# Trabalho Prático 0

Grupo 17, constituído por:

-- Joana Castro e Sousa, PG47282

-- Tiago Taveira Gomes, PG47702

## Pergunta 1:

### Criar uma comunicação privada assíncrona entre um agente Emitter e um agente Receiver que cubra os seguintes aspectos:

a) Autenticação do criptograma e dos metadados (associated data). Usar uma cifra simétrica num modo HMAC que seja seguro contra ataques aos “nounces” .

b) Os “nounces” são gerados por um gerador pseudo aleatório (PRG) construído por um função de hash em modo XOF.

c) O par de chaves cipher_key, mac_key , para cifra e autenticação, é acordado entre agentes usando o protocolo DH com autenticação dos agentes usando assinaturas DSA.

In [ ]:
```
#imports
import os
import time

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

import multiprocessing
from multiprocessing import Pipe, Process

# utilizar uma comunicação por pipes para simular uma comunicação privada assíncrona entre um agente Emitter e um agente Receiver
class PipeCommunication:
    def __init__(self,left, right, timeout=None):
        '''
            Classe responsável por ligar 2 entidades através de um Pipe para poderem comunicar entre si.
            A cada entidade será atribuída uma extremidado do pipe.
            Será criado um processo para cada entidade onde o processo terá como alvo a entidade respetiva e
                passar-lhe-á como argumento a extremidade da conexão que lhe é correspondente.
        '''
        left_end, right_end = Pipe()
        self.timeout = timeout
        self.left_process = Process(target=left, args=(left_end,))
        self.right_process = Process(target=right, args=(right_end,))
    def run(self):
        self.left_process.start()
        self.right_process.start()
        self.left_process.join(self.timeout)
        self.right_process.join(self.timeout)
```

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 || SIG(g^x,g^y) (a sua chave pública || as duas chaves públicas assinadas)

3 - Emitter envia a Receiver: SIG(g^x,g^y) (as duas chaves públicas assinadas)

Neste ponto, ambos geram a chave partilhada. De realçar que qualquer mensagem enviada que envolva assinaturas, é verificada na outra entidade antes do processo continuar.

Deste modo, comecemos com a criação dos parâmetros para as chaves do protocolo DH e as chaves para o protocolo DSA.

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

```
Gerar os parâmetros para o DH.
Parâmetros criados!

Gerar os parâmetros para as assinaturas DSA.
Parâmetros criados!
```

De seguida, apresentamos a implementação de todos os métodos necessários que involvam chaves DH.

In [ ]:
```
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)
```

A implementação de todos os métodos necessários que involvem chaves DSA e assinaturas digitais.

In [ ]:
```
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()
        )
```

Gerar as chaves privadas e públicas do Emitter e do Receiver. Nesta fase, foi optado tornar estas chaves como variáveis globais, de forma a evitar as trocas dessas chaves, uma vez que não achamos que seria esse o principal objetivo.

In [ ]:
```
dsaSig = DSASignatures()

emitter_dsa_privateKey = dsaSig.generate_DSA_PrivateKey()
emitter_dsa_publicKey = dsaSig.generate_DSA_PublicKey(emitter_dsa_privateKey)

receiver_dsa_privateKey = dsaSig.generate_DSA_PrivateKey()
receiver_dsa_publicKey = dsaSig.generate_DSA_PublicKey(receiver_dsa_privateKey)
```

Também, apresentamos métodos que serão comuns tanto para o Receiver como para o Emitter. Na comunicão entre os agentes foi implementeada a cifra simétrica AES, usando autenticação de cada criptograma com HMAC, na qual foi usado o modo CTR (counter) para ser seguro contra ataques aos 'nounces'.

In [ ]:
```
class Encription:
    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


    # a função de Hash que calcula a hash de um dado input. resultado é o "nounce", construído em XOF (com SHA256)
    def Hash(self, s):
        digest = hashes.Hash(hashes.SHA256(),backend=default_backend())
        digest.update(s)
        return digest.finalize()

    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 encript(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 decript(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 quem envia as mensagens ao Receiver. Apenas receberá mensagens do Receiver quando estiverem no protocolo Diffie-Hellman.

Assim, este agente foi dividido em dois processos diferentes: um que trata do protocolo Diffie-Hellman e outro para enviar as mensagens.

### Emitter_DH

In [ ]:
```
# método responsável por representar o Emitter na troca de chaves DiffieHellman.
def Emitter_DH(conn):
    diffieHellman = DiffieHellman()
    dsaSign = DSASignatures()
    print('Emitter_DH: 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('Emitter_DH: 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('Emitter_DH: Esperando a chave pública do Receiver')
        pubkey = conn.recv()
        break
    while True:
        print('Emitter_DH: 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('Emitter_DH: Assinatura válida!')
        receiver_dh_public_key = pubkey
        print('Emitter_DH: Já obtive a chave pública do Receiver')
        sign = dsaSign.sign_message(aux,emitter_dsa_privateKey)
        conn.send(sign)
    except(InvalidSignature):
        print('Emitter_DH: 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('Emitter_DH: Assinatura válida!')

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

## Receiver:

O Receiver é responsável por receber as mensagens do Emitter, decifra-las e dar print.

Assim, este também foi dividido em dois processos: um para a troca de chaves e outra para receber as mensagens.

### Receiver_DH

In [ ]:
```
def Receiver_DH(conn):
    diffieHellman = DiffieHellman()
    dsaSigns = DSASignatures()
    print('Receiver_DH: 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('Receiver_DH: Enviando a minha chave pública')
    conn.send(receiver_dh_public_bytes_key)
    conn.send(sign)

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

    try:
        dsaSigns.verify_Signature(publicKeys,msg,emitter_dsa_publicKey)
        print('Receiver_DH: 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('Receiver_DH: Shared Key criada!')
    return receiver_dh_shared_key
```

Por fim, as implementações dos Emitter e Receiver

## Emitter

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

    encription = Encription()
    dsaSig = DSASignatures()

    text1 = b'Mensagem 1'
    text2 = b'Mensagem 2'
    text3 = b'Mensagem 3'
    text4 = b'Mensagem 4'
    text5 = b'Mensagem 5'
    text6 = b'Mensagem 6'
    msgs=[text1,text2,text3,text4,text5,text6]
```

```
    i = 0
    while(i < 6):
        salt,key = encription.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 = encription.encript(Ckey,Hkey, msgs[i])
        sig = dsaSig.sign_message(cipher_text, emitter_dsa_privateKey)
        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('Todas as mensagens enviadas!')
```

## Receiver

In [ ]:

```
max_msg = 6
def Receiver(conn):
    sharedKey = Receiver_DH(conn)
    #print('R: sharedKey- ' + str(sharedKey))
    time.sleep(2)
    print('Receiver: Tenho o segredo partilhado.\n\n')
    encription = Encription()
    dsaSig = DSASignatures()
    i = 0
    while (i < max_msg):
        #Esperar 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
```

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

        i += 1

    print('Todas as mensagens chegaram!')
```

## Definição de uma função de teste

In [ ]:

```
def main():
    PipeCommunication(Emitter,Receiver,timeout=600).run()


multiprocessing.set_start_method("fork") # "Python 3.8 on MacOS by default now uses "spawn" instead of "fork" as start method for new processes"
if __name__ == "__main__":
    main()
```

```
Emitter_DH: Iniciar Processo de DiffieHellman
Emitter_DH: Enviando a minha chave públicaReceiver_DH: Iniciar Processo de DiffieHellman.

Emitter_DH: Esperando a chave pública do Receiver
Receiver_DH: Enviando a minha chave pública
Emitter_DH: Esperando a assinatura da chave pública
Emitter_DH: Assinatura válida!
Emitter_DH: Já obtive a chave pública do Receiver
Receiver_DH: Assinatura válida!


 Acordo Realizado!


Emitter_DH: Assinatura válida!Receiver_DH: Shared Key criada!

Emitter_DH: Shared Key criada!
Receiver: Tenho o segredo partilhado.


Emitter: Tenho o segredo partilhado.


b'Mensagem 1'
b'Mensagem 2'
b'Mensagem 3'
b'Mensagem 4'
b'Mensagem 5'
b'Mensagem 6'
Todas as mensagens chegaram!
Todas as mensagens enviadas!
```