TP1_EX3

March 7, 2023

1 TP1

1.1 Grupo 17:

PG50315 - David Alexandre Ferreira Duarte

PG51247 - João Rafael Cerqueira Monteiro

1.2 Exercício 3.

Use o "package" Cryptography para - Implementar uma AEAD com "Tweakable Block Ciphers" conforme está descrito na última secção do texto +Capítulo 1: Primitivas Criptográficas Básicas. A cifra por blocos primitiva, usada para gerar a "tweakable block cipher", é o AES-256 ou o ChaCha20. - Use esta cifra para construir um canal privado de informação assíncrona com acordo de chaves feito com "X448 key exchange" e "Ed448 Signing&Verification" para autenticação dos agentes. Deve incluir uma fase de confirmação da chave acordada.

[1]:
```
!pip install cryptography
```

```
Requirement already satisfied: cryptography in
/home/sleiman/mambaforge/envs/sage/lib/python3.10/site-packages (39.0.2)
Requirement already satisfied: cffi>=1.12 in
/home/sleiman/mambaforge/envs/sage/lib/python3.10/site-packages (from
cryptography) (1.15.1)
Requirement already satisfied: pycparser in
/home/sleiman/mambaforge/envs/sage/lib/python3.10/site-packages (from
cffi>=1.12->cryptography) (2.21)
```

[2]:
```
#Imports
import os
from logging import raiseExceptions
from pydoc import plain
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms
from cryptography.hazmat.primitives import hmac, hashes
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives.asymmetric.x448 import X448PrivateKey
from cryptography.hazmat.primitives.asymmetric.ed448 import Ed448PrivateKey
from cryptography.hazmat.backends import default_backend
```

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
```

Decidimos dividir o programa em duas classes, emissor e recetor e entre eles, foram utilizados o acordo com "X448 key exchange" e "Ed448 Signing&Verification" para a autenticação entre os dois agentes.

Na primeira classe (emissor):

[3]:
```
class emissor:
    # variavés iniciadas para comunicação e cifrar da mensagem
    def __init__(self, assinatura):
        self.mensagemAssinatura = assinatura.encode('utf-8')
        self.mac = None
        self.chavePrivadaEd448 = self.gerarChavePrivadaEd448()
        self.chavePublicaEd448 = self.gerarChavePublicaEd448()
        self.assinatura = self.gerarAssinaturaEd448()

        self.chavePrivadaX448 = self.gerarChavePrivadaX448()
        self.chavePublicaX448 = self.gerarChavePublicaX448()
        self.chavePartilhadaX448 = None



    # "Ed448 Signing&Verification" metodo para gerar a assinatra com a␣
 ↪mensagemAssinatura
    def gerarAssinaturaEd448(self):
        return self.chavePrivadaEd448.sign(self.mensagemAssinatura)

    # gerar a chave privada ED448
    def gerarChavePrivadaEd448(self):
        return Ed448PrivateKey.generate()

    # gerar a chave pública ED448 a partir da privada já gerada
    def gerarChavePublicaEd448(self):
        return self.chavePrivadaEd448.public_key()

    # gerar a chave privada X448
    def gerarChavePrivadaX448(self):
        return X448PrivateKey.generate()

    # gerar a chave pública X448 a partir da privada já gerada
    def gerarChavePublicaX448(self):
        return self.chavePrivadaX448.public_key()

    # gera a chave partilhada a partir da mistura da sua privada e publica do␣
 ↪receiver
    def gerarChavePartilhadaX448(self, chavePublicaRecetorX448):
        key = self.chavePrivadaX448.exchange(chavePublicaRecetorX448)
```

```
        self.chavePartilhadaX448 = HKDF(
            algorithm = hashes.SHA256(),
            length = 32,
            salt = None,
            info = b'handshake data',
        ).derive(key)

    # gera a chave que o receiver tem de confirmar para saber se está a receber a informação de quem pretende
    def chaveParaConfirmar(self):
        nonce = os.urandom(16)
        algorithm = algorithms.ChaCha20(self.chavePartilhadaX448, nonce)
        cipher = Cipher(algorithm, mode=None)
        encryptor = cipher.encryptor()
        ciphered = encryptor.update(self.chavePartilhadaX448)
        ciphered = nonce + ciphered
        return ciphered

    # cria a autenticação utilizando o HMAC
    def criarAutenticacao(self, message):
        h = hmac.HMAC(self.chavePartilhadaX448, hashes.SHA256(), backend=default_backend())
        h.update(message)
        self.mac = h.finalize()


    # gerar um tweak: nonce de 8 bytes + contador de 7 bytes + 1 byte da tag
    def gerarTweak(self, contador, tag):
        nonce = os.urandom(8)
        return nonce + contador.to_bytes(7,byteorder = 'big') + tag.to_bytes(1,byteorder = 'big')


    # método para cifrar
    def cifrar(self, mensagem):
        mensagem = mensagem.encode('utf-8')
        # ver o tamanho da mensagem
        tamanho = len(mensagem)
        # tratar do padding da mensagem
        padder = padding.PKCS7(64).padder()
        padded = padder.update(mensagem) + padder.finalize()
        criptograma = b''
        contador = 0
        # dividir em blocos de 16
        for i in range(0,len(padded),16):
            p = padded[i:i+16]
            # é o último bloco?
```

```
            if (i+16+1 > len(padded)):
                # ultimo com tag 1
                tweak = self.gerarTweak(tamanho,1)
                criptograma += tweak
                meio = b''
                for index, byte in enumerate(p):
                    # aplicar a máscara XOR aos blocos . Esta mascara é␣
↪compostas pela shared_key + tweak
                    mascara = self.chavePartilhadaX448 + tweak
                    meio += bytes([byte ^ mascara[0:16][0]])
                criptograma += meio
            #não é o último?
            else:
                #Blocos intermédios com tag 0
                tweak = self.gerarTweak(contador,0)
                #O bloco é cifrado com AES256, num modo de utilização de tweaks
                cipher = Cipher(algorithms.AES(self.chavePartilhadaX448),␣
↪mode=modes.XTS(tweak))
                encryptor = cipher.encryptor()
                ultimaParte = encryptor.update(p)
                criptograma += tweak + ultimaParte
            contador += 1
        #a mensagem final cifrada é composta por tweak(16)+bloco(16)
        #Adicionalmente é enviada uma secção de autenticação para verificação␣
↪antes de decifrar a mensagem
        self.criarAutenticacao(criptograma)
        criptogramaFinal = self.mac + criptograma
        return criptogramaFinal
```

Para o recetor foram criados os métodos para a decifragem e criação das respetivas chaves X448 para a comunicação:

[4]:
```
class recetor:
    # variavés iniciadas para comunicação e decifrar da mensagem
    def __init__(self, assinatura):
        self.chavePrivadaX448 = self.gerarChavePrivadaX448()
        self.chavePublicaX448 = self.gerarChavePublicaX448()
        self.chavePartilhadaX448 = None
        self.tweakable = None
        self.mensagemAssinatura = assinatura.encode('utf-8')

    # verificação das assinaturas
    def verificarAssinaturaEd448(self, assinatura, chavePublica):
            try:
                chavePublica.verify(assinatura, self.mensagemAssinatura)
            except: #InvalidSignature:
                raiseExceptions("Autenticação dos agentes falhou!")
```

```
    # gerar a chave privada X448
    def gerarChavePrivadaX448(self):
        # Generate a private key for use in the exchange.
        return X448PrivateKey.generate()

    # gerar a chave pública X448
    def gerarChavePublicaX448(self):
        return self.chavePrivadaX448.public_key()

    # gerar a chave partilhada a partir da mistura da sua privada e publica do␣
↪receiver
    def gerarChavePartilhadaX448(self, chavePublicaEmissorX448):
        key = self.chavePrivadaX448.exchange(chavePublicaEmissorX448)
        self.chavePartilhadaX448 = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'handshake data',
        ).derive(key)


    # gerir os tweaks
    def handleTweak(self, tweak):
        self.tweakable = tweak
        self.final_key = self.chavePartilhadaX448 + self.tweakable

    # confirmação das chave
    def confirmarChave(self, cpht):
        #16 bytes reservados para o nonce
        nonce = cpht[0:16]
        #o restante do texto cifrado corresponde à key
        key = cpht[16:]
        #Utilização do Chacha20
        algorithm = algorithms.ChaCha20(self.chavePartilhadaX448, nonce)
        cipher = Cipher(algorithm, mode=None)
        decryptor = cipher.decryptor()
        d_key = decryptor.update(key)
        if d_key == self.chavePartilhadaX448:
            print("\nChaves acordadas com sucesso!\n")
        else:
            raiseExceptions("Erro na verificacao das chaves acordadas")

    # verificar e autenticar a mensagem que recebeu
    def verificarAutenticarMensagem(self, mac_signature, ciphertext):
        h = hmac.HMAC(self.chavePartilhadaX448, hashes.SHA256(),␣
↪backend=default_backend())
```

```
        h.update(ciphertext)
        h.verify(mac_signature)

    # degenerar o tweak
    def separarTweak(self, tweak):
        nonce = tweak[0:8]
        contador = int.from_bytes(tweak[8:15], byteorder = 'big')
        tagFinal = tweak[15]
        return nonce, contador, tagFinal

    # decifrar a mensagem
    def decifrar(self, criptograma):
        #32 bytes de autenticação
        mac = criptograma[0:32]
        ct = criptograma[32:]
        try:
            #verificar o mac
            self.verificarAutenticarMensagem(mac, ct)
        except:
            raiseExceptions("Autenticação com falhas!")
            return
        #decifrar
        plaintext = b''
        f = b''

        #no total: bloco + tweak corresponde a corresponde a 32 bytes.
        tweak = ct[0:16]
        block = ct[16:32]
        i = 1
        _, contador, tagFinal = self.separarTweak(tweak)
        #Se não for o último bloco:
        while(tagFinal!=1):
            #decifrar com o algoritmo AES256 e o respetivo tweak
            cipher = Cipher(
                algorithms.AES(self.chavePartilhadaX448),
                mode = modes.XTS(tweak)
            )
            decryptor = cipher.decryptor()
            f = decryptor.update(block)
            plaintext += f
            #obtem o proximo tweak e o proximo bloco
            tweak = ct[i*32:i*32 +16]
            block = ct[i*32 +16:(i+1)*32]
            #desconstroi o proximo tweak
            _, contador, tagFinal = self.separarTweak(tweak)
            i+= 1
        #Se for o ultimo bloco
```

```
        if (tagFinal == 1):
            c =b''
            for index, byte in enumerate(block):
                #aplicar as máscaras XOR aos blocos para decifrar
                mascara = self.chavePartilhadaX448 + tweak
                c += bytes([byte ^ mascara[0:16][0]])
            plaintext += c

        #realiza o unpadding
        unpadder = padding.PKCS7(64).unpadder()
        unpadded_message = unpadder.update(plaintext) + unpadder.finalize()

        #Uma vez que o último bloco possui o tamanho da mensagem cifrada, basta␣
↪verificar se correspondem os valores e não houve perdas de blocos da␣
↪mensagem
        if (len(unpadded_message.decode("utf-8")) == contador):
            print("Tweak de autenticação validado!")
            return unpadded_message.decode("utf-8")
        else: raiseExceptions("Tweak de autenticação inválido")
```

Na parte final iniciamos a testagem se tudo feito em cima estava funcional:

1- criar uma assinatura e uma mensagem para cifrar e decifrar

2- iniciar o emissor com a respetiva mensagem e assinatura

3- iniciar o recetor mas apenas com a assinatura

4- começar a autenticação dos agentes enviando ao recetor a assinatura do emissor e a sua chave publica ED448

5- estabelecer a chave partilhada em X448 entre o emissor e o recetor

6- Verificar e confirmar a chave do texto cifrado

7- O emissor cifrar a mensagem num criptograma e o recetor decifrar o mesmo criptograma

8- Mostrar a mensagem decifrada e comparando com a original observamos que a operação ocurreu com sucesso

[5]:
```
# 1
assinatura = "grupo17"
mensagem = "Estruturas Criptograficas"

# 2
emissor = emissor(assinatura)

# 3
recetor = recetor(assinatura)

# 4
```

```
recetor.verificarAssinaturaEd448(emissor.assinatura, emissor.chavePublicaEd448)

# 5
emissor.gerarChavePartilhadaX448(recetor.chavePublicaX448)
recetor.gerarChavePartilhadaX448(emissor.chavePublicaX448)

# 6
chaveTextoCifrado = emissor.chaveParaConfirmar()
recetor.confirmarChave(chaveTextoCifrado)

# 7
criptograma = emissor.cifrar(mensagem)
plaintext = recetor.decifrar(criptograma)

# 8
print("A mensagem decifrada: " , plaintext)
```

```
Chaves acordadas com sucesso!

Tweak de autenticação validado!
A mensagem decifrada:  Estruturas Criptograficas
```