Projeto 2+3 - Digital Rights Management

Segurança Informática e nas Organizações

Lucas Sousa - 93019

Tiago Oliveira - 93456

INDEX

1. Intro

1.1 Objectives

- Negotiate Cipher Suite
- Diffie-Hellman Key Exchange
- Encrypt Communications
- Authentication of Entities
- Protect Media at Rest

1.2 Cipher Suite

These are the fields negotiated:

- Algorithm - [“AES”, “CHACHA20”]
- Mode - [“CFB”,“OFB”,“CTR”]
- Hash - [“SHA-256”,“SHA-512”,“MD5”]

We picked AES and CHACHA20 because these were the algorithms with implemented code on cryptography.io thus saving us precious time.

The modes and hashes were picked based on what we already knew from the lab classes and from researching.

The negotiation of this cipher suite allowed for compatibility between server-client and, most importantly, allowed us to encrypt messages sent between these entities.

1.3 Diffie-Hellman Key Exchange

This method allowed us to securely exchange cryptographic keys publicly (allowing a third-party to witness the key exchange) between two parties that have no prior knowledge of each other by establishing a shared (secret) key.

1.4 Encrypted Communications

As stated, before the messages sent between server and client (after negotiating the cipher suite) are encrypted with the aid of the shared key thus allowing for encrypted communication.

To encrypt all the communications we decided to encrypt the whole json (after we construct it) and proceed to send the initial vector + encrypted json to be able to decrypt.

From the receiving side the first 16 bytes are read (initial vector/nonce) and it's used to decrypt the message which is then 'loaded' to be accessed

1.5 Authentication of Entities

Encrypting communications goes a long way to secure a session, but it's not foolproof. To further increase the security of a session we should validate the session's entities (in this case - server and client).

In this section we validate the server with it's certificate and we validate a client using hardware tokens. In this case, we used the Potuguese Citizen Card.

1.6 Viewing licenses

If the user wants to use the system and listen to music he needs a valid license for the occasion. To solve this problem we give the user a certificate and while that certificate is valid he can use the system to listen to some music.

1.7 Protect Media at Rest

To prevent outside sources from accessing the music files without permission we encrypted them so only the server knows how to decrypt them and access them.

2. Diffie-Hellman Key Exchange

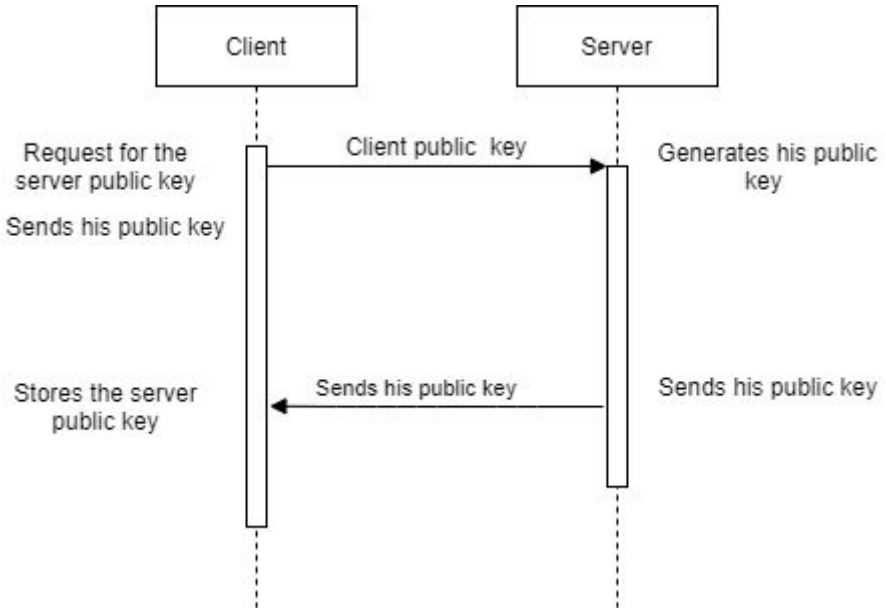


Diagram 1: Sequential diagram about Diffie-Hellman

```
print(
    "################################### DIFFIE-HELLMAN ###################################"
)

#----/ Generate Client Keys /----#
print('Generating keys...')
pubk, privk = generate_public_and_private_keys()

#----/ Send Public Key /----#
print('Sending public key to server')
server_params = send_pubk(pubk)

#----/ Build Server Public Key based on response parameters /----#
server_pubk = get_server_public_key(server_params)

#----/ Perform key exchange and derivation /----#
shared_key = exchange_keys(privk, server_pubk)
```

Fig.1 : Client side Diffie-Hellman

(this next section will be explained in a client-side view, bear in mind that the server has to do the same steps on his side)

We perform several steps to implement this method:

- Generate entity's private and public keys
- Send public key to other entity
- Get other entity's public key
- Perform key exchange to get shared (secret) key

We use 'get' to send the client's public key and at the same time get the server's public key by parsing the request's response.

Generating these keys is pretty straight-forward, but the problem lies in sending them to the server. Because the generated keys are python objects they cannot be sent directly in a message, so we 'dismantle' the key to get it's parameters numbers (p, g, y) and that's what we send over to the server. The server then re-builds the key with the received parameters.

So, by now we've sent the client's key parameters and through the response of the 'get' request we receive the server's parameters as well. So, the client must now build the server's public key with the received parameters to finally exchange the keys.

```
def exchange_keys(privk, server_pubk):
    """Perform key exchange and key derivation"""
    #----/ Exchange keys /----#
    print("Creating shared key...")
    shared_key = privk.exchange(server_pubk)

    #----/ Key Derivation /----#
    derived = HKDF(algorithm=hashes.SHA256(),
                   length=32,
                   salt=None,
                   info=b'handshake data',
                   backend=default_backend()).derive(shared_key)

    print("Derived key ", derived)
    return derived
```

Fig.2 : Function used to exchange keys between the client and the server

The shared key is derived to obtain a derived-shared-key. Key derivation further improves the secrecy of a key.

Diffie-Hellman is now complete and both server and client have a shared key that holds a common value for both of them.

3. Server Authentication

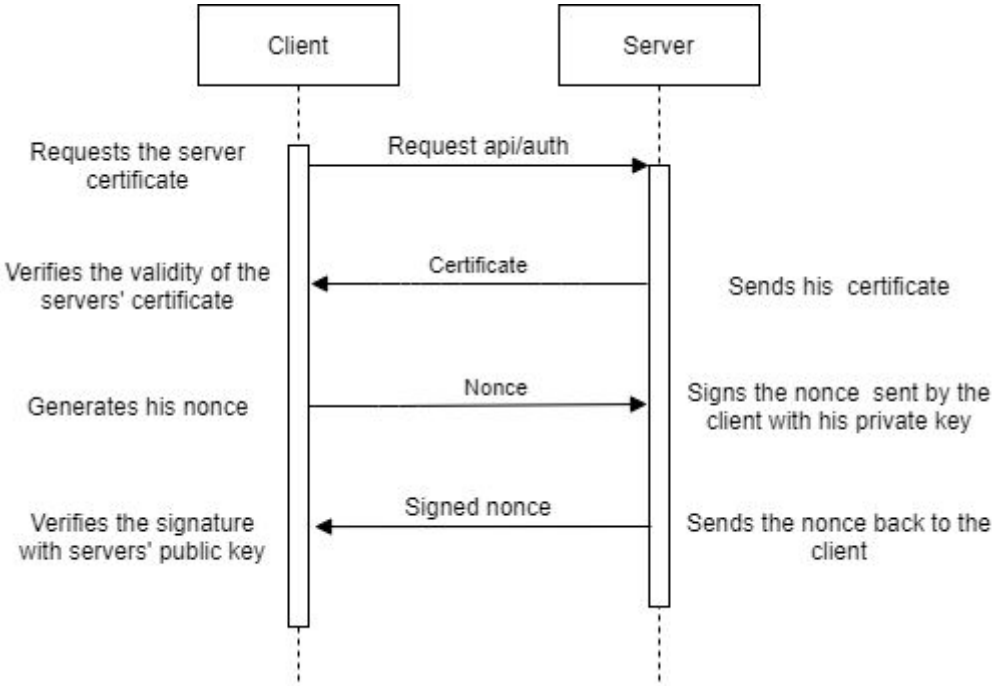


Diagram 2: Server authentication

Using the program XCA we managed to create a root CA (self-signed) and a certificate for the server which is validated by the CA.

Steps for the client to authenticate the server:

- Server sends his certificate to the client
- Client builds and verifies the validity of the certificate's chain
- Client generates and sends a nonce to the server
- Server signs the nonce (with private key) and sends the signature back to the client
- Client verifies the signature with the server's public key

If either one of these steps is invalid the process gets shut down.

```python
def valid_cert_chain(chain, cert, roots):
    chain.append(cert)
    issuer = cert.issuer
    subject = cert.subject

    # Quando chegar à root (em self-signed certificates o issuer é igual ao subject)
    if issuer == subject and subject in roots:
        return True

    elif issuer in roots:
        return valid_cert_chain(chain, roots[issuer], roots)

    else:
        print("Invalid Chain!")
        return False
```

Fig.3 : Function used to validate the certificate chain

To validate the chain, we iterate recursively every issuer of a certificate and when it finds a CA (*i.e* self-signed certificate) it validates the chain. If not, the chain is considered invalid.

4. Client Authentication

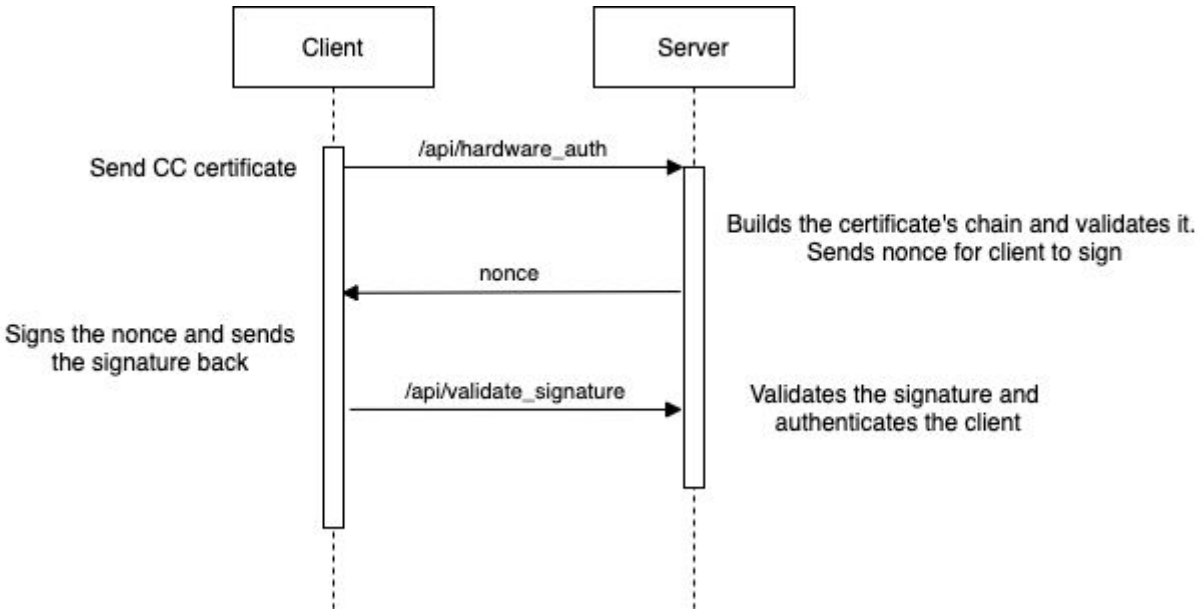


Diagram 3 - Client Authentication

Client authentication is done via a hardware token which, in this case, is the Portuguese citizen card (CC). This process is very similar to the server's authentication but instead of using a certificate created with XCA we use the certificate that came from the CC.
The code provided to read the CC and extract the information we needed came from the lab classes despite having made some changes to accommodate our needs.

We send the CC's certificate to the server via a 'post' request and the server loads it to a readable format. Next, we generate the certificate 'chain' - it's not properly a chain in our case, but a collection of lists that hold the 'ranks' or 'hierarchy' of our chain.
Meaning, we have several directories under 'cc_certs':

- etc/ssl/certs/
- CA
- cc
- intermediates

The list above is ordered by certificate hierarchy, which means that the CC certificate is validated by a certificate in the 'intermediates' -> this certificate that validates the CC certificate is validated by a certificate in 'cc' -> and it goes on and on until we reach the root certificates that are usually found in the OS 'etc/ssl/certs'.

We load all these certificates under these directories to independent lists and then, after the CC certificate is loaded on the server, we run it through the 'validate_cc_chain' function which checks if:

- In fact the certificate was issued by another certificate
- The certificate's dates are not expired
- The certificate doesn't belong to a CRL

If all this goes according to plan and the chain is valid, the server proceeds to send a nonce for the client to sign and thus granting him authenticity. The process is very similar to when the client sends his nonce but the code syntax differs a bit. We used the code from the lab classes to sign the nonce (client-side) and to verify the signature (server-side).

5. Viewing Licenses

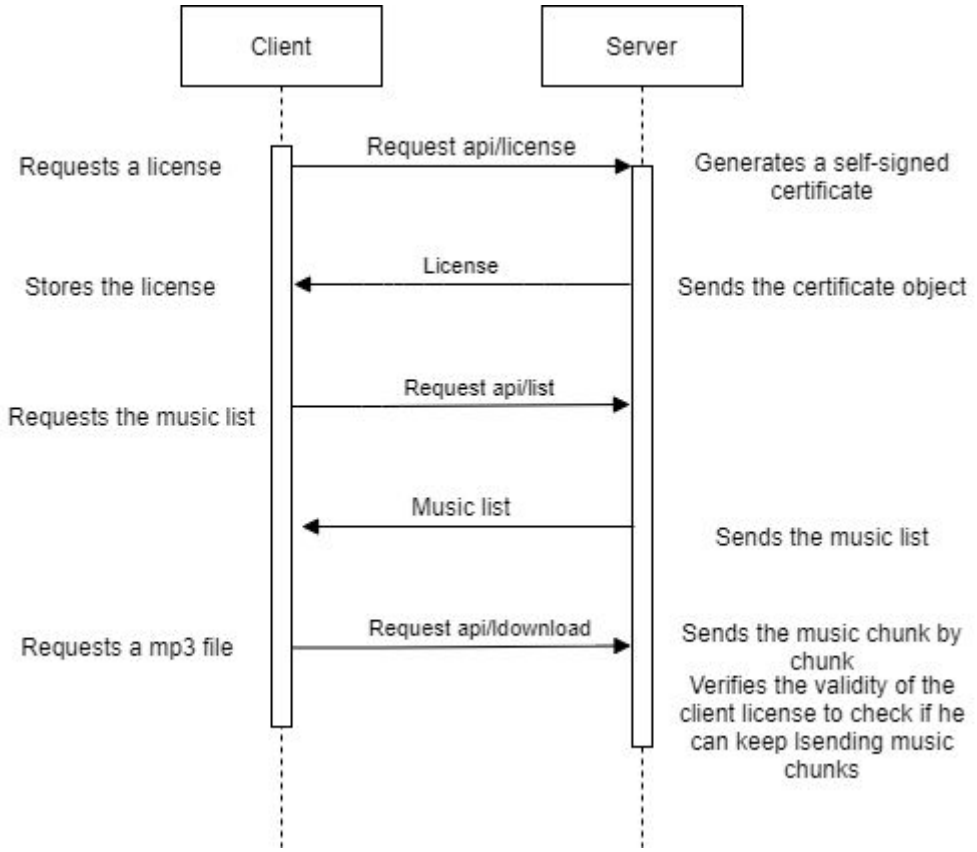


Diagram 3: Viewing licenses

To prevent the users from abusing the system, the server must know if the user can keep using it. So we used viewing licenses. In our case the license is a self-signed certificate.

After the client and the user negotiate with each other the protocols, the server creates a self signed certificate (signed with his private key so no one can replicate the signature). That certificate has an expiration date and until that date is valid the client can use the system.

From the server perspective before he sends a chunk he verifies if the date in the certificate(license) he gave to the client is still valid, if it is not he stops sending chunks and the client will be disconnected until he asks for a new license. The license we create in the server and send to the client has only 1 minute of validity so we could test it, we could change it to hours or even days by changing one parameter in the creation of the license.

```
#Verify if the client still has a valid license
with open("certs/Client_licence.pem","rb") as f:
    pem_data = f.read()
    f.close()

cli_license = x509.load_pem_x509_certificate(pem_data, default_backend())

license_val = cli_license.not_valid_after
now_date = datetime.datetime.now()
if now_date > license_val:
    request.setResponseCode(300)
    return b''
```

Fig.4 : Verification of the client license

6. Chunk by Chunk Authentication

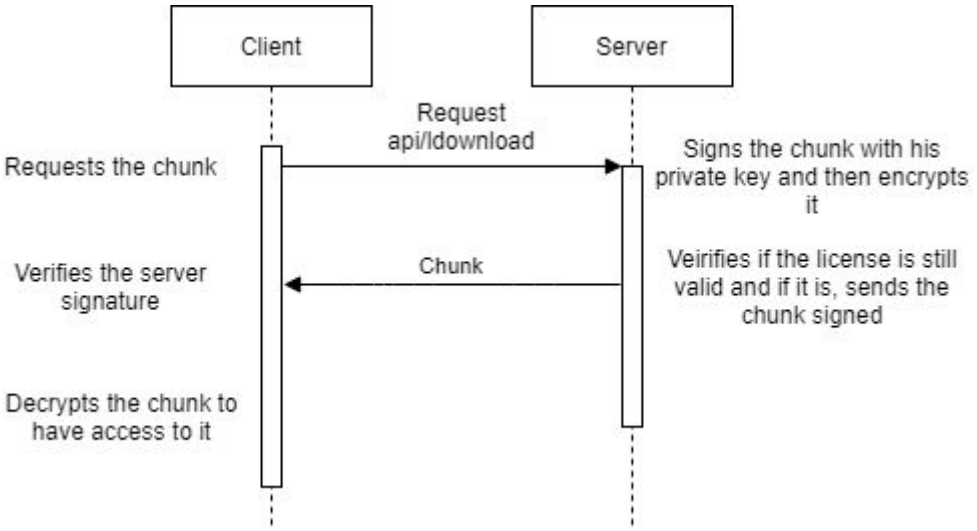


Diagram 4: Chunk by chunk authentication

As we explained we send the information to the client chunk by chunk and so to prevent men in the middle attacks we decided to authenticate the server chunk by chunk as well as encrypting each chunk. Before the server sends the chunk he signs it with his private key so the signature can not be copied and then he sends the signature concatenated with the encrypted chunk.

From the client's perspective as soon as he receives each chunk he splits it into the signature part and the encrypted chunk part. First of all he verifies if the signature is really from the server to prevent men in the middle attacks, and if it is valid then he decrypts the chunk and loads its content.

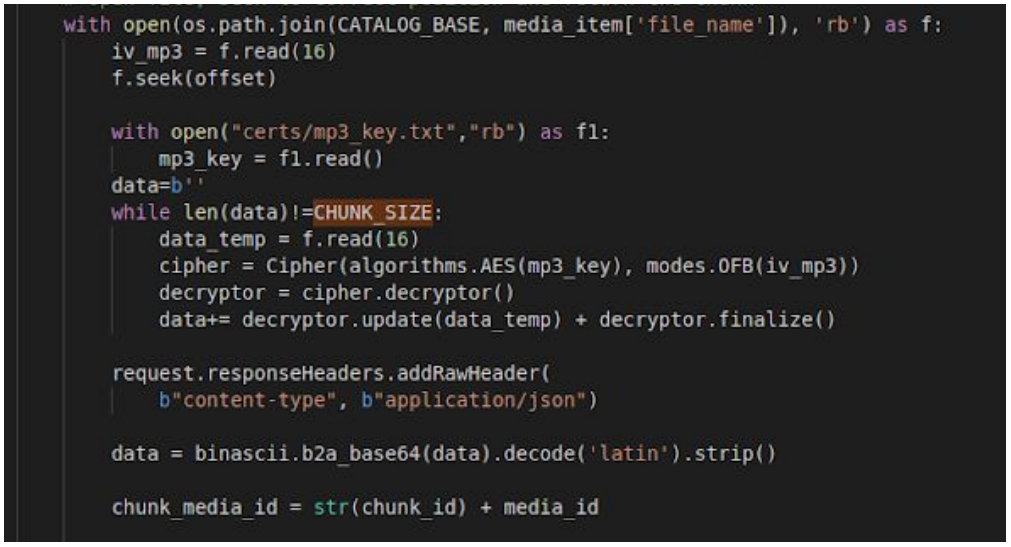
```
# Get data from server and send it to the ffplay stdin through a pipe
for chunk in range(media_item['chunks'] + 1):
    chunk_id = chunk

    req = requests.get(
        f'{SERVER_URL}/api/download?id={media_item["id"]}&chunk={chunk}')

    if req.status_code == 300:
        print("Not valid license!")
        exit()

    chunk = json.loads(req.content[256:])

    #Server signature
    signature = req.content[0:256]

    #Verify if is really is the server signature
    if server_cert_pubk.verify(
        signature,
        req.content[256:],
        padding.PSS(
            mgf=padding.MGF1(matched_hash),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        matched_hash
    ) is not None:
        exit(1)
```

Fig. 5 : Client splitting the chunk he receives and verifying the server signature

7. Encrypted Media at Rest

Earlier at this report we explained that the client needs to have a valid license to access the mp3 files on the server. To prevent him of trying to access them without a valid license we decided to encrypt the mp3 files in the disk.

To do it we made a python script 'encrypt_script.py' that would access the mp3 files and encrypt them chunk by chunk and write them in a new file. We also generated a random key(mp3_key.pem) that would be used to decrypt the file and placed it in a safe file. So we could decrypt the mp3 files we would need the initial vector as we used AES with the OFB mode and we decided that the first 16 bytes of the encrypted mp3 file was going to be the initial vector. After we encrypted the mp3 files once a client would ask to access them with a valid license we needed to decrypt them chunk by chunk and then encrypt the chunk again with the agreed protocols to finally send it to the client. As the chunk size was 4096 bytes and we encrypted the mp3 files with chunks of 16, we had to decrypt a few mp3 files chunks to satisfy the 4096 chunk size to send the client.



```
with open(os.path.join(CATALOG_BASE, media_item['file_name']), 'rb') as f:
    iv_mp3 = f.read(16)
    f.seek(offset)

    with open("certs/mp3_key.txt","rb") as f1:
        mp3_key = f1.read()
    data=b''
    while len(data)!=CHUNK_SIZE:
        data_temp = f.read(16)
        cipher = Cipher(algorithms.AES(mp3_key), modes.OFB(iv_mp3))
        decryptor = cipher.decryptor()
        data+= decryptor.update(data_temp) + decryptor.finalize()

    request.responseHeaders.addRawHeader(
        b"content-type", b"application/json")

    data = binascii.b2a_base64(data).decode('latin').strip()

    chunk_media_id = str(chunk_id) + media_id
```

Fig.6 : Decrypting mp3 file stored and encrypting with the client agreed protocols