



Universidade do Minho

Departamento de Informática

Criptografia e Segurança em Redes

Engenharia de Telecomunicações e Informática

Trabalho Prático 3

Grupo :

Diogo Araújo A 101778

Fernando Mendes A101263

Junlin Lu A101270

Conteúdo

1.Introdução.....	2
2.Experiência I - Funções de sentido único.....	3
2.1 Códigos.....	3
2.2 Procedimento.....	4
2.3 Conclusão.....	5
3.Experiência III - Cifra por blocos.....	6
3.1 Códigos.....	6
3.2 Procedimento.....	8
3.3 Conclusão.....	9

1.Introdução

O objetivo do Trabalho Prático 3 é consolidar os conhecimentos abordados ao longo do semestre através de experiências práticas envolvendo o desenvolvimento de aplicações que recorrem a diferentes sistemas criptográficos.

Para aprofundar o nosso conhecimento,escolhamos :

Experiência I - Funções de sentido único

Experiência III - Cifra por blocos

Essas duas experiências cobrem aspectos importantes da criptografia

2.Experiência I - Funções de sentido único

Esta experiência é particularmente relevante no contexto atual, em que a segurança das senhas é uma preocupação crescente, especialmente diante dos recentes vazamentos de dados na internet. A capacidade de transformar senhas em um formato seguro utilizando o algoritmo SHA 256. Nesta experiência, nosso objetivo foi buscar códigos no site indicado, calcular o valor de hash de cada código usando SHA256 e, em seguida, comparar esses valores com o hash fornecido no enunciado. O propósito era encontrar o código que correspondesse ao hash dado.

Findings			
All countries		Get the 2019-2022 password list	
RANK	PASSWORD	TIME TO CRACK IT	COUNT
1	123456	< 1 Second	4,524,867
2	admin	< 1 Second	4,008,850
3	12345678	< 1 Second	1,371,152
4	123456789	< 1 Second	1,213,047
5	1234	< 1 Second	969,811
6	12345	< 1 Second	728,414
7	password	< 1 Second	710,321

2.1 Códigos

```
import hashlib

# fichiero
file_path = './cod.txt'
# hash do cod
orig_hash = '96cae35ce8a9b0244178bf28e4966c2ce1b8385723a96a6b838858cdd6ca0a1e'

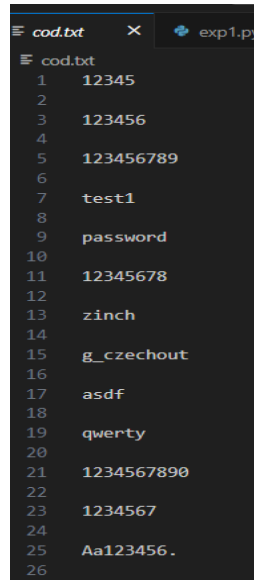
def read_from_file(file_path, orig_hash):
    with open(file_path, 'r') as file:
        for line in file:
            line = line.strip()
            passhash = hashlib.sha256(line.encode()).hexdigest()
            if passhash == orig_hash:
                return line # Return linha se encontrou
    return None # Return None se não existe

found_cod = read_from_file(file_path, orig_hash)

print(found_cod)
```

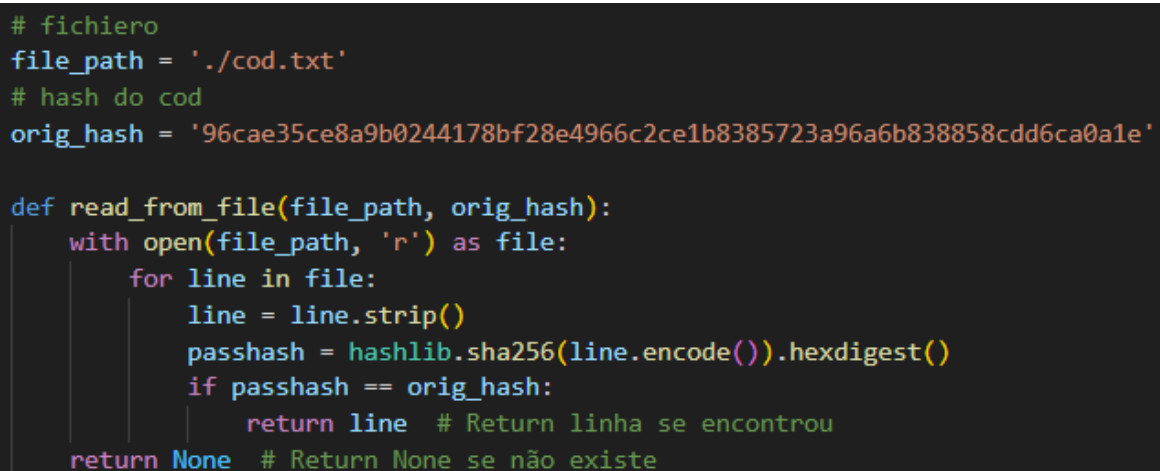
2.2 Procedimento

1- Obtemos uma lista das 200 senhas mais comuns disponíveis publicamente no site NordPass, e guardamos num ficheiro do texto(cod.txt).



```
cod.txt
1 12345
2
3 123456
4
5 123456789
6
7 test1
8
9 password
10
11 12345678
12
13 zinch
14
15 g_czechout
16
17 asdf
18
19 qwerty
20
21 1234567890
22
23 1234567
24
25 Aa123456.
26
```

2- O script lê cada senha do arquivo “cod.txt”, calcula o seu hash e compara com o hash fornecido no enunciado do trabalho. Para cada senha no arquivo cod.txt, o script usa a função hashlib.sha256() para gerar o hash correspondente. O hash calculado é então convertido para uma representação hexadecimal com a funçãohexdigest().



```
# fichiero
file_path = './cod.txt'
# hash do cod
orig_hash = '96cae35ce8a9b0244178bf28e4966c2ce1b8385723a96a6b838858cdd6ca0a1e'

def read_from_file(file_path, orig_hash):
    with open(file_path, 'r') as file:
        for line in file:
            line = line.strip()
            passhash = hashlib.sha256(line.encode()).hexdigest()
            if passhash == orig_hash:
                return line # Return linha se encontrou
    return None # Return None se não existe
```

3- Finalmente, o script retorna a senha que corresponde ao valor hash fornecido. Se nenhuma correspondência for encontrada em todo o arquivo, o script retorna None.

2.3 Conclusão

Ao correr o script é encontrada a senha correspondente, 123123.

```
PS D:\3Ano\3ano\Cryptography\TP3\Exp1> python exp1.py
123123
PS D:\3Ano\3ano\Cryptography\TP3\Exp1> █
```

Também é possível ressaltar a vulnerabilidade das senhas comuns e a facilidade com que podem ser comprometidas, especialmente se os hashes correspondentes são conhecidos e as técnicas como Rainbow table attack.

Para evitar esse tipo de ataque, recomenda-se a utilização de “salt” para aumentar a complexidade da hash, um valor aleatório adicionado à senha antes do hash.

```
case 2:
    encrypted_salt = msg
    self.salt = self.private_key.decrypt(
        encrypted_salt,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
```

```
def encrypt(self, msg: str|bytes) -> str:
    if isinstance(msg, str): msg = msg.encode()
    random_bytes = os.urandom(48)
    iv = random_bytes[:16]
    key = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length = 32,
        salt=self.salt,
        iterations = 1000
    ).derive(random_bytes[16:])
    cipher = AES.new(key, AES.MODE_CBC, iv)
    encrypt_data = random_bytes + cipher.encrypt(pad(msg, AES.block_size))
    return base64.b64encode(encrypt_data).decode()
```

-um exemplo de utilização de salt

3.Experiência III - Cifra por blocos

O AES é uma das duas cifras simétricas escolhidas para os novos protocolos de transporte, nomeadamente o TLSv1.3 (cf. IETF RFC 8446), sendo a sua utilização obrigatória, nomeadamente com tamanho de chave de 128 bits e no modo de operação GCM (i.e., AES128-GCM). O modo de operação GCM (Galois Counter Mode) é cada vez mais utilizado devido a sua performance, e combina o CTR (Counter Mode) com a autenticação de Galois. O resultado do modo de operação GCM é uma sequência de bytes que contém o IV, ciphertext, e uma autenticação tag (utilizada para verificar a autenticidade e integridade da restante sequência de bytes).

O objetivo deste ponto prático é o desenvolvimento de uma aplicação que através da leitura direta da linha de comando e através da utilização do AES-128-GCM (com IV de 12 bytes aleatório e diferente em cada utilização) seja possível cifrar ou decifrar uma mensagem de acordo com as instruções dadas.

3.1 Códigos

```
import argparse
from Crypto.Cipher import AES
from Crypto.Hash import SHA256
import os

def hash_key(key):
    hash_key = SHA256.new(key.encode()).digest() # Usa SHA256 para
    calc hash da chave
    return hash_key

def encrypt_file(key, input_file, output_file):
    nonce = os.urandom(12)
    cipher = AES.new(key, AES.MODE_GCM, nonce=nonce,)

    with open(input_file, 'rb') as f_in:
        plaintext = f_in.read()
        ciphertext, tag = cipher.encrypt_and_digest(plaintext)

    with open(output_file, 'wb') as f_out:
        f_out.write(nonce)
        f_out.write(tag)
        f_out.write(ciphertext)
```

```

def decrypt_file(key, input_file, output_file):
    with open(input_file, 'rb') as f_in:
        nonce = f_in.read(12)
        tag = f_in.read(16)
        ciphertext = f_in.read()

        cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)
        plaintext = cipher.decrypt_and_verify(ciphertext, tag)

    with open(output_file, 'wb') as f_out:
        f_out.write(plaintext)

parser = argparse.ArgumentParser()
parser.add_argument('operacao', choices=['cifra', 'decifra'],
                    help='Operação: cifra ou decifra')
parser.add_argument('chave', help='Chave de criptografia (texto)')
parser.add_argument('input_file', help='Arquivo de entrada')
parser.add_argument('output_file', help='Arquivo de saída')
args = parser.parse_args()

try:
    key = hash_key(args.chave)

    if args.operacao == 'cifra':
        encrypt_file(key, args.input_file, args.output_file)
    elif args.operacao == 'decifra':
        decrypt_file(key, args.input_file, args.output_file)
except Exception as e:
    print(f"Erro: {str(e)}")

#para cifra: python exp2.py cifra <chave> <input_file> <output_file>

#para decifra: python exp2.py decifra <chave> <input_file>
<output_file>

#exemplo: python exp2.py cifra/decifra 12345 a.txt b.txt

```


3.2 Procedimento

1- Começamos com a definição da linha de comando. O `argparse.ArgumentParser` é utilizado para gerenciar os argumentos da linha de comando. Deve especificar a operação (cifra ou decifra), a chave de criptografia (texto) e os nomes dos arquivos de entrada e saída.

```
parser = argparse.ArgumentParser()
parser.add_argument('operacao', choices=['cifra', 'decifra'], help='Operação: cifra ou decifra')
parser.add_argument('chave', help='Chave de criptografia (texto)')
parser.add_argument('input_file', help='Arquivo de entrada')
parser.add_argument('output_file', help='Arquivo de saída')
args = parser.parse_args()
```

2- Recebe uma chave de texto fornecida pelo utilizador e a transforma em um hash seguro usando SHA256. O resultado é uma chave de tamanho fixo adequada para uso na criptografia AES, assim o utilizador pode passar uma chave de tamanho qualquer.

```
def hash_key(key):
    hash_key = SHA256.new(key.encode()).digest() # Usa SHA256 para calc hash da chave
    return hash_key
```

3- Função `encrypt_file` para Cifragem:

Quando a operação de cifra é escolhida, a função `encrypt_file` é chamada.

Um nonce de tamanho 12bytes (IV) é gerado aleatoriamente para garantir a segurança da cifragem.

O arquivo de entrada é lido e cifrado usando AES no modo GCM, e o resultado (texto cifrado e a tag de autenticação) é escrito no arquivo de saída.

Função `decrypt_file` para Decifragem:

Quando a operação de decifra é escolhida, a função `decrypt_file` é utilizada.

O script lê o nonce, a tag de autenticação e o texto cifrado do arquivo de entrada.

Ele então decifra o texto e verifica a autenticidade usando a tag de autenticação.

```
def encrypt_file(key, input_file, output_file):
    nonce = os.urandom(12)
    cipher = AES.new(key, AES.MODE_GCM, nonce=nonce,)

    with open(input_file, 'rb') as f_in:
        plaintext = f_in.read()
        ciphertext, tag = cipher.encrypt_and_digest(plaintext)

    with open(output_file, 'wb') as f_out:
        f_out.write(nonce)
        f_out.write(tag)
        f_out.write(ciphertext)
```

```
def decrypt_file(key, input_file, output_file):
    with open(input_file, 'rb') as f_in:
        nonce = f_in.read(12)
        tag = f_in.read(16)
        ciphertext = f_in.read()


    cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)
    plaintext = cipher.decrypt_and_verify(ciphertext, tag)

    with open(output_file, 'wb') as f_out:
        f_out.write(plaintext)
```

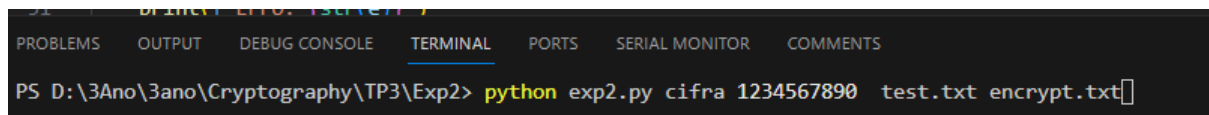
4- Executa a operação escolhida e trata exceções, imprimindo mensagens de erro se algo der errado.

3.3 Conclusão

Preparamos um arquivo txt, e faz cifra através dessa script com chave: 1234567890

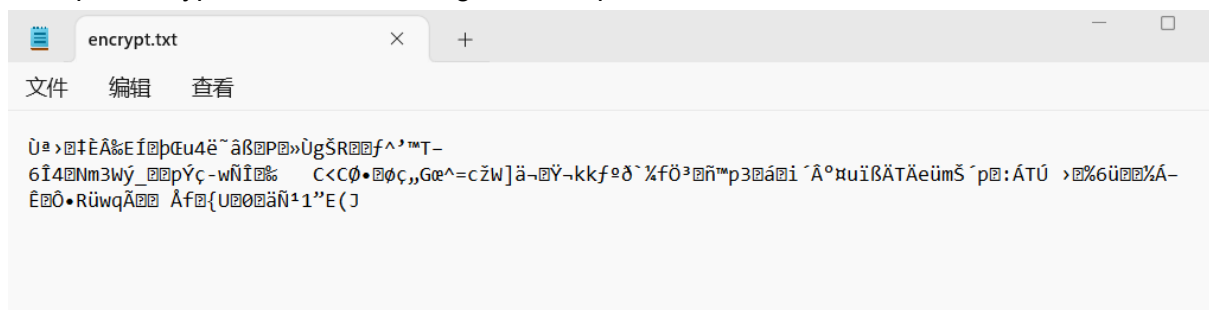


```
test.txt  
文件 编辑 查看  
hello hello hello world world  
je221321@4312  
!! 3432Dfdsdasgfdswedyhtjy
```



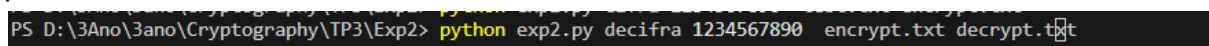
```
PS D:\3Ano\3ano\Cryptography\TP3\Exp2> python exp2.py cifra 1234567890 test.txt encrypt.txt
```

e Output encrypt.txt, assim mensagem do arquivo é cifrado



```
encrypt.txt  
文件 编辑 查看  
Ûa>0iÊÂ%ÊÍ0p0Eu4ë~âß0P0»ÛgŠR00f^'™T-  
6i40Nm3Wÿ_00pÿç-wÑi0% C<C0•00ç,,Gø^=cžw]ä-0ÿ-kkf0ð`%f0³0ñ™p30á0i`Ã°huïßÄTÄeümŠ`p0:ÁTú >0%6ü00%Á-  
Ê00•RüwqÃ00 Áf0{U000äÑ¹1"E(ÿ
```

para decifrar, utiliza a mesma chave, e obtemos ficheiro decifra



```
PS D:\3Ano\3ano\Cryptography\TP3\Exp2> python exp2.py decifra 1234567890 encrypt.txt decrypt.txt
```



```
decrypt.txt  
文件 编辑 查看  
hello hello hello world world world world world world world world world  
dasjiofjidsf  
je221321@4312  
!! 3432Dfdsdasgfdswedyhtjy
```

O resultado do modo de operação GCM é uma sequência de bytes que contém o IV, ciphertext, e uma autenticação tag (utilizada para verificar a autenticidade e integridade da restante sequência de bytes).