TP1 | Exercício 3 | Grupo 15

Pedro Araújo - pg50684

Nuno Dias Mata - Pg44420

In [4]:

```
import os
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
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 logging import raiseExceptions
```

Emitter

Começando por analisar o desenvolvimento do agente emitter, que irá cifrar a mensagem e envia-lo para o receiver

- **Foi utilizado algumas variáveis de instância para a classe , principalmente para guardar os valores das chaves publicas e privadas de autenticação**

In [10]:

```
class emitter:
    sign = b"Signing Message" # assinatura
    mac = b""

    X_private_key = b""
    X_public_key = b""
    X_shared_key = b""

    Ed_private_key = b""
    Ed_public_key = b""
    Ed_signature = b""
```

- **Geração das chaves públicas e privadas e também da assinatura, utilizando a curva elíptica de Edwards (Ed448) , para realizar a autenticação dos agentes**

In [11]:

```
# geracao da chave privada
def gen_edPrivateKey(self):
    self.Ed_private_key = Ed448PrivateKey.generate()

# geracao da chave publica a partir da chave privada
def gen_edPublicKey(self):
    self.Ed_public_key = self.Ed_private_key.public_key()

# assinatura com a chave privada
def sign_edPrivateKey(self):
    self.sign = self.Ed_private_key.sign(self.sign)
```

- **Geração da chave privada, pública e partilhada utilizada na autenticação por troca de chaves X448 com o receiver, utilizando KDF**

In [12]:

```
# geracao da chave privada
def gen_XprivateKey(self):
    self.X_private_key = X448PrivateKey.generate()

# geracao da chave publica do emitter
def gen_XpublicKey(self):
    self.X_public_key = self.X_private_key.public_key()
# gera chave partilhada de sua chave privada misturada com a chave publica do receiver
def gen_XsharedKey(self, recPublickey): # esta public key é referente ao receiver
    key = self.X_private_key.exchange(recPublickey)

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

- **Para verificar o acordo entre as chaves, cifrando a chave, de forma que apenas o receiver consiga verificar a chave**

In [13]:

```
def agree_key(self):
        nonce = os.urandom(16)
        algorithm = algorithms.ChaCha20(self.X_shared_key, nonce)
        cifra = Cipher(algorithm, mode=None).encryptor().update(self.X_shared_key)
        key = nonce + cifra
        return key
```

- **Agora passando para a parte de cifrar a mensagem , foi criado uma função auxiliar para gerar os `tweaks`. Foi estruturado o tweak seguindo o Capítulo 1 proposto , utilizando 16 bits, sendo os 8 iniciais para o `nonce` , 7 para o `contador` (caso a tag = 0) ou `tamanho` da mensagem (caso tag = 1) e por fim o último bit para a `tag` (0 caso seja os blocos intermediários, ou 1 caso seja o último bloco).**

In [ ]:

```
def gen_tweak(self, tag, count):
        #tweak = 8 bytes de nonce + 7 de contador + 1 de tag
        nonce = os.urandom(8)
        return nonce + count.to_bytes(7,byteorder = 'big') + tag.to_bytes(1,byteorder =
'big')
```

- **Passando para a encriptação, segue 3 passos , `padding` que coloca bits a mais na mensagem necessário para a divisão dos blocos, `cifragem` utilizando o TPBC que utiliza duas chaves para encriptação (a chave compartilhada que é fixa e o tweak que varia a cada bloco), e no final é acrescentado bits para a `autenticação` da mensagem.**

In [14]:

```
def cipher(self,message):
    # Guardar o tamanho da mensagem
    msg_tam = len(message)
    # Fazer o padding da mensagem
    padder = padding.PKCS7(64).padder()
    padded = padder.update(message) + padder.finalize()
    padded_size = len(padded)
    criptograma = b''
    count = 0
    #Dividir em blocos de 16 a msg
    for i in range(0,padded_size,16):
        p=padded[i:i+16]
```

```
        # Os primeiros blocos cifrados com a TPBC controlada por uma chave k mas com tweaks w distintos
        if (padded_size>i+16+1):
            #Blocos intermédios com tag 0
            w = self.gen_tweak(0,count)
            #cifra com AES256, no modo de tweaks (XTS)
            cipher = Cipher(algorithms.AES(self.X_shared_key), mode=modes.XTS(w))
            encryptor = cipher.encryptor().update(p)
            criptograma += w + encryptor
        #O último bloco como um XOR de uma máscara gerada
        else:
            #Ultimo bloco com tag 1
            w = self.generate_tweak(1,msg_tam)
            criptograma += w
            mid = b''
            for index, byte in enumerate(p):
                #aplicar a máscara XOR aos blocos . Esta mascara é compostas pela shared_key + tweak
                mask = self.X_shared_key + w
                mid += bytes([byte ^ mask[0:16][0]])
            criptograma += mid

        count += 1


    #Adicionalmente é enviada uma secção de autenticação para verificação antes de decifrar a mensagem
    h = hmac.HMAC(self.X_shared_key, hashes.SHA256(), backend=default_backend())
    h.update(criptograma)
    self.mac = h.finalize()
    ciphertext = self.mac + criptograma
    return ciphertext
```

Receiver

Agora iremos analisar o agente receiver que irá verificar a autenticação das chaves e assinatura , e depois irá decifrar o criptograma

In [17]:

```
class receiver:
    X_private_key = b""
    X_public_key = b""
    X_shared_key = b""
    tweak = b""
    sign = b"Signing Message"
```

- **Da mesma forma que foi realizado no emitter , é gerado as chaves X448, e chave partilhada vinda do emitter**

In [16]:

```
# geracao da chave privada
def gen_XprivateKey(self):
    self.X_private_key = X448PrivateKey.generate()

# geracao da chave publica do emitter
def gen_XpublicKey(self):
    self.X_public_key = self.X_private_key.public_key()
# gera chave partilhada de sua chave privada misturada com a chave publica do receiver
def gen_XsharedKey(self, emiPublickey): # esta public key é referente ao emitter
    key = self.X_private_key.exchange(emiPublickey)

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

- **O receiver recebe a chave pública do emitter e confere se assinatura é a correspondente**

In [ ]:

```
def check_EdSign(self, signature, public_key):
        try:
            public_key.verify(signature, self.sign)
        except:
            raiseExceptions("A Autenticação falhou!")
```

- **Verificar se o acordo entre as chaves foi realizado com sucesso**

In [19]:

```
def check_key(self, cryptog):
        nonce = cryptog[0:16]
        key = cryptog[16:]
        algorithm = algorithms.ChaCha20(self.X_shared_key, nonce)
        cipher = Cipher(algorithm, mode=None)
        decryptor = cipher.decryptor()
        key = decryptor.update(key)
        #Se corresponder à chave partilhada :
        if key == self.X_shared_key:
            print("\nAs chaves foram autenticadas com sucesso\n")
        else:
            raiseExceptions("Houve erro durante a verificação das chaves")
```

- **Verificar se assinatura do criptograma é igual ao que foi acordado**

In [20]:

```
def check_MacAuth(self, cptext, signature):
        h = hmac.HMAC(self.X_shared_key, hashes.SHA256())

        h.update(cptext)

        r = True
        if not h.verify(signature):
            r = False
        return r
```

- **Para decifrar o criptograma é necessário retirar os componentes do tweak que foram utilizados na cifragem**

In [ ]:

```
def recov_tweak(self, tweak):
        count = int.from_bytes(tweak[8:15], byteorder = 'big')
        tag = tweak[15]
        return count, tag
```

- **Seguindo a mesma lógica na cifragem , para gerar o `cleartext` segue os passos de : checar `autenticidade` da mensagem, `decifrar` e `unpadding`.**

In [51]:

```
def decipher(self, criptograma):
        # primeiros bytes de autenticação MAC
        mac = criptograma[0:32]
        # o resto é o criptograma
        cpt = criptograma[32:]
        if(self.check_MacAuth(cpt,mac)==False):
            raiseExceptions("Erro com autenticação MAC!")
            return
```

```
        plaintext = b''
        msgBlock = b''


        tweak = cpt[0:16]
        block = cpt[16:32]
        i = 1
        count, tag = self.recov_tweak(tweak)
        while(tag!=1):
            cipher = Cipher(algorithms.AES(self.X_shared_key), mode=modes.XTS(tweak))
            decryptor = cipher.decryptor()
            msgBlock = decryptor.update(block)
            plaintext += msgBlock
            tweak = cpt[i*32:i*32 +16]
            block = cpt[i*32 +16:(i+1)*32]
            count, tag = self.degenerate_tweak(tweak)
            i+= 1
        if (tag == 1):
            lastBlock =b''
            for _, byte in enumerate(block):
                #máscaras XOR
                mask = self.X_shared_key + tweak
                lastBlock += bytes([byte ^ mask[0:16][0]])
            plaintext += lastBlock

        # Fazer unpadding da msg
        unpadder = padding.PKCS7(64).unpadder()
        unpadded_message = unpadder.update(plaintext) + unpadder.finalize()

        # O último "contador" vai ser o comprimento da mensagem decifrada , então verifi
car se não houve perdas
        if (len(unpadded_message.decode("utf-8")) == count):
            print("Sucesso na decifra")
            return unpadded_message.decode("utf-8")
        else: raiseExceptions("Houve erros na decifra")
```

- **O algoritmo todo da classe emitter**

In [15]:

```
class emitter:
    message = b"Hello World" # the message to be transmited
    sign = b"Signing Message" # assinatura
    mac = b""

    X_private_key = b""
    X_public_key = b""
    X_shared_key = b""

    Ed_private_key = b""
    Ed_public_key = b""
    Ed_signature = b""

    # geracao da chave privada
    def gen_edPrivateKey(self):
        self.Ed_private_key = Ed448PrivateKey.generate()

    # geracao da chave publica a partir da chave privada
    def gen_edPublicKey(self):
        self.Ed_public_key = self.Ed_private_key.public_key()

    # assinatura com a chave privada
    def sign_edPrivateKey(self):
        self.sign = self.Ed_private_key.sign(self.sign)


    # geracao da chave privada
    def gen_XprivateKey(self):
        self.X_private_key = X448PrivateKey.generate()
```

```
    # geracao da chave publica do emitter
    def gen_XpublicKey(self):
        self.X_public_key = self.X_private_key.public_key()
    # gera chave partilhada de sua chave privada misturada com a chave publica do receive
r
    def gen_XsharedKey(self, recPublickey): # esta public key é referente ao receiver
        key = self.X_private_key.exchange(recPublickey)

        self.X_shared_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'this is handshake',
        ).derive(key)

    def agree_key(self):
        nonce = os.urandom(16)
        algorithm = algorithms.ChaCha20(self.X_shared_key, nonce)
        cifra = Cipher(algorithm, mode=None).encryptor().update(self.X_shared_key)
        key = nonce + cifra
        return key


    def gen_tweak(self, tag, count):
        #tweak = 8 bytes de nonce + 7 de contador + 1 de tag
        nonce = os.urandom(8)
        return nonce + count.to_bytes(7,byteorder = 'big') + tag.to_bytes(1,byteorder =
'big')


    def cipher(self,message):
    # Guardar o tamanho da mensagem
        msg_tam = len(message)
        # Fazer o padding da mensagem
        padder = padding.PKCS7(64).padder()
        padded = padder.update(message) + padder.finalize()
        padded_size = len(padded)
        criptograma = b''
        count = 0
        #Dividir em blocos de 16 a msg
        for i in range(0,padded_size,16):
            p=padded[i:i+16]
            # Os primeiros blocos cifrados com a TPBC controlada por uma chave k mas com
tweaks w distintos
            if (padded_size>i+16+1):
                #Blocos intermédios com tag 0
                w = self.gen_tweak(0,count)
                #cifra com AES256, no modo de tweaks (XTS)
                cipher = Cipher(algorithms.AES(self.X_shared_key), mode=modes.XTS(w))
                encryptor = cipher.encryptor().update(p)
                criptograma += w + encryptor
            #O último bloco como um XOR de uma máscara gerada
            else:
                #Ultimo bloco com tag 1
                w = self.gen_tweak(1,msg_tam)
                criptograma += w
                mid = b''
                for index, byte in enumerate(p):
                    #aplicar a máscara XOR aos blocos . Esta mascara é compostas pela sh
ared_key + tweak
                    mask = self.X_shared_key + w
                    mid += bytes([byte ^ mask[0:16][0]])
                criptograma += mid

            count += 1


        #Adicionalmente é enviada uma secção de autenticação para verificação antes de de
cifrar a mensagem
        h = hmac.HMAC(self.X_shared_key, hashes.SHA256(), backend=default_backend())
        h.update(criptograma)
        self.mac = h.finalize()
```

```
        ciphertext = self.mac + criptograma
        return ciphertext
```

- **O algoritmo completo da classe receiver**

In [16]:

```
class receiver:
    X_private_key = b""
    X_public_key = b""
    X_shared_key = b""
    tweak = b""
    sign = b"Signing Message"

        # geracao da chave privada
    def gen_XprivateKey(self):
        self.X_private_key = X448PrivateKey.generate()

    # geracao da chave publica do emitter
    def gen_XpublicKey(self):
        self.X_public_key = self.X_private_key.public_key()
    # gera chave partilhada de sua chave privada misturada com a chave publica do receive
r
    def gen_XsharedKey(self, emiPublickey): # emitter public key
        key = self.X_private_key.exchange(emiPublickey)

        self.X_shared_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'this is handshake',
        ).derive(key)

    def check_EdSign(self, signature, public_key):
            try:
                public_key.verify(signature, self.sign)
            except:
                raiseExceptions("A Autenticação falhou!")

    def check_key(self, cryptog):
            nonce = cryptog[0:16]
            key = cryptog[16:]
            algorithm = algorithms.ChaCha20(self.X_shared_key, nonce)
            cipher = Cipher(algorithm, mode=None)
            decryptor = cipher.decryptor()
            key = decryptor.update(key)
            #Se corresponder à chave partilhada :
            if key == self.X_shared_key:
                print("\nAs chaves foram autenticadas com sucesso seguindo X448\n")
            else:
                raiseExceptions("Houve erro durante a verificação das chaves")

    def check_MacAuth(self, cptext, signature):
            h = hmac.HMAC(self.X_shared_key, hashes.SHA256())

            h.update(cptext)

            r = True
            if not h.verify(signature):
                r = False
            return r

    def recov_tweak(self, tweak):
            count = int.from_bytes(tweak[8:15], byteorder = 'big')
            tag = tweak[15]
            return count, tag

    def decipher(self, criptograma):
            # primeiros bytes de autenticação MAC
            mac = criptograma[0:32]
```

```
            # o resto é o criptograma
            cpt = criptograma[32:]

            plaintext = b''
            msgBlock = b''

            try:
                self.check_MacAuth(cpt,mac)
            except:
                raiseExceptions("Erro com autenticação MAC!")
                return

            tweak = cpt[0:16]
            block = cpt[16:32]
            i = 1
            count, tag = self.recov_tweak(tweak)
            while(tag!=1):
                cipher = Cipher(algorithms.AES(self.X_shared_key), mode=modes.XTS(tweak)
)
                decryptor = cipher.decryptor()
                msgBlock = decryptor.update(block)
                plaintext += msgBlock
                tweak = cpt[i*32:i*32 +16]
                block = cpt[i*32 +16:(i+1)*32]
                count, tag = self.recov_tweak(tweak)
                i+= 1
            if (tag == 1):
                lastBlock =b''
                for _, byte in enumerate(block):
                    #máscaras XOR
                    mask = self.X_shared_key + tweak
                    lastBlock += bytes([byte ^ mask[0:16][0]])
                plaintext += lastBlock

            # Fazer unpadding da msg
            unpadder = padding.PKCS7(64).unpadder()
            unpadded_message = unpadder.update(plaintext) + unpadder.finalize()

            # O último "contador" vai ser o comprimento da mensagem decifrada , então ve
rificar se não houve perdas
            if (len(unpadded_message.decode("utf-8")) == count):
                print("Sucesso na decifra")
                return unpadded_message.decode("utf-8")
            else:
                raiseExceptions("Houve erros na decifra")
                return
```

- **Agora foi realizado testes na comunicação entre o receiver e o emitter**

In [17]:

```
def x448_init(emitter,receiver):
    emitter.gen_XprivateKey()
    receiver.gen_XprivateKey()
    emitter.gen_XpublicKey()
    receiver.gen_XpublicKey()

def ed_init(emitter):
    emitter.gen_edPrivateKey()
    emitter.sign_edPrivateKey()
    emitter.gen_edPublicKey()


def gen_sharedKey(emitter,receiver):
    emitter.gen_XsharedKey(receiver.X_public_key)
    receiver.gen_XsharedKey(emitter.X_public_key)

msg = input("Introduza a mensagem para cifrar:")
emitter = emitter()
receiver = receiver()
```

```
ed_init(emitter)
receiver.check_EdSign(emitter.sign, emitter.Ed_public_key)
x448_init(emitter,receiver)
gen_sharedKey(emitter,receiver)

# Verificar se as chaves foram tão autenticadas seguindo X448
emiKey= emitter.agree_key()
receiver.check_key(emiKey)

ciphertext = emitter.cipher(msg.encode('utf-8'))
plaintext = receiver.decipher(ciphertext)
print("Criptograma: \n" , ciphertext)
print("Mensagem : \n" , plaintext)
```

```
As chaves foram autenticadas com sucesso seguindo X448

Sucesso na decifra
Criptograma:
 b'\xf1[?\xa9h\xc4<\xb7\xe5\xf8\xd6\x17f\xd9\xb72 \xf8\xe2\xf9\xa2\xea\x1a\xf3\x08\x05\xe
0\xd7\xcc)\xda\x19\x94\xd5^q\x00\xa7\xc3k\x00\x00\x00\x00\x00\x00\x0c\x01\xab\xa8\xa5\xe4
\xa3\xb6\xb1\xb4\xab\xe4\xf5\xf1\xc0\xc0\xc0\xc0'
Mensagem :
 ola grupo 15
```