Premises

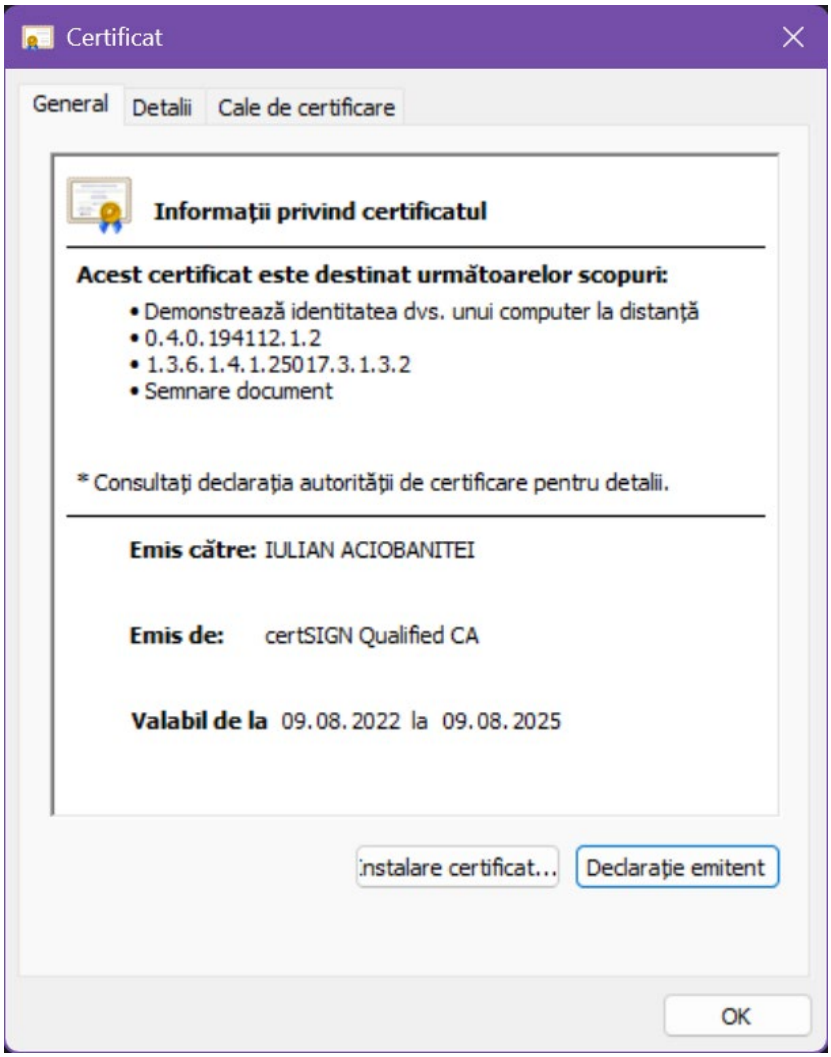
Lab Activity – Liviu-Ioan ZECHERU

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

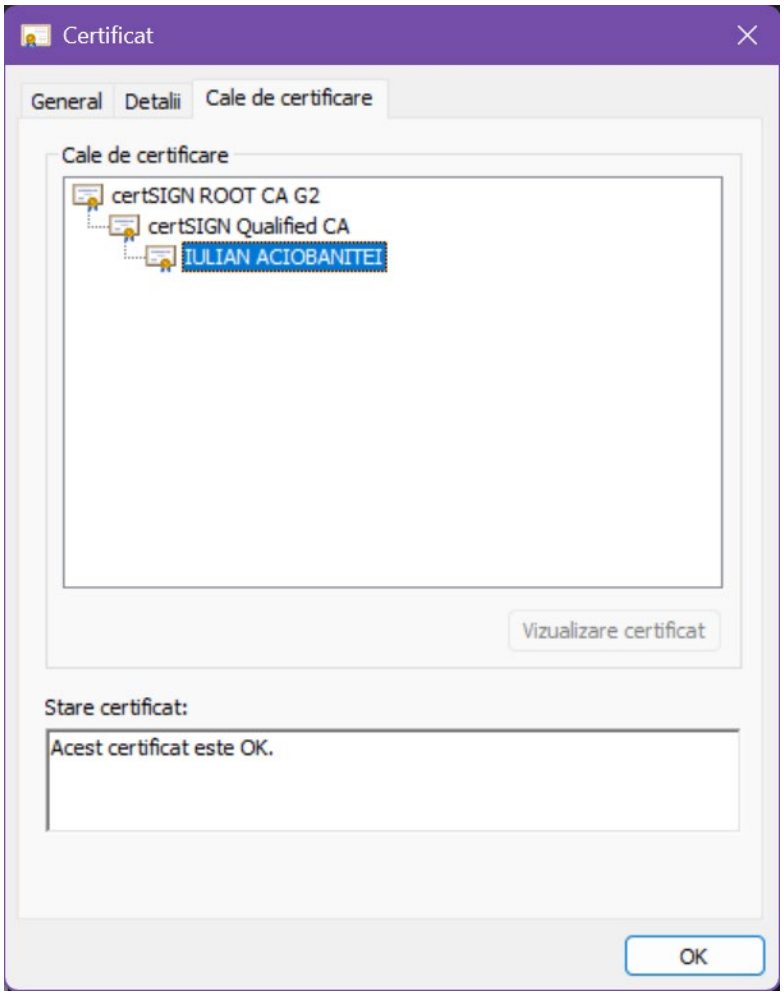
Certificates

1 Download this certificate

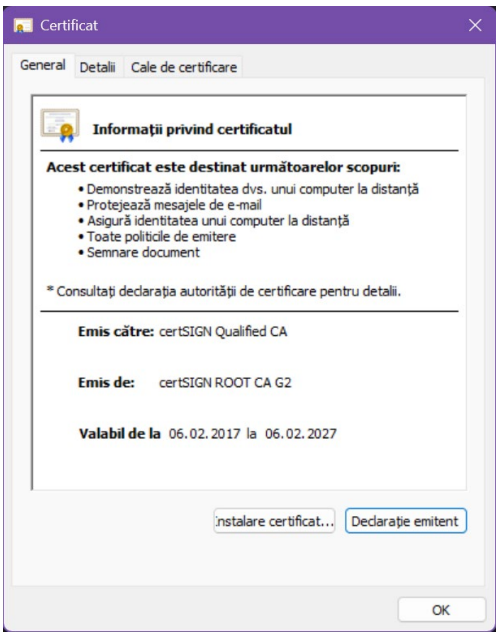
(link behind - https://drive.google.com/file/d/16KXExKkzw4b35OedxrJ7hAA6YTLbQk72/view?usp=share_link)

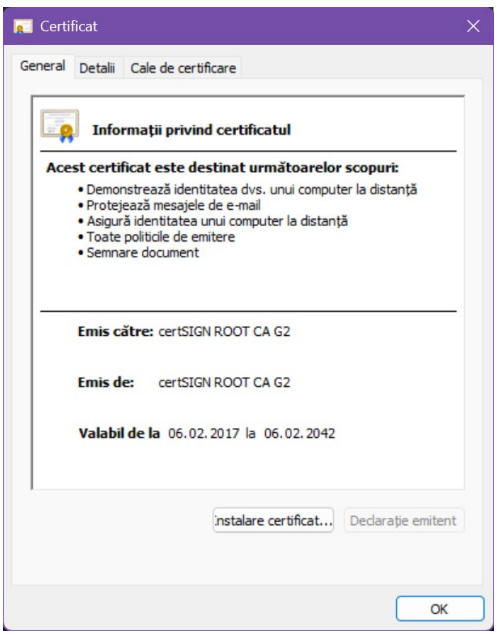


2. Download all the certificates in the certification path



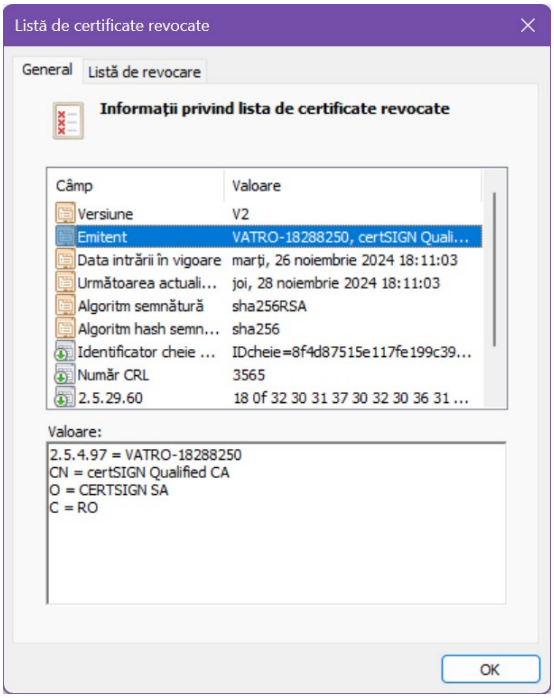
The certificates in the certification path are: certSIGN Qualified CA and certSIGN ROOT CA G2. Here they are:

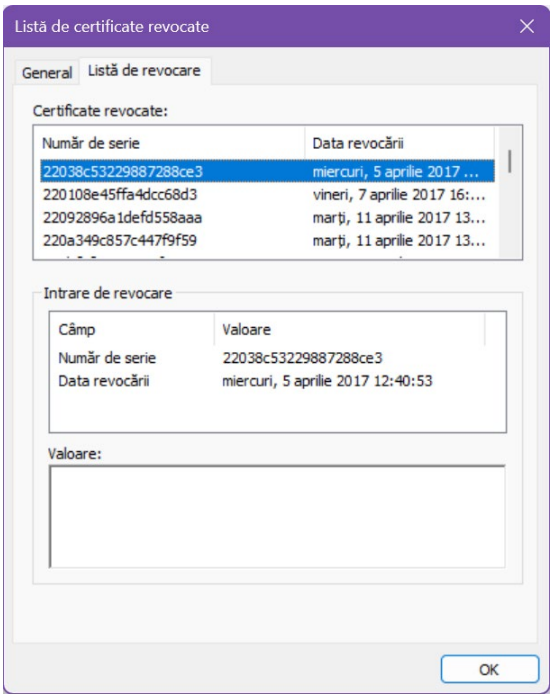




3 Download the CRL manually

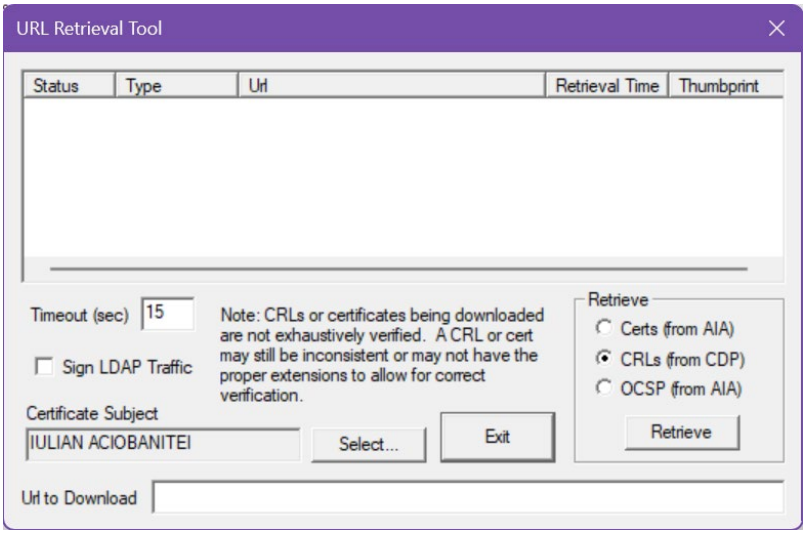
According to the details of the certificate, the CRL is located here: http://crl.certsign.ro/certsign-qualifiedca.crl. Accessing the link will give us this file:



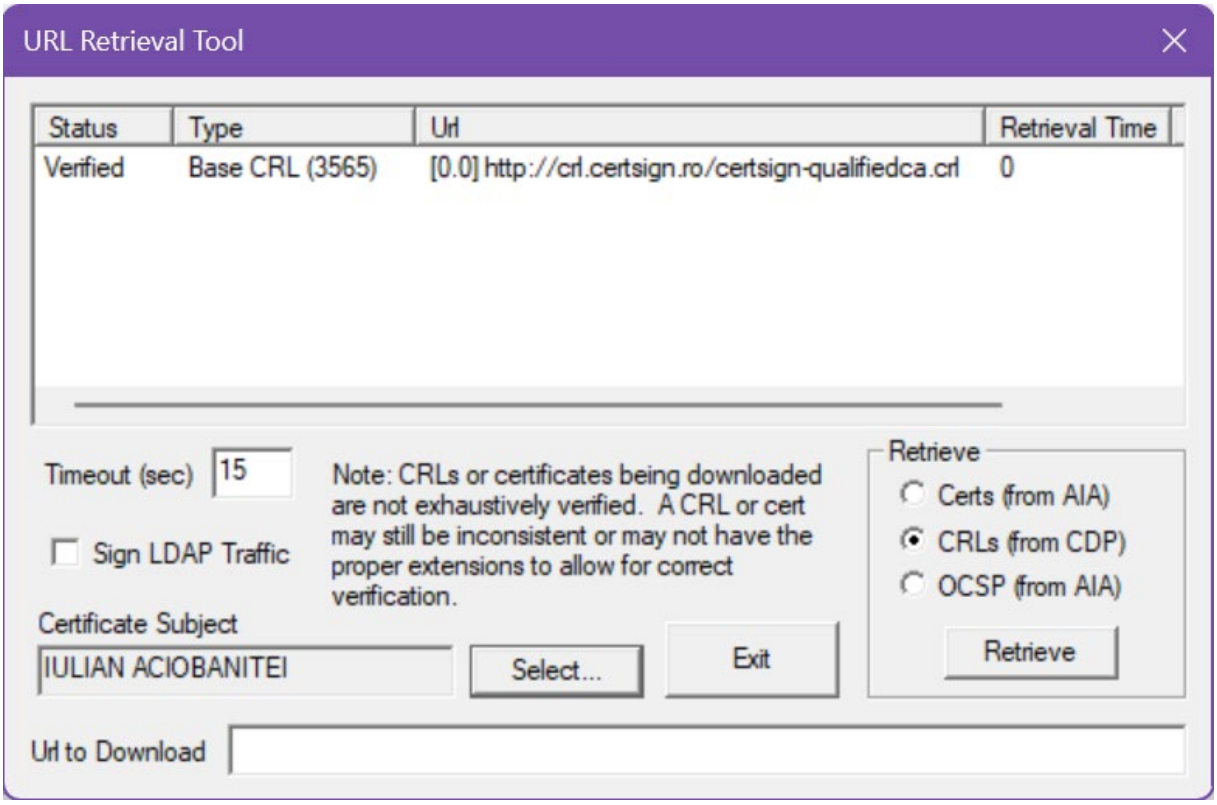


4 Verify the certificate using certutil and CRL

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:

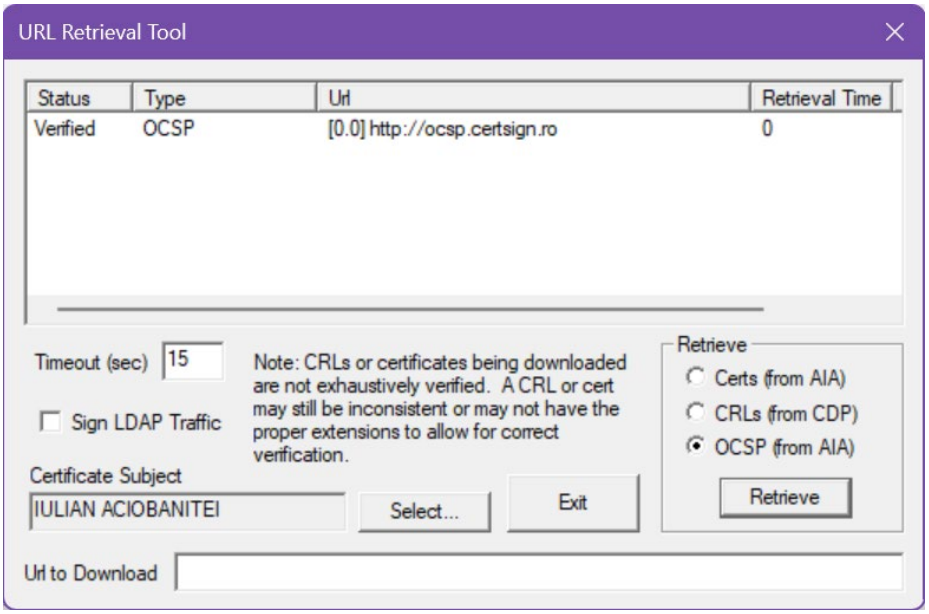


To verify the certificate using the CRL we select the **CRLs (from CDP)** as retrieval method then click on **Retrieve** button. Then we can see the status in the view area.



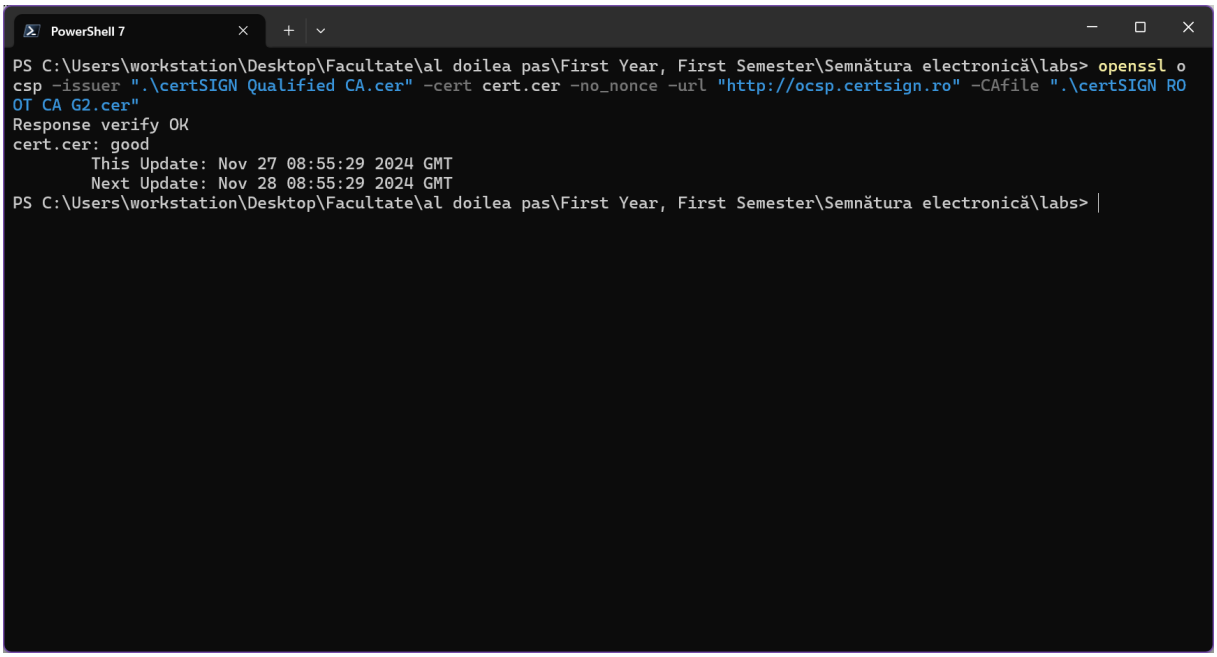
5 Verify the certificate using certutil and OCSP

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.



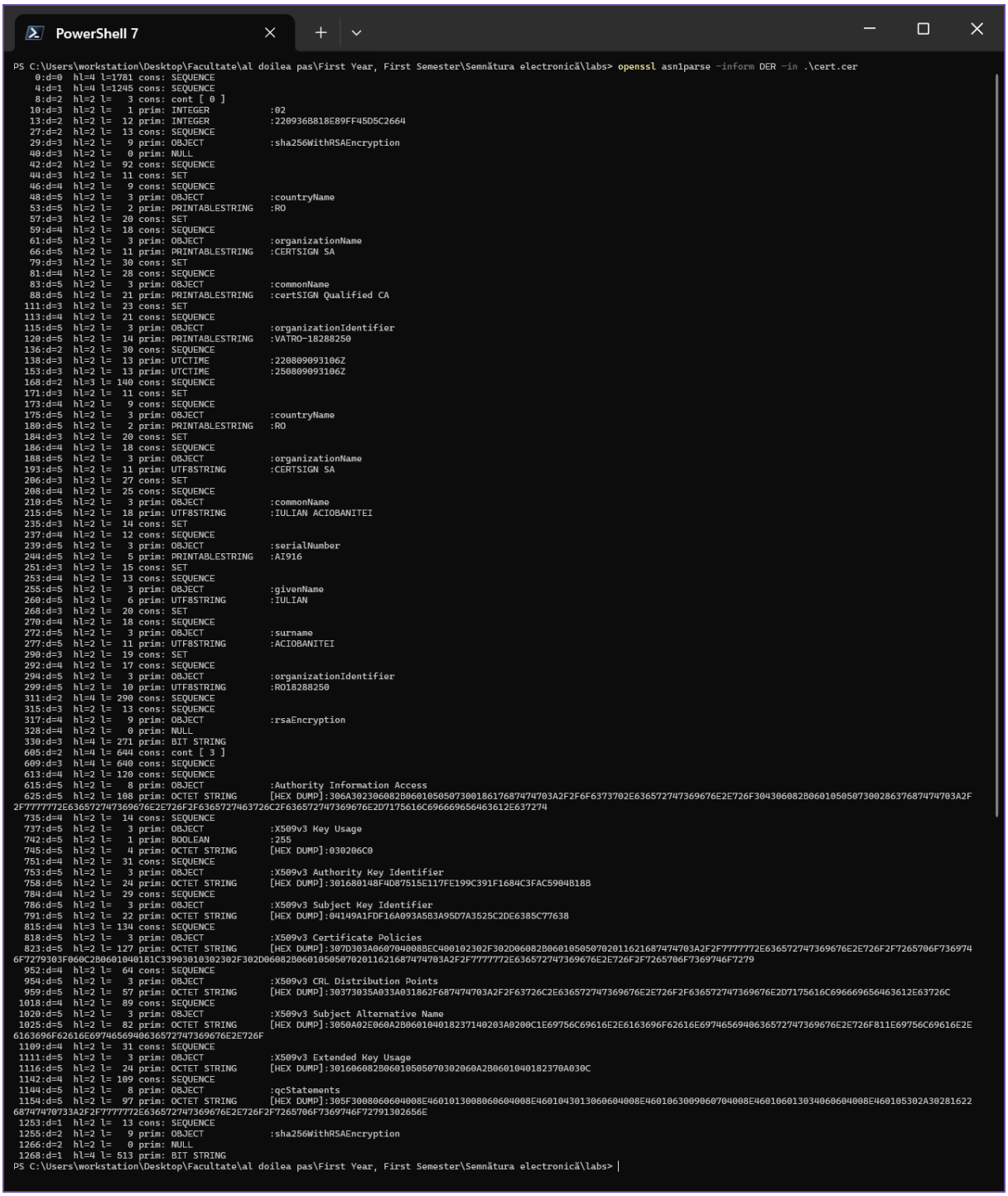
6 Verify the certificate using openssl*

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:



7 View certificate using an ASN.1 viewer (openssl cmd or lapo.it)

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:



8 Extract the OID of an atribute from Subject extension, at will.

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

Common Name (CN)

Organization (O)

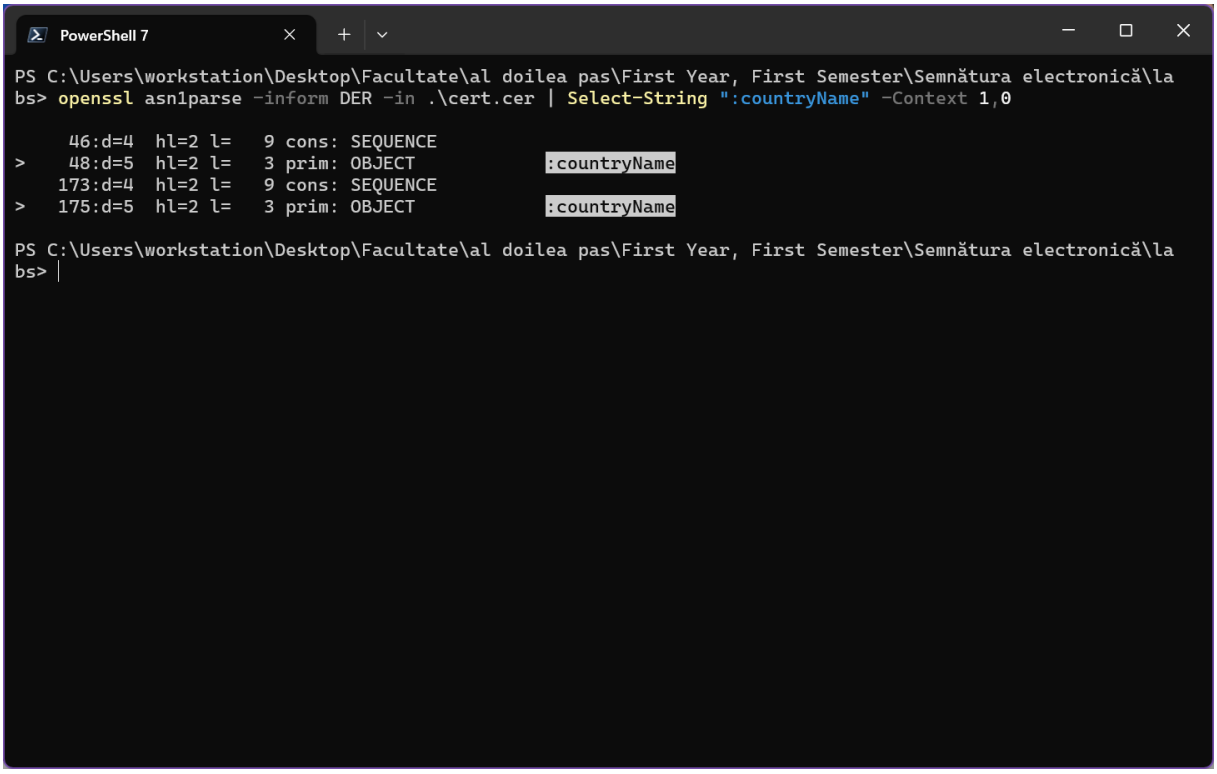
Organizational Unit (OU)

Country (C)

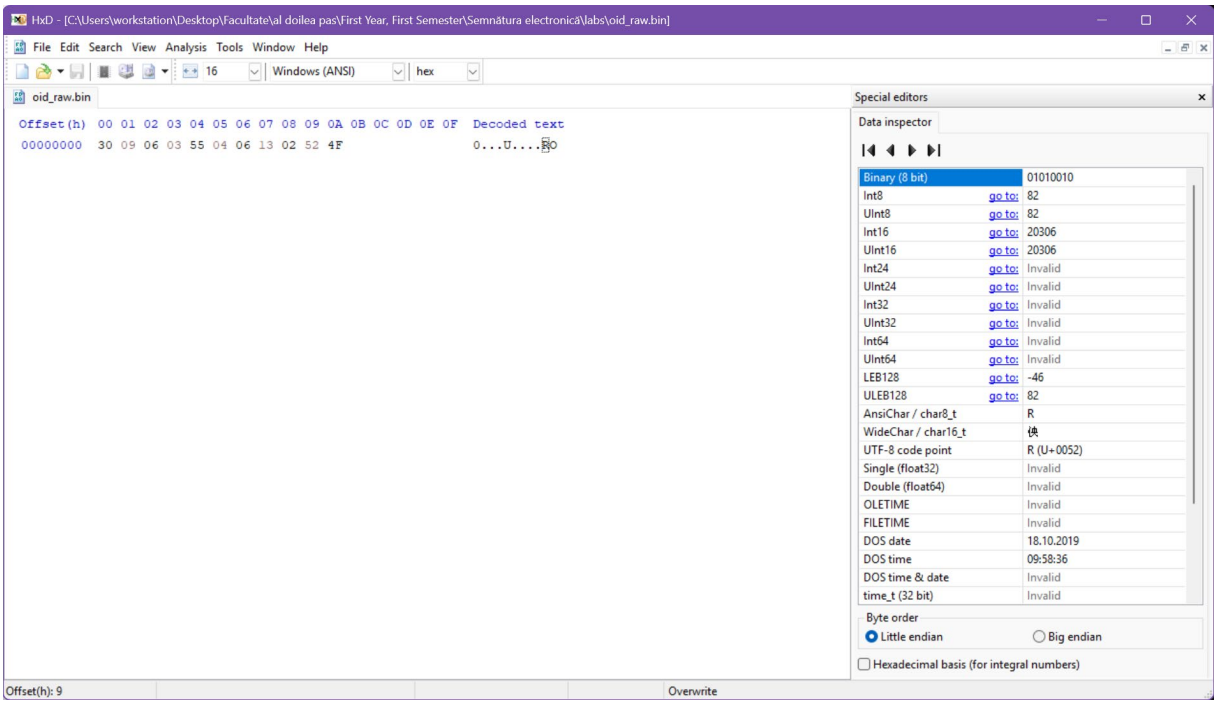
State or Province Name (ST)

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:



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:



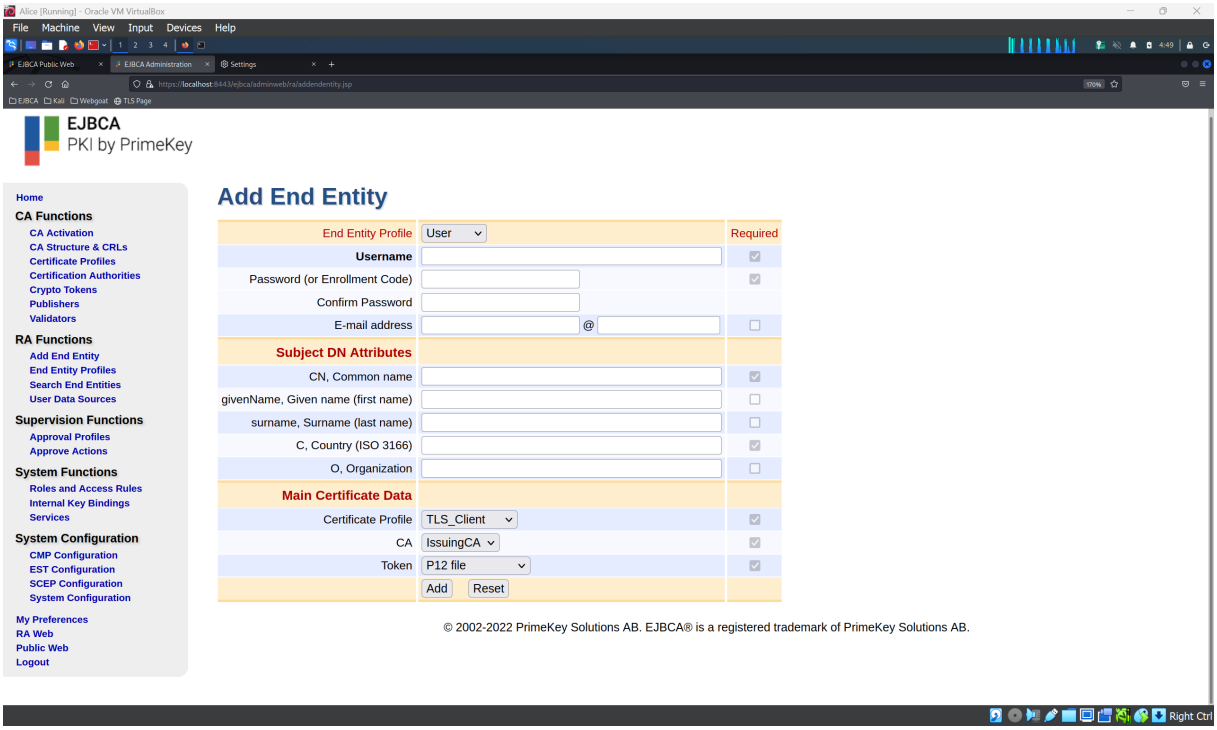
We know that OID for the countryName is **2.5.4.6**. Considering the ASN1 encoding, the first byte encodes the first two components by the formula 40 * X + Y. So, in our case it will be 40 * **2** + **5** = 85 which is **55** in hexadecimal format. The subsequent components are encoded as a series of 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!

---

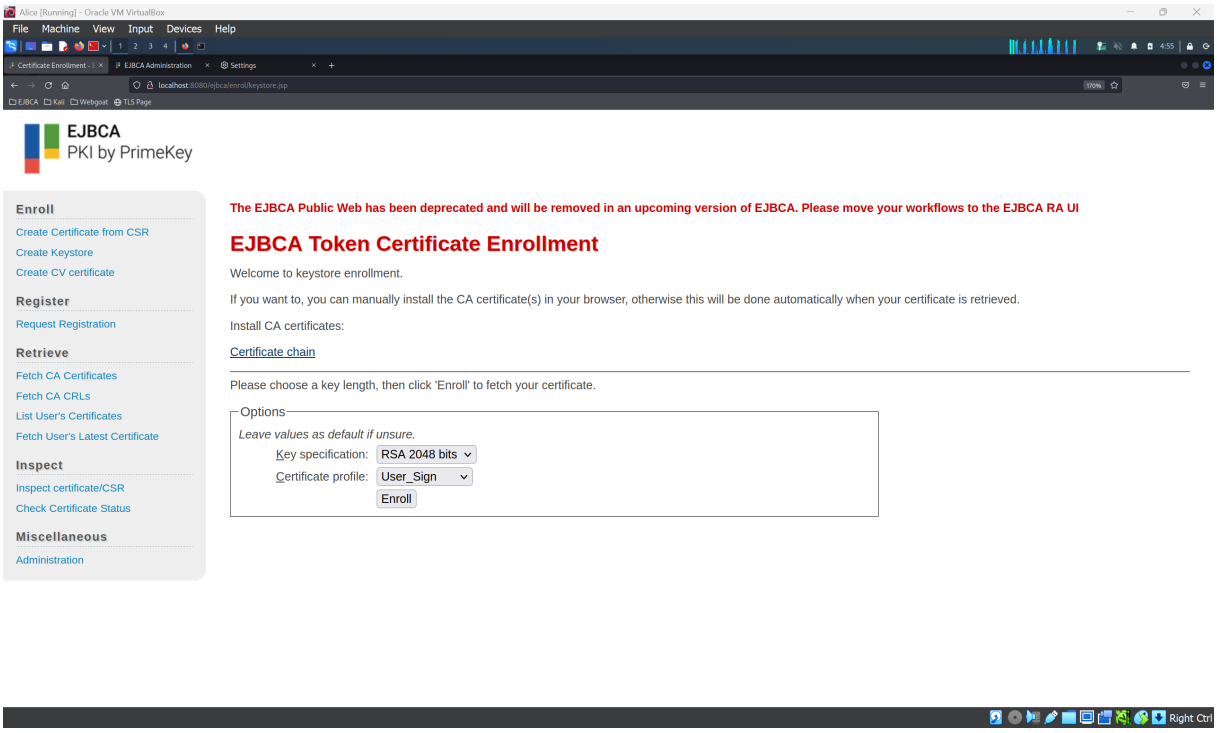
1. From EJBCA's admin interface, create a new end entity of type User_Sign or TLS_Client
2. From EJBCA's public interface, actually generate the certificate and obtain the .p12 file.
3. Import the generated certificate into the Windows Store.

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.

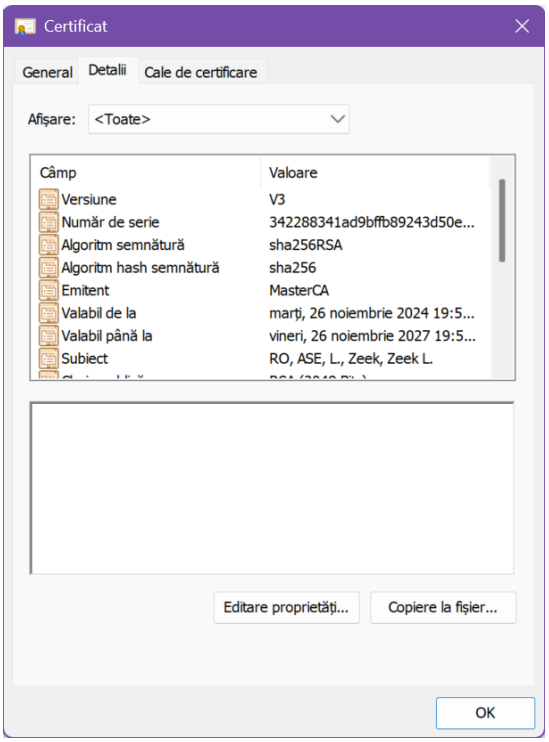
Here is the end entity creation page where, me, as CA, had to issue a new certificate for an end user:

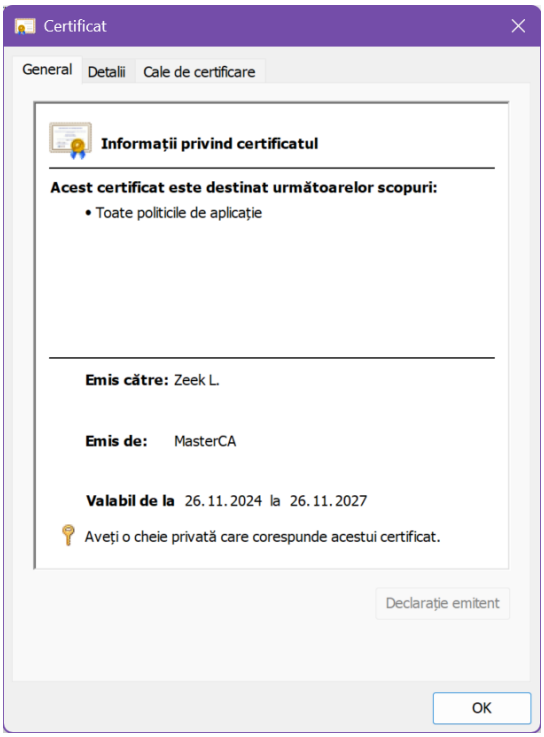


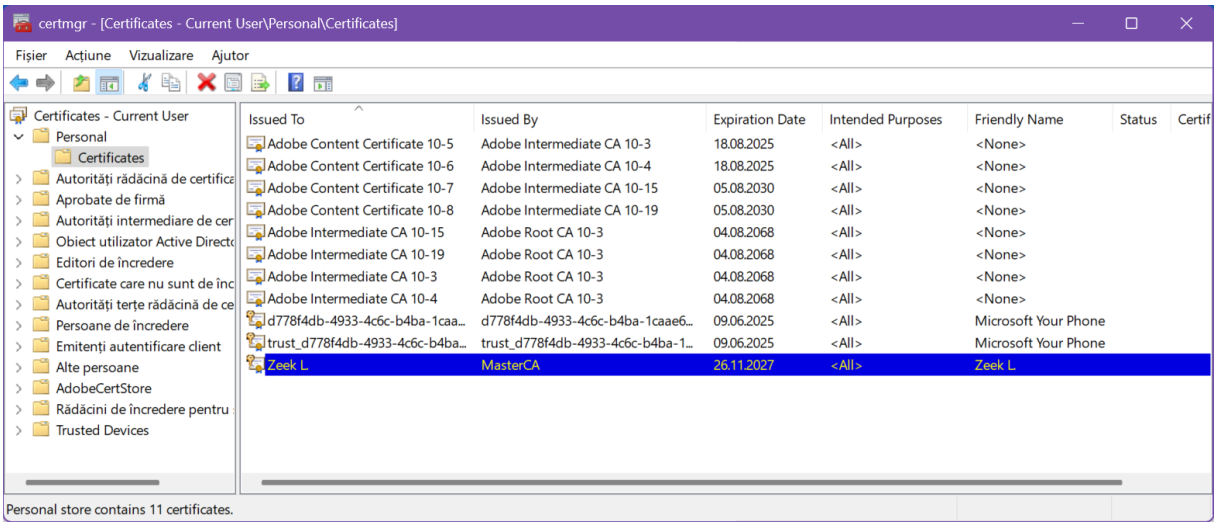
Here, as the end user, on the public page, I logged on with the created credentials and generated my PKCS#12 file.



Then, I imported the certificate into Windows Store:



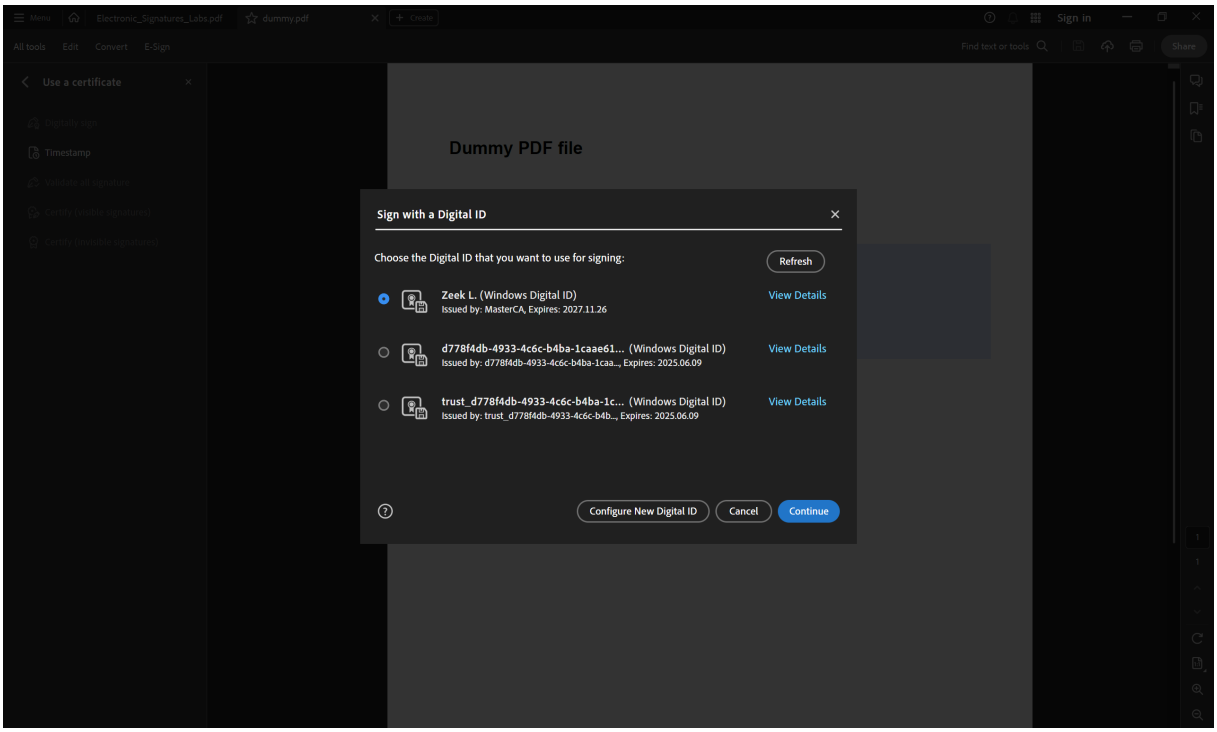




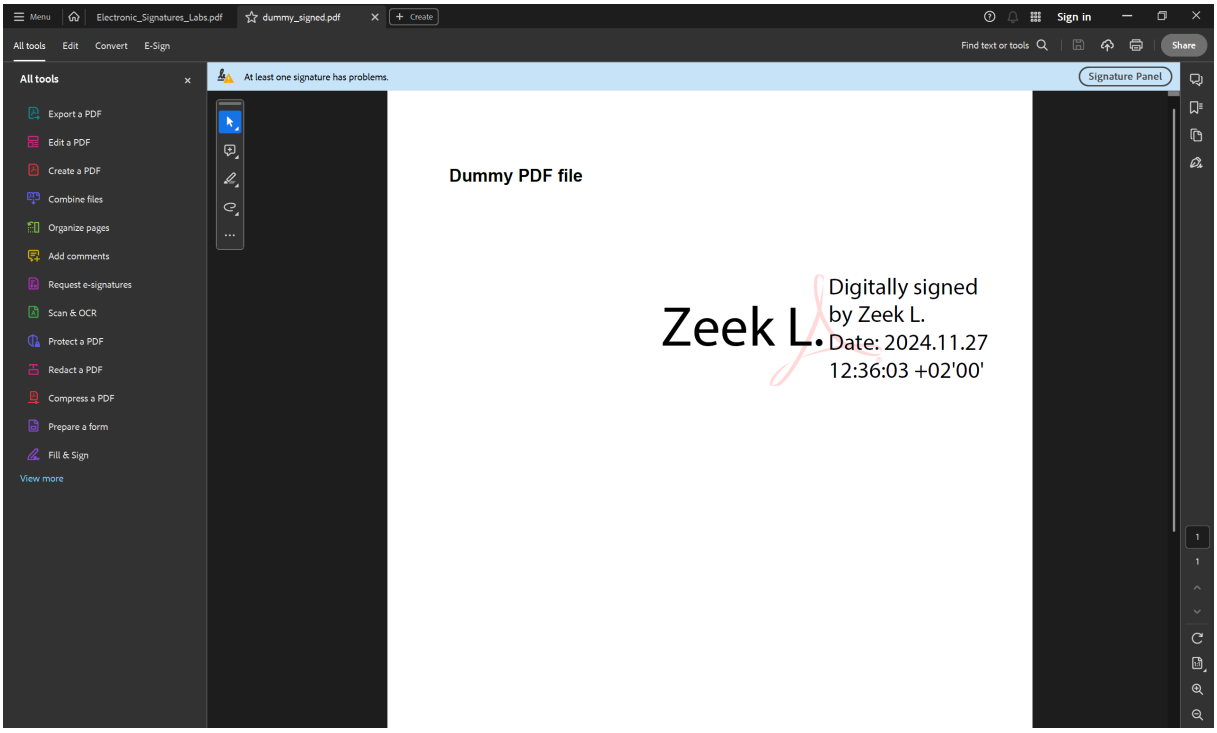
PDF Signatures

1 Sign a document

The PDF document can be signed as shown on the live lab. I will choose a dummy PDF found online from official W3C website.

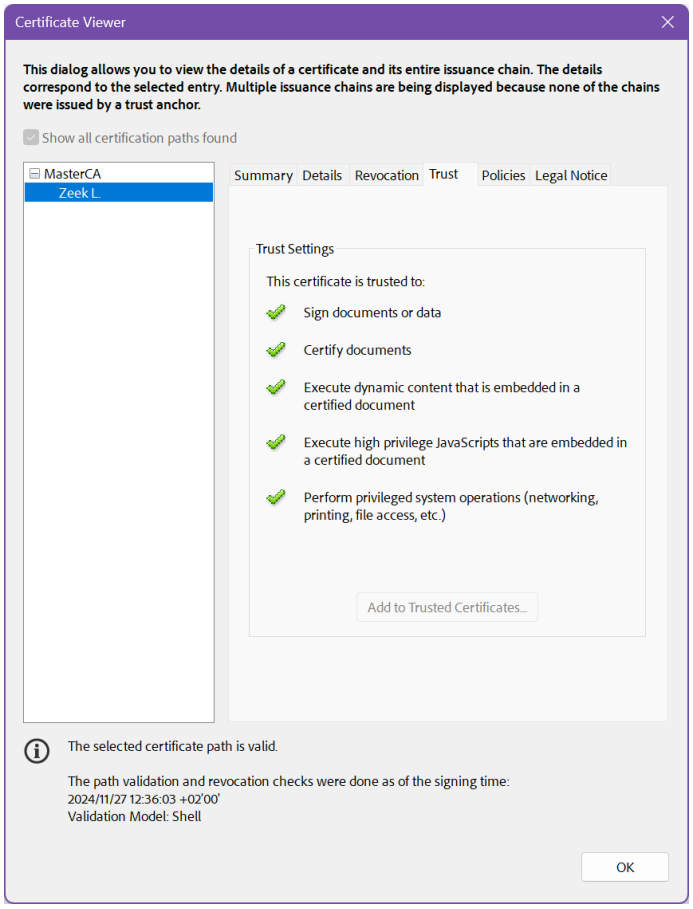


I am signing with the certificate generated from the ejbca CA and imported to Windows Store. This is the result.



2 Validate a document

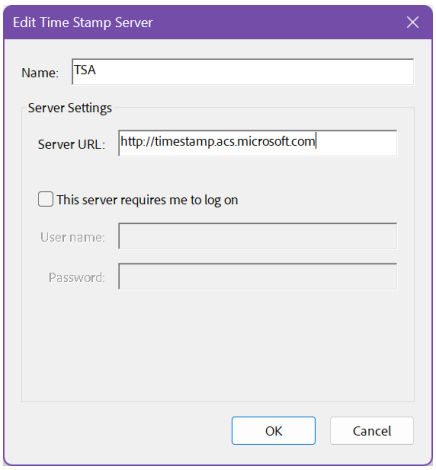
To validate the document, we go to the validation panel. I port-forwarded the required port for OCSP and the signature is validated. The only problem is that the revocation status of the MasterCA cannot be established due to no specification of a CLR or another method for the CA. If we add MasterCA to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

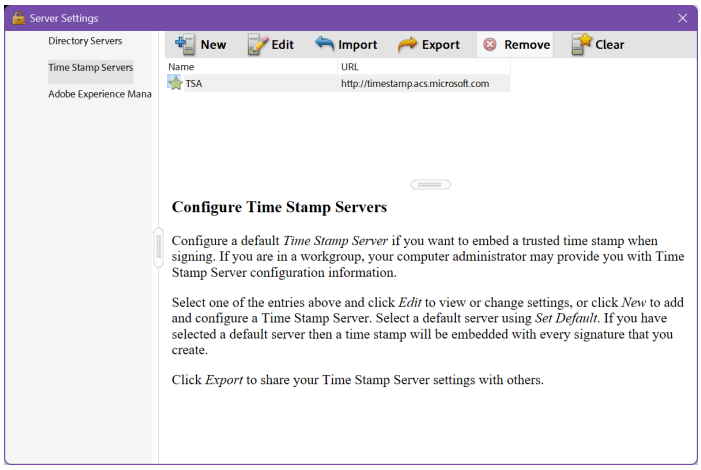


---

3. Search for a free timestamp server. Hint: github
4. Apply a timestamp to the document
5. Validate the document and check the signature format this time

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:



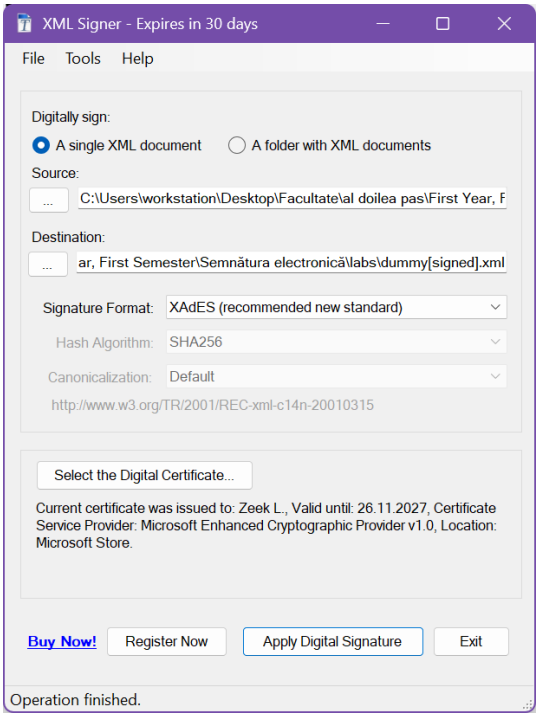


After that use the **Apply a document timestamp** from the **Use a certificate** sub-menu. The result will be a timestamped signed document.

XML Signatures

1 Sign an XML document

We use the XML Signer tool to do that on a dummy XML from the official Microsoft site.



2 Validate the signed document

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:

```
┌──(root㉿kali)-[/home/kali/Desktop/shared]
└─# xmlsec1 --verify --trusted-pem full_chain.pem dummy\[signed\].xml

OK
SignedInfo References (ok/all): 2/2
Manifests References (ok/all): 0/0
```

3 Validate an altered version of the document

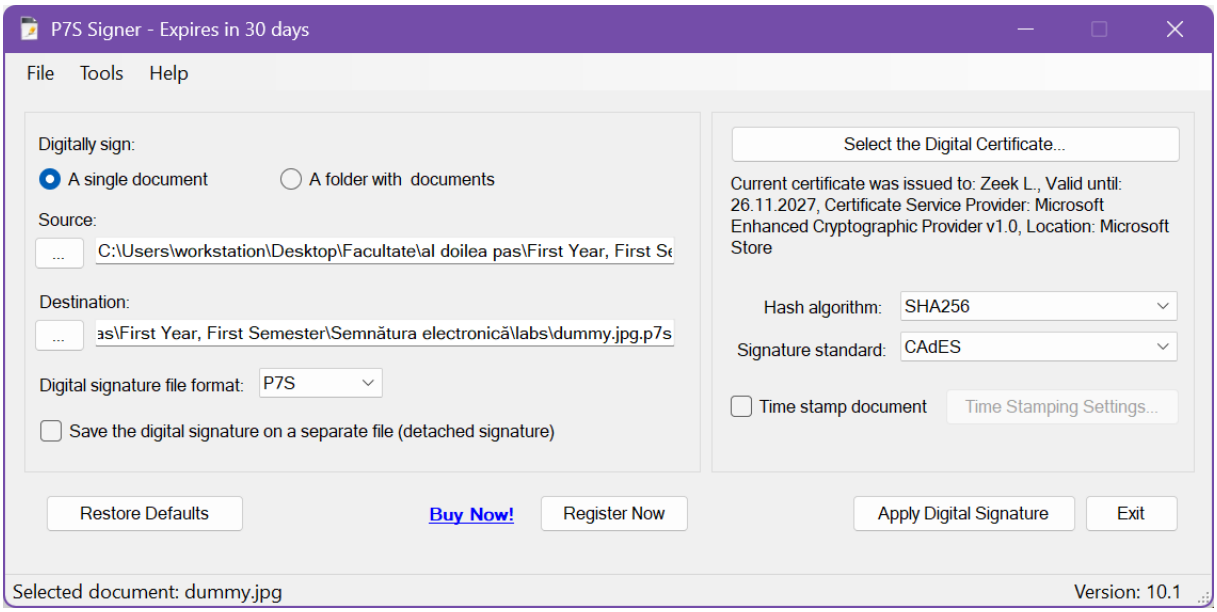
I will edit the price of a book. Let's say **Midnight Rain by Kim Ralls**. The new price will be **5,96**. Only the last digit will be modified! The signature is now invalid because hash of the document has changed.

```
┌──(root㉿kali)-[/home/kali/Desktop/shared]
└─# xmlsec1 --verify --trusted-pem full_chain.pem dummy\[signed\].xml
func=xmlSecOpenSSLEvpDigestVerify:file=digests.c:line=355:obj=sha256:subj=unknown:e
rror=12:invalid data:data and digest do not match
FAIL
SignedInfo References (ok/all): 0/1
Manifests References (ok/all): 0/0
Error: failed to verify file "dummy[signed].xml"
```

PKCS#7 Signatures

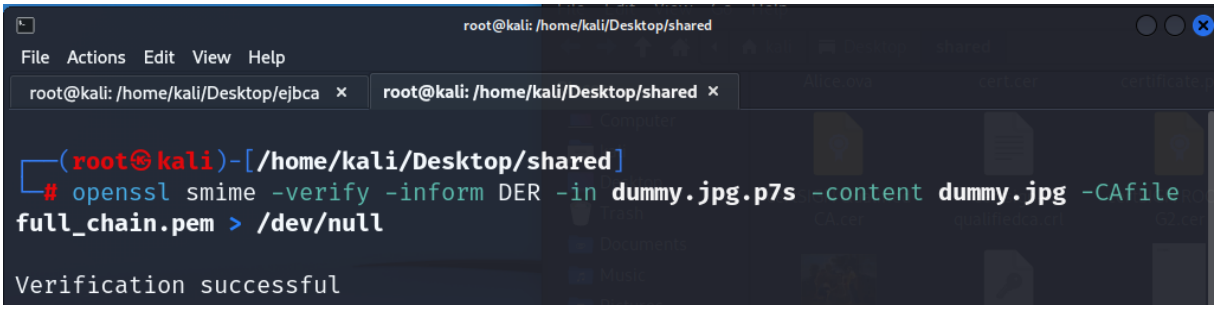
1. Use a media document - a picture, a text file, or other file type.
2. Obtain an attached signature

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



3 Validate it

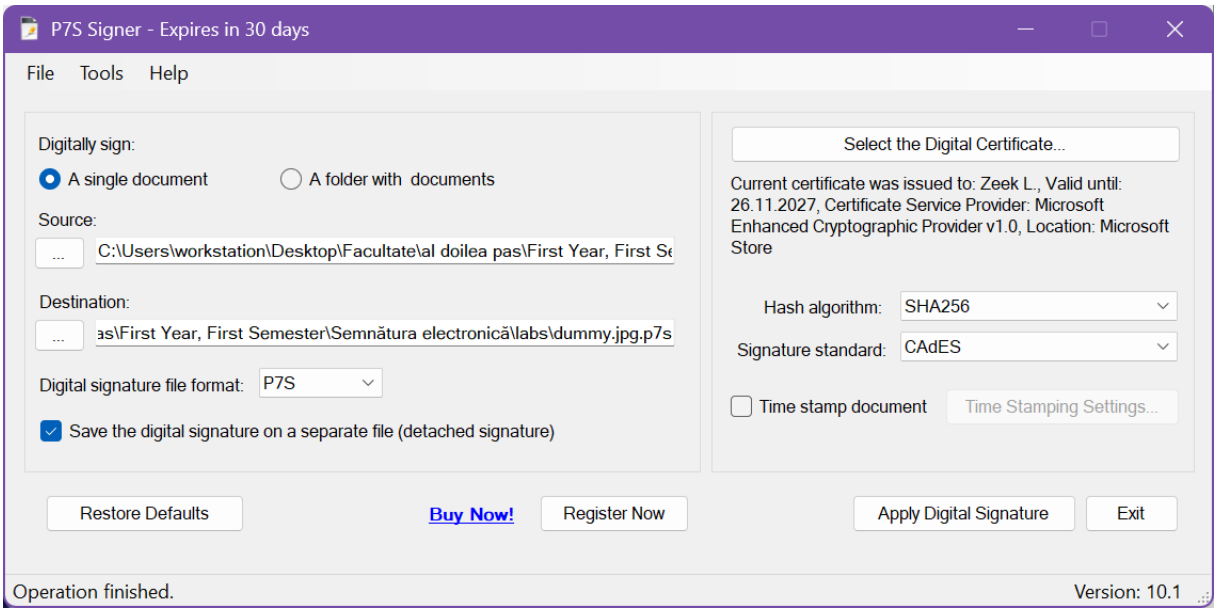
To validate the signature we use this command **openssl smime -verify -inform DER -in dummy.jpg.p7s -content dummy.jpg -CAfile full_chain.pem > /dev/null**. The result:



4 Obtain a detached signature

5 Validate it

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

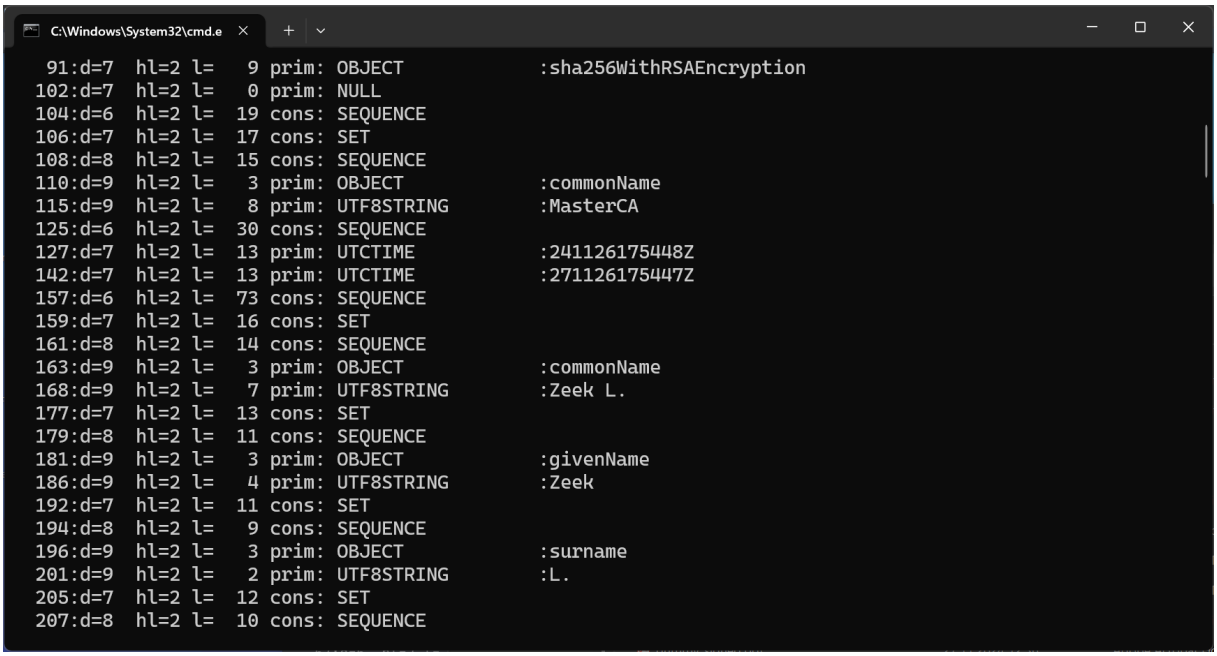


In order to validate the detached signature, we use the command **openssl smime -verify -inform DER -in dummy_detached.jpg.p7s -content dummy.jpg -CAfile full_chain.pem > /dev/null**. The result:

```
┌──(root㉿kali)-[/home/kali/Desktop/shared]
└─# openssl smime -verify -inform DER -in dummy_detached.jpg.p7s -content dummy.jpg
 -CAfile full_chain.pem > /dev/null

Verification successful
```

6 Check the obtained file with ASN.1 viewer.

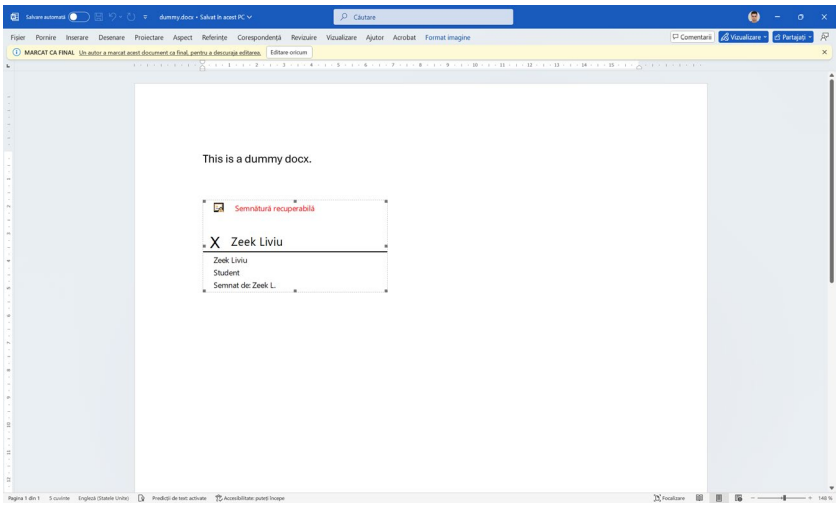
I will use the standard command from **OpenSSL**: **openssl asn1parse -inform DER -in dummy_detached.jpg.p7s.** The output (a snapshot of it) will be:



Microsoft Word Signatures

1 Sign a document using Microsoft Word.

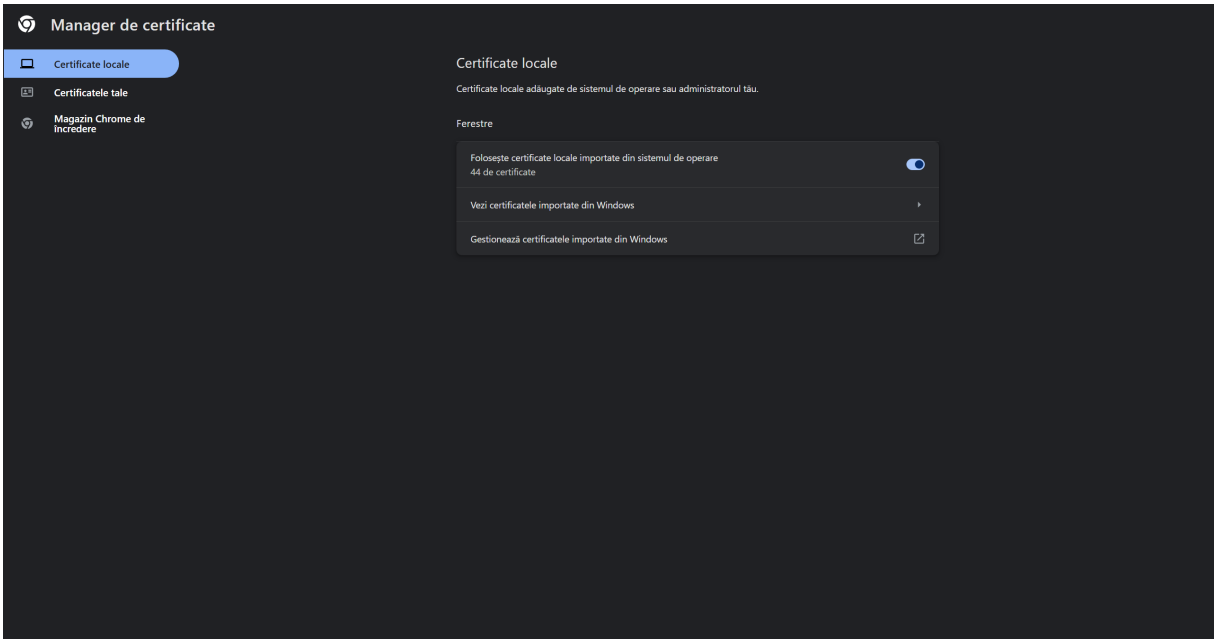
I did follow the steps from here. The result:



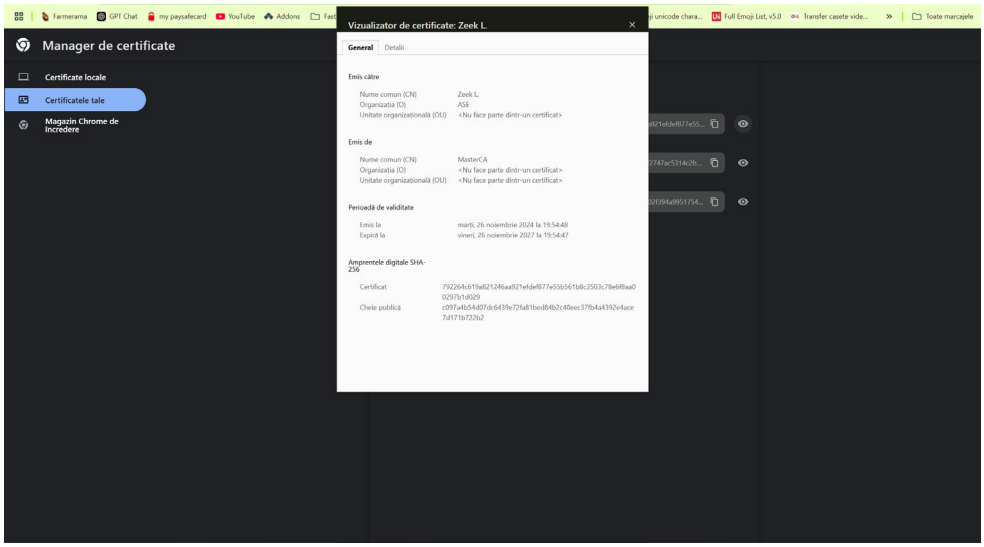
Stores

1. Prove that we are using different stores to validate the certificate of the web server.
2. We can use https://testssl.certsign.ro/
3. Hint: Should use different browsers (Mozilla+Chrome, for example)

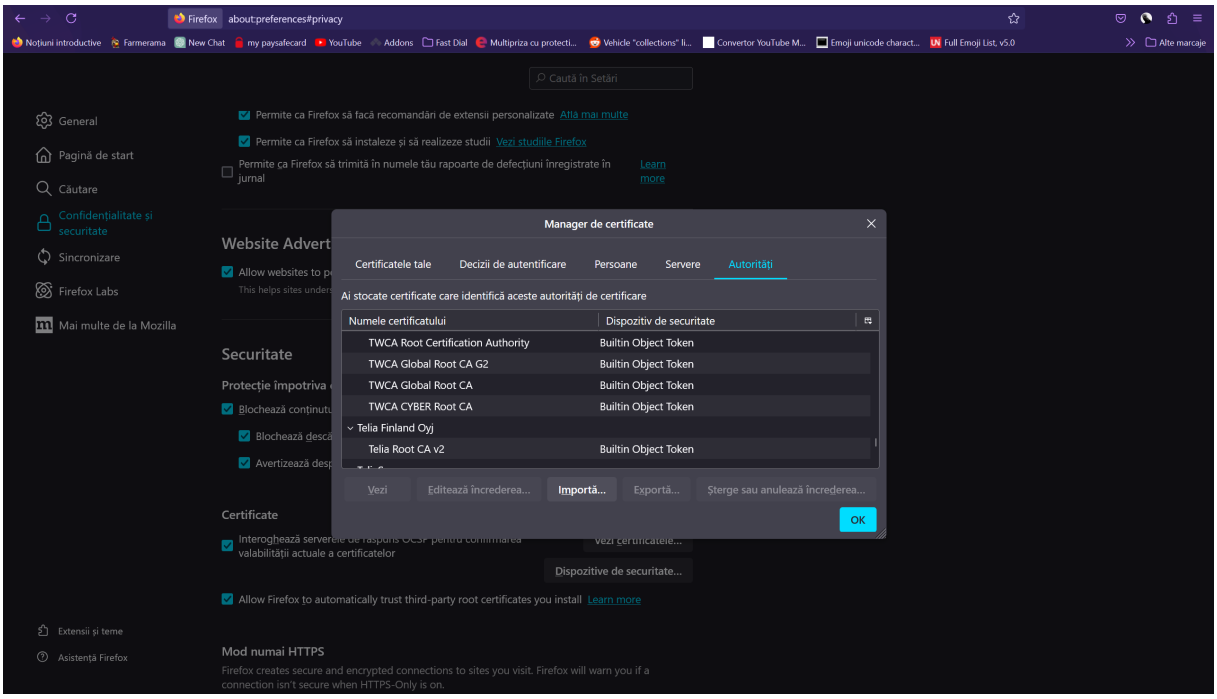
We will start with Chrome. Chrome uses the Windows Trust store. We can see that in the settings:



Here we can see our certificate:



Now let's head to Firefox.



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

1 Generate a certificate request using openssl.

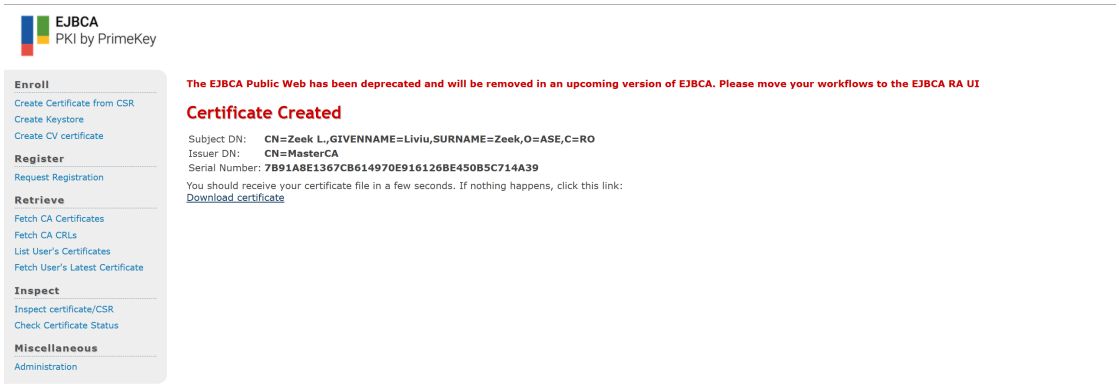
We use the provided command (adapted to our use case) **openssl req -newkey rsa:2048 -keyout private.key -out openssl.csr** and fill in the required fields. Here is the result:

PowerShell 7

```
PS C:\Users\workstation\Desktop\Facultate\al doilea pas\First Year, First Semester\Semnătura electronică\labs> openssl r
eq -newkey rsa:2048 -keyout private.key -out openssl.csr
.+.....+.....+...+.+.....+.+.....+....+..++++++++++++++++++++++++++++++++++++++++*..+.+.............+.+.....+.........
+..+.+.....+...+...........+.........+.....+.....+..+.....+.....+.+..++++++++++++++++++++++++++++++++++++++++*.....
.....+....+.....+.......+.....+...+.....+...+.....+.............+...........+.....+.........+....+..+.+..+...+
.......+..........+.............+...+..+....+.............+...........+.....+...+..+.+............+.+...+..
+.........+....+..+.............+..+...+....+.................+...+.....+.....+...+.....+..+.+............+....
.............+...+.......+.....+.+.....+...+.....+..........+.+...+..+........+.........+...+..+.....+...+....+..
.+.....+....+...+..+.........+........+...+.........+..+.....+.+.....+.....+...+.....+.+.....+.......+..........
..+..+................+.............+...+.....+.....+.............+...+.....................+...+.....+.....+....
......+..........+.........+.......+........+.....+.............+...+.....+.....+..+...........+.+.....+........
....+.............+.....+.............+.+.....+..+.............+...+...+.+.....+........+....+..+....+..+...+..
..............+.....+.....+.....+..+.+..........+....+.....+....+...+.....+.....+.+........+....+..+.....+..
..........+..+....+.....+.........+............+...+.................+.+..+.+.....+.+...+..........+..........
.............+....+.........+.........+.....+.....+.+...+.....+.....+.............+.+.....+.....+...+....+.....+..
..............+............+.........+...+..+.+..+...+.........+.+...........+...+..+.......+.....+...+.........+...
+.............+.....+.......+.....+.+.............+.+...+..+.+.........+.....................+.....+...........+...
+...........+..+.......+.....+...+...+...+.+...+......+.....+.+.....+.............+.........+...+...+..+.....+.....
.+....+.....+.......+................+.+............+........+.......++++++
..+......+......++++++++++++++++++++++++++++++++++++++++++++++*.+.........+..+.+........+.........+..+.........+.....+....+..
......+.+..+...+.............+...+...+..+......+......+.+..++++++++++++++++++++++++++++++++++++++++*.......+....+.....
......+.......+..+.......+.........+..+...+..............+.....................+..........+........+.......+.....+...+
..+.........+.....+.......+..+......+.......+......+.....+.+...+.....................+.....+........+.........+...
..+..........+..+...+.........+....+.....+.+.........+.....+.+........+.....................+.+...+.....+..............
............+..........+...+.....+...+....+...+...+...+..+.+.......+..+....++++++
Enter PEM pass phrase:

Verifying - Enter PEM pass phrase:

-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:RO
State or Province Name (full name) [Some-State]:Romania
Locality Name (eg, city) []:Bucharest
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Academia de Studii Economice Bucuresti
Organizational Unit Name (eg, section) []:CSIE
Common Name (e.g. server FQDN or YOUR name) []:Zeek L.
Email Address []:zecheruliviu21@stud.ase.ro

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:kali
An optional company name []:ASE
PS C:\Users\workstation\Desktop\Facultate\al doilea pas\First Year, First Semester\Semnătura electronică\labs>
```

2 Use the generated CSR to issue a new certificate, from EJBCA

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.

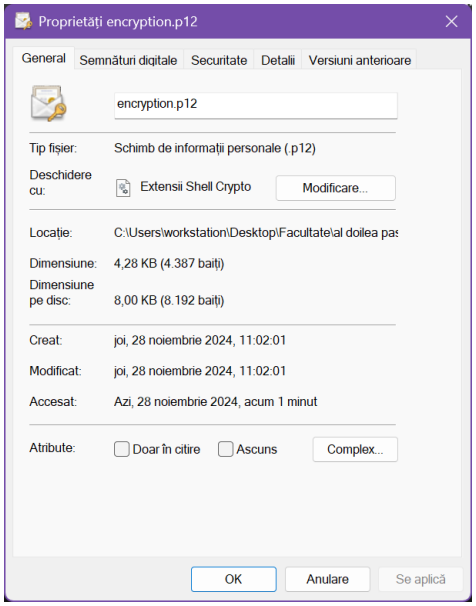


3 Use openssl to create a pkcs12 file from certificate and private key

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:

```
PS C:\Users\workstation\Desktop\Facultate\al doilea pas\First Year, First Semester\Semnătura electronică\labs> openssl
pkcs12 -export -in .\ZeekL.pem -inkey .\private.key -out encryption.p12
Enter pass phrase for .\private.key:

Enter Export Password:

Verifying - Enter Export Password:

PS C:\Users\workstation\Desktop\Facultate\al doilea pas\First Year, First Semester\Semnătura electronică\labs>
```

And the PKCS#12 output file:

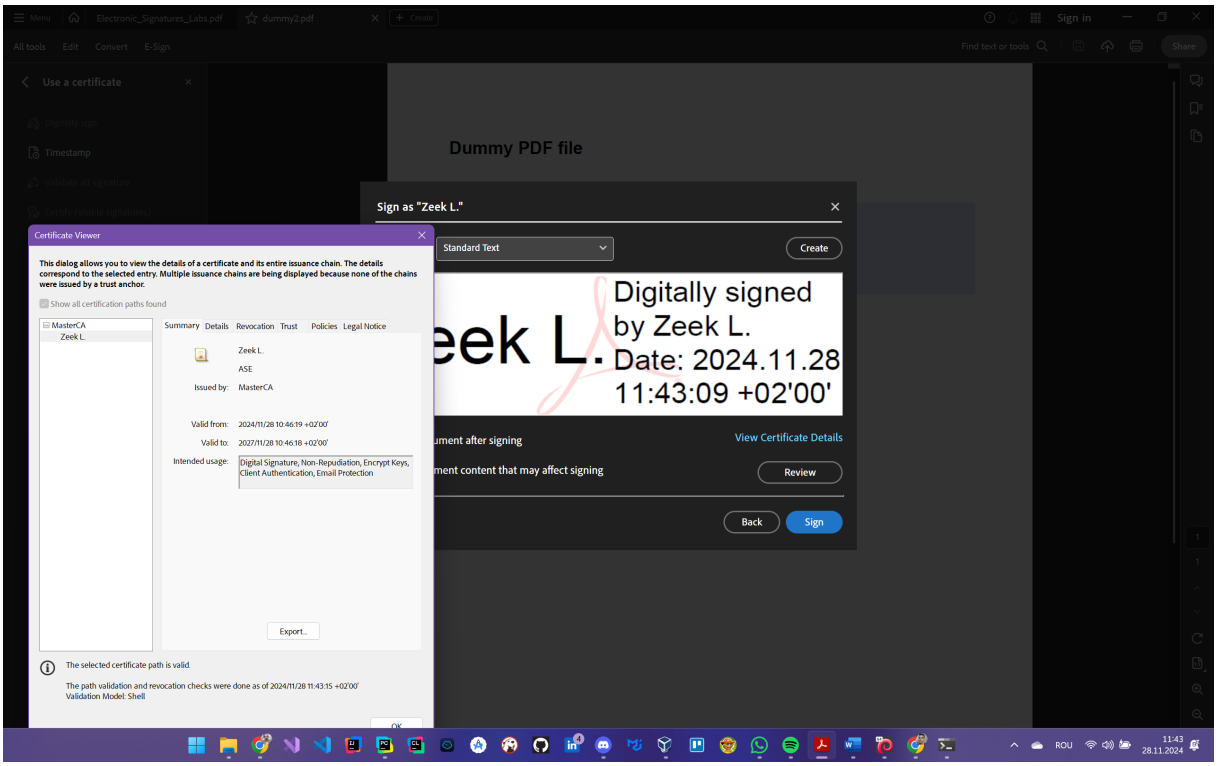


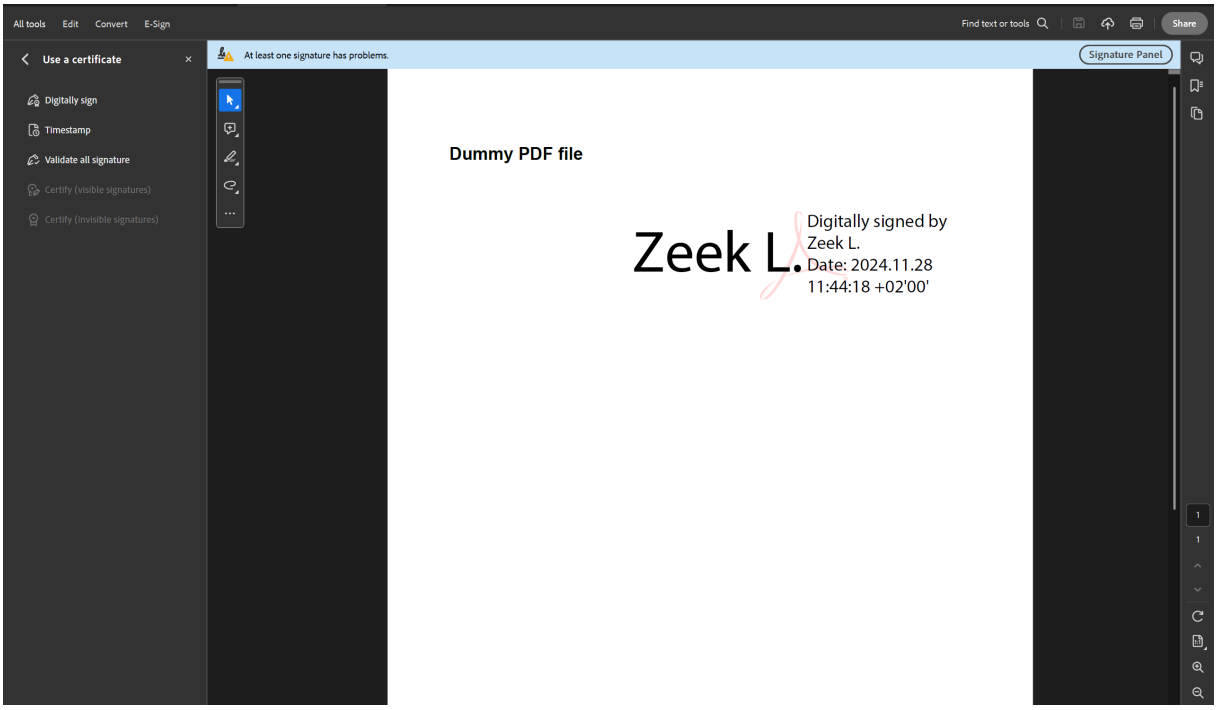
4 Import the certificate to Windows store

5 Try to sign a PDF file

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:

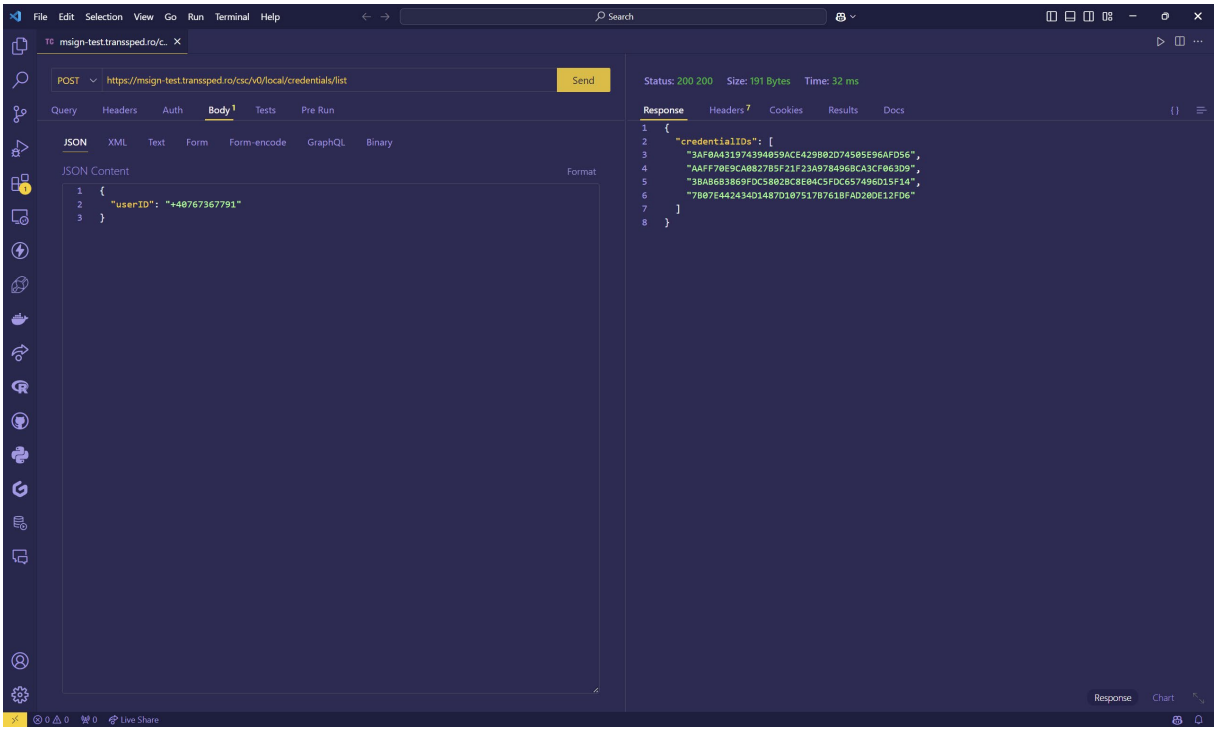




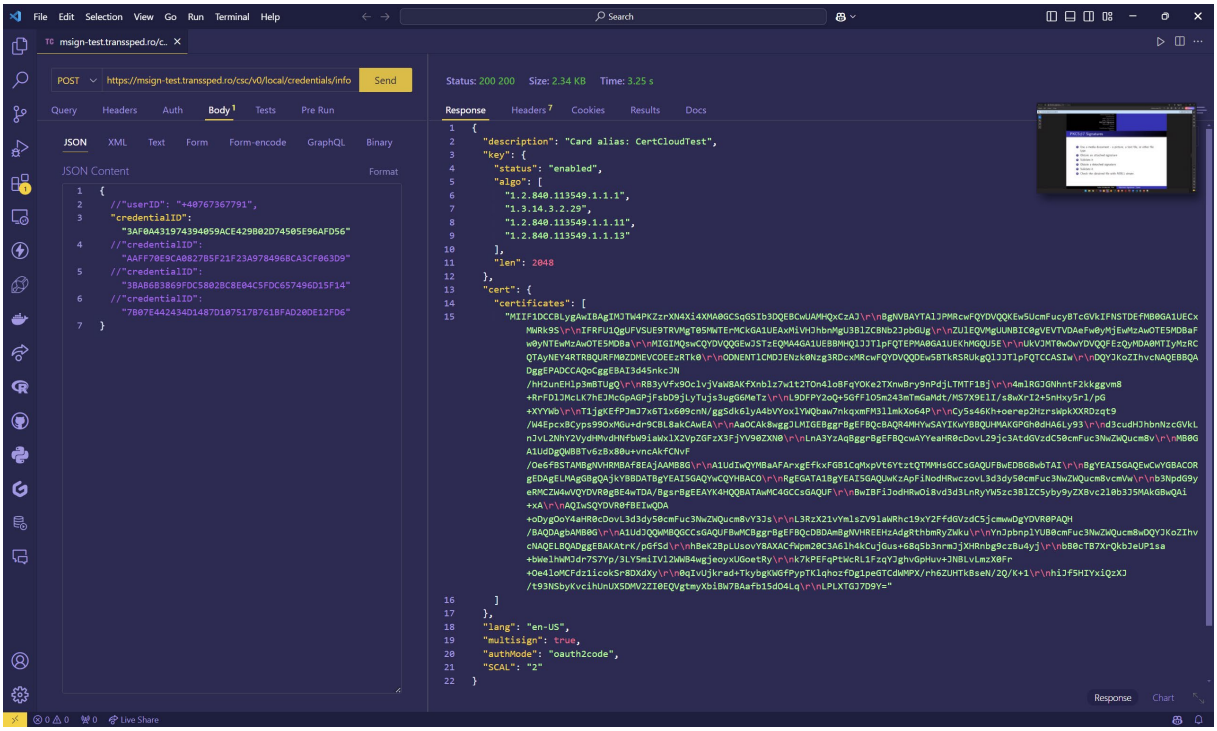
CSC Protocol – 'Hello world' level

Obtain one of the cloud certificates of the user with phone number +40767367791, using the CSC protocol.

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



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



And here we have our cloud certificate.

Bonus - Python Certificate Requests

The Python source code:

```python
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

- Write a python script that can create a PKCS#7 or an XML signature.

```
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
```

```python
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**.