TP0-Problema1

March 7, 2022

1 TRABALHO PRÁTICO 0 - GRUPO 14

1.1 Problema 1

O problema 1 consiste em criar uma comunicação privada e assíncrona entre um emissor (emitter) e um recetor (receiver). A comunicação inicia-se com a transmissão de duas chaves públicas do emissor para o recetor e vice versa. Cada entidade irá gerar duas chave partilhadas, uma a ser usada na autenticação e outra para a cifragem. A comunicação deve manter a autenticidade e integridade das mensagens trocadas através do uso de assinaturas digitais (DSA). O emissor irá enviar mensagens ao recetor que sejam autenticadas com a chave partilhada de autenticação e cifradas com a chave de cifragem. O recetor irá fazer os processor inversos para obter a mensagem enviada. De seguida apresentamos a abordagem usada para a resolução do problema juntamente com o código em Python explicado.

1.1.1 Resolução do Problema

Imports

[1]:
```
import os
from multiprocessing import Process, Pipe
from pickle import dumps, loads
from cryptography.hazmat.primitives import hashes, hmac, serialization
from cryptography.hazmat.primitives.asymmetric import dh, dsa
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.serialization import load_pem_public_key
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.exceptions import InvalidSignature
```

Geração de chaves assimétricas A resolução deste problema começou pelo processo de gerar as chaves assimétricas necessárias para que o emissor e o recetor obtenham as chaves usadas nos processos de autenticação e cifragem. Para isso utilizamos o protocolo DH (Diffie–Hellman key exchange) que consiste num método seguro de troca de chaves públicas de forma a que tanto o emissor como o rector consigam acordar numa chave comum, isto é, uma chave partilhada entre ambos.

O desenvolvimento deste protocolo iniciou-se na geração dos parâmetros necessários para à posterior criação de chaves privadas e públicas. É de notar que tanto o emissor como recetor devem criar as chaves assimétricas utilizando os mesmo parâmetros de criação. Após a geração dos parâmetros foi necessário gerar dois pares de chaves assimétricas, uma para a autenticação e outra para a cifragem. A chave pública de cada um destes pares será enviado do emissor para o recetor e vice-versa. Desta forma, caso um intruso tenha acesso à conversa entre as entidades este só consegue obter a chave pública de cada uma mas não consegue gerar a chave partilhada pois não tem acesso aos parâmetros que criam o par de chaves. Para todo este processo utilizou-se o algoritmo assimétrico **DH** da *package Cryptography*.

[2]:
```
#geração dos parâmetros para a criação dos pares de chaves
parameters = dh.generate_parameters(generator=2, key_size=2048)

#FUNÇÃO: criação dos pares de chaves
def generateKeys():

    #par de chaves para o processor de cifragem
    privateKey_cipher = parameters.generate_private_key()
    publicKey_cipher = privateKey_cipher.public_key()

    #par de chaves para o processor de autenticação
    privateKey_mac = parameters.generate_private_key()
    publicKey_mac = privateKey_mac.public_key()

    #mensagem a ser enviada entre as entidades
    package = { 'pk_cipher': publicKey_cipher.
 ↪public_bytes(encoding=serialization.Encoding.PEM,
                                                          format=serialization.
 ↪PublicFormat.SubjectPublicKeyInfo),
                'pk_mac': publicKey_mac.public_bytes(encoding=serialization.
 ↪Encoding.PEM,
                                                      format=serialization.
 ↪PublicFormat.SubjectPublicKeyInfo)}

    return dumps(package), privateKey_cipher, privateKey_mac
```

Geração da assinatura digital De forma a manter a autenticidade, integridade e não-repúdio na transmissão das chaves públicas entre as entidades, implementou-se o algoritmo DSA. Este algoritmo consiste em assinar a mensagem a ser enviada utilizando um par de chaves assimétricas. A chave privada será usada para criar a assinatura digital, enquanto que a chave pública será usada para verificar se a assinatura é válida. Desta forma, uma entidade (emissor ou recetor) consegue confirmar se a mensagem que recebeu foi corretamente assinada, verificando se a mensagem é autêntica ou não. Para o desenvolvimento deste protocolo foi utilizado o algoritmo assimétrico **DSA** da *package Cryptography* que irá gerar o par de chaves e a assinatura. De forma a que a entidade consiga confirmar a autenticidade da mensagem é necessário enviar a assinatura e a chave pública juntamente com a mensagem.

[3]:
```
#FUNÇÃO: criação da assinatura digital
def generateSignature(pkg):

    #criar o par de chaves a ser usado na assinatura
    privateKey_DSA = dsa.generate_private_key(key_size=1024)

    #assina a mensage pkj
    signature = privateKey_DSA.sign(pkg,hashes.SHA256())

    #mensagem a ser enviada
    finalPkg = {'message': pkg, 'signature': signature,
                'pub_key':privateKey_DSA.public_key().
 ↪public_bytes(encoding=serialization.Encoding.PEM,

 ↪format=serialization.PublicFormat.SubjectPublicKeyInfo)}

    return finalPkg
```

Geração das chaves partilhadas Após receber a mensagem com as chaves públicas, cada entidade terá de gerar as chaves partilhadas que serão usadas no processo de autenticação e cifragem. A partir do algoritmo DH conseguimos criar a chave partilhada utilizando a chave pública recebida na mensagem e a chave privada de cada entidade através da função *exchange*. Como a chave partilhada tem um tamanho de 2048 bytes tal como foi definido nos parâmetros de geração das chaves esta chave não pode ser utilizada em algoritmos como AES e por isso é necessário reduzir o tamanho da chave para um tamanho fixo de 32 de bytes. Para isso utilizou-se o algoritmo de derivação de chaves **HKDF**, sendo que as chaves geradas com este algoritmo serão utilizadas como chave de autenticação e chave de cifragem.

[4]:
```
#FUNÇÃO: geração das chaves partilhadas entre as entidades
def generateSharedKey(pkg_msg,privateKey_cipher,privateKey_mac):

    #transforma os bytes recebidos em chave pública
    entity_publicKey_cipher = load_pem_public_key(pkg_msg['pk_cipher'])
    entity_publicKey_mac = load_pem_public_key(pkg_msg['pk_mac'])

    #geração da chave partilhada
    key_cipher = privateKey_cipher.exchange(entity_publicKey_cipher)
    key_mac = privateKey_mac.exchange(entity_publicKey_mac)

    #geração da chave partilhada de tamanho fixo de bytes
    sharedKey_cipher = HKDF( algorithm=hashes.SHA256(),length=32,
                        salt=None, info=b'handshake data',
                       ).derive(key_cipher)
    sharedKey_mac= HKDF( algorithm=hashes.SHA256(),length=32,
                        salt=None, info=b'handshake data',
                       ).derive(key_mac)
```

```
    return sharedKey_cipher,sharedKey_mac
```

Cifragem A mensagem que o emissor vai enviar ao recetor será cifrada através da função `encrypt` utilizando a cifra simétrica **AES** no modo **GCM** que utiliza a chave partilhada de cifragem como chave para a cifragem. O algoritmo simétrico AES (Advanced Encryption Standard) é um cifrador por blocos que não só é rápido como é criptograficamente forte, já o modo GCM é um modo que permite a proteção da mensagem contra ataques nonce. Estes ataques consistem em intercetar e copiar uma mensagem sendo que a sua cópia será enviada ao destino as vezes que o atacante quiser. Com a introdução de um valor pseudo-aleatório, nonce, gerado a partir de uma função HASH em modo XOF (SHAKE256) cada mensagem é identificada com um número único, mitigando a potencialidade destes ataques.

Após a geração do valor *nonce* e do cifrador (*cipher*) AES-GCM, autenticamos os metadados e ciframos e autenticamos o *ciphertext* através da função `generateMac`. Esta função recebe a chave partilhada de autenticação e cria um código de autenticação que irá autenticar a mensagem a ser enviada. Para que o recetor consiga autenticar a mensagem recebida, o emissor terá de enviar juntamente com a mensagem o código gerado pela função `generateMac`. Todos este processo é realizado com o uso de um algoritmo hash responsável por códigos de autenticação de mensagens, **HMAC**

[5]:
```
#FUNÇÃO: geração de código de autenticação
def generateMac(key, crypto):
    h = hmac.HMAC(key, hashes.SHA256(), backend = default_backend())
    #autentica e cria hash em função do parâmetro recebido
    h.update(crypto)
    return h.finalize()

#FUNÇÃO: processo de cifrar uma mensagem
def encrypt(plaintext, keyCipher, keyMac, ad):

    #geração de um valor pseudo-aletatório a ser usado como nonce
    digest = hashes.Hash(hashes.SHA256())
    nonce = digest.finalize()

    #criação de um Cipher AES-GCM (cifragem)
    encryptor = Cipher(algorithms.AES(keyCipher),
                       modes.GCM(nonce),
                       backend=default_backend()).encryptor()

    #autenticação de metadados
    encryptor.authenticate_additional_data(ad)

    #transformação da mensagem de plaintext para ciphertext
    ciphertext = encryptor.update(plaintext.encode()) + encryptor.finalize()

    # mensagem a ser enviada
```

```
    pkg = { 'nonce': nonce, 'tag': encryptor.tag, 'cipher': ciphertext }

    #gera um valor que autentica mensagem
    hmac = generateMac(keyMac, dumps(pkg))

    return {'message' : pkg, 'tag' : hmac}
```

Decifragem A mensagem que o recetor recebeu será decifrada por um processo inverso ao da cifragem. Começa-se por verificar a autenticidade da mensagem através da geração do código HMAC utilizando a chave partilhada de autenticação. Como o emissor tem a mesma chave de autenticação então o código obtido será o mesmo logo caso os códigos sejam diferentes então a mensagem não é autêntica. A seguir, utilizando o decifrador criado, transforma-se o *ciphertext* recebido em *plaintext*.

[6]:
```
#FUNÇÃO: processo de decifrar uma mensagem
def decrypt(ciphertext,keyCipher, keyMac, ad):

    #retirar da mensagem o ciphertext e código de autenticação
    text = ciphertext['message']
    hmac = ciphertext['tag']

    #verificar o código de autenticação
    macDest = generateMac(keyMac, dumps(text))
    if (hmac != macDest):
        return 'ERROR - MAC/Password is not equal'

    nonce = text['nonce']
    tag = text['tag']
    message = text['cipher']

    #criação de um Cipher AES-GCM (decifragem)
    decryptor = Cipher(algorithms.AES(keyCipher),
                       modes.GCM(nonce, tag),
                       backend=default_backend()).decryptor()

    #autentica os metadados
    decryptor.authenticate_additional_data(ad)

    #transforma ciphertext em plaintext
    plaintext = decryptor.update(message) + decryptor.finalize()

    return plaintext.decode()
```

EMITTER O emissor como é o primeiro a enviar mensagens para o recetor, começa por gerar os pares de chaves necessários para a criação das chaves partilhadas e assina a mensagem que

contém as suas chaves públicas e envia para o recetor. Após receber as chaves públicas do recetor, confirma a assinatura desta através da função *verify* do algoritmo DSA, utilizando a assinatura e chave pública da assinatura que recebeu juntamente com a mensagem. Caso a mensagem não seja autêntica então é lançada uma exceção *InvalidSignature*, caso contrário o emissor gera as chaves partilhadas utilizando a função `generateSharedKey`. A seguir, é utilizada a função `encrypt` que irá cifrar a mensagem *message from emitter to receiver*. Finalmente é gerado um novo código de autenticação com o uso da função `generateMac` que autentica a mensagem final a ser enviada ao recetor.

[7]:
```
#FUNÇÃO: funcionalidades do emitter
def Emitter(conn):

    #geração de pares de chaves e assinaturas
    pkg, privateKey_cipher, privateKey_mac = generateKeys()
    finalPkg = generateSignature(pkg)

    #envio da mensagem com chaves publicas
    print("E: Sending public keys to receiver...")
    conn.send(finalPkg)

    #receber mensagem com chaves publicas do emissor
    msg = conn.recv()
    print("E: Receiving public keys from receiver...")
    public_DSA = load_pem_public_key(msg['pub_key'])

    try:
        #verificar assinatura da mensagem recebida
        public_DSA.verify(msg['signature'],msg['message'],hashes.SHA256())
        print("E: The message is authentic.")

        #geração das chaves partilhadas
        pkg_msg = loads(msg['message'])
        sharedKey_cipher, sharedKey_mac =␣
↪generateSharedKey(pkg_msg,privateKey_cipher,privateKey_mac)

        #mensagem a ser enviada para o receiver
        text = "Message from emitter to receiver"
        print('Inicial message: ' + text)

        #geração dos metadados como valores pseudo-aleatórios
        associatedData = os.urandom(16)

        #geração do código de autenticação para a mensagem final
        hmac_key = generateMac(sharedKey_mac,sharedKey_mac)

        #cifrar a mensagem
        print("E: Encrypting message...")
```

```
        message = encrypt(text,sharedKey_cipher,sharedKey_mac,associatedData)

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

        #envia uma mensagem pelo seu lado do Pipe
        print("E: Sending ciphertext ...")
        conn.send(message)

    except InvalidSignature:
        print("E: The message is not authentic.")

    conn.close()
```

RECEIVER O recetor começa por gerar a seus pares de chaves e assinatura através das funções `generateKeys` e `generateSignature`. De seguida, espera por receber as chaves públicas vindas do emissor, verificando de seguida a assinatura da mensagem tal como aconteceu no emissor. O recetor envia para o emissor as suas chaves públicas e espera por mensagens vindas do emissor. Após receber uma mensagem cifrada, é verificada a autenticidade desta através da comparação dos códigos. Caso a mensagem seja autêntica então é decifrada usando a função `decrypt`, obtendo-se a mensagem enviada pelo o emissor.

[8]:
```
#FUNÇÃO: funcionalidades do receiver
def Receiver(conn):

    #geração de pares de chaves e assinaturas
    pkg, privateKey_cipher, privateKey_mac = generateKeys()
    finalPkg = generateSignature(pkg)

    #recebe a mensagem do seu lado do Pipe
    msg = conn.recv()
    print("R: Receiving public keys from emitter...")

    public_DSA = load_pem_public_key(msg['pub_key'])

    try:
        #verificar assinatura da mensagem recebida
        public_DSA.verify(msg['signature'],msg['message'],hashes.SHA256())
        print("R: The message is authentic.")

        #geração das chaves partilhadas
        pkg_msg = loads(msg['message'])
        sharedKey_cipher, sharedKey_mac =␣
↪generateSharedKey(pkg_msg,privateKey_cipher,privateKey_mac)

        #envio da mensagem com chaves publicas
```

```
        print("R: Sending public keys to emitter...")
        conn.send(finalPkg)

        #recebe a mensagem do seu lado do Pipe
        message = conn.recv()
        print('R: Cipher message received ')
        print(message)

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

        #verificar o código de autenticação
        if hmac_key == generateMac(sharedKey_mac,sharedKey_mac):

            #decifrar a mensagem
            print("R: Decrypting message...")
            final_message = decrypt(message, sharedKey_cipher, sharedKey_mac,␣
↪associatedData)

            print("Final Message: " + final_message)
        else:
            print('ERROR - Different keys used.')

    except InvalidSignature:
        print("R: The message is not authentic.")

    conn.close()
```

COMUNICAÇÃO A classe Pipe tem como objetivo criar um pipe entre o emissor e o recetor de forma a que estes consigam trocar mensagens entre eles. Para que ambas as entidades corram assíncronamente tanto o emissor como o recetor correm em processos independentes, contudo é o emissor o primeiro a iniciar a comunicação e enquanto umas das entidades espera pela resposta da outra o processo fica em espera até que receba informação.

[9]:
```
#CLASSE: cria o pipe onde ocorre a comunicação
class Conn(object):

    #FUNÇÃO: determina todos os parâmetros da comunicação
    def __init__(self, emitter, receiver, timeout=None):

        emitter_end, receiver_end = Pipe()
        self.timeout = timeout

        #os processos ligados ao Pipe
        self.eproc = Process(target=emitter, args=(emitter_end,))
        self.rproc = Process(target=receiver, args=(receiver_end,))
```

```
    #FUNÇÃO: corre os dois processos independente (em dois processos distintos)␣
↪
    def auto(self, proc=None):
        if proc == None:
            self.eproc.start()
            self.rproc.start()
            self.eproc.join(self.timeout)
            self.rproc.join(self.timeout)
        else: # corre só o processo passado como parâmetro
            proc.start(); proc.join()

Conn = Conn(Emitter, Receiver)
#inicia a comunicação
Conn.auto()
```

```
E: Sending public keys to receiver…
R: Receiving public keys from emitter…
R: The message is authentic.
R: Sending public keys to emitter…
E: Receiving public keys from receiver…
E: The message is authentic.
Inicial message: Message from emitter to receiver
E: Encrypting message…
E: Sending ciphertext …
R: Cipher message received
{'message': {'nonce': b"\xe3\xb0\xc4B\x98\xfc\x1c\x14\x9a\xfb\xf4\xc8\x99o\xb9$'
\xaeA\xe4d\x9b\x93L\xa4\x95\x99\x1bxR\xb8U", 'tag':
b'\x83\xa8N\xb27\xa92\xd3\xee\xd0\x16\xb1R\xb2\xfc\xb9', 'cipher': b'\xe8\xa7\xc
0p*\xff\x16\x0f\xf5E\x89\x88\xefj1\xd0\xd0^\xc3\xfc\x18\x8es\x0c\x9e2\xaa\xb4\x0
3{\xd3\xcb'}, 'tag': b'\xab\xae\xcc\xa9\xa2\xde\xcbM-{$.\xd3\xaa\x94I\x12+\xe8\x
c5\x04ym\xe98\xd4\xd5<Y\xcc\x950', 'hmac_key': b'\x17\xad\xd1T\xeb$\x13\xab\xfa\
xc6\x01\x99\x1f\xdf\x950\x8f\xa5\\,3Eu\x80\x96\t\x9c2\x8c\xdb\x94\x01',
'associated_data': b'\xb92\xda\xae\xa2>\xa8t\xb2k\xf5\x13\xeba\x9e\x8c'}
R: Decrypting message…
Final Message: Message from emitter to receiver
```

1.1.2 Cenários de Teste

[10]:
```
!python Problema1.py 'mensagem a ser enviada para o recetor'
```

```
E: Sending public keys to receiver…
R: Receiving public keys from emitter…
R: The message is authentic.
R: Sending public keys to emitter…
E: Receiving public keys from receiver…
The receiver message is authentic.
```

```
Inicial message: mensagem a ser enviada para o recetor
E: Encrypting message…
E: Sending ciphertext …
Cipher message:
{'message': {'nonce': b"\xe3\xb0\xc4B\x98\xfc\x1c\x14\x9a\xfb\xf4\xc8\x99o\xb9$'
\xaeA\xe4d\x9b\x93L\xa4\x95\x99\x1bxR\xb8U", 'tag':
b'p\x9e\xe2\xf0\xd9\r\x08\xbd\xf5\x17ll\xce\xfdd+', 'cipher': b'Yke?\xcf\xc6\\\x
1d\xcfv\x9c>\xaf\xf6\x1cD!<M\x8c\x01(r\xb6\x8by\x84\x85\xa4\xa4\xa4\x0e\xbb\xc8l
>\x1b'}, 'tag': b'\x9a\xc6\x11\xa9\xe4;\x84B\xf7\xacn\xc4\xe7\x10\x02\x19\x8c\x0
0\xdb\x9f\x07y,\xbfKI\xb1\xcb\xb3\x84\xd4D', 'hmac_key': b'\xcfZ\xe1(\x08C\x9bxo
Z)\xd3o\xd4\xf0\x1c\xde\x16\x93773e\xeb\xa3\xb9\x03\x87\x91\xcb\x0f\xe8',
'associated_data': b'\x83\x89\x93\rV+\xd7 \x88kY\t~\xd0\x8fi'}
R: Decrypting message…
MESSAGE: mensagem a ser enviada para o recetor
```

[12]:
```
!python Problema1.py 'Segunda mensagem a ser enviada para o recetor'
```

```
E: Sending public keys to receiver…
R: Receiving public keys from emitter…
R: The message is authentic.
R: Sending public keys to emitter…
E: Receiving public keys from receiver…
The receiver message is authentic.
Inicial message: Segunda mensagem a ser enviada para o recetor
E: Encrypting message…
E: Sending ciphertext …
Cipher message:
{'message': {'nonce': b"\xe3\xb0\xc4B\x98\xfc\x1c\x14\x9a\xfb\xf4\xc8\x99o\xb9$'
\xaeA\xe4d\x9b\x93L\xa4\x95\x99\x1bxR\xb8U", 'tag':
b'\xb8h}I\xd2g\xe7\xd4\xa7Mpk\xe3\xf05a', 'cipher': b'f\xe1i\xa0b\x9e\x94w\x8a\x
99\xe2\xa8\xef\x15C\xcf\x0f\xae_\x9a1l}RD\xda\xe9\xe7\xec\x17\x7fIu\xd9s\xc44N\x
ef\xb7\xa4?\xecb\xbe'}, 'tag': b'\xa80#\xb3xt\xab\x9e\xbe\n\x1bl\xd8tnI\xa5iMa\x
ac/\x1cB\xf5$\xd1\xb2\x01\xf6o;', 'hmac_key': b'\x85F\xff\xdd\x84I\xd4o\x17\x823
\x1a\x03\x99k\xffK\xddA\xdc\x95i\xa2\x0fD%\x9b\tc\x1cf+', 'associated_data':
b'3\xaf\xfe\x02\xac\xb2yY\x06ps\xa3\x83\xc4\x9dg'}
R: Decrypting message…
MESSAGE: Segunda mensagem a ser enviada para o recetor
```