TP0_EC_Grupo9

March 15, 2021

1 Trabalho Prático 0 de Estruturas Criptográficas

Autores: (Grupo 9)

- Nelson Faria (A84727)
- Miguel Oliveira (A83819)

2 Exercício 1

[1]:
```
# Imports necessários à execução do código presente neste notebook

import os
import sys
import timeit

from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
```

2.1 Derivar uma chave com uma KDF(Key Derivation Function) a partir de uma password

Para derivar uma chave com a password foi necessário usar uma KDF(no nosso caso usamos a HKDF da biblioteca python *cryptography.io*) para derivar uma chave com 256 bits(32 bytes).

[2]:
```
'''
Funcao usada para derivar uma chave
'''


def derivationKey(password,salt):
    info = None
    hkdf = HKDF(
        hashes.SHA256(),
        32,
        salt,
```

```
        info,
    )
    return hkdf.derive(password)
```

2.2 Verificar que o MAC da chave gerada pelo *Emitter* ou *Receiver* está correto

Para verificar o MAC da chave gerada pelo Receiver foi necessário usar o **HMAC** e a função *verify()* para verificar que o MAC que o receiver gerou é igual ao MAC gerado pelo emitter!

[3]:
```
'''
Funcao que serve para verificar a chave que foi recebida pelo receiver
'''


def verifyKey(key2, key):
    h = hmac.HMAC(key, hashes.SHA256())
    h.update(key)
    h.verify(key2)
```

2.3 Funcao que serve para autenticar as chaves aquando da troca inicial entre as partes

O **HMAC** é usado somente para a autenticidade da troca de chaves entre o *Emitter* e o *Receiver*! Assim, a informação da chave é cifrada com a própria chave e, deste modo, o *Emitter* ou *Receiver* só podem acordar a chave a usar se e só se as chaves que estes tiverem são iguais. De lembrar que para que isso aconteça ambos têm de possuír a mesma *password* e o mesmo *salt*!!!

[4]:
```
'''
Funcao que serve para autenticar a chave
'''


def authData(key2):
    h = hmac.HMAC(key2, hashes.SHA256())
    h.update(key2)
    return h.finalize()
```

2.4 Funcao que serve para cifrar uma mensagem usando a cifra AESGCM

A partir de uma chave, que foi gerada neste caso a partir da função *derivationKey*, da mensagem a cifrar e ainda de um nonce que deve ser gerado gerado de forma pseudo-aleatória através por exemplo do uso da função *os.urandom()*, esta função retorna o criptograma resultante. De notar que se está a usar também uma variável exemplo(**ASSOCIATED_DATA**) para colocar alguns dados que não serão cifrados mas serão autenticados.

[5]:
```
'''
Funcao usada para cifrar e autenticar a mensagem
'''


def cifraGCM(key, mensagem, nonce):
    aesgcm = AESGCM(key)

    ct = aesgcm.encrypt(nonce, mensagem, ASSOCIATED_DATA)

    return ct
```

2.5 Funcao que serve para decifrar uma mensagem usando a cifra AESGCM

A partir de uma chave, que foi gerada neste caso a partir da função *derivationKey*, do criptograma a decifrar e ainda de um nonce que neste caso foi recebido pela outra parte comunicante(*emitter* ou *receiver*) para que seja possível obter a mensagem original, esta função retorna o texto original. De notar que aqui também se está a usar uma variável exemplo (**ASSOCIATED_DATA**) que deve ser a mesma da que foi usada para cifrar.

[6]:
```
'''
Funcao usada para decifrar e autenticar a mensagem
'''


def decifraGCM(key, criptograma, nonce):
    aesgcm = AESGCM(key)

    msg = aesgcm.decrypt(nonce, criptograma, ASSOCIATED_DATA)

    return msg
```

2.6 Validação das chaves entre o Emitter e o Receiver

Função usada para validar as chaves usadas durante a comunicação entre as partes. Deste modo, ambas as partes devem entrar em acordo relativamente à chave a usar, por isso é fundamental que a password introduzida tanto pelo *emitter* como pelo *receiver* seja igual. Por outro lado, como é o *emitter* que pretende enviar mensagens para o *receiver*, este deve enviar inicialmente o *salt* usado para gerar a chave a partir da password (neste caso, usou-se a *Key Derivation Function* **HKDF**). Além disso, deve haver uma autenticação das chaves(neste caso usou-se o **HMAC**) para que se tenha a certeza que ambas as partes estão em acordo com a chave e que ambos estejam a usar a mesma chave. Como o *emitter* gera a chave inicialmente, este pode fazer o MAC da chave com a chave que gerou e seguidamente enviar esse MAC para o *receiver* e ainda o salt que usou para gerar a chave, de forma a que o *receiver* consiga gerar a mesma chave e verificar a sua integridade.

Assim, a negociação da chave a usar, que aqui está simulada, é feita da seguinte forma:

```
**Emitter** <-------------------------------> **Receiver**
                    *salt + MAC_key_(key)*
                ------------------------------>
                        *MAC_key_(key)*
                <------------------------------
```

[7]:
```
'''
Funcao que serve para validar a chave entre o Emitter e o Receiver
'''


def validateKey(pwE,pwR):

    '''EMITTER'''
    # Geracao do salt para a derivacao da chave
    saltE = os.urandom(16)
    # Derivar a chave do Emitter
    keyE = derivationKey(pwE.encode("utf-8"), saltE)
    print("Chave gerada pelo Emitter: ")
    print(keyE)
    # Autenticar a chave com a propria chave
    keyEA = authData(keyE)
    # Enviar o salt e o MAC(key) ao receiver
    BUFFER = saltE + keyEA
    '''RECEIVER'''
    # Esperar pelo salt e o mac da chave do emitter
    saltR = BUFFER[0:16]
    keyEA_r = BUFFER[16:len(BUFFER)]
    # Gerar a chave a partir da password do Receiver
    keyR = derivationKey(pwR.encode("utf-8"), saltR)
    print("Chave gerada pelo Receiver: ")
    print(keyR)
    # Verificar se o Mac enviado pelo emitter corresponde ao mac da chave gerada
    try:
        verifyKey(keyEA_r, keyR)
    except InvalidSignature as e:
        print("The key sent by the emitter does not match: %s" % e)
        sys.exit(0)
    # Autenticar a chave com a propria chave
    keyRA = authData(keyR)
    # Enviar o mac da chave que foi gerada
    BUFFER = keyRA
    '''EMITTER'''
    # Comparar se a resposta é mesmo igual ao mac da chave do emitter
    try:
        verifyKey(BUFFER, keyE)
    except InvalidSignature as e:
```

```
        print("The key sent by the receiver does not match: %s" % e)
        sys.exit(0)

    return keyE
```

2.7 Comunicação entre o Emitter e o Receiver

Função principal para haver a simulação da comunicação entre as duas entidades. Deste modo, primeiro é pedido que se introduza a password do *emitter* e depois a do *receiver*, que devem ser as mesmas. Depois, a função *validateKey* entra em ação para gerar as chaves para ambas as partes usarem. Se tudo correr bem, será possível que o *Emitter* envie uma mensagem ao *Receiver* e seguidamente que este receba a respetiva resposta. Como se pode ver pelos vários *prints* efetuados, poderá se ver quais os textos cifrados que foram feitos e ainda as mensagens trocadas.

[8]:
```
ASSOCIATED_DATA = b"Exemplo de Associated Data para o TP0 de EC"

BUFFER = b""


# Funcao que serve para dar inicio à comunicação entre o Emitter<->Receiver
def communicate():

    # Introducao da password por parte do emitter
    pwE = input("[Emitter] Introduza a password: ")

    # Introducao da password por parte do emitter
    pwR = input("[Receiver] Introduza a password: ")

    if len(pwE) > 0 and len(pwR) > 0:
        # Validar as chaves que foram geradas tanto pelo Emitter como o Receiver
        key = validateKey(pwE,pwR)

        '''EMITTER'''
        # Emitter escreve a mensagem para o receiver
        pt = input("Emitter message: ")
        if len(pt) > 0:
            # Nonce usado para cifrar a mensagem
            nonceE = os.urandom(12)
            # Enviar a mensagem ao receiver
            BUFFER = nonceE + cifraGCM(key, pt.encode("utf-8"), nonceE)
            print("Criptograma do Emitter: ")
            print(BUFFER)
            '''RECEIVER'''
            # Receber a mensagem do emitter
            nonceR = BUFFER[0:12]
            crypto = BUFFER[12:len(BUFFER)]
```

```
            # Decifrar a mensagem
            msg = decifraGCM(key, crypto, nonceR).decode("utf-8")
            print("Mensagem enviada pelo Emitter: " + msg)
            # Receiver escreve a mensagem para o emitter
            pt = input("Receiver message: ")
            if len(pt) > 0:
                # Nonce usado para cifrar a mensagem
                nonceR = os.urandom(12)
                # Enviar a resposta ao Emitter
                BUFFER = nonceR + cifraGCM(key, pt.encode("utf-8"), nonceR)
                print("Criptograma do Receiver: ")
                print(BUFFER)
                '''EMITTER'''
                # Receber a mensagem do receiver
                nonceE = BUFFER[0:12]
                crypto = BUFFER[12:len(BUFFER)]
                # Decifrar a mensagem
                msg = decifraGCM(key, crypto, nonceE).decode("utf-8")
                print("Mensagem enviada pelo Receiver: " + msg)
            else:
                print("Insira uma mensagem válida")
        else:
                print("Insira uma mensagem válida")
```

Chamada da função *communicate* por onde se dá inicio ao processo de simulação da comunicação entre um **Emitter** e um **Receiver**. (Para isso, deve-se correr as funções todas que estão acima)

[9]:
```
communicate()
```

```
[Emitter] Introduza a password: informatica
[Receiver] Introduza a password: informatica
Chave gerada pelo Emitter:
b'\xaa\xca\xa1b\xac=\xfdx\xfa\x80\xa5\x1e\x9eN\x8b\x11\xd3\x95&p^ve\xafTz\xcf\x0
3P\xa0\xc1\x14'
Chave gerada pelo Receiver:
b'\xaa\xca\xa1b\xac=\xfdx\xfa\x80\xa5\x1e\x9eN\x8b\x11\xd3\x95&p^ve\xafTz\xcf\x0
3P\xa0\xc1\x14'
Emitter message: Olá, tudo bem contigo?
Criptograma do Emitter:
b'(=\x19\x9a\xde\xa3\x95\xd7u\x92\x85\xa0\xc1`d1X\x89/\xe4\xb2]\xb76I\xdbi\x19\x
bf\xc9%\xe9\xc1d\xf1\x88/\x93\xe0\xeaSR\xa3_\xf0\xa6c\xd90q\xb5'
Mensagem enviada pelo Emitter: Olá, tudo bem contigo?
Receiver message: Sim, obrigado pela tua atenção :)
Criptograma do Receiver:
b'\xc2g\x9dn\'\xd5~\x93\xbcK\xbd\xe4\xd7\xf4\x8d.\x9b\xb7\x94\xd2\xb5\x11N-\x85_
^^\xe3\xdeK\xe9\xb0\xe0\xe9&8\xcf4\x0ezL\xfb"\xfaC^D\xa4\x97\xd4=\xe0\tm\x15\x07
a\xee\xc6\xa3N\x8e'
```

```
Mensagem enviada pelo Receiver: Sim, obrigado pela tua atenção :)
```

3 Exercício 2

3.1 Alínea a

Nesta primeira alínea o objetivo era implementar um **gerador pseudo-aleatório do tipo XOF("*extened output function*")** usando o **SHAKE256**, com a finalidade de gerar uma sequência de palavras de 64 bits.

Com isto em mente, era necessário conhecer um **parâmetro N**, que influencia o número de palavras de 64 bits a serem geradas. De um modo mais concreto, o gerador é responsável por gerar 2^N palavras. Outro aspeto bastante relevante é a necessidade de ser conhecida uma **password**, a partir da qual será gerada uma **chave**, através do uso do **HKDF**, que será usada para a geração da sequência de palavras.

A função responsável por derivar uma chave de 256 bits a partir de uma password já consta deste mesmo documento, a qual é designada de **derivationKey()**.

Definimos uma macro que guarda então o tamanho de cada palavra gerada pelo PRNG

[10]:
```
SIZE_BLOCK = 8
```

A função apresentada de seguida **(generateRandomWords())** é aquela que, mediante um **parêmtro N** e a chave gerada a partir da função mencionada anteriormente, é capaz de gerar 2^N palavras. Para o armazenamento destas palavras, o nosso grupo optou por usar uma **lista de string** e não **long integers** como era pedido, tendo em conta que tal tipo de dados já não consta do **python3**.

[11]:
```
def gerador(seed, param_n):
    # A sequencia de palavras tem de ter tamanho suficiente para as 2^n palavras
    digest = hashes.Hash(hashes.SHAKE256(SIZE_BLOCK * (2 ** param_n)))
    digest.update(seed)
    return digest.finalize()

def generateRandomWords(key, param_n):
    # Sequencia aleatoria gerada pelo gerador
    s = gerador(key, param_n)
    # Criar as palavras como long integers
    blocos = []
    for i in range(2 ** param_n):
        blocos.append(s[i * SIZE_BLOCK:i * SIZE_BLOCK + SIZE_BLOCK])
    return blocos
```

3.2 Alinea b

Obtendo essa lista de palavras através do **gerador pseudo-aleatório**, conseguiremos aplicar diretamente a cifra **One Time Pad (OTP)**. Para tal, em primeiro lugar foi necessário construir uma função em que, dadas duas sequencias de bytes, realizar o

xor entre essas duas sequências e retornar o respetivo resultado dessa mesma operação. Essa função é apresentada de seguida.

[12]:
```
def xor_str(str1, str2):
    return bytes([_a ^ _b for _a, _b in zip(str1, str2)])
```

Exemplificamos o seu uso com um caso prático bastante simples:

[13]:
```
xor_str(b'miguel',b'nelson')
```

[13]:
```
b'\x03\x0c\x0b\x06\n\x02'
```

[14]:
```
xor_str(b'nelson',b'\x03\x0c\x0b\x06\n\x02')
```

[14]:
```
b'miguel'
```

Tendo isto, agora, era necessário partir a mensagem em **blocos de 64 bits**, de modo a realizar o xor com as respetivas **palavras geradas** pelo gerador pseudo-aleatório, quer aquando da **cifragem**, quer aquando da **decifragem** de mensagens. Mostramos então de seguida ambas as funções com a capacidade de cifrar ou decifrar, mediante a apresentação da **lista de palavras** geradas.

[15]:
```
def cifrar(message, words):
        i = 0
        ciphertext = b''
        for word in words:
            ciphertext += xor_str(message[i * SIZE_BLOCK:(i + 1) * SIZE_BLOCK],␣
 ↪word)
            i += 1
        return ciphertext
```

[16]:
```
def decifrar(ciphertext,words):
        i = 0
        plaintext = b''
        for word in words:
            plaintext += xor_str(ciphertext[i * SIZE_BLOCK:(i + 1) *␣
 ↪SIZE_BLOCK], word)
            i += 1
        return plaintext
```

Apresentamos agora de seguida um exemplo prático da execução destes dois oráculos:

[17]:
```
key = derivationKey(b'password',os.urandom(12))
words = generateRandomWords(key,2) # São geradas 2^2=4 palavras de 64-bits
msg = b"Uma mensagem ultra secreta"

ciphertext = cifrar(msg,words)
decipheredtext = decifrar(ciphertext,words)
```

```
print(b'PlainText: ' + msg)
print(b'CipherText: ' + ciphertext)
print(b'DecipheredText: ' + decipheredtext)
```

```
b'PlainText: Uma mensagem ultra secreta'
b'CipherText: H"\x9bK\xa8\xee\xe4g\xc6\xf5
\x02F\xd8ACK\x837\xee\x87\xaeA\xd78\x0f'
b'DecipheredText: Uma mensagem ultra secreta'
```

3.3 Alinea c

No que à comparação de eficiência entre a cifra "criada" por nós e à usada no primeiro exercício deste trabalho prático, elaboramos o seguinte cenário de teste:

> Para cada uma das cifrar, é testado o tempo que demora a **cifrar** e **decifrar** uma qualquer mensagem aleatória N vezes. De seguida apresentamos as respetivas funções que apresentam esse mesmo procedimento para cada uma das cifras.

[18]:
```
N = 10  # Vamos ter sequencias de 1024 bytes


def homeMadeCipher():
    pwd = b"password"
    key = derivationKey(pwd,os.urandom(12))
    plaintext = os.urandom(2 ** N)
    words = generateRandomWords(key,N) # São geradas 2^2=4 palavras de 64-bits
    ciphertext = cifrar(plaintext,words)
    #print(ciphertext)
    #print(plaintext == decifrar(ciphertext,words))

def aesgcmCipher():
    pwd = b"password"
    key = derivationKey(pwd,os.urandom(12))
    nonce = os.urandom(12)
    plaintext = os.urandom(2 ** N)
    ciphertext = cifraGCM(key, plaintext, nonce)
    #print(ciphertext)
    #print(plaintext == decifraGCM(key, ciphertext, nonce))
```

> Por fim, e de modo a analisar a eficiencia de cada uma das cifrar recorremos à biblioteca **timeit** do python. De seguida mostramos a função que nos permitiu obter os tempos de execução de cada uma das rotinas anteriores num número de **repetições** que é passado como parâmetro do método

[19]:
```
HM = '''
homeMadeCipher()
'''

AESGCMM = '''
```

```
aesgcmCipher()
'''

setup = '''
from __main__ import homeMadeCipher
from __main__ import aesgcmCipher
'''

def timeTester(repetitions):
    print('> Iniciado processo de cifragem usando a nossa cifra...')
    timeHM = timeit.timeit(stmt=HM, number=repetitions, setup=setup)
    print("Done.")

    print('> Iniciado processo de cifragem usando a cifra AESGCM...')
    timeAESGCM = timeit.timeit(stmt=AESGCMM, number=repetitions, setup=setup)
    print("Done.")

    print("\n[TIMES]")
    print("Home made: " + str(timeHM))
    print("AESGCM: " + str(timeAESGCM))


timeTester(1000)
```

```
> Iniciado processo de cifragem usando a nossa cifra…
Done.
> Iniciado processo de cifragem usando a cifra AESGCM…
Done.

[TIMES]
Home made: 1.2095120999999978
AESGCM: 0.06375680000000017
```

De facto, para mensagens com 2^{10} blocos de 64-bits, a execução das rotinas de cifragem e decifragem das mesmas 1000 vezes, mostra-nos claramente que a nossa cifra é mais ineficiente. De facto, e usando **One Time Pad** a chave gerada tem que ter o tamanho da mensagem, o que não acontece com o **AESGCM**, pelo que tal resultado já seria de esperar.