TP1_EX2

March 7, 2023

1 TP1

1.1 Grupo 17:

PG50315 - David Alexandre Ferreira Duarte

PG51247 - João Rafael Cerqueira Monteiro

1.2 Exercício 2.

Use o package Cryptography para criar uma cifra com autenticação de meta-dados a partir de um PRG - Criar um gerador pseudo-aleatório do tipo XOF ("extened output function") usando o SHAKE256, para gerar uma sequência de palavras de 64 bits. - O gerador deve poder gerar até um limite de

$$2^n$$

palavras (

$$n$$

é um parâmetro) armazenados em long integers do Python. - A "seed" do gerador funciona como

`cipher_key`

e é gerado por um KDF a partir de uma "password" . - A autenticação do criptograma e dos dados associados é feita usando o próprio SHAKE256. - 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".

[8]:
```
# Imports
import os
import time

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

# n -> parametro necessário para gerar as palavaras
N = 17
BLOCKSIZE = 8 # 64 bits = 8 bytes
```

Utilizamos a função KDF com o objetivo de gerar uma chave para uma password, sendo usado um algoritmo pseudo aleatério

[9]:
```python
def derivarChave(password, salt):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=10000
    )
    chave = kdf.derive(password)
    return chave
```

De seguida, através do PRG do tipo XOF utilizando o SHAKE256 geramos palavras de 64 bits

[10]:
```python
def prg(seed):
    digest = hashes.Hash(hashes.SHAKE256(BLOCKSIZE * pow(2,N)))
    digest.update(seed)
    palavras = digest.finalize()
    return palavras
```

Para a alínea b) começamos por criar a função para cifrar, nesta função é necessário controlar se o padding é ou não preciso ser aplicado

[11]:
```python
def cifrar(chave, mensagem):
    textoCifrado = b''
    padder = padding.PKCS7(64).padder()

    # adição de padding ao bloco final da mensagem de forma que o tamanho seja␣
 ↪multiplo
    # dele mesmo
    padded = padder.update(mensagem) + padder.finalize()

    # Divisão da mensagem em blocos de 8 bytes
    mensagemBlocos = [padded[i:i + BLOCKSIZE] for i in range(0, len(padded),␣
 ↪BLOCKSIZE)]

    # Dijunção dos bytes do bloco da mensagem com os bytes do bloco de palavras␣
 ↪chave
    for x in range(len(mensagemBlocos)): # loop para percorrer os blocos
        for indice, byte in enumerate(mensagemBlocos[x]): # loop para percorrer␣
 ↪os bytes do bloco
            textoCifrado += bytes([byte ^ chave[x:(x+1) * BLOCKSIZE][indice]])

    return textoCifrado
```

De seguida no caso de decifrar, tivemos que dividir os respetivos blocos de 64 bits e remover o padding caso fosse implementado na cifra

[12]:
```
def decifrar(chave, textoCifrado):
    plainText = b''
    # Divisão do texto cifrado em blocos de 8 bytes
    mensagemBlocos = [textoCifrado[i:i + BLOCKSIZE] for i in range(0,␣
 ↪len(textoCifrado), BLOCKSIZE)]

    # Dijunção dos bytes do bloco do texto cifrado com os bytes do bloco de␣
 ↪palavras chave
    for x in range(len(mensagemBlocos)): #loop para percorrer os blocos do␣
 ↪texto cifrado
        for indice, byte in enumerate(mensagemBlocos[x]): #loop para percorrer␣
 ↪os bytes do bloco do texto cifrado
            plainText += bytes([byte ^ chave[x:(x+1) * BLOCKSIZE][indice]])

    # Algoritmo para retirar o padding
    unpadder = padding.PKCS7(64).unpadder()
    # Retirar os bytes adicionados
    unpadded = unpadder.update(plainText) + unpadder.finalize()

    return unpadded.decode("utf-8")
```

Por final, idealizamos este main para testar todas as funções implementadas acima

[13]:
```
def main():
    # password a ser utilizada na partilha
    password = "grupo17"

    # salt variavel necessária para derivar a chave
    salt = os.urandom(17)

    # gerador do seed aleatério
    seed = derivarChave(password.encode("utf-8"), salt)

    # gerador da chave utilizado o seed gerado
    chave = prg(seed)

    # Texto cifrado pela cifra
    textoCifrado = cifrar(chave, "Estruturas Criptograficas".encode("utf-8"))

    # Print do texto cifrado
    print(textoCifrado)
    print("")
    # Print do Plain text
    print(decifrar(chave, textoCifrado))
```

[14]:
```
if __name__ == "__main__":
    main()
```

```
b'bKj\x88-
\x8ay\x89Ym\xda\x1b\x8ce\x8b\x02q\x9d*\x9fj\x92\x15\x7f\x89_\xf9\x0b\xfcq\x19\xe
c'

Plain Text: Estruturas Criptograficas
```