Trabalho Prático 1

Grupo 17, constituído por:

-- Joana Castro e Sousa, PG47282

-- Tiago Taveira Gomes, PG47702

-- João Carlos Pereira Rodrigues, PG46534

Pergunta 1:

Use o "package" Cryptography para:

a) 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.

b) Use esta construção 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.

In [ ]:
```
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
```

De forma a conseguirmos estabelecer uma conexão assíncrono com acordo de chaves feito com "X448 key exchange" e "Ed448 Signing&Verification" para autenticação dos agentes, foi implementada a seguinte classe para estabelecer um emissor:

In [ ]:
```
class emitter:

    def __init__(self, mensagem, assinatura):
        self.message = mensagem.encode('utf-8')
        self.signing_message = assinatura.encode('utf-8')
        self.mac = None
        self.Ed448_private_key = self.generate_Ed448_private_key()
        self.Ed448_public_key = self.generate_Ed448_public_key()
        self.signature = self.generate_Ed448_signature()

        self.X448_private_key = self.generate_X448_private_key()
        self.X448_public_key = self.generate_X448_public_key()
        self.X448_shared_key = None
```

```
    #“Ed448 Signing&Verification”
    def generate_Ed448_signature(self):
        return self.Ed448_private_key.sign(self.signing_message)

    #gerar a chave privada
    def generate_Ed448_private_key(self):
        return Ed448PrivateKey.generate()

    # gerar a chave pública
    def generate_Ed448_public_key(self):
        return self.Ed448_private_key.public_key()

    #“X448 key exchange”
    def generate_X448_private_key(self):
        # Gera a private key utilizando X448
        return X448PrivateKey.generate()

    def generate_X448_public_key(self):
        # Gera a chave pública a partir da privada já gerada
        return self.X448_private_key.public_key()

    #Gera a chave partilhada a partir da mistura da sua privada e publica do receiver
    def generate_X448_shared_key(self, X448_receiver_public_key):
        key = self.X448_private_key.exchange(X448_receiver_public_key)
        self.X448_shared_key = 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 key_to_confirm(self):
        nonce = os.urandom(16)
        algorithm = algorithms.ChaCha20(self.X448_shared_key, nonce)
        cipher = Cipher(algorithm, mode=None)
        encryptor = cipher.encryptor()
        ciphered = encryptor.update(self.X448_shared_key)
        ciphered = nonce + ciphered
        return ciphered

    def create_authentication(self, message):
        h = hmac.HMAC(self.X448_shared_key, 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 generate_tweak(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 encript_implementation(self):
        # ver o tamanho da mensagem
        size_msg = len(self.message)
        # tratar do padding da mensagem
```

```
        padder = padding.PKCS7(64).padder()
        padded = padder.update(self.message) + padder.finalize()
        cipher_text = 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.generate_tweak(size_msg,1)
                cipher_text += tweak
                middle = b''
                for index, byte in enumerate(p):
                    # aplicar a máscara XOR aos blocos . Esta mascara é compostas pela shared_key + tweak
                    mascara = self.X448_shared_key + tweak
                    middle += bytes([byte ^ mascara[0:16][0]])
                cipher_text += middle
            #não é o último?
            else:
                #Blocos intermédios com tag 0
                tweak = self.generate_tweak(contador,0)
                #O bloco é cifrado com AES256, num modo de utilização de tweaks
                cipher = Cipher(algorithms.AES(self.X448_shared_key), mode=modes.XTS(tweak))
                encryptor = cipher.encryptor()
                ct = encryptor.update(p)
                cipher_text += tweak + ct
            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.create_authentication(cipher_text)
        final_ciphered = self.mac + cipher_text
        return final_ciphered
```

Definido o Emissor, é necessário implementar também todos os métodos relativos ao Recetor

In [ ]:

```
class receiver:
    def __init__(self, assinatura):
        self.X448_private_key = self.generate_X448_private_key()
        self.X448_public_key = self.generate_X448_public_key()
        self.X448_shared_key = None
        self.tweakable = None
        self.signing_message = assinatura.encode('utf-8')

    # verificação das assinaturas
    def verify_Ed448_signature(self, signature, public_key):
            try:
                public_key.verify(signature, self.signing_message)
            except: #InvalidSignature:
                raiseExceptions("Autenticação dos agentes falhou!")

    #gerar a chave privada
    def generate_X448_private_key(self):
        # Generate a private key for use in the exchange.
        return X448PrivateKey.generate()

    #gerar a chave pública
```

```
    def generate_X448_public_key(self):
        return self.X448_private_key.public_key()

    #chave partilhada
    def generate_X448_shared_key(self, X448_emitter_public_key):
        key = self.X448_private_key.exchange(X448_emitter_public_key)
        self.X448_shared_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'handshake data',
        ).derive(key)


    # gerir os tweaks
    def handle_tweak(self, tweak):
        self.tweakable = tweak
        self.final_key = self.X448_shared_key + self.tweakable

    #confirmação das chave
    def confirm_key(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.X448_shared_key, nonce)
        cipher = Cipher(algorithm, mode=None)
        decryptor = cipher.decryptor()
        d_key = decryptor.update(key)
        if d_key == self.X448_shared_key:
            print("\nChaves acordadas com sucesso!\n")
        else:
            raiseExceptions("Erro na verificacao das chaves acordadas")

    #verificar e autenticar a mensagem que recebeu
    def verify_authenticate_message(self, mac_signature, ciphertext):
        h = hmac.HMAC(self.X448_shared_key, hashes.SHA256(), backend=default_backend())
        h.update(ciphertext)
        h.verify(mac_signature)

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

    # decifrar a mensagem
    def decript_implementation(self, ctt):
        #32 bytes de autenticação
        mac = ctt[0:32]
        ct = ctt[32:]
        try:
            #verificar o mac
            self.verify_authenticate_message(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, tag_final = self.degenerate_tweak(tweak)
        #Se não for o último bloco:
        while(tag_final!=1):
            #decifrar com o algoritmo AES256 e o respetivo tweak
            cipher = Cipher(algorithms.AES(self.X448_shared_key), 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, tag_final = self.degenerate_tweak(tweak)
            i+= 1
        #Se for o ultimo bloco
        if (tag_final == 1):
            c =b''
            for index, byte in enumerate(block):
                #aplicar as máscaras XOR aos blocos para decifrar
                mascara = self.X448_shared_key + 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")
```

Tendo todas estas classes da conexão implementadas, basta-nos começar a tratar do teste e implementar o algoritmo.

Desta forma, primeiramente tem de existir um acordo de chaves entres os agentes.

Uma vez estabelecido este acordo, bastará enviar a mensagem cifrada (o criptograma) e decifrar este criptograma.

In [ ]:
```
# estabelecer uma assinatura
assinatura = "aslkdasdkjasdkjaskjsdkj"
# estabelecer a mensagem a cifrar e decifrar
mensagem = "uma mensagem a enviar para ser cifrada e decifrada"

# estabelecer o emissor com a mensagem e a assinatura
emitter = emitter(mensagem, assinatura)
#estabelecer o recetor com a assinatura
receiver = receiver(assinatura)
```

```
# fase de autenticação dos agentes (Ed448)
receiver.verify_Ed448_signature(emitter.signature, emitter.Ed448_public_key)

# fase do estabelecer as chaves (X448)
emitter.generate_X448_shared_key(receiver.X448_public_key)
receiver.generate_X448_shared_key(emitter.X448_public_key)

# verificar e confirmar todo este acordo
key_ciphertext = emitter.key_to_confirm()
receiver.confirm_key(key_ciphertext)


# enviar a mensagem encriptada
ciphertext = emitter.encript_implementation()
# decifrar o criptograma
plaintext = receiver.decript_implementation(ciphertext)
# verificar o criptograma decifrado
print("A mensagem decifrada: " , plaintext)
```

```
Chaves acordadas com sucesso!

Tweak de autenticação validado!
A mensagem decifrada:  uma mensagem a enviar para ser cifrada e decifrada
```