TP1 | Exercício 1 | Grupo 15

Nuno Mata - pg44420

Pedro Araújo - pg50684

Problema 1

Use a package Criptography para:

1. **Criar um comunicação privada assíncrona entre um agente Emitter e um agente Receiver que cubra os seguintes aspectos:**
    - **Autenticação do criptograma e dos metadados (associated data). Usar uma cifra simétrica num modo HMAC que seja seguro contra ataques aos "nounces".**
    - **Os "nounces" são gerados por um gerador pseudo aleatório (PRG) construído por um função de hash em modo XOF.**
    - **O par de chaves *cipher_key*, *mac_key*, para cifra e autenticação, é acordado entre agentes usando o protocolo ECDH com autenticação dos agentes usando assinaturas ECDSA.**

In [74]:

```
!pip install cryptography
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publi
c/simple/
Requirement already satisfied: cryptography in /usr/local/lib/python3.8/dist-packages (39
.0.2)
Requirement already satisfied: cffi>=1.12 in /usr/local/lib/python3.8/dist-packages (from
cryptography) (1.15.1)
Requirement already satisfied: pycparser in /usr/local/lib/python3.8/dist-packages (from
cffi>=1.12->cryptography) (2.21)
```

In [75]:

```
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes, hmac, cmac
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import serialization,hashes
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.exceptions import *
import os
import base64
import sys
import io
```

Partindo da criação das chaves privadas para o Emitter e Reciever é possível gerar as suas respetivas chaves públicas, que serão necessárias no processo de verificação. Estas chaves (privadas e públicas) são ainda usadas para gerar chaves partilhadas entre ambas as entidades para depois serem utilizadas na fase de autênticação das mensagens de forma a assegurar a sua autênticidade e integridade.

In [76]:

```
def generate_public_key(private_key):
        return private_key.public_key()

def key_derivation(private_key,public):
        shared_key = private_key.exchange(ec.ECDH(), public)
```

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

        return derived_key
```

Em seguida geramos uma assinatura através da combinação da chave partilhada e a função hash SHA-256, que vai ser concatenada na mensagem enviada para o recetor (Receiver) para que este posteriormente possa verificar a autenticidade do emissor (Emitter).

In [77]:

```
def auth(message,derived_key):
        h = hmac.HMAC(derived_key, hashes.SHA256())
        h.update(message)
        return h.finalize()
```

Aqui é gerado um nonce, que vai assegurar que a mensagem *plaintext* cifrada vai sempre resultar num *ciphertext* diferente. Para isto é utilizada a funçao *urandom()* que cria uma string, neste caso de 16 bytes, totalmente aleatória.

A cifra usada posteriormente é a AESGCM, fazendo a cifra da mensagem ao qual é concatenado o nonce e a assinatura para que seja enviada ao recetor.

curiosidade: A função *urandom()* é mais recomendada para este tipo de tarefa do que a mais conhecida *random* do *python* pois afirma-se que a sua "aleatoriedade" provém uma entropia de muitas fontes imprevisíveis, tornando-a mais aleatória.

In [78]:

```
def cifraGCM(nonce,message,derived_key):
    aesgcm = AESGCM(derived_key)
    return aesgcm.encrypt(nonce, message, b'some associated data')

def send_message(message,derived_key):
        signature = auth(b'Assinatura OK.',derived_key)
        message = message.encode('utf-8')
        nonce = os.urandom(16)
        concatenated_m = cifraGCM(nonce,message,derived_key)

        return signature + nonce + concatenated_m
```

Dividem-se ou faz-se o desempacotamento dos dados do lado do recetor.

In [79]:

```
def unpack_data(dados):
      signature = dados[0:32]
      nonce = dados[32:32+16]
      concatenated_m = dados[32+16:]

      return signature, nonce, concatenated_m
```

Depois de desempacotados os dados faz-se a verificação da assinatura (usando o HMAC) da mensagem recebida pelo recetor para podermos passar para o passo de decifragem da mensagem.

In [80]:

```
def verify(signature,derived_key):
      h = hmac.HMAC(derived_key, hashes.SHA256())
      h.update(b'Assinatura OK.')
      return h.verify(signature)
```

Validação da assinatura da mensagem recebida e passagem à decifragem da mensagem caso a verifi

validando-se a assinatura da mensagem recebida passamos a decifragem da mensagem usando a cifra AESGCM

In [81]:

```
def decifraGCM(nonce, concatenated_m, derived_key):
    aesgcm = AESGCM(derived_key)
    return aesgcm.decrypt(nonce, concatenated_m, b'some associated data')

def read_message(concatenated_m, derived_key):
      signature, nonce, concatenated_m = unpack_data(concatenated_m)
      try :
          verify(signature, derived_key)
      except:
          raise Exception("Falha na autenticidade da chave")

      texto_limpo = decifraGCM(nonce, concatenated_m, derived_key)

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

É enviada a mensagem partindo do emissor para o recetor, que é todo o processo que envolve ser utilizada assinatura, chave partilhada, mensagem e nonce para assegurar a integridade e autenticidade da mensagem.

Por fim é lida a mensagem pelo recetor após efetuar os passos de verificação da mensagem anteriores.

In [82]:

```
def main():
    receiver_private_key = ec.generate_private_key(ec.SECP384R1())
    emiter_private_key = ec.generate_private_key(ec.SECP384R1())
    emiter_derived_key = key_derivation(emiter_private_key,generate_public_key(receiver_p
rivate_key))
    receiver_derived_key = key_derivation(receiver_private_key,generate_public_key(emiter
_private_key))

    dados = send_message("Olá Grupo 15", emiter_derived_key)
    print('encrypted text:',dados)

    try:
        plain_text = read_message(dados,receiver_derived_key)
        print('decrypted text:', plain_text)
    except:
        print("Falha na autenticação da chave")

main()
```

```
encrypted text: b'\xe0\xfbIA\x9d\xf8D`1yf\xbb\xbf\xa3 \x9f\x1d\xb0\xa3\xe8"\x99\x92\xb5\x
ae\xc5\xa9ZN\xf3\xa1\xa4Y\xd8s\xf1\x10\n\r\xcc1\xa9\x94q0\xd7\x17qh\x99\xdb\x9fM3\xf7\xd3
\xac\x15"\xf9}D\n7"3\xae++4\xbd\xd1\x98\xb6\xc2\xf4\xd7'
decrypted text: Olá Grupo 15
```