

TP1 - Exercício 2

February 27, 2024

Nota: Devido ao módulo **asyncio**, o código disponibilizado deve ser executado através de ficheiros diferentes (para o *emitter* e para o *receiver*) e não no presente *notebook*.

1 Estruturas Criptográficas

1.1 Trabalho Prático 1 - Exercício 2

1.1.1 José de Matos Moreira - PG53963

1.1.2 Pedro Freitas - PG52700

1.2 Enunciado do problema

Use o *package* **Cryptography** para: * implementar uma **AEAD** com “**Tweakable Block Ciphers**” conforme está descrito na última secção do texto “Capítulo 1: Primitivas Criptográficas Básicas”. A cifra por blocos primitiva, usada para gerar a “**Tweakable Block Cipher**”, é o **AES-256** ou o **ChaCha20** * use esta cifra para construir um canal privado de informação assíncrona com acordo de chaves feito com “**X448 key exchange**” e “**Ed448 Signing&Verification**” para autenticação dos agentes. Deve incluir uma fase de confirmação da chave acordada

1.3 Resolução

Em primeiro lugar, procedeu-se ao *import* dos módulos necessários, em ambos os ficheiros *python* utilizados:

```
[ ]: import asyncio
import os
import hmac
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import x448, ed448
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import serialization
from cryptography.exceptions import InvalidSignature
from pickle import dumps, loads
```

1.3.1 Emitter e Receiver

Inicialmente, apresenta-se as funções usadas para geração das chaves de cifra: privada, pública e partilhada. Como pedido no enunciado, recorreu-se ao **X448 key exchange**. Deste modo e,

recorrendo a ambas as funções, cada um dos agentes consegue: * criar as chaves privada e pública * criar uma chave partilhada, recorrendo à sua chave privada e à chave pública partilhada pelo outro agente

Deste modo, consegue-se a geração de uma chave partilhada que pode ser usada por ambos os agentes para cifragem/decifragem de todas as mensagens transportadas entre os mesmos.

```
[ ]: def generate_keys():
    private_key = x448.X448PrivateKey.generate()
    public_key = private_key.public_key()

    return private_key, public_key

def generate_shared(private_key, peer_public_key):
    shared_key = private_key.exchange(peer_public_key)
    derived_key = HKDF(algorithm=hashes.SHA256(), length=32, salt=None,
    ↪info=b'handshake data').derive(shared_key)

    return derived_key
```

De seguida e, mais uma vez, como requerido pelo enunciado do problema, elaborou-se as funções com a capacidade de gerar as chaves privada e pública responsáveis pela assinatura de uma mensagem e, obviamente, a função responsável por aplicar essa mesma assinatura à respetiva mensagem passada como argumento. Tudo isto foi feito com base no **Ed448 signing**. Deste modo, cada um dos agentes consegue criar as suas chaves de assinatura, privada e pública e, através da sua chave privada gerada, assinar qualquer mensagem a ser enviada para que, através da chave pública, consiga ser verificada pelo agente com o qual está a estabelecer contacto.

```
[ ]: def generate_sign_keys():
    private_key = ed448.Ed448PrivateKey.generate()
    public_key = private_key.public_key()

    return private_key, public_key

def sign_this(private_key, message):
    sign = private_key.sign(message)

    return sign
```

1.3.2 Emitter

Agora, passa-se a enumerar e explicar as diversas funções desenvolvidas a serem usadas (principalmente) pelo *emitter*. Em primeiro lugar, começou-se por escolher o algoritmo de criptografia a utilizar, o **AES-256**. O mesmo caracteriza-se por ser altamente seguro, uma vez que utiliza chaves de 256 bits de comprimento. Porém, este mesmo algoritmo atua em blocos de 16 bytes de comprimento cada, o que leva à implementação de funcionalidades que preencham um bloco (**padding**)

de forma a que o mesmo passe a possuir 16 bytes de extensão, para que possa ser alvo do algoritmo referido.

```
[ ]: def padding(blocks):
    last_len = len(blocks[-1])
    blocks[-1] += b"\x00" * (16 - last_len)

    return blocks, last_len
```

Por outro lado, surgiu também a necessidade de desenvolver código responsável por aplicar o algoritmo **AES-256** a cada um dos blocos alvo. Como estudado, o mecanismo geral de aplicação da cifra (**AEAD** com “**Tweakable Block Ciphers**”) funciona da seguinte forma: * os primeiros $(m - 1)$ blocos **P** são cifrados com a **TPBC**, controlada por uma só chave k mas com “*tweaks*” w distintos * o último bloco **P** é cifrado de forma distinta: como um **XOR** de uma máscara gerada cifrando t (tamanho do último bloco antes do *padding*) * o último passo é a geração de um “*tag*” de autenticação a partir da paridade do “*plaintext*”

Os “*tweaks*” w (usados na cifragem dos blocos) e w^* (usado na autenticação) são construídos como se ilustra em seguida: * existe um “*nounce*” (“name only used once”) que ocupa os primeiros $b/2$ bits de cada “*tweak*” * os m “*tweaks*” usados na cifra distinguem-se do restante por terminarem num bit 0. O “*tweak*” da autenticação termina em 1 * a componente intermédia é um contador, incrementado em uma unidade em cada bloco, quando usado na cifragem dos blocos, ou então é um parâmetro l , igual ao comprimento total do “*plaintext*” (sem “*pad*”), quando usado na autenticação

Assim, apresenta-se a função responsável por cifrar um bloco, recorrendo ao **AES-256**, em modo **CBC** (*Cipher Block Chaining*):

```
[ ]: def tpbc (tweak, key, block, iv):
    derived_tweak_key = hmac.digest(key, tweak, 'sha256')
    cipher = Cipher(algorithms.AES(derived_tweak_key), modes.CBC(iv))
    encryptor = cipher.encryptor()
    cipher_block = encryptor.update(block) + encryptor.finalize()

    return cipher_block
```

Tendo desenvolvido esta função, nasceu a necessidade de desenvolver uma outra com o poder de aplicar a cifra aos $(m - 1)$ primeiros blocos, tal como manda o mecanismo. É importante reiterar que a mesma também calcula, simultaneamente, a autenticação:

```
[ ]: def apply_tpbc(blocks, nonce, key, iv):
    cipher_blocks = b""
    init_counter = os.urandom(7)
    counter = init_counter
    auth = 16 * b"\x00"

    for block in blocks:
        tweak = nonce + counter + b"\x00"
        cipher_block = tpbc(tweak, key, block, iv)
        cipher_blocks += cipher_block
```

```

counter_length = len(counter)
int_counter = int.from_bytes(counter, 'big') + 1
counter = int_counter.to_bytes(counter_length, 'big')

auth = bytes(a ^ b for (a,b) in zip(auth, block))

return cipher_blocks, auth, counter, init_counter

```

Por outro lado e, seguindo, mais uma vez, o mecanismo anteriormente explicado, escreveu-se a função responsável por executar o mesmo ao último bloco do nosso *plaintext*:

```

[ ]: def solve_last(tweak, key, last_len, iv, last_block):
    bytes_last_len = last_len.to_bytes(16, 'big')
    mask = tpbc(tweak, key, bytes_last_len, iv)
    last_cipher = bytes(a ^ b for (a,b) in zip(last_block, mask))

    return last_cipher

```

Deste modo e, tendo as principais funções desenvolvidas, passou-se à codificação de uma função principal, **encrypt**, com a grande responsabilidade de fazer o trabalho todo de cifragem do conteúdo passado. Assim, a mesma possui a capacidade de fazer o seguinte: * produzir o *nounce* * produzir o *iv* * dividir a mensagem/*plaintext* em blocos * aplicar o *padding* ao último bloco (quando necessário) * aplicar o mecanismo de cifragem aos (m - 1) primeiros blocos * cifrar o último bloco, juntando-o aos blocos anteriormente trabalhados * continuar o processo de autenticação * gerar a *tag*

```

[ ]: def encrypt(message, key):
    nounce = os.urandom(8)
    iv = os.urandom(16)

    blocks = [bytes(message[counter : counter + 16], 'utf-8') for counter in
↳range(0, len(message), 16)]
    padded_blocks, last_len = padding(blocks)

    cipher_blocks, auth, counter, init_counter = apply_tpbc(padded_blocks[:-1],
↳nounce, key, iv)
    last_cipher = solve_last(nounce + counter + b"\x00", key, last_len, iv,
↳padded_blocks[-1])
    cipher_blocks += last_cipher

    auth = bytes(a ^ b for (a,b) in zip(auth, padded_blocks[-1]))

    m_length = len(message).to_bytes(16, 'big')
    tweak = nounce + m_length + b"\x01"
    tag = tpbc(tweak, key, auth, iv)

```

```

    return {'cipher_blocks': cipher_blocks, 'tag': tag, 'nonce': nonce,
    ↪ 'init_counter': init_counter, 'last_len': last_len, 'iv': iv}

```

Por fim, concluiu-se o desenvolvimento do código principal relativo ao *emitter*, produzindo a sua função **main**. A mesma faz o seguinte: * conexão assíncrona ao *localhost*, na porta 8888 * criação das chaves de cifra, privada e pública * assinatura de uma mensagem para confirmação da chave * comunicação, com o outro agente, informando-o das suas chaves públicas e da sua mensagem assinada * leitura das chaves públicas do outro agente e da sua mensagem assinada * verificação da assinatura do outro agente * criação da chave partilhada (para ser usada no processo de cifragem) * pedido de *input* da mensagem a ser cifrada e, posteriormente, enviada * cifragem da mensagem * assinatura do conteúdo * envio dos dados obtidos

```

[ ]: async def main():
    reader, writer = await asyncio.open_connection('localhost', 8888)

    private_cipher_key, public_cipher_key = generate_keys()
    private_sign_key, public_sign_key = generate_sign_keys()

    public_cipher_key_bytes = public_cipher_key.
    ↪ public_bytes(encoding=serialization.Encoding.Raw, format=serialization.
    ↪ PublicFormat.Raw)
    sign_message = sign_this(private_sign_key, public_cipher_key_bytes)

    writer.write(public_cipher_key_bytes)
    writer.write(public_sign_key.public_bytes(encoding=serialization.Encoding.
    ↪ Raw, format=serialization.PublicFormat.Raw))
    writer.write(sign_message)
    await writer.drain()
    print('[emitter] Public keys sent...')

    peer_cipher_key = await reader.read(56)
    peer_sign_key = await reader.read(57)
    peer_sign = await reader.read(114)
    print('[emitter] Peer keys received...')

    try:
        peer_sign_new_key = ed448.Ed448PublicKey.
        ↪ from_public_bytes(peer_sign_key)

        peer_sign_new_key.verify(peer_sign, peer_cipher_key)
        print("[emitter] Signature validated...")

        shared_key = generate_shared(private_cipher_key, x448.X448PublicKey.
        ↪ from_public_bytes(peer_cipher_key))
        print("[emitter] Shared key created...")

        plaintext = input("[emitter] Type the message: ")

```

```

cipher_result = encrypt(plaintext, shared_key)
print("[emitter] Message encrypted...")

cipher_result_tosend = dumps(cipher_result)
sig = sign_this(private_sign_key, cipher_result_tosend)

final = {'sign': sig, 'content': cipher_result_tosend}
writer.write(dumps(final))
await writer.drain()
print("[emitter] Message sent...")

except InvalidSignature:
    print("[emitter] Couldn't verify the signature!")

```

Por fim, a **main** do *emitter* é chamada pelo **run**, do **asyncio**.

```
[ ]: asyncio.run(main())
```

1.3.3 Receiver

Ao contrário do que foi apresentado até ao momento, descreve-se, de seguida, as funções utilizadas pelo *receiver* em todo o processo de decifragem. Em primeiro lugar, surgiu a necessidade de escrever uma função capaz de decifrar, recorrendo ao **AES-256**, o conteúdo presente num bloco de cifra.

```
[ ]: def undo_tpbcb (tweak, key, block, iv):
    derived_tweak_key = hmac.digest(key, tweak, 'sha256')
    cipher = Cipher(algorithms.AES(derived_tweak_key), modes.CBC(iv))
    decryptor = cipher.decryptor()
    plain_block = decryptor.update(block) + decryptor.finalize()

    return plain_block

```

Tal como no *emitter*, seguindo o mecanismo geral de cifra, projetou-se a função responsável por decifrar os $(m - 1)$ primeiros blocos, sendo a mesma apresentada da seguinte forma:

```
[ ]: def apply_undo_tpbcb(blocks, init_counter, nonce, key, iv):
    plaintext = b""
    counter = init_counter
    auth = 16 * b"\x00"

    for block in blocks:
        tweak = nonce + counter + b"\x00"
        plain_block = undo_tpbcb(tweak, key, block, iv)
        plaintext += plain_block

    counter_length = len(counter)

```

```

int_counter = int.from_bytes(counter, 'big') + 1
counter = int_counter.to_bytes(counter_length, 'big')

auth = bytes(a ^ b for (a,b) in zip(auth, plain_block))

return plaintext, auth, counter

```

Do mesmo modo, nasceu, também, a função capaz de reverter o que foi aplicado ao último bloco.

```

[ ]: def unsolve_last(tweak, key, last_len, iv, last_block):
    bytes_last_len = last_len.to_bytes(16, 'big')
    mask = tpbc(tweak, key, bytes_last_len, iv)
    last_cipher = bytes(a ^ b for (a,b) in zip(last_block, mask))

    return last_cipher

```

Assim e, analogamente ao que foi feito anteriormente, codificou-se a **decrypt**, função principal capaz de juntar todo o trabalho útil daquelas que foram apresentadas até ao momento, relativamente, claro, ao corpo do *receiver*. Deste modo, elaborou-se a função capaz do seguinte: * dividir o conteúdo cifrado em blocos * decifrar os primeiros (m - 1) blocos * aplicar o processo de decifragem ao último bloco * dar *unpad* a esse mesmo bloco, juntando-o com os outros blocos de *plaintext* * continuar com o processo de cálculo da autenticação * calcular a *tag* * verificar se a *tag* é igual àquela enviada pelo outro agente

```

[ ]: def decrypt(content, key):
    cipher_blocks = [content['cipher_blocks'][counter : counter + 16] for
    ↪ counter in range(0, len(content['cipher_blocks']), 16)]
    plaintext, auth, counter = apply_undo_tpbc(cipher_blocks[:-1],
    ↪ content['init_counter'], content['nonce'], key, content['iv'])
    last_plain = unsolve_last(content['nonce'] + counter + b"\x00", key,
    ↪ content['last_len'], content['iv'], cipher_blocks[-1])

    plaintext += last_plain[:content['last_len']]

    auth = bytes(a ^ b for (a,b) in zip(auth, last_plain))

    m_length = len(plaintext).to_bytes(16, 'big')
    tweak = content['nonce'] + m_length + b"\x01"
    tag = tpbc(tweak, key, auth, content['iv'])

    tag_status = True
    if tag != content['tag']:
        tag_status = False

    return plaintext, tag_status

```

Assim, surgiu a *handle_connection*, com a capacidade de reunir todo este trabalho útil explicado nas funções anteriores. Caracteriza-se a mesma da seguinte forma: * leitura das chaves públicas do

outro agente e da sua mensagem assinada * criação das chaves de cifra, privada e pública * assinatura de uma mensagem para confirmação da chave * comunicação, com o outro agente, informando-o das suas chaves públicas e da sua mensagem assinada * verificação da assinatura do outro agente * criação da chave partilhada (para ser usada no processo de decifragem) * leitura do conteúdo enviado pelo outro agente * verificação da assinatura presente nesse mesmo conteúdo * decifragem dos dados e obtenção do *plaintext* * *print* da mensagem, em caso de sucesso na verificação da *tag*

```
[ ]: async def handle_connection(reader, writer):
    peer_cipher_key = await reader.read(56)
    peer_sign_key = await reader.read(57)
    peer_sign = await reader.read(114)
    print('[receiver] Peer keys received...')

    private_cipher_key, public_cipher_key = generate_keys()
    private_sign_key, public_sign_key = generate_sign_keys()

    public_cipher_key_bytes = public_cipher_key.
    ↪public_bytes(encoding=serialization.Encoding.Raw, format=serialization.
    ↪PublicFormat.Raw)
    sign_message = sign_this(private_sign_key, public_cipher_key_bytes)

    writer.write(public_cipher_key_bytes)
    writer.write(public_sign_key.public_bytes(encoding=serialization.Encoding.
    ↪Raw, format=serialization.PublicFormat.Raw))
    writer.write(sign_message)
    await writer.drain()
    print('[receiver] Public keys sent...')

    try:
        peer_sign_new_key = ed448.Ed448PublicKey.
        ↪from_public_bytes(peer_sign_key)

        peer_sign_new_key.verify(peer_sign, peer_cipher_key)
        print("[receiver] Signature validated...")

        shared_key = generate_shared(private_cipher_key, x448.X448PublicKey.
        ↪from_public_bytes(peer_cipher_key))
        print("[receiver] Shared key created...")

        read_content = await reader.read()
        cipher = loads(read_content)
        print("[receiver] Cipher received...")

        try:
            peer_sign_new_key.verify(cipher['sign'], cipher['content'])
            print("[receiver] Signature validated...")
```



```

        content = loads(cipher['content'])
        plaintext, tag_status = decrypt(content, shared_key)

        if tag_status == False:
            print("[receiver] Invalid tag...")

        else:
            print("[receiver] Message received: " + plaintext.decode())

    except InvalidSignature:
        print("[receiver] Couldn't verify the signature!")

except InvalidSignature:
    print("[receiver] Couldn't verify the signature!")

```

Não podendo faltar, apresenta-se a função **main**, que cria um servidor assíncrono, no *localhost*, na porta 8888, colocando-se à escuta no mesmo, fazendo o devido tratamento das conexões, recorrendo à função anterior.

```

[ ]: async def main():
    server = await asyncio.start_server(handle_connection, 'localhost', 8888)

    async with server:
        await server.serve_forever()

```

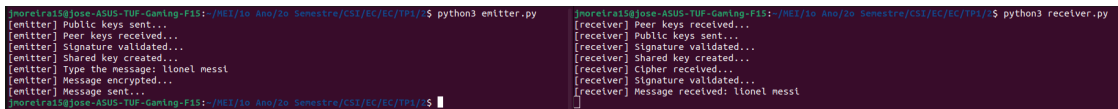
Tal como acontece com o código desenvolvido para o outro agente, esta mesma função principal é chamada pelo **run**, do **asyncio**.

```

[ ]: asyncio.run(main())

```

1.3.4 Testes de aplicação



```

jmoreira@jose-ASUS-TUF-Gaming-F15: ~/E1/1o_Ano/2o_Semestre/CSI/EC/TP1/2$ python3 emitter.py
[emitter] Public keys sent...
[emitter] Peer keys received...
[emitter] Signature validated...
[emitter] Shared key created...
[emitter] Type the message: Lionel messi
[emitter] Message encrypted...
[emitter] Message sent...
jmoreira@jose-ASUS-TUF-Gaming-F15: ~/E1/1o_Ano/2o_Semestre/CSI/EC/TP1/2$

jmoreira@jose-ASUS-TUF-Gaming-F15: ~/E1/1o_Ano/2o_Semestre/CSI/EC/TP1/2$ python3 receiver.py
[receiver] Peer keys received...
[receiver] Public keys sent...
[receiver] Signature validated...
[receiver] Shared key created...
[receiver] Cipher received...
[receiver] Signature validated...
[receiver] Message received: Lionel messi

```

Execução correta de ambos os agentes