TP1-Problema1

March 28, 2022

1 TRABALHO PRÁTICO 1 - 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 uma chave pública do emissor para o recetor e vice versa. Cada entidade irá gerar um chave partilhada a ser utilizada no processo de cifra. A comunicação destas chaves deve manter a autenticidade e integridade através do uso de assinaturas digitais (EDSA). Após a obtenção da chave partilhada, o emissor irá enviar mensagens ao recetor que serão cifradas utilizando um AEAD com "Tweakable Block Ciphers". 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, sys
from multiprocessing import Process, Pipe
from pickle import dumps, loads
from cryptography.hazmat.primitives import hashes, hmac, serialization
from cryptography.hazmat.primitives.asymmetric import ed448, x448
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

numberOfBytes = 16
```

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 a chave usada no processo de 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. Para implementar este protocol foi utilizado a curva eliptica X448, usada no protocol DH. Esta curva permite a duas entidades concordarem em parâmetros iguais para gerarem uma chave partilhada num canal inseguro.

O desenvolvimento deste protocolo iniciou-se com a geração da chave privada e pública utilizando

a curva X448. A chave pública deste par será enviada 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 **x448** do *package Cryptography*.

[2]:
```
#FUNÇÃO: Geração do par de chaves
def generateKeys():

    #criação das chaves utilizando o x448 key exchange
    privateKey_cipher = x448.X448PrivateKey.generate()
    publicKey_cipher = privateKey_cipher.public_key()

    #mensagem com a chave publica a ser enviada a outra entidade
    package = { 'pk_cipher': publicKey_cipher.
 ↪public_bytes(encoding=serialization.Encoding.PEM,
                                                        format=serialization.
 ↪PublicFormat.SubjectPublicKeyInfo)}

    return dumps(package), privateKey_cipher
```

Geração da assinatura digital De forma a manter a autenticidade, integridade e não-repúdio na transmissão da chave pública entre as entidades, implementou-se o algoritmo EDSA. 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 **ED448** da *package Cryptography*. O ED448 consite numa curva elíptica que utiliza o algoritmo EDSA e irá gerar o par de chaves e a assinatura, de forma a que a entidade consiga confirmar a autenticidade.Para isso é necessário enviar à outra entidade, juntamente com a mensagem, a assinatura e a chave pública gerada.

[3]:
```
#FUNÇÃO: Geração da assinatura
# Recebe a mensagem a ser assinada
def generateSignature(pkg):

    #geração da chave a ser utilizada para assinatura
    privateKey_ED = ed448.Ed448PrivateKey.generate()

    #assina pacote a ser enviado
    signature = privateKey_ED.sign(pkg)

    #mensagem a ser envidada com a assinatura e chave publica da assinatura
    finalPackage = {'message': pkg, 'signature': signature,
                    'pub_key':privateKey_ED.public_key().
 ↪public_bytes(encoding=serialization.Encoding.PEM,
```

```
↪format=serialization.PublicFormat.SubjectPublicKeyInfo)}
    return finalPackage
```

Geração das chaves partilhadas Após receber a mensagem com a chave pública, cada entidade terá de gerar a chave partilhada que será usada no processo de autenticação e cifragem. A partir da curva eliptica x448 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 56 bytes tal como é definido nos parâmetros da curva esta chave não pode ser utilizada no algoritmo como AES e por isso é necessário reduzir o tamanho da chave para um tamanho fixo de 16 bytes. Para isso utilizou-se o algoritmo de derivação de chaves **HKDF**, sendo que a chave gerada com este algoritmo será utilizadas como chave de cifragem.

[4]:
```
#FUNÇÃO: Geração da chave partilhada a ser utilizada no processo de cifra
# Recebe a mensagem e a chave privada da entidade
def generateSharedKey(pkg_msg,privateKey_cipher):

    #retira a chave publica da mensagem que recebeu
    entity_publicKey_cipher = load_pem_public_key(pkg_msg['pk_cipher'])

    #atraves da chave publica da outra entidade e da sua chave privada gera a␣
↪chave partilhada
    key_cipher = privateKey_cipher.exchange(entity_publicKey_cipher)

    #deriva um chave de 16 bytes a ser utilizada no processo de cifra
    sharedKey_cipher = HKDF( algorithm=hashes.SHA256(),length=16,
                        salt=None, info=b'handshake data',
                       ).derive(key_cipher)

    return sharedKey_cipher
```

Criação dos blocos da mensagem Como forma de preparar a mensagem a ser utilizada na cifra AEAD, é necessário que a mensagem seja divida em blocos de 16 bytes (*numberBytes*), tal como etsá indicado a cima. Quando o tamanho da mensagem não é múltipla de 16 então o último bloco vai ter tamanho r, com r $<$ 16, sendo por isso necessário que seja acrescentado ao bloco um conjunto de bits a 0 (*padding*) de tamanho n, com n = 16 - r. Desta forma, o último bloco vai ter um tamanho igual aos restantes blocos. Esta função vai não só devolver os blocos da mensagem, como o tamanho do último sem o padding (r) que será utilizado no processo de cifra.

[5]:
```
#FUNÇÃO: Divisão da mensagem por bocos de tamanho numberOfBytes
# Recebe o plaintext a ser dividido em blocos
def createBlockMessage(plaintext):

    #Divide a mensagem em blocos de 16 bytes
    blockMessage = [plaintext[i:i+numberOfBytes] for i in␣
↪range(0,len(plaintext),numberOfBytes)]
```

```
    m = len(blockMessage)-1
    message =[]

    #transforma os blocos em byte string
    for x in range(0,m+1):
        message.insert(x, blockMessage[x].encode('utf-8'))

    r = len(message[m])

    #verifica se o último bloco tem a necessidade de padding
    if r < numberOfBytes:

        #acrescenta um padding de 0 ao ultimo bloco
        message[m] = message[m].ljust(numberOfBytes,b'\0')

    return r, message
```

Geração dos tweaks A cifra AEAD com "Tweakable Block Ciphers" utiliza um input adicional designado de *tweak*. Estes tweaks tem uma função semelhante à chave de cifra, contudo, enquanto a chave é sempre a mesma para cada bloco, o *tweak* é distinto para cifra de cada bloco. Assim, a utilização dos tweaks torna a cifra menos vulnerável a ataques. A função `generateTweak` tem como objetivo gerar 2 tipos de *tweaks* com tamanho igual ao dos blocos da mensagem (16 bytes, *numberOfBytes*), um tipo a ser usado na cifra dos blocos e outro para a autenticação do *ciphertext*.

Os tweaks de cifra são gerados utilizando um valor nonce que ocupa metade do tamanho do tweak (8 bytes) e um contador que é incrementado de bloco em bloco que ocupa os restantes 8 bytes. Como se trata de tweak de cifra, então este deve terminar com um bit a 0, assim, o tweak final será composto por: `w = [nonce|i|0], com i = 0...m-1, m = número de blocos`.

O tweaks de autenticação é gerado utiliza um também valor nonce que ocupa metade do tamanho do tweak (8 bytes) e o comprimento da mensagem sem o padding que ocupa os restantes 8 bytes. Como se trata de tweak de autentiação, então este deve terminar com um bit a 1, assim, o tweak final será composto por: `w* = [nonce|length|1], com length = comprimento do plaintext`.

Nota: É de notar que o *nonce* utilizado para gerar todos os *tweaks* é sempre os mesmo e que a função `modifyBit` tem como objetivo alterar o último bit do *tweak* de autenticação para 1.

[6]:
```
#FUNÇÃO: Adiciona o último bit do tweak de autenticação com valor 1
def modifyBit( n,  p,  b):
    mask = 1 << p
    return (n & ~mask) | ((b << p) & mask)

#FUNÇÃO: Geração dos tweaks
# Recebe o numero de blocos, comprimento do plaintext e o nonce
def generateTweak(numberBlocks, length, nonce):

    tweakBlock = []
```

```
    #Gera os tweaks para cifrar os blocos da mensagem
    for i in range(0,numberBlocks):

        #Adiciona o nonce e o respetivo contador ao tweak
        tweak = nonce + int(i).to_bytes(numberOfBytes // 2, byteorder='big')

        tweak = int.from_bytes(tweak, byteorder='big')
        #Remove o último bit
        tweak = tweak >> 1
        #Adiciona um bit a 0 na última posição do tweak - [N||i||0]
        tweak = tweak << 1
        tweak = tweak.to_bytes(numberOfBytes, byteorder='big')

        #Insere o tweak gerado no array dos tweaks
        tweakBlock.insert(i,tweak)

    #Gera o tweak de autenticação utilizando o nonce e o comprimento da␣
↪mensagem a ser cifrada
    tweakAutenticate = nonce + length.to_bytes(numberOfBytes // 2,␣
↪byteorder='big')

    tweakAutenticate = int.from_bytes(tweakAutenticate, byteorder='big')
    #Adiciona um bit a 1 na última posição do tweak - [N||b||1]
    tweakAutenticate = modifyBit(tweakAutenticate, 0, 1)
    tweakAutenticate = tweakAutenticate.to_bytes(numberOfBytes, byteorder='big')

    return tweakBlock, tweakAutenticate
```

Cifragem O processo de cifra é realizado em 3 partes distintas. Após a geração dos tweaks e dos blocos da mensagem, começamos a cifrar todos os blocos exceto o último bloco (parte 1). De seguida ciframos o último bloco (parte 2) e finalmente geramos a tag de autenticação associada ao *ciphertext* (parte 3). O processo de cifra a ser aplicado aos blocos é constituido por `Ẽ(w,k,x) = E(k, w  E(k,x))` que foi implementado utilizando a função `generateCiphertext`. Esta cifra é composta por uma operação de XOR implementada pela função `xor` juntamente com a cifra AES no modo de CTR, que utiliza um valor de nonce para aumentar a segurança da cifra.

PARTE 1: Nesta parte percorremos m-1 blocos da mensagem (m = número de blocos da mensagem) de modo a cifrá-los um a um utilizando a cifra da função `generateCiphertext`, o *tweak* associado a esse bloco e a chave partilhada gerada. O output desta parte será um conjunto de m-1 blocos do *ciphertext*, com comprimento igual ao tamanho dos blocos (16 bytes).

PARTE 2: Nesta segunda parte vamos cifrar o último bloco da mensagem. Para isso, começamos por cifrar o comprimento do bloco sem o *padding* utilizando a cifra $\tilde{E}$. Este processo devolve um *ciphertext* de comprimento de 16 bytes que vai ser utilizado na operação de `xor` juntamente com o último bloco. No final desta fase, o *output* devolvido vai corresponder ao *ciphertext* do último bloco.

PARTE 3: Na última fase determinamos a tag de autenticação que terá o comprimento igual ao

do último bloco sem o *padding* (comprimento r). Para gerar a tag começamos por gerar o *checksum* que corresponde à operação de xor entre os blocos de mensagem incluíndo o último bloco com o *padding*, isto é, `M1 ... Mm-1 [Mm\0*]`. Utilizando o checksum, o tweak de autenticação e a chave partilhada na cifra $\tilde{E}$, obtemos um output onde os r primeiros valores correspondem à tag de autenticação da cifra.

Para o recetor será enviado, o *ciphertext*, a tag, o valor de *nonce* que gerou os *tweaks* para que este consiga fazer o mesmo e o valor de *nonce* utilizado pela cifra AES no modo CTR. Para permitir que o recetor consiga confirmar a autenticação dos dados associados à mensagem é gerado um valor de hash apartir da função `generateMAC`.

[7]:
```
#FUNÇÃO: Gera uma hash para autenticação da mensagem cifrada
def generateMac(key, crypto):
    h = hmac.HMAC(key, hashes.SHA256(), backend = default_backend())
    h.update(crypto)
    return h.finalize()

#FUNÇÃO: Operação de XOR
def xor(blockL, blockR):

    return [(a^b).to_bytes(1,byteorder='big') for (a,b) in zip(blockL, blockR)]

#FUNÇÃO: Implementação da cifra a ser utilizada pelo Tweakable Block Ciphers
# Recebe a chave, o nonce, tweak, e a mensagem a cifrar
def generateCiphertext(key, nonce, tweak, message):

    #Criação da cifra AES
    encryptor = Cipher(algorithms.AES(key), modes.CTR(nonce)).encryptor()

    #Cifra a mensagem utilizando a chave
    normalCipher = encryptor.update(message)

    #Operação de xor entreo tweak e o output da cifra
    xorCipher = b"".join(xor(tweak,normalCipher))

    #Retorna a cifra da operação xor
    return encryptor.update(xorCipher) + encryptor.finalize()

#FUNÇÃO: Cifra o plaintext
# Recebe o plaintext e a chave partilhada
def encrypt(plaintext, keyCipher):

    #Divisão da mensagem em blocos
    r, message = createBlockMessage(plaintext)
    length = len(plaintext)
    numberBlocks = len(message)

    #Geração dos tweaks
```

```
    nonceTweak = os.urandom(numberOfBytes//2)
    tweakBlock, tweakAutenticate = generateTweak(numberBlocks,length,nonceTweak)

    m = numberBlocks-1
    cipherBlock = []

    nonce = os.urandom(16)

    #Parte 1: Cifra os blocs de 0 a m-1, com m = número de blocos - 1
    for w in range(0,m) :

        ct = generateCiphertext(keyCipher, nonce, tweakBlock[w], message[w])
        cipherBlock.append(ct)

    #Parte 2: Cifra o último bloco - bloco m
    lastBlock = int(r).to_bytes(numberOfBytes, byteorder='big')
    ctLastBlock = generateCiphertext(keyCipher, nonce, tweakBlock[m], lastBlock)

    cipherLast = b"".join(xor(ctLastBlock, message[m]))
    cipherBlock.append(cipherLast)

    #Parte 3: Autenticação da mensagem final
    checksum = message[0]

    #Formação do checksum
    for i in range(1,m+1):
        z2 = message[i]
        checksum = b"".join(xor(checksum,z2))

    tag = generateCiphertext(keyCipher, nonce, tweakAutenticate, checksum)[:r]

    #Junta todos os blocos cifrados
    ciphertext = b"".join(cipherBlock)

    #Mensagem a ser enviadad para o recetor
    pkg = {'ciphertext' : ciphertext, 'tag' : tag, 'nonce': nonce, 'nonceTweak':
↪ nonceTweak}

    #Autentica a mensagem a ser enviada utilizando a função de hash
    hmac = generateMac(keyCipher, ciphertext)
    send = {'pkg' : pkg, 'hmac' : hmac}

    return send
```

Decifragem O processo de decifra é realizado em 2 partes distintas. Após a geração dos tweaks e dos blocos da mensagem utilizando os valores que o rector recebeu na mensagem, começamos a decifrar todos os blocos (parte 1). De seguida comparamos a tag de autenticação da mensagem da

gerada pelo o processo de decifra (parte 2). O processo de decifra a ser aplicado aos blocos é dado pela função `generateCiphertext`, tal como aconteceu no processo de cifra.

PARTE 1: Nesta parte percorremos os m blocos do *ciphertex* (m = número de blocos da mensagem) de modo a decifrá-los um a um utilizando a cifra da função `generateCiphertext`, o *tweak* associado a esse bloco e a chave partilhada gerada. O output desta parte será um conjunto de m blocos do *plaintext*, com comprimento igual ao tamanho dos blocos (16 bytes). É de notar que neste processo de decifra, o último bloco gerado conterá o *padding* que foi implementado no processo de cifra.

PARTE 3: Na última fase determinamos a tag de autenticação do mesmo modo realizado para o processo de cifra. Para gerar a tag começamos por gerar o *checksum* que corresponde à operação de xor entre os blocos gerados do *plaintext* incluíndo o último bloco com o *padding* e utilizamos a cifra Ẽ para obtermos um output onde os r primeiros valores correspondem à tag de autenticação da cifra. De seguida, comparamos a tag gerada com a tag que recebemos do emissor, caso for igual então podemos admitir que a mesagem é autentica caso não for igual então podemos excluir a mensagem.

[8]:
```
#FUNÇÃO: Decifra a mensagem recebida
# Recebe a mensagem cifrada e a chave partilhada
def decrypt(cipherPkg,keyCipher):

    #Retirada mensagem os valores que recebeu
    ciphertext = cipherPkg['ciphertext']
    tag = cipherPkg['tag']
    nonce = cipherPkg['nonce']
    nonceTweak = cipherPkg['nonceTweak']

    #Divide a mensagem cifrada em blocos de 16 bytes
    message = [ciphertext[i:i+numberOfBytes] for i in␣
↪range(0,len(ciphertext),numberOfBytes)]

    numberBlocks = len(message)
    m = numberBlocks-1
    n = len(ciphertext)
    r = len(tag)
    length = n - (numberOfBytes - r)

    #Cria os tweaks utilizando o nonce recbeido na mensagem cifrada
    tweakBlock, tweakAutenticate = generateTweak(numberBlocks,length,nonceTweak)

    plainBlock = []

    #PARTE 1: Decifra as mensagens de 0 a m , com m = numero de blocos
    for w in range(0,m+1) :

        ct = generateCiphertext(keyCipher, nonce, tweakBlock[w], message[w])
        plainBlock.append(ct)
```

```
    #PARTE 2: Autenticação da mensagem

    #Gera o checksum para autenticar a mensagem
    checksum = plainBlock[0]
    for i in range(1,m+1):
        z2 = plainBlock[i]
        checksum = b"".join(xor(checksum,z2))

    #Gera tag de autenticação apartir do checksum
    generatedTag = generateCiphertext(keyCipher, nonce, tweakAutenticate,␣
↪checksum)[:r]

    #Verifica se a mensagem está autenticada
    if tag == generatedTag:

        plainBlock[m] = plainBlock[m][:r]

        #Junta todas as mensagem decifradas
        plaintext = b"".join(plainBlock)

    else :
        return "R: Message not autenticated"


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

EMITTER O emissor como é o primeiro a enviar mensagens para o recetor, começa por gerar o par de chaves necessária para a criação da chave partilhada, assina a mensagem que contém a sua chaves pública e envia a mensagem para o recetor. Após receber a chave pública do recetor, confirma a assinatura desta através da função *verify* do algoritmo EDSA, 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 a chave partilhada utilizando a função `generateSharedKey`. De seguida é verificado se a chave partilhada gerada corresponde à chave partilhada que possui a outra entidade, neste caso o recetor. Para isso geramos um hash apartir da chave partilhada utilizando a função `generateMAC`, sendo esse valor enviado ao recetor. O recetor também lhe envia um valor de hash para que este também consiga validar a chave partilhada gerada. Para isso ele confirma se hash que o ele gerou apartir da sua chave partilhada é igual à hash que recebeu do recetor. Finalmente, é utilizada a função `encrypt` que irá cifrar uma mensagem, enviado o output da função ao recetor.

[12]:
```
#FUNÇÃO: Implementa o emissor
def Emitter(conn):

    #Gera o par de chaves
    pkg, privateKey_cipher = generateKeys()
    #Gera a assinatura
```

```
    finalPkg = generateSignature(pkg)

    print("E: Sending public keys to receiver...")
    conn.send(finalPkg)

    msg = conn.recv()
    print("E: Receiving public keys from receiver...")

    #Retira a chave publica da assinatura da mensagem que recebeu
    public_ED = load_pem_public_key(msg['pub_key'])

    try:
        #Verifica a assinatura da mensagem
        public_ED.verify(msg['signature'],msg['message'])
        print("E: The message is authentic.")

        #Gera a chave partilhada
        pkg_msg = loads(msg['message'])
        sharedKey_cipher = generateSharedKey(pkg_msg,privateKey_cipher)

        #Gera hash de autenticação para confirmar a chave partilhada
        hmac_key = generateMac(sharedKey_cipher,sharedKey_cipher)
        confirmKey = {'hmac': hmac_key}

        print("E: Sending confirmation for shared key ...")
        conn.send(dumps(confirmKey))

        hmacMessage = conn.recv()
        print("E: Receiving confirmation for shared key...")

        hmacReceiver = loads(hmacMessage)['hmac']

        #Confirma autenticação da achave partilhada
        if hmacReceiver == generateMac(sharedKey_cipher,sharedKey_cipher):

            print("E: Shared key is equal")

            if len(sys.argv) == 0 :
                text = "Before he moved to the inner city, he had always␣
↪believed that security complexes were psychological."
            else:
                text = sys.argv[1]

            print('Inicial message: ' + text)

            print("E: Encrypting message...")
            message = encrypt(text,sharedKey_cipher)
```

```
            #associatedData = os.urandom(16)
            #adMac = generateMac(sharedKey_cipher, associatedData)
            #message['ad'] = adMac

            print('Cipher message:')
            print(message)

            print("E: Sending ciphertext ...")
            conn.send(message)

        else:
            print('E: The shared keys are different.')

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

    conn.close()
```

RECEIVER O recetor começa por gerar o seu par de chaves e assinatura através das funções `generateKeys` e `generateSignature`. De seguida, espera por receber a chave pública do emissor, verificando de seguida a assinatura da mensagem tal como aconteceu no emissor. O recetor envia para o emissor a sua chave pública e espera pelo o hash de confirmação da chave partilhada vinda do emissor. Após receber essa mensagem, confirma se a chave partilhada é igual através da igualdade entre os hash que gerou e aquele que recebeu do emissor. De seguida, envia para o emissor a hash associada à chave partilhada para que este consiga tmabém confirmar a sua chave partilhada. De seguida o receetor recebe uma mensagem cifrada vinda do emissor, e verifica a autenticidade desta através da comparação dos valores de hash. Caso a mensagem seja autêntica então é decifrada usando a função `decrypt`, obtendo-se a mensagem enviada pelo o emissor.

[13]:
```
#FUNÇÃO: Implementação do recetor
def Receiver(conn):

    #Geração do par de chaves
    pkg, privateKey_cipher = generateKeys()
    #Geração de assinatura
    finalPkg = generateSignature(pkg)

    msg = conn.recv()
    print("R: Receiving public keys from emitter...")

    public_ED = load_pem_public_key(msg['pub_key'])

    try:
        #Verifica a assinatura da mensagem
        public_ED.verify(msg['signature'],msg['message'])
        print("R: The message is authentic.")
```

```
        #Gera a chave partilhada
        pkg_msg = loads(msg['message'])
        sharedKey_cipher = generateSharedKey(pkg_msg,privateKey_cipher)

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

        hmacMessage = conn.recv()
        print("R: Receiving confirmation for shared key...")

        hmacEmitter = loads(hmacMessage)['hmac']

        #Confirma se a autenticação da chave partilhada
        if hmacEmitter == generateMac(sharedKey_cipher,sharedKey_cipher):

            print("R: Shared key is equal")

            #Gera hash de autenticação para confirmar a chave partilhada
            hmac_key = generateMac(sharedKey_cipher,sharedKey_cipher)
            confirmKey = {'hmac': hmac_key}

            print("R: Sending confirmation for shared key ...")
            conn.send(dumps(confirmKey))

            message = conn.recv()

            ciphertext =  message['pkg']
            hmac =  message['hmac']
            #associatedData = message['associated_data']

            #Confirma autenticação da mensagem
            if hmac == generateMac(sharedKey_cipher,ciphertext['ciphertext']):

                print("R: Decrypting message...")
                final_message = decrypt(ciphertext, sharedKey_cipher)

                print('Final Message: ' + final_message)
            else :
                print("R: Message not autenticated")

        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.

[14]:
```
#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.
E: Sending confirmation for shared key …
R: Receiving confirmation for shared key…
R: Shared key is equal
R: Sending confirmation for shared key …
E: Receiving confirmation for shared key…
E: Shared key is equal
```

```
Inicial message: -f
E: Encrypting message…
Cipher message:
{'pkg': {'ciphertext': b'\xd6\x19\x89X\x0fc4y\x19N\x08\xeb\xeaDGr', 'tag':
b'\xd6\x19', 'nonce': b'N\xdb-\x8e\x9c\xbd(\x17{]\x9c\x9a8+\x98o', 'nonceTweak':
b'\xb4\x01 \xd6\xa1Q\x9a\x11'}, 'hmac': b'\x8d\x8dP9\xd9\x8btu|\xb2\xe4\xca\x0bc
\xce\xae\x861C\xb3\xc2\x10\x1f\\\xf3\xe9\xf5\t8M\x9a\xeb'}
E: Sending ciphertext …
R: Decrypting message…
Final Message: -f
```

1.1.2 Cenários de Teste

[15]:
```
!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…
E: The message is authentic.
E: Sending confirmation for shared key …
R: Receiving confirmation for shared key…
R: Shared key is equal
R: Sending confirmation for shared key …
E: Receiving confirmation for shared key…
E: Shared key is equal
Inicial message: mensagem a ser enviada para o recetor
E: Encrypting message…
Cipher message:
{'pkg': {'ciphertext': b'\x8a\xc7\x82\xd8Y\xb1\xe1\x18\x88\x8e\xeaP\xf5\x01\xd3r
\x89\xd4\x85\xca\\\xb7\xa4\x05\xc9\x9d\xab\x03\xffS\x81r\x84\xc7\x98\xc4J\xd6\x8
4u\xa8\xef\xca#\x90s\xf3\x10', 'tag': b'\x87\xd4\x9f\xd6O', 'nonce':
b'L\x9b\xd6V\xc2\x84\xf9\x1fN\x02\xa3c\nj\xc1\xea', 'nonceTweak':
b'\xa4\x01:0\x9ff\x94Z'}, 'hmac': b'{\xa6\rv\xec\x0c\x17\x8e\x1f\x1eY\x00@\x8d\x
f6\xa4\x9e\xcf\xf3:\r9&\x9a\xf7\xa4H\xf5\x01\xba\x89\xff'}
E: Sending ciphertext …
R: Decrypting message…
Final Message: mensagem a ser enviada para o recetor
```

[16]:
```
!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…
E: The message is authentic.
E: Sending confirmation for shared key …
```

```
R: Receiving confirmation for shared key…
R: Shared key is equal
R: Sending confirmation for shared key …
E: Receiving confirmation for shared key…
E: Shared key is equal
Inicial message: Segunda mensagem a ser enviada para o recetor
E: Encrypting message…
Cipher message:
{'pkg': {'ciphertext': b'\x90aMN\x02zQ\xaap\xbf\xf3\x86\x11\x05M\xb3\xe3e\nH\tl\
x10\xefs\xac\xf4\x94\x14\x03\x08\xae\xa2vK\x1b\x03>B\xef~\xbf\xe9\x9a\x02b(\xd1'
, 'tag': b'\xd1r\x0c\x1d\x08(\x03\xaa}\xac\xee\x88\x07', 'nonce':
b'\x0e\x11\x85K\x106\xcc3\xbcx/$.\xb6\xdc\xfa', 'nonceTweak':
b'\x95\xa8!\xbd+\t\xce\xfe'}, 'hmac': b'\xceI\x86X\x16\xd6\x97\xec@\x87\xad\r\xa
8\x13OT\x02\xefz\xc1\xb9\x16\xca;\x9fS\xfb\xe9\x08i\xdfj'}
E: Sending ciphertext …
R: Decrypting message…
Final Message: Segunda mensagem a ser enviada para o recetor
```

[ ]: