Estruturas criptográficas: TP1 problema 2

Descrição do problema

No problema 2, utilizando o "package" **Cryptography**, foi-nos pedida uma implementação de:

1. Uma AEAD ("**authenticated encryption with associated data**") com **"Tweakable Block Ciphers"**. A cifra por blocos primitiva, usada para gerar a "tweakable block cipher", é o AES-256 ou o ChaCha20.

2. Um canal privado de informação assíncrona com acordo de chaves feito com **"X448 key exchange"** e **"Ed448 Signing&Verification"** para autenticação dos agentes. Com uma fase de confirmação da chave acordada.

Implementação

Relativamete à comunicação assíncrona, utillizamos a mesma abordagem do problema 1, com a excessao de que aqui a troca de chaves e posterior confirmação é feita na função emitter e sempre que queremos enviar uma mensagem, chamamos a funcao send_message.

Em primerio lugar, começamos por implementar a **"Tweakable Block Ciphers"**, e para isso, optamos pela cifra por blocos primitiva **ChaCha20**. De modo a cifrar a mensagem em blocos, dividimos a mesma em blocos de 64 bytes (variável global "tam_bloco"), e no momento de cifrar um bloco com ChaCha20, utilizamos um tweak para modificar a chave (Operação XOR), que seria incrementado a cada bloco cifrado (função incrementar_tweak). De notar que o primeiro tweak (initial vector) é uma sequência de 16 bytes pseudo aleatória. Para cifrar o último bloco verificamos se a mensagem a ser cifrada tem o mesmo tamanho do bloco, caso a condição não se verificasse acrescentavamos uma sequência de zeros, de forma a preencher o resto do bloco (padding).

Depois de obtermos o criptograma, juntamos-lhe os dados associados, que incluem o initial vector, numero de blocos do criptograma, o tamanho do padding e o nonce, esta informação será fulcral para que o recetor consiga decifrar a mensagem com sucesso. No final, assinamos a mensagem, acrescentamos a assinatura à mensagem e enviamos ao recetor.

Relativamente à troca de chaves e assinatura, o recetor gera uma chave privada e uma chave pública X448, e envia a sua chave pública ao emissor. O emissor faz o mesmo processo, mas além de enviar a chave publica X448, envia tambem uma chave publica Ed448, de modo a que o recetor possa verificar a assinatura da mensagem feita com a chave privada ED448 do emissor. Após a troca de chaves ambos obtêm uma chave comum através da chave pública X448 recebida que posteriormente é derivada para obter uma chave partilhada com 32 bytes, para ser utilizada na cifra ChaCha20.

Antes da troca de mensagem existe uma etapa de confirmação de chaves, em que o emissor envia uma assinatura da chave compartilhada, por sua vez o recetor verifica essa assinatura usando a sua chave compartilhada, comprovando que as chaves compartilhadas são efetivamente iguais.

O recetor quando recebe a mensagem do emissor, lê os primeiros 114 bytes da mesma (tamanho da assinatura Ed448) e verifica a autenticidade e integridade dos dados, ou seja, se a verificação for bem sucedida o recetor poderá garantir que a mensagem foi enviada pelo emissor e que a mesma não foi alterada. De seguida é feita a verificação do nonce, onde o recetor guarda cada nonce recebido e verifica se o novo nonce ja foi repetido, para detetar se não houve quaisquer ataques por repetição e consequentemente avançará com o processo de decifragem.

```
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric.x448 import
X448PrivateKey, X448PublicKey
from cryptography.hazmat.primitives.asymmetric.ed448 import
Ed448PrivateKey, Ed448PublicKey
from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
import math
import asyncio

tam_bloco = 64

# Recetor
async def handle_emitter_ex2(reader, writer):

    nonces = []

    #gerar chaves X448
    private_key = X448PrivateKey.generate()
    public_key = private_key.public_key()


    # Envia a publicKey
    writer.write(public_key.public_bytes_raw())
    await writer.drain()

    #Lê a public key X448 do emissor
    public_key_emitter = await reader.readexactly(56)
    public_key_emitter =
X448PublicKey.from_public_bytes(public_key_emitter)

    #Lê a public key ED448
    verification_key = await reader.readexactly(57)
    verification_key =
Ed448PublicKey.from_public_bytes(verification_key)

    #obtem a chave compartilhada
    shared_key = private_key.exchange(public_key_emitter)
```

```
    #deriva a chave para 32 bytes (ChaCha20)
    key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=None,
    ).derive(shared_key)


    # Confirmação de chave acordada
    message_signature = await reader.read(114)

    # Confirma a assinatura
    try:
        verification_key.verify(message_signature, key)
    except InvalidSignature:
        print("Signature Key Confirmation failed.")
        return

    while True:
        # Lê a assinatura
        signature = await reader.read(114)


        # Lê o nonce
        nonce = await reader.read(16)

        #Lê os dados associados
        ad = await reader.readuntil(b'_ADEnd_') # lê até à flag
        iv = ad[:32] # o inicial vector são os primeiros 16 bytes

        #retira a flag
        associated_data = ad[32:].decode()[:-7]

        #separa os dados associados
        num_blocos = int(associated_data.split('+')[1])
        padding = int(associated_data.split('+')[2])
        size = num_blocos * tam_bloco

        #Lê o criptograma
        ciphertext = await reader.readexactly(size)

        #verifica a assinatura dos dados
        try:
            verification_key.verify(signature, nonce + ad +
ciphertext)
        except InvalidSignature:
            print("Invalid Singnature.")
            return
```

```
        # Verifica validade do nonce
        if nonce not in nonces:
            nonces.append(nonce)
        else:
            print("Message replayed.")
            return

        # inicia a string com o texto decifrado
        decrypted_text = b""

        #inicia o tweak
        tweak = iv

        #decifra a mensagem bloco a bloco
        for i in range(num_blocos):
            # Xor entre chave e tweak
            tweaked_key = bytes([a ^ b for a, b in zip(key, tweak)])

            #remove o padding do último bloco
            if i != num_blocos - 1:
                cipher = Cipher(algorithms.ChaCha20(tweaked_key,
nonce), mode=None, backend=default_backend())
                decryptor = cipher.decryptor()
                decrypted_text +=
decryptor.update(ciphertext[:tam_bloco]) + decryptor.finalize()

                ciphertext = ciphertext[tam_bloco:]

            # Se for o último bloco, retira o padding
            else:
                cipher = Cipher(algorithms.ChaCha20(tweaked_key,
nonce), mode=None, backend=default_backend())
                decryptor = cipher.decryptor()
                decrypted_text +=
decryptor.update(ciphertext[:tam_bloco - padding]) +
decryptor.finalize()

            #incrementa o tweak
            tweak = incrementar_tweak(tweak)


        print(decrypted_text.decode())


# Emissor
async def send_message_ex2(reader, writer, key, ed_private_key,
attack):

    message = input("Enter message: ").encode()
```

```
    # Divide a mensagem em blocos
    blocos, pad = divide_em_blocos_com_padding(message, tam_bloco)

    # Gera um InitialVector aleatório
    iv = os.urandom(32)

    # Gera um nonce aleatório
    nonce = os.urandom(16)

    if attack != None and attack != "ciphertext":
        nonce = attack

    ciphertext = b""

    tweak = iv
    for bloco in blocos:
        # Xor entre chave e tweak
        tweaked_key = bytes([a ^ b for a, b in zip(key, tweak)])
        # Inicialização do objeto de cifra
        cipher = Cipher(algorithms.ChaCha20(tweaked_key, nonce),
mode=None, backend=default_backend())
        # Cifragem
        encryptor = cipher.encryptor()
        ciphertext += encryptor.update(bloco) + encryptor.finalize()
        # Incremento do tweak
        tweak = incrementar_tweak(tweak)

    # Cria dados associados (iv + num_blocos + padding)
    associated_data = iv + b'+' + str(len(blocos)).encode() + b'+' +
str(pad).encode()

    # Junta o nonce e os dados associados com o criptograma
    ciphertext = nonce + associated_data + b'_ADEnd_' + ciphertext

    # Assina a mensagem
    siganture = ed_private_key.sign(ciphertext)

    ciphertext = siganture + ciphertext

    if attack == "ciphertext":
        ciphertext = ciphertext[:-3] + b'\x00\x00\x00'

    # Envia a mensagem para o receiver
    writer.write(ciphertext)


    return nonce
```

```python
def divide_em_blocos_com_padding(mensagem, tamanho_bloco):
    bytes_adicionais = 0
    numero_blocos = math.ceil(len(mensagem) / tamanho_bloco)
    blocos = []
    for i in range(numero_blocos):
        inicio = i * tamanho_bloco
        fim = (i + 1) * tamanho_bloco
        bloco = mensagem[inicio: fim]
        if len(bloco) < tamanho_bloco:  # Adicionar padding com zeros
            bytes_adicionais = tamanho_bloco - len(bloco)
            bloco += b'\x00' * bytes_adicionais
        blocos.append(bloco)
    return blocos, bytes_adicionais

def incrementar_tweak(tweak):
    # Bytes para inteiro
    valor = int.from_bytes(tweak, byteorder='big')

    # Incrementar o valor do tweak
    novo_valor = valor + 1

    # Inteiro para bytes
    novo_tweak = novo_valor.to_bytes(32, byteorder='big')

    return novo_tweak

#receiver
async def receiver_ex2():
    server = await asyncio.start_server(lambda reader, writer:
handle_emitter_ex2(reader, writer), '127.0.0.1', 8888)

    print("Receiver ready...\n")

    async with server:
        await server.serve_forever()

#emitter
async def emitter_ex2():

    reader, writer = await asyncio.open_connection('127.0.0.1', 8888)

    # Gera as chaves Ed448
    ed_private_key = Ed448PrivateKey.generate()
    verification_key = ed_private_key.public_key()


    # Lê a chave pública X448
    public_key_receiver = await reader.readexactly(56)
    public_key_receiver =
```

```
X448PublicKey.from_public_bytes(public_key_receiver)

    # Gera as chaves X448
    private_key = X448PrivateKey.generate()
    public_key = private_key.public_key()

    # Envia a chave pública X448 e a chave pública Ed448
    writer.write(public_key.public_bytes_raw() +
verification_key.public_bytes_raw())
    await writer.drain()

    # Obtem a chave partilhada
    shared_key = private_key.exchange(public_key_receiver)

    # Derivacao da chave
    key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=None,
    ).derive(shared_key)

    # Confirmação da chave acordada
    # Envia uma assinatura da chave
    siganture = ed_private_key.sign(key)
    writer.write(siganture)
    await writer.drain()

    return reader, writer, key, ed_private_key
```

Testes

Iniciar o receiver

```
asyncio.create_task(receiver_ex2())
```

```
<Task pending name='Task-5' coro=<receiver_ex2() running at C:\Users\
Utilizador\AppData\Local\Temp\ipykernel_16408\1698952022.py:207>>
```

```
Receiver ready...
```

Iniciar o Emitter (establecer conexao)

Mensagem: In recent years, the use of deep learning in image super-resolution has become the mainstream. The results achieved by deep learning models have surpassed traditional methods, offering enhanced image resolution across various applications.

```
#Create emitter
#In recent years, the use of deep learning in image super-resolution
has become the mainstream. The results achieved by deep learning
models have surpassed traditional methods, offering enhanced image
resolution across various applications.
reader, writer, key, ed_private_key = await emitter_ex2()
```

Caso certo

```
n = await send_message_ex2(reader, writer, key, ed_private_key, attack
= None)

In recent years, the use of deep learning in image super-resolution
has become the mainstream. The results achieved by deep learning
models have surpassed traditional methods, offering enhanced image
resolution across various applications.
```

Ataque por repetição (nounce repetido)

passamos o nounce da mensagem anterior no campo 'attack'

```
n = await send_message_ex2(reader, writer, key, ed_private_key, attack
= n)

Message replayed.
```

Ataque ao criptograma

Modifica bytes do criptograma

```
n = await send_message_ex2(reader, writer, key, ed_private_key, attack
= "ciphertext")

Invalid Singnature.
```