Universidade de Aveiro

Departamento Eletrónica, Telecomunicações e Informática

Projects 2+3: Digital Rights Management

Engenharia Informática

Trabalho Realizado por:

- Diogo Cunha nºmec: 95278

Indice

Introdução

Este trabalho foi desenvolvido no âmbito da Unidade Curricular Segurança Informática e nas Organizações em que o objetivo de implementar um media player.
No desenvolvimento deste trabalho vão ser abordados temas como encriptação, autenticação e o uso do cartão de cidadão.
Para correr o programa os requirements.txt terão de ser instalados. Será obrigatório um leitor de smartcards para uso do cartão de cidadão.

Após isto será necessário apenas correr o server.py e introduzir a password "ola" e depois iniciar o client.py Ter em atenção que a licença é estática e esta definida para a visualização de 100 chunks.

Abordagem ao problema

A minha abordagem ao problema para que fosse garantida a confidencialidade e autenticidade dos dados foi uma troca diffie-hellman em que ambas as partes têm de ser autenticadas.

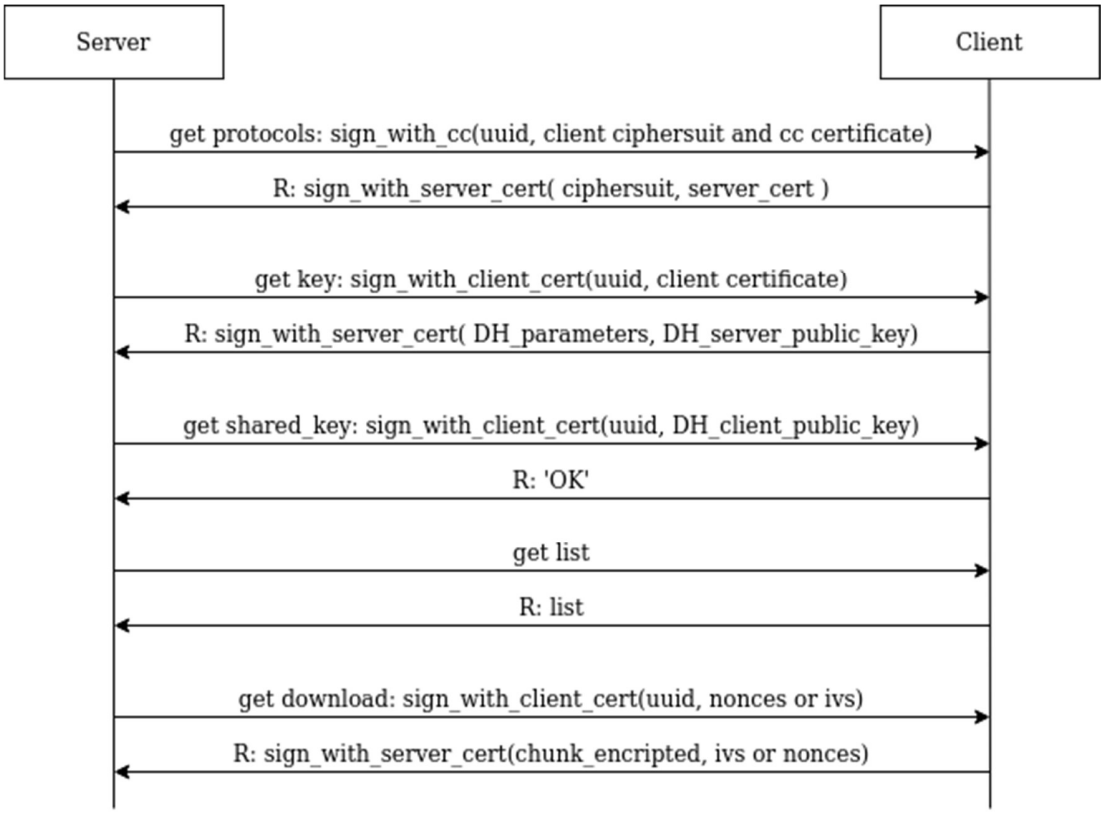
Para garantir a autenticidade o client tem um certificado assinado pela CA, o user terá de se autenticar com recurso ao seu cartão de cidadão e o servidor terá também um certificado assinado pela CA. O servidor e o client verificam se o certificado recebido está assinado pela CA. Esse certificado que é trocado no início, vai servir para verificar todas as assinaturas no lado do server e no lado do client.

As cifras suportadas no projeto são AES256 com modos CBC e GCM e ChaCha20.
Os digests sao SHA256 e SHA512.
As chaves negociadas com o diffiel-hellman são efémeras por isso só servem para uma única sessão. As shared key é derivada a cada chunk.
Todas as comunicações são assinadas. Todas as comunicações são encriptadas a partir do momento em que há uma shared key, antes de haver uma shared key a informação trocada não é encriptada, mas também não é relevante para um atacante.
A integridade das mensagens também é confirmada através da assinatura da mesma quer do lado do client quer do lado do server.
As licenças são geradas quando o client se conecta pela primeira vez ao server. As licenças são guardadas num dicionário encriptado.
O programa suporta vários clients em simultâneo.

Workflow do programa

Na abordagem ao problema vai ser usada a troca de chaves por Diffie Hellman para que servidor e client consigam obter uma shared key de forma secreta.

- Ao arrancar o **server** é necessário introduzir a chave para desencriptar as musicas.
- Ao arrancar o **client** necessita obrigatoriamente de introduzir o código de autenticação do cartão de cidadão. E após isso faz um pedido ao servidor em que envia uma mensagem com um uuid (Universally unique identifier), as ciphersuits suportadas e o certificado do cartão de cidadão. Essa mensagem e assinada pela chave privada do cartão de cidadão.
- O **server** ao receber o pedido retorna ao client uma mensagem assinada pela chave privada do servidor com a ciphersuite escolhida e o certificado do servidor.
- O **client** faz depois outro pedido get ao servidor para receber os parâmetros para a troca diffie hellman. Nesse pedido o client envia um uuid e o certificado do client assinado com a chave privada do seu certificado
- O **server** ao receber o pedido do client retorna os parâmetros e a chave publica para a troca de chaves diffie hellman
- O **client** ao receber essa informação do servidor gera as suas chaves diffie hellman, gera uma shared key e envia a chave publica diffie hellman assinada para o servidor.
- O **server** ao receber essa chave publica do client consegue fazer o exchange e obter uma shared key igual a do client.
- Após servidor e cliente estarem de acordo com a ciphersuite e ambos terem a mesma chave partilhada as comunicações podem ser encriptadas e o **cliente** vai pedir ao servidor a lista de músicas.
- O **server** envia para o client a sua lista de músicas.
- Após o **client** escolher a sua música vai pedir ao servidor a musica chunk a chunk.
- O servidor recebe o pedido de cada chunk encriptado e envia o chunk pedido também encriptado.


Server
Client
get protocols: sign_with_cc(uuid, client ciphersuit and cc certificate)
R: sign_with_server_cert(ciphersuit, server_cert)
get key: sign_with_client_cert(uuid, client certificate)
R: sign_with_server_cert(DH_parameters, DH_server_public_key)
get shared_key: sign_with_client_cert(uuid, DH_client_public_key)
R: 'OK'
get list
R: list
get download: sign_with_client_cert(uuid, nonces or ivs)
R: sign_with_server_cert(chunk_encripted, ivs or nonces)

Explicação do código

A primeira coisa a ser feita para o **server** é inserir a chave para que possam ser desencriptadas as músicas quando pedido o chunk das mesmas. Neste caso para a unica musica ha um ficheiro com os nonce e os salts, a chave introduzida `e derivada segundo o salt contido no ficheiro e `e inicializada a cifra para desincriptar a musica mais tarde.

```
#Decrypt song
key_to_files = getpass()

with open("catalog/898a08080d1840793122b7e118b27a95d117ebce_nonce_salt", "rb") as nonce_salt_reader:
    nonce_salt_json = nonce_salt_reader.read()

nonce_salt_json = json.loads(nonce_salt_json.decode('latin'))
nonce = base64.b64decode(nonce_salt_json["nonce"])
salt = base64.b64decode(nonce_salt_json["salt"])

kdf = PBKDF2HMAC(
algorithm=hashes.SHA512(),
length=32,
salt=salt,
iterations=100000,
)
key_to_files = kdf.derive(key_to_files.encode())

files_cipher = Cipher(algorithms.ChaCha20(key_to_files, nonce), mode=None)
decryptor_x = files_cipher.decryptor()
```

server.py

O **servidor** carrega também ao iniciar o seu certificado e o certificado da certification authoraty para usar mais tarde.

```
#Certificados do servidor e private key
with open("Server_certificate.pem", "rb") as key_file:
    server_cert_private_key = serialization.load_pem_private_key(
        key_file.read(),
        password=None,
    )
with open("Server_certificate.crt", "rb") as cert_file:
    server_cert = cert_file.read()
with open("Certification_Authority.crt", "rb") as cert_file:
    CA_cert = x509.load_pem_x509_certificate(cert_file.read())
    CA_public_key = CA_cert.public_key()
```

server.py

A primeira cosa que o **client** faz é autenticar o utilizador com o cartão de cidadão e enviar os ciphersuits suportados assinados com a chave privada do client, garantindo assim a integridade e autenticidade da mensagem.

```
CLIENT_CIPHERSUITS = ["AES256_CBC_SHA256", "AES256_CBC_SHA512", "AES256_GCM_SHA256", "AES256_GCM_SHA512", "GCM_None_SHA256", "GCM_None_SHA512"]

lib ='/usr/local/lib/libpteidpkcs11.so'
pkcs11 = PyKCS11.PyKCS11Lib()
pkcs11.load(lib)
slots = pkcs11.getSlotList()
slot = slots[0]
session = pkcs11.openSession(slot)

obj = session.findObjects([(PyKCS11.CKA_CLASS, PyKCS11.CKO_CERTIFICATE),
                           (PyKCS11.CKA_LABEL, 'CITIZEN AUTHENTICATION CERTIFICATE')])[0]
all_atributes = [PyKCS11.CKA_VALUE]
attributes = session.getAttributeValue(obj, all_atributes)[0]
cert = x509.load_der_x509_certificate(bytes(attributes))
cc_cert_pem = cert.public_bytes(encoding=serialization.Encoding.PEM)


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


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

data = {"uuid": uuid_c, "client_ciphersuits": CLIENT_CIPHERSUITS, "cc_cert": cc_cert_pem.decode('latin')}
data = json.dumps(data)
signature = bytes(session.sign(cc_private_key, data, mechanism))


payload = {"data": data, "signature": base64.b64encode(signature).decode('latin')}
req = requests.get(f'{SERVER_URL}/api/protocols', data= json.dumps(payload))
```

client.py

O **server** na funcao do_get_protocols() escolhe uma ciphersuite suportada por ambos client e server. E cria um dicionário com as informações referentes a cada user.

```
SERVER_CIPHERSUITS = ["AES256_CBC_SHA256", "AES256_CBC_SHA512", "AES256_GCM_SHA256", "AES256_GCM_SHA512", "ChaCha20_None_SHA256", "ChaCha20_None_SHA512"]

ciphersuit = random.choice(CLIENT_CIPHERSUITS)

#Procura uma ciphersuit suportada pelo client e pelo servidor
while len(CLIENT_CIPHERSUITS) > 0:
    if ciphersuit in SERVER_CIPHERSUITS:
        logger.debug(f'Ciphersuit defined: {ciphersuit}')
        break
    else:
        CLIENT_CIPHERSUITS.remove(ciphersuit)
        ciphersuit = random.choice(CLIENT_CIPHERSUITS)
        if len(CLIENT_CIPHERSUITS) == 0:
            return b'Server doesnt support clients ciphersuit'


algorithms_modes_digests = ciphersuit.split("_")
algorithm = algorithms_modes_digests[0]
mode = algorithms_modes_digests[1]
digest_c = algorithms_modes_digests[2]


files_cipher = Cipher(algorithms.ChaCha20(key_to_files, nonce_files), mode=None)
decryptor_files = files_cipher.decryptor()


#verificar se o user ja estava inscrito
if client_uuid in self.users.keys():
    self.users[client_uuid]["algorithm"] = algorithm
    self.users[client_uuid]["mode"] = mode
    self.users[client_uuid]["digest"] = digest_c
    self.users[client_uuid]["decriptor"] = decryptor_files
else:
    licenca = json.dumps({ client_uuid: { "usos": 100 } }).encode()
    cipher = Cipher(algorithms.ChaCha20(key_to_files, self.nonce_for_licence), mode=None)
    encryptor = cipher.encryptor()
    licenca_e = encryptor.update(licenca) + encryptor.finalize()
    self.users[client_uuid] = { "algorithm": algorithm, "mode": mode, "digest": digest_c, "licenca": licenca_e, "cc_cert": cc_cert, "Autenticado": True, "decriptor": decryptor_files }
```

server.py

O **server** vai assinar as mensagens conforme a digest escolhida para cada client. As assinaturas são enviadas no formato de base64.

```
data = { "ciphersuit": ciphersuit, "server_cert": self.server_cert.decode('latin') }
data = json.dumps(data)

if self.users[client_uuid]["digest"] == "SHA256":
    signature = self.server_cert_private_key.sign(
            data.encode(),
            paddingAsymetric.PSS(
                mgf=paddingAsymetric.MGF1(hashes.SHA256()),
                salt_length=paddingAsymetric.PSS.MAX_LENGTH
                ),
                hashes.SHA256()
            )
elif self.users[client_uuid]["digest"] == "SHA512":
    signature = self.server_cert_private_key.sign(
            data.encode(),
            paddingAsymetric.PSS(
                mgf=paddingAsymetric.MGF1(hashes.SHA512()),
                salt_length=paddingAsymetric.PSS.MAX_LENGTH
                ),
                hashes.SHA512()
            )
else:
    print("Erro")


payload = { "data": data, "signature": base64.b64encode(signature).decode('latin') }
request.setResponseCode(200)
request.responseHeaders.addRawHeader(b"content-type", b"application/json")
return json.dumps(payload).encode('latin')
```

server.py

O **client** depois de receber o certificado e a ciphersuite do server vai verificar se o certificado do server esta assinado pela CA e depois verificar a assinatura para garantir a autenticidade e integridade dos conteúdos recebidos.

```
req = requests.get(f'{SERVER_URL}/api/protocols', data= json.dumps(payload))
req = req.json()

data_signed = json.loads(req["data"])
algorithms_modes_digests = data_signed["ciphersuit"].split("_")

algorithm = algorithms_modes_digests[0]
mode = algorithms_modes_digests[1]
digest_c = algorithms_modes_digests[2]

signature = base64.b64decode(req["signature"].encode())

with open("Certification_Authority.crt", "rb") as CA_cert_file:
    CA_cert = x509.load_pem_x509_certificate(CA_cert_file.read())
    CA_public_key = CA_cert.public_key()

server_cert = x509.load_pem_x509_certificate(data_signed["server_cert"].encode())
server_public_key_rsa = server_cert.public_key()

#Verificar o certificado
CA_public_key.verify(
    server_cert.signature,
    server_cert.tbs_certificate_bytes,
    paddingAsymetric.PKCS1v15(),
    server_cert.signature_hash_algorithm,
)

#verificar assinatura
```

client.py

Para a funcao do_get_keys() o **server** vai receber o certificado do client e garantir que esta assinado pela CA e vai verificar a assinatura.

```
client_cert = x509.load_pem_x509_certificate(data_signed["client_cert"].encode())
client_public_key_rsa = client_cert.public_key()


#Verificar que o certificado recebido esta assinado pela CA
self.CA_public_key.verify(
    client_cert.signature,
    client_cert.tbs_certificate_bytes,
    paddingAsymetric.PKCS1v15(),
    client_cert.signature_hash_algorithm,
)

#Verificar a assinatura
if self.users[client_uuid]["digest"] == "SHA256":
    client_public_key_rsa.verify(
        signature,
        req["data"].encode(),
        paddingAsymetric.PSS(
            mgf=paddingAsymetric.MGF1(hashes.SHA256()),
            salt_length=paddingAsymetric.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
elif self.users[client_uuid]["digest"] == "SHA512":
    client_public_key_rsa.verify(
        signature,
        req["data"].encode(),
        paddingAsymetric.PSS(
            mgf=paddingAsymetric.MGF1(hashes.SHA512()),
            salt_length=paddingAsymetric.PSS.MAX_LENGTH
        ),
        hashes.SHA512()
    )
else:
    print("Erro")
    sys.exit(0)

self.users[client_uuid]["client_cert"] = client_cert
```

server.py

Depois são gerados no **server** os parâmetros para a troca diffie-hellman e gerada a chave privada e publica em que os parâmetros e chave publica são assinados e enviados para o client.

```
parameters = dh.generate_parameters(generator=2, key_size=2048)
parameters_pem = parameters.parameter_bytes(Encoding.PEM, ParameterFormat.PKCS3)

# Generate a private key DH
self.server_private_key = parameters.generate_private_key()

# Generate a public DH
server_public_key_pem = self.server_private_key.public_key().public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo
)
```

server.py

O **client** depois de verificar a assinatura vai utilizar os parâmetros para criar uma chave publica e uma privada e vai utilizar a sua chave privada e a chave publica do client para gerar um shared key essa shared key é depois derivada.

Vai depois assinar a sua chave publica e enviar para o server.

```
parameters_pem = req["parameters"].encode()
server_pub_key_pem = req["server_pub_key"].encode()

server_pub_key = load_pem_public_key(server_pub_key_pem)
parameters = load_pem_parameters(parameters_pem)

client_private_key = parameters.generate_private_key()
client_pub_key_pem = client_private_key.public_key().public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
        )

client_shared_key = client_private_key.exchange(server_pub_key)
if digest_c == "SHA256":
    shared_key_derived = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
    ).derive(client_shared_key)
elif digest_c == "SHA512":
    shared_key_derived = HKDF(
        algorithm=hashes.SHA512(),
        length=32,
        salt=None,
        info=b'handshake data',
    ).derive(client_shared_key)
else:
    print("Erro ao derivar a shared key")
    sys.exit(0)

data = {"uuid": uuid_c, "client_pub_key": client_pub_key_pem.decode('utf-8')}
data = json.dumps(data)
signature = client_sign(digest_c, data)

payload = {"data": data, "signature": base64.b64encode(signature).decode('latin') }
req = requests.post(url=f'{SERVER_URL}/api/shared_key', data=json.dumps(payload))
```

client.py

O **server** ao receber a chave publica do client vai conseguir gerar uma shared key tal como aconteceu no client. Essa chave vai ser igual no servidor e no client e vai ser utilizada para encriptar todos os dados daqui para a frente. Essa shared key `e guardada no dicionário dos users associando assim cada shared key a cada user.

```
client_pub_key = load_pem_public_key(client_pub_key_pem)
shared_key = self.server_private_key.exchange(client_pub_key)

if self.users[client_uuid]["digest"] == "SHA256":
    shared_key_derived = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
    ).derive(shared_key)
elif self.users[client_uuid]["digest"] == "SHA512":
    shared_key_derived = HKDF(
        algorithm=hashes.SHA512(),
        length=32,
        salt=None,
        info=b'handshake data',
    ).derive(shared_key)
else:
    sys.exit(0)

self.users[client_uuid]["shared_key"] = shared_key_derived
```

server.py

O client vai depois pedir a lista ao server e selecionar a música que quer ouvir.
Depois disso vai começar a pedir chunks da música. Esses pedidos são encriptados para que o atacante não saiba que musica esta a ser ouvida nem que chunk esta a ser pedido pelo client. Junto com o pedido de chunks vai um payload que contem nonce ou o iv para desencriptar (dependendo da cifra usada) e uma assinatura desse payload.

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

    if algorithm == "AES256":
        if mode == "CBC":
            media_id, iv = encrypt_AES(shared_key_derived, media_item["id"].encode(), "CBC")
            iv = base64.b64encode(iv).decode('latin')

            chunk, iv2 = encrypt_AES(shared_key_derived, str(chunk).encode(), "CBC")
            iv2 = base64.b64encode(iv2).decode('latin')

            info = json.dumps({"uuid": uuid_c, "iv": iv, "iv2": iv2})

        elif mode == "GCM":
            media_id, iv, tag1 = encrypt_AES(shared_key_derived, media_item["id"].encode(), "GCM")
            iv = base64.b64encode(iv).decode('latin')
            tag1 = base64.b64encode(tag1).decode('latin')

            chunk, iv2, tag2 = encrypt_AES(shared_key_derived, str(chunk).encode(), "GCM")
            iv2 = base64.b64encode(iv2).decode('latin')
            tag2 = base64.b64encode(tag2).decode('latin')

            info = json.dumps({"uuid": uuid_c, "iv": iv, "iv2": iv2, "tag1": tag1, "tag2": tag2})

    elif algorithm == "ChaCha20":
        media_id, nonce = encrypt_ChaCha20(shared_key_derived, media_item["id"].encode())
        nonce = base64.b64encode(nonce).decode('latin')

        chunk, nonce2 = encrypt_ChaCha20(shared_key_derived, str(chunk).encode())
        nonce2 = base64.b64encode(nonce2).decode('latin')

        info = json.dumps({"uuid": uuid_c, "nonce": nonce, "nonce_chunk": nonce2})
    else:
        print("erro")
        sys.exit(0)

    media_id = base64.urlsafe_b64encode(media_id).decode('latin')
    chunk = base64.urlsafe_b64encode(chunk).decode('latin')

    signature = client_sign(digest_c, info)
    payload = { "data": info, "signature": base64.b64encode(signature).decode('latin') }

    req = requests.get(f'{SERVER_URL}/api/download?id={media_id}&chunk={chunk}', data=json.dumps(payload))
```

client.py

O **server** na funcao do_download() inicialmente vai verificar a assinatura do pedido do client e depois que o user que esta a fazer o pedido foi autenticado com o cartão de cidadão quando iniciou a comunicação com o server.

```
if self.users[uuid]["Autenticado"]:
    pass
else:
    return "Erro, user nao autenticado"
```

server.py

O **server** tem depois de desencriptar o pedido feito pelo servidor (media_id e chunk)

```
#decript media id e chunk
if self.users[uuid]["algorithm"] == "AES256":
    iv = data_signed["iv"].encode()
    iv = base64.b64decode(iv)
    iv2 = data_signed["iv2"].encode()
    iv2 = base64.b64decode(iv2)


    key = self.users[uuid]["shared_key"]

    if self.users[uuid]["mode"] == "CBC":
        media_id = self.decrypt_AES(key, iv, media_id, "CBC")
        chunk_id = self.decrypt_AES(key, iv2, chunk_id, "CBC")
    elif self.users[uuid]["mode"] == "GCM":
        tag1 = data_signed["tag1"].encode()
        tag1 = base64.b64decode(tag1)
        tag2 = data_signed["tag2"].encode()
        tag2 = base64.b64decode(tag2)
        media_id = self.decrypt_AES(key, iv, media_id, "GCM", tag1)
        chunk_id = self.decrypt_AES(key, iv2, chunk_id, "GCM", tag2)

elif self.users[uuid]["algorithm"] == "ChaCha20":
    nonce = data_signed["nonce"].encode()
    nonce = base64.b64decode(nonce)
    nonce2 = data_signed["nonce_chunk"].encode()
    nonce2 = base64.b64decode(nonce2)

    key = self.users[uuid]["shared_key"]
    media_id = self.decrypt_ChaCha20(key, nonce, media_id)
    chunk_id = self.decrypt_ChaCha20(key, nonce2, chunk_id)
```

server.py

Ao fim de saber o pedido do client, o **server** vai ter de verificar a licenças do user. Para as licenças do user no caso do meu programa é definido o valor de 100 chunks e são encriptadas. As licenças são desencriptadas apenas para verificação e logo após são encriptadas de novo.

```
licenca = self.users[uuid]["licenca"]
cipher_l = Cipher(algorithms.ChaCha20(key_to_files, self.nonce_for_licence), mode=None)
decryptor_l = cipher_l.decryptor()
licenca_d = decryptor_l.update(licenca) + decryptor_l.finalize()
licenca = json.loads(licenca_d.decode())


if licenca[uuid]["usos"] == 0 :
    print("Client nao tem licenca para ouvir a musica")

    data = {'error': 'Nao tem licenca para ouvir a musica'}
    data = json.dumps(data)
    signature = self.sign(data, self.users[uuid]["digest"])

    request.responseHeaders.addRawHeader(b"content-type", b"application/json")
    return json.dumps({"data": data, "signature": base64.b64encode(signature).decode('latin')}).encode('latin')
else:
    licenca[uuid]["usos"] -= 1
    licenca = json.dumps(licenca).encode()
    cipher_l = Cipher(algorithms.ChaCha20(key_to_files, self.nonce_for_licence), mode=None)
    encryptor_l = cipher_l.encryptor()
    licenca_e = encryptor_l.update(licenca) + encryptor_l.finalize()
    self.users[uuid]["licenca"] = licenca_e
```

server.py

O **server** vai para cada chunk pedido pelo client desencriptar a música para depois encriptar com a shared key. Essa shared key é derivada para cada chunk pedido pelo client, sendo o server a gerar o salt para essa mesma derivação, enviando depois para o client.

```
f.seek(offset)
data = f.read(CHUNK_SIZE)

decryptor_filess = self.users[uuid]["decriptor"]
data = decryptor_filess.update(data)
```

server.py

O **server** vai então encriptar o chunk com a chave derivada, enviando depois para o client o chunk encriptado, o salt, o MAC e uma assinatura.

Exemplo com ChaCha20:

```
if self.users[uuid]["algorithm"] == "ChaCha20":
    salt = os.urandom(16)

    #Encript then MAC
    if self.users[uuid]["digest"] == "SHA256":
        #Derivar chave
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=100000,
        )
        key = kdf.derive(self.users[uuid]["shared_key"])

        data_encrypted, nonce = self.encrypt_ChaCha20(key, info)

        h = hmac.HMAC(key, hashes.SHA256())
        h.update(data_encrypted)
        MAC = h.finalize()

    elif self.users[uuid]["digest"] == "SHA512":
        #Derivar chave
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA512(),
            length=32,
            salt=salt,
            iterations=100000,
        )
        key = kdf.derive(self.users[uuid]["shared_key"])

        data_encrypted, nonce = self.encrypt_ChaCha20(key, info)
        h = hmac.HMAC(key, hashes.SHA512())
        h.update(data_encrypted)
        MAC = h.finalize()

    else:
        print("Erro")
        sys.exit(0)

    nonce = base64.b64encode(nonce).decode('latin')
    data_encrypted = base64.b64encode(data_encrypted).decode('latin')
    MAC = base64.b64encode(MAC).decode('latin')
    salt = base64.b64encode(salt).decode('latin')

    data = { "data": data_encrypted, "nonce": nonce, "MAC": MAC, "salt": salt}
    data = json.dumps(data)

    signature = self.sign(data, self.users[uuid]["digest"])
    payload = {"data": data, "signature": base64.b64encode(signature).decode('latin') }

    request.setResponseCode(200)
    request.responseHeaders.addRawHeader(b"content-type", b"application/json")
    return json.dumps(payload).encode('latin')
```

server.py

Existe também a possibilidade de o **server** em vez de enviar o chunk encriptado enviar uma mensagem assinada a dizer que a licença do user já não permite visualizar mais o conteúdo.

O **client** para tratar disso vai verificar a assinatura como sempre e depois vai tentar receber o chunk se a mensagem não for de envio de chunks vau haver uma exceção e o client vai parar de ouvir a musica.

```
elif algorithm == "ChaCha20":
    try:
        nonce = req["nonce"].encode()
        nonce = base64.b64decode(nonce)
        data_encrypted = req["data"].encode()
        data_encrypted = base64.b64decode(data_encrypted)
        MAC = req["MAC"].encode()
        MAC = base64.b64decode(MAC)
        salt = req["salt"].encode()
        salt = base64.b64decode(salt)

    except:
        print(req["error"])
        proc.kill()
        break
        return 0
```

client.py

Quando a mensagem vem com a informação para desencriptar o chunk o client vai verificar o MAC nessa mensagem e depois usar as funções para desencriptar e depois reproduzir o chunk.

```
    elif digest_c == "SHA512":
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA512(),
            length=32,
            salt=salt,
            iterations=100000,
        )
        key = kdf.derive(shared_key_derived)

        h = hmac.HMAC(key, hashes.SHA512())
        h.update(data_encrypted)
        h.verify(MAC)

    else:
        print("ERRO")
        sys.exit(0)

    data = decrypt_ChaCha20(key, nonce, data_encrypted)
    info = json.loads(data.decode('latin'))

    data = info["data"]
    data = binascii.a2b_base64(data)
else:
    print("Erro")
    sys.exit(0)

try:
    proc.stdin.write(data)
except:
    break
```

client.py

Ao fim de reproduzir o client começa novamente tudo de início, ou seja, as chaves são efémeras, o client após parar de ouvir a música todo o processo vai ser repetido desde a negociação de cifras ate à reprodução de uma nova musica.

A função utilizada para encriptar o ficheiro com o áudio é a seguinte (encrypt_file.py):

```
key = key_to_files = getpass()

salt = os.urandom(16)

kdf = PBKDF2HMAC(
    algorithm=hashes.SHA512(),
    length=32,
    salt=salt,
    iterations=100000,
    backend=backends.default_backend()
)
key = kdf.derive(key.encode())

nonce = os.urandom(16) #a variavel iv tem o nonce para simplificar o print
cipher = Cipher(algorithms.ChaCha20(key, nonce), mode=None, backend=backends.default_backend())
encryptor = cipher.encryptor()

#encrypt
song_file = open("server/catalog/898a08080d1840793122b7e118b27a95d117ebce_non_encrypted.mp3", "rb")     #som nao encriptado
writer_encrypted_song = open("server/catalog/898a08080d1840793122b7e118b27a95d117ebce.mp3", "wb")       #som encriptado

with open("server/catalog/898a08080d1840793122b7e118b27a95d117ebce_nonce_salt", "wb") as nonce_salt_writer:
    nonce = base64.b64encode(nonce).decode('latin')
    salt = base64.b64encode(salt).decode('latin')
    json_data = json.dumps({"nonce": nonce, "salt": salt}).encode('latin')
    nonce_salt_writer.write(json_data)


blocksize = 4096
while True:
    chunk = song_file.read(blocksize)
    if chunk:
        if len(chunk)!=blocksize:
            ct = encryptor.update(chunk) + encryptor.finalize()
        else:
            ct = encryptor.update(chunk)

        writer_encrypted_song.write(ct)

    else:
        break


song_file.close()
nonce_salt_writer.close()
writer_encrypted_song.close()
```

encrypt_file.py

Para cada media encriptado no catálogo vai existir um ficheiro com o salt e nonce do respetivo ficheiro de modo a que se possa ser desencriptado pelo server quando requisitado por um client.

Seria necessário depois apagar ou fazer um backup do media desencriptado para apenas estar acessível a um atacante o ficheiro encriptado.

Funções utilizadas no server.py:

```python
def encrypt_AES(self, key, info, modo):
    if modo == "CBC":
        iv = os.urandom(16)
        blocksize = 16

        cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
        encryptor = cipher.encryptor()
        padder = padding.PKCS7(blocksize*8).padder()

        info = padder.update(info) + padder.finalize()
        data_encrypted = encryptor.update(info) + encryptor.finalize()

        return data_encrypted, iv

    elif modo == "GCM":
        iv = os.urandom(12)

        encryptor = Cipher(
            algorithms.AES(key),
            modes.GCM(iv),
        ).encryptor()

        encryptor.authenticate_additional_data(b'associated_data')
        data_encrypted = encryptor.update(info) + encryptor.finalize()


        return data_encrypted, iv, encryptor.tag
    else:
        print("ERRO")
        sys.exit(0)


def encrypt_ChaCha20(self, key, info):
    nonce = os.urandom(16)

    algorithm = algorithms.ChaCha20(key, nonce)
    cipher = Cipher(algorithm, mode=None)
    encryptor = cipher.encryptor()
    data_encrypted = encryptor.update(info)
    return data_encrypted, nonce

def decrypt_AES(self, key, iv, data_encrypted, modo, tag=None):
    if modo == "CBC":
        cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
        decryptor = cipher.decryptor()
        unpadder = padding.PKCS7(128).unpadder()

        data = decryptor.update(data_encrypted) + decryptor.finalize()
        info = unpadder.update(data) + unpadder.finalize()
        return info

    elif modo == "GCM":
        decryptor = Cipher(
            algorithms.AES(key),
            modes.GCM(iv, tag),
        ).decryptor()

        decryptor.authenticate_additional_data(b'associated_data')
        return decryptor.update(data_encrypted) + decryptor.finalize()
    else:
        print("Erro")
        sys.exit(0)

def decrypt_ChaCha20(self, key, nonce, data_encrypted):
    algorithm = algorithms.ChaCha20(key, nonce)
    cipher = Cipher(algorithm, mode=None)
    decryptor = cipher.decryptor()
    info = decryptor.update(data_encrypted)

    return info
```

Funções utilizadas no client.py:

```python
def encrypt_AES(key, info, modo):
    if modo == "CBC":
        iv = os.urandom(16)
        blocksize = 16

        cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
        encryptor = cipher.encryptor()
        padder = padding.PKCS7(blocksize*8).padder()

        info = padder.update(info) + padder.finalize()
        data_encrypted = encryptor.update(info) + encryptor.finalize()

        return data_encrypted, iv

    elif modo == "GCM":
        iv = os.urandom(12)

        encryptor = Cipher(
            algorithms.AES(key),
            modes.GCM(iv),
        ).encryptor()

        encryptor.authenticate_additional_data(b'associated_data')
        data_encrypted = encryptor.update(info) + encryptor.finalize()

        return data_encrypted, iv, encryptor.tag
    else:
        print("ERRO")
        sys.exit(0)


def encrypt_ChaCha20(key, info):
    nonce = os.urandom(16)

    algorithm = algorithms.ChaCha20(key, nonce)
    cipher = Cipher(algorithm, mode=None)
    encryptor = cipher.encryptor()
    data_encrypted = encryptor.update(info)
    return data_encrypted, nonce

def decrypt_AES(key, iv, data_encrypted, modo, tag=None):
    if modo == "CBC":
        cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
        decryptor = cipher.decryptor()
        unpadder = padding.PKCS7(128).unpadder()

        data =  decryptor.update(data_encrypted) + decryptor.finalize()
        data = unpadder.update(data) + unpadder.finalize()

        return data

    elif modo == "GCM":
        decryptor = Cipher(
            algorithms.AES(key),
            modes.GCM(iv, tag),
        ).decryptor()

        decryptor.authenticate_additional_data(b'associated_data')
        return decryptor.update(data_encrypted) + decryptor.finalize()
    else:
        print("Erro")
        sys.exit(0)

def decrypt_ChaCha20(key, nonce, data_encrypted):
    algorithm = algorithms.ChaCha20(key, nonce)
    cipher = Cipher(algorithm, mode=None)
    decryptor = cipher.decryptor()
    info = decryptor.update(data_encrypted)

    return info
```

Conclusão

Para concluir penso que este trabalho deu para pôr em prática e aprofundar algumas noções de criptografia e autenticidade através da troca de chaves, uso de chaves assimétricas e simétrica e a criação e uso de certificados.

Penso que praticamente todas as funcionalidades pedidas no enunciado para o programa estão implementadas e que no geral é um programa seguro.