TP1_Ex1

March 7, 2023

1 Exercício 1

1.1 Descrição do Problema

Para este primeiro exercício, foi nos pedido que implementassemos um comunicação privada assíncrona entre o emissor de uma mensagem (*Emitter*) e o respetivo recetor (*Receiver*). Para cifrar as mensagens deve-se utilizar uma cifra simétrica segura contra ataques aos "nounces", utilizando a autenticação da mensagem, do criptograma e dos metadados associados a ela; englobando-se, então, na classe **AEAD** (*Authenticated Encryption with Associated Data*). Deste modo, o exercício englobará a implementação de duas funções, uma para **cifrar** a mensagem, e outra, para **decifrar**. Adicionalmente, os *"nounces"* criados devem ser gerados aleatoriamente e devem ser utilizados apenas uma vez para trazer mais segurança à cifra.

No processo deverá ocorrer dois tipos de autenticação: o do criptograma e metadados, num modo *HMAC*, recorrendo ao uso de "*nounces*" e o dos participante após receber a mensagem (que se encontrará assinada pelo *emitter*).

As chaves utilizadas, acordadas entre os agentes, para cifrar a mensagem (*cipher_key*) e para calcular os códigos de autenticação do criptograma e metadados (*mac_key*) serão criadas recorrendo ao protocolo **ECDH**. A assinaturas **ECDSA** serão utilizados no âmbito de autenticar os agentes.

Para resolver este problema iremos utilizar a biblioteca **cryptography** do python.

[1]:
```
# Todos os imports
import os, json
import asyncio
import random
from pickle import dumps, loads
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes, hmac, serialization
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.exceptions import InvalidSignature

# Para a porção de código associada com as queues (na comunicação entre␣
 ↪participantes)
import nest_asyncio

nest_asyncio.apply()
```

1.2 Criação das chaves privadas/públicas

Previamente foi mencionado que iremos desenvolver uma cifra simétrica com autenticação de mensagem e de metadados. Para que se possa construir esta cifra é necessário que os participantes da comunicação conheçam a chave com a qual deverão cifrar/decifrar as mensagens, dado que a mesma será usada para ambas operações. Contudo, a partilha de uma chave é uma operação bastante arriscada, dado que um atacante pode intersetar a mensagem onde é partilhada a chave, e passar a ter capacidade de descodificar todas as mensagens trocadas. Para mitigar o risco desta operação, iremos utilizar o protocolo **ECDH** para obter duas chaves partilhadas, uma chave utilizada na cifragem da mensagem e outra na autenticação (*cipher_key* e *mac_key*, respetivamente).

Para que as chaves partilhadas possam ser criadas iremos necessitar que cada participante possua um par de chaves (**privada** e **pública**), dado que estes são necessárias para o protocolo **ECDH**. A chave **privada** é gerada sobre uma instância da **curva elíptica**, gerando uma chave do tipo *EllipticCurvePrivateKey*, retirada da biblioteca *cryptography* e a chave **pública** é criada a partir da privada.

O método :

```
def generateKeys()
```

cria e retorna estas duas chaves na forma de um **tuplo** (*private_key, public_key*).

[2]:
```
def generateKeys():
    curve = ec.SECP384R1()

    # gera uma chave privada
    private_key = ec.generate_private_key(curve)
    # gera uma chave publica em função da chave privada
    public_key = private_key.public_key()

    return private_key, public_key
```

1.3 Assinar a mensagem

No enunciado do exercício é indicado que as mensagens devem ser assinadas utilizando o algoritmo **ECDSA**. A assinatura gerada por este permite que o recetor possa verificar que a entidade que enviou a mensagem é o emissor com ele estava a comunicar.

A assinatura digital é obtida através da chave privada de um utilizador e qualquer pessoa que possua a chave pública poderá verificar que a assinatura está associada à chave privada.

O método permite assinar uma dada mensagem :

```
def signMsg(prv_key, msg)
```

...sendo o atributo *prv_key* a chave usada para assinar.

[3]:
```
def signMsg(prv_key, msg):

    signature = prv_key.sign(
    msg,
```

```
    ec.ECDSA(hashes.SHA3_256()))

    return signature
```

1.4 Criação das chaves partilhadas

Como mencionamos anteriormente, nesta cifra simétrica iremos utilizar o protocolo **ECDH** para obter uma chave partilhada por ambos os participantes. Para obter esta chave é necessário que cada participante possua uma chave privada e uma chave pública. Este par de chaves foi obtido previamente.

Cada participante deverá enviar ao outro duas chaves públicas, uma para cifrar a mensagem e outra para autenticação. Após terem recebido a chave pública do outro participante, será aplicada uma *exchange* entre as chaves. Desta forma obtemos duas chaves partilhadas *cipher_key* e *mac_key*, que podem ser usadas para cifrar/decifrar as mensagens trocadas e autenticação, respetivamente; evitando o envio de chaves partilhadas por um canal público.

Assim, cria-se o segredo partilhado entre os agentes para cifrar e autenticar os dados.

O método:

```
def generateShared(prv_cipher, peer_cipher, prv_mac, peer_mac)
```

... utiliza uma chave privada do agente e uma pública do seu *peer* para cada chave.

Neste caso, receberemos a *prv_cipher* que é a chave privada do próprio agente; e *peer_cipher* que é a chave pública do peer do agente que serão utilizadas para gerar a chave partilhada de cifragem.

Para gerar a chave partilhada para autenticar, receberemos *prv_mac* é a chave privada do próprio agente e *peer_mac* é a chave pública do peer do agente.

Faz-se a derivação da chave para ter mais segurança, destruindo alguma possível estrutura que possa existir, ao adicionar mais informação à chave.

[4]:
```
# Função que recebe as chaves privadas de um agente e as chaves públicas do␣
 ↪outro participante
# Devolve duas chaves partilhadas.
def generateShared(prv_cipher, peer_cipher, prv_mac, peer_mac):

    # Turn a string into a key
    peer_cipher_key = serialization.load_pem_public_key(peer_cipher)

    # Create shared key for encrypting
    cipher_key = prv_cipher.exchange(
        ec.ECDH(), peer_cipher_key)

    # Perform key derivation for protection
    cipher_derived_key = HKDF(
        algorithm=hashes.SHA3_256(),
        length=32,
        salt=None,
```

```
        info=b'handshake data',
    ).derive(cipher_key)

    # Turn a string into a key
    peer_mac_key = serialization.load_pem_public_key(peer_mac)

    # Create shared key for autentication
    mac_key = prv_mac.exchange(
        ec.ECDH(), peer_mac_key)

    # Perform key derivation
    mac_derived_key = HKDF(
        algorithm=hashes.SHA3_256(),
        length=32,
        salt=None,
        info=b'handshake data',
    ).derive(mac_key)

    return cipher_derived_key, mac_derived_key
```

1.5 Cifrar a mensagem

Neste caso, o método:

```
def encrypt(msg, cipher_key, mac_key)
```

... utiliza a cifragem autenticada com dados associados (AEAD), com uma cifra por blocos num modo **GCM** (*Galois Counter Mode*). Recorre-se à construção *AES-GCM* da biblioteca *cryptography*, com a chave *cipher_key* como seu argumento.

Para o método *encrypt* é necessário proporcionar um *nounce* e *associated data*. O *nounce* é óbtido através de um **gerador pseudo aleatório** (PRG), neste caso com um algoritmo de *hashing SHA3_256()*. Quanto à *associated data*, esta é gerada com um número aleatório de 16 bits.

Por fim, calcula-se o código para autenticação do criptograma e dos metadados associados, recorrendo a *mac_key*.

O dicionário retornado pelo método inclui texto cifrado, código *HMAC*, *nounce* utilizado e *associated data*.

[5]:
```
def encrypt(msg, cipher_key, mac_key):

    digest = hashes.Hash(hashes.SHA3_256())
    nounce = digest.finalize()

    ad = os.urandom(16)

    aesgcm = AESGCM(cipher_key)
    ct = aesgcm.encrypt(nounce, msg, ad)
```

```
    h = hmac.HMAC(mac_key, hashes.SHA3_256())
    h.update(ct)
    Hmac = h.finalize()

    return {"cipher": ct, "HMAC": Hmac, "nounce": nounce, "ad": ad}
```

1.6 Decifrar a mensagem

Para decifrar a mensagem recebida pelo emitter, devemos verificar o código *HMAC* de autenticação e, de seguida, decifrar com o método *decrypt*.

Para verificar se o código *HMAC* está correto, i.e. validar a autenticação do agente, é calculado um novo código *HMAC*, utilizando a chave *mac_ key* do agente que está a decifrar.

Este método será o inverso do método *encrypt*, portanto, no caso de passar a autenticação da mensagem, poderemos utilizar o método *decrypt* do AESGCM, com o *nounce* e *associated data* para decifrar o criptograma recebido e obter o *plaintext*.

[6]:
```
def decrypt(cipher, Hmac, cipher_key, mac_key, nounce, ad):

    h = hmac.HMAC(mac_key, hashes.SHA3_256())
    h.update(cipher)
    h_new = h.finalize()

    aesgcm = AESGCM(cipher_key)

    if h_new == Hmac :
        print("* HMAC validated. *")
        msg = aesgcm.decrypt(nounce, cipher, ad)

    else:
        print("* HMAC invalid. Aborting. *")
        return None

    return msg
```

1.6.1 Funções auxiliares para o Emitter/Receiver

Encontram-se, de seguida, 4 métodos auxiliares. Estes métodos serão utilizados para inicializar os agentes, isto é, criar das chaves privadas, públicas e de assinatura necessárias; para envio de mensagens; e para receção de mensagem.

Os métodos para envio (*send*) e receção (*receive*) de mensagens têm em consideração o código proporcionado pela equipa docente que utiliza as estruturas *queues*.

O método para **inicialização** (_init_comm()*emitter*, _init_comm()*receiver*) irá criar os três pares de chaves pública-privada (para cifrar, autenticar e assinar), para o *emitter*; criar dois pares de chaves (para decifrar e autenticar), para o *receiver*. Posteriormente, será feita a serialização das chaves públicas (para enviar ao *peer*) que serão adicionadas a um dicionário que será enviado ao

outro participante. Por último, esta função retorna um tuplo com as chaves e o pacote a enviar ao peer.

[7]:
```
def init_comm_emitter():
    ## Gerar chaves (cifrar e autenticar + assinar)
    prv_cipher_key, pub_cipher_key = generateKeys()
    prv_mac_key, pub_mac_key = generateKeys()
    prv_sign_key, pub_sign_key = generateKeys()

    ## Dicionário com a chavess públicas (serializadas)
    msg = {'cipher_key': pub_cipher_key.public_bytes(encoding=serialization.
 ↪Encoding.PEM,
                                        format=serialization.PublicFormat.
 ↪SubjectPublicKeyInfo
                                    ),
           'mac_key': pub_mac_key.public_bytes(encoding=serialization.Encoding.
 ↪PEM,
                                        format=serialization.PublicFormat.
 ↪SubjectPublicKeyInfo
                                    ),
           'sign_key': pub_sign_key.public_bytes(encoding=serialization.Encoding.
 ↪PEM,
                                        format=serialization.PublicFormat.
 ↪SubjectPublicKeyInfo
                                    )
            }

    return prv_cipher_key, prv_mac_key, prv_sign_key, msg

def init_comm_receiver():
    ## Gerar chaves (cifrar e autenticar)
    prv_cipher_key, pub_cipher_key = generateKeys()
    prv_mac_key, pub_mac_key = generateKeys()

    ## Dicionário com a chavess públicas (serializadas)
    msg = {'cipher_key': pub_cipher_key.public_bytes(encoding=serialization.
 ↪Encoding.PEM,
                                        format=serialization.PublicFormat.
 ↪SubjectPublicKeyInfo
                                    ),
           'mac_key': pub_mac_key.public_bytes(encoding=serialization.Encoding.
 ↪PEM,
                                        format=serialization.PublicFormat.
 ↪SubjectPublicKeyInfo
                                    )
            }
```

```
    return prv_cipher_key, prv_mac_key, msg

async def send(queue, msg):

    await asyncio.sleep(random.random())

    # put the item in the queue
    await queue.put(msg)

    await asyncio.sleep(random.random())

async def receive(queue):
    item = await queue.get()

    await asyncio.sleep(random.random())
    aux = loads(item)

    return aux
```

1.7 Emitter

A função que se segue, permite executar o comportamento do Emitter no protocolo desenvolvido. Nesta função são invocados os métodos que permitem gerar as diversas chaves públicas e privadas e partilhar as chaves públicas. Nesta são também recebidas as chaves públicas do Receiver, seguido pela obtenção das chaves partilhadas para cifragem e autenticação.

A função de cifrar é também chamada nesta função. Após a assinatura da mensagem, é feito o envio da mesma, terminando assim a execução do emitter.

[8]:
```
## Emitter Code
async def emitter(plaintext, queue):

    ## Gerar as chaves (privada e publica) & partilhar com participante
    prv_cipher_key, prv_mac_key, prv_sign_key, msg = init_comm_emitter()

    ## Enviar a chaves públicas para o peer
    await send(queue, dumps(msg))
    print("[E] SENDING PUBLIC KEYS")

    ## Receber as chaves públicas do peer
    msg = await receive(queue)
    print("[E] RECEIVED PEER PUBLIC KEYS")

    pub_peer_cipher = msg['cipher_key']
    pub_peer_mac = msg['mac_key']
    # print("[E] Receiver pub_key_cipher: " +str(msg['cipher_key']))
    # print("[E] Receiver pub_key_mac: " +str(msg['mac_key']))
```

```
    ## Criar as chaves partilhadas (cifrar/autenticar)
    cipher_shared, mac_shared = generateShared(prv_cipher_key, pub_peer_cipher,
                                               prv_mac_key, pub_peer_mac )

    ## Cifrar a mensagem
    pkg = encrypt(bytes(plaintext, 'utf-8'), cipher_shared, mac_shared)
    print("[E] MESSAGE ENCRYPTED")

    ## Assinar e enviar a mensagem
    pkg_b = dumps(pkg)
    sig = signMsg(prv_sign_key, pkg_b)

    # Enviar
    msg_final = {'sig': sig, 'msg': dumps(pkg)}

    print("[E] SENDING MESSAGE")
    await send(queue, dumps(msg_final))

    print("[E] END")
```

1.8 Receiver

Na função que se segue, definimos o código que será executado pelo Receiver, quando em comunicação com o Emitter.

Este deverá começar a sua execução a gerar as suas chaves privadas e públicas, e à espera da chegada das chaves públicas do emitter. Após receber as chaves públicas, este deverá criar as chaves partilhadas e enviar as suas chaves públicas.

Após a receção da mensagem, o receiver deverá verificar a assinatura digital enviada juntamente com a mensagem. Caso a assinatura seja inválida, uma exceção será apresentada. Se a assinatura for valida, poderá passar à decifração, onde será verificado o valor do HMAC. Se o processo de descodificação for bem sucedido, a mensagem *plaintext* deverá ser apresentada.

[9]:
```
## Receiver Code
async def receiver(queue):

    ## Gerar as chaves (privada e publica) & partilhar com participante
    prv_cipher_key, prv_mac_key, msg = init_comm_receiver()

    ## Receber as chaves publicas do peer
    pub_keys = await receive(queue)

    pub_peer_cipher = pub_keys['cipher_key']
    pub_peer_mac = pub_keys['mac_key']
    pub_peer_sign = pub_keys['sign_key']

    # print("[R] Emitter pub_key_cipher: " +str(pub_peer_cipher))
```

```
    # print("[R] Emitter pub_key_mac: " +str(pub_peer_mac))

    ## Gerar shared keys
    cipher_shared, mac_shared = generateShared(prv_cipher_key, pub_peer_cipher,
                                               prv_mac_key, pub_peer_mac)

    ## Enviar as chaves públicas ao peer
    await send(queue, dumps(msg))
    print("[R] AWAIT CIPHER")
    ciphertext = await receive(queue)
    print("[R] CIPHER RECEIVED")

    ## Receber a mensagem (Assinatura)
    peer_sign_key = serialization.load_pem_public_key(pub_peer_sign)

    ## Validar a correçaõ da assinatura
    try:
        peer_sign_key.verify(ciphertext['sig'], ciphertext['msg'], ec.
↪ECDSA(hashes.SHA3_256()))
    except InvalidSignature:
        print(" *Assinatura inválida. Aborting. *")

    msg_dict = loads(ciphertext['msg'])

    ## Decifrar essa mensagem
    plain_text = decrypt(msg_dict['cipher'],msg_dict['HMAC'], cipher_shared,␣
↪mac_shared, msg_dict['nounce'], msg_dict['ad'])

    if(plain_text != None):
        ## Apresentar no terminal
        print("[R] Plaintext: " + plain_text.decode('utf-8'))
```

1.9 Função "main" para teste da comunicação

A função que se segue permite-nos testar todo o protocolo implementado. Para tal, iremos utilizar uma queue para simular um canal de comunicação entre o Emitter e o Receiver.

[10]:
```
def ex1(msg):
    loop = asyncio.get_event_loop()
    queue = asyncio.Queue(10)
    asyncio.ensure_future(emitter(msg, queue), loop=loop)
    loop.run_until_complete(receiver(queue))
```

1.9.1 Testes:

[11]:
```
ex1("HELLO WORLD!")
```

```
[E] SENDING PUBLIC KEYS
[R] AWAIT CIPHER
[E] RECEIVED PEER PUBLIC KEYS
[E] MESSAGE ENCRYPTED
[E] SENDING MESSAGE
[R] CIPHER RECEIVED
* HMAC validated. *
[R] Plaintext: HELLO WORLD!
```

[12]:
```
ex1("olá, estou a enviar esta mensagem para verificar o protocolo de comunicação␣
↪implementado")
```

```
[E] END
[E] SENDING PUBLIC KEYS
[E] RECEIVED PEER PUBLIC KEYS
[E] MESSAGE ENCRYPTED
[E] SENDING MESSAGE
[R] AWAIT CIPHER
[E] END
[R] CIPHER RECEIVED
* HMAC validated. *
[R] Plaintext: olá, estou a enviar esta mensagem para verificar o protocolo de
comunicação implementado
```