ex2

March 7, 2022

1 Problema 2

2. Criar uma cifra com autenticação de meta-dados a partir de um PRG
    1. Criar um gerador pseudo-aleatório do tipo XOF ("extened output function") usando o SHAKE256, para gerar uma sequência de palavras de 64 bits.
        1. O gerador deve poder gerar até um limite de ,2^n, palavras (n é um parâmetro) armazenados em long integers do Python.
        2. A "seed" do gerador funciona como cipher_key e é gerado por um KDF a partir de uma "password" .
        3. A autenticação do criptograma e dos dados associados é feita usando o próprio SHAKE256.
    2. Defina os algoritmos de cifrar e decifrar : para cifrar/decifrar uma mensagem com blocos de 64 bits, os "outputs" do gerador são usados como máscaras XOR dos blocos da mensagem. Essencialmente a cifra básica é uma implementação do "One Time Pad".

[28]:
```
import os
import random
import sys
import time
import teste
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives import padding
from ast import Try
from hmac import digest
from inspect import signature
import re
from select import select
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
import cryptography.exceptions
import os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
```

2 Introdução

Fazendo uma breve introdução ao exercício, foram criados dois utilizadores a Alice e o Bob sendo estes o emissor e o recetor, respetivamente. Como é pedido no enunciado é gerada uma seed através de uma password comum aos dois intervinentes. Com esta seed é depois gerada a key comum a ambos. Esta key é usada posteriormente para codificar a mensagem fazendo o XOR ao texto cifrado. No lado do recetor é realizado o processo inverso para descodificação.

3 Emissor

[29]:
```
#hMAc
class Alice:
    def __init__(self):
        self.chiper_key=None
        self.salt=os.urandom(16)


    def setKey(self,key):
        self.chiper_key=key

#autenticacao modo hmac
    def authenticate(self):
        tent=b"olaololaol"
        h=hmac.HMAC(self.chiper_key,hashes.SHA256())
        h.update(tent)
        signature=h.finalize()
        return signature


#algoritmo para a geracao da seed atraves da password
    def generate_seed(self,password):
        kdf= PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=64,
        salt=self.salt,
        iterations=390000,
    )
        seed=kdf.derive(password)
        return seed

    #codificacao da mensagem a enviar atraves da cifra one time pad
    #usa-se primeiramente padding para preencher a mensagem com 64 bits e␣
 ↪depoisa aplica-se o xor
    def code(self,message):
        ct=b''
        padder=padding.PKCS7(64).padder()
        padded_data= padder.update(message)
```

```
        padded_data += padder.finalize()
        for x in range (0,len(padded_data),8):
            p=padded_data[x:x+8]
            for index, byte in enumerate(p):
                ct += bytes([byte ^ self.chiper_key[x*8:(x+1)*8][index]])
        return ct

    #envio da mensagem concatenada com o salt e a signature
    def send_message(self,data):
        signature=self.authenticate()
        ct= self.code(data)
        return signature+self.salt+ct
```

Nesta classe temos os metodos referentes ao emissor. A classe inicia-se com uma chave e um salt que vão ser usados para a codificação da mensagem.Temos também o metodo de autenticação que é feito através do modo hmac que usa o sha256.A seguir temos o metodo para a criação da seed através da password inserida pelo utilizador que é feita através do SHAKE256, gerando uma sequencia com tamanho de 64 bits, parametro estipulado no metodo, e é usado também o salt gerado aleatoriamente do utilizador como parametro. A seguir temos o metodo de codifcação da mensagem que utiliza a cifra one time pad que é uma cifra que o tamanho da chave tem que ser igual ao tamanho de texto a ser codificado.Devido a este facto aplicamos o padding a mensagem para esta ter um tamanho de 64 bits e a seguir aplicamos um xor a cada byte de texto usando o respetivo byte da chave.Por fim temos o metodo de enviar a mensagem que utiliza o metodo de autenticação para obter a assinatura do emissor e concatena a mensagem com o salt e o texto codificado.

4 Recetor

[30]:
```
class Bob:
    def __init__(self):
            self.chiper_key=None
            self.salt=None

    def setKey(self,key):
        self.chiper_key=key

#verifica a signature recebida do emissor
    def verify(self,signature):
        tent=b"olaololaol"
        h=hmac.HMAC(self.chiper_key,hashes.SHA256())
        h.update(tent)
```

```
        try:
            h.verify(signature)
            return True
        except cryptography.exceptions.InvalidSignature:
            return False


    #gera a seed através da password inserida
    def generate_seed(self, password):
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=64,
            salt=self.salt,
            iterations=390000,
            )
        seed = kdf.derive(password)
        return seed

    def getSalt(self,text):
        self.salt=text[32:48]

    #descodifica a mensagem recebida aplicando o xor ao texto codificado e
    #depois retira-se os bytes adicionados anteriormente com o unppading
    def decode(self, ct):
        pt = b''
        for x in range (0,len(ct),8):
            p=ct[x:x+8]
            for index, byte in enumerate(p):
                pt += bytes([byte ^ self.chiper_key[x*8:(x+1)*8][index]])
    # Algoritmo para retirar padding para decifragem
        unpadder = padding.PKCS7(64).unpadder()
    # Retira bytes adicionados
        unpadded = unpadder.update(pt) + unpadder.finalize()
        return unpadded.decode("utf-8")

#rececao da mensagem e divisão da mensagem nas suas componentes
#verifica a signature e procede-se ao decode da mensagem
    def recieve_message(self,cpr):
        (signature,salt,ct)=parser(cpr)
        autenticacao=self.verify(signature)
        if autenticacao ==False:
            print("erro")
        else:
            text_plain=self.decode(ct)
            print(text_plain)
```

Os métodos da classe do Recetor em relação à geração da key são idênticos aos da classe do emissor. Temos também o método que verifica se a assinatura recebida é igual à que o recetor tem. Caso isto se verifique procede-se à descodificação da mensagem que aplica o XOR ao texto cifrado para obter o texto descodificado. Após este passo procede-se ao unpadding da mensagem para retirar os bytes adicionados na codificação e obtém-se assim o texto descodificado.

[31]:
```
#parser para a mensagem recebida
def parser(msg):
    signature=msg[:32]
    salt=msg[32:48]
    ct=msg[48:]
    return (signature,salt,ct)


#metodo para a geracao da chave atraves da seed
def prg(seed,N):
    digest=hashes.Hash(hashes.SHAKE256(8*pow(2,N)))
    digest.update(seed)
    long_integers=digest.finalize()
    return long_integers
```

[32]:
```
def main() :
    alice = Alice()
    bob =Bob()

    password=input ("password: ")

    start=time.time()
    #gera-se a seed para a alice através da password
    seed_emitter=alice.generate_seed(password.encode())

   #gera-se a chave da alice através da seed
    key_emitter=prg(seed_emitter,4)
    alice.setKey(key_emitter)

    #envio da mensagem para o bob
    ct=alice.send_message("Daniel e Joao a estudar Estruturas criptograficas a␣
 ↪segunda no mestrado de engenharia informatica na universidade do minho".
 ↪encode())

    #define-se o salt do bob através da mensagem recebida
    bob.getSalt(ct)
    #com o salt recebido gera-se a seed do bob e uma vez que os parametros são␣
 ↪os mesmos a seed e a respetiva password
    #vao coincidir
    seed_reciever= bob.generate_seed(password.encode())
    key_reciever=prg(seed_reciever,4)
```

```
    bob.setKey(key_reciever)

    #bob recebe a mensagem da alice
    bob.recieve_message(ct)
    end=time.time()
    print(end-start)

if __name__== "__main__":
    main()
```

```
Daniel e Joao
0.9853951930999756
```

Na main simula-se a comunicação assíncrona onde o emissor e o recetor geram a seed através da password comum. A seguir com esta seed o emissor gera a sua chave e envia a mensagem codificada. O recetor recebe a mensagem, divide-a nos seus componentes (signature,salt,texto) e usa o salt recebido do emissor para gerar uma key através da seed. Como os parâmetros da criação são iguais as chaves vão ser as mesmas. Depois só é preciso proceder à descodificação da mensagem. É usado o time para ver o tempo que o algoritmo demora a executar.