

ex2Report

February 25, 2020

1 TP0 - 2) Hello World para Cryptography

Neste exercício temos como objetivo implementar uma comunicação privada assíncrona entre um agente Emitter e um agente Receiver. Esta comunicação tem que ter algumas características como:

- * Autenticação do criptograma e dos metadados (associated data). Usar uma cifra simétrica num dos modos stream cipher (e.g. GCM).
- * Derivação da chave a partir de uma password usando um KDF; ambos os agentes devem ler essa password para poder gerar a chave
- * Autenticação prévia da chave usando um MAC.

1.1 Solução

A nossa solução para o problema em causa passou por criar dois agentes (*Receiver* e *Emitter*). Estes dois agentes irão comunicar através de *Sockets*, sendo que o Receiver terá o papel de “servidor” e o Emitter o papel de “cliente”. Eles começarão todo o processo quando o Emitter se conectar com o Receiver. Após essa conexão será pedida uma password a ambos que terá de ser igual para o processo de encriptação/descriptação correr sem problemas. Depois da introdução da password, o Emitter poderá enviar qualquer mensagem que o Receiver irá descriptar e mostrar a respetiva mensagem pelo terminal.

Neste momento do relatório vamos demonstrar a nossa solução. Primeiro os **imports** necessários

```
[ ]: from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.exceptions import InvalidSignature
import getpass
import socket
import os
```

Agora temos de ter em conta que alguns métodos serão comuns entre os dois agentes por isso serão implementadas fora das classes:

```
[ ]: '''
Key Derivation Function
Esta função receberá um salt e depois de introduzir as passwords irá ser_
↳ usada para derivar essa mesma password
numa key para a cifra
```

```

'''
def passKDF(salt):
    kdf = PBKDF2HMAC(
        algorithm = hashes.SHA256(),
        length = 32,
        salt = salt,
        iterations =100000,
        backend = default_backend())
    return kdf
'''

Pedir password
Esta função serve para pedir a password para ser derivada
'''
def askPassword():
    password = getpass.getpass() #Pede a password como input
    return str.encode(password)

'''

Message Authentication Code
Esta função recebe apenas a chave e um texto e calcula o MAC , devolvendo
→a tag.
'''
def mac(key,source, tag=None):
    h = hmac.HMAC(key,hashes.SHA256(),default_backend())
    h.update(source)
    if tag == None:
        return h.finalize()
    h.verify(tag)

'''

Função de Hash
Esta função recebe uma string e calcula a sua hash
'''
def Hash(s):
    digest = hashes.Hash(hashes.SHA256(),backend=default_backend())
    digest.update(s)
    return digest.finalize()

```

1.1.1 Receiver

Agora temos de falar do agente **Receiver** , que, tal como já foi referido, atua como um servidor:

```

[ ]: max_msg_size = 9999
class Receiver:
    """ Classe que implementa a funcionalidade de um Receiver. """
    def __init__(self):

```

```

        """ Construtor da classe. """
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.orig = ('', 5000)

def startSocket(self):
    """
    Função que coloca o servidor à escuta
    """
    self.socket.bind(self.orig)
    self.socket.listen(1)
    print("\n\n Servidor à escuta na porta: 5000 \n\n")

def decryption(self, key, iv, tag, msg):
    """
    Função que recebe: CHAVE, IV, TAG, MENSAGEM e que descripta devolvendo
    → o TEXTO LIMPO
    """
    cipher = Cipher(algorithms.AES(key), modes.GCM(iv, tag), default_backend())
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(msg) + decryptor.finalize()
    return plaintext

def run(self):
    """
    Função principal
    """
    print('Receiver RUN')
    self.startSocket()
    i = 0
    while(i < 1):
        con, cliente = self.socket.accept()
        print("Conectado por: ", cliente)
        i+=1
        password = askPassword()

        while True:
            msg = con.recv(max_msg_size)
            if (not msg):
                break
            else:
                salt = msg[0:16]
                tagK = msg[16:48]
                iv = msg[48:64]
                tag = msg[64:80]
                ciphertext = msg[80:]

                key = passKDF(salt).derive(password)

```

```

        try:
            plaintext = self.decription(key,iv,tag,ciphertext)
            if (tagK == mac(Hash(key),plaintext)):
                print(plaintext.decode())
            else:
                print('Mensagem comprometida. Chave não autenticada')
        except:
            print('Mensagem comprometida\n')
            self.breakCon(con)
        self.breakCon(con)
    return

def breakCon(self,con):
    con.close()
    print("\n\nConexão fechada\n\n")
    return

def mainReceiver():
    #erro
    receiver = Receiver()
    receiver.run()
    return

mainReceiver()

```

1.1.2 Emitter

Quanto ao agente Emitter este irá atuar como cliente conectando-se com o servidor e enviando mensagens ao mesmo.

```

[ ]: class Emitter:
    """ Classe que implementa a funcionalidade de um Emitter. """
    def __init__(self):
        """ Construtor da classe. """
        self.socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.dest = ('127.0.0.1', 5000)

    def encryption(self,key,msg):
        """
        Função que recebe: CHAVE, MENSAGEM e que a encripta devolvendo o
        ↪ CIPHERTEXT
        """
        iv = os.urandom(16)
        cipher = Cipher(algorithms.AES(key), modes.GCM(iv), default_backend())
        encryptor = cipher.encryptor()

```

```

    encrypted_text = encryptor.update(msg) + encryptor.finalize()
    return iv, encrypted_text, encryptor.tag

def askMessage(self):
    """
    Função que pede mensagem
    """
    print('Input message to send (empty to finish)') #pede a mensagem
    return input().encode()

def server_connection(self):
    """
    Função que faz o cliente conectar-se ao Servidor
    """
    self.socket.connect(self.dest)
    print("\n\n Cliente conectado na porta: 5000 \n\n")

def endConnection(self):
    """
    Função de desconexão ao servidor
    """
    self.socket.close()
    print('\n Conexão terminada. \n')

def run(self):
    """
    Função principal
    """
    self.server_connection() #Conecta-se ao Servidor
    password = askPassword()
    salt = os.urandom(16)
    key = passKDF(salt).derive(password)

    while True:
        msg = self.askMessage()

        try:

            tagPT = mac(Hash(key), msg)

            iv, ciphertext, tagCT = self.encription(key, msg)

            new_msg = salt + tagPT + iv + tagCT + ciphertext
            if(len(new_msg)>0):
                self.socket.send(new_msg)
            else:
                self.endConnection()

```

```
        except:
            print("Erro no Emissor")

def mainEmitter():
    #erro
    emitter = Emitter()
    emitter.run()

mainEmitter()
```

1.1.3 Nota

De realçar que esta implementação funcionará com dois terminais abertos. Aqui no Jupyter Notebook tivemos alguns problemas em implementar isto com Sockets e poderíamos tê-lo feito com Pipes.