Segurança Informática e nas Organizações

João Paulo Barraca

2nd and 3rd Projects: Digital Rights Management

Rui Fernandes, 92952
Pedro Bastos, 93150

DETI
Universidade de Aveiro
January 2, 2021

Contents

1 Introduction

This document intends to describe our solution to the proposed task in these 2 projects.

We were asked to plan and implement a protocol that allows safe communications, therefore confidential and integrate, between a server and a client trough an API with the objective of securely distributing media files.

In a second part we had to enhance our solution, implementing Authentication mechanisms to it.

2 Confidentiality and Integrity

2.1 Objectives

For this first part of the project we were challenged to create a protocol that establishes a safe session between the client and server, these we the requirements in the guide provided:

1. We had to negotiate a cipher suite between the server and client, given this, our code had to support at least 2 symmetric ciphers, 2 digests and 2 cipher modes.
2. Negotiate and exchange ephemeral keys between both parties, in our case using the Diffie Hellman key exchange.
3. Encrypting all further communications, to allow confidentiality
4. Implement Integrity checking of all communications.
5. Include a viewing license system, in our case a limit on number of views.
6. Implementation of a key rotation mechanism after a certain volume of data is exchanged.

2.2 Message Flow

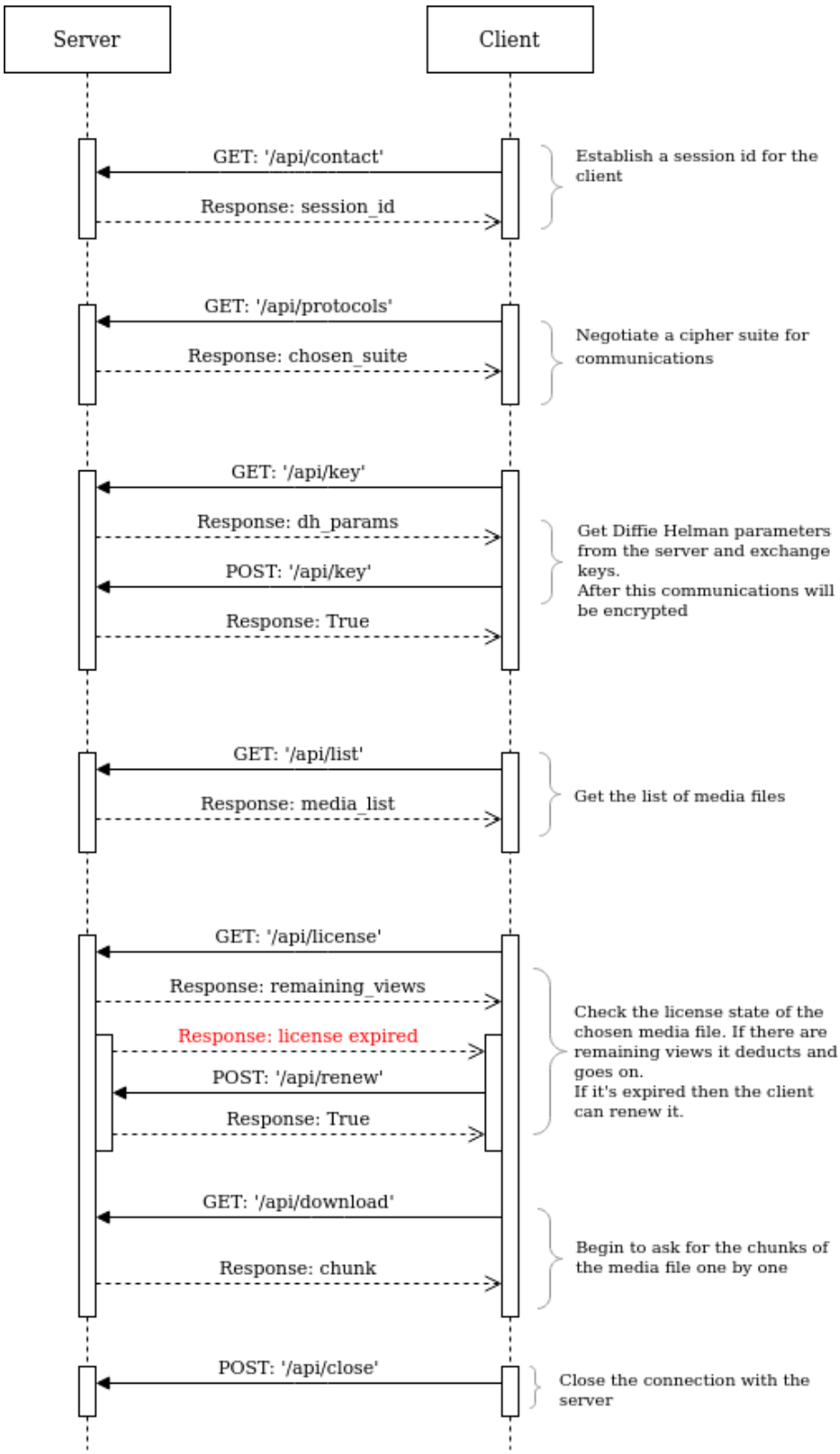
To incorporate all the points mentioned, we needed a message flow planned, this was altered along the way, but the following diagram will show the current generic flow of communication between the two parties.

Its important to note a few aspects, the communication is split into obviously depicted phases:

1. Initial contact of the client to the server, allows the server to create a session corresponding to said client and send the respective ID.
2. Negotiation of a cipher suite supported by both parties.
3. Diffie Hellman key exchange, triggered by the client.
4. Acquirement of the list of available media by the client.
5. After choosing a media file, the client will star the process of downloading from the server. This is preceded by a step where the server checks the validity of the client's license, two outcomes can arise, its still valid and the client can listen again, or it has expired and needs renewal. The download itself is done in chunks, each one comprising a GET by the client and the chunk in a response by the server.
6. Finally the client chooses to disconnect and sends a POST informing the server.

It's also relevant to mention that after step 3 above, most communication between the parties is encrypted according to the chosen cipher suite and the exchanged keys. Additionally, the encrypted messages include a MAC to ensure integrity.

NOTE: The code displayed in this first half of the work will highlight the relevant portions relating to Confidentiality and Integrity, ignoring the Authentication portion that will come afterwards. This being said some code may not entirely correspond to the finished product in the python scripts.


Server
Client
GET: '/api/contact'
Response: session_id
Establish a session id for the client
GET: '/api/protocols'
Response: chosen_suite
Negotiate a cipher suite for communications
GET: '/api/key'
Response: dh_params
POST: '/api/key'
Response: True
Get Diffie Helman parameters from the server and exchange keys.
After this communications will be encrypted
GET: '/api/list'
Response: media_list
Get the list of media files
GET: '/api/license'
Response: remaining_views
Response: license expired
POST: '/api/renew'
Response: True
Check the license state of the chosen media file. If there are remaining views it deducts and goes on.
If it's expired then the client can renew it.
GET: '/api/download'
Response: chunk
Begin to ask for the chunks of the media file one by one
POST: '/api/close'
Close the connection with the server

2.3 Negotiation

After the first contact and the session ID being established, the cipher suite negotiation will begin, the client sends a GET request were it includes its sessionID and the suites it supports.

Code Snippet 1: Client's supported suites

```
class Client():
    def __init__(self):
        self.client_suites = ['DH_AES128_CBC_SHA256', 'DH_CHACHA20_SHA256']
```

Code Snippet 2: Negotiation GET request

```
req = requests.get(f'{SERVER_URL}/api/protocols?sessionID={json.dumps(self.
session_id)}&suites={json.dumps(self.client_suites)}')
req = req.json()
self.chosen_suite = req['chosen_suite']
```

On the server side, receiving the client's suites it will check if it supports any and choose the one it deems safest, this is done by simply getting the first from the server list that the client supports, this list in on line 73 and is ordered by general safety.

Code Snippet 3: Server choosing suite

```
suite_list = request.args.get(b'suites', [None])[0].decode('latin')
suite_list = json.loads(suite_list)

session_id = json.loads(request.args.get(b'sessionID', [None])[0].decode('latin'))
session = self.open_sessions[session_id]

chosen_suite = None
for s in self.SERVER_SUITES:
    if s in suite_list:
        chosen_suite = s
        params = s.split('_')
        session['cipher'] = params[1]
        if len(params)==4:
            session['mode'] = params[2]
            session['digest'] = params[3]
        else:
            session['digest'] = params[2]
            session['mode'] = ''
        break
session['suite'] = chosen_suite
```

The Server (and the client) supports the following cipher suites:

- DH_CHACHA20_SHA384
- DH_CHACHA20_SHA256
- DH_AES128_GCM_SHA384
- DH_AES128_GCM_SHA256
- DH_AES128_CBC_SHA384
- DH_AES128_CBC_SHA256

All the suites rely on the Diffie Hellman key exchange algorithm (DH), the two symmetric ciphers we have are AES 128-bit and CHACHA20. The modes, which are only relevant for AES128 are Cipher Block Chaining and Galois/-Counter Mode. Finally the digest we have are SHA384 and SHA256.

2.4 Diffie Hellman

After the previous step the client initiates the DH key exchange process, starting with a GET request to '/api/key. The server will generate parameters and a private and public key of its own, sending the parameters and public component of the key as a response to the client.

Code Snippet 4: Server side DH keys

```
def do_dh_keys(self, request):
    session_id = json.loads(request.args.get(b'sessionID', [None])[0].
        decode('latin'))
    session = self.open_sessions[session_id]

    parameters = dh.generate_parameters(generator=2, key_size=512,
        backend=default_backend())

    session['private_key'] = parameters.generate_private_key()

    peer_public_key = session['private_key'].public_key()
    p = parameters.parameter_numbers().p
    g = parameters.parameter_numbers().g

    session['public_key'] = peer_public_key.public_numbers().y

    session['p'] = p
    session['g'] = g

    request.responseHeaders.addRawHeader(b"content-type", b"application/json")
    return json.dumps((p,g,session['public_key']), indent=4).encode('latin')
```

Receiving these parameters the client will generate his own keys, public and private, and further it will generate a shared key and derive it to get a usable symmetric key.

Afterwards it will send his public key to the server in a POST request so that the server can do the same process of exchanging keys and deriving the result

Code Snippet 5: Client GET and POST to /api/key

```
def hande_dh(self):
    req = requests.get(f'{SERVER_URL}/api/key?sessionID=
␣␣␣␣␣␣␣␣{json.dumps(self.session_id)}')

    dh_params = req.json()

    self.DH_make_keys(dh_params[0],dh_params[1],dh_params[2])

    req = requests.post(f'{SERVER_URL}/api/key', data={'sessionID':
        self.session_id, 'pubkey':self.public_key})
```

Code Snippet 6: Client Side key generation

```
def DH_make_keys(self,p,g,server_key):
    pnum = dh.DHParameterNumbers(p, g)
    parameters = pnum.parameters(default_backend())
    self.private_key = parameters.generate_private_key()

    peer_public_key = self.private_key.public_key()

    self.public_key = peer_public_key.public_numbers().y

    self.shared_key = self.private_key.exchange(dh.DHPublicNumbers
        (server_key,pnum).public_key())

    self.gen_symmetric_key()

def gen_symmetric_key(self):

    if(self.DIGEST=="SHA256"):
        algorithm=hashes.SHA256()
    elif(self.DIGEST=="SHA384"):
        algorithm=hashes.SHA384()

    key = HKDF(
        algorithm=algorithm,
        length=32,
        salt=None,
        info=b'handshake␣data',
        backend=default_backend()
    ).derive(self.shared_key)

    if self.CIPHER == 'AES128':
        self.symmetric_key = key[:16]
    elif self.CIPHER == 'CHACHA20':
        self.symmetric_key = key[:32]
```

NOTE: Server side generation of the shared and symmetric keys is done in the methods gen_shared_key and gen_symmetric_key which are almost identical to the ones above.

2.5 Confidentiality

Now that both parties have their symmetric keys, all content sent in responses from the server as well as data sent in client's POST requests are fully encrypted to allow for confidentiality.

To encrypt a message it need to go through two methods, secure() and encryption(), the former is meant to re format the message and also calls on integrity control, the latter actually encrypts the message according to the chosen cipher suite.

The methods that mirror the previous two and correspond to the decryption of the encrypted data are extract_content() and decryption()

For simplicity we will only show some relevant snippets of code to serve as an example, the actual methods can be seen on the python scripts, it's important to note that the Client and Server versions of these methods are slightly different as they save their ciphers, digests, etc in disparate ways.

Code Snippet 7: Data encryption

```
content = {'media_list': media_list}
secure_content = self.secure(content, session)

def secure(self, content, session):
    secure_content = {'payload': None}
    payload = json.dumps(content).encode()

    criptogram,iv,nonce,tag = self.encryption(payload, session)
    secure_content['payload'] = base64.b64encode(criptogram).decode()
    ...
    return secure_content
```

Code Snippet 8: Data decryption

```
req = requests.get(f'{SERVER_URL}/api/list?sessionID={json.dumps
    (self.session_id)}')
secure_content = req.json()
media_list = self.extract_content(secure_content)['media_list']

def extract_content(self, secure_content):
    iv = base64.b64decode(secure_content['iv'])
    tag = base64.b64decode(secure_content['tag'])
    nonce = base64.b64decode(secure_content['nonce'])
    payload = base64.b64decode(secure_content['payload'])
    ...
    return json.loads(self.decryption(payload,iv,nonce,tag))
```

2.6 Integrity Control

In the methods secure() and extract_content() mentioned above we also include integrity control, since every message received needs to have it's integrity verified before it's decrypted.

We followed the Encrypt-then-MAC approach so the message sender will generate a MAC with it's shared key and corresponding cipher suite, then send the MAC alongside the encrypted message and the possible iv, nonce and/or tag. the receiving end a MAC will also be generated and it will be compared to the received MAC ensuring integrity.

Code Snippet 9: MAC generation

```
def secure(self, content, session):
    ...
    mac = self.make_MAC(criptogram, session)
    secure_content['MAC'] = base64.b64encode(mac).decode()
    ...
    return secure_content

def make_MAC(self, data, session):

    if(session['digest']=="SHA256"):
        h = hmac.HMAC(session['symmetric_key'], hashes.SHA256(),
            backend=default_backend())
    elif(session['digest']=="SHA384"):
        h = hmac.HMAC(session['symmetric_key'], hashes.SHA384(),
            backend=default_backend())

    h.update(data)

    return binascii.hexlify(h.finalize())
```

Code Snippet 10: MAC validation

```
def extract_content(self, secure_content):
    ...
    mac = base64.b64decode(secure_content['MAC'])

    if self.check_MAC(mac, payload):
        print("Message passed Integrity check")
    else:
        print("Message failed Integrity check, Shutting Down...")
        self.disconnect()
    ...

def check_MAC(self, server_mac, data):
    client_mac = self.make_MAC(data)

    if client_mac == server_mac:
        return True
    else:
        return False
```

2.7 License Management

Another objective for this project was to implement a system that would limit the viewing of each client, in our case we limited it by number of views.

A client needs a viewing license for each media file, any given license as a validity of 5 views, when a client first asks for a file a license is emitted for that file.

Whenever the client asks to download a certain file, it first checks the license validity and if valid subtracts a view from it. When the first and consecutive licenses expire, the server will warn the client and block it from viewing said file unless the client renews its license, this is simulated with a prompt asking for payment, but it's just a simple yes or no prompt.

If the client chooses to renew the license the server will once again fill it to 5 views and proceed.

Code Snippet 11: Client checking license

```
media_item = media_list[selection]

req = requests.get(f'{SERVER_URL}/api/license?sessionID={json.dumps
␣␣␣␣(self.session_id)}&id={media_item["id"]}')
if req.status_code == 402:
    if self.renew_license(media_item["id"]):
        views = 4
    else:
        return
else:
    views = self.extract_content(req.json())
```

NOTE: the variable views is not stored by the client, it's simply the remaining views of said media item and it's displayed for the client's information

Code Snippet 12: Client license renewal

```
def renew_license(self, media_item):
    while True:
        print(...)
        selection = input("(Y)es/(N)o:␣")
        if selection.strip() == 'Y':
            data = self.secure({'id': media_item})
            data['sessionID'] = self.session_id
            req = requests.post(f'{SERVER_URL}/api/renew', data=data)
            return True
        elif selection.strip() == 'N':
            return False
        else:
            continue
```

NOTE: print prompt in not shown because of LaTeX problems.

Code Snippet 13: Server checking license

```
def check_license(self, request):
    ...
    if media_id not in session['licenses']:
        session['licenses'][media_id] = 4
    else:
        if session['licenses'][media_id] < 1:
            request.setResponseCode(402)
            request.responseHeaders.addRawHeader(b"content-type",
                b"application/json")
            return json.dumps(self.secure({'error':
                'license for this media expired'}, session)).encode('latin')
        else:
            session['licenses'][media_id]-=1

    request.setResponseCode(200)
    request.responseHeaders.addRawHeader(b"content-type", b"application/json")
    return json.dumps(self.secure(session['licenses'][media_id], session),
        indent=4).encode('latin')
```

Code Snippet 14: Server renewing license

```
def renew_license(self, request):
    ...
    secure_data = request.args
    data = self.extract_content(secure_data)

    media_id = data['id']

    session['licenses'][media_id] = 4

    request.responseHeaders.addRawHeader(b"content-type", b"application/json")
    return json.dumps(True, indent=4).encode('latin')
```

NOTE: the views are set to 4 because it takes into account the view that is already set to play

2.8 Key rotation

To ensure an additional layer of safety, it is necessary to implement a chunk based key rotation mechanism ensuring that the parties switch their keys after a certain amount of chunks have been downloaded by the client.

This was fairly simple to implement as the server already transmitted the files in chunks. All we needed to do was keep a chunk download count for each client in the server and have a flag in the response to any GET request for download that will signal if key rotation is needed.

Once rotation is needed both parties will hold download progress and will completely redo the Diffie Hellman key exchange protocol, getting new keys and exchanging them before continuing the download.

Code Snippet 15: Server checking for key rotation in do_download() method

```
session['download_count']+=1

if session['download_count']==500:
    logger.debug(f'Reached 500 chunk downloads, requesting new key exchange')
    needs_rotation = True
    session['download_count'] = 0
else:
    needs_rotation = False

request.responseHeaders.addRawHeader(b"content-type", b"application/json")
return json.dumps(self.secure(
        {
            'media_id': media_id,
            'chunk': chunk_id,
            'data': binascii.b2a_base64(data).decode('latin').strip(),
            'needs_rotation': needs_rotation
        }, session),indent=4
    ).encode('latin')
```

Code Snippet 16: Server checking for key rotation in do_download() method

```
for chunk in range(media_item['chunks']):

    req = requests.get(f'{SERVER_URL}/api/download?sessionID={json.dumps(self.
␣␣␣␣␣␣␣␣session_id)}&id={media_item["id"]}&chunk={chunk}')

    chunk = self.extract_content(req.json())

    data = binascii.a2b_base64(chunk['data'].encode('latin'))
    try:
        proc.stdin.write(data)
    except:
        break

    if chunk['needs_rotation']==True:
        self.hande_dh()
```

3 Authentication and Isolation

3.1 Objectives

In this part of the project, it is intended to authenticate both the server and the client, as well as the music content and the user. The following requirements were expected:

1. Mutually authenticate the client and the server supported by a custom PKI
2. Authenticate the user that is viewing the media content
3. Authenticate the content itself, so that the client can trust it
4. Integrate hardware tokens to authenticate users (Citizen's Card)
5. Protect the media content at rest in the server

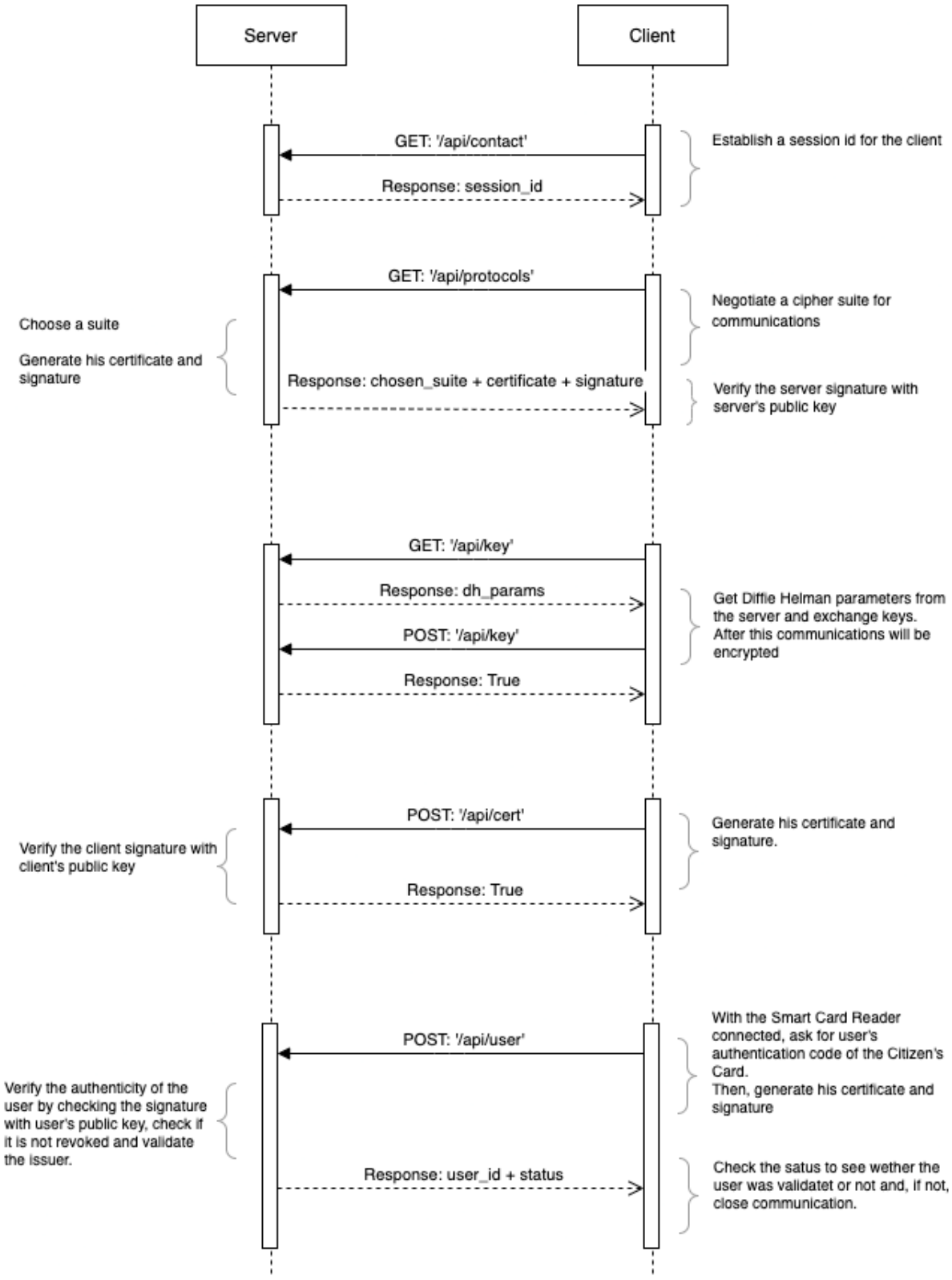
The objective is to create a chain of trusted identities so that they can communicate safely. The XCA application was used to create all private keys and CAs, as well as the certificates. To authenticate the user, the Smart Card Reader was used to read the Citizen's Card.

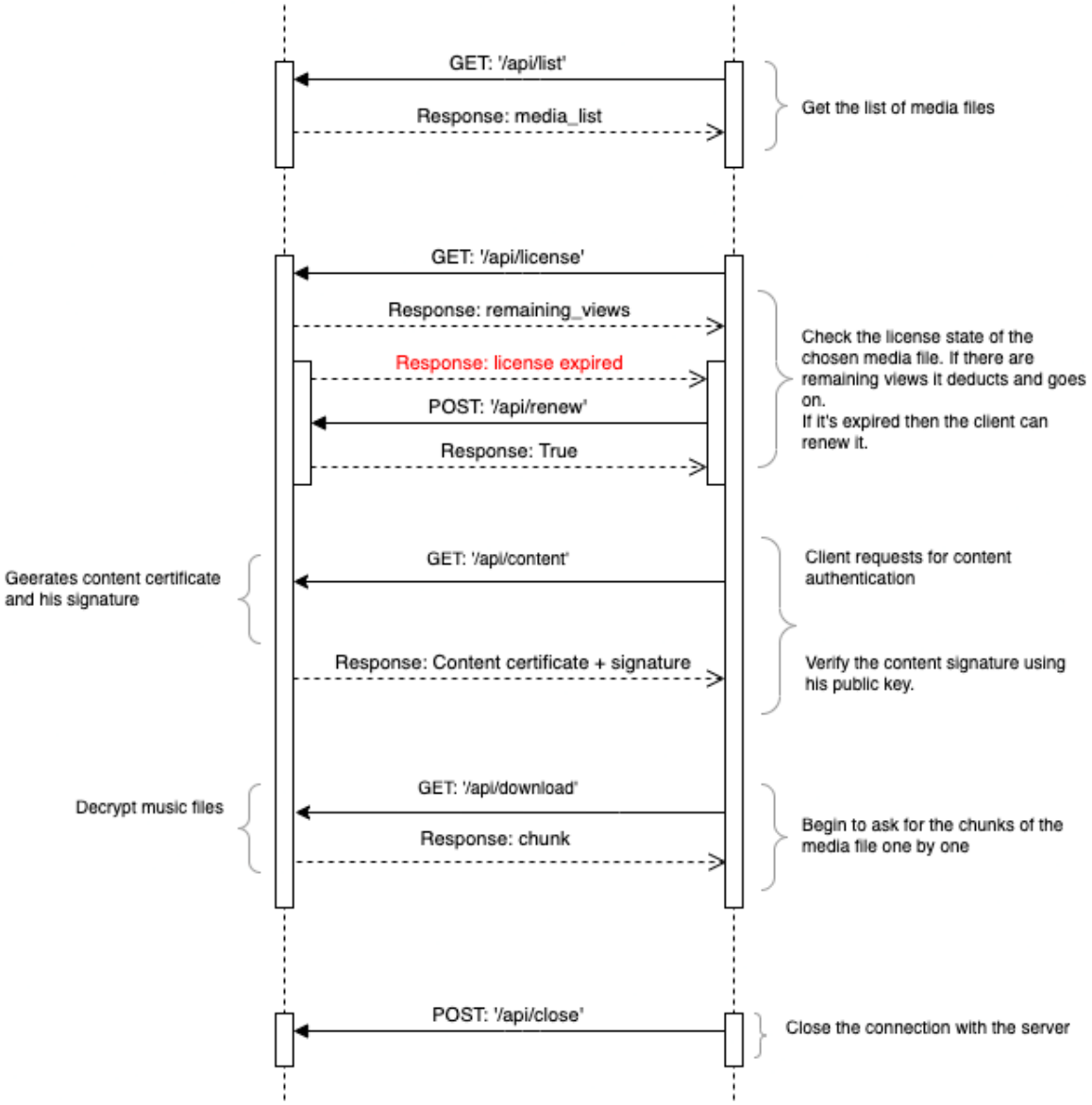
3.2 Message Flow (With both parts)

As the first part was already explained in the first message flow diagram, we will only explain the authentication part:

1. Server generates his certificate and signature and sends it to the client.
2. Client verifies the signature with the server's public key.
3. Client generates his certificate and signature and sends it to the server.
4. Server verifies the signature with the client's public key.
5. Client reads the Citizen's Card and sends its info (certificate, signature) to server.
6. Server verifies the CC's info to check if it is valid and responds with the user id.
7. Before downloading the chunks, the Client requests for a content authentication.
8. Server generates content certificate and signature and sends it to the Client.
9. Client checks the content signature validation with its public key.
10. Before sending chunks, the server decrypts the media file.

With these items, the full diagram of the message flow looks like this:


Server
Client
GET: '/api/contact'
Establish a session id for the client
Response: session_id
GET: '/api/protocols'
Negotiate a cipher suite for communications
Choose a suite
Generate his certificate and signature
Response: chosen_suite + certificate + signature
Verify the server signature with server's public key
GET: '/api/key'
Response: dh_params
Get Diffie Helman parameters from the server and exchange keys. After this communications will be encrypted
POST: '/api/key'
Response: True
POST: '/api/cert'
Generate his certificate and signature.
Verify the client signature with client's public key
Response: True
POST: '/api/user'
With the Smart Card Reader connected, ask for user's authentication code of the Citizen's Card.
Then, generate his certificate and signature
Verify the authenticity of the user by checking the signature with user's public key, check if it is not revoked and validate the issuer.
Response: user_id + status
Check the satus to see wether the user was validatet or not and, if not, close communication.


GET: '/api/list'
Response: media_list
Get the list of media files
GET: '/api/license'
Response: remaining_views
Response: license expired
POST: '/api/renew'
Response: True
Check the license state of the chosen media file. If there are remaining views it deducts and goes on.
If it's expired then the client can renew it.
Geerates content certificate and his signature
GET: '/api/content'
Client requests for content authentication
Response: Content certificate + signature
Verify the content signature using his public key.
Decrypt music files
GET: '/api/download'
Response: chunk
Begin to ask for the chunks of the media file one by one
POST: '/api/close'
Close the connection with the server

The server debug looks like this:

```
Server started
URL is: http://IP:8080
[server.py:396 -           render_GET() ] Received request for b'/api/contact'
[server.py:396 -           render_GET() ] Received request for b'/api/protocols?sessionID=0&suites=%5B%22DH_AES128_CBC_SHA256%22%5D'
[server.py:105 -     do_get_protocols() ] Negotiate: args: {b'sessionID': [b'0'], b'suites': [b'["DH_AES128_CBC_SHA256"]']}
[server.py:109 -     do_get_protocols() ] Negotiate: suites: b'["DH_AES128_CBC_SHA256"]'
[server.py:133 -     do_get_protocols() ] Chosen suite: DH_AES128_CBC_SHA256
[server.py:165 -           do_dh_keys() ] server public key: 1234082339960855892923058148577462800389997531861694605504546999741130102665869831244187070012992484243696898608468507180700368874779775133016
[server.py:396 -           render_GET() ] Received request for b'/api/key?sessionID=0'
[server.py:436 -          render_POST() ] Received POST for b'/api/key'
[server.py:436 -          render_POST() ] Received POST for b'/api/cert'
[server.py:580 -      extract_content() ] Message passed Integrity check
[server.py:203 -         check_client() ] Client certificate validated successfully.
[server.py:436 -          render_POST() ] Received POST for b'/api/user'
[server.py:580 -      extract_content() ] Message passed Integrity check
[server.py:396 -           render_GET() ] Received request for b'/api/list?sessionID=0'
```

And the client debug looks like this:

```
|--------------------------------------|
|          SECURE MEDIA CLIENT         |
|--------------------------------------|

Contacting Server
Contacted Server!
Ended Negotiation
Server Certificate validated successfully.
Got parameters for DH keys
Exchanged keys
Sending client certificate to server ...
Client certificate successfully verified by server.
Wait for authentication app to open and introduce your authentication code ...
Message passed Integrity check
User BI303141018 validated successfully.
Got Server List
Message passed Integrity check
MEDIA CATALOG

0 - Sunny Afternoon - Upbeat Ukulele Background Music
1 - Never Gonna Give You Up - Rick Astley
----
Select a media file number (q to quit):
```

3.3 Server authentication

First of all, using XCA, we created a private key (RSA, 4096 bits) as the server's private key. Then, we generated the CA using that same key to sign it. In the server side, both are opened and read. Then, we generated the signature. For this, we used the server's private key to sign. As the data, we chose to use the DH parameters so that the client can verify it and make sure that it is valid. On the client side, he receives the information and validates the signature using the server's public key. If some attacker tries to change the info in the middle of the communication, the validation will fail and the program closes.

```
# server signature
signature = self.sign(session['suite'], str(dh_params[2]).encode() + str(dh_params[0]).encode() + str(dh_params[1]).encode())
```

Figure 1: Sign call

```
def sign(self, suite, data):
    # if SHA384 is used
    if "SHA384" in suite:
        signature = SERVER_PK.sign(
            data,
            padding.PSS(
                mgf = padding.MGF1(hashes.SHA384()),
                salt_length = padding.PSS.MAX_LENGTH
            ),
            hashes.SHA384()
        )

    # if SHA256 is used
    elif "SHA256" in suite:
        signature = SERVER_PK.sign(
            data,
            padding.PSS(
                mgf = padding.MGF1(hashes.SHA256()),
                salt_length = padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )

    return signature
```

Figure 2: Signing function

```python
def check_sign(self, req):
    req = req.json()

    # data
    pubkey = int(req['y'])
    p = int(req['p'])
    g = int(req['g'])

    # load server certificate
    cert = x509.load_pem_x509_certificate(req['certificate'].encode())

    SERVER_PUBLIC_KEY = cert.public_key()
    self.chosen_suite = req['chosen_suite']

    # verify server signature
    if "SHA256" in self.chosen_suite:
        hash_mode = hashes.SHA256()
    elif "SHA384" in self.chosen_suite:
        hash_mode = hashes.SHA384()

    try:
        SERVER_PUBLIC_KEY.verify(
            req['signature'].encode('latin'),
            str(pubkey).encode() + str(p).encode() + str(g).encode(),
            padding.PSS(
                mgf = padding.MGF1(hash_mode),
                salt_length = padding.PSS.MAX_LENGTH
            ),
            hash_mode
        )
    except:
        raise Exception('INVALID SIGNATURE FROM SERVER CERTIFICATE.')
        self.disconnect()
```

Figure 3: Client check Server sign function

3.4 Client authentication

For the client authentication we used the same method. Using XCA, we created a private key (RSA, 4096 bits) as the client's private key. Then, we generated the CA using that same key to sign it. In the client side, both are opened and read. Then, we generated the signature. For this, we used the client's private key to sign. As the data, we chose to use the client's public key so that the server can verify it and make sure that it is valid. On the server side, he receives the information and validates the signature using the client's public key. Again, it is protected against man in the middle attacks.

```
# client signature
client_sign = self.make_sign(self.chosen_suite, str(self.public_key).encode())
```

Figure 4: Sign call

```
def make_sign(self, suite, data):
    # if SHA384 is used
    if "SHA384" in suite:
        signature = CLIENT_PK.sign(
            data,
            padding.PSS(
                mgf = padding.MGF1(hashes.SHA384()),
                salt_length = padding.PSS.MAX_LENGTH
            ),
            hashes.SHA384()
        )

    # if SHA256 is used
    elif "SHA256" in suite:
        signature = CLIENT_PK.sign(
            data,
            padding.PSS(
                mgf = padding.MGF1(hashes.SHA256()),
                salt_length = padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )

    return signature
```

Figure 5: Signing function

```
def check_sign(self, signature, suite, pubkey, data):
    # if SHA384 is used
    if "SHA384" in suite:
        pubkey.verify(
            signature,
            data,
            padding.PSS(
                mgf = padding.MGF1(hashes.SHA384()),
                salt_length = padding.PSS.MAX_LENGTH
            ),
            hashes.SHA384()
        )

    # if SHA256 is used
    elif "SHA256" in suite:
        pubkey.verify(
            signature,
            data,
            padding.PSS(
                mgf = padding.MGF1(hashes.SHA256()),
                salt_length = padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
```

Figure 6: Server check Client sign function

3.5 User authentication

To authenticate the user, we chose to use the Smart Card Reader to read the hardware tokens from the Citizen's Card. For this, the 'Autenticação.gov' application is needed in order to allow the user to introduce the authentication PIN of his CC. Then, the CA, certificate and signature are generated from the card. The signature is generated using the CC's private key.

```
def user_auth(self):

    # check what OS is running
    if(sys.platform == 'darwin'):
        LIB = '/usr/local/lib/libpteidpkcs11.dylib'
    elif(sys.platform == 'linux'):
        LIB = '/usr/local/lib/libpteidpkcs11.so'

    # read and generate all the hardware info
    pkcs11_lib = PyKCS11.PyKCS11Lib()
    pkcs11_lib.load(LIB)

    session = pkcs11_lib.openSession(pkcs11_lib.getSlotList()[0])

    all_atributes = list(PyKCS11.CKA.keys())
    all_atributes = [attr for attr in all_atributes if isinstance(attr, int)]

    private_key = session.findObjects([(PyKCS11.CKA_CLASS, PyKCS11.CKO_PRIVATE_KEY), (PyKCS11.CKA_LABEL, 'CITIZEN AUTHENTICATION KEY')])[0]

    attr = session.getAttributeValue(session.findObjects([(PyKCS11.CKA_LABEL, 'AUTHENTICATION SUB CA')])[0], all_atributes)
    attr = dict(zip(map(PyKCS11.CKA.get, all_atributes), attr))

    auth_sub_ca_cert = bytes(attr['CKA_VALUE']).decode("latin")

    attr = session.getAttributeValue(session.findObjects([(PyKCS11.CKA_LABEL, 'ROOT CA')])[0], all_atributes)
    attr = dict(zip(map(PyKCS11.CKA.get, all_atributes), attr))

    root_ca_cert = bytes(attr['CKA_VALUE']).decode("latin")

    attr = session.getAttributeValue(session.findObjects([(PyKCS11.CKA_LABEL, 'CITIZEN AUTHENTICATION CERTIFICATE')])[0], all_atributes)
    attr = dict(zip(map(PyKCS11.CKA.get, all_atributes), attr))

    citizen_certificate = x509.load_der_x509_certificate(bytes(attr['CKA_VALUE']))
    citizen_cert = bytes(attr['CKA_VALUE']).decode("latin")

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

    signature = bytes(
        session.sign(
            private_key,
            citizen_certificate.subject.get_attributes_for_oid(NameOID.SERIAL_NUMBER)[0].value.encode(),
            mechanism
        )
    )

    return [citizen_cert, auth_sub_ca_cert, root_ca_cert], signature
```

Figure 7: Generate Citizen's Card info function

Then, this information is sent to server to be validated. On the server side, The signature is verified using the public key of the user. Then, all the information will be verified in the 'check_chain' function, such as the CRL Delta and the Issuer. This is to make sure that it is not revoked and has a valid issuer.

```python
def check_user(self, request):
    # decrypt data
    secure_data = request.args
    data = self.extract_content(secure_data)

    # cc info
    cc_list = json.loads(data['data'])

    # session info
    session_id = json.loads(request.args.get(b'sessionID', [None])[0].decode('latin'))
    session = self.open_sessions[session_id]

    # cc certificate
    citizen_cert = x509.load_der_x509_certificate(cc_list['certificate'][0].encode('latin'))

    # user id
    user_id = citizen_cert.subject.get_attributes_for_oid(NameOID.SERIAL_NUMBER)[0].value

    try:
        # verifying cc signature
        citizen_cert.public_key().verify(
            cc_list['signature'].encode('latin'),
            user_id.encode(),
            padding.PKCS1v15(),
            hashes.SHA1()
        )

        # check all cc info
        if self.check_chain(cc_list['certificate']):
            session['user_id'] = user_id
            request.responseHeaders.addRawHeader(b"content-type", b"application/json")
            return json.dumps(self.secure({'user_id': user_id, 'status': 0}, session)).encode('latin')

        # something is invalid
        else:
            request.responseHeaders.addRawHeader(b"content-type", b"application/json")
            return json.dumps(self.secure({'user_id': user_id, 'status': 1}, session)).encode('latin')

    except:
        request.responseHeaders.addRawHeader(b"content-type", b"application/json")
        return json.dumps(self.secure({'user_id': user_id, 'status': 1}, session)).encode('latin')
```

Figure 8: Check user authenticity

3.6 Content Authentication

Using the same method, we created a private key and a certificate for the media. Before downloading the media chunks, the client requests an authentication for the media. Then, the server opens the private key, signs it (with the DH parameters as data) and sends it to the client, as well as the certificate. Afterwards the client just needs to check if the signature is valid, using the content public key. The functions that do this only differ in the certificates and keys used, so there is no need to present them here. On the server, the function "auth_content" is called and then the "sign_content" function signs it. The information is sent to the client and then he verifies it in the function "check_sign_content". With this, the client can be assured that the media is safe.

3.7 Hardware tokens

Using the Smart Card Reader, the Citizen's Card is used to authenticate the users. This allows the client to use its hardware tokens in the authentication. The only way possible to successfully open the client is to have a valid CC in the Smart Card Reader.

3.8 Media at rest protection

A new script, 'file_encrypt.py' was created to encrypt the media files. We used a PBKDF with a password to generate a 16 bits key. Then, we used the AES128 cipher to encrypt the files, using the previously generated key and a random iv. With this, the media files are now protected by this encryption. Then, on the server side, we load the files and, before sending chunks, decrypt the file and save it only in the memory. This way, anyone that gets to the media folder cannot extract the media.

```
# Open file, seek to correct position and return the chunk
with open(os.path.join(CATALOG_BASE, media_item['file_name']), 'rb') as f:
    # read key
    key = f.read(16)
    # read iv
    iv = f.read(16)
    # cipher
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    # decrypt file
    decryptor = cipher.decryptor()
    d = f.read()
    ct = decryptor.update(d) + decryptor.finalize()
    # unpad
    unpadder = pad.PKCS7(128).unpadder()
    ct1 = unpadder.update(ct) + unpadder.finalize()

    # chunk
    data = ct1[offset : offset + CHUNK_SIZE]
```

Figure 9: Media decryption

4 Using our program

To run the server, it is needed to install the 'requirements.txt' file using 'pip3 install -r server/requirements.txt'.

To run the client, it is also needed to install the requirements using 'pip3 install -r client/requirements.txt'. Also it is needed to install the PyKCS11 package that is used to read the Citizen's Card. Besides that, the application 'Aplicação.gov' needs to be installed so that the user can introduce the authentication PIN of his CC. The app can be downloaded from https://www.autenticacao.gov.pt/cc-aplicacao

After this, with the Smart Card Reader connected, the program is ready to run.

The following lines of code contain values that maybe be relevant to change for quicker testing of the program:

- client.py line 44: has the client's suites, can be changed to test other suites
- server.py line 298: has the chunk threshold for key rotation, currently at 500
- server.py line 327350: amount of views on a license currently at 4 (meaning 5 total views)

5 Bibliography

[1] Guides from the practical classes

[2] https://cryptography.io/en/latest/