDRESS, email),

    ])

*# Create CSR*

    csr = x509.CertificateSigningRequestBuilder().subject\_name(subject).sign(

        private\_key, hashes.SHA256()

    )

    return csr

*# Save private key to file*

def save\_private\_key(private\_key, filename):

    with open(filename, "wb") as f:

        f.write(

            private\_key.private\_bytes(

                encoding=serialization.Encoding.PEM,

                format=serialization.PrivateFormat.TraditionalOpenSSL,

                encryption\_algorithm=serialization.NoEncryption(),

            )

        )

*# Save CSR to file*

def save\_csr(csr, filename):

    with open(filename, "wb") as f:

        f.write(csr.public\_bytes(serialization.Encoding.PEM))

*# Main function*

if \_\_name\_\_ == "\_\_main\_\_":

*# CSR details*

    country = "US"

    state = "California"

    locality = "San Francisco"

    organization = "Example Corp"

    organizational\_unit = "IT Department"

    common\_name = "example.com"

    email\_address = "admin@example.com"

*# Generate private key*

    private\_key = generate\_private\_key()

    save\_private\_key(private\_key, "private\_key.pem")

*# Generate CSR*

    csr = generate\_csr(

        private\_key, country, state, locality, organization, organizational\_unit, common\_name, email\_address

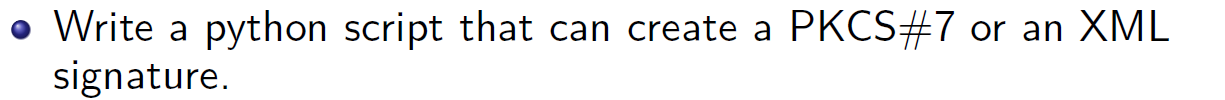
    )

    save\_csr(csr, "certificate\_request.csr")

    print("Private key and CSR have been generated.")

Next, the same steps as generating the CSR with **OpenSSL** must be followed to sign the PDF. I will not reproduce them again because I see it as a redundancy.

Bonus – Coding



from M2Crypto import SMIME, X509, BIO  
  
# Paths to necessary files  
PRIVATE\_KEY\_FILE = "private.key" # Path to your private key  
CERTIFICATE\_FILE = "certificate.pem" # Path to your certificate  
DATA\_FILE = "dummy.txt" # Path to the file to be signed  
OUTPUT\_SIGNATURE = "signed\_dummy.p7s" # Output PKCS#7 signature file  
  
  
def create\_pkcs7\_signature(data\_file, private\_key\_file, certificate\_file, output\_signature):  
 # Create an SMIME object  
 smime = SMIME.SMIME()  
  
 # Load the signer's certificate  
 smime.load\_key(private\_key\_file, certificate\_file)  
  
 # Read the data to be signed  
 with open(data\_file, "rb") as f:  
 data\_bio = BIO.MemoryBuffer(f.read())  
  
 # Create a PKCS#7 signature  
 pkcs7 = smime.sign(data\_bio, SMIME.PKCS7\_DETACHED)  
  
 # Write the signature to a file  
 with open(output\_signature, "wb") as f:  
 out\_bio = BIO.File(f)  
 smime.write(out\_bio, pkcs7, data\_bio)  
  
 print(f"PKCS#7 signature created and saved to {output\_signature}")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 create\_pkcs7\_signature(DATA\_FILE, PRIVATE\_KEY\_FILE, CERTIFICATE\_FILE, OUTPUT\_SIGNATURE)

The verification went smoothly with the same approach of **OpenSSL**.