

Universidade do Minho

MESTRADO EM ENGENHARIA INFORMÁTICA

**Engenharia de segurança**

**Ficha de exercício 3**

**Grupo 1**

RUI CARLOS AZEVEDO CARVALHO - PG47633

DANIEL BARBOSA MIRANDA - PG47123

ANA LUÍSA LIRA TOMÉ CARNEIRO - PG46983

## Parte V: Funções de sentido único

### Pergunta P1.1

O problema 1 do TP2 consistia em criar um programa que conseguisse cifra e decifrar um ficheiro de texto utilizando a cifra simétrica Chacha20. Para o problema 1 da parte V do TP3, implementou-se as mesmas etapas que no problema 1 do TP passado, contudo foi utilizado uma função de derivação de chaves (KDF) de forma a derivar uma chave de 32 bytes a partir do *input* do utilizador.

A implementação do programa foi realizado em Python e utilizou a biblioteca *PyCryptodome* para a implementação da cifra Chacha20. A implementação inicia-se com a apresentação de um menu ao utilizador, para que este consiga escolher se pretende cifrar ou decifrar o ficheiro.

```
# Opções do menu do utilizador
menu = {
    0: 'SAIR',
    1: 'Cifragem',
    2: 'Decifragem',}

# Menu para determinar se o utilizador pretende
# cifrar ou decifrar uma mensagem
while True:
    for key in menu.keys():
        print(key, '--', menu[key] )
    option = int(input('=> '))
    if option == 1:
        encrypt()
    elif option == 2:
        decrypt()
    elif option == 0:
        exit()
    else:
        print('Opção Inválida.')
```

Caso o utilizador pretenda cifrar um ficheiro é chamada a função *encrypt* que irá perguntar ao utilizador qual o ficheiro a cifrar, chave a ser utilizada pela cifra ChaCha20 e o nome do ficheiro cifrado de output.

É de notar que a chave que realmente será utilizada pela cifra terá de ter um tamanho de 32 bytes (256 bits). Para isso, utilizou-se a função de derivação de chaves *Script* que vai derivar uma chave de comprimento qualquer numa chave de comprimento de 32 bytes. Utilizou-se esta função devido ao facto desta ser bastante difícil de quebrar ou contrário de outras funções. Para que se consiga utilizar esta função é necessário gerar um valor pseudo-aleatório, *salt*, que vai tornar a função de KDF mais segura contra ataques.

```
#FUNÇÃO: Cifra ficheiro
def encrypt():
```

```

#ficheiro a cifrar
doc = input('Ficheiro a cifrar: ')
#chave a ser utilizada na cifra
cipherKey = input('Chave: ').encode('utf-8')
#nome do ficheiro cifrado
fileName = input('Ficheiro de output: ')

#retira a mensagem do ficheiro a cifrar
f = open(doc, "r")
msg = f.read()
f.close()

print("MENSAGEM")
print(msg)

#gera o salt a ser utilizado na geração da chave de 32 bytes
salt = get_random_bytes(16)
#gera chave de 32 bytes utilizando o input do utilizador
kdf = Script( salt=salt,length=32, n=2**14,r=8,p=1)
key = kdf.derive(cipherKey)

#cria ficheiro cifrado
ciphertext = buildCiphertext(msg,key,salt)

print("CIPHERTEXT - ** Armazenado no Ficheiro " + fileName + " **")
print(json.dumps(ciphertext))

#escreve a mensagem cifrada em bytes no ficheiro
# com nome igual à variável filename
output = open(fileName, "w")
output.write(json.dumps(ciphertext))
output.close()

```

Como forma de criar o ficheiro cifrado é utilizada a função *buildCiphertext* que recebe o ficheiro, a chave a ser utilizada no processo de cifra e o *salt* que foi utilizado na geração da chave para que no processo de decifra se consiga derivar a mesma chave de 32 bytes. Nesta função é gerada um valor *nonce* de 12 bytes a ser utilizado no algoritmo ChaCha20 e é criado um pacote representativo todo o ficheiro cifrado que contém o texto cifrado, o valor do *nonce* e *salt*.

*#FUNÇÃO: Cria o ciphertext através da cifra Chacha20*

```
def buildCiphertext(msg,key,salt):
```

```

    #cria o nonce de 12 bytes de forma pseudo-aleatória
    nonce = get_random_bytes(12)

```

```

    #cria a cifra Chacha20 utilizando a chave e o nonce

```

```

cipher = ChaCha20.new(key=key, nonce=nonce)
#cifra a mensagem
ciphertext = cipher.encrypt(dumps(msg))

#transforma em bytes a mensagem, nonce
# e o salt para geração da chave
ct = b64encode(ciphertext).decode('utf-8')
nc = b64encode(nonce).decode('utf-8')
st = b64encode(salt).decode('utf-8')

#mensagem cifrada - ciphertext
pkg = {'ciphertext' : ct, 'nonce': nc, 'salt' : st}

return pkg

```

Caso o utilizador pretenda decifrar um ficheiro é chamada a função *decrypt* que irá perguntar ao utilizador qual o ficheiro a decifrar, chave a ser utilizada pela cifra ChaCha20 e o ficheiro de *output* com o *plaintext*.

É de notar que a chave que realmente será utilizada neste processo terá um tamanho de 32 bytes (256 bits) e será derivada a partir da chave que o programa recebeu como input e do *salt* presente no ficheiro cifrado.

*#Função: Decifra um ficheiro*

```

def decrypt():

    #ficheiro a decifrar
    doc = input('Ficheiro a decifrar: ')
    #chave a ser utilizada na decifra
    cipherKey = input('Chave: ').encode('utf-8')
    #nome do ficheiro com a mensagem decifrada
    fileName = input('Ficheiro de output: ')

    #retira bytes da mensagem do ficheiro a dcifrar
    f = open(doc, "rb")
    msg = f.read()

    print("MENSAGEM")
    print(msg)

    #gera a chave de 32 bytes utilizando o input do utilizador
    # e o salt presente na mensagem cifrada
    ciphertext = json.loads(msg)
    salt = b64decode(ciphertext['salt'])
    kdf = Script( salt=salt,length=32, n=2**14,r=8,p=1)
    key = kdf.derive(cipherKey)

    #cria a mensagem decifrada
    plaintext = buildPlaintext(ciphertext,key)

```

```

print("PLAINTEXT - ** Armazenado no Ficheiro " + fileName + " **")
print(loads(plaintext))

#escreve no ficheiro a mensagem cifrada
output = open(fileName, "w")
output.write(loads(plaintext))

```

Como forma de criar o ficheiro decifrado é utilizada a função *buildPlaintext* que recebe o ficheiro e a chave a ser utilizada no processo de decifra. Nesta função é retirado do ficheiro o valor *nonce* de 12 bytes que foi gerado no processo de cifra para que o algoritmo ChaCha20 tenha os mesmos parâmetros que na cifragem, conseguindo-se assim decifrar o ficheiro corretamente.

```

#FUNÇÃO: Cria o plaintext através da cifra ChaCha20
def buildPlaintext(msg,key):

```

```

    #retira o valor nonce da mensagem cifrada
    nonce = b64decode(msg['nonce'])
    #retira a mensagem cifrada do ciphertext
    ciphertext = b64decode(msg['ciphertext'])

    #cria a decifra Chacha20 utilizando a chave e o non
    cipher = ChaCha20.new(key=key, nonce=nonce)
    #decifra mensagem
    plaintext = cipher.decrypt(ciphertext)

    return plaintext

```

Finalmente, a cifra ChaCha20 desenvolvida em Python encontra-se implementada na íntegra no seguinte [link](#) do repositório do Github. Para conseguir correr o programa pode-se utilizar o ficheiro *.txt* para cifra e posterior decifra presente no [link](#)

## Pergunta P1.2

### Pergunta 1

A maioria dos serviços online atualmente utilizam *passwords* de forma a autenticar utilizadores no sistema. Para isso é necessário que estes serviços mantenham armazenada numa base de dados as diversas *passwords* dos utilizadores para que o sistema consiga comparar o *login* fornecido pelo utilizador e o registo presente na base de dados e assim autenticar e validar o utilizador. Contudo, estas *passwords* nunca devem estar armazenadas em claro na BD para evitar que atacantes consigam aceder facilmente a credenciais dos utilizadores do sistema. Como forma de armazenar em segurança todas as *passwords*, o serviço armazena na base dados representações dessas *passwords* utilizando, por exemplo, funções de *hash*.

As funções hash tem como objetivo mapear mensagens de comprimento arbitrário (neste caso as *passwords*) num contradomínio de tamanho fixo (a representação *hash*

da *password*). Apesar destas funções serem seguras e utilizadas em diversas funcionalidades, não são usadas para o armazenamento de *passwords*, pois a sua utilização pode permitir a atacantes realizar ataques de dicionário. Este método de ataque, semelhante a ataques de força bruta, permite a atacantes catalogar todo o espaço de chaves e verificar padrões e possíveis representações de *passwords*, principalmente as mais simples. Assim, *passwords* simples como 12345 são facilmente quebráveis pois são utilizadas por uma quantidade maior de pessoas, sendo por isso possível verificar as diversas representações hash presentes na BD e determinar que registos é que utilizam estes tipo de *passwords*, comprometendo a autenticação do utilizador associado ao registo.

Uma forma de resolver esta vulnerabilidade, será utilizar funções de derivação de *passwords* (PBKDF). Estas funções, semelhantes às funções de *hash*, geram uma representação *hash* da password que será armazenada na BD, contudo esta representação é gerada utilizando fatores aleatório (*salt* ou *iv*) que impedem atacantes verificar padrões entre as representações. Estes fatores aleatórios serão concatenadas com o segredo, para que seja possível comprar e validar o *login* com o registo presente na BD. Além disso, estas funções tem um peso computacional maior que as funções de *hash* normais, dificultando a realização dos ataques de dicionário em tempo real.

## Pergunta 2

Para determinarmos a *password* representada em formato hexadecimal pela *hash* 96cae35ce8a9b0244178bf28e4966c2ce1b8385723a96a6b838858cdd6ca0a1e, decidiuse implementar um programa que através do top 200 de *passwords* mais usadas, vai testando por força bruta as diferentes possibilidades de password tentando, desta forma, encontrar a *password* representa pela *hash* apresentada.

Inicialmente, criou-se um ficheiro que contenha o top 200 das *passwords* mais utilizadas, segundo o *website* NordPass[1]. O ficheiro criado com este top 200 pode ser acedido no [link](#).

De seguida foi desenvolvido um programa em Python que irá armazenar o top 200 presente no ficheiro na lista *top*. Esta lista será percorrida de forma a gerar a representação hexadecimal do valor de hash de cada uma das passwords. Cada representação será depois comparada com o valor do hash da password que estamos a tentar encontrar e verificar se é igual. Caso seja igual então a password foi descoberta. O programa desenvolvido encontra-se abaixo ou no [link](#).

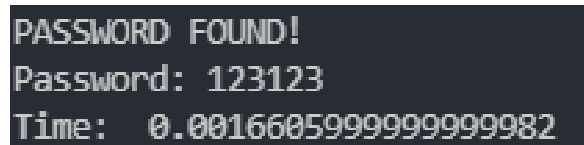
```
1 import timeit
2 from Crypto.Hash import SHA256
3
4 #Hash hexadecimal da password que queremos descobrir
5 hashPassword =
6     ↪ "96cae35ce8a9b0244178bf28e4966c2ce1b8385723a96a6b838858cdd6ca0a1e"
7
8 #Armazena na lista top o top 200 das password mais usadas
9 with open("top200.txt") as file:
10     top = [line.rstrip() for line in file]
11
12 #Iniciar timer para medir o tempo de descoberta da password
13 starttime = timeit.default_timer()
```

```

13
14 #Ciclo que percorrer o top 200 para determinar a password
15 for password in top:
16
17     #Determina a hash hexadecimal de uma password
18     h = SHA256.new()
19     h.update(password.encode('utf-8'))
20     hashPass = h.hexdigest()
21
22     #Verifica se a hash calculada é igual à hash que queremos
23     ↪ descobrir
24     if hashPassword == hashPass:
25
26         #Se for igual, então encontramos a password!
27         print("PASSWORD FOUND!")
28         print("Password: " + password)
29         print("Time: ", timeit.default_timer() - starttime)
30         break

```

Na figura abaixo encontra-se o resultado do programa. Como podemos ver, a password encontrada é 123123 e foi descoberta em menos de 1 segundo.



```

PASSWORD FOUND!
Password: 123123
Time: 0.0016605999999999982

```

Figure 1: *Output* do programa

## Pergunta P1.3

Para obter o HMAC-SHA1 de todos os ficheiros numa diretoria basta executar o comando `openssl dgst -sha1 -hmac [chave] [diretoria]/*` ou, alternativamente, `openssl sha1 -hmac [chave] [diretoria]/*`. Se estivermos dentro da diretoria não é necessário colocar o nome da mesma no parâmetro diretoria, ou seja em vez de `[diretoria]/*` basta colocar `*`.



```

rhezzus@RhEzZuS:~$ openssl dgst -sha1 -hmac "key" Documents/*
HMAC-SHA1(Documents/exemplo2.txt)= d8df87d1fc4443719383abffff1b26a51277c799
HMAC-SHA1(Documents/exemplo3.txt)= 5ac62615ff07abf4cdb2568de04be4f67e2c3e7f
HMAC-SHA1(Documents/exemplo.txt)= 939a4afa31e387bc9d174384b907e87c5541a9ff
rhezzus@RhEzZuS:~$ cd Documents/
rhezzus@RhEzZuS:~/Documents$ openssl dgst -sha1 -hmac "key" *
HMAC-SHA1(exemplo2.txt)= d8df87d1fc4443719383abffff1b26a51277c799
HMAC-SHA1(exemplo3.txt)= 5ac62615ff07abf4cdb2568de04be4f67e2c3e7f
HMAC-SHA1(exemplo.txt)= 939a4afa31e387bc9d174384b907e87c5541a9ff
rhezzus@RhEzZuS:~/Documents$ vim exemplo.txt
rhezzus@RhEzZuS:~/Documents$ openssl dgst -sha1 -hmac "key" *
HMAC-SHA1(exemplo2.txt)= d8df87d1fc4443719383abffff1b26a51277c799
HMAC-SHA1(exemplo3.txt)= 5ac62615ff07abf4cdb2568de04be4f67e2c3e7f
HMAC-SHA1(exemplo.txt)= 04e6143f5f5c78834c888dd2f0ad31db298ac9c1
rhezzus@RhEzZuS:~/Documents$

```

Figure 2: Comando OpenSSL

Para efeitos exemplificativos, temos a figura acima, onde foram criados 3 ficheiros numa diretoria e após a aplicação do comando com a chave "key" e a diretoria Documents podemos observar o HMAC-SHA1 de cada um dos mesmos.

No que toca a saber quais ficheiros foram alterados pela última vez, basta executar o comando novamente e verificar se o HMAC-SHA1 foi de facto modificado.

Assim sendo, na mesma figura exemplificativa mencionada anteriormente, o ficheiro exemplo.txt é modificado, com o comando `vim exemplo.txt`, e de seguida é executado o comando `openssl sha1 -hmac *` novamente e, como seria esperado, pode-se realmente verificar que realmente o HMAC-SHA1 para o ficheiro em questão alterou-se.

## Parte VI: Acordo de chaves

### Pergunta P.VI.1.1

De modo a implementar o protocolo Diffie-Hellman utilizou-se a linguagem python e o package `cryptography`.

O protocolo começa pela definição dos parâmetros iniciais que devem ser conhecidos tanto pela Alice como pelo Bob. Deste modo, utilizou-se um valor de  $g = 2$  e um tamanho de chave igual a 2048 bits.

Deste modo, a Alice começa por gerar os parametros e obtém o valor de  $p$ , sendo os valores de  $p$  e  $g$  enviados para o Bob. Este utiliza os valores para gerar uma instância do objeto `DHParameters`, que vai ser utilizada para gerar o valor de  $b$ .

```
1  #---- Geracao dos parametros ----
2  print("##### GERAÇÃO DE PARAMETROS #####\n\n")
3  g = 2
4  keySize = 1024
5
6  #Alice cria os parametros
7  alice_parameters = dh.generate_parameters(generator = g,
8  ↪ key_size=keySize)
9  print("[ALICE] O valor de g é: " + str(g))
10
11 #Obtem o p
12 p = alice_parameters.parameter_numbers().p
13 print("[ALICE] O valor de p é: " + str(p))
14
15 #Alice envia p e g para o bob
16 print("[ALICE] Enviei os parametros p e g para o Bob")
17 bob_parameters = dh.DHParameterNumbers(p, g).parameters()
```

Após ambos terem as instâncias dos objetos `DHParameters`, a Alice gera os valores de  $a$  e  $g^a$ . O método utilizado para gerar o valor  $a$  foi o `"generate_private_key()"`. Já o método `"public_key()"` que gera o valor de  $g^a$  utiliza a instância gerada pelos método anterior.

```
1  #----- Alice cria a e g^a -----
2
```



```

3 print("\n\n#### TROCA DAS CHAVES E OBTENÇÃO DO SEGREDO ####\n\n")
4 #Alice cria o parametro a (valor secreto e privado)
5 alice_a = alice_parameters.generate_private_key()
6 print("[ALICE] O valor de a foi gerado.")
7
8 #Alice cria o parametro g^a (valor que e enviado ao bob)
9 alice_ga = alice_a.public_key()
10 print("\n[ALICE] O valor de g^a foi gerado e é:\n " +
    ↪ str(alice_ga.public_bytes(encoding=Encoding.PEM,
    ↪ format=PublicFormat.SubjectPublicKeyInfo)))

```

O bob realiza o mesmo processo mas gera os valores de b e  $g^a$ .

```

1 #----- Bob cria b e g^b -----
2
3 #Bob cria o parametro b (valor secreto e privado)
4 bob_b = bob_parameters.generate_private_key()
5 print("\n[BOB] O valor de b foi gerado." )
6
7 #bob cria o parametro g^b (valor que e enviado a Alice)
8 bob_gb = bob_b.public_key()
9 print("\n[BOB] O valor de g^b é: \n" +
    ↪ str(bob_gb.public_bytes(encoding=Encoding.PEM,
    ↪ format=PublicFormat.SubjectPublicKeyInfo)))

```

Por conseguinte, a ALice envia o valor de  $g^a$  para o Bob e este última envia o valor de  $g^b$  para a Alice. A Alice utiliza a instância do objeto privado que gerou (corresponde ao valor a) e utiliza o método "exchange" com o valor recebido pelo bob para obter o segredo partilhado por ambos (ou chave partilhada). Este processo corresponde a gerar o valor de  $g^{(ab)}$ . Tal como a Alice, o Bob segue os mesmos passos e obtém o mesmo valor que a Alice.

```

1 #---- Bob envia g^b para a Alice e esta obtem o segredo partilhado
   ↪ -----#
2 print("\n[BOB] A enviar g^b para a Alice")
3 alice_sharedSecret = alice_a.exchange(bob_gb)
4 print("\n[ALICE] O segredo partilhado entre ambos é: \n" +
   ↪ str(alice_sharedSecret))
5
6 #---- Alice envia g^a para o Bob e este obtem o segredo partilhado
   ↪ -----#
7
8 print("\n[ALICE] A enviar g^a para o Bob")
9 bob_sharedSecret = bob_b.exchange(alice_ga)
10 print("\n[BOB] O segredo partilhado entre ambos é: \n" +
    ↪ str(bob_sharedSecret))

```

Após ambos possuírem o mesmo segredo, é possível iniciar o processo de troca de mensagens. Para realizar este processo utilizou-se a cifra AES no modo GCM, o que

levou à necessidade de derivar uma chave que fosse compatível a partir dos segredos que ambos possuem.

```
1  #----- Troca de mensagens -----
2
3  print("\n\n##### TROCA DE MENSAGENS #####\n\n")
4  #Utiliza-se o AES no modo GCM para trocar as mensagens
5  #Logo e necessario criar chaves a partir dos segredos do Bob e Alice
6
7  bob_AesKey = HKDF(algorithm=hashes.SHA256(), length=32, salt=None,
    ↪   info=None).derive(bob_sharedSecret)
8  print("\n[BOB] A chave derivada a partir do segredo é: \n" +
    ↪   str(bob_AesKey))
9
10 alice_AesKey = HKDF(algorithm=hashes.SHA256(), length=32, salt=None,
    ↪   info=None).derive(alice_sharedSecret)
11 print("\n[ALICE] A chave derivada a partir do segredo é: \n" +
    ↪   str(alice_AesKey))
```

De seguida o Bob começou por enviar a sua mensagem: "Ola eu sou o Bob.". Assim sendo, utilizou a cifra anterior para cifrar a mensagem, para a qual teve de criar um nonce ao passo que no final obteve uma tag de autenticação. Este criptograma foi enviado para a Alice que utilizou o nonce e tag para decifrar e autenticar a mensagem, o que levou a que obtivesse o *plaintext* original.

```
1  #Bob cifra a mensagem 'Ola eu sou o Bob'
2
3  bob_plaintext = "Ola eu sou o Bob."
4  print("\n[BOB] vou enviar a mensagem: " + str(bob_plaintext))
5  iv_Bob = os.urandom(16)
6  encryptor = Cipher(algorithms.AES(bob_AesKey),
    ↪   modes.GCM(iv_Bob)).encryptor()
7  bob_Ciphertext = encryptor.update(bob_plaintext.encode()) +
    ↪   encryptor.finalize()
8  bob_tag = encryptor.tag
9
10 print("\n[BOB] O ciphertext obtido é: \n" + str(bob_Ciphertext))
11
12 #Bob envia o criptograma para a alice
13
14 print("\n[BOB] A enviar o criptograma para o Bob...")
15
16 #Alice decifra a mensagem
17 print("\n[ALICE] O criptograma recebido foi: \n" + str(bob_Ciphertext))
18 decryptor = Cipher(algorithms.AES(bob_AesKey), modes.GCM(iv_Bob,
    ↪   bob_tag)).decryptor()
19 bob_Deciphered = decryptor.update(bob_Ciphertext) +
    ↪   decryptor.finalize()
```

20

```
21 print("\n[ALICE] Após decifrar o criptograma, a mensagem obtida foi: "
    ↪ + str(bob_Deciphered.decode()))
```

Por fim, a Alice responde ao Bob com a mensagem "Ola Bob! Eu sou a Alice.", através da utilização do processo anterior, ou seja, criou o objeto da cifra, gerou um novo nonce e obteve uma nova tag. De seguida enviou o criptograma para o Bob que o decifrou e obteve a mensagem original.

```
1  #Alice responde e cifra a mensagem 'Ola Bob! Eu sou a Alice.'
2  alice_plaintext = "Ola Bob! Eu sou a Alice."
3  print("\n[ALICE] vou enviar a mensagem: " + str(alice_plaintext))
4
5  iv_Alice = os.urandom(16)
6  encryptor = Cipher(algorithms.AES(alice_AesKey),
    ↪  modes.GCM(iv_Alice)).encryptor()
7  alice_Ciphertext = encryptor.update(alice_plaintext.encode()) +
    ↪  encryptor.finalize()
8  alice_tag = encryptor.tag
9
10 print("\n[ALICE] O ciphertext obtido é: \n" + str(alice_Ciphertext))
11
12 #Alice envia o criptograma para o Bob
13 print("\n[ALICE] A enviar o criptograma para o Bob...")
14
15 #Bob decifra a mensagem
16 print("\n[BOB] O criptograma recebido foi: \n" + str(alice_Ciphertext))
17
18 decryptor = Cipher(algorithms.AES(bob_AesKey), modes.GCM(iv_Alice,
    ↪  alice_tag)).decryptor()
19 alice_Deciphered = decryptor.update(alice_Ciphertext) +
    ↪  decryptor.finalize()
20
21 print("\n[BOB] Após decifrar o criptograma, a mensagem obtida foi: " +
    ↪  str(alice_Deciphered.decode()))
```

O código encontra-se neste [link](#). Para o executar basta ter o *package* cryptography instalado e utilizar o comando: `python3 diffie-hellman.py`

# Bibliography

- [1] "Top 200 most common passwords" [online]. Disponível em: <https://nordpass.com/most-common-passwords-list/> [Acedido em março de 2022].
- [2] "Cryptography" [online]. Disponível em: <https://cryptography.io/en/latest/> [Acedido em março de 2022].