Estruturas Criptográficas 2022/23

TP1. Problema 1

Grupo 7. Leonardo Berteotti e Paulo R. Pereira

Pretende-se a criação de uma comunicação privada e assíncrona entre um agente *Emitter* e um agente *Receiver*.

A comunicação começa com a troca das duas chaves públicas de cada agente (criadas para construir tanto a chave para cifra como a chave para a autenticação). Depois, cada agente irá gerar as duas respetivas chaves partilhadas, *cipher_key* (chave de cifra) e *mac_key* (chave de autenticação).

As assinaturas digitais (neste caso **ECDSA**) serão usadas para garantir a autenticação dos agentes.

Note-se que o *Emitter* envia mensagens ao *Receiver* que sejam autenticadas com a chave partilhada de autenticação e cifradas com a chave de cifra.

In [369]:

```
import asyncio
import nest_asyncio
nest_asyncio.apply()
import os
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from pickle import dumps, loads
```

Como já referido, 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) **ECDH** (Elliptic Curve Diffie-Hellman Key Exchange).

Como também já referido, é necessário que cada agente gere dois pares de chaves assimétricas, uma para cifrar as mensagems e a outra para a autenticação dos agentes. 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 as 4 chaves referidas.

In [370]:

```
def generate_keys():
    cipher_sk = ec.generate_private_key(ec.SECP384R1())
    cipher_pk = cipher_sk.public_key()

    mac_sk = ec.generate_private_key(ec.SECP384R1())
    mac_pk = mac_sk.public_key()

    return (cipher_sk, cipher_pk, mac_sk, mac_pk)
```

A função *generate_shared_keys* trata de gerar as chaves partilhadas, *cipher_key* e *mac_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 [371]:

```
def generate_shared_keys(cipher_sk, received_cipher_pk, mac_sk, received_mac_pk):
    cipher_key = HKDF(
        algorithm = hashes.SHA256(),
        length = 32,
        salt = None,
        info = b'handshake data',
    ).derive(cipher_sk.exchange(ec.ECDH(), received_cipher_pk))

    mac_key = HKDF(
        algorithm = hashes.SHA256(),
        length = 32,
        salt = None,
        info = b'handshake data',
    ).derive(mac_sk.exchange(ec.ECDH(), received_mac_pk))

    return cipher_key, mac_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 **ECDSA**. 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. Note-se que para verificar é necessário usar o mesmo algoritmo de *hashing*.

In [372]:

```
def sign_message(message):
    private_key = ec.generate_private_key(
        ec.SECP384R1()
    )
    signature = private_key.sign(
        message,
        ec.ECDSA(hashes.SHA256())
    )
    packet = {'signature' : signature,
              'message'   : message,
              'ecdsa_pk'  : private_key.public_key().public_bytes(
                                                    encoding=serialization.Encoding.PEM,
                                                    format=serialization.PublicFormat.SubjectPublicKeyInfo
                                                )
             }
    return packet
```

A função *encrypt* será usada pela *Emitter* para cifrar a mensagem a enviar ao *Receiver*. A cifra a ser utilizada é a cifra simétrica **AES** no modo **GCM**, com a chave de cifra partilhada. Foi escolhida esta cifra por fornecer tanto confidencialidade como integridade do texto cifrado e por fornecer integridade para dados associados que não são cifrados. Além disso, o modo **GCM** é seguro contra ataques aos *nonce*. Neste tipo de ataque, o atacante envia a mensagem copiada para o destinatário diversas vezes. O *nonce*, que é gerado a partir de uma função de hash em modo XOF (SHAKE256), permite identificar cada mensagem com um identificador exclusivo, o que reduz significativamente a probabilidade desses ataques serem bem sucedidos.

In [373]:

```
def encrypt(plaintext, cipher_key, mac_key, ad):

    digest = hashes.Hash(hashes.SHAKE256(16))
    nonce = digest.finalize()

    aesgcm = AESGCM(cipher_key)
    ciphertext = aesgcm.encrypt(nonce, plaintext, ad)

    h = hmac.HMAC(mac_key, hashes.SHA256())
    h.update(ciphertext)
    h.update(ad)
    tag = h.finalize()

    return (ciphertext, nonce, tag)

def decrypt(ciphertext, nonce, tag, cipher_key, mac_key, ad):
    h = hmac.HMAC(mac_key, hashes.SHA256())
    h.update(ciphertext)
    h.update(ad)

    try:
        h.verify(tag)
    except:
        return 'ERROR --- Different MAC key used'

    # decrypt ciphertext
    aesgcm = AESGCM(cipher_key)
    original_plaintext = aesgcm.decrypt(nonce, ciphertext, ad)
    return original_plaintext
```

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*, o *nonce*, os dados associados e a *tag* de autenticação. Com estes dados, o *Receiver* é então capaz de decifrar a mensagem.

In [374]:

```
async def emitter(emitter_queue, receiver_queue):
    cipher_sk, cipher_pk, mac_sk, mac_pk = generate_keys()

    public_keys = {'cipher_pk' : cipher_pk.public_bytes(
                                    encoding=serialization.Encoding.PEM,
                                    format=serialization.PublicFormat.SubjectPublicKeyInfo
                                    ),
                   'mac_pk'    : mac_pk.public_bytes(
                                    encoding=serialization.Encoding.PEM,
                                    format=serialization.PublicFormat.SubjectPublicKeyInfo
                                    )
                  }

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

    receiver_pks_bytes = await receiver_queue.get()
    receiver_pks = loads(receiver_pks_bytes)
    print("\nEmitter is receiving Receiver's public keys...")

    print("\nEmitter is verifying Receiver's message signature...")
    receiver_ecdsa_pk = serialization.load_pem_public_key(receiver_pks['ecdsa_pk'])

    try:
        receiver_ecdsa_pk.verify(receiver_pks['signature'], receiver_pks['message'], ec.ECDSA(hashes.SHA256()))
    except:
        return 'ERROR --- Different ECDSA key used'

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

    msg = loads(receiver_pks['message'])
    receiver_cipher_pk = serialization.load_pem_public_key(msg['cipher_pk'])
    receiver_mac_pk = serialization.load_pem_public_key(msg['mac_pk'])

    # generate shared keys
    cipher_key, mac_key = generate_shared_keys(cipher_sk, receiver_cipher_pk, mac_sk, receiver_mac_pk)

    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, nonce, tag = encrypt(dumps(msg_to_send), cipher_key, mac_key, ad)

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

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

In [375]:

```
async def receiver(emitter_queue, receiver_queue):
        cipher_sk, cipher_pk, mac_sk, mac_pk  = generate_keys()

        public_keys = {'cipher_pk': cipher_pk.public_bytes(
                                        encoding=serialization.Encoding.PEM,
                                        format=serialization.PublicFormat.SubjectPublicKeyInfo
                                    ),
                       'mac_pk'   : mac_pk.public_bytes(
                                        encoding=serialization.Encoding.PEM,
                                        format=serialization.PublicFormat.SubjectPublicKeyInfo
                                    )
                      }

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

        emitter_ecdsa_pk = serialization.load_pem_public_key(emitter_pks['ecdsa_pk'])

        try:
            emitter_ecdsa_pk.verify(emitter_pks['signature'], emitter_pks['message'], ec.ECDSA(hashes.SHA256()))
        except:
            return 'ERROR --- Different ECDSA key used'

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

        msg = loads(emitter_pks['message'])
        emitter_cipher_pk = serialization.load_pem_public_key(msg['cipher_pk'])
        emitter_mac_pk = serialization.load_pem_public_key(msg['mac_pk'])

        # generate shared keys
        cipher_key, mac_key = generate_shared_keys(cipher_sk, emitter_cipher_pk, mac_sk, emitter_mac_pk)

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

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

        plaintext = decrypt(packet['ciphertext'], packet['nonce'], packet['tag'], cipher_key, mac_key, packet['ad'])

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

In [376]:

```
# Create two queues
emitter_queue = 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_queue, receiver_queue), receiver(emitter_queue, receiver_queue))

asyncio.run(main())
```

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

Receiver is receiving Emitter's public keys...

Emitter's message is authentic!

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

Emitter is receiving Receiver's public keys...

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:
  îÖ@    ©H ±GJ  Ô  gÌ®  [↕ ÈÓTò& Ô  eêò³ù îÍ ?  kýa|gi=oÙq  ïe#±ÆÎé

Sending Emitter's message...

Receiver received ciphertext and is trying to decrypt it...

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