Estruturas Criptográficas 2022/23

TP1. Problema 2

Grupo 7. Leonardo Berteotti e Paulo R. Pereira

Pretende-se criar uma cifra com autenticação de meta-dados a partir de um gerador pseudo-aleatório do tipo XOF. Este gerador, que tem como "seed" uma chave gerada por um **KDF** a partir de uma password fornecida, produzirá uma sequência de palavras ("outputs") de 64 bits. A mensagem será cifrada por blocos, i.e. será numa primeira fase dividida, e cada bloco será cifrado com o respetivo "output" gerado pelo gerador. Uma vez cifrada a mensagem, esta juntamente com os meta-dados serão autenticados utilizando a própria "seed" do gerador (a chave obtida pela **KDF**).

In [24]:

```
import os
from pickle import dumps
from cryptography.hazmat.primitives import hashes, hmac, padding
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
```

A chave é gerada utilizando uma primitiva **KDF**, nomeadamente **PBKDF2HMAC**. Para tal, é usada uma *password* inserida pelo utilizador para a derivar. A chave resultante de 32 bytes funciona como *seed* do gerador pseudo-aleatório do tipo XOF.

Sendo um PRG do tipo XOF, o algoritmo de *hash* utilizado foi o **SHAKE256**. O gerador tem um limite de 2^n palavras de 8 bytes (64 bits). Assim, o gerador deverá gerar $2^n * 8$ bytes. O parâmetro n é também inserido pelo utilizador.

In [25]:

```
def generate_cipher_key(password):
    kdf = PBKDF2HMAC(
        algorithm = hashes.SHA256(),
        length = 32,
        salt = os.urandom(16),
        iterations = 480000,
    )
    return kdf.derive(password)

def generator(seed, n):
    digest = hashes.Hash(hashes.SHAKE256(2**n * 8))
    digest.update(seed)
    return digest.finalize()
```

Cifragem e Decifragem

A função *encrypt*, responsável por cifrar a mensagem, começa por dividi-la em blocos de 64 bits. Note-se que a função já recebe o *output* do gerador em blocos de 64 bits. É também verificado se o último bloco têm o tamanho necessário (64 bits), e caso não tenha, é efetuado o *padding*. A *flag* que indica se o *padding* foi efetuado ou não é guardada e enviada juntamente com o *ciphertext* para posterior *unpadding*. De seguida, cada bloco da mensagem (*plaintext*) é cifrado utilizando um *output* do gerador como máscara XOR, com recurso ao operador ^ do Python.

De modo a garantir a autenticidade do criptograma, é gerado um **MAC** que identifica unicamente o *ciphertext*. Este *tag* é gerado a partir de uma chave "acordada" pelos agentes, de nome *auth_key*.

A função *decrypt* faz o processo inverso à função *encrypt*. Começa por verificar a autenticação do criptograma com recurso ao *tag* recebido, decifra com a mesma lógica através das máscaras XOR, e, caso o último bloco tenha sido submetido ao processo de *padding*, efetua o *unpadding*.

In [26]:

```
auth_key = b'accorded authentication key'

def encrypt(plaintext, outputs, ad):
    blocks = []
    pad = False
    for i in range(0, len(plaintext), 8):
        block = plaintext[i:i+8]
        # padding
        if len(block) < 8:
            pad = True
            padder = padding.PKCS7(64).padder()
            padded_block = padder.update(block)
            padded_block += padder.finalize()
            blocks.append(padded_block)
        else:
            blocks.append(block)

    ciphertext = bytearray()
    for block, output in zip(blocks, outputs):
        for b, o in zip(block, output):
            ciphertext.append(b ^ o)

    h = hmac.HMAC(auth_key, hashes.SHA256())
    h.update(ciphertext)
    h.update(ad)

    return (ciphertext, h.finalize(), pad)


def decrypt(ciphertext, outputs, signature, ad, pad):
    h = hmac.HMAC(auth_key, hashes.SHA256())
    h.update(ciphertext)
    h.update(ad)

    try:
        h.verify(signature)
    except:
        print('ERROR --- Different keys used')
    else:
        # decrypt ciphertext
        blocks = [ciphertext[i:i+8] for i in range(0, len(ciphertext), 8)]

        plaintext = bytearray()
        for block, output in zip(blocks, outputs):
            for b, o in zip(block, output):
                plaintext.append(b ^ o)

        plaintext_blocks = [plaintext[i:i+8] for i in range(0, len(plaintext), 8)]
        # check if block was padded or not
        if pad:
            unpadder = padding.PKCS7(64).unpadder()
            unpadded_block = unpadder.update(plaintext_blocks[-1])
            plaintext_blocks[-1] = unpadded_block + unpadder.finalize()

        message = ''.join([ba.decode() for ba in plaintext_blocks])
        return message
```

In [27]:

```
def run():

    message = input('Enter the message: ')
    print('Original message:\n' + message)
    n = int(input('Enter the n: '))
    password = input('Enter the password: ').encode('utf-8')

    cipher_key = generate_cipher_key(password)

    words = generator(cipher_key, n)
    outputs = [words[i:i+8] for i in range(0, len(words), 8)]

    ad = os.urandom(16)
    crypto = encrypt(bytes(message, 'utf-8'), outputs, ad)

    ciphertext = crypto[0]
    print("\nCiphertext:")
    print(bytes(ciphertext).decode('unicode_escape'))
    signature = crypto[1]

    pad = crypto[2]
    final_message = decrypt(ciphertext, outputs, signature, ad, pad)

    print("\nDecrypted message:")
    print(final_message)

run()
```

```
Original message:
hello there :)

Ciphertext:
e·3 õF ´C |ZÓê

Decrypted message:
hello there :)
```