TP0-Problema2

March 7, 2022

1 TRABALHO PRÁTICO 0 - GRUPO 14

1.1 Problema 2

O problema 2 consiste em criar uma cifra com autenticação de meta-dados a partir de um PRG do tipo XOF. O problema inicia-se com a geração da chave a ser utilizada como seed para o PRG através da password do utilizador. O gerador PRG vai gerar um conjunto de outputs de 64 bits que serão usados para a cifra de decifra da mensagem. A mensagem será divida em blocos de 64 bits com cada bloco a ser cifrado/decifrado por um dos outputs de 64 bits gerados. A mensagem gerada no processo de cifragem e os respetivos meta-dados são depois autenticados utilizando a seed do PRG. De seguida apresentamos a abordagem usada para a resolução do problema juntamente com o código em Python explicado.

1.1.1 Resolução do Problema

Imports

[1]:
```
import os, sys
from string import printable
from getpass import getpass
from pickle import dumps
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.hashes import SHAKE256
```

Geração da chave utilizando a password A geração da chave é realizada com recurso a uma primitiva **KDF**, nomeadamente, a **PBKDF2HMAC**, que se encontra disponível no módulo **Cryptography**. Como argumento é passada a `password` introduzida pelo utilizador. A chave de 32 bytes será derivada a partir da **password** e funcionará como *seed* do gerador XOF.

[2]:
```
numberOfBytes = 8

#FUNÇÃO: gera a chave apartir da password para ser usada no PRG
def generateKey(password):

    salt = os.urandom(16)
    backend = default_backend()
```

```
    #deriva uma chave de 32 bytes da password que recebeu
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=backend
    )

    # método que deriva a chave apartir da password
    key = kdf.derive(password)

    return key
```

Criação do PRG do tipo XOF A utilização de um gerador pseudoaleatório do tipo XOF, utilizando o algoritmo de hash SHAKE256, permite gerar uma sequência de bytes através de uma *seed* que neste caso foi gerada utilizando a função `generateKey`. Como o gerador tem um limite de 2^n palavras com cada palavra a ter 8 bytes (64 bits) então a sequência de bytes a serem gerados terá $2^n * 8$ bytes, com n a ser indicado pelo o utilizador.

[3]:
```
#FUNÇÃO: gera uma sequencia de (2^n)*8 bytes
def generatorXOF(n, seed):

    #gera uma sequencia de bytes
    digest = hashes.Hash(hashes.SHAKE256((2**n)*8))
    #gera um hash utilizando a seed (chave)
    digest.update(seed)
    words = digest.finalize()

    return words
```

Cifragem Nesta secção foram definidas as funções utilizadas para cifrar o ciphertext. Após a geração da chave com recurso ao método `generateKey`, ciframos *ciphertext* através da função`encryptMessage`. Esta função começa por dividir a mensagem a enviar e a sequência de bytes gerados na função `generatorXOF` em blocos de 64 bits. De seguida, cada bloco do *plaintext* será cifrado utilizando um bloco da sequência de bytes, através da função `encrypt`. A cada iteração do ciclo desta função, vai ser lançada uma operação XOR que vai utilizar caracteres do *plaintext* e do bloco de bytes para gerar o *ciphertext*. Posteriormente os resultados dos ciclos são concatenados e é retornada a mensagem cifrada completa.

[4]:
```
#FUNÇÃO: aloca um bloco de 64 bits do plaintext com um bloco de 64 bits da␣
 ↪sequencia de bytes
def encryptMessage(plaintext, words):

    ciphertext = ""
```

```
    #divide a mensagem em plaintext em blocos de 64 bits
    blockMessage = [plaintext[i:i+numberOfBytes] for i in range(0,␣
↪len(plaintext), numberOfBytes)]
    #divide a sequncia de bytes em blocos de 64 bits
    outputs = [words[i:i+numberOfBytes] for i in range(0, len(words),␣
↪numberOfBytes)]

    for block, output in zip(blockMessage, outputs):
        #algoritmo de cifra utilizando um bloco da mensagem e outro bloco de␣
↪sequencia de bytes
        text = encrypt(block, output)
        ciphertext += (text)

    return ciphertext

#FUNÇÃO: cifra um bloco da mensagem
def encrypt(plainBlock, output):

    ciphertext = ""

    for text_character, out_character in zip(plainBlock, output):
        if text_character not in printable:
            raise ValueError(f"Text value: {text_character} provided is not␣
↪printable ascii")

        #operação de XOR
        xored_value = ord(text_character) ^ out_character
        #utiliza o resultado da operação de XOR e converte para um caracter
        ciphertext_character = chr(xored_value)
        ciphertext += (ciphertext_character)

    return ciphertext
```

Decifragem O processo de decifragem será basicamente o inverso da cifragem. Ou seja, a função `decryptMessage` receberá o *ciphertext* completo, que será divido em blocos de 64 bits juntamente com a sequência de bytes gerados na função `generatorXOF` e será efetuada a decifragem de cada bloco com recurso à função `decrypt`. Esta função é o processo inverso ao realizado na `encrypt`. De seguida, os resultados são concatenados e é retornada a mensagem decrifrada completa.

[5]:
```
#FUNÇÃO: aloca um bloco de 64 bits do ciphertext com um bloco de 64 bits da␣
↪sequencia de bytes
def decryptMessage(ciphertext, words):

    plaintext = ""

    #divide a mensagem em ciphertext em blocos de 64 bits
```

```
    blockMessage = [ciphertext[i:i+numberOfBytes] for i in range(0,␣
↪len(ciphertext), numberOfBytes)]
    #divide a sequncia de bytes em blocos de 64 bits
    outputs = [words[i:i+numberOfBytes] for i in range(0, len(words),␣
↪numberOfBytes)]

    for block, output in zip(blockMessage, outputs):

        #algoritmo de cifra utilizando um bloco da mensagem e outro bloco de␣
↪sequencia de bytes
        text = decrypt(block, output)
        plaintext += (text)

    return plaintext

#FUNÇÃO: decifra um bloco da mensagem
def decrypt(cipherBlock, output):

    plaintext = ""

    for out_character, block_number in zip(output, cipherBlock):

        #operação de XOR
        xored_value = out_character ^ ord(block_number)
        #utiliza o resultado da operação de XOR e converte para um caracter
        plaintext += chr(xored_value)

    return plaintext
```

CLIENTE A execução do programa começa pela geração da chave a partir da `password` inserida pelo utilizador. Tanto a mensagem como o n que será usado como parâmetro para a função `generatorXOF` são recebidos como argumento do programa. De seguida é gerada a sequência de bytes através do `generatorXOF` e cifra-se a mensagem utilizando o `encryptMessage`. A mensagem cifrada é enviada juntamento com os meta-dados que são gerados recorrendo à função `urandom`. Como forma de autenticar-se a mensage completa a ser enviada utiliza-se a função `generateMac` que para o efeito foi utilizado o **HMAC_SHA256** que gera o MAC que identifica unicamente o *ciphertext* em questão. Esta função recebe a chave proveniente da "seed" do gerador e cria um código MAC de autenticação que irá autenticar a mensagem. No processo de decifragem, é feita uma verificação de autenticidade antes de ser efetuado a decifra da mensagem. No final da execução e em caso de inexistência de erro, a mensagem *plaintext* inserida inicialmente deverá ser imprimida no ecrã.

[1]:
```
#FUNÇÃO: geração de código de autenticação
def generateMac(key, crypto):

    h = hmac.HMAC(key, hashes.SHA256(), backend = default_backend())
    h.update(crypto)
```

```
    return h.finalize()


#FUNÇÃO: cliente que cifra e decifra mensagem
def client():

    passwd = getpass('ChooPassword: ').encode('utf-8')

    plaintext = sys.argv[1]
    #parametro a ser enviado para o gerador XOF
    n = int(sys.argv[2])
    key = generateKey(passwd)

    print('Message: ' + plaintext)

    #gera outputs do gerador XOF
    outputs = generatorXOF(n, key)
    print("XOF Generator")
    print(outputs)

    #cifrar e autenticar a mensagem a ser enviada
    print("Encrypting message...")
    ciphertext = encryptMessage(plaintext,outputs)

    #gera metadados
    associatedData = os.urandom(16)
    pkg = {'text': ciphertext, 'ad': associatedData}
    hmac_key = generateMac(key,dumps(pkg))

    print("Cipher Message")
    print(pkg)

    # verificar autenticação
    if hmac_key == generateMac(key,dumps(pkg)):

            #decifra ciphertext
            print("Sending ciphertext ...")
            menssage = decryptMessage(pkg['text'], outputs)
            print('Final Message: ')
            print(menssage)
    else:
            print('ERROR - Different keys used.')


#iniciar cliente
client()
```

1.1.2 Cenários de Teste

[ ]:
```
!python Problema2.py 'Mensagem a ser enviada para o recetor' 5
```

[ ]:
```
!python Problema2.py 'Mensagem' 4
```