TP1_Ex1

March 7, 2023

1 TP1

1.1 Grupo 17:

PG50315 - David Alexandre Ferreira Duarte

PG51247 - João Rafael Cerqueira Monteiro

1.2 Exercício 1.

Use a package Criptography para 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.

[1]:
```
!pip install cryptography
!pip install nest_asyncio
```

```
Requirement already satisfied: cryptography in
/home/sleiman/mambaforge/envs/sage/lib/python3.10/site-packages (39.0.2)
Requirement already satisfied: cffi>=1.12 in
/home/sleiman/mambaforge/envs/sage/lib/python3.10/site-packages (from
cryptography) (1.15.1)
Requirement already satisfied: pycparser in
/home/sleiman/mambaforge/envs/sage/lib/python3.10/site-packages (from
cffi>=1.12->cryptography) (2.21)
Requirement already satisfied: nest_asyncio in
/home/sleiman/mambaforge/envs/sage/lib/python3.10/site-packages (1.5.6)
```

[2]:
```
# Imports
import os, sys
import random
import asyncio
import nest_asyncio
from pickle import dumps, loads
from cryptography.hazmat.primitives import hashes, hmac, serialization
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives.serialization import load_pem_public_key
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.exceptions import InvalidSignature
```

[3]:
```
nest_asyncio.apply()
```

1.2.1 Geração de Chaves

A resuloção deste problema começou por criar funções que gerassem chaves asimétricas, gerassem uma assinatura de acordo com a *cipher_key* privada e com um pacote que contém a *cipher_key* e *mac_key* públicas. Após isso foram feitas funções que gerassem uma chave partilhada de acordo com as chaves privadas do *emmiter* e/ou *receiver*, que de acordo com as chaves partilhadas geradas gerassem as chaves derivadas, estas ao ser utilizadas trazem uma maior segurança à cifragem das mensagens. Por fim foi feita uma função que gerava o mac de acordo com a *mac_key* partilhada e com um criptograma.

[4]:
```
def gerarChaves():
    cifraChavePrivada = ec.generate_private_key(ec.SECP384R1())
    cifraChavePublica = cifraChavePrivada.public_key()

    macChavePrivada = ec.generate_private_key(ec.SECP384R1())
    macChavePublica = macChavePrivada.public_key()

    mensagem = {'sms_cipher': cifraChavePublica.
 ↪public_bytes(encoding=serialization.Encoding.PEM, format=serialization.
 ↪PublicFormat.SubjectPublicKeyInfo),
                'sms_mac': macChavePublica.public_bytes(encoding=serialization.
 ↪Encoding.PEM, format= serialization.PublicFormat.SubjectPublicKeyInfo)}

    return dumps(mensagem), cifraChavePrivada, macChavePrivada
```

[5]:
```
def gerarAssinatura(sms, chavePrivada):

    assinatura = chavePrivada.sign(sms, ec.ECDSA(hashes.SHA3_256()))

    smsFinal = {'message': sms,
                'signature': assinatura,
```

```
                'pub_key': chavePrivada.public_key().
↪public_bytes(encoding=serialization.Encoding.PEM, format=serialization.
↪PublicFormat.SubjectPublicKeyInfo)}

    return smsFinal
```

[6]:
```
def gerarChavePartilhada(sms, cifraChavePrivada, macChavePrivada):
    cifraChavePublica = load_pem_public_key(sms['sms_cipher'])
    macChavePublica = load_pem_public_key(sms['sms_mac'])

    cifraChavePartilhada = cifraChavePrivada.exchange(ec.ECDH(),␣
↪cifraChavePublica)
    macChavePartilhada = macChavePrivada.exchange(ec.ECDH(), macChavePublica)

    return cifraChavePartilhada, macChavePartilhada
```

[7]:
```
def gerarChaveDerivada(cifraChavePartilhada, macChavePartilhada):
    cifraDerivada = HKDF(algorithm=hashes.SHA3_256(), length=32, salt=None,␣
↪info=b'handshake data',).derive(cifraChavePartilhada)
    macDerivada = HKDF(algorithm=hashes.SHA3_256(), length=32, salt=None,␣
↪info=b'handshake data',).derive(macChavePartilhada)

    return cifraDerivada, macDerivada
```

[8]:
```
def gerarMac(chave, crypto):
    h = hmac.HMAC(chave, hashes.SHA3_256())
    h.update(crypto)
    return h.finalize()
```

1.2.2 Cifragem e Decifragem

Nesta fase do trabalho, decidimos fazer duas funções separadas para cifrar uma mensagem de acordo com as chaves *cipher_key* e *mac_key*. Tanto para a cifragem como para a decifragem utilizou-se o algoritmo **SHA3_256**

Para a função de cifrar a mensagem, começamos por gerar um *nonce* em modo *XOF*. De seguida, criou-se uma variável `cifra` que vai permitir cifrar o texto através do `AESGCM` com a *cipher_key*, com o *nonce* e com dados adicionais gerados na altura da cifragem.

Após a cifragem do texto, é necessário gerar um *mac* com o pacote que contém o *nonce* e a mensagem cifrada.

Por fim, esta função retorna um dicionário com o pacote mencionado anteriormente, a *mac_key* que foi gerada e com os dados adicionais.

[9]:
```
def cifrar(plaintext, cifraChave, macChave):

    xof = hashes.Hash(hashes.SHA3_256())
```

```
    nonce = xof.finalize()
    additionalData = os.urandom(16)

    try:
        cifra = AESGCM(cifraChave)
    except ValueError as e:
        print(f"Erro ao gerar chave: {e}")
    except Exception as e:
        print(f"Erro inesperado: {e}")


    ciphertext = cifra.encrypt(nonce, bytes(plaintext, "utf-8"), additionalData)
    sms = {'nounce': nonce,
           'cipher_text': ciphertext}
    chaveMac = gerarMac(macChave, dumps(sms))

    return {'message': sms, 'mac_key': chaveMac, 'associated_data':␣
↪additionalData}
```

Por sua vez, a função de decifrar o texto cifrado funciona como a função de cifrar, onde este obtém o *nonce* através do pacote que se passa como argumento, a criação da variável *cifra* que permite decifrar o texto utilizando o `AESGCM.decrypt()`

[10]:
```
def decifrar(ciphertext, cifraChave, macChave):
    texto = ciphertext['message']

    #xof = hashes.Hash(hashes.SHA3_256())
    #nonce = xof.finalize()
    nonce = texto['nounce']
    additionalData = ciphertext['associated_data']

    cifra = AESGCM(cifraChave)
    plaintext = cifra.decrypt(nonce, texto['cipher_text'], additionalData)

    return plaintext.decode()
```

1.2.3 Emitter

A função *emitter* é uma função `async` que recebe uma `queue` que vai servir como meio de comunicação com a função *receiver*.

Esta função começa por gerar e armazenar as suas chaves privadas, e a sua assinatura que por sua vez será colocada na `queue` para o outro lado da comunicação a receber e poder validar as chaves enviadas.

Após isso, a função entra num estado de espera enquanto aguarda que seja colocado algo na`queue` para poder validar a assinatura e chaves do *receiver*.

Um vez feita a validação, inicia-se a geração das chaves partilhadas e por sua vez as chaves derivadas,

para poder cifrar a mensagem. Quando terminada a geração destas chaves, é feita então a cifragem da mensagem pretendida e a geração do *mac* que será colocado no dicionário devolvido pela função de cifragem.

De seguida, envia-se este dicionário com o texto cifrado e outra informação necessária pela `queue` para o *receiver*

[11]:
```
async def emitter(queue, plaintext):

    print("E: " + plaintext)

    sms, cifraChavePrivada, macChavePrivada = gerarChaves()
    assinatura = gerarAssinatura(sms, cifraChavePrivada)

    print("E: Enviar Assinatura...")
    await asyncio.sleep(random.random())
    await queue.put(assinatura)
    print("E: Assinatura Enviada")
    await asyncio.sleep(random.random())

    print("E: A Receber Assinatura...")
    chavesRecetor = await queue.get()
    print("E: Assinatura Recebida")

    if chavesRecetor is None:
        print("ERRO - MENSAGEM VAZIA")

    ecdsaPublico = load_pem_public_key(chavesRecetor['pub_key'])

    ecdsaPublico.verify(chavesRecetor['signature'], chavesRecetor['message'],␣
↪ec.ECDSA(hashes.SHA3_256()))
    pacoteSMS = loads(chavesRecetor['message'])
    cifraChavePartilhada, macChavePartilhada = gerarChavePartilhada(pacoteSMS,␣
↪cifraChavePrivada, macChavePrivada)
    cifraChaveDerivada, macChaveDerivada =␣
↪gerarChaveDerivada(cifraChavePartilhada, macChavePartilhada)

    mensagem = cifrar(plaintext, cifraChaveDerivada, macChaveDerivada)
    hmacChave = gerarMac(macChaveDerivada, macChaveDerivada)
    print("E: Mensagem cifrada")

    mensagem['hmac_key'] = hmacChave

    print("E: Enviar Mensagem...")
    await asyncio.sleep(random.random())
    await queue.put(mensagem)
    print("E: Mensagem Enviada")
    await asyncio.sleep(random.random())
```

1.2.4 Receiver

A função *receiver* é uma função `async` que funciona como a função *receiver*.

Esta função começa por gerar e armazenar as suas chaves privadas, e a sua assinatura.

Após isso, a função entra num estado de espera enquanto aguarda que seja colocado algo na`queue` para poder validar a assinatura e chaves do *emitter*.

Um vez feita a validação, inicia-se a geração das chaves partilhadas e por sua vez as chaves derivadas, para poder cifrar a mensagem. Quando terminada a geração destas chaves, é feita então a cifragem da mensagem pretendida e a geração do *mac* que será colocado no dicionário devolvido pela função de cifragem. E coloca-e na `queue` a assinatura, para o outro lado da comunicação a receber e poder validar as chaves enviadas.

Após isso, esta função entra no modo espera enquanto aguarda pelo dicionário com o texto cifrado e outra informação relevante seja colocado na `queue`. Quando recebido essa mensagem, começa-se por verificar a chave *mac* que está no dicionário, e caso esta seja válida inicia-se a decifragem do texto com a função *decifrar*.

[12]:
```
async def receiver(queue):

    pkg, cifraChavePrivada, macChavePrivada = gerarChaves()
    assinatura = gerarAssinatura(pkg, cifraChavePrivada)

    print("R: Receber Assinatura")
    assinaturaEmissor = await queue.get()
    print("R: Assinatura Recebida")

    ecdsaPublico = load_pem_public_key(assinaturaEmissor['pub_key'])

    try:
        ecdsaPublico.
↪verify(assinaturaEmissor['signature'],assinaturaEmissor['message'], ec.
↪ECDSA(hashes.SHA3_256()))

        pkg_msg = loads(assinaturaEmissor['message'])
        cifraChavePartilhada, macChavePartilhada =␣
↪gerarChavePartilhada(pkg_msg,cifraChavePrivada,macChavePrivada)
        cifraChaveDerivada, macChaveDerivada =␣
↪gerarChaveDerivada(cifraChavePartilhada, macChavePartilhada)

        print("R: Enviar Assinatura...")
        await asyncio.sleep(random.random())
        await queue.put(assinatura)
        print("R: Assinatura Enviada")
        await asyncio.sleep(random.random())

        print("R: Receber Mensagem...")
        message = await queue.get()
```

```
        print("R: Mensagem Recebida")

        hmac_key = message['hmac_key']
        associatedData = message['associated_data']

        if hmac_key == gerarMac(macChaveDerivada, macChaveDerivada):
            final_message = decifrar(message, cifraChaveDerivada,␣
↪macChaveDerivada)
            print("R: " + final_message)
        else:
            print('ERRO - Chaves diferentes em uso.')
    except InvalidSignature:
        print("ERRO: A mensagem não é autenticada.")
```

1.2.5 Main

A função *main* é a função encarregue de iniciar a `queue` e o `loop` necessário para que exista uma comunicação entre o *emitter* e o *receiver*

[13]:
```
def main():

    plaintext = 'Estruturas Criptograficas: Grupo 17'

    loop = asyncio.get_event_loop()
    queue = asyncio.Queue(10)
    asyncio.ensure_future(emitter(queue, plaintext), loop=loop)
    loop.run_until_complete(receiver(queue))
```

[14]:
```
main()
```

```
E: Estruturas Criptograficas: Grupo 17
E: Enviar Assinatura…
R: Receber Assinatura
E: Assinatura Enviada
R: Assinatura Recebida
R: Enviar Assinatura…
E: A Receber Assinatura…
R: Assinatura Enviada
E: Assinatura Recebida
E: Mensagem cifrada
E: Enviar Mensagem…
R: Receber Mensagem…
E: Mensagem Enviada
R: Mensagem Recebida
R: Estruturas Criptograficas: Grupo 17
```