

TP1 - 1)

Neste exercício temos como objetivo implementar uma comunicação privada síncrona entre um agente Emitter e um agente Receiver. Esta comunicação tem que ter algumas características como:

- Um gerador de nounces: um nonce, que nunca foi usado antes, deve ser criado aleatoriamente em cada instância da comunicação
- A cifra simétrica AES usando autenticação de cada criptograma com HMAC e um modo seguro contra ataques aos vectores de iniciação (iv's).
- O protocolo de acordo de chaves Diffie-Hellman com verificação da chave, e autenticação dos agentes através do esquema de assinaturas DSA.

Solução

A nossa solução para o problema em causa passou por criar dois agentes (*Receiver* e *Emitter*). Estes dois agentes irão comunicar através de *Pipes* (esta comunicação está definida no ficheiro `PipeCommunication.py`), sendo que o Receiver terá o papel de "servidor" e o Emitter o papel de "cliente".

Neste momento do relatório vamos demonstrar a nossa solução. Primeiro os **imports** necessários

```
In [1]: import os
import time

from PipeCommunication import PipeCommunication

from cryptography.exceptions import *

from cryptography.hazmat.backends import default_backend

from cryptography.hazmat.primitives.asymmetric import dh, dsa
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes, hmac, serialization
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
```

Implementou-se o protocolo de acordo de chaves Diffie-Hellman com verificação da chave e autenticação mútua dos agente através do esquema de assinaturas Digital Signature Algorithm. O protocolo Diffie-Hellman contém 3 algoritmos:

- A criação dos parâmetros
- O agente Emitter gera a chave privada, a sua respetiva chave pública e envia ao Receiver
- O agente Receiver gera a chave privada, a sua respetiva chave pública e envia ao Emitter
- De seguida, ambos os agentes geram a chave partilhada.

O Processo de troca de chaves públicas para gerar a chave partilhada é executada tal como o protocolo está definido:

- 1) **Emitter envia a Receiver:** g^x (a sua chave pública)
- 2) **Receiver envia a Emitter:** $g^y \parallel \text{SIG}(g^x, g^y)$ (a sua chave pública \parallel as duas chaves públicas assinadas)
- 3) **Emitter envia a Receiver:** $\text{SIG}(g^x, g^y)$ (as duas chaves públicas assinadas)

A partir daqui ambas geram a chave partilhada.

De realçar que qualquer mensagem enviada que envolva assinaturas, é verificada na outra entidade antes do processo continuar.

Na célula seguinte vemos a criação dos parâmetros para as chaves do protocolo Diffie-Hellman e as chaves para o protocolo DSA.

```
In [2]: print('Gerando os parâmetros para o Diffie-Hellman . . .')
parameters_dh = dh.generate_parameters(generator=2, key_size=1024, backend=default_backend())
print(' . . . Parâmetros criados!')
print('')
print('Gerando agora os parâmetros para as assinaturas DSA . . .')
parameters_dsa = dsa.generate_parameters(key_size=1024, backend=default_backend())
print(' . . . Parâmetros criados!')
```

```
Gerando os parâmetros para o Diffie-Hellman . . .
. . . Parâmetros criados!
```

```
Gerando agora os parâmetros para as assinaturas DSA . . .
. . . Parâmetros criados!
```

Agora iremos implementar todos os métodos necessários que envolvam chaves DH.

```
In [3]: class DiffieHellman:
    def generate_DH_PrivateKey(self):
        private_key = parameters_dh.generate_private_key()
        return private_key

    def generate_DH_PublicKey(self, private_key):
        public_key = private_key.public_key()
        return public_key

    def generate_DH_PublicBytes(self, public_key):
        return public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo)
```

Naturalmente iremos também implementar todos os métodos necessários que envolvem chaves DSA e assinaturas digitais.

```
In [5]: class DSASignatures:
    def generate_DSA_PrivateKey(self):
        private_key = parameters_dsa.generate_private_key()
        return private_key

    def generate_DSA_PublicKey(self, private_key):
        public_key = private_key.public_key()
        return public_key

    def generate_DSA_PublicBytes(self, public_key):
        return public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo)

    def sign_message(self, message, own_private_key):
        signature = own_private_key.sign(
            message,
            hashes.SHA256()
        )
        return signature

    def verify_Signature(self, message, signature, other_public_key):
        other_public_key.verify(
            signature,
            message,
            hashes.SHA256()
        )
```

Na célula seguinte vamos gerar as chaves privadas e públicas do Emitter e do Receiver. Optamos por torna-las globais para evitar mais trocas de chaves (visto que não achamos que seria o principal objetivo).

```
In [6]: dsaSig = DSASignatures()  
  
emitter_dsa_privateKey = dsaSig.generate_DSA_PrivateKey()  
emitter_dsa_publicKey = dsaSig.generate_DSA_PublicKey(emitter_dsa_p  
privateKey)  
  
receiver_dsa_privateKey = dsaSig.generate_DSA_PrivateKey()  
receiver_dsa_publicKey = dsaSig.generate_DSA_PublicKey(receiver_dsa  
_privateKey)
```

Na comunicação entre os agentes foi implementada a cifra AES na qual foi usado o modo Counter Mode (CTR).

```
In [7]: class Encryption:
    def kdf(self, password, mySalt=None):
        if mySalt is None:
            auxSalt = os.urandom(16)
        else:
            auxSalt = mySalt
        kdf = PBKDF2HMAC(
            algorithm = hashes.SHA256(),    # SHA256
            length=32,
            salt=auxSalt,
            iterations=100000,
            backend=default_backend()        # openssl
        )
        key = kdf.derive(password)
        if mySalt is None:
            return auxSalt, key
        else:
            return key

    def mac(self, key, msg, tag=None):
        h = hmac.HMAC(key, hashes.SHA256(), default_backend())
        h.update(msg)
        if tag is None:
            return h.finalize()
        h.verify(tag)

    def encrypt(self, Ckey, Hkey, msg):
        iv = os.urandom(16)
        cipher = Cipher(algorithms.AES(Ckey), modes.CTR(iv), default_backend())
        encryptor = cipher.encryptor()
        ciphertext = encryptor.update(msg) + encryptor.finalize()
        tag = self.mac(Hkey, ciphertext)
        return iv, ciphertext, tag

    def decrypt(self, Ckey, iv, msg):
        cipher = Cipher(algorithms.AES(Ckey), modes.CTR(iv), default_backend())
        decryptor = cipher.decryptor()
        cleant = decryptor.update(msg) + decryptor.finalize()
        return cleant
```

Emitter

O Emitter é responsável por enviar mensagens ao Receiver. Este apenas recebe mensagens do Receiver quando estes estão no protocolo Diffie-Hellman. Este agente foi dividido em dois processos diferentes. Um que trata do protocolo Diffie-Hellman e outro para enviar as mensagens.

Emitter_DH

Este método é o responsável por representar o Emitter na troca de chaves DiffieHellman.

```

In [10]: def Emitter_DH(conn):
    diffieHellman = DiffieHellman()
    dsaSign = DSASignatures()
    print('EmitterDH: Iniciar Processo de DiffieHellman')

    emitter_dh_privateKey = diffieHellman.generate_DH_PrivateKey()
    #print('Emitter: Chave privada criada')
    emitter_dh_publicKey = diffieHellman.generate_DH_PublicKey(emitter_dh_privateKey)
    #print('Emitter: Chave pública criada')
    print('EmitterDH: Enviando a minha chave pública')
    emitter_dh_public_bytes_key = diffieHellman.generate_DH_PublicBytes(emitter_dh_publicKey)
    conn.send(emitter_dh_public_bytes_key)

    while True:
        print('EmitterDH: Esperando a chave pública do Receiver')
        pubkey = conn.recv()
        break

    while True:
        print('EmitterDH: Esperando a assinatura da chave pública')
        signature = conn.recv()
        break

    try:
        aux = emitter_dh_public_bytes_key + pubkey
        dsaSign.verify_Signature(aux, signature, receiver_dsa_publicKey)

        print('EmitterDH: Assinatura válida!')
        receiver_dh_public_key = pubkey
        print('EmitterDH: Já obtive a chave pública do Receiver')
        sign = dsaSign.sign_message(aux, emitter_dsa_privateKey)
        conn.send(sign)
    except (InvalidSignature):
        print('EmitterDH: Assinatura não válida! Conexão fechada!')

    while True:
        msg = conn.recv()
        break

    while True:
        sig = conn.recv()
        break

    try:
        dsaSign.verify_Signature(msg, sig, receiver_dsa_publicKey)
        print('EmitterDH: Assinatura válida!')

        emitter_dh_shared_key = emitter_dh_privateKey.exchange(serialization.load_pem_public_key(
            receiver_dh_public_key,
            backend = default_backend()))
        print('EmitterDH: Shared Key criada!')
        return emitter_dh_shared_key
    except (InvalidSignature):
        print('Emitter: Assinatura inválida! Conexão fechada!')

```

Receiver

O Receiver é responsável por receber as mensagens do Emitter, decifra-las e dar print. Este também foi dividido em dois processos, um para a troca de chaves e outra para receber as mensagens.


```

In [11]: def Receiver_DH(conn):
    diffieHellman = DiffieHellman()
    dsaSigns = DSASignatures()
    print('ReceiverDH: Iniciar Processo de DiffieHellman.')

    receiver_dh_privateKey = diffieHellman.generate_DH_PrivateKey()
    #print('Receiver: Chave privada criada.')
    receiver_dh_publicKey = diffieHellman.generate_DH_PublicKey(receiver_dh_privateKey)
    #print('Receiver: Chave pública criada - - - ')
    receiver_dh_public_bytes_key = diffieHellman.generate_DH_PublicBytes(receiver_dh_publicKey)

    #print('Receiver: Esperando chave pública do Emitter')
    while True:
        emitter_dh_public_key = conn.recv()
        #print('Receiver: Já obtive a chave pública do Emitter')
        #print(emitter_dh_public_key)
        break;

    publicKeys = emitter_dh_public_key + receiver_dh_public_bytes_key
    sign = dsaSigns.sign_message(publicKeys, receiver_dsa_privateKey)

    print('ReceiverDH: Enviando a minha chave pública')
    conn.send(receiver_dh_public_bytes_key)
    conn.send(sign)

    while True:
        ''' Esperando pela assinatura do emitter (ultimo passo do Diffie-Hellman)'''
        msg = conn.recv()
        break;

    try:
        dsaSigns.verify_Signature(publicKeys,msg,emitter_dsa_publicKey)

        print('ReceiverDH: Assinatura válida!')
        print('\n\n Acordo Realizado!\n\n')
        msg = b'ACORDO REALIZADO!'
        sig = dsaSigns.sign_message(msg,receiver_dsa_privateKey)
        conn.send(msg)
        conn.send(sig)
    except:
        print('Receiver DH: Assinatura inválida')

    receiver_dh_shared_key = receiver_dh_privateKey.exchange(serialization.load_pem_public_key(
        emitter_dh_public_key,
        backend=default_backend()))
    print('ReceiverDH: Shared Key criada!')
    return receiver_dh_shared_key

```

```

In [16]: def Emitter(conn):
    shared_key = Emitter_DH(conn)
    # print('E: sharedKey- ' + str(shared_key))
    time.sleep(2)
    print('Emitter: Tenho o segredo compartilhado.\n\n')

    encryption = Encryption()
    dsaSig = DSASignatures()

    text1 = b'Ola! Vamos enviar 4 mensagens(sendo esta a primeira)
para o Receiver!'
    text2 = b'Todas estas mensagens serao encriptadas. Sera ele cap
az de as desencriptar?'
    text3 = b'Cada criptograma sera autenticado com um HMAC e vai a
ssinado com a minha chave privada DSA'
    text4 = b'Se correr bem, todas estas 4 mensagens foram printada
s!'
    text5 = b'Assinado: Emitter'
    text6 = b'PS: Afinal foram 6'
    msgs=[text1,text2,text3,text4,text5,text6]

    i = 0
    while(i < 6):
        salt,key = encryption.kdf(shared_key)
        Ckey = key[0:16]
        #print('E: Ckey- ' + str(Ckey))
        Hkey = key[16:32]
        #print('E: Hkey- ' + str(Hkey))
        iv,cipher_text, tag = encryption.encrypt(Ckey,Hkey, msgs[i]
)
        sig = dsaSig.sign_message(cipher_text, emitter_dsa_privateK
ey)

        conn.send(salt)
        #print('E: SALT- ' + str(salt))
        conn.send(iv)
        #print('E: IV- ' + str(iv))
        conn.send(cipher_text)
        #print('E: MSG- ' + str(cipher_text))
        conn.send(tag)
        #print('E: TAG- ' + str(tag))
        conn.send(sig)
        #print('E: SIG- ' + str(sig))
        time.sleep(2)
        i+=1
    print('ALL MESSAGES SENDED!')

    #conn.send(b'welelele')

```

```

In [17]: max_msg = 6
def Receiver(conn):
    sharedKey = Receiver_DH(conn)
    #print('R: sharedKey- ' + str(sharedKey))
    time.sleep(2)
    print('Receiver: Tenho o segredo compartilhado.\n\n')
    encryption = Encryption()
    dsaSig = DSASignatures()
    i = 0
    while (i < max_msg):
        '''
        Esperemos sempre 5 mensagem por cada criptograma. Um com o
        salt, outra com o iv, outra com a tag,
        outra com a assinatura e outra com a mensagem cifrada
        '''

        while True: #salt
            mySalt = conn.recv()
            #print('R: SALT- ' + str(mySalt))
            while True: #iv
                iv = conn.recv()
                #print('R: IV- ' + str(iv))
                while True: #mensagem
                    msg = conn.recv()
                    #print('R: MSG- ' + str(msg))
                    while True: #tag
                        tag = conn.recv()
                        #print('R: TAG- ' + str(tag))
                        while True: #sign
                            sig = conn.recv()
                            # print('R: SIG- ' + str(sig))
                            break
                        break
                    break
                break
            break
        break

    try:
        dsaSig.verify_Signature(msg, sig, emitter_dsa_publicKey
)

        key = encryption.kdf(sharedKey, mySalt)
        Ckey = key[0:16]
        Hkey = key[16:32]
        #print('R: CKEY- ' + str(Ckey))
        #print('R: HKEY- ' + str(Hkey))
        try:
            encryption.mac(Hkey, msg, tag)
            plaintext = encryption.decrypt(Ckey, iv, msg)
            print(plaintext)
        except(InvalidSignature):
            print('Tag inválida!')
    except(InvalidSignature):
        print('Assinatura inválida!')

    i += 1

    print('MAX MESSAGE REACHED')

```

```
In [18]: def main():  
         PipeCommunication(Emitter,Receiver,timeout=600).run()
```

```
In [19]: main()
```

```
EmitterDH: Iniciar Processo de DiffieHellman  
ReceiverDH: Iniciar Processo de DiffieHellman.  
EmitterDH: Enviando a minha chave pública  
EmitterDH: Esperando a chave pública do Receiver  
ReceiverDH: Enviando a minha chave pública  
EmitterDH: Esperando a assinatura da chave pública  
EmitterDH: Assinatura válida!  
EmitterDH: Já obtive a chave pública do Receiver  
ReceiverDH: Assinatura válida!
```

Acordo Realizado!

```
ReceiverDH: Shared Key criada!  
EmitterDH: Assinatura válida!  
EmitterDH: Shared Key criada!  
Receiver: Tenho o segredo compartilhado.
```

Emitter: Tenho o segredo compartilhado.

```
b'Ola! Vamos enviar 4 mensagens(sendo esta a primeira) para o Receiver!  
b'Todas estas mensagens serao encriptadas. Sera ele capaz de as de  
sencriptar?'  
b'Cada criptograma sera autenticado com um HMAC e vai assinado com  
a minha chave privada DSA'  
b'Se correr bem, todas estas 4 mensagens foram printadas!'  
b'Assinado: Emitter'  
b'PS: Afinal foram 6'  
MAX MESSAGE REACHED  
ALL MESSAGES SENDED!
```