Tp0e2

February 26, 2020

Exercício 2 - Comunicação privada assíncrona

Nesta fase do trabalho criou-se uma comunicação privada assíncrona entre um agente *Emitter* e um agente *Receiver*.

0.0.1 Definições

`class BiConn` A classe BiConn serve para a criação dos dois processos necessários que permite a comunicação entre os dois agentes, mais concretamente a função "Pipe" . Recebemos como argumentos os processos que irão estar relacionados em que um dos lados envia a mensagem pretendida e fecha a ligação, e do outro lado é recebida a mensagem, e a ligação é fechada também.

[5]:
```
from multiprocessing import Process, Pipe
class BiConn(object):
    def __init__(self,left,right,timeout=None):
        """
        left : a função que vai ligar ao lado esquerdo do Pipe
        right: a função que vai ligar ao outro lado
        timeout: (opcional) numero de segundos que aguarda pela terminação do␣
↪processo
        """
        left_end, right_end = Pipe()
        self.timeout=timeout
        self.lproc = Process(target=left, args=(left_end,))       # os␣
↪processos ligados ao Pipe
        self.rproc = Process(target=right, args=(right_end,))
        self.left  = lambda : left(left_end)                       # as funções␣
↪ligadas já ao Pipe
        self.right = lambda : right(right_end)

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

```
    def manual(self):   #  corre as duas funções no contexto de um mesmo␣
↪processo Python
        self.left()
        self.right()
```

PBKDF As Password Based Key Derivation Functions têm dois principais objetivos: - Permite a utilização de passwords de modo a que sejam memorizáveis pelos humanos. - Resistência a ataques por força bruta, uma vez que introduzem complexidade computacional artificial, aumentando o tempo necessário para cada tentativa.

Tanto o Emitter como o Receiver devem conhecer a *password* para gerar a chave e de forma a introduzir aleatoriedade adicional foi gerado pseudo-aleatoriamente um `salt`1 que deve ser do conhecimento de ambos os agentes.

Cifra Um dos requisitos do enunciado era usar uma cifra simétrica de stream. Para tal e algo que foi previamente estudado, decidimos optar pelo ChaCha20. Foi necessário garantir a aleatoriedade dos criptogramas e para tal utilizamos um `nonce`1 gerado pseudo-aleatoriamente. Assim, tendo o nonce ter sido acordado previamente entre os dois agentes, as funções `cifra` e `decifra` cifram uma mensagem e decifraram um criptograma respetivamente.

HMAC Um HMAC é um código que se adiciona ao final de uma mensagem para proteger a integridade da mesma, garantindo que ela foi recebida pelo destinatário sem alterações acidentais. O algoritmo de Hash escolhido foi o SHA256. É utilizado o HMAC para garantir a integridade da chave como também dos dados e metadados.

Notas:

1 - Os números pseudo-aleatórios foram gerados usando a função `urandom` da package `os`

[6]:
```
from cryptography.hazmat.primitives.ciphers.algorithms import ChaCha20
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 getpass import getpass
from os import urandom
from datetime import date

nonce = urandom(16)
salt = urandom(16)

def kdf(salt):
    return PBKDF2HMAC(
                algorithm=hashes.SHA256,
                length=32,
                salt=salt,
                iterations=100000,
```

```
                backend=default_backend()
               )

def cifra(key, msg):
    c = Cipher(algorithms.ChaCha20(key, nonce), mode=None,␣
 ↪backend=default_backend())
    enc = c.encryptor()
    return enc.update(msg)


def decifra(key, nonce, cypher_text):
    c = Cipher(algorithms.ChaCha20(key, nonce), mode=None,␣
 ↪backend=default_backend())
    dec = c.decryptor()
    return dec.update(cypher_text)


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)
```

0.0.2 Agentes da comunicação

Os agentes devem partilhar uma *password* a qual será usada para gerar a chave da cifra e para gerar/validar o HMAC.

Primeiramente, é necessário fazer uma autenticação prévia da chave.Para isso, o `Emitter`, gera a chave a partir da password, e depois gera o HMAC da mesma que é enviada ao `Receiver` juntamente com a mensagem cifrada. Quando é recebida a mensagem,o `Receiver`,também gera a chave a partir de uma password e valida o HMAC da mesma. A mensagem só é decifrada se correr tudo bem com a autenticação da chave.

Depois de cifrar a mensagem o `Emitter` ,adiciona aos metadados da mensagem a data e a hora em que a mesma foi gerada.

Visto que é pedido a autenticação do criptograma e dos metadados, o HMAC é calculado de formas diferentes consoante o lado(Emmiter ou Receiver) em que estejamos. No lado do `Emitter` o HMAC é calculado sobre o criptograma concatenado com o HMAC da chave e respetiva data. Deste modo, apesar de os metadados serem enviados sem confidencialidade, é garantida a sua integridade. Após autenticar a chave, no lado do `Receiver`, valida o HMAC dos metadados concatenados com o criptograma e caso corra tudo bem, o criptograma é decifrado.

[4]:
```
def Emitter(conn):
    password = bytes(getpass('Password do Emmiter:'), 'utf-8')
```

```
    key = kdf(salt).derive(password)
    mac_key = mac(key, key)

    msg = input('Que mensagem pretende enviar?\n')
    msg_bytes = bytes(msg,'utf-8')
    cipher_text = cifra(key, msg_bytes)

    data = bytes(str(date.today()), 'utf-8')
    tag = mac(key, mac_key + data + cipher_text)
    obj = {'mac_key': mac_key, 'data' : data, 'tag' : tag, 'msg' : cipher_text}
    conn.send(obj)

    conn.close()


def Receiver(conn):
    obj = conn.recv()
    mac_key = obj['mac_key']
    data = obj['data']
    tag = obj['tag']
    cipher_text = obj['msg']

    try:
        password = bytes(getpass('Password do Receiver:'), 'utf-8')
        key = kdf(salt).derive(password)
        mac(key, key, mac_key)
        try:
            mac(key, mac_key + data + cipher_text, tag)
            print('Sucess')
            msg = decifra(key,nonce,cipher_text)
            print('Mensagem Recebida:')
            print(msg)
        except:
            print('FAIL')
    except:
        print('Key errada')
    finally:
        conn.close()

BiConn(Emitter,Receiver).manual()
```

```
Password do Emmiter:·····
Que mensagem pretende enviar?
Mensagem que quero enviar
Password do Receiver:·····
Sucess
Mensagem Recebida:
```

```
b'Mensagem que quero enviar'
```