Ex1 - Diffie-Hellman, com DSA

Neste trabalho prático, numa primeira parte, é pretendido construir uma sessão sincrona entre dois agentes, combinando:

- um gerador de nounces, sendo que estes têm de ser únicos
- uma cifra simétrica AES com autentificação HMAC, sendo que o modo utilizado tem de ser seguro contra ataques ao iv
- o protocolo de acordo de chaves Diffie-Hellman com verificação da chave, e autentificação de assinaturas DSA

Para tal, primeiro realizamos os imports dos modulos necessários(de notar que para a geração do texto, utilizamos o modulo 'lorem', que poderá ser preciso de instalar, com " `pip install lorem` ")

In [102]:
```
import os, io
from BiConn import BiConn
from Auxs import hashs, mac, kdf, default_algorithm
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.exceptions import *
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, hmac, serialization
from cryptography.hazmat.primitives.asymmetric import dh, dsa
import lorem
```

Começamos por gerar os parametros necessários para o acordo de chaves Diffie-Hellman, e para DSA, que serão utilizados para gerar as chaves privadas do Emitter e do Reciever

In [103]:
```
#gerar parametros DH
dh_parameters = dh.generate_parameters(generator=2, key_size=1024,backend=de
fault_backend())

#gerar parametros DSA
dsa_parameters = dsa.generate_parameters(key_size=1024, backend=default_back
end())
```

Na função KeyAgree() começamos por gerar as chaves privadas para o Emitter e para o Reciever, sendo claro que cada um usa uma estância da função, e como tal é gerado para cada um chaves distintas.

Depois de terem sido criadas as chaves privadas para DH e DSA, geramos uma chave publica apartir das chaves privadas e convertemos para o formato PEM.

Utilizamos a chave privada DSA para assinar a mensagem, que neste caso será a chave publica DH e de seguida é enviado a chave publica DSA, a chave publica DH (ambas em formato PEM) e a assinatura, sendo que depois é recebida a informação envada no outro lado da conexão. Verifica-se se a assinatura é valida, e caso seja, é realizado o "DH handshake" e verificamos se no outro lado da conexão obteve-se a mesma chave partilhada. Caso seja, returnamos a hash gerada apartir da chave partilhada, que será utilizada como chave para cifrar

In [104]:
```
def KeyAgree(conn, name):
    #gera chave privada (DH)
    dhpk = dh_parameters.generate_private_key()

    #gera chave publica (DH, formato PEM)
    dhpubkey = dhpk.public_key().public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo)

    #gera chave dsa privada
    dsapk = dsa_parameters.generate_private_key()

    #pem publico dsa
    dsapub_pem = dsapk.public_key().public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo)

    #assina a mensagem (assina-se com a chave privada)
    sig = dsapk.sign(dhpubkey, default_algorithm())

    #envia a chave publica dsa, a chave publica dh e a assinatura
    conn.send((dsapub_pem,dhpubkey,sig))

    ##recebe a chave publica dsa, a chave publica dh e a assinatura
    dsa_pub_recv,dh_pub_recv,sig_recv = conn.recv()

    peer_dsa_pub_recv = serialization.load_pem_public_key(
            dsa_pub_recv,
            backend=default_backend())
    #valida a assinatura
    try:
        peer_dsa_pub_recv.verify(sig_recv,dh_pub_recv,default_algorithm())
        print(name +' DSA Signature ok')
        # shared_key calculation
        peer_dh_pub_key = serialization.load_pem_public_key(
                dh_pub_recv,
                backend=default_backend())
        shared_key = dhpk.exchange(peer_dh_pub_key)

        # confirmation
        my_tag = hashs(bytes(shared_key))
        conn.send(my_tag)
        peer_tag = conn.recv()
        if my_tag == peer_tag:
            print(name + ' DH OK')
            return my_tag
        else:
            print(name + 'DH FAIL')
    except InvalidSignature:
        print(name + 'DSA Signature fail')



    conn.close()       # fechar a conecção
```

Cifra AES no modo CTR

Escolhemos o modo CTR porque é considerado criptograficamente forte, mas os outros modos de operação seriam também opções viaveis (como o CFB, CBC, OFB), desde que o IV utilizado seja único e utilizado apenas uma vez.

Aqui geramos um iv aleatorio e enviamos para o Reciever,e encriptamos a mensagem com AES (utilizando a hash da chave partilhada como chave de cifragem) no modo CTR, mensagem esta gerada aleatoriamente pelo modulo **lorem**. Depois de encriptar a mensagem, autentificamos (tambem garante integridade) a mensagem com o HMAC, sendo que é enviado para o Reciever a hash e a mensagem criptografada.

In [105]:
```
def Emitter(conn):

    key = KeyAgree(conn, "Emitter")
    iv  = os.urandom(16)


    text = lorem.text().encode('utf-8',"ignore")
    encryptor = Cipher(algorithms.AES(key),modes.CTR(iv),backend=default_bac
kend()).encryptor()

    conn.send(iv)

    encrypttext = encryptor.update(text) + encryptor.finalize()
    this_mac = mac(key,encrypttext)

    conn.send((this_mac,encrypttext))

    conn.close()
```

Por conseguinte, o Receiver vai receber o iv, e juntamente com a hash da chave partilhada vai desencriptar a mensagem encriptada, sendo que primeiramente é verificado a hash do Hmac recebida. Caso seja validada, o processo fica concluido e o texto desencriptado é imprimido no ecrã

In [106]:
```
def Reciever(conn):

    key = KeyAgree(conn,"Reciever")

    iv = conn.recv()
    decryptor = Cipher(algorithms.AES(key),modes.CTR(iv),backend=default_bac
kend()).decryptor()

    peer_mac, peer_msg = conn.recv()
    try:
        mac(key,peer_msg,peer_mac)
        try:
            decrypttext = decryptor.update(peer_msg) + decryptor.finalize()
            print(decrypttext.decode('utf-8',"ignore"))
        except InvalidSignature:
            print("autenticação do ciphertext falhou")
    except InvalidSignature:
        print('Hmac didnt match')



    conn.close()
```

In [107]:
```
BiConn(Emitter,Reciever,timeout=30).auto()
```

```
Emitter DSA Signature ok
Reciever DSA Signature ok
Reciever DH OK
Emitter DH OK
Quaerat etincidunt sit labore. Velit consectetur adipisci sit magnam ipsum. L
abore velit quiquia voluptatem tempora quaerat. Porro magnam modi non velit n
on quiquia aliquam. Magnam adipisci porro ipsum sit numquam. Etincidunt ut si
t sed adipisci. Sed etincidunt numquam magnam.

Non tempora quaerat sed dolore sed labore sit. Sit quisquam quisquam magnam q
uiquia modi. Quisquam dolore neque non. Sit ut ut voluptatem tempora dolorem
etincidunt. Tempora etincidunt ipsum labore numquam. Quiquia velit aliquam co
nsectetur eius amet. Ut dolor etincidunt sit adipisci consectetur. Numquam se
d modi dolorem.

Tempora dolor porro est modi tempora. Velit etincidunt dolore est adipisci ip
sum. Numquam numquam dolor amet quisquam modi. Tempora etincidunt dolore magn
am tempora quaerat sed. Dolor modi consectetur porro dolor sit. Porro numquam
velit velit non. Dolor ipsum neque ut. Aliquam ipsum quiquia adipisci sed qua
erat. Aliquam tempora ut eius magnam ut.
```

Ex2 - Elliptic-curve Diffie–Hellman, com Elliptic-curve DSA

Nesta segunda parte do exercicio, é pedido uma alteração ao exercicio anterior, sendo estas as seguintes:

- Utilizar **Elliptic-curve Diffie–Hellman (ECDH)** em vez de **Diffie–Hellman (DH)** e **Elliptic-curve DSA (ECDSA)** em vez de **DSA**
- A cifra simétrica por **ChaCha20Poly1305**

Os imports necessários:

In [108]:
```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from Auxs import hashs
from BiConn import BiConn
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes, hmac, cmac, serialization
from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305
import os
import lorem
```

O funcionamento do programa em si é muito semelhante ao anterior, sendo que o processo nesta função é igual ao anterior, onde:

- Geramos as chaves privadas para ECDH e ECDSA, gera-se as chaves publicas e convertemos para formato PEM
- Utilizamos a chave privada ECDSA para assinar a mensagem, que será a chave pública ECDH
- Envia-se a chave publica do ECDH e ECDSA e a mensagem assinada
- Verifica-se se a assinatura é válida, e caso seja, realiza-se o ECDH handshake e verficamos se o outro lado obteve a mesma chave.
- Também é retornado a hash gerada apartir da chave partilhada

In [109]:
```
def KeyAgree(conn):

        ecdh_private_key = ec.generate_private_key(
                        ec.SECP256R1(), default_backend())

        ecdh_public_key = ecdh_private_key.public_key().public_bytes(
                        encoding=serialization.Encoding.PEM,
                        format=serialization.PublicFormat.SubjectPublicKeyIn
fo)

        ecdsa_private_key = ec.generate_private_key(ec.SECP256R1(), default_
backend())

        ecdsa_public_key  = ecdsa_private_key.public_key().public_bytes(
                        encoding=serialization.Encoding.PEM,
                        format=serialization.PublicFormat.SubjectPublicKeyIn
fo)

        sign = ecdsa_private_key.sign(ecdh_public_key, ec.ECDSA(hashes.SHA25
6()))

        conn.send((ecdsa_public_key,ecdh_public_key,sign))

        ecdsa_pub_recv,ecdh_pub_recv,sign_recv = conn.recv()

        peer_ecdsa_pub_recv = serialization.load_pem_public_key(
            ecdsa_pub_recv,
            backend=default_backend())
        try:
                peer_ecdsa_pub_recv.verify(sign_recv, ecdh_pub_recv, ec.ECDS
A(hashes.SHA256()))
                print("ECDSA Signature OK")

        peer_ecdh_pub_key = serialization.load_pem_public_key(
            ecdh_pub_recv,
            backend=default_backend())
            shared_key = ecdh_private_key.exchange(ec.ECDH(),peer_ecdh_pub_k
ey)

            my_tag = hashs(bytes(shared_key))
            conn.send(my_tag)
            peer_tag = conn.recv()
            if my_tag == peer_tag:
                print('DH OK')
                return my_tag
            else:
                print('DH FAIL')
        except InvalidSignature:
                print('ECDSA Signature fail')



        conn.close()       # fechar a conecção
```

No Emitter, como pedido, é agora utilizado o ChaCha20Poly1305, e a hash da chave partilhada como chave. Geramos um nonce aleatório, que é enviado para o Reciever, e encriptamos a mensagem, que é novamente gerada com o uso do módulo **lorem**

A mensagem encriptada é depois enviada para o Reciever.

In [110]:
```
def Emitter(conn):

        key = KeyAgree(conn)

        nonce = os.urandom(12)
        conn.send(nonce)

        text = lorem.text().encode('utf-8',"ignore")
        aad = b"authenticated but unencrypted data"
        conn.send(aad)
        chacha = ChaCha20Poly1305(key)
        ct = chacha.encrypt(nonce, text,aad)
        conn.send(ct)
```

O Reciever recebe o nonce, e utilizando a hash da chave partilhada, desencripta a mensagem e imprime no ecrã

In [111]:
```
def Reciever(conn):
        key = KeyAgree(conn)
        nonce = conn.recv()
        chacha = ChaCha20Poly1305(key)
        aad = conn.recv()
        ct = conn.recv()
        try:
                decriptedtext = chacha.decrypt(nonce, ct,aad)
                print(decriptedtext)
        except InvalidSignature:
                print("Decription Failed")
```

In [112]:
```
BiConn(Emitter,Reciever,timeout=30).auto()
```

```
ECDSA Signature OK
ECDSA Signature OK
DH OK
DH OK
b'Neque ut eius numquam dolore eius. Est magnam velit ut. Quaerat consectetur
sed aliquam tempora amet dolorem etincidunt. Ipsum numquam magnam magnam eiu
s. Voluptatem quaerat numquam ut quisquam neque ut. Tempora magnam labore ips
um dolor adipisci neque quaerat. Porro tempora adipisci porro dolor quiquia u
t quisquam. Dolorem labore quaerat sed aliquam consectetur adipisci. Magnam i
psum labore voluptatem porro adipisci.\n\nDolor magnam sed porro adipisci no
n. Neque est etincidunt dolor sed modi labore dolorem. Ipsum voluptatem velit
magnam. Amet quiquia eius etincidunt. Quiquia labore magnam voluptatem consec
tetur adipisci quaerat. Adipisci consectetur etincidunt ut tempora consectetu
r sit. Quiquia modi voluptatem sed numquam aliquam consectetur quaerat. Quaer
at non quisquam labore etincidunt sit. Velit dolorem porro quisquam quaerat d
olorem aliquam. Quiquia sed aliquam dolorem neque amet.\n\nUt quiquia dolor e
st magnam. Dolore dolore labore consectetur voluptatem. Amet tempora modi ame
t adipisci. Consectetur tempora adipisci voluptatem adipisci aliquam. Sed qui
quia quaerat modi. Tempora sit est eius modi etincidunt aliquam. Dolorem quis
quam magnam sit. Neque tempora quisquam porro magnam labore neque.\n\nEst por
ro numquam adipisci dolorem amet ut velit. Velit est ut consectetur sit quisq
uam amet est. Amet numquam porro quaerat numquam consectetur ipsum numquam. A
dipisci tempora labore dolore porro. Quiquia dolorem velit quisquam etincidun
t.\n\nNeque modi dolor voluptatem. Numquam dolor velit labore labore. Amet et
incidunt aliquam neque sit etincidunt. Adipisci aliquam sed ut. Adipisci dolo
r amet voluptatem voluptatem velit. Ut eius neque ut amet modi.'
```