Trabalho Prático 0

Grupo 17, constituído por:

-- Joana Castro e Sousa, PG47282

-- Tiago Taveira Gomes, PG47702

Pergunta 2:

a) Criar um gerador pseudo-aleatório do tipo XOF (“extended output function”) usando o SHAKE256, para gerar uma sequência de palavras de 64 bits.

i) O gerador deve poder gerar até um limite de 2^n palavras (n é um parâmetro) armazenados em long integers do Python.

ii) A “seed” do gerador funciona como cipher_key e é gerado por um KDF a partir de uma “password” .

iii) A autenticação do criptograma e dos dados associados é feita usando o próprio SHAKE256.

In [ ]:
```
# imports
import os

from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import padding

# N necessário para gerar as palavras
N = 5
BLOCK_SIZE = 8 # 64 bits = 8 bytes
```

Utilizar um KDF para gerar através de uma password

In [ ]:
```
def derivate_key(password, salt):
    # derivar
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        )
    key = kdf.derive(password)
    return key
```

Desta maneira, é possível criar o PRG do tipo XOF usando o SHAKE256, para sequências de palavras de 64 bits

In [ ]:
```
def prg(seed):
    digest = hashes.Hash(hashes.SHAKE256(BLOCK_SIZE * pow(2,N)))
    digest.update(seed)
```

```
    words = digest.finalize()
    return words
```

b) Defina os algoritmos de cifrar e decifrar : para cifrar/decifrar uma mensagem com blocos de 64 bits, os “outputs” do gerador são usados como máscaras XOR dos blocos da mensagem. Essencialmente a cifra básica é uma implementação do “One Time Pad”.

Para cifrar, é necessário ter em atenção se é necessário efetuar um padding ou não

In [ ]:
```
def encode(key,message):
    ct = b''
    padder = padding.PKCS7(64).padder()
    # Adiciona padding ao último bloco de bytes da mensagem de modo a esta ter tamanho múltiplo do bloco
    padded = padder.update(message) + padder.finalize()
    # Divide mensagem em blocos de 8 bytes
    p = [padded[i:i+BLOCK_SIZE] for i in range(0, len(padded), BLOCK_SIZE)]
    # XOR dos bytes do bloco da mensagem com os bytes do bloco de palavras chave
    for x in range (len(p)): # Percorre blocos do texto limpo
        for index, byte in enumerate(p[x]): # Percorre bytes do bloco do texto limpo
            ct += bytes([byte ^ key[x:(x+1)*BLOCK_SIZE][index]])
    return ct
```

Já no caso de decifrar, é necessário dividir o texto por blocos de 64 bits e fazer "unpadding" quando necessário

In [ ]:
```
def decode(key, ct):
    pt = b''
    # Divide texto cifrado em blocos de 8 bytes
    p = [ct[i:i+BLOCK_SIZE] for i in range(0, len(ct), BLOCK_SIZE)]
    # XOR dos bytes do bloco do texto cifrado com os bytes do bloco de palavras chave
    for x in range (len(p)): # Percorre blocos do texto cifrado
        for index, byte in enumerate(p[x]): # Percorre bytes do bloco do texto cifrado
            pt += bytes([byte ^ key[x:(x+1)*BLOCK_SIZE][index]])
    # Algoritmo para retirar padding para decifragem
    unpadder = padding.PKCS7(64).unpadder()
    # Retira bytes adicionados
    unpadded = unpadder.update(pt) + unpadder.finalize()
    return unpadded.decode("utf-8")
```

Por fim, podemos finalmente testar a cifra implementada

In [ ]:
```
def main():
    # a password que queremos ter partilhada
    password = "uma password"
    # a salt necessária para derivar a chave
    salt = os.urandom(16)
    # gerar a 'seed'
    seed = derivate_key(password.encode("utf-8"), salt)
    # assim é possível gerar a chave com essa seed
    key = prg(seed)
    # e o cypher_text
    ct = encode(key, "Segredo".encode("utf-8"))
    print("Cypher_text: "); print(ct)
    print("")
    print("Texto limpo: "); print(decode(key, ct))
```

```
if __name__ == "__main__":
    main()
```

```
Cypher_text:
b'\x08\xdd2\xe1;\xd8\xb2\x9e'

Texto limpo:
Segredo
```