Trabalho 1

Neste trabalho pretende-se que se efetue a construção idêntica à executada na aula, num ficheiro fornecido pelo professor, relativamente à criação de uma comunicação assincrona cifrada entre um agente **Emitter** e um agente **Receiver**. Para isso, foi necessária a criação de dois processos, neste caso left e right, com um mecanismo de comunicação básico entre elas, o **Pipe**.

Estes processos recebem como parâmetro uma conecção entre eles: o Conn. De seguida, um dos lados envia a mensagem e fecha a ligação, e do outro lado a mensagem é recebida, é feito o print da mesma, e a ligação é fechada.

Para que ambos se juntem, é necessário criar uma Classe em que cada instante é uma ligação binária. Esta ligação tem variáveis de estado, fase de iniciação e métodos, e é preciso colocar o "self" porque é relativo àquela própria instância.

Assim, é importado o Process e o Pipe do *multiprocessing*, para criar a ligação e fornecer os dois lados da mesma.

```
import os
from getpass import getpass
from multiprocessing import Process, Pipe

#criação do canal de comunicação
class BiConn(object):
    def __init__(self,left,right,timeout=None):
        """
        left : a função que vei ligar ao lado esquerdo do Pipe
        right: a função que vai ligar ao outro lado
        timeout: (opcional) numero de segundos que aguarda pela terminação do processo
        """
        left_end, right_end = Pipe()
        self.timeout=timeout
        self.lproc = Process(target=left, args=(left_end,))       # os processos
ligados ao Pipe
        self.rproc = Process(target=right, args=(right_end,))
        self.left  = lambda : left(left_end)                       # as funções ligadas
já ao Pipe
        self.right = lambda : right(right_end)

    def auto(self, proc=None):
        if proc == None:             # corre os dois processos independentes
            self.lproc.start()
            self.rproc.start()
            self.lproc.join(self.timeout)
            self.rproc.join(self.timeout)
        else:                        # corre só o processo passado como parâmetro
            proc.start(); proc.join()

    def manual(self):   #  corre as duas funções no contexto de um mesmo processo
Python
        self.left()
```

```
        self.right()
```

Normalmente, os agentes humanos que escolhem as *passwords*, estão muito longe da aleatoriedade, é necessário utilizar Key Derivation Fuctions, KDFs, que são funções que transformam as passwords, isto é, que aumentam a entropia final das palavras-passe, tornando-as chaves reconhecíveis e memorizáveis por máquinas e não por humanos. Ao mesmo tempo, protegem as chaves armazenadas em memórias de ataques de força bruta desta forma o sistema torna-se razoavelmente mais seguro.

```
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, hmac

default_algorithm = hashes.SHA256 # seleciona-se um dos vários algorimos implementados
na package

#funções necessárias para a cifra e autenticação
def kdf(salt):
    return PBKDF2HMAC(
        algorithm=default_algorithm(),   # SHA256
        length=32,
        salt=salt,
        iterations=100000,
        backend=default_backend()        # openssl
        )


def Hash(s):
    digest = hashes.Hash(default_algorithm(),backend=default_backend())
    digest.update(s)
    return digest.finalize()
```

De seguida temos a autenticação da informação, conforme é pedido na alínea a), que é feita através de **funções Hash**. A representação e a autenticação dos dados é de grande importância principalmente quando a quantidade de informação é muito elevada, uma vez que esta é uma forma fiável e eficiente de representação com "poucos" bits. É também criada uma **função hmac** que é uma ferramenta para calcular *Message Authentication Codes* usando uma função Hash juntamente com uma secret key.

```
def hmacfuncao(key,mensagem,tag=None):


    h=hmac.HMAC(key,hashes.SHA256(),backend=default_backend())
    h.update(mensagem)
    if tag== None:
        return h.finalize()
    h.verify(tag)
```

```
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
```

Neste import, é selecionada da package Cryptography, o modo de autenticação de cifras que pretendemos utilizar neste trabalho, AESGCM, ou seja, *Galois Counter Mode*, um algoritmo de cifra por blocos.

OPERAÇÃO ASSÍNCRONA

Neste regime, a cifra recebe um plaintext (finito) e produz um ciphertext (exemplo: chamada telefónica).

Através da package *os*, utilizando o urandom, é gerado um parâmetro,my_para, que se trata de um número que adiciona aleatoriedade extra à chave gerada, interna ao sistema.

Aquando da criação das funções **Emitter1** e **Receiver1**, que recebe os inputs *password* e *mensagem*, vemo-nos forçados a serializar a estrutura de dados em strings de bits e só depois calcular o seu código de Hash, visto que esta função apenas recebe parâmetros do tipo booleano (1 e 0).

```
my_salt=os.urandom(16)
#permite introduzir aleatoriedade á password

def Emitter1(conn):
    password=bytes(getpass('Palavra passe do emissor:'),'utf-8')
    mensagem = bytes(input('Escreva a mensagem que quer cifrar:'), 'utf-8')
    #cria uma mensagem e uma password

    try:
        key = kdf(my_salt).derive(password)
        tag=hmacfuncao(key,key)
        aesgcm=AESGCM(key)
        dadosassociados=tag
        ciphertext=aesgcm.encrypt(my_salt,mensagem,dadosassociados)
        print(ciphertext)

        obj = {'mess' : ciphertext , 'tag': tag}
        conn.send(obj)
    except:
        print("Erro no emissor")
```

```
    conn.close()

def Receiver1(conn):

    password = bytes(getpass('Palavra passe do recetor: '),'utf-8')

# verify password
    try:
        # Recuperar a informação
        obj = conn.recv()
        ciphertext = obj['mess']
        tag = obj['tag']

        # Gerar a chave
        key = kdf(my_salt).derive(password)

        # Verificação prévia e decifrar o ciphertext obtendo a mensagem
        if tag == hmacfuncao(key,key):
            aesgcm=AESGCM(key)
            dadosassociados=obj['tag']
            mensagem=aesgcm.decrypt(my_salt,ciphertext,dadosassociados)
            print('Mensagem decifrada:',mensagem)
        else:
            raise
    except:
        print('FAIL')

    conn.close()
```

OPERAÇÃO SÍNCRONA

Nos sistemas de comunicação, são utilizadas operações síncronas quando é necessário ter em simultâneo o processamento da cifra e a extração do seu output, mesmo sem conhecer completamente o input. Deve-se ter em atenção que este tipo de operações tem sempre associado um tempo de latência, uma vez que o input é iniciado num instante t_0 e o output, iniciado depois do período de latência, num instante t_1, começa sem que o primeiro termine.

```
import io
from io import BytesIO
mysalt=os.urandom(16)


def Emitter2(conn):

    password=bytes(getpass('Palavra passe do emissor:'),'utf-8')
    mensagem = bytes(input('Escreva a mensagem que quer cifrar:'), 'utf-8')
```

```
    key = kdf(my_salt).derive(password)
    tag=hmacfuncao(key,key)


    inp_stream=io.BytesIO(mensagem)
    inp_buffer=bytearray(64)


    while True:
        ler=inp_stream.readinto(inp_buffer)
        chunk=bytes(inp_buffer)
        aesgcm=AESGCM(key)
        dadosassociados=tag
        ciphertext=aesgcm.encrypt(my_salt,chunk,dadosassociados)


        print('Texto cifrado enviado:',ciphertext)

        obj = {'mess' : ciphertext , 'tag': tag}
        conn.send(obj)


        if not ler:
            break

    conn.close()


def Receiver2(conn):


    password = bytes(getpass('Palavra passe do recetor: '),'utf-8')

    try:
        key=kdf(my_salt).derive(password)
        out_stream=io.BytesIO()

        while True:
            obj= conn.recv()
            ciphertext = obj['mess']
            tag = obj['tag']


            if tag==hmacfuncao(key,key):
                print('Chunk cifrado: ', ciphertext, '/n')
                out_stream.write(ciphertext)


                aesgcm=AESGCM(key)
                dadosassociados=obj['tag']
                mensagem=aesgcm.decrypt(my_salt, ciphertext, dadosassociados)
                mensagemstr=str(mensagem,'utf-8')
                print('Mensagem decifrada: ',mensagemstr, '/n')
```

```
            else:
                raise

    except:
        print('Falha no recetor')



    conn.close()
```

SÍNCRONO OU ASSÍNCRONO

```
metodo=input("Escolha regime síncrono (digite: sinc) ou regime assíncrono (digite:
assinc)?")
if metodo=='assinc':
    BiConn(Emitter1,Receiver1, timeout=40).manual()
if metodo=='sinc':
    BiConn(Emitter2,Receiver2, timeout=40).manual()
```

```
Escolha regime síncrono (digite: sinc) ou regime assíncrono (digite: assinc)?assinc
Palavra passe do emissor:·····
Escreva a mensagem que quer cifrar:ola
b'\x9a^\xeaQ\x87\x97LQ\xed\xf4\xc1\xafO\x02\xa30}\xaf"'
Palavra passe do recetor: ·····
Mensagem decifrada: b'ola'
```