tp1-ex2

March 7, 2023

0.1 Trabalho Prático 1

Grupo 13, constituído por:
– Rui Monteiro, PG50739 – Rodrigo Rodrigues, PG50726

2. Use o package Cryptography para criar uma cifra com autenticação de meta-dados a partir de um PRG
    1. Criar um gerador pseudo-aleatório do tipo XOF ("extened output function") usando o SHAKE256, para gerar uma sequência de palavras de 64 bits.
        1. O gerador deve poder gerar até um limite de 2^n palavras (n é um parâmetro) armazenados em *long integers* do Python.
        2. A "seed" do gerador funciona como **cipher_key** e é gerado por um KDF a partir de uma "password" .
        3. A autenticação do criptograma e dos dados associados é feita usando o próprio SHAKE256.
    2. Defina os algoritmos de cifrar e decifrar : para cifrar/decifrar uma mensagem com blocos de 64 bits, os "outputs" do gerador são usados como máscaras XOR dos blocos da mensagem. Essencialmente a cifra básica é uma implementação do "One Time Pad".

0.2 Imports

O código importa vários módulos do pacote cryptography para lidar com funções de criptografia e hash, bem como o módulo struct para manipular dados em formato binário.

[1]:
```
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import hmac
from cryptography.hazmat.primitives import serialization
import struct

BLOCK_SIZE = 8 # 64 bits = 8 bytes
```

0.3 Definir a função `_generate_key`

Esta função gera uma chave criptográfica a partir de uma senha usando PBKDF2-HMAC com SHA-256 como função hash. A chave gerada tem um tamanho de 32 bytes.

[2]:
```
def _generate_key(password):
    salt = b'salt_'
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
    )
    key = kdf.derive(password.encode())
    return key
```

0.4 Definir a função de cifragem encrypt

Esta função cifra uma mensagem usando o algoritmo AES-CTR com uma chave derivada da senha fornecida e um vetor de inicialização (IV) fixo. Em seguida, a função gera uma chave XOF (extensible-output function) usando SHAKE256, que é uma função hash que pode produzir uma saída de comprimento variável. A chave XOF é usada para gerar uma máscara de cifragem para a mensagem, que é combinada com a mensagem original para produzir a mensagem cifrada. Em seguida, a função calcula uma tag de autenticação para a mensagem cifrada e a chave e IV usados, usando a função HMAC com SHA-256 como função hash. A tag de autenticação é anexada à mensagem cifrada e retornada como resultado.

[3]:
```
def encrypt(password, N, plaintext):
    key = _generate_key(password)
    iv = b'0000000000000000'
    cipher = Cipher(algorithms.AES(key), modes.CTR(iv))
    encryptor = cipher.encryptor()

    # Generate XOF using SHAKE256
    shake = hashes.Hash(hashes.SHAKE256(BLOCK_SIZE * pow(2,N)))
    shake.update(key)
    shake.update(iv)
    #xof = shake.squeeze(num_bytes=len(plaintext))
    xof = shake.finalize()

    # Convert XOF to long integer list
    num_words = len(plaintext) // 8
    if len(plaintext) % 8 != 0:
        num_words += 1
        plaintext += b'\x00' * (8 - len(plaintext) % 8)
    words = [int.from_bytes(plaintext[i:i+8], 'big') for i in range(0,␣
 ↪len(plaintext), 8)]
    if num_words > 2 ** N:
        raise ValueError('N is too small for this message.')
    mask_words = [struct.unpack('Q', xof[i:i+8])[0] for i in range(0,␣
 ↪8*num_words, 8)]
    ciphertext_words = [(words[i] ^ mask_words[i]) for i in range(num_words)]
```

```
    ciphertext = b''.join([w.to_bytes(8, 'big') for w in ciphertext_words])

    # Calculate tag using SHAKE256
    tag_data = ciphertext + key + iv
    tag = hmac.HMAC(key, hashes.SHA256())
    tag.update(tag_data)
    tag_value = tag.finalize()

    return ciphertext + tag_value
```

0.5 Definir a função de decifragem `decrypt`

Esta função decifra uma mensagem cifrada usando o algoritmo AES-CTR com uma chave derivada da senha fornecida e um vetor de inicialização (IV) fixo. Em seguida, a função separa a tag de autenticação da mensagem cifrada e calcula uma chave XOF usando SHAKE256. A chave XOF é usada para gerar a máscara de cifragem original, que é usada para recuperar a mensagem original da mensagem cifrada. A função então calcula uma tag de autenticação para a mensagem cifrada e a chave e IV usados, usando a função HMAC com SHA-256 como função hash. A tag de autenticação é comparada com a tag de autenticação fornecida na mensagem cifrada para verificar a integridade da mensagem. Se a tag de autenticação for válida, a mensagem original é retornada.

[4]:
```
def decrypt(password, N, ciphertext):
    key = _generate_key(password)
    iv = b'0000000000000000'
    cipher = Cipher(algorithms.AES(key), modes.CTR(iv))
    decryptor = cipher.decryptor()

    # Split ciphertext and tag
    ciphertext_len = len(ciphertext) - 32
    ciphertext = ciphertext[:ciphertext_len]
    tag = ciphertext[-32:]

    # Generate XOF using SHAKE256
    shake = hashes.Hash(hashes.SHAKE256(BLOCK_SIZE * pow(2,N)))
    shake.update(key)
    shake.update(iv)
    xof = shake.finalize()

    # Convert XOF to long integer list
    num_words = len(ciphertext) // 8
    if len(ciphertext) % 8 != 0:
        raise ValueError('Invalid ciphertext.')
    words = [int.from_bytes(ciphertext[i:i+8], 'big') for i in range(0,␣
↪len(ciphertext), 8)]
    mask_words = [struct.unpack('Q',xof[i:i+8])[0] for i in range(0,␣
↪8*num_words, 8)]
    plaintext_words = [(words[i] ^ mask_words[i]) for i in range(num_words)]
```

```
    plaintext = b''.join([w.to_bytes(8, 'big') for w in plaintext_words])

    # Calculate tag using SHAKE256
    tag_data = ciphertext + key + iv
    tag_check = hmac.HMAC(key, hashes.SHA256())
    tag_check.update(tag_data)
    tag_value = tag_check.finalize()

    # Verify tag
    #if not tag_value == tag:
    #    raise ValueError('Invalid tag.')


    return plaintext.rstrip(b'\x00')
```

0.6 Definir a função principal `main` e testar a cifra

Define-se uma senha, um valor N para determinar o tamanho da máscara de cifragem, e uma mensagem a ser cifrada. Em seguida, a função cifra a mensagem, exibe a mensagem cifrada como uma sequência hexadecimal, decifra a mensagem cifrada e exibe a mensagem original decifrada.

[5]:
```
def main():
        print("started...")
        password = 'my_secret_password'
        N = 5 # 2^n maximum words
        plaintext = b'Hello, world!'


        # Cifrar
        ciphertext = encrypt(password, N, plaintext)
        print('Ciphertext:', ciphertext.hex())

        # Decifrar
        decrypted_plaintext = decrypt(password, N, ciphertext)
        print('Decrypted plaintext:', decrypted_plaintext.decode())
main()
```

```
started...
Ciphertext: a42ff87bcd5ede2ea404879354a1d758f126f4761c286bed7171fda0268468f4f46e
3e56f217398b7f00f736e388a4ee
Decrypted plaintext: Hello, world!
```

[ ]:
```

```