TP1 | Exercício 2 | Grupo 15

Pedro Araújo - pg50684

Nuno Dias Mata - Pg44420

In [1]:

```
import os
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
import cryptography.exceptions
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives.asymmetric import dsa
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
```

Emitter

Iremos começar por falar da classe emitter e suas funcionalidades

In [2]:

```
class emitter:
    seed = b''
    sign = b'Group15 signature'
```

- **Primeiramente é necessário gerar a "seed" do gerador de chave utilizando um KDF , escolheu-se o PBKDF2HMAC, a partir de uma password**

In [3]:

```
def gen_seed(self, password):
        #gerar a seed para o PRG usando KDF
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=os.urandom(16),
            iterations=310000,
            )
        self.seed = kdf.derive(bytes(password, 'utf-8'))
```

- **Após gerado a seed, é possíve criar o gerador pseudo-aleatório utilizando o SHAKE256 para gerar 2^N palavras de 64 bits (8bytes)**

In [4]:

```
def gen_prg(self,N):
    msgaux = hashes.Hash(hashes.SHAKE256(8 * pow(2,N)))
    msgaux.update(self.seed)
    return msgaux.finalize()
```

- **Após gerado a chave pseudo aleatória , é possível realizar a cifra que é a implementação do "One time Pad", e conta com 3 fases : padding, fazer XOR aos blocos e acrescentar a autenticação da mensagem ao final**

In [5]:

```
def cipher(self,prgMsg,msg):
```

```
    # Fazer o padding
    padder = padding.PKCS7(64).padder()
    padded = padder.update(msg) + padder.finalize()
    cipherText = b''
    for i in range (0,len(padded),8):
        m = padded[i:i+8]
        for index, block in enumerate(m):
            #fazer XOR aos blocos
            cipherText += bytes([block ^ prgMsg[i*8:(i+1)*8][index]])

    # Incluir a autenticação na mensagem
    hmac = hmac.HMAC(prgMsg, hashes.SHA256())
    hmac.update(self.sign)
    return hmac.finalize() + cipherText
```

Receiver

Após analisado as funcionalidade da classe Emitter , iremos verificar a classe receiver desnvolvida

In [6]:

```
class receiver:
    sign = b'Group15 signature'
```

- **Como foi realizado e incluido a autenticação no criptograma, é necessário o receiver verificar esta autenticidade na mensagem**

In [7]:

```
def check_MacAuth(self,prg, message):
        h = hmac.HMAC(prg, hashes.SHA256())
        h.update(message)
        try:
            h.verify(self.sign)
            return True
        except cryptography.exceptions.InvalidSignature:
            return False
```

- **Já no processo de decifrar o criptograma , é primeiro verificado a autenticidade da mensagem, depois passa para fase de decifragem e depois realizar o unpadding daquilo que foi adicionado anteriormente**

In [8]:

```
def decipher(self, prg, ciphertext):

        mac = ciphertext[:32]
        try:
            self.check_MacAuth(prg,mac)
        except:
            print("Erro com autenticação MAC!")
            return

        criptog = ciphertext[32:]
        plaintext = b''

        for i in range (0,len(criptog),8):
            msgBlock = criptog[i:i+8]
            for ind, block in enumerate(msgBlock):
                plaintext += bytes([block ^ prg[i*8:(i+1)*8][ind]])

        #fazer o unpadd para remover o padd acrescentado
        unpadder = padding.PKCS7(64).unpadder()
        unpadded = unpadder.update(plaintext) + unpadder.finalize()
        return unpadded.decode('utf-8')
```

Código completo do Emitter

In [9]:

```
class emitter:
    seed = b''
    sign = b'Group15 signature'

    def gen_seed(self, password):
        #gerar a seed para o PRG usando KDF
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=os.urandom(16),
            iterations=310000,
            )
        self.seed = kdf.derive(bytes(password, 'utf-8'))

    def gen_prg(self,N):
        msgaux = hashes.Hash(hashes.SHAKE256(8 * pow(2,N)))
        msgaux.update(self.seed)
        return msgaux.finalize()

    def cipher(self,prgMsg,msg):
        # Fazer o padding
        padder = padding.PKCS7(64).padder()
        padded = padder.update(msg) + padder.finalize()
        cipherText = b''
        for i in range (0,len(padded),8):
            m = padded[i:i+8]
            for index, block in enumerate(m):
                #fazer XOR aos blocos
                cipherText += bytes([block ^ prgMsg[i*8:(i+1)*8][index]])

        mac = hmac.HMAC(prgMsg, hashes.SHA256())
        mac.update(self.sign)
        return mac.finalize() + cipherText
```

Código completo do Receiver

In [10]:

```
class receiver:
    sign = b'Group15 signature'

    def check_MacAuth(self,prg, message):
        h = hmac.HMAC(prg, hashes.SHA256())
        h.update(message)
        try:
            h.verify(self.sign)
            return True
        except cryptography.exceptions.InvalidSignature:
            return False

    def decipher(self, prg, ciphertext):

        mac = ciphertext[:32]
        try:
            self.check_MacAuth(prg,mac)
        except:
            print("Erro com autenticação MAC!")
            return

        criptog = ciphertext[32:]
        plaintext = b''

        for i in range (0,len(criptog),8):
            msgBlock = criptog[i:i+8]
            for ind, block in enumerate(msgBlock):
                plaintext += bytes([block ^ prg[i*8:(i+1)*8][ind]])
```

```
        #fazer o unpadd para remover o padd acrescentado
        unpadder = padding.PKCS7(64).unpadder()
        unpadded = unpadder.update(plaintext) + unpadder.finalize()
        return unpadded.decode('utf-8')
```

Testes realizado nas classes

In [11]:

```
msg = input("Introduza a mensagem para cifrar:")
n = input("Valor de N:")
password = input("Password:")

print("Mensagem a ser cifrada: ", msg)
print("Valor de N escolhido: ", n)
emitter = emitter()
receiver = receiver()

emitter.gen_seed(password)
prg = emitter.gen_prg(int(n))

ciphertext = emitter.cipher(prg,msg.encode('utf-8'))
cleartext = receiver.decipher(prg,ciphertext)

print("Cipher text: ", ciphertext)
print("Clear text: ", cleartext)
```

```
Mensagem a ser cifrada:  Oi grupo 15
Valor de N escolhido:  10
Cipher text:  b'\x12G\x1d\x8e&\x994\xb3\xfdD\xa0\x19\xcc\x0f\xb9D\x81]\xb6\xf2!\xd5\x1c.\
x97\xc9,qt\xd7\xed0\x85\x1cP\xa9D\xc3\x13d\x1b\x9b\x96\x0e\x91\xf0$\xfc'
Clear text:  Oi grupo 15
```

In [ ]:

```

```