# TP1_Ex2

March 7, 2023

## 1 Exercício 2:

Neste exercício foi nos pedido que implementassemos um algoritmo de codificação com um modo semelhante a uma cifra *one-time pad*. Por outras palavras, é esperado que seja desenvolvida uma cifra que, dada uma *"chave"*, deverá realizar a operação XOR entre a mensagem que se pretende partilhar e a *"chave"*.

Esta cifra deve apresentar um comportamento semelhante ao das cifras em bloco, dado que a mensagem deverá ser dividida em blocos de 64 bits, e cada um destes blocos deverá ser cifrado com um bloco da *"chave"*.

Esta *"chave"* será obtida utilizando um gerador definido recorrendo à função hash SHAKE-256. Este gerador terá uma seed definida em função de uma password.

Adicionalmente, esta cifra deverá implementar uma autenticação de metadados e, para tal, iremos utilizar o algoritmo *HMAC*. Este algortimo permite-nos fazer autenticação da mensagem. Uma vez que o algoritmo utilizado não requer o uso de metadados, iremos criar um pacote que associe metadados ao texto cifrado e fazer a autenticação deste.

Para resolvermos este exercicio iremos utilizar funções que se encontram disponiveis na biblioteca **Cryptography**.

[1]:
```
## Imports:
import os
from pickle import dumps, loads
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.hashes import SHAKE256
from cryptography.hazmat.primitives import hashes, padding, hmac
```

### 1.1 Gerador de uma *seed*

Como já foi mencionado para a nossa implementação, iremos precisar de obter uma seed que será aplicada a um gerador pseudo-aleatório. O output deste será, posteriormente, utilizado numa operação XOR.

Para obtermos esta seed, utilizaremos um algoritmo KDF (key derivation function). A este algoritmo será passada uma password escolhida pelo utilizador. Para além disto, este algoritmo deverá receber um length e um salt. O salt será um número random, e o length será 32. A seed gerada será a nossa **cipher_key**.

[2]:
```
# Função que permite derivar uma seed de uma password:
def generateSeed(pwd):
    salt = os.urandom(16)
    length = 32 # Length in bytes das chaves derivadas

    kdf = PBKDF2HMAC(algorithm=hashes.SHA3_256(),
                     length=length,
                     salt=salt,
                     iterations = 480000,
                    )
    seed = kdf.derive(pwd)

    return seed
```

### 1.2 Gerador pseudo-aleatório

Como já referimos, será necessário utilizar um gerador pseudo-aleatório para obter uma "chave" que será utilizado na codificação/descodificação da mensagem. Para tal, iremos utilizar como gerador pseudo-aleatório a função hash SHAKE-256. A esta função será passada a cipher_key (seed). A utilização de uma seed é crucial, uma vez que será necessário que ambos os participantes possam obter o mesmo output de um gerador, dado que se estes forem diferentes poderá não ser possível descodificar a mensagem.

Este gerador deverá gerar 2^n palavras, tal como foi especificado no enunciado. Sabendo que um bloco terá o tamanho de 64 bits (8bytes) e cada palavra deverá ter o mesmo tamanho que um bloco, precisamos, portanto, de um gerador cujo output é 2^n * 8 bytes.

[3]:
```
# Função com um gerador pseudo-aleatório que gera um output com 2^n * 8 bits
def generator(n, seed):
    digest = hashes.Hash(hashes.SHAKE256(2**n*8))

    digest.update(seed) # Define seed

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

### 1.3 Padding

Dado que estamos a trabalhar com uma cifra simétrica de blocos, uma parte essencial do modo desta cifra é garantir que todos os blocos tem o mesmo tamanho fixo. Neste caso, o tamanho fixo será 64 bits.

Dado que não é garantido que a mensagem tenha um tamanho divisivel por 8 bytes, poderá ser necessário implementar algoritmos para adicionar algum tipo de padding ao último bloco da mensagem. Assim, de seguida, apresentamos as funções de adicionar e remover padding.

[4]:
```
# Função que acrescenta bytes ao último bloco de uma mensagem, para que esta␣
 ↪tenha o tamanho size_block
def pad(last_block, size_block):
```

```
    padding_val = size_block - len(last_block)

    last_block = bytes(last_block,'utf-8')

    #Padder precisa de conhecer o número de bits que este deve acrescentar
    padder = padding.PKCS7(padding_val*8).padder()
    last_block = padder.update(last_block)
    last_block += padder.finalize()

    last_block = last_block.decode("utf-8")

    return last_block, padding_val


# Função que permite remover os bytes de padding do último bloco
def unpad(last_block, size_padding):

    last_block = bytes(last_block,'utf-8')

    # Unpadder precisa de conhecer o número de bits que deverá remover
    unpadder = padding.PKCS7(size_padding*8).unpadder()

    last_block = unpadder.update(last_block)
    last_block += unpadder.finalize()

    last_block = last_block.decode("utf-8")

    return last_block
```

### 1.4 Cifrar a mensagem

De seguida iremos implementar a função de codificação. Para tal, após obter a seed em função da password, será necessário gerar a "chave" utilizando o gerador pseudo-aleatório desenvolvido anteriormente.

A mensagem a cifrar e a "chave" a utilizar deverão ser partidas em blocos e a cada bloco será aplicada a operação XOR. Sabendo que em python um char corresponde a um byte, e que 64 bits equivale a 8 bytes, devemos dividir uma mensagem em blocos de 8 chars. O resultado da operação XOR entre cada bloco da mensagem e output do gerador, será concatenado para obtermos o texto cifrado.

Ao último bloco da mensagem poderá ser aplicado o padding, caso este tamanho tamanho inferior ao fixado.

Por último, após a mensagem ter sido cifrada, será criado um package com metadados aleatórios. De forma, a implementar uma certa autenticação de metadados, será gerado um valor HMAC do package criado, que será verificado à receção da mensagem.

A mensagem enviada conterá o texto cifrado, metadados e o número de bytes de padding adicionados ao último bloco do plaintext. Posteriormente, será gerado o valor HMAC que será também enviado.

[5]:
```
def encrypt(msg, cipher_key, n):
    Nbytes = 8

    # obtem um output a partir de um gerador pseudo-random cuja seed é a␣
↪cipher_key
    output = generator(n, cipher_key)

    # partir a mensagem em blocos de Nbytes
    block_plaintext = [msg[i:i+Nbytes] for i in range(0, len(msg), Nbytes)]

    # se o length da mensagem não for divisivel por 0
    # é necessário acrescentar padding à última mensagem
    mod = len(msg) % Nbytes
    if(mod > 0):
        coef = len(msg) // Nbytes
        padded_block, padded_size = pad(block_plaintext[coef], Nbytes)
        block_plaintext[coef] = padded_block
    else:
        padded_size = 0

    # partir o output do gerador em blocos de 8 bytes
    block_output = [output[i:i+Nbytes] for i in range(0, len(output), Nbytes)]

    # aplicar a operação XOR a cada bloco da mensagem e a cada bloco do output␣
↪do gerador
    cipher = opXOR(block_plaintext, block_output)

    # obter valores random que serão utilizados como metadados/ associated data
    metadata = os.urandom(16)

    # mensagem que será enviada com o texto cifrado, metadata e tamanho do␣
↪padding
    msg = {"ciphertext" : cipher, "metadata": metadata, "padsize": padded_size}

    # obter um valor hmac para a mensagem obtida anteriormente
    # para tal iremos utilizar a cipher_key
    h = hmac.HMAC(cipher_key, hashes.SHA3_256())
    h.update(dumps(msg))
    Hmac = h.finalize()

    pkg = {'msg': dumps(msg) , 'hmac': Hmac}
    return pkg
```

Utilizaremos esta função auxiliar tanto para decifrar como para cifrar, dado que esta permite aplicar a operação XOR a dois conjuntos de blocos:

[6]:
```python
# msg, key are sepated in blocks
def opXOR(msg, key):
    output = ""
    for msg_block, key_block in zip(msg, key):
        for i, j in zip(msg_block, key_block):
            word = ord(i) ^ j # Xor needs to be used between chars
            output += chr(word) # Guardar esta word
    return output
```

### 1.5 Decifrar a mensagem

De forma a completar o modo da cifra é necessário implementar o método de decifrar o texto cifrado recebido. Para que esta descodificação seja possível, é necessário que o participante que pretende executar essa funcionalidade já tenha conhecimento da chave usada para codificar a mensagem. Assim, este deverá receber a **cipher_key** usada para cifrar o plaintext.

Com o package recebido, deverá ser feita uma verificação do valor do HMAC. Para tal, será obtido o valor do HMAC da mensagem e comparado com o valor HMAC recebido no package. Se estes forem diferentes, a operação deverá ser interrompida, senão esta deverá continuar normalmente.

Com a **cipher_key**, deverá ser obtido o resultado de um gerador pseudo-aleatório. Com este resultado, iremos executar uma operação XOR a blocos de 64 bits do ciphertext e a 64 bits do output do gerador.

Adicionalmente, se for indicado que a mensagem tem bytes de padding, estes deverão ser removidos antes que o plaintext seja apresentado.

[7]:
```python
def decrypt(pkg, cipher_key, n):
    Nbytes = 8

    hmac_rcv = pkg['hmac']
    msg = loads(pkg['msg'])

    ct = msg['ciphertext']
    meta = msg['metadata']
    padded_size = msg['padsize']

    # obter o valor hmac para a mensagem (ciphertext, metadata, padsize) recebido
    h = hmac.HMAC(cipher_key, hashes.SHA3_256())
    h.update(dumps(msg))
    HMAC = h.finalize()

    # compara o valor de hmac recebido e hmac determinado nesta função
    # se os valores hmac forem distintos, a função deverá abortar
    if(hmac_rcv == HMAC):
        print(" * HMAC verified. * ")
    else:
        print(" * HMAC invalid. Aborting. * ")
        return
```

```
    # obtem um output de um gerador pseudo-aleatório cuja seed é cipher_key
    output = generator(n, cipher_key)

    # parte o texto cifrado recebido em blocos de Nbytes
    block_ciphertext = [ct[i:i+Nbytes] for i in range(0, len(ct), Nbytes)]

    # separa o output do PRG em blocks de Nbytes
    block_output = [output[i:i+Nbytes] for i in range(0, len(output), Nbytes)]

    # se a mensagem tiver sido padded, o último bloco será argumento da função␣
↪XOR separadamente
    # da restante ciphertext, dado que é necessário remover o padding antes de␣
↪poder adicionar
    # o último bloco de plaintext à string final
    if(padded_size > 0):
        plaintext = opXOR(block_ciphertext[:-1], block_output[:-1])
        coef = len(ct) // Nbytes
        last_cipherblock = block_ciphertext[coef]
        last_outputblock = block_output[coef]
        last_plainblock = opXOR([last_cipherblock], [last_outputblock])
        unpadded_block = unpad(last_plainblock, padded_size)
        plaintext += unpadded_block
    else: # Se não existirem bytes de padding, todo o ciphertext será argumento␣
↪da função XOR.
        plaintext = opXOR(block_ciphertext, block_output)

    return plaintext
```

### 1.6 Função Principal

Por último, definimos uma função que engloba todo o processo definido anteriormente. Nesta função teremos uma chamada da função de cifragem e uma chamada à função de decifragem, permitindo-nos testar se os métodos se encontram corretamente desenvolvidos. Se este for o caso deverá ser possível verificar que a mensagem plaintext é igual à mensagem introduzida.

[8]:
```
def ex2(pwd, n, msg):

    # gerar uma seed em função da password
    cipher_key = generateSeed(bytes(pwd, 'utf-8'))

    # execução do algoritmo de codificação
    pkg = encrypt(msg, cipher_key, n)
    msg_cipher = pkg['msg']
    ciphertext = loads(msg_cipher)['ciphertext']
    print("> CIPHERTEXT: "+ciphertext+"\n")
```

```
    # execução do algoritmo de descodificação
    plaintext = decrypt(pkg, cipher_key, n)
    print("> PLAINTEXT: "+plaintext)
```

### 1.7 Testes da implementação

[9]:
```
ex2("password", 10, "Esta mensagem é apenas um teste para verificar o␣
↪funcionamento do programa.")
```

```
> CIPHERTEXT:
dý¸-^5î{ôtPÿäs(o¦ÕÍ¿·øPýÌä¶a&ÃtSH±¾½1¯«6ÇF¾ãŒyù9km ªIµþ°5ZûäYWfeï

 * HMAC verified. *
> PLAINTEXT: Esta mensagem é apenas um teste para verificar o funcionamento do
programa.
```

[10]:
```
ex2("dg45sSaho8I2", 7, "Vamos testar uma mensagem que não tenham padding")
```

```
> CIPHERTEXT: ÿ|Q$m¡V<{<CÖçQØÝiE
5÷É¯ÜXY¼kO¡çÿmrtÈU¼W

 * HMAC verified. *
> PLAINTEXT: Vamos testar uma mensagem que não tenham padding
```