ex1

March 7, 2022

1 Problema 1

1. Criar um comunicação privada assíncrona entre um agente Emitter e um agente Receiver que cubra os seguintes aspectos:
    1. Autenticação do criptograma e dos metadados (associated data). Usar uma cifra simétrica num modo HMAC que seja seguro contra ataques aos "nounces" .
    2. Os "nounces" são gerados por um gerador pseudo aleatório (PRG) construído por um função de hash em modo XOF.
    3. O par de chaves cipher_key, mac_key, para cifra e autenticação, é acordado entre agentes usando o protocolo DH com autenticação dos agentes usando assinaturas DSA.

[1]:
```
from ast import Try
from hmac import digest
from inspect import signature
import re
from select import select
import time
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 problema, foram criados dois utilizadores a Alice e o Bob sendo estes o emissor e o recetor, respetivamente. Como pede no enunciado do problema, antes de fazer qualquer troca de mensagens estabelecem a chave comum entre eles através do protocolo DH. Após esta chave ser verificada procede-se ao envio da mensagem cifrada concatenando um nounce e uma assinatura digital produzida através do HMAC.

3 Emissor

[2]:
```
class Alice:
    def __init__(self):
        self.private_key=None
        self.shared_key=None

    #gera-se a chave publica
    def generate_key(self,parameters):
        self.private_key = parameters.generate_private_key()

    #gera-se a chave partilhada entre ambos para o protocolo DH
    def generate_derived_key(self,bob_private_key):
        derived_key=self.private_key.exchange(bob_private_key)
        self.shared_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'handshake data',
).derive(derived_key)

    #gera-se a signature para autenticação da mensagem através do HMAC
    def authenticate(self):
        chave=b"olaololaol"
        h=hmac.HMAC(self.shared_key,hashes.SHA256())
        h.update(chave)
        signature=h.finalize()
        return signature


    def encrypt1 (self,data,associated_data):
        aesgcm = AESGCM(self.shared_key)
        nonce=os.urandom(16)
        ct = aesgcm.encrypt(nonce,data,associated_data)
        return nonce,ct

    #gera-se a signature através do HMAC e contatena-se a mensagem com o nonce␣
 ↪gerado aleatoriamente
    # a associated_data que não é encriptada e data que é encriptada
    def send_message(self,associated_data,data):
        signature=self.authenticate()
        nonce,ct= self.encrypt1(data,associated_data)
        return signature+nonce+associated_data+ct
```

Nesta classe, temos os métodos referentes ao emissor. Criamos a classe com duas chaves, a chave

privada que só o emissor tem acesso e a chave partilhada que é conseguida através do método generate_derived_key(), que usando a função exchange entre a chave privada do emissor e a chave pública obtemos uma chave que à partida espera-se que seja igual entre os dois intervinientes da comunicação. Para além disso, temos o método de autenticação que dada uma chave atribuída pelo utilizador e a chave pública vai gerar uma assinatura através do modo HMAC que verifica do lado do recetor se as chaves coincidem. Por fim, temos o método de enviar a mensagem,que usando um método de Galois e um nounce aleatoriamente gerado codifica a mensagem e posteriormente concatena todos os componentes relevantes para o envio da mensagem (signature,nonce, associated_data e a mensagem) e envia para o outro interviniente.

4 Recetor

[3]:
```
class Bob:
    def __init__(self):
        self.private_key=None
        self.shared_key=None


    #gera-se a chave privada
    def generate_key(self,parameters):
        self.private_key = parameters.generate_private_key()

    #gera-se a chave partilhada entre ambos
    def generate_derived_key(self,alice_private_key):
        derived_key=self.private_key.exchange(alice_private_key)
        self.shared_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'handshake data',
).derive(derived_key)


    #utiliza-se o HMAC para autenticar a mensagem recebida
    def verify(self,signature):
        chave=b"olaololaol"
        h=hmac.HMAC(self.shared_key,hashes.SHA256())
        h.update(chave)
        try:
            h.verify(signature)
            return True
        except cryptography.exceptions.InvalidSignature:
            return False



    def decrypt1 (self,ciphertext,nonce,associated_data) :
```

```
        aesgcm = AESGCM(self.shared_key)
        try:
            plain_text=aesgcm.decrypt(nonce,ciphertext,associated_data)
        except cryptography.exceptions.InvalidTag:
            return 1,None
        return None,plain_text.decode('utf-8')


#realiza-se o parsing da mensagem recebida,
#verifica-se se a signature corresponde a signature que o bob tem
#realiza-se a descodificação da mensagem
    def recieve_message(self,cpr):
        (signature,nonce,associated_data,ct)=parser(cpr)
        autenticacao=self.verify(signature)
        if autenticacao ==False:
            print("erro")
        else:
            error_code,text_plain=self.decrypt1(ct,nonce,associated_data)
            print(text_plain)
```

Nesta classe, temos os métodos referentes ao recetor. Criamos a classe com duas chaves, a chave privada que só o recetor tem acesso e a chave partilhada que é conseguida através do método generate_derived_key() que usando a função exchange entre a chave privada do emissor e a chave pública obtemos uma chave que à partida espera-se que seja igual entre os dois intervinientes da comunicação. Para além disso temos o método de verificação da assinatura digital da mensagem recebida e só vai ser feita a descodificação da mensagem caso esta coincida com a cifra gerada por HMAC do recetor. No método de receção da mensagem realizamos o parsing da mensagem referida e retiramos os componentes importantes à nossa comunicação. Usamos a signature no método de verificação referido na frase anterior e o texto encriptado, o nounce, e os dados associados são passados para o método de descodificação. No método de descodificação através de um método de Galois obtemos o texto inicial enviado pelo emissor.

[4]:
```
#parser da mensagem enviada pela alice ao bob
def parser(msg):
    signature=msg[:32]
    nonce =msg[32:48]
    associated_data = msg[48:60]
    ct = msg[60:]
    return (signature,nonce,associated_data,ct)
```

[5]:
```
def main() :

    #gera-se os parametros para a criação da chave privada
    parameters = dh.generate_parameters(generator=2, key_size=512)
    alice = Alice()
    bob =Bob()
```

```
    #gera-se a chave privada para a alice e o bob
    start=time.time()
    alice.generate_key(parameters)
    bob.generate_key(parameters)



    #utiliza-se a chave publica para gerar a chave partilhada entre o bob e␣
 ↪alice
    alice.generate_derived_key(bob.private_key.public_key())
    bob.generate_derived_key(alice.private_key.public_key())

    #envio de mensagem
    msg=b"Daniel e Joao a estudar Estruturas criptograficas a segunda no␣
 ↪mestrado de engenharia informatica na universidade do minho fewifmeqmfqfewq␣
 ↪fkeowqmfewioqfimewq fewijqfmioewqoifewq fijeqijfewoqife jpofewqjpofewqjpof␣
 ↪fjpewqpjfewqp"
    msg2=os.urandom(12)

    #se a chave coincidir procede-se ao envio e receção da mensagem autenticada␣
 ↪por HMAC
    if(alice.shared_key==bob.shared_key) == True:
        ct=alice.send_message(msg2,msg)
        print(ct)
        bob.recieve_message(ct)
    else:
        "erro"

    end=time.time()
    print(end-start)


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

```
b'\xa8\xfe G\xdbS\x8fEs\x18\xd8\xe1HtX\x84\x10\x93d.i\x10\xa4\x98\xfar\xbc\xc6\x
cfiy\tU\x16\x16\x88^Z.6Z\xffz\xbeF\xa6nD\x1e\xe1\xc0\xad\x10\x03\xbf9\x0fK\xf0\x
b6\x04#:\xd9M$F\xefN\x98\xa1\xc3\xd9\x93\x89\xc5)\xdd\x1f\xe4\x11\x9f~x\x19\xa47
\x16\xa1\xcdT\xb0QLO\xa6\xca\xa3\x1b\x9d\xf7\xefgP\xdc\x0e\xf5}\xf3\x8f\xeb\xabe
\x1e\xd0\xa4\x11\x11\xb5\x85G\x8dn\xe2u\x013\xfcuj\x89\xe9\xfaS\x1c\xc6|\xcf\xdb
\xa4\xec\xea_\xdd\x83,\x9f\x7f0[{\xb7\xa9\xe0\x87\x9b\xca\xa1\xb40!H\x87\xf1\x01
\xde0`n\xc34w\x85\'\x19\x13F\xb1B2\xe8\xa4\x95fI\x14U\xcf\xd8\x1a\x97\x8d,(\x88v
V\x82y\xe6\xaf\x156V@\xc0\xf2\x1e{\xbc\x00\xfd\x05\xa7\xd4\xc10~[\xf3Zr~\x00F\x9
e+0\xad\xf0#\xe1Fe>\x9e\xd7\xa6\xbb\x97#\xc1%u|w%\xb7\xd7\xceO\x91\xa2\xda\x95)m
\xb8\xe5\x82\xd9my\xef\xacd\x05l"\x1b\xe8\x8d\xd0\x93[\xcd\x10\x82\x0c\t\x18\x8e
\xb2s\xfb,!\x9a\xcd\x16\xa98c~\xe4\xab\xd8\xb8\xd3\xe5+$\xaa'
Daniel e Joao a estudar Estruturas criptograficas a segunda no mestrado de
```

```
engenharia informatica na universidade do minho fewifmeqmfqfewq
fkeowqmfewioqfimewq fewijqfmioewqoifewq fijeqijfewoqife jpofewqjpofewqjpof
fjpewqpjfewqp
0.0014171600341796875
```

Na main é onde é simulada a comunicação assíncrona entre o bob e a alice. Inicialmente são gerados parâmetros comuns a ambos para estes gerarem a respetiva chave privada. De seguida, cada um usa a chave pública e aplica o método generate_derived_key() para gerar a chave comum entre ambos. Após este método verifica-se se a chave partilhada dos dois coincide e se coincidir procede-se ao envio da mensagem e a respetiva receção. Utilizou-se a biblioteca time para as medições de performance referentes a pergunta três.