

# TP0

---

## 1.

---

A interação entre as entidades 'Emitter' e 'Receiver' é feita com 2 threads que comunicam a partir de uma Queue bloqueante.

Alguns imports relevantes para o processo:

```
from threading import Thread
from queue import Queue
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
import os
from cryptography.hazmat.primitives.ciphers import (Cipher, algorithms, modes)
```

Primeiro definimos as funções para cifrar e decifrar as mensagens.

```
def encrypt(message, key, metadata):
    iv = os.urandom(12)
    encryptor = Cipher(
        algorithms.AES(key),
        modes.GCM(iv)
    ).encryptor()
    encryptor.authenticate_additional_data(metadata)
    ciphertext = encryptor.update(message) + encryptor.finalize()
    return (iv, ciphertext, encryptor.tag)

def decrypt(iv, cipher, key, tag, metadata):
    decryptor = Cipher(
        algorithms.AES(key),
        modes.GCM(iv, tag),
    ).decryptor()
    decryptor.authenticate_additional_data(metadata)
    return decryptor.update(cipher) + decryptor.finalize()
```

Para o 'Emitter' temos uma thread que envia ao 'Receiver' o resultado da cifragem da mensagem "Hello World".

```
# Emitter thread function
def emitter(queue):
    global key
    msg = b"Hello World!"
    iv_ct_tag = encrypt(msg, key, b"METADATA")
    queue.put(iv_ct_tag)
    print("[Emitter] Sent > {}".format(iv_ct_tag[1]))
```

Para o 'Receiver' temos uma thread que irá receber e decifrar o criptograma que o 'Emitter' enviou.

```
# Receiver thread function
def receiver(queue):
    global key
```

```

iv, ct, tag = queue.get()
print("[Receiver] Received > {}".format(ct))
msg = decrypt(iv, ct, key, tag, b"METADATA")
print("[Receiver] Decrypted > {}".format(msg))

```

Para gerar a chave criptográfica para utilizar na cifração simétrica lemos do STDIN os bytes de uma string que vai ser usada como password.

```

pwd = bytes(input("Shared Password > "), "utf-8")

```

E derivamos a chave a ser utilizada usando PBKDF, e obtemos a chave na variável 'key'.

```

kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=16,
    salt=b"\x00"*16,
    iterations=100000
)
key = kdf.derive(pwd)

```

Para começar toda a interação entre o 'Emitter' e 'Receiver' criamos as threads com as funções e argumentos respetivos e começamos as mesmas.

```

q = Queue(5)

# Create emitter and receiver threads
e = Thread(target=emitter, args=(q,))
r = Thread(target=receiver, args=(q,))

# Start both threads
e.start()
r.start()

# Wait for them to finish to exit program
e.join()
r.join()
print("[INFO] > Finished program execution")

```

## 2.

Definimos um PRG a partir de SHAKE256 e parametrizamos com o valor n para gerar n blocos de 64 bits (8 bytes)

```

from cryptography.hazmat.primitives.hashes import (Hash, SHAKE256)

def prg(password: bytes, n: int) -> bytes:
    alg = Hash(SHAKE256(8*n)) # 64 bits == 8 bytes
    alg.update(password)
    return alg.finalize()

```

De seguida definimos as funções de 'encrypt' e 'decrypt', que para este esquema são implementadas da mesma forma

Como especificado, este esquema irá recorrer ao XOR da mensagem com a chave aleatória gerada.

Como python não tem 'overload' do operador de XOR para o tipo 'bytes' então definimos uma função auxiliar `bytes_xor` para concretizar esta ação.

Nota: O tamanho da mensagem deve ter tamanho divisível por 8 (64 bits). Como este esquema é equivalente a uma cifra de Vernam temos que a mensagem e a chave devem ter o mesmo número de blocos

```
def bytes_xor(a: bytes, b: bytes):
    if len(a) != len(b):
        raise ValueError("bytes arguments must have the same length")
    return bytes([_a ^ _b for _a, _b in zip(a, b)])

def encrypt(plaintext: bytes, key: bytes):
    return bytes_xor(plaintext, key)

def decrypt(ciphertext: bytes, key: bytes):
    return bytes_xor(ciphertext, key)
```

Finalmente, testamos esta implementação com o bloco de texto `b"olaolaol"` que tem 1 palavra de 64 bits

```
key = prg(b"ola mundo cruel", 1)
ct = encrypt(b"olaolaol", key)
msg = decrypt(ct, key)
print("key", key)
print("ct", ct)
print("msg", msg)
```