Estruturas Criptográficas 2022/23

TP1. Problema 3

Grupo 7. Leonardo Berteotti e Paulo R. Pereira

É pretendida a construção de um canal de comunicação privada e assíncrona com acordo de chaves feito com “X448 key exchange” e “Ed448 Signing&Verification” para a autenticação dos agentes. A comunicação com a troca das chaves públicas entre os agentes (Alice e Bob). Cada um irá gerar a respetiva chave partilhada a ser utilizada nas funções para cifrar e decifrar. Note que deverão ser serão usadas assinaturas digitais (EDSA) de modo a garantir a autenticidade e integridade das chaves públicas partilhadas. Após a obtenção da chave partilhada por ambos os agentes, a Alice irá enviar mensagens ao Bob que serão cifradas utilizando uma AEAD com “Tweakable Block Ciphers”. A cifra por blocos primitiva, usada para gerar a “tweakable block cipher”, é o AES-256.

In [141]:

```
import os
from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric.x448 import X448PrivateKey
from cryptography.hazmat.primitives.asymmetric.ed448 import Ed448PrivateKey
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

from pickle import dumps, loads
import asyncio
import nest_asyncio
nest_asyncio.apply()
```

Numa primeira fase, cada agente deverá gerar os pares de chaves assimétricas de modo a poderem acordar num par de chaves partilhadas. Para tal, utilizou-se o protocolo (seguro) **X448 key exchange**.

Como também já referido, é necessário que cada agente crie o respetivo par de chaves assimétricas. As respetivas chaves públicas serão então trocadas pelos dois agentes, por um canal controlado pelo atacante. No entanto, o atacante apenas conhecerá as chaves públicas, e por isso não consegue gerar as chaves partilhadas, pois não tem a informação das chaves privadas.

A função *generate_keys* trata de gerar a chave privada e pública. Deve ser então invocada por cada agente.

In [142]:

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

    return private_key, public_key
```

A função *generate_shared_key* trata de gerar a chave partilhada *shared_key*.

É de notar que çara a maioria das aplicações, a chave partilhada deve ser passada a uma função de derivação de chaves (**KDF**). Isso permite a mistura de informações adicionais na chave, a derivação de várias chaves e a destruição de qualquer estrutura que possa estar presente.

In [143]:

```
def generate_shared_key(private_key, peer_public_key):

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

    return shared_key
```

Relativamente à autenticação dos agentes (não apenas para garantir a autenticidade, mas também a integridade e o não-repúdio na troca de chaves), foi utilizado o algoritmo de assinatura **Ed448 Signing&Verification**. A chave privada é usada para assinar a mensagem e a chave pública para verificar a validade da assinatura. Assim, um agente consegue verificar se a mensagem que recebeu é fidedigna.

A função *sign_message* é usada para assinar uma mensagem, retornando a assinatura, a própria mensagem e a chave pública.

In [144]:

```
def sign_message(message):
    private_key = Ed448PrivateKey.generate()
    signature = private_key.sign(message)

    packet = {'signature' : signature,
              'message'   : message,
              'ed448_pk'  : private_key.public_key().public_bytes(
                                                      encoding=serialization.Encoding.PEM,
                                                      format=serialization.PublicFormat.SubjectPublicKeyInfo
                                                  )
             }
    return packet
```

Implementação da TPBC

A cifra AEAD com "Tweakable Block Ciphers" utiliza um input adicional designado de *tweak*. Estes funcionam como chaves únicas de cada bloco, enquanto que a chave propriamente dita é a mesma em todos os blocos, tornando a cifra menos vulnerável a ataques.

A função *generate_tweaks* tem como objetivo gerar os *tweaks* com tamanho igual ao dos blocos da mensagem (32 bytes). Ela retorna os *tweaks* a serem usados na cifra dos blocos e um outro *tweak* para a autenticação do *ciphertext*.

Os *tweaks* usados na cifra têm um *nonce* que ocupa metade do tamanho e um contador, incrementado uma unidade em cada bloco, ocupando a restante metade. Este *tweak* termina com um bit a 0.

$w_i = [nonce|i|0]$, com $i = 0...m-1$, m = número de blocos.

O *tweak* de autenticação têm também um *nonce* que ocupa metade do tamanho e, na outra metade, o comprimento da mensagem sem o *padding*. Este *tweak* termina com um bit a 1.

$w^* = [nonce|length(plaintext)|1]$

In [145]:

```
def generate_tweaks(number_of_blocks, plaintext_length, nonce):
    cipher_tweaks = []
    # cipher tweaks [nonce|counter|0]
    for i in range(0, number_of_blocks):
        tweak = nonce + int(i).to_bytes(32 // 2, byteorder='big')
        tweak = int.from_bytes(tweak, byteorder='big')

        # remove last bit and add the final bit 0
        tweak = tweak >> 1
        tweak = tweak << 1

        tweak = tweak.to_bytes(32, byteorder='big')
        cipher_tweaks.append(tweak)

    # authentication tweak [nonce|plaintext_length|0]
    auth_tweak = nonce + plaintext_length.to_bytes(32 // 2, byteorder='big')
    auth_tweak = int.from_bytes(auth_tweak, byteorder='big')

    # last bit of auth_tweak to 1
    mask = 0b1
    auth_tweak = auth_tweak | mask
    auth_tweak = auth_tweak.to_bytes(32, byteorder='big')

    return cipher_tweaks, auth_tweak
```

Foi seguida a abordagem de o *tweak* modificar o "plaintext" x .

$\tilde{E}(w,k,x) \equiv E(k, w \oplus E(k,x))$

O algoritmo usado para cifrar em $E(k,x)$ é o **ChaCha20**. A função $\tilde{E}(w,k,x)$ corresponde à função *get_ciphertext*.

Assumindo uma mensagem dividida em M blocos, os primeiros $M-1$ blocos são cifrados do seguinte modo: usa-se a função *get_ciphertext* para cifrar o respetivo bloco junto com o respetivo *tweak* e a chave partilhada; o último bloco é cifrado usando o próprio tamanho (sem o *padding*), sendo efetuado o XOR ao resultado junto com o bloco.

Por fim, é gerado o *tag* de autenticação a partir da operação **XOR** entre todos os blocos do *plaintext* (com o *padding* no último bloco). O *output* gerado pela cifra utilizando a chave juntamente com o *tweak* de autenticação é usado para extrair a *tag* de autenticação da cifra.

In [146]:

```
def get_ciphertext(cipher_key, nonce, tweak, plaintext, ad):

    chacha = ChaCha20Poly1305(cipher_key)
    ciphertext = chacha.encrypt(nonce, plaintext, ad)

    xored = [(a^b).to_bytes(1,byteorder='big') for (a,b) in zip(tweak, ciphertext)]

    return b"".join(xored)
```

In [147]:

```
def encrypt(plaintext, cipher_key, ad):
    # divide plaintext into blocks
    blocks = []
    for i in range(0, len(plaintext), 32):
        block = plaintext[i:i+32].encode('utf-8')
        # padding
        r = len(block)
        if r < 32:
            blocks.append(block.ljust(32, b'\0'))
        else:
            blocks.append(block)

    length = len(plaintext)
    number_of_blocks = len(blocks)

    # generate tweaks
    nonce_tweak = os.urandom(32 // 2)
    cipher_tweaks, auth_tweak = generate_tweaks(number_of_blocks, length, nonce_tweak)

    encrypted_blocks = []

    nonce = os.urandom(12)

    # encrypt first m-1 blocks
    for w in range(0, number_of_blocks - 1):
        ciphertext = get_ciphertext(cipher_key, nonce, cipher_tweaks[w], blocks[w], ad)
        encrypted_blocks.append(ciphertext)

    # encrypt last block
    r_in_bytes = int(r).to_bytes(32, byteorder='big')
    ct = get_ciphertext(cipher_key, nonce, cipher_tweaks[number_of_blocks-1], r_in_bytes, ad)

    xored = [(a^b).to_bytes(1,byteorder='big') for (a,b) in zip(ct, blocks[number_of_blocks-1])]
    last_ciphertext = b"".join(xored)

    encrypted_blocks.append(last_ciphertext)

    # authentication phase
    auth = blocks[0]
    for i in range(1, number_of_blocks):
        xored = [(a^b).to_bytes(1,byteorder='big') for (a,b) in zip(auth, blocks[i])]
        auth = b"".join(xored)

    tag = get_ciphertext(cipher_key, nonce, auth_tweak, auth, ad)[:r]

    # join all encrypted blocks
    ciphertext = b"".join(encrypted_blocks)

    return ciphertext, tag, nonce, nonce_tweak
```

In [148]:

```
def decrypt(ciphertext, tag, nonce, nonce_tweak, cipher_key, ad):
    # divide plaintext into blocks
    blocks = [ciphertext[i:i+32] for i in range(0, len(ciphertext), 32)]

    number_of_blocks = len(blocks)
    n = len(ciphertext)
    r = len(tag)
    length = n - (32 - r)

    # generate tweaks
    cipher_tweaks, auth_tweak = generate_tweaks(number_of_blocks, length, nonce_tweak)

    decrypted_blocks = []

    # decrypt blocks
    for w in range(0, number_of_blocks):
        plaintext = get_ciphertext(cipher_key, nonce, cipher_tweaks[w], blocks[w], ad)
        decrypted_blocks.append(plaintext)

    # authentication phase
    auth = decrypted_blocks[0]
    for i in range(1, number_of_blocks):
        xored = [(a^b).to_bytes(1,byteorder='big') for (a,b) in zip(auth, decrypted_blocks[i])]
        auth = b"".join(xored)

    generated_tag = get_ciphertext(cipher_key, nonce, auth_tweak, auth, ad)[:r]

    # verify authentication
    if tag == generated_tag:
        decrypted_blocks[number_of_blocks - 1] = decrypted_blocks[number_of_blocks - 1][:r]
        plaintext = b"".join(decrypted_blocks)

    else :
        return "ERROR! Different tag used in authentication."

    return plaintext.decode('utf-8')
```

Comunicação

A comunicação começa com o *Emitter* enviando as suas chaves públicas devidamente assinadas pelo mesmo ao *Receiver*. Este, por sua vez, verifica a autenticidade das chaves públicas, e, após a verificação, envia as suas chaves públicas ao *Emitter* também assinadas por si. Do mesmo modo, o *Emitter* verifica a autenticidade das chaves públicas, e estando a verificação feita, a comunicação começa. O *Emitter* cria uma mensagem, cifra a mensagem e envia o *ciphtertext*, os *nonces*, os dados associados e a *tag* de autenticação. Com estes dados, o *Receiver* é então capaz de decifrar a mensagem.

In [149]:

```
async def emitter(emitter_queue, receiver_queue):
    private_key, public_key = generate_keys()

    public_key = public_key.public_bytes(
                                encoding=serialization.Encoding.PEM,
                                format=serialization.PublicFormat.SubjectPublicKeyInfo
                            )

    print("Emitter is sending his public key (signed by him)...")
    await emitter_queue.put(dumps(sign_message(public_key)))

    receiver_pk_bytes = await receiver_queue.get()
    receiver_pk = loads(receiver_pk_bytes)
    print("\nEmitter is receiving Receiver's public key...")

    print("\nEmitter is verifying Receiver's message signature...")
    receiver_ed448_pk = serialization.load_pem_public_key(receiver_pk['ed448_pk'])

    try:
        receiver_ed448_pk.verify(receiver_pk['signature'], receiver_pk['message'])
    except:
        return 'ERROR --- Different Ed448 key used'

    print("\nReceiver's message is authentic!")

    receiver_public_key = serialization.load_pem_public_key(receiver_pk['message'])

    # generate shared key
    shared_key = generate_shared_key(private_key, receiver_public_key)

    msg_to_send = "Finally we can communicate! :)"
    print('\nORIGINAL MESSAGE (FROM EMITTER):\n' + msg_to_send)

    # cipher message
    print("\nEncrypting Emitter's message...")
    ad = os.urandom(16)
    ciphertext, tag, nonce, nonce_tweak = encrypt(msg_to_send, shared_key, ad)

    print("\nENCRYPTED MESSAGE: ")
    print(ciphertext.decode('unicode_escape'))

    # send packet
    packet = {
        'ciphertext' : ciphertext,
        'tag' : tag,
        'nonce' : nonce,
        'nonce_tweak' : nonce_tweak,
        'ad' : ad
    }
    print("\nSending Emitter's message...")
    await emitter_queue.put(dumps(packet))
```

In [150]:

```
async def receiver(emitter_queue, receiver_queue):
        private_key, public_key = generate_keys()

        public_key = public_key.public_bytes(
                                    encoding=serialization.Encoding.PEM,
                                    format=serialization.PublicFormat.SubjectPublicKeyInfo
                                )

        emitter_message = loads(await emitter_queue.get())
        print("\nReceiver is receiving Emitter's public key...")

        emitter_ed448_pk = serialization.load_pem_public_key(emitter_message['ed448_pk'])

        try:
            emitter_ed448_pk.verify(emitter_message['signature'], emitter_message['message'])
        except:
            return 'ERROR --- Different ECDSA key used'

        print("\nEmitter's message is authentic!")

        emitter_public_key = serialization.load_pem_public_key(emitter_message['message'])
        shared_key = generate_shared_key(private_key, emitter_public_key)

        print("\nReceiver is sending his public key (signed by him)...")
        await receiver_queue.put(dumps(sign_message(public_key)))

        # receives ciphertext
        packet_bytes = await emitter_queue.get()
        packet = loads(packet_bytes)
        print('\nReceiver received ciphertext and tries to decrypt it.')

        plaintext = decrypt(packet['ciphertext'], packet['tag'], packet['nonce'], packet['nonce_tweak'], shared_key, p

        print("\nDECRYPTED MESSAGE: \n" + plaintext)
```

In [151]:

```
# Create two queues
emitter_message = asyncio.Queue()
receiver_queue = asyncio.Queue()

# Start the event loop and run the emitter and receiver coroutines
async def main():
    await asyncio.gather(emitter(emitter_message, receiver_queue), receiver(emitter_message, receiver_queue))

asyncio.run(main())
```

```
Emitter is sending his public key (signed by him)...

Receiver is receiving Emitter's public key...

Emitter's message is authentic!

Receiver is sending his public key (signed by him)...

Emitter is receiving Receiver's public key...

Emitter is verifying Receiver's message signature...

Receiver's message is authentic!

ORIGINAL MESSAGE (FROM EMITTER):
Finally we can communicate! :)

Encrypting Emitter's message...

ENCRYPTED MESSAGE:
¤ ´Ãµq· ;Ýýøö&±  D¯õwVTTÞÿÀóÌß

Sending Emitter's message...

Receiver received ciphertext and tries to decrypt it.

DECRYPTED MESSAGE:
Finally we can communicate! :)
```