Trabalho Prático 0 - Problema 2 - Grupo 18

Introdução ao problema

Neste problema foi-nos proposto criar uma cifra com autenticação de metadados a partir de um gerador pseudo aleatório, com as seguintes restrições:

- O gerador terá de ser do tipo "Extended Output Function", usar o SHAKE256 e gerar sequências de palavras de 64 bits;
- O gerador deve ser capaz de gerar um limite de 2^n palavras, sendo n um parâmetro, armazenando-as em *long integers*;
- A "seed" do gerador funciona como uma chave de cifragem e é gerada por um KDF, a partir de uma "password";
- A autenticação do criptograma e dos dados associados é feita usando o SHAKE256;
- É necessário definir os algoritmos de cifrar e de decifrar, sendo que para cifrar ou decifrar uma mensagem com blocos de 64 bits os "outputs" do gerador são usados como máscaras XOR dos blocos da mensagem. Desta forma, a cifra básica torna-se uma implementação do algoritmo "One Time Pad".

Resolução do problema

Imports

```
import os, sys, string, getpass, pickle

from pickle import dumps
from string import printable

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
```

Gerador pseudo aleatório do tipo XOF

A função seguinte gera 2^n sequências de 8 bytes, usando a seed obtida através da password.

```
# A cryptographic hash function takes an arbitrary block of data and
calculates a fixed-size bit string (a digest), such that different
data results (with a high probability) in different digests.

# Função responsável pelo gerador XOF que cria sequências de palavras
de 64 bits
def geradorXOF(n, generatedKey):
```

```
    # A cryptographic hash function takes an arbitrary block of data
and calculates a fixed-size bit string (a digest),
    # such that different data results (with a high probability) in
different digests.
    # Dentro de hashes.SHAKE256((2 ** n) * 8) estamos a definir o
tamanho do digest (maxímos de bits dado o número máximo de palavras)
    digest = hashes.Hash(hashes.SHAKE256(nrPalavras*8))
    digest.update(generatedKey)
    words = digest.finalize()

    return words
```

Gerador de seed

O gerador seguinte é responsável por receber uma password e gerar uma seed, através de uma KDF, neste caso a PBKDF2HMAC.

```
# Função responsável por gerar a seed que será utilizada na RPG
(Pseudo Random Generator)
def seedGeneration(palavraChave):

    # Salts devem ser gerads de forma aleatória (melhor forma
possível, apesar de poder ser escolhido arbitrariamente, mas não é a
melhor forma)
    salt = os.urandom(16)

    # derive
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=390000,
        backend=default_backend()
    )

    # Derivação da seed
    seed = kdf.derive(palavraChave)

    return seed
```

Gerador de valor de autenticação

Este gerador recebe uma chave de autenticação e um pacote e retribui o seu valor de autenticação.

```
# Função responsável pela geração do valor de autenticação
def generateMac(hmac_key, package):

    h = hmac.HMAC(hmac_key, hashes.SHA256(), backend =
default_backend())
```

```
    h.update(package)

    return h.finalize()
```

Cifragem de um bloco

Esta função é responsável por cifrar um único bloco de 64 bits.

O algoritmo começa por agrupar os argumentos da função em tuplos de caracteres. Depois, itera sobre estes e executa a operação XOR para cada tuplo, guardando o seu valor numa lista de caracteres cifrados. No final da execução, retorna esta lista, senda ela correspondente ao bloco cifrado.

```
# Função responsável por cifrar cada bloco de 64 bits
def encryptBlock(plainTextBlock, generatedEncryptedWord):

    # String variable that will contain all the shifted values
    ciphertext = ""

    for text_character, generatedEncryptedWord_character in
zip(plainTextBlock, generatedEncryptedWord):
        if text_character not in printable:
            raise ValueError(f"Text value: {text_character} provided
is not printable ascii")

        # Completed the XOR of the characters ordinance (integer
representation)
        xored_value = ord(text_character) ^
generatedEncryptedWord_character

        # Takes resulting integer from XOR operation and converts it
to a character
        ciphertext_character = chr(xored_value)

        # Add the generated character to the ciphertext
        ciphertext += (ciphertext_character)

    # Return do bloco cifrado
    return ciphertext
```

Cifragem de uma mensagem

Esta função é responsável por cifrar uma mensagem inteira.

O algoritmo divide os argumentos da função em blocos de 64 bits e itera sobre eles, usando a função encryptBlock para cada um. Em cada iteração, vai reunindo o valor retornado numa lista de caracteres e retorna-a no fim, sendo esta a mensagem cifrada.

```
# Função responsável por cifrar uma mensagem completa
def encryptMessage(plainTextMessage, generatedEncryptedWords):
```

```
    # String variable that will contain all the shifted values
    ciphertext = ""

    # Criar array que dvide a mensagem em blocos de 2^nrBytesPalavra
Bits
    messageBlocks = []
    for i in range(0, len(plainTextMessage), nrBytesPalavra):
        messageBlocks.append(plainTextMessage[i : i+nrBytesPalavra])

    # Criar array que dvide palavras geradas por XOF em blocos de
2^nrBytesPalavra Bits
    wordBlocks = []
    for i in range(0, len(generatedEncryptedWords), nrBytesPalavra):
        wordBlocks.append(generatedEncryptedWords[i :
i+nrBytesPalavra])

    # Cifrar cada um dos blocos de 2^nrBytesPalavra Bits e acrescentar
ao cipherText final
    for plainTextBlock, outputBlock in zip(messageBlocks, wordBlocks):
        cipherBlock = encryptBlock(plainTextBlock, outputBlock)
        ciphertext += cipherBlock

    # Returno do texto cifrdo
    return ciphertext
```

Decifragem de um bloco

Esta função é responsável por decifrar um único bloco de 64 bits.

O algoritmo começa por agrupar os argumentos da função em tuplos de caracteres. Depois, itera sobre estes e executa a operação XOR para cada tuplo, guardando o seu valor numa lista de caracteres decifrados. No final da execução, retorna esta lista, senda ela correspondente ao bloco decifrado.

```
# Função responsável por decifrar cada bloco de 64 bits
def decryptBlock(ciphertext, generatedEncryptedWord):

    # String variable that will contain all the plain text values
    plaintext = ""

    for generatedEncryptedWord_character, ciphertext_number in
zip(generatedEncryptedWord, ciphertext):
        xored_value = generatedEncryptedWord_character ^
ord(ciphertext_number)
        plaintext  += chr(xored_value)

    return plaintext
```

Decifragem de uma mensagem

Esta função é responsável por decifrar uma mensagem inteira.

O algoritmo divide os argumentos da função em blocos de 64 bits e itera sobre eles, usando a função decryptBlock para cada um. Em cada iteração, vai reunindo o valor retornado numa lista de caracteres e retorna-a no fim, sendo esta a mensagem decifrada.

```
# Função responsável por decifrar uma mensagem cifrada completa
def decryptMessage(ciphertextMessage, generatedEncryptedWords):

    # String variable that will contain all the plain text values
    plainText = ""

    # Criar array que dvide a mensagem em blocos de 2^nrBytesPalavra
Bits
    messageBlocks = []
    for i in range(0, len(ciphertextMessage), nrBytesPalavra):
        messageBlocks.append(ciphertextMessage[i : i+nrBytesPalavra])

    # Criar array que dvide palavras geradas por XOF em blocos de
2^nrBytesPalavra Bits
    wordBlocks = []
    for i in range(0, len(generatedEncryptedWords), nrBytesPalavra):
        wordBlocks.append(generatedEncryptedWords[i :
i+nrBytesPalavra])

    # Cifrar cada um dos blocos de 2^nrBytesPalavra Bits e acrescentar
ao cipherText final
    for cipherTextBlock, outputBlock in zip(messageBlocks,
wordBlocks):
        plainBlock = decryptBlock(cipherTextBlock, outputBlock)
        plainText += plainBlock

    return plainText
```

Execução do programa

Aqui podemos encontrar a execução de programa que é capaz de criar uma cifra e cifrar um pacote contendo uma mensagem e metadados.

Este começa por definir algumas constantes para a sua execução. Depois, gera a seed através da password e usa o gerador pseudo aleatório XOF para gerar palavras de 64 bits. Usando estas palavras, cifra o texto inicial, convertendo-o em texto cifrado. Por fim, junta o texto cifrado e os metadados e calcula o seu valor de autenticação, juntado-os num pacote, pronto para ser enviado.

De forma a verificar a correta execução da programa, é testado se o valor de autenticação calculado anteriormente coincide com o valor esperado do pacote.

```python
# Número arbitrário n (máximo 2^n palavras)
n = 4

# Número máimo de palavras
nrPalavras = 2**n

# Numero de bytes (palavras de 64 bits)
nrBytesPalavra = 8

# Numero de bits de cada palavras
nrBitsPalavra = 2**nrBytesPalavra




# Palavra chave escolhida pelo Utilizador
passwordUtilizador = "password"

# Plaintext para cifrar
textoInicial = "Eu quero cifrar esta frase com um one time pad"

print('Original Text: ' + textoInicial)



# Gerar seed através de uma KDF (temos de passar palavra-chave em
bytes)
seed = seedGeneration(passwordUtilizador.encode('utf-8'))

# Gerar através de XOF sequências de palavras de 64 bits (utilizando
como fatores o n arbitrário e a key conseguida)
generatedWords = geradorXOF(n, seed)




# Cifrar o texto incial
cipherText = encryptMessage(textoInicial, generatedWords)




# Gerar dos metadados como valores pseudo-aleatórios
associatedData = os.urandom(16)

# Criar dicionário com o textoCifrado e os metadados
message = {'text': cipherText, 'ad': associatedData}
```

```
# Gerar código de autenticação para a mensagem final
hmac_key = generateMac(seed,dumps(message))

# Criar o pacote contendo a mensagem e o valor de autenticação
package = {'message': message, 'hmac': hmac_key}



# Verificar se o Utilizador é autenticado corretamente e, se sim,
decifrar o texto cifrado
if hmac_key == generateMac(seed,dumps(package['message'])):

        decrypt_text = decryptMessage(package['message']['text'],
generatedWords)
        print('Decrypt Text: ' + decrypt_text)
else:
        print('ERROR - Different keys used.')
```

```
Original Text: Eu quero cifrar esta frase com um one time pad
Decrypt Text: Eu quero cifrar esta frase com um one time pad
```