

# Informe Módulo 2

## Criptografía

Fecha: 9 de Enero de 2024

Autor: **Azael Ramírez Pérez**

Mail: **keepcoder\_test@gmail.com** (ficticio)

Empresa: **KeepCoder.inc** (ficticio)

# Ámbito y alcance

El presente trabajo está enfocado en contestar una serie de 15 ejercicios referentes al módulo abordado (Criptografía), en este mismo se documentan los procedimientos realizados.

## Características de Hardware

Laptop	MSI
Procesador	Intel Core i7, 9th Gen, 2.60 GHz
Memoria	16 GB

## Características de Software

S.O.	Windows 11 x64
Compilador	Python 3.10.9

### Ejercicio 1.

Tenemos un sistema que usa claves de 16 bytes. Por razones de seguridad vamos a proteger la clave de tal forma que ninguna persona tenga acceso directamente a la clave. Por ello, vamos a realizar un proceso de disociación de la misma, en el cuál tendremos, una clave fija en código, la cual, sólo el desarrollador tendrá acceso, y otra parte en un fichero de propiedades que rellenará el Key Manager. La clave final se generará por código, realizando un XOR entre la que se encuentra en el properties y en el código.

La clave fija en código es **B1EF2ACFE2BAEEFF**, mientras que en desarrollo sabemos que la clave final (en memoria) es **91BA13BA21AABB12**.

¿Qué valor ha puesto el Key Manager en properties para forzar dicha clave final?

**Respuesta:** **0X20553975C31055ED**

### Procedimiento de la solución (Pregunta 1 - Ejercicio 1)

Una vez que se ha analizado el problema, observamos que para la primera interrogativa, nos han dado 2 claves cada clave con 16 bytes, la primera (A) se debe asignar en el código y la segunda clave (C) es la clave resultante de realizar la operación de XOR a 2 claves previas A XOR B.

A modo de ejemplo podemos decir que  $A \text{ XOR } B = C$ , es decir:

A = **B1EF2ACFE2BAEEFF**

B = ?

C = **91BA13BA21AABB12** (en memoria)

Así que lo que necesitamos hacer es encontrar el valor correspondiente a la clave B para así poder comprobar que la clave de C dada es correcta.

Siguiendo nuestro razonamiento, podemos hacer lo siguiente:

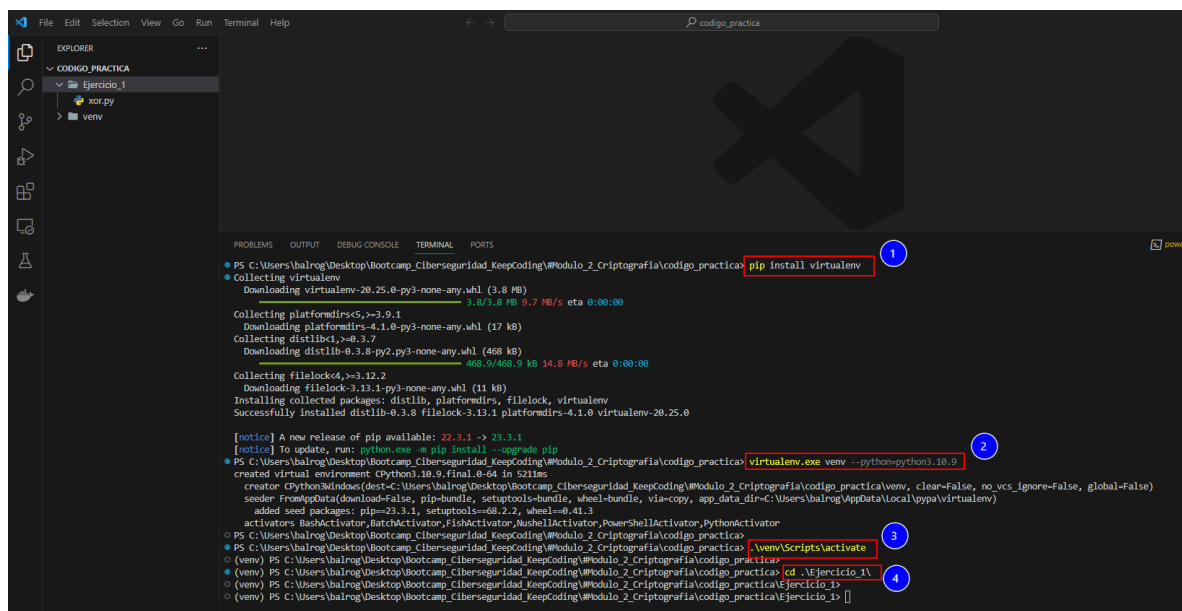
$B = A \text{ XOR } C$

De esta manera encontraremos el valor de B, después para poder comprobar a C es necesario realizar lo siguiente, es decir hacer la operación XOR de A con B tomando el valor que se ha obtenido de B:

A XOR B = C

De esta manera podremos obtener valor de C y al comparar ambos valores C (en memoria) y C (el que se ha calculado) podemos comprobar si el valor de B es el correcto.

Comenzamos configurando nuestro ambiente virtual en Python (ver figura 1)



```
File Edit Selection View Go Run Terminal Help
codigo_practica

EXPLORER
CODIGO_PRACTICA
  ejercicio_1
    main.py
    venv

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica> pip install virtualenv
Collecting virtualenv
  Downloading virtualenv-20.25.0-py3-none-any.whl (3.8 MB)
    3.8/3.8 MB 9.7 MB/s eta 0:00:00
Collecting platformdirs<5,>=3.9.1
  Downloading platformdirs-4.1.0-py3-none-any.whl (17 kB)
Collecting distlib<=0.3.7
  Downloading distlib-0.3.8-py2.py3-none-any.whl (458 kB)
    458.9/458.9 kB 14.8 MB/s eta 0:00:00
Collecting filelock<=3.12.2
  Downloading filelock-3.13.1-py3-none-any.whl (11 kB)
Installing collected packages: distlib, platformdirs, filelock, virtualenv
Successfully installed distlib-0.3.8 filelock-3.13.1 platformdirs-4.1.0 virtualenv-20.25.0

[notice] A new release of pip available: 22.3.1 -> 23.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica> virtualenv.exe venv --python=python3.10.9
created virtual environment (Python3.10.9.final.0-64) in 521ms
creator: CPythonWindows(dest=C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\venv, clear=False, no_vcs_ignore=False, global=False)
seeders: FromAppData(download=False, pip-bundle, setuptools-bundle, wheel-bundle, via-copy, app_data_dir=C:\Users\balrog\AppData\Local\ypa\virtualenv)
added seed packages: pip==23.3.1, setuptools==68.2.2, wheel==0.41.3
activators: BashActivator, BatchActivator, FishActivator, NushellActivator, PowerShellActivator, PythonActivator
PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica>
PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica> venv\Scripts\activate
(venv) PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica> cd .\ejercicio_1\
(venv) PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\ejercicio_1>
```

Figura 1. Configuración de nuestro ambiente virtual en Python

Haciendo referencia a los puntos marcados en la figura anterior, tenemos que se hicieron las actividades siguientes:

- 1-Instalamos Virtualenv
- 2-Creamos un ambiente virtual
- 3-Activamos el ambiente virtual
- 4-Cambiamos de directorio donde está alojado nuestro código fuente

Ejecución del programa en Python aplicando las operaciones de XOR para encontrar el valor de B y después comparar ambos valores para comprobar si este valor encontrado es el correcto.

```
1 print("\n")
2 print("Ejercicio 1")
3 print("-----")
4 print("Definición de variables")
5 print("a=0b11F2ACF2B4EFF")
6 print("b = ?")
7 print("c=0b1BA1BA21AAB12")
8 print("-----")
9
10 a=0b11F2ACF2B4EFF
11 c=0b1BA1BA21AAB12
12
13 #Operacion a realizar
14 # b = a XOR c
15
16 print("b = a XOR c")
17 #Resultado de b
18 b=hex(a^c)
19 print("b = ",b.upper()) #b=0X20553975C31055ED
20 print()
21
22 # Comparación
23 #Operacion a realizar
24 # c = a XOR b
25
26 print("c = a XOR b")
27 valor_c=hex(a^b)
28 print("valor_c = ", valor_c.upper())
29 print()
30
31 #se convierte el valor inicial de C esta estando en bytes a int y luego a hexadecimal
32 #print("c_hexa=",c)
33 valor_c_int = int(c)
34 #print("c=",valor_c_int)
35 c_hexadecimal_memoria= hex(valor_c_int)
36 #print("c_hexadecimal",c_hexadecimal_memoria)
37
38 #Comparamos el valor que se nos ha dado (memoria) contra el resultado
39 #es a ver si es decir el valor de la variable: valor_c
40 if c_hexadecimal_memoria == valor_c:
41     print("Comparando las 2 claves: clave memoria vs clave obtenida")
42     print(c_hexadecimal_memoria,"", valor_c)
43     print("Los valores son iguales!")
44     print("La llave correcta que se debe poner en el properties es:",b.upper())
45     print()
46 else:
47     print("Los valores no coinciden!")
```

```
Ejercicio 1
-----
Definición de variables
a=0b11F2ACF2B4EFF
b = ?
c=0b1BA1BA21AAB12
-----

b = a XOR c
Resultado de b
b = 0X20553975C31055ED

Comparando las 2 claves: clave memoria vs clave obtenida
0b1ba1ba21aabb12 - 0b1ba1ba21aabb12
Los valores son iguales!
La llave correcta que se debe poner en el properties es: 0X20553975C31055ED

(wmv) PS C:\Users\haleg\Desktop\bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_1>
```

Figura 2. Ejecución del programa en Python

En los puntos marcados en la figura anterior se observa el código en el punto 1, en el punto 2 al 4 simplemente se activa el ambiente virtual, nos cambiamos de directorio y finalmente se ejecuta el programa escrito en python (punto 4).

En el punto 5 se observa el valor correspondiente a B (0X20553975C31055ED) siendo el valor que se ha puesto en el properties en el sistema de numeración hexadecimal.

La clave fija, recordemos es **B1EF2ACFE2BAEEFF**, mientras que en producción sabemos que la parte dinámica que se modifica en los ficheros de propiedades es **B98A15BA31AE8B3F**.

¿Qué clave será con la que se trabaje en memoria?

**Respuesta:** **0X8653F75D31455C0**

### Procedimiento de la solución (Pregunta 2 - Ejercicio 1)

Para dar solución a este planteamiento solo es necesario realizar la operación XOR tomando los valores que se nos han dado, es decir se nos ha proporcionado la clave fija (asignada en código) y la clave del archivo de properties, nuestra formula quedaría de la siguiente manera:

```
clave_memoria = clave_fija XOR clave_properties
```

En la siguiente imagen se realiza la operación de XOR a través del lenguaje de programación Python tomando como referencia la formula descrita anteriormente.

The screenshot shows a Python IDE with two files: 'Ejercicio\_1.py' and 'Ejercicio\_2.py'. In 'Ejercicio\_1.py', lines 61 and 62 are highlighted with a red box and a red circle labeled '1'. These lines declare variables 'clave\_fija' and 'clave\_properties' and perform an XOR operation to calculate 'clave\_memoria'. The output of the program is shown in the terminal, displaying the hexadecimal values for 'clave\_fija', 'clave\_properties', and the resulting 'clave\_memoria' (0X8653F75D31455C0). A red box and a red circle labeled '2' highlight the command 'python.exe .\xor' in the terminal. A third red box and a red circle labeled '3' highlight the final output of the XOR operation in the terminal.

```
#Operacion a realizar
61 clave_memoria = clave_fija XOR clave_properties
62
63 print("\n")
64 print("Ejercicio 2")
65 print("-----")
66 print("Definición de variables")
67 print("clave_fija=0XB1EF2ACFE2BAEEFF")
68 print("clave_properties=0XB98A15BA31AE8B3F")
69 print("clave_memoria=?")
70
71 clave_fija=0XB1EF2ACFE2BAEEFF
72 clave_properties=0XB98A15BA31AE8B3F
73 clave_memoria=(clave_fija XOR clave_properties)
74 print("clave_memoria= clave_fija XOR clave_properties")
75 print("clave_memoria =",clave_memoria.upper())
76 print()
```

```
PS C:\Users\ba1rog\Desktop\bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_1> python.exe .\xor
Ejercicio 1
Definición de variables
a=0XB1EF2ACFE2BAEEFF
b=?
c=0XB98A15BA31AE8B3F

b = a XOR c
b = 0X20553975C31055ED

c = a XOR b
valor_c = 0X91BA13BA21A0B12

Comparando las 2 claves: clave memoria vs clave obtenida
0X91BA13BA21A0B12 = 0X91BA13BA21A0B12
Los valores son iguales!
la llave correcta que se debe poner en el properties es: 0X20553975C31055ED

Ejercicio 2
Definición de variables
clave_fija=0XB1EF2ACFE2BAEEFF
clave_properties=0XB98A15BA31AE8B3F
clave_memoria=?

clave_memoria= clave_fija XOR clave_properties
clave_memoria = 0X8653F75D31455C0
```

En el punto 1 podemos ver el código en Python, en la fila 61 y 62 se puede ver que se están declarando 2 variables, cada una de estas se les ha asignado el valor que se nos ha dado y posteriormente se ha realizado la operación XOR, dando como resultado la clave en memoria siendo **0X8653F75D31455C0**.

## Ejercicio 2.

Dada la clave con etiqueta "**cifrado-sim-aes-256**" que contiene el keystore. El iv estará compuesto por el hexadecimal correspondiente a ceros binarios ("00").

Se requiere obtener el dato en claro correspondiente al siguiente dato cifrado:

```
TQ9SOMKc6aFS9SlxhfK9wTl8UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nvvJbBiG+le1ZhpR84oI=
```

Para este caso, se ha usado un AES/CBC/PKCS7.

Si lo desciframos, ¿qué obtenemos?

**Respuesta:** Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.

¿Qué ocurre si decidimos cambiar el padding a x923 en el descifrado?

**Respuesta:** El programa se ha ejecutado correctamente debido que en el padding para este caso particular tiene solo 1 byte (01) al final de la cadena cifrada y al descifrar el mensaje es un caso en el que pkcs7, x923 y iso10126 tienen el mismo tipo de padding de manera que no afectara al descifrar el mensaje original.

¿Cuánto padding se ha añadido en el cifrado?

**Respuesta:** Ha quedado con el mismo padding 1 byte (01) al final de la cadena cifrada y no se le ha agregado padding adicional, pues comparten el mismo tipo de padding.

Se valorará positivamente, obtener el dato de la clave desde el keystore mediante codificación en Python (u otro lenguaje).

**Nota:** Se logró recuperar la clave del KeyStore a través de Python, ver figura 7.

## Procedimiento de la solución (Pregunta 1 - Ejercicio 2)

Para este caso, se ha usado un AES/CBC/PKCS7.

Si lo desciframos, ¿qué obtenemos?

**Respuesta:** Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.

En la siguiente figura se muestra la llave extraída del **KeyStore** para poder ocuparla dentro de nuestro código en Python y así poder descifrar el mensaje.

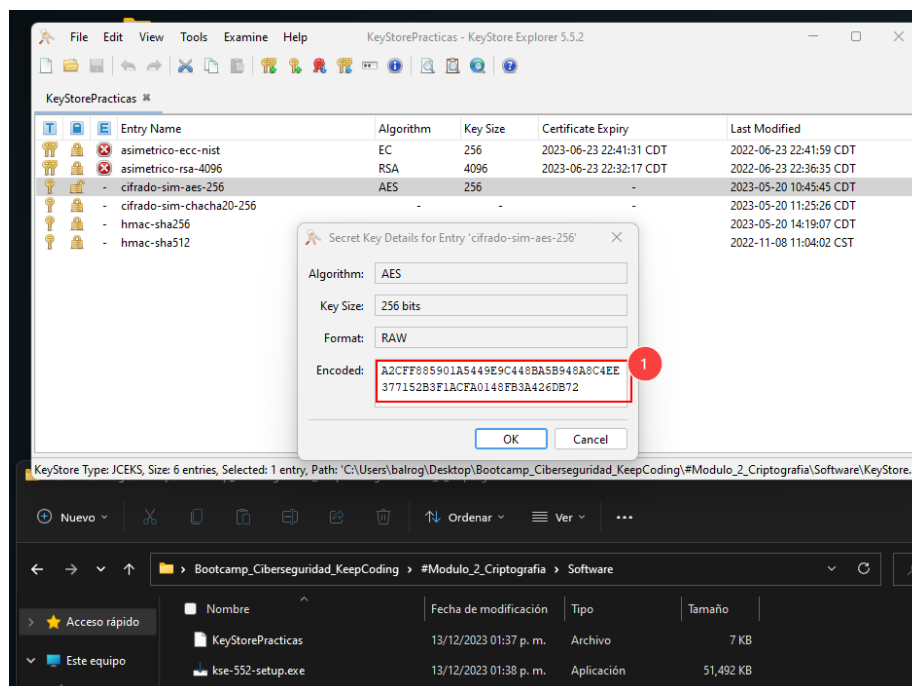


Figura 1. Importación de Keys en KeyStore Explorer

En el punto 1 de la siguiente figura se puede apreciar que se han asignado los valores necesarios para poder descifrar el mensaje y visualizarlos en texto plano, esto se realizó a través de un programa realizado en Python.

La clave que se obtuvo de manera manual fue:

**A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72**



```
1 import json
2 from base64 import b64encode, b64decode
3 from Crypto.Cipher import AES
4 from Crypto.Util.Padding import pad, unpad
5 from Crypto.Random import get_random_bytes
6
7 # Proceso de Descifrado
8 # -----
9 try:
10     # Se asignan los datos necesarios para poder hacer el descifrado del mensaje como son:
11     # la clave, el iv y el mensaje cifrado (este viene en base 64)
12     clave=bytes.fromhex('A2CFF8B5901A5449E9C448B8B948ABC4EE377152B3F1ACFA0148FB3A426DB72')
13     iv=bytes.fromhex('00000000000000000000000000000000')
14     texto_cifrado_bytes=b64decode('IQ9S0Wkc6aF5951xhFK9w118UXpPCd505XfS3/5nLI70f/o0QKlWxg3nu1RRz4QwElezdrLAD5L04UST3aB/150nvv7bB1G+1e1ZhpR84oI-')
15
16     cipher = AES.new( clave, AES.MODE_CBC, iv)
17     mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size, style='pkcs7')
18
19     print()
20     print("El texto en claro es:", mensaje_des_bytes.decode("utf-8"))
21     print()
22
23 except (ValueError, KeyError) as error:
24     print('Problemas para descifrar....')
25     print("El motivo del error es: ", error)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

(venv) PS C:\Users\balrog\Desktop\Bootcamp\_Ciberseguridad\_KeepCoding\Modulo\_2\_Criptografia\codigo\_practica\Ejercicio\_2> python.exe .\aes\_cbc\_pkcs7.py

El texto en claro es: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.

Figura 2. Descifrado del mensaje

En el punto 2 de la figura 2 se realiza la impresión del valor descifrado teniendo una codificación UTF-8, finalmente en el punto 3 podemos ver el mensaje en claro (Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.).

## Procedimiento de la solución (Pregunta 2 - Ejercicio 2)

¿Qué ocurre si decidimos cambiar el padding a x923 en el descifrado?

**Respuesta:** El programa se ha ejecutado correctamente debido que en el padding para este caso particular tiene solo 1 byte (01) al final de la cadena cifrada y al descifrar el mensaje es un caso en el que pkcs7, x923 y iso10126 tienen el mismo tipo de padding de manera que no afectara al descifrar el mensaje original.

Una vez que se ha modificado el padding de pkcs7 a x923 se ha vuelto a ejecutar el programa, este mismo se ha ejecutado bien y no ha marcado ningún error, ver figura 3.

```
1 import json
2 from base64 import b64encode, b64decode
3 from Crypto.Cipher import AES
4 from Crypto.Util.Padding import pad, unpad
5 from Crypto.Random import get_random_bytes
6
7 # Proceso de Descifrado
8 # -----
9 try:
10     # Se asignan los datos necesarios para poder hacer el descifrado del mensaje como son:
11     # la clave, el iv y el mensaje cifrado (este viene en base 64)
12     clave_bytes.fromhex('A2CFF8B5901A5440E9C440BA5B048A8C4EE377152B3F1ACFA0148FB3A426D072')
13     iv_bytes.fromhex('00000000000000000000000000000000')
14     texto_cifrado_bytes=b64decode('TQ95QPKc6aF5951xhfK9wT18UXpCd505Xf5j/5nLI7Of/o0QKlNXg3nu1RRz4QNElezdLAD5L04U5t3aB/150nvv7bB1G+1e1ZhpR84oI=')
15
16     cipher = AES.new( clave, AES.MODE_CBC, iv)
17     mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size, style="x923")
18
19     print()
20     print("El texto en claro es:", mensaje_des_bytes.decode("utf-8"))
21     print()
22
23 except (ValueError, KeyError) as error:
24     print("Problemas para descifrar...")
25     print("El motivo del error es: ", error)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

(venv) PS C:\Users\balrog\Desktop\Bootcamp\_Ciberseguridad\_KeepCoding\Modulo\_2\_Criptografia\codigo\_practica\Ejercicio\_2> python.exe .\aes\_cbc\_pkcs7.py

El texto en claro es: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.

(venv) PS C:\Users\balrog\Desktop\Bootcamp\_Ciberseguridad\_KeepCoding\Modulo\_2\_Criptografia\codigo\_practica\Ejercicio\_2> |

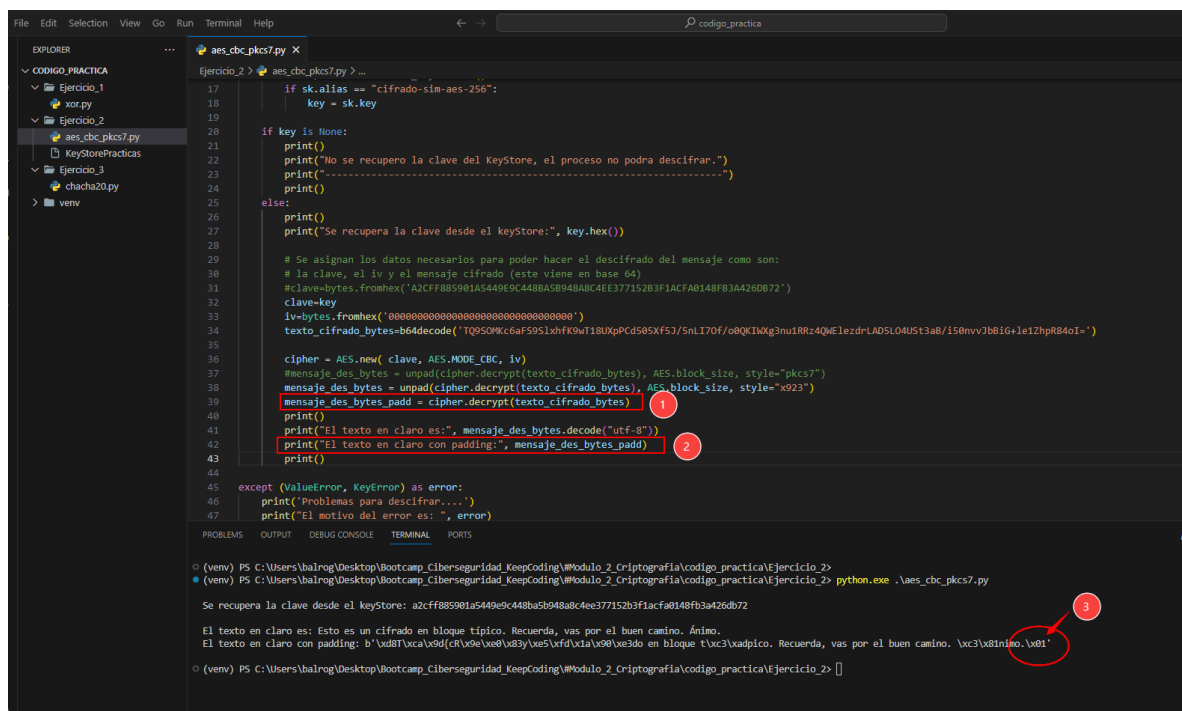
Figura 3. Modificación del padding a x923

## Procedimiento de la solución (Pregunta 3 - Ejercicio 2)

¿Cuánto padding se ha añadido en el cifrado?

**Respuesta:** Ha quedado con el mismo padding 1 byte (01) al final de la cadena cifrada y no se le ha agregado padding adicional, pues comparten el mismo tipo de padding.

En la siguiente imagen se hizo una ejecución del programa para realizar nuevamente el descifrado con la modificación en el tipo de padding (x923), lo que pudimos observar es que logra descifrar correctamente, adicionalmente se optó por poner la impresión del padding que contiene el mensaje (01 al final de la cadena), ver punto 3.



```
File Edit Selection View Go Run Terminal Help
aes_cbc_pkcs7.py
Ejercicio_2 > aes_cbc_pkcs7.py
17 if sk.alias == "cifrado-sin-aes-256":
18     key = sk.key
19
20 if key is None:
21     print()
22     print("No se recupero la clave del KeyStore, el proceso no podra descifrar.")
23     print("-----")
24     print()
25 else:
26     print()
27     print("Se recupera la clave desde el keyStore:", key.hex())
28
29 # Se asignan los datos necesarios para poder hacer el descifrado del mensaje como son:
30 # la clave, el iv y el mensaje cifrado (este viene en base 64)
31 #clave-bytes.fromhex('A2CFF885901A5448E9C448BA58948A8C4E377152B3F1ACFA0148F83A426D672')
32 clave=key
33 iv=bytes.fromhex('00000000000000000000000000000000')
34 texto_cifrado_bytes=b64decode('TQ95OWkc6af5951xhFK9wT18UXpCd585XF5J/5nL170f/o8QK1Wxg3nu1RRz4QW6IezdrLAD5L04Ust3aB/158nv7b8iG+le1ZhpR84oI=')
35
36 cipher = AES.new( clave, AES.MODE_CBC, iv)
37 #mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size, style='pkcs7')
38 mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size, style='x923')
39 mensaje_des_bytes_padd = cipher.decrypt(texto_cifrado_bytes)
40 print()
41 print("El texto en claro es:", mensaje_des_bytes.decode("utf-8"))
42 print("El texto en claro con padding:", mensaje_des_bytes_padd)
43 print()
44
45 except (ValueError, KeyError) as error:
46     print("Problemas para descifrar....")
47     print("El motivo del error es: ", error)
48
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(venv) PS C:\Users\halrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_2>
(venv) PS C:\Users\halrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_2> python.exe .\aes_cbc_pkcs7.py
Se recupera la clave desde el keyStore: a2cff885901a5448e9c448ba58948a8c4e377152b3f1acfa0148f83a426d672
El texto en claro es: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.
El texto en claro con padding: b'\xd8t\xca\x9d{Rv9e\x80\x83y\x85\xfd\xda\x90\x83do en bloque t\x83\xadpico. Recuerda, vas por el buen camino. \xc3\x81nimo.\x01'
```

Figura 4. Proceso de descifrado con padding x923

En una posterior ejecución del proceso de descifrado (ver figura 5 y 6) comparamos el padding de ambos tipos, es decir el descifrado con el padding usando pkcs7 (ver figura 5) y el descifrado usando el padding x923 (ver figura 6), concluyendo así, que no se observan un incremento en este mismo, ver puntos 2 y 3 de la figura 6.

```
9 try:
10     # Se asignan los datos necesarios para poder hacer el descifrado del mensaje como son:
11     # la clave, el iv y el mensaje cifrado (este viene en base 64)
12     clave=bytes.fromhex('A2CFF8B5901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426D872')
13     iv=bytes.fromhex('00000000000000000000000000000000')
14     texto_cifrado_bytes=b64decode('TQ9SOMKc6aF59S1xhfK9wT18UXpPCd505Xf5J/5nLI70f/o0QKIwXg3nu1RRz4QWEleZdrLAD5L04USt3aB/i50nvvJbBiG+leIzhpR84oI=')
15
16     cipher = AES.new( clave, AES.MODE_CBC, iv)
17     mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size, style="pkcs7")
18     mensaje_des_bytes_padd = cipher.decrypt(texto_cifrado_bytes)
19     print()
20     print("El texto en claro es:", mensaje_des_bytes.decode("utf-8"))
21     print("El texto en claro es:", mensaje_des_bytes_padd)
22     print()
23
24 except (ValueError, KeyError) as error:
25     print('Problemas para descifrar....')
26     print("El motivo del error es: ". error)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

(venv) PS C:\Users\balrog\Desktop\Bootcamp\_Ciberseguridad\_KeepCoding\Modulo\_2\_Criptografia\codigo\_practica\Ejercicio\_2> python.exe .\aes\_cbc\_pkcs7.py

El texto en claro es: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.

El texto en claro es: b'\xd8t\xca\x9d{cR\x9e\xe0\x83y\xe5\xfd\x1a\x90\xe3do en bloque t\xc3\xadpico. Recuerda, vas por el buen camino. \xc3\x81nimo.\x01'

(venv) PS C:\Users\balrog\Desktop\Bootcamp\_Ciberseguridad\_KeepCoding\Modulo\_2\_Criptografia\codigo\_practica\Ejercicio\_2> |

Figura 5. Ejecución del programa usando PKCS7

```
4 from Crypto.Util.Padding import pad, unpad
5 from Crypto.Random import get_random_bytes
6
7 # Proceso de Descifrado
8 # -----
9 try:
10     # Se asignan los datos necesarios para poder hacer el descifrado del mensaje como son:
11     # la clave, el iv y el mensaje cifrado (este viene en base 64)
12     clave=bytes.fromhex('A2CFF8B5901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426D872')
13     iv=bytes.fromhex('00000000000000000000000000000000')
14     texto_cifrado_bytes=b64decode('TQ9SOMKc6aF59S1xhfK9wT18UXpPCd505Xf5J/5nLI70f/o0QKIwXg3nu1RRz4QWEleZdrLAD5L04USt3aB/i50nvvJbBiG+leIzhpR84oI=')
15
16     cipher = AES.new( clave, AES.MODE_CBC, iv)
17     mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size, style="x923")
18     mensaje_des_bytes_padd = cipher.decrypt(texto_cifrado_bytes)
19     print()
20     print("El texto en claro es:", mensaje_des_bytes.decode("utf-8"))
21     print("El texto en claro es:", mensaje_des_bytes_padd)
22     print()
23
24 except (ValueError, KeyError) as error:
25     print('Problemas para descifrar....')
26     print("El motivo del error es: ". error)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

(venv) PS C:\Users\balrog\Desktop\Bootcamp\_Ciberseguridad\_KeepCoding\Modulo\_2\_Criptografia\codigo\_practica\Ejercicio\_2> python.exe .\aes\_cbc\_pkcs7.py

El texto en claro es: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.

El texto en claro es: b'\xd8t\xca\x9d{cR\x9e\xe0\x83y\xe5\xfd\x1a\x90\xe3do en bloque t\xc3\xadpico. Recuerda, vas por el buen camino. \xc3\x81nimo.\x01'

(venv) PS C:\Users\balrog\Desktop\Bootcamp\_Ciberseguridad\_KeepCoding\Modulo\_2\_Criptografia\codigo\_practica\Ejercicio\_2>

(venv) PS C:\Users\balrog\Desktop\Bootcamp\_Ciberseguridad\_KeepCoding\Modulo\_2\_Criptografia\codigo\_practica\Ejercicio\_2> python.exe .\aes\_cbc\_pkcs7.py

El texto en claro es: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.

El texto en claro es: b'\xd8t\xca\x9d{cR\x9e\xe0\x83y\xe5\xfd\x1a\x90\xe3do en bloque t\xc3\xadpico. Recuerda, vas por el buen camino. \xc3\x81nimo.\x01'

(venv) PS C:\Users\balrog\Desktop\Bootcamp\_Ciberseguridad\_KeepCoding\Modulo\_2\_Criptografia\codigo\_practica\Ejercicio\_2> |

Figura 6. Ejecución del programa usando X923

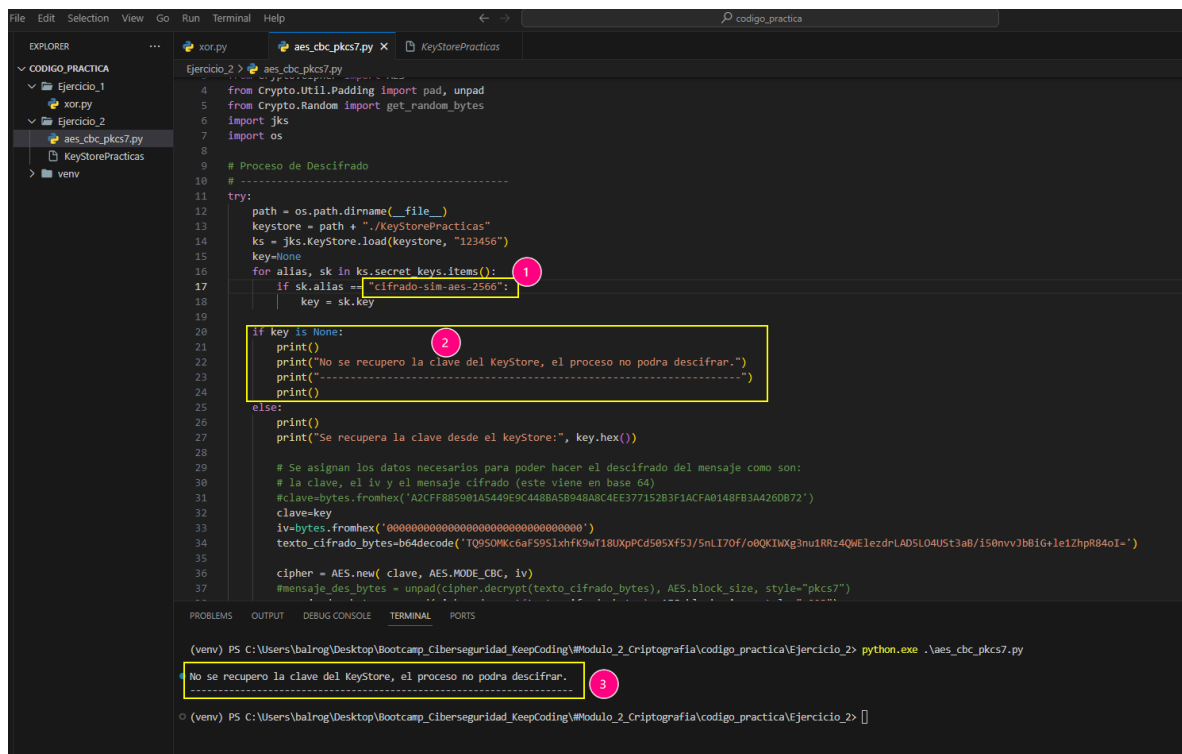
Se valorará positivamente, obtener el dato de la clave desde el keystore mediante codificación en Python (u otro lenguaje).

**Nota:** Se logró recuperar la clave del KeyStore a través de Python, ver figura 7.

En la siguientes imágenes se mencionan 2 escenarios realizados para recupera la clave del KeyStore desde Python.

En este **primer escenario** modifique el alias de la llave para que no pueda recuperar la clave debido a que no hay una clave con el alias **cifrado-sim-aes-2566** en el KeyStore.

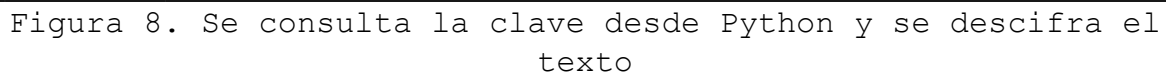
En el punto 1 de la siguiente figura se muestra el cambio en el alias de la clave, para el punto 2 se valida que la variable key no sea NULL y finalmente en el tercer punto nos muestra el mensaje.



```
File Edit Selection View Go Run Terminal Help
aes_cbc_pkcs7.py x KeyStorePracticas
Ejercicio_2 > aes_cbc_pkcs7.py
4 from Crypto.Util.Padding import pad, unpad
5 from Crypto.Random import get_random_bytes
6 import jks
7 import os
8
9 # Proceso de Descifrado
10 # -----
11 try:
12     path = os.path.dirname(__file__)
13     keystore = path + "/KeyStorePracticas"
14     ks = jks.KeyStore.load(keystore, "123456")
15     key=None
16     for alias, sk in ks.secret_keys.items():
17         if sk.alias == "cifrado-sim-aes-2566":
18             key = sk.key
19
20 if key is None:
21     print()
22     print("No se recupero la clave del KeyStore, el proceso no podra descifrar.")
23     print("-----")
24     print()
25 else:
26     print()
27     print("Se recupera la clave desde el keyStore:", key.hex())
28
29 # Se asignan los datos necesarios para poder hacer el descifrado del mensaje como son:
30 # la clave, el iv y el mensaje cifrado (este viene en base 64)
31 #clave=bytes.fromhex("A2CFF8B5901A5449E9C448BA5B948ABC4EE377152B3F1ACFA0148FB3A426D872")
32 clave=key
33 iv=bytes.fromhex("00000000000000000000000000000000")
34 texto_cifrado_bytes=b64decode("TQ9S0MKc6aF59S1xhfK9wI8UXpPCd505XF5J/5nLI7Of/oBQKIXg3nu1RRz4QnE1ezdrLAD5L04Ust3aB/150nnvv7b81G+1e12hpR84oI-")
35
36 cipher = AES.new( clave, AES.MODE_CBC, iv)
37 #mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size, style="pkcs7")
38
39 PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(venv) PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_2> python.exe .\aes_cbc_pkcs7.py
(venv) PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_2> []
```

Figura 7. Se modifica el alias de la clave desde Python

Una vez que se ha encontrado la clave, asignamos esta misma codificada previamente en hexadecimal (punto 4) a la variable **clave** (punto 5) cuyo objetivo es, no hacer más modificaciones en el nombre de nuestras variables para el proceso de descifrado, logrando así poder obtener la clave y descifrar el texto.



### Ejercicio 3.

Se requiere cifrar el texto **"KeepCoding te enseña a codificar y a cifrar"**. La clave para ello, tiene la etiqueta en el Keystore **"cifrado-sim-chacha-256"**. El nonce **"9Yccn/f5nJJhAt2S"**.

El algoritmo que se debe usar es un Chacha20.

**Respuesta 1:** El mensaje cifrado con el algoritmo Chacha20 es:

**69ac4ee7c4c552537a00a19bcaf7f0aaed7c9c8f769956a09bce6fadedf6c3  
535f2211c9467067cf5c4a842ab**

¿Cómo podríamos mejorar de forma sencilla el sistema, de tal forma, que no sólo garanticemos la confidencialidad sino, además, la integridad del mismo?

Se requiere obtener el dato cifrado, demuestra, tu propuesta por código, así como añadir los datos necesarios para evaluar tu propuesta de mejora.

**Respuesta 2:** Se puede mejorar el sistema implementando el algoritmo Chacha20 Poly 1305 debido a que particularmente este algoritmo cifra y además puede validar la autenticación del mensaje.

El mensaje cifrado con el algoritmo Chacha20 Poly 1305 es:

**4ec95921ca8b757e2336605c7dbab8f4d40b5b4d220e66aa978f740d3e59b  
0cd70e3217242991cc140bb1e1a**

## Procedimiento de la solución (Pregunta 1 - Ejercicio 3)

Se requiere cifrar el texto:

**"KeepCoding te enseña a codificar y a cifrar"**

La clave para ello, tiene la etiqueta en el Keystore **"cifrado-sim-chacha-256"**.

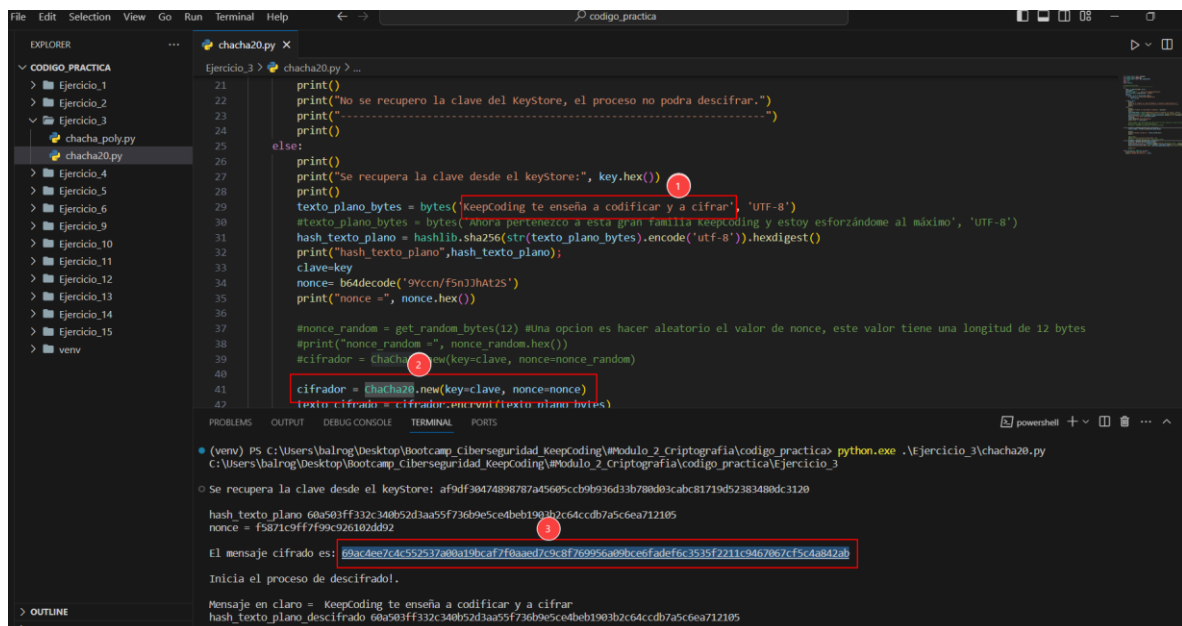
El nonce **"9Yccn/f5nJJhAt2S"**.

El algoritmo que se debe usar es un Chacha20.

**Respuesta 1: El mensaje cifrado con el algoritmo Chacha20 es:**

**69ac4ee7c4c552537a00a19bcaf7f0aaed7c9c8f769956a09bce6fadedf6c3535f2211c9467067cf5c4a842ab**

En la siguiente figura 1 se observa la implementación del algoritmo Chacha20 en Python, este mismo algoritmo se uso para poder cifrar el texto **"KeepCoding te enseña a codificar y a cifrar"**, la cadena resultante se puede observar en el punto 3.



```
21 print()
22 print("No se recupero la clave del KeyStore, el proceso no podra descifrar.")
23 print("-----")
24 print()
25 else:
26     print()
27     print("Se recupera la clave desde el keystore:", key.hex())
28     print()
29     texto_plano_bytes = bytes("KeepCoding te enseña a codificar y a cifrar", 'UTF-8')
30     @texto_plano_bytes = bytes("Ahora pertenezco a esta gran familia keepcoding y estoy esforzándome al máximo", 'UTF-8')
31     hash_texto_plano = hashlib.sha256(str(texto_plano_bytes).encode("utf-8")).hexdigest()
32     print("hash_texto_plano", hash_texto_plano)
33     clave=key
34     nonce= b64decode("9Yccn/f5nJJhAt2S")
35     print("nonce =", nonce.hex())
36
37     #nonce_random = get_random_bytes(12) #Una opcion es hacer aleatorio el valor de nonce, este valor tiene una longitud de 12 bytes
38     #print("nonce_random =", nonce_random.hex())
39     scifrador = ChaCha20(key=clave, nonce=nonce)
40
41     cifrador = ChaCha20.new(key=clave, nonce=nonce)
42     texto_cifrado = cifrador.encrypt(texto_plano_bytes)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(venv) PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo 2_Criptografia\codigo practica> python.exe .\Ejercicio_3\chacha20.py
C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo 2_Criptografia\codigo practica\Ejercicio_3
O Se recupera la clave desde el keystore: af9df30474898787a45605c9b936d33b780d93cab81719d52383480dc3120
hash_texto_plano 60a503ff332c340b52d3aa55f736b9e5c04beb1903b2c64cdb7a5c6ea712105
nonce = f5871c9ff7f99c926102dd92
El mensaje cifrado es: 69ac4ee7c4c552537a00a19bcaf7f0aaed7c9c8f769956a09bce6fadedf6c3535f2211c9467067cf5c4a842ab
Inicia el proceso de descifrado!.
Mensaje en claro = KeepCoding te enseña a codificar y a cifrar
hash_texto_plano_descifrado 60a503ff332c340b52d3aa55f736b9e5c04beb1903b2c64cdb7a5c6ea712105
```

Figura 1. Implementación de ChaCha20 en Python



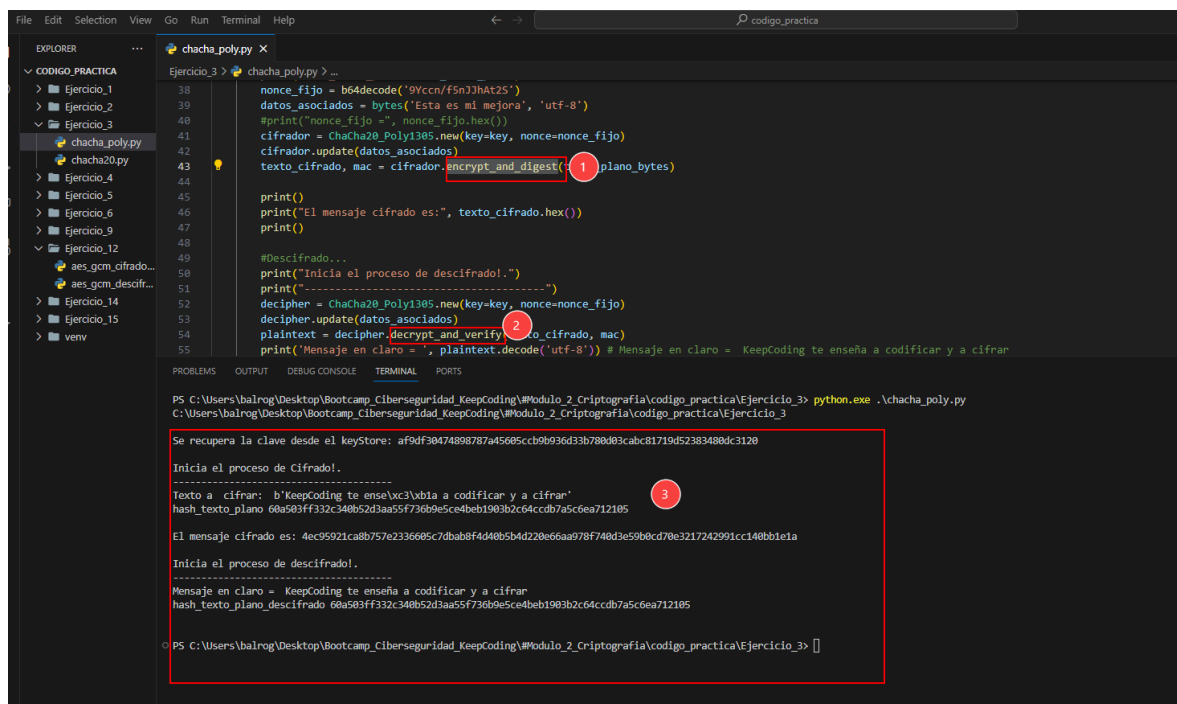
## Procedimiento de la solución (Pregunta 2 - Ejercicio 3)

¿Cómo podríamos mejorar de forma sencilla el sistema, de tal forma, que no sólo garanticemos la confidencialidad sino, además, la integridad del mismo?

Se requiere obtener el dato cifrado, demuestra, tu propuesta por código, así como añadir los datos necesarios para evaluar tu propuesta de mejora.

**Respuesta 2:** Se puede mejorar el sistema implementando el algoritmo ChaCha20 Poly 1305 debido a que particularmente este algoritmo cifra y además puede validar la autenticación del mensaje.

En la siguiente figura se observa la implementación del algoritmo ChaCha20\_poly\_1305 en Python debido a que este algoritmo nos permite realizar el cifrado y el autenticado de los datos, de modo que nos provee de confidencialidad y autenticación de la información.



```
File Edit Selection View Go Run Terminal Help
chacha_poly.py X
EXPLORER
CODIGO_PRACTICA
  Ejercicio_1
  Ejercicio_2
  Ejercicio_3
  chacha_poly.py
  chacha20.py
  Ejercicio_4
  Ejercicio_5
  Ejercicio_6
  Ejercicio_9
  Ejercicio_12
  aes_gcm_cifrado...
  aes_gcm_descifr...
  Ejercicio_14
  Ejercicio_15
  veru
Ejercicio_3 > chacha_poly.py > ...
38 nonce_fijo = b4decode('9vccn/f5nJhAt25')
39 datos_asociados = bytes('Esta es mi mejora', 'utf-8')
40 #print("nonce_fijo =", nonce_fijo.hex())
41 cifrador = ChaCha20_Poly1305.new(key=key, nonce=nonce_fijo)
42 cifrador.update(datos_asociados)
43 texto_cifrado, mac = cifrador.encrypt_and_digest(1 plano_bytes)
44
45 print()
46 print("El mensaje cifrado es:", texto_cifrado.hex())
47 print()
48
49 #Descifrado...
50 print("Inicia el proceso de descifrado.")
51 print("-----")
52 decipher = ChaCha20_Poly1305.new(key=key, nonce=nonce_fijo)
53 decipher.update(datos_asociados)
54 plaintext = decipher.decrypt_and_verify(2 to_cifrado, mac)
55 print("Mensaje en claro = ", plaintext.decode('utf-8')) # Mensaje en claro = KeepCoding te enseña a codificar y a cifrar

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_3> python.exe .\chacha_poly.py
C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_3

Se recupera la clave desde el keyStore: af9df30474898787a45605c9b936d33b780d03cab81719d52383480dc3120

Inicia el proceso de Cifrado.
-----
Texto a cifrar: b'KeepCoding te enseña a codificar y a cifrar'
hash_texto_plano 60a503ff332c340b52d3aa55f730b9e5ce4beb1903b2c64ccdb7a5c6ea712105

El mensaje cifrado es: 4ec95921ca8b757e2336605c7dbab8f4d40b5b4d220e6aa978f740d3e59b0cd70e3217242991cc140bb1e1a

Inicia el proceso de descifrado.
-----
Mensaje en claro = KeepCoding te enseña a codificar y a cifrar
hash_texto_plano_descifrado 60a503ff332c340b52d3aa55f730b9e5ce4beb1903b2c64ccdb7a5c6ea712105

PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_3> []
```

Figura 2. Implementación de ChaCha20\_Poly\_1305

#### Ejercicio 4.

Tenemos el siguiente jwt, cuya clave es "Con KeepCoding aprendemos".

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImIzTm9ybWFsIiwiaWF0IjoxNjY3OTMzMzZfQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE
```

¿Qué algoritmo de firma hemos realizado?

**Respuesta:** El algoritmo utilizado para la firma es HS256

¿Cuál es el body del jwt?

**Respuesta:** El payload del JWT es el json siguiente:

```
{
  "usuario": "Don Pepito de los palotes",
  "rol": "isNormal",
  "iat": 1667933533
}
```

Un hacker está enviando a nuestro sistema el siguiente jwt:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImIzQWRtaW4iLCJpYXQiOiE2Njc5MzMzMzZfQ.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNV2CIAODlHRI
```

¿Qué está intentando realizar?

**Respuesta:** Está intentando hacerse pasar por un token valido, sin embargo la parte de la verificación de la firma no es válida con la clave que nos han otorgado al inicio del enunciado. Quizás esté intentando hacer una escalación de privilegios al hacerse pasar por un usuario administrador.

¿Qué ocurre si intentamos validarlo con pyjwt?

**Respuesta:** Se intentó validar la firma del JWT pero fallo debido a que la firma no fue generada a partir de la clave dada en el enunciado.

## Procedimiento de la solución (Pregunta 1 - Ejercicio 4)

¿Qué algoritmo de firma hemos realizado?

**Respuesta:** El algoritmo utilizado para la firma es HS256, ver el punto 1, de la figura 1.

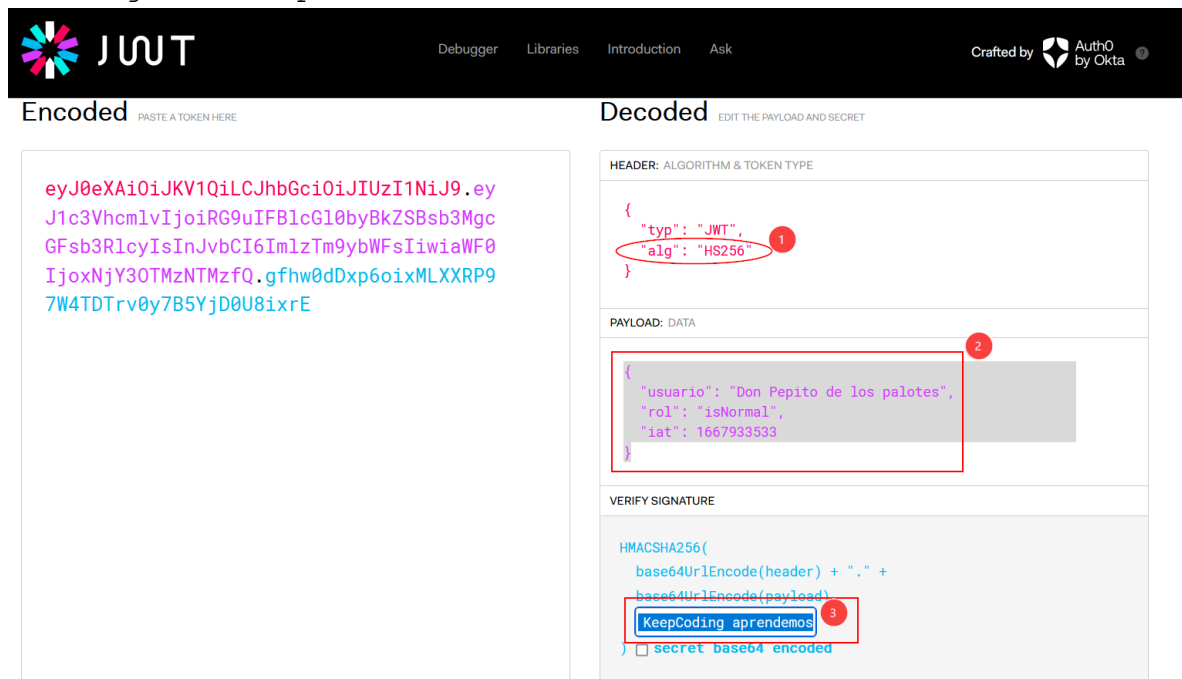
## Procedimiento de la solución (Pregunta 2 - Ejercicio 4)

¿Cuál es el body del jwt?

**Respuesta:** El payload del JWT es el json siguiente, ver punto 2 de la figura 1.

```
{
  "usuario": "Don Pepito de los palotes",
  "rol": "isNormal",
  "iat": 1667933533
}
```

En la siguiente imagen podemos ver el tipo de algoritmo utilizado para poder firmar el JWT (ver punto 1), así como el Payload (cuerpo) ver punto 2 y en la también la clave que se le asigno (ver punto 3).



The screenshot shows the JWT.io website interface. On the left, the 'Encoded' field contains a long JWT token. On the right, the 'Decoded' section shows the token's components:

- HEADER: ALGORITHM & TOKEN TYPE:** A JSON object with "typ": "JWT" and "alg": "HS256". A red circle with the number 1 highlights the "alg" field.
- PAYLOAD: DATA:** A JSON object with "usuario": "Don Pepito de los palotes", "rol": "isNormal", and "iat": 1667933533. A red box with the number 2 highlights the entire payload object.
- VERIFY SIGNATURE:** Shows the HMACSHA256 algorithm and the base64 encoded header and payload. A red box with the number 3 highlights the "KeepCoding" checkbox and the "aprendemos" text.

Figura 1. Validacion del JWT con la página JWT io

### Procedimiento de la solución (Pregunta 3 - Ejercicio 4)

¿Qué está intentando realizar?

**Respuesta:** Está intentando hacerse pasar por un token valido, sin embargo la parte de la verificación de la firma no es válida con la clave que nos han otorgado al inicio del enunciado. Quizás esté intentando hacer una escalación de privilegios al hacerse pasar por un usuario administrador.

En la siguiente figura se ha intentado validar la firma del JWT con la clave que se usó para firmar el token, claramente la validación fallo ver el output.



Figura 1. Validación de la firma del JWT

En la siguiente imagen podemos ver que en el Payload nos arroja en el campo de **rol:isAdmin** puede ser que esté intentando hacerse pasar por un token valido con permisos de Administrador, también se ha observado que en el punto 2 en el bloque de signature la verificación de la firma ha fallado pues la clave que se usó para firmar el token muestra una cadena distinta a la del JWT inicial.

The image shows the JWT.io web application interface. The top navigation bar includes the JWT logo, links for Debugger, Libraries, Introduction, and Ask, and a note 'Crafted by Auth0 by Okta'. The main content is split into two panels: 'Encoded' and 'Decoded'.

**Encoded Panel:** Contains a text input field with the following JWT token: `eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3Vhcm1vIjoiriRG9uIFB1cG10byBkZSBsb3MgcGFsb3RlcysInJvbmCI6Im1zQWRtaW4iLCJpYXQiOiJlE2Njc5MzM1MzN9.SsNEK94ze_NK_FTJvNSHxQoNhHg7PMq2vnW4rEy37Ag`. A red circle with the number '2' highlights the signature part of the token.

**Decoded Panel:** Displays the decoded components of the token.

- HEADER: ALGORITHM & TOKEN TYPE:** Shows `{ "typ": "JWT", "alg": "HS256" }`.
- PAYLOAD: DATA:** Shows `{ "usuario": "Don Peto de los palotes", "rol": "isAdmin", "iat": 1667933533 }`. A red circle with the number '1' highlights the `"rol": "isAdmin"` field.
- VERIFY SIGNATURE:** Shows the signature verification process: `HMACSHA256( base64UrlEncode(header) + "." + base64UrlEncode(payload), KeepCoding aprendemos )`. A red circle with the number '3' highlights the `KeepCoding aprendemos` string, which is the secret key used for verification.

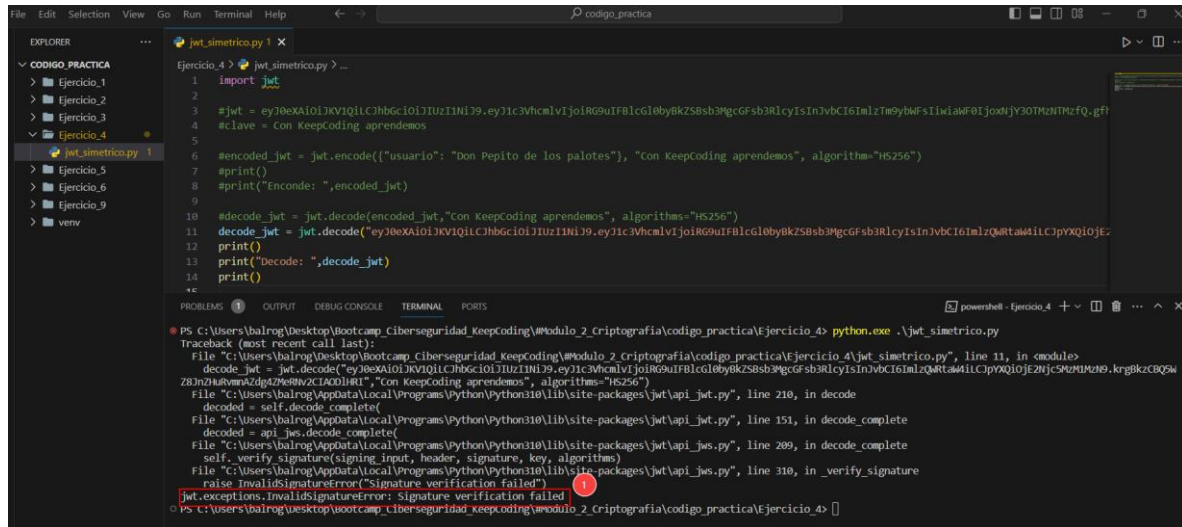
Figura 2. Decodificación del JWT a través de JWT io

## Procedimiento de la solución (Pregunta 4 - Ejercicio 4)

¿Qué ocurre si intentamos validarlo con pyjwt?

**Respuesta:** Se intentó validar la firma del JWT pero fallo debido a que la firma no fue generada a partir de la clave dada en el enunciado.

En la siguiente imagen se puede visualizar que ha fallado la verificación de la firma ver el punto 1.



```
File Edit Selection View Go Run Terminal Help
codigo_practica

EXPLORER
CODIGO_PRACTICA
  Ejercicio_1
  Ejercicio_2
  Ejercicio_3
  Ejercicio_4
  jwt_simetrico.py 1
  Ejercicio_5
  Ejercicio_6
  Ejercicio_9
  venv

jwt_simetrico.py 1
1 import jwt
2
3 #jwt = eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VhcnVlIjoiaR69uIF8lCGl0byBkZS8sb3MgcG9sb3RlcysIn3VbC16ImZlbn9ybmF5IiwiaWF0IjoxNjY3OTM2NTMzfqQ.gff
4 #clave = Con KeepCoding aprendemos
5
6 #encoded_jwt = jwt.encode({"usuario": "Don Pepito de los palotes"}, "Con KeepCoding aprendemos", algorithm="HS256")
7 #print()
8 #print("Encoded: ", encoded_jwt)
9
10 #decode_jwt = jwt.decode(encoded_jwt, "Con KeepCoding aprendemos", algorithm="HS256")
11 decode_jwt = jwt.decode("eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VhcnVlIjoiaR69uIF8lCGl0byBkZS8sb3MgcG9sb3RlcysIn3VbC16ImZlbn9ybmF5IiwiaWF0IjoxNjY3OTM2NTMzfqQ.gff", "Con KeepCoding aprendemos", algorithm="HS256")
12 print()
13 print("Decode: ", decode_jwt)
14
15

PROBLEMS
PS C:\Users\balrog\Desktop\bootcamp_ciberseguridad\keepcoding\modulo_2_criptografia\codigo_practica\Ejercicio_4> python.exe .\jwt_simetrico.py
Traceback (most recent call last):
  File "C:\Users\balrog\Desktop\bootcamp_ciberseguridad\keepcoding\modulo_2_criptografia\codigo_practica\Ejercicio_4\jwt_simetrico.py", line 11, in <module>
    decode_jwt = jwt.decode("eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VhcnVlIjoiaR69uIF8lCGl0byBkZS8sb3MgcG9sb3RlcysIn3VbC16ImZlbn9ybmF5IiwiaWF0IjoxNjY3OTM2NTMzfqQ.gff", "Con KeepCoding aprendemos", algorithm="HS256")
  File "C:\Users\balrog\AppData\Local\Programs\Python\Python310\lib\site-packages\jwt\api_jwt.py", line 210, in decode
    decoded = self.decode_complete(
  File "C:\Users\balrog\AppData\Local\Programs\Python\Python310\lib\site-packages\jwt\api_jwt.py", line 151, in decode_complete
    decoded = api_jwt.decode_complete(
  File "C:\Users\balrog\AppData\Local\Programs\Python\Python310\lib\site-packages\jwt\api_jwt.py", line 209, in decode_complete
    self.verify_signature(signing_input, header, signature, key, algorithms)
  File "C:\Users\balrog\AppData\Local\Programs\Python\Python310\lib\site-packages\jwt\api_jwt.py", line 310, in verify_signature
    raise InvalidSignatureError("Signature verification failed")
jwt.exceptions.InvalidSignatureError: Signature verification failed
PS C:\Users\balrog\Desktop\bootcamp_ciberseguridad\keepcoding\modulo_2_criptografia\codigo_practica\Ejercicio_4>
```

Figura 3. Decodificación del JWT a través de Python

### Ejercicio 5.

El siguiente hash se corresponde con un SHA3 Keccak del texto "En KeepCoding aprendemos cómo protegernos con criptografía".

**bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe**

¿Qué tipo de SHA3 hemos generado?

**Respuesta:** Es un hash SHA3256, hemos obtenido la misma cadena resultante:

**bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe**

Y si hacemos un SHA2, y obtenemos el siguiente resultado:

**4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833**

¿Qué hash hemos realizado?

**Respuesta:** Se ha realizado un hash usando el algoritmo SHA512 y hemos obtenido la misma cadena que el enunciado.

**4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833**

Genera ahora un SHA3 Keccak de 256 bits con el siguiente texto:

**"En KeepCodingaprendemos cómo protegernos con criptografía."**

¿Qué propiedad destacarías del hash, atendiendo a los resultados anteriores?

**Respuesta:** Se ha utilizado ahora un texto con un . al final, esto hace que sintácticamente el texto sea distinto al de inicio, por lo que el hash resultante será totalmente otro, esto hace referencia a la propiedad de difusión en la que si cambiamos el texto de entrada nos devolverá un nuevo hash.

## Procedimiento de la solución (Pregunta 1 - Ejercicio 5)

El siguiente hash se corresponde con un SHA3 Keccak del texto "En KeepCoding aprendemos cómo protegernos con criptografía".

bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

¿Qué tipo de SHA3 hemos generado?

**Respuesta:** Es un hash SHA3256, hemos obtenido la misma cadena resultante:

bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

En la siguiente imagen se puede observar la ejecución del programa usando algoritmo sha3\_256 ver el punto 1, en el punto 3 obtenemos el hash resultante.

```
File Edit Selection View Go Run Terminal Help
codigo_practica

EXPLORER
CODIGO_PRACTICA
  Ejercicio_1
  Ejercicio_2
  Ejercicio_3
  Ejercicio_4
  Ejercicio_5
  keccak.py
  venv

Ejercicio_5 > keccak.py
1 import hashlib
2
3 algorithm = hashlib.sha3_256()
4 print()
5 print("Nombre del algoritmo: ",algorithm.name)
6 print("Tamaño de BYTES: ",algorithm.digest_size)
7 #texto de entrada para calcular el hash
8 algorithm.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía","UTF-8"))
9 print("Hash resultante: ",algorithm.hexdigest())
10 print()
11

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_criptografia\codigo_practica\Ejercicio_5> python.exe .\keccak.py
Nombre del algoritmo: sha3_256
Tamaño de BYTES: 32
Hash resultante: bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe
PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_criptografia\codigo_practica\Ejercicio_5>
```

Figura 1. Calculo del Hash usando el algoritmo sha3\_256



## Procedimiento de la solución (Pregunta 2 - Ejercicio 5)

Y si hacemos un SHA2, y obtenemos el siguiente resultado:

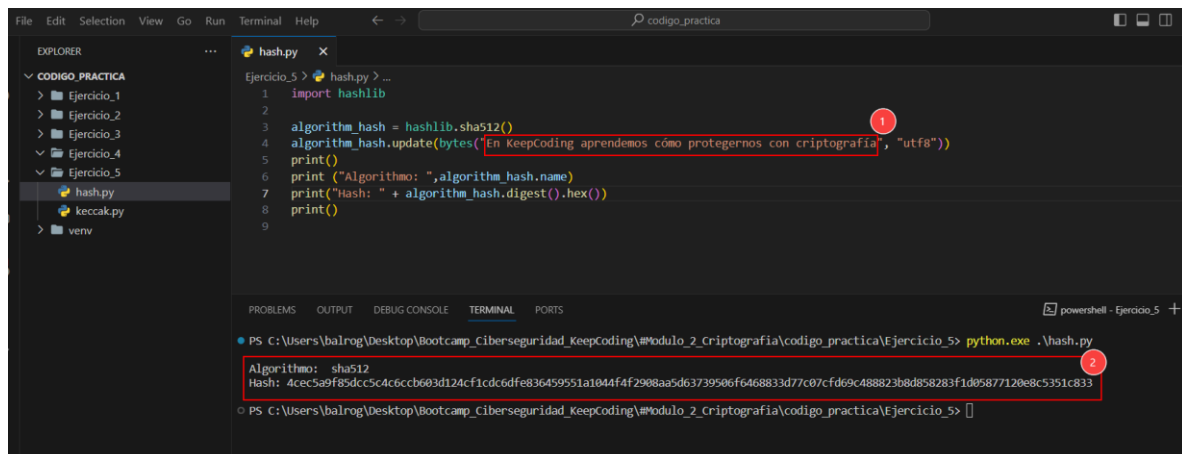
```
4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa
5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c53
51c833
```

¿Qué hash hemos realizado?

**Respuesta:** Se ha realizado un hash usando el algoritmo SHA512 y hemos obtenido la misma cadena que el enunciado.

```
4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa
5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c53
51c833
```

En la siguiente figura se ha realizado el cálculo del hash del texto "En KeepCoding aprendemos cómo protegernos con criptografía" se ha usado un el algoritmo SHA512, en el punto 2 podemos visualizar el hash resultante, siendo el mismo que nos han dado en el enunciado.



The screenshot shows a Visual Studio Code editor with a file explorer on the left showing a project structure with folders 'Ejercicio\_1' through 'Ejercicio\_5' and files 'hash.py', 'keccak.py', and 'venv'. The main editor window displays a Python script named 'hash.py' with the following code:

```
1 import hashlib
2
3 algorithm_hash = hashlib.sha512()
4 algorithm_hash.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía", "utf8"))
5 print()
6 print("Algoritmo: ", algorithm_hash.name)
7 print("Hash: " + algorithm_hash.digest().hex())
8 print()
9
```

Below the code, the terminal output is shown, indicating the execution of the script using 'python.exe .\hash.py'. The output displays the algorithm used and the resulting hash:

```
Algorithm: sha512
Hash: 4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833
```

Red circles with numbers 1 and 2 highlight the 'algorithm\_hash.update' line in the code and the resulting hash in the terminal output, respectively.

Figura 2. Cálculo del Hash usando el algoritmo sha2\_512

### Procedimiento de la solución (Pregunta 3, ejercicio 5)

Genera ahora un SHA3 Keccak de 256 bits con el siguiente texto:

**"En KeepCoding aprendemos cómo protegernos con criptografía."**

¿Qué propiedad destacarías del hash, atendiendo a los resultados anteriores?

**Respuesta:** Se ha utilizado ahora un texto con un . al final, esto hace que sintácticamente el texto sea distinto al de inicio, por lo que el hash resultante será totalmente otro, esto hace referencia a la propiedad de difusión en la que si cambiamos el texto de entrada nos devolverá un nuevo hash.

**El hash calculado es:**

**302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf**

En la figura 3 siguiente hemos calculado la función hash del texto (ver punto 1), nos ha arrojado un hash muy distinto y solo se ha hecho un pequeño cambio al texto original añadiendo un punto(.) al final, esto hace notar la propiedad de difusión pues ese mínimo cambio al texto se ha reflejado en el nuevo hash.

```
File Edit Selection View Go Run Terminal Help
codigo_practica

EXPLORER
CODIGO_PRACTICA
  Ejercicio_1
  Ejercicio_2
  Ejercicio_3
  Ejercicio_4
  Ejercicio_5
    hash.py
    keccak.py
  venv

Ejercicio_5 > keccakpy > ...
1 import hashlib
2
3 algorithm = hashlib.sha3_256()
4 print()
5 print("Nombre del algoritmo: ",algorithm.name)
6 print("Tamaño de BYTES: ",algorithm.digest_size)
7 #Texto de entrada para calcular el hash
8
9 # Ejercicio 5 - pregunta 1
10 algorithm.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía","UTF-8"))
11
12 # Ejercicio 5 - pregunta 3
13 algorithm.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía.", "UTF-8"))
14 print("Hash resultante: ",algorithm.hexdigest())
15 print()
16

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
powershell - Ejercicio_5

PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_5> python.exe .\keccak.py

Nombre del algoritmo: sha3_256
Tamaño de BYTES: 32
Hash resultante: 302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_5> |
```

Figura 3. Calculo del Hash usando el algoritmo sha3\_256

## Ejercicio 6.

Calcula el hmac-256 (usando la clave contenida en el Keystore) del siguiente texto:

**Siempre existe más de una forma de hacerlo, y más de una solución válida.**

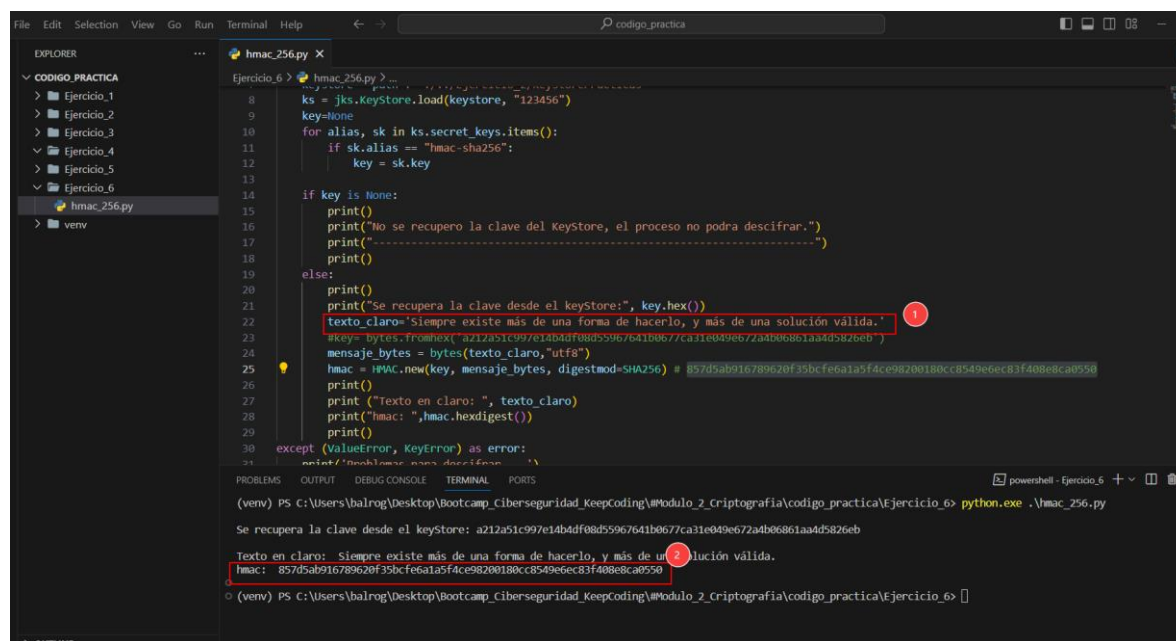
**Respuesta:** El hash resultante del texto de entrada es:

**857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550**

Se debe evidenciar la respuesta. Cuidado si se usan herramientas fuera de los lenguajes de programación, por las codificaciones es mejor trabajar en hexadecimal.

### Procedimiento de la solución (Pregunta 1 - Ejercicio 6)

Se puede apreciar en la siguiente figura que se recupera la clave del KeyStore desde Python, después se calcula el hmac256 del texto indicado (ver punto 1), finalmente obtenemos el hash ver punto 2.



```
File Edit Selection View Go Run Terminal Help
hmac_256.py X
Ejercicio_6 > hmac_256.py
8 ks = jks.KeyStore.load(keystore, "123456")
9 key=None
10 for alias, sk in ks.secret_keys.items():
11     if sk.alias == "hmac-sha256":
12         key = sk.key
13
14 if key is None:
15     print()
16     print("No se recupero la clave del KeyStore, el proceso no podra descifrar.")
17     print("-----")
18     print()
19 else:
20     print()
21     print("Se recupera la clave desde el keyStore:", key.hex())
22     texto_claro="Siempre existe más de una forma de hacerlo, y más de una solución válida."
23     #key= bytes.fromhex("a212a51c997e14b4df08d55967641b0677ca31e049e672a4b06861aa4d5826eb")
24     mensaje_bytes = bytes(texto_claro,"utf8")
25     hmac = HMAC.new(key, mensaje_bytes, digestmod=SHA256) # 857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550
26     print()
27     print("Texto en claro: ", texto_claro)
28     print("Hmac: ",hmac.hexdigest())
29     print()
30 except (ValueError, KeyError) as error:
31     print("No se pudo recuperar la clave desde el KeyStore.")
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(powershell - Ejercicio_6 + - -)
(venv) PS C:\Users\balog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_6> python.exe .\hmac_256.py
Se recupera la clave desde el keyStore: a212a51c997e14b4df08d55967641b0677ca31e049e672a4b06861aa4d5826eb
Texto en claro: Siempre existe más de una forma de hacerlo, y más de una solución válida.
Hmac: 857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550
(venv) PS C:\Users\balog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_6>
```

Figura 1. Calculo del Hmac256 en Python

## Ejercicio 7.

Trabajamos en una empresa de desarrollo que tiene una aplicación web, la cual requiere un login y trabajar con passwords. Nos preguntan qué mecanismo de almacenamiento de las mismas proponemos.

Tras realizar un análisis, el analista de seguridad propone un hash SHA-1. Su responsable, le indica que es una mala opción.

¿Por qué crees que es una mala opción?

**Respuesta:** Es una mala opción debido a que ese algoritmo ha sido vulnerado es decir a día de hoy ya no es seguro usar dentro del campo de la criptografía.

Después de meditarlo, propone almacenarlo con un SHA-256, y su responsable le pregunta si no lo va a fortalecer de alguna forma. ¿Qué se te ocurre?

**Respuesta:** Pues se puede ocupar el algoritmo SHA-256 + Salt (random de gran tamaño) + Pepper que se añade ya sea al inicio o al final del password para después obtener el hash.

Parece que el responsable se ha quedado conforme, tras mejorar la propuesta del SHA-256, no obstante, hay margen de mejora.

¿Qué propondrías?

**Respuesta:** Pues lo que propondrías sería ocupar la función de derivación de clave Argon2 que hasta el momento es el algoritmo más seguro a ataques de fuerza bruta.

### Ejercicio 8.

Tenemos la siguiente API REST, muy simple.

Request:

Post /movimientos

Campo	Tipo	Requiere Confidencialidad	Observaciones
idUsuario	Number	N	Identificador
Usuario	String	S	Nombre y Apellidos
Tarjeta	Number	S	

Petición de ejemplo que se desea enviar:

```
{"idUsuario": 1, "usuario": "José Manuel Barrio Barrio",  
"tarjeta":4231212345676891}
```

Response:

Campo	Tipo	Requiere Confidencialidad	Observaciones
idUsuario	Number	N	Identificador
movTarjeta	Array	S	Formato del ejemplo
Saldo	Number	S	Tendra formato 12300 para indicar 123.00
Moneda	String	N	EUR, DOLLAR

```
{  
  "idUsuario": 1,  
  "movTarjeta": [{  
    "id": 1,  
    "comercio": "Comercio Juan",  
    "importe": 5000  
  }, {  
    "id": 2,  
    "comercio": "Rest Paquito",  
    "importe": 6000  
  }  
}
```

```

    }],
    "Moneda": "EUR",
    "Saldo": 23400
}

```

Como se puede ver en el API, tenemos ciertos parámetros que deben mantenerse confidenciales. Así mismo, nos gustaría que nadie nos modificase el mensaje sin que nos enterásemos. Se requiere una redefinición de dicha API para garantizar la integridad y la confidencialidad de los mensajes. Se debe asumir que el sistema end to end no usa TLS entre todos los puntos.

**Respuesta 1:** Lo que yo propondría sería usar criptografía simétrica en la que se use 1 sola clave para poder cifrar la información que se va a compartir, cabe mencionar que esa clave se tendrá que compartir con un custodio para que de esta manera la clave sea resguardada de manera segura y pueda ser utilizada por las entidades/empresas/personas que tendrán interacción.

El algoritmo que propondría utilizar para cifrar la información es AES256 + GCM, pues este algoritmo tiene la bondad de poder cifrar y verificar la información.

La información que se tendrá que cifrar será el campo de **MovTarjeta** que es un arreglo y también el campo de **Saldo** pues se tiene que garantizar la confidencialidad. Para poder proveer integridad en la API se tiene que asignar en los datos asociados el campo de **Idusuario** y el campo de **Moneda** de esta manera se podrá hacer segura la Api y poder compartir la información por un medio de comunicación inseguro.

```

{
  "idUsuario": 1,
  "movTarjeta": [{
    "id": 1,
    "comercio": "Comercio Juan",
    "importe": 5000
  }, {
    "id": 2,
    "comercio": "Rest Paquito",
    "importe": 6000
  }],
  "Moneda": "EUR",
  "Saldo": 23400
}

```

## Ejercicio 9.

Se requiere calcular el KCV de las siguiente clave AES:

**A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72**

Para lo cual, vamos a requerir el KCV(SHA-256) así como el KCV(AES). El KCV(SHA-256) se corresponderá con los 3 primeros bytes del SHA-256.

**Respuesta:** El valor de KCV(SHA-256) que corresponde a los 3 primeros Bytes es: db7df2

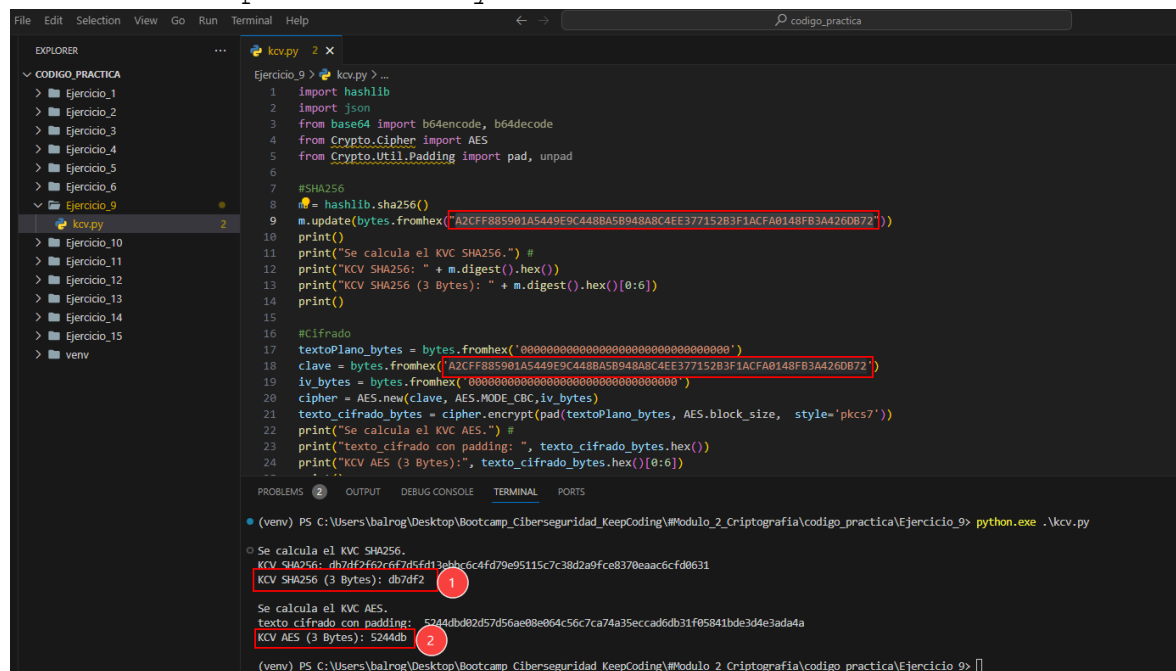
Mientras que el KCV(AES) se corresponderá con cifrar un texto del tamaño del bloque AES (16 bytes) compuesto con ceros binarios (00), así como un iv igualmente compuesto de ceros binarios.

**Respuesta:** El valor de KCV(AES) que corresponde a los 3 primeros Bytes es: 5244db

Obviamente, la clave usada será la que queremos obtener su valor de control.

## Procedimiento de la solución (Pregunta 1, ejercicio 9)

En la siguiente figura se puede apreciar el resultado obtenido y los valores de KCV AES Y KCV SHA-256, en los puntos 1 y 2 se observa los primeros 3 Bytes de cada KCV.



The image shows a VS Code editor with a Python script named `kvcpy.py` and its terminal output. The script calculates the KCV for a given AES key using SHA-256 and AES encryption. The terminal output shows the results of these calculations, with the first three bytes of each KCV highlighted and numbered 1 and 2.

```
File Edit Selection View Go Run Terminal Help
kvcpy 2 X
CODIGO_PRACTICA
  Ejercicio_1
  Ejercicio_2
  Ejercicio_3
  Ejercicio_4
  Ejercicio_5
  Ejercicio_6
  Ejercicio_9
  kvcpy 2
  Ejercicio_10
  Ejercicio_11
  Ejercicio_12
  Ejercicio_13
  Ejercicio_14
  Ejercicio_15
  venv
kvcpy 2 X
Ejercicio_9 > kvpy > ...
1 import hashlib
2 import json
3 from base64 import b64encode, b64decode
4 from Crypto.Cipher import AES
5 from Crypto.Util.Padding import pad, unpad
6
7 #SHA256
8 h = hashlib.sha256()
9 m.update(bytes.fromhex("A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"))
10 print()
11 print("Se calcula el KVC SHA256.") #
12 print("KVC SHA256: " + m.digest().hex())
13 print("KVC SHA256 (3 Bytes): " + m.digest().hex()[0:6])
14 print()
15
16 #Cifrado
17 textoPlano_bytes = bytes.fromhex("00000000000000000000000000000000")
18 clave = bytes.fromhex("A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72")
19 iv_bytes = bytes.fromhex("00000000000000000000000000000000")
20 cipher = AES.new(clave, AES.MODE_CBC, iv_bytes)
21 texto_cifrado_bytes = cipher.encrypt(pad(textoPlano_bytes, AES.block_size, style="pkcs7"))
22 print("Se calcula el KVC AES.") #
23 print("texto_cifrado con padding: ", texto_cifrado_bytes.hex())
24 print("KVC AES (3 Bytes):", texto_cifrado_bytes.hex()[0:6])
25
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(venv) PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_9> python.exe .\kvpy
Se calcula el KVC SHA256.
KVC SHA256: db7df2f60c6f7d5fd13ebc6c4fd79e95115c7c38d2a9fce8370eac6cf0631
KVC SHA256 (3 Bytes): db7df2 1
Se calcula el KVC AES.
texto_cifrado con padding: 5244dbd02d57d56ae08e064c56c7ca74a35eccad6db31f85841bde3d4e3ada4a
KVC AES (3 Bytes): 5244db 2
(venv) PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_9> |
```

Figura 1. Obtenemos los valores de Kcv de la Clave AES

### Ejercicio 10.

El responsable de Raúl, Pedro, ha enviado este mensaje a RRHH:

**Se debe ascender inmediatamente a Raúl. Es necesario mejorarle sus condiciones económicas un 20% para que se quede con nosotros.**

Lo acompaña del siguiente fichero de firma PGP (MensajeRespoDeRaulARRHH.txt.sig).

Nosotros, que pertenecemos a RRHH vamos al directorio a recuperar la clave para verificarlo. Tendremos los ficheros Pedro-priv.txt y Pedro-publ.txt, con las claves privada y pública.

Las claves de los ficheros de RRHH son RRHH-priv.txt y RRHH-publ.txt que también se tendrán disponibles.

**Se requiere verificar la misma, y evidenciar dicha prueba.**

**Respuesta:** Se verifico

Así mismo, se requiere firmar el siguiente mensaje con la clave correspondiente de las anteriores, simulando que eres personal de RRHH.

**Viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario. Saludos.**

**Respuesta:** Se firmo

Por último, cifra el siguiente mensaje tanto con la clave pública de RRHH como la de Pedro y adjunta el fichero con la práctica.

**Estamos todos de acuerdo, el ascenso será el mes que viene, agosto, si no hay sorpresas.**

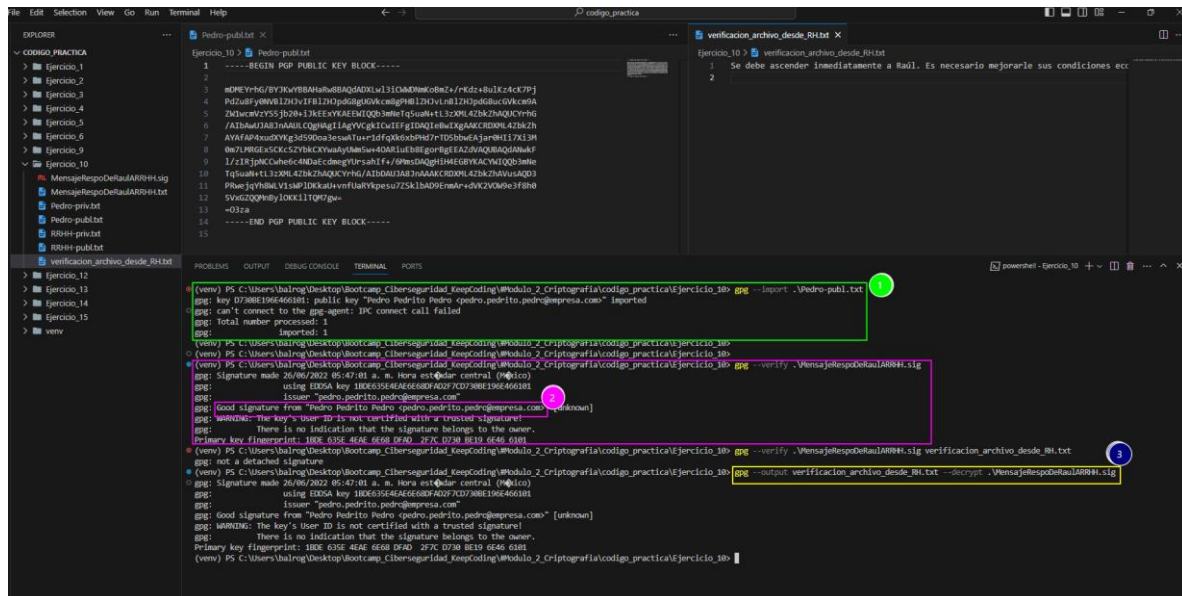
**Respuesta:** Se cifro el archivo



## Procedimiento de la solución (Pregunta 1 -Ejercicio 10)

Primeramente se tiene que importar la clave pública de Pedro (ver punto 1) pues al firmar y enviar el archivo .txt.sig lo ha firmado con su propia clave privada, para validar la firma, el equipo de RH tendrá que ocupar la llave publica de Pedro.

En el punto 2 se realiza la verificación del firma cuyo objetivo es garantizar (autenticidad) que el mensaje lo ha enviado Pedro (ver punto 2), finalmente se ha obtenido el mensaje enviado en un archivo .txt solo para fines de prueba.



```
gpg --import .Pedro-publ.txt
gpg: key 0780E19684668B1: public key "Pedro Pedro Pedro <pedro.pedrito.pedro@empresa.com>" imported
gpg: can't connect to the gpg-agent: IPC connect call failed
gpg: Total number processed: 1
gpg:      imported: 1

(venv) PS C:\Users\hainag\Desktop\bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo practica\Ejercicio_10> gpg --verify .MensajeRespOdeRauARRH.sig
gpg: Signature made 26/06/2022 05:47:01 +0200 using EDDSA key 18063554E668DFA073CD730E19684668B1
gpg: issuer "pedro.pedrito.pedro@empresa.com"
gpg: Good signature from "Pedro Pedro Pedro <pedro.pedrito.pedro@empresa.com>" [unknown]
gpg: WARNING: the key's user ID is not certified with a trusted signature!
gpg: There is no indication that the signature belongs to the owner.
Primary key fingerprint: 1806 3554 E668 DFA0 73CD 730E 1968 4668 B1

(venv) PS C:\Users\hainag\Desktop\bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo practica\Ejercicio_10> gpg --output verificacion_archivo_desde_RH.txt --decrypt .MensajeRespOdeRauARRH.sig
gpg: Signature made 26/06/2022 05:47:01 +0200 using EDDSA key 18063554E668DFA073CD730E19684668B1
gpg: issuer "pedro.pedrito.pedro@empresa.com"
gpg: Good signature from "Pedro Pedro Pedro <pedro.pedrito.pedro@empresa.com>" [unknown]
Primary key fingerprint: 1806 3554 E668 DFA0 73CD 730E 1968 4668 B1

(venv) PS C:\Users\hainag\Desktop\bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo practica\Ejercicio_10>
```

Figura 1. Validación de la firma digital al archivo de Pedro

## Procedimiento de la solución (Pregunta 2 -Ejercicio 10)

Se realiza el firmado de un mensaje con la clave privada, el mensaje lo envía RH a Pedro, en la siguiente imagen se puede apreciar el archivo (ver punto 1) y en el contenido del mismo se encuentra el mensaje, en el punto 2 se ha importado la llave privada de RH, para lograr la correcta importación, nos pidió ingresar el passphrase asociado, en el punto 3 se ha construido el archivo firmado **.sig**, por cuestiones de seguridad nos ha pedido el passphrase de la llave privada para así poder generar el archivo firmado que se tiene que mandar a Pedro.

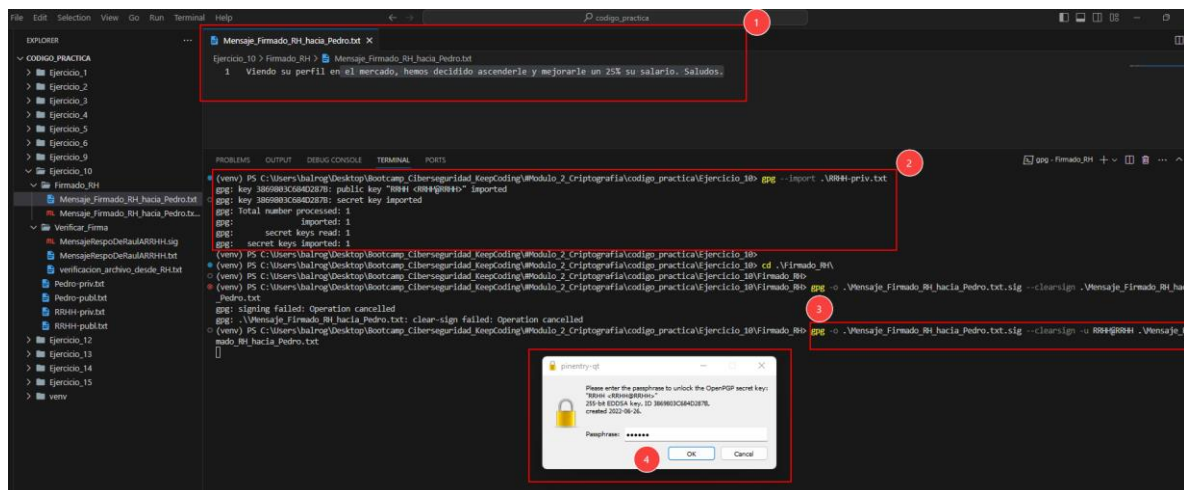


Figura 2. Firmado de un archivo con la llave privada

En la siguiente figura podemos apreciar el contenido del archivo firmado **.sig**.

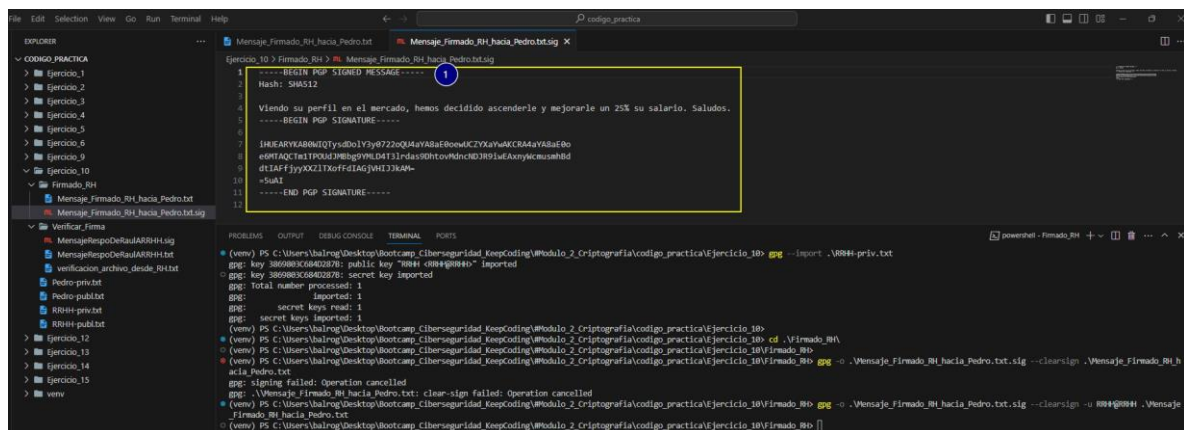


Figura 3. Se muestra el archivo resultante al aplicar la firma con la llave privada de RH

## Procedimiento de la solución (Pregunta 3 -Ejercicio 10)

Para poder realizar el cifrado del mensaje con las claves públicas de RH y de Pedro es necesario importartas, después crear el archivo con el mensaje que se desea cifrar, finalmente se ejecutan los comandos utilizando gpg para poder cifrar el mensaje ocupando las llaves públicas.

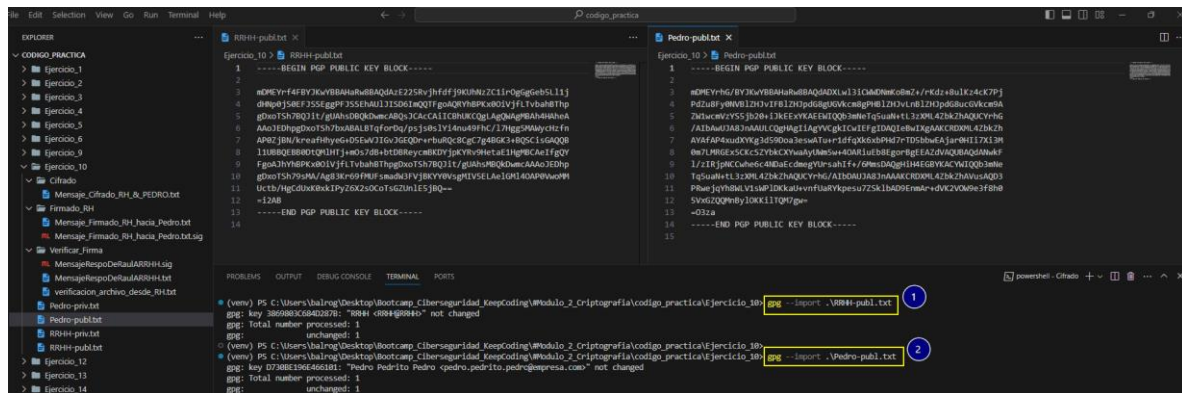


Figura 4. Importación de llaves gpg públicas

En la siguiente imagen se puede observar el cifrado de un archivo usando los UID's de las llaves públicas de RH (F2B1D0E8958DF2D3BDB6A1053869803C684D287B) y de Pedro (1BDE635E4EAE6E68DFAD2F7CD730BE196E466101), finalmente en el punto 4 se construye el archivo cifrado .gpg.

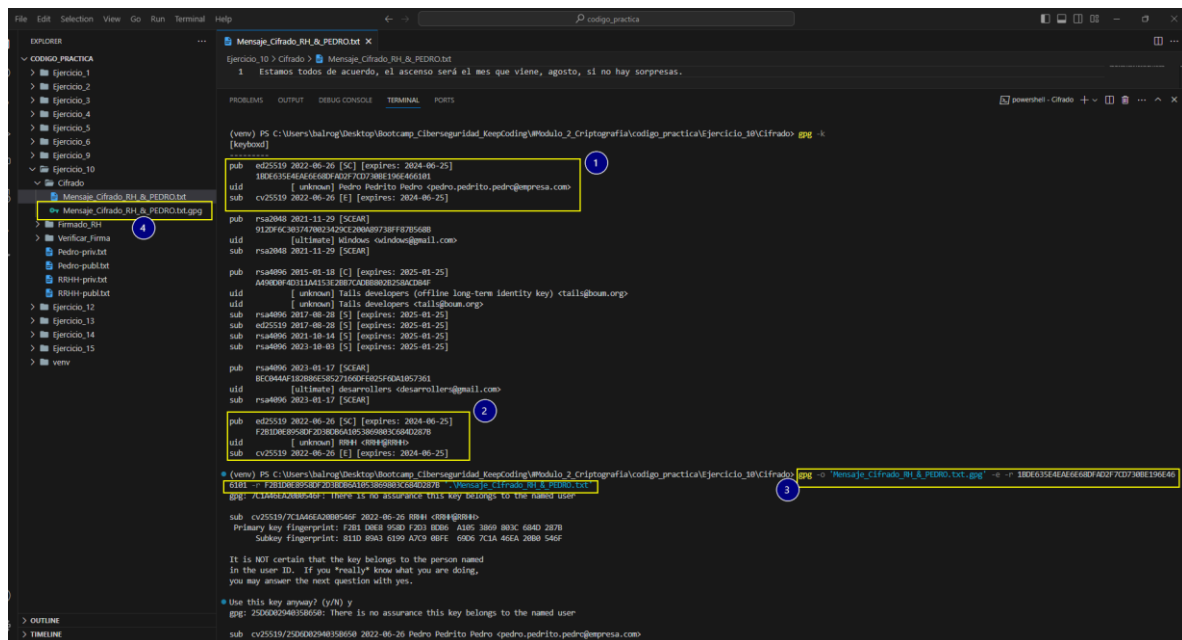


Figura 5. Cifrado de un archivo con las claves públicas

## Ejercicio 11.

Nuestra compañía tiene un contrato con una empresa que nos da un servicio de almacenamiento de información de videollamadas. Para lo cual, la misma nos envía la clave simétrica de cada videollamada cifrada usando un RSA-OAEP. El hash que usa el algoritmo interno es un SHA-256.

El texto cifrado es el siguiente:

```
b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e170
9b30c96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d04e3
d3d4ad629793eb00cc76d10fc00475eb76bfbcb1273303882609957c4c0ae2c4f5b
a670a4126f2f14a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8
624071be78cceef573d896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f6
7d3f98b4cd907f27639f4b1df3c50e05d5bf63768088226e2a9177485c54f72407
fdf358fe64479677d8296ad38c6f177ea7cb74927651cf24b01dee27895d4f05fb
5c161957845cd1b5848ed64ed3b03722b21a526a6e447cb8ee
```

Las claves pública y privada las tenemos en los ficheros clave-rsa-oaep-publ.pem y clave-rsa-oaep-priv.pem.

Si has recuperado la clave, vuelve a cifrarla con el mismo algoritmo. ¿Por qué son diferentes los textos cifrados?

**Respuesta:** La clave simetría que se obtuvo es:

e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

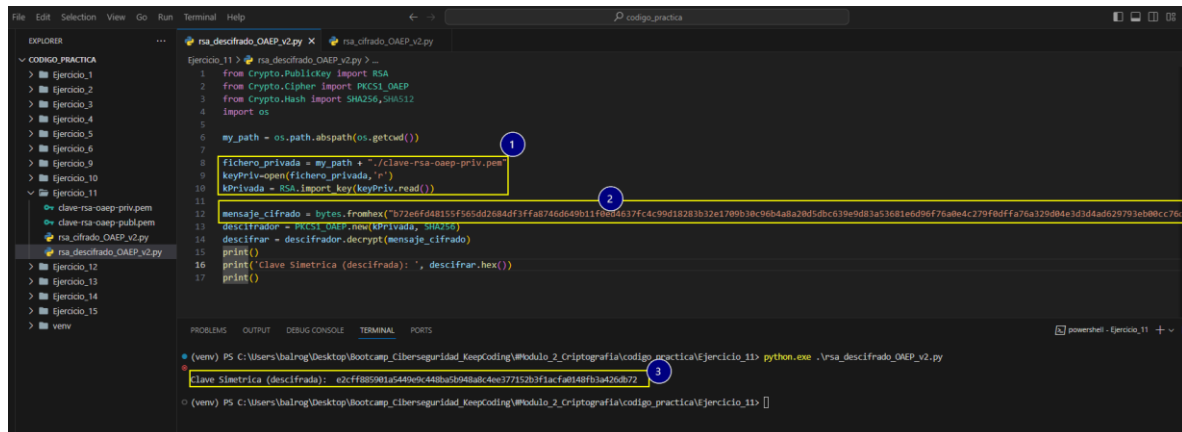
Se ha vuelto a cifrar la clave obtenida nuevamente con el mismo algoritmo usando la clave pública, evidente mente nos hemos percatado que ambos textos tanto el texto inicial como el texto final no son iguales debido a que es una característica del padding que usa el algoritmo esto hace que la cadena resultante sea aleatoria lo cual provee mecanismos de seguridad.

Texto resultante al cifrar la clave simétrica por segunda ocasión:

```
745867f18b2372d3513f95eab2dc0db0d01ac62c63bf16f5e7b06165df8c972115
988bf7692ed38c1ed0cb01dcb6ec75b7a560270c4d28326adaf75380612da74fa1
f467bcc9bbb263dc69e392b3626fc8d819c746241636d0c5d6400ba259dd71084d
ac07f27039a2bc68202c7a679f45ddeedfc635751d2b4266e05692275ee4c2f078
44e41faca110354773c8ae7e8fac606a622fb59bd93ecca74f970c7b2951367c77
b01b1100fc6f0a3a21e0deaac5516a750990a73a52b66a2665dec27f4e91bcd1bd1
fa954b2b1114430a51fee85492324c00c608d28ce58f5c0cf2bab9f21b982545b2
0da5421d2aa9623e383bea8fbd1383544c4a111c650e215fda
```

## Procedimiento de la solución (Pregunta 1 -Ejercicio 11)

Una vez que hemos recibido la clave simétrica de manera cifrada (cifrada con la llave pública), procedemos a descifrar con el mismo algoritmo (RSA-OAEP) que se ha usado para cifrar, adicionalmente es necesario contar con la llave privada para poder descifrar dicha llave, en la imagen siguiente se puede observar que se ha logrado obtener la clave (ver punto 3):



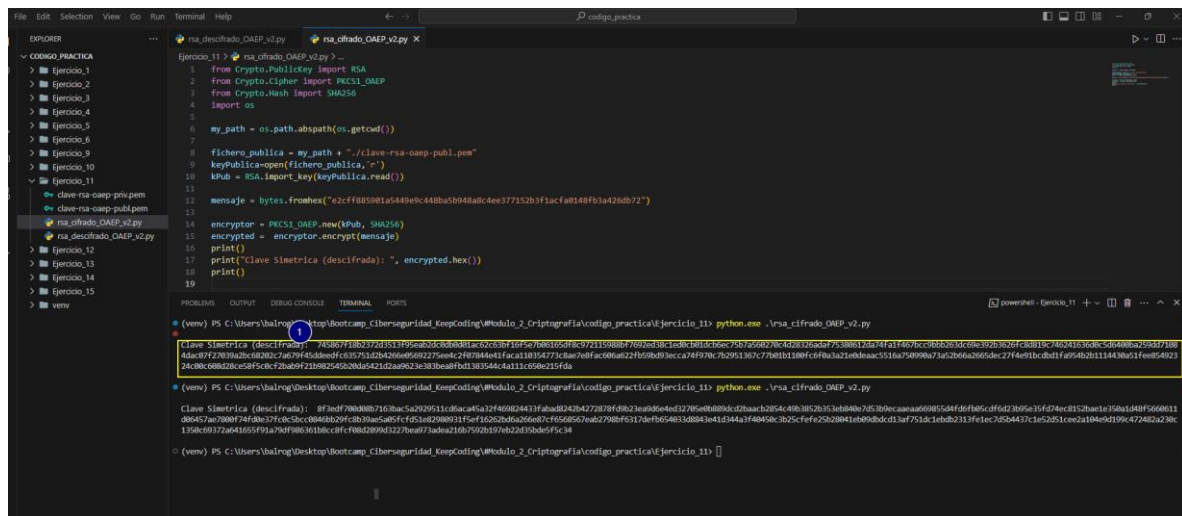
```
1 from Crypto.PublicKey import RSA
2 from Crypto.Cipher import PKCS1_OAEP
3 from Crypto.Hash import SHA256
4 import os
5
6 my_path = os.path.abspath(os.getcwd())
7
8 fichero_privada = my_path + "\\clave-rsa-oaep-priv.pem"
9 keyPriv=open(fichero_privada, "r")
10 kPriv = RSA.import_key(keyPriv.read())
11
12 mensaje_cifrado = bytes.fromhex("b72e6fd4815f565d2684df3ffa8746d649b11f8d4637fc4c9d18283b32e1789b38c9eb4a8a20d5dbc39e9d83a53681e6d96f76a0e4c279fedffa76a329d84e3d3d4a629793eb0cc76")
13
14 descifrador = PKCS1_OAEP.new(kPriv, SHA256)
15 descifrar = descifrador.decrypt(mensaje_cifrado)
16 print()
17 print('Clave Simétrica (descifrada): ', descifrar.hex())
18 print()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(venv) PS C:\Users\baig\Desktop\bootcamp_ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_11> python.exe .\rsa_descifrado_OAEP_v2.py
Clave Simétrica (descifrada): e2cfff885901a5449ebc440ba5b94b4c4ee377152b3facfa0148fb3a426db72
```

Figura 1. Descifrado del mensaje para obtener la clave simétrica.

Se ha intentado cifrar nuevamente la clave simétrica con la clave pública usando RSA-OAEP y nos hemos dado cuenta que la cadena cifrada no es la misma que nos han entregado inicialmente, con esto confirmamos una propiedad de la criptografía asimétrica (clave pública) que es la aleatoriedad.



```
1 from Crypto.PublicKey import RSA
2 from Crypto.Cipher import PKCS1_OAEP
3 from Crypto.Hash import SHA256
4 import os
5
6 my_path = os.path.abspath(os.getcwd())
7
8 fichero_publica = my_path + "\\clave-rsa-oaep-pub.pem"
9 keyPublica=open(fichero_publica, "r")
10 kPub = RSA.import_key(keyPublica.read())
11
12 mensaje = bytes.fromhex("e2cfff885901a5449ebc440ba5b94b4c4ee377152b3facfa0148fb3a426db72")
13
14 encryptor = PKCS1_OAEP.new(kPub, SHA256)
15 encrypted = encryptor.encrypt(mensaje)
16 print()
17 print('Clave Simétrica (descifrada): ', encrypted.hex())
18 print()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(venv) PS C:\Users\baig\Desktop\bootcamp_ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_11> python.exe .\rsa_cifrado_OAEP_v2.py
Clave Simétrica (descifrada): 7458b7f1b237d513195ea20c3d0858acac3c6f10f5e50b1050f8c972115988b7692d38c1e6c3b6c75b7a568078c402836def7a380132a74f411670cc0b2b33c6e392b367fcb815c74e241b36dc36408a2580710
4dab0727839a2bcb02027a679465ddedc8351512b426e0f592275e0c2407846e11fac118354773c4e70f8fac0b06022165b093eccc74f978c72951367c770b1100fc0ba3a21e0baac551ba750990a73a2b66a2605dec27f4e91bcbdf6954b2b11443ba51fe0e4923
26c8c6e026e5a5f5c0e22080254020a454321d2a8023e3838a0b1b383544c4111650e021f6a

(venv) PS C:\Users\baig\Desktop\bootcamp_ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_11> python.exe .\rsa_cifrado_OAEP_v2.py
Clave Simétrica (descifrada): 8f3ed770a0b73163ac2a2029511d0acaf5a12f400824431f4bad1242427287f1d923e0d0e4d32785e0b0b0dc2baac32854c49b3852b353eb0ba70530becaa0a6085d04f0f0b0c0f5d23095a35f474ec81212bae3a39a5408f560611
206527a0780f74f0b37f0c82cc0a0229f1cb39a5405fcf053a82908011f4ef162620b0a266d0c76508567eab279dbf6317d0f0540134084344344a3f40458c3b25cf6e2520801e0b0d0c03af753d1ebd2113f0ec70504437c1e2d51ce0a109ed199c47482a230c
130e09372a4d505f91a39f0b030303c0f040209432270e073a0a210b7920b3930b22050d0c9fc4
```

Figura 2. Se ha vuelto a cifrar la clave obtenida (simétrica) con la clave pública.

## Ejercicio 12.

Nos debemos comunicar con una empresa, para lo cual, hemos decidido usar un algoritmo como el AES/GCM en la comunicación. Nuestro sistema, usa los siguientes datos en cada comunicación con el tercero:

Key:

**E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74**

Nonce: **9Yccn/f5nJJhAt2S**

¿Qué estamos haciendo mal?

**Respuesta:** El nonce que se nos ha asignado de inicio es fijo y debería ser aleatorio, adicionalmente mencionar que hay que obtener el MAC/TAG para poder compartirlo, pues quien vaya a descifrar el mensaje necesitara esta información para poder recuperar el texto en claro.

Cifra el siguiente texto:

**He descubierto el error y no volveré a hacerlo mal**

Usando para ello, la clave, y el nonce indicados. El texto cifrado preséntalo en hexadecimal y en base64.

**Respuesta:**

**Cadena cifrada en formato Hexadecimal:**

**0fff75c264ef54986089c43d639ead8ab567da719af3567a77b4e83119d524ae80005b4fc45eb752b746f4e59f91a1008ae2a8**

**Cadena en Base64:**

**D/91wmTvVJhgicQ9Y56tirVn2nGa81Z6d7ToMRnVJK6AAftPx°F63UrdG9OWfk  
aEAiuKo**



## Procedimiento de la solución (Pregunta 2 -Ejercicio 12)

En la siguiente figura se muestra el cifrado usando AES/GCM, podemos apreciar que se han hecho uso de la clave y el nonce dado en el enunciado, ver punto 1, finalmente en la terminal de VsCode observamos la cadena cifrada en formato hexadecimal y en Base64 (ver punto 2), adicionalmente se ha descifrado la cadena y se ha obtenido el texto en claro ver punto 3.

```
6 #Cifrado
7 textoPlano_bytes = bytes('He descubierto el error y no volveré a hacerlo mal', 'UTF-8')
8 clave = bytes.fromhex('E2CF8B5901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148F83A4260B74')
9 nonce = bytes('9Yccn/f5nJhAT2S', 'utf8')
10 datos_asociados_bytes = bytes('', 'UTF-8')
11
12
13 cipher = AES.new(clave, AES.MODE_GCM, nonce=nonce)
14 cipher.update(datos_asociados_bytes)
15 texto_cifrado_bytes, tag = cipher.encrypt_and_digest(textoPlano_bytes)
16
17 b64 = b64encode(bytes.fromhex(texto_cifrado_bytes.hex())).decode()
18 print()
19 print("texto_cifrado_hexadecimal: " + texto_cifrado_bytes.hex())
20 print("texto_en_base_64: ", b64)
21 print("tag: " + tag.hex())
22 print()
23
```

```
(venv) PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_12> python.exe .\aes_gcm_cifrado.py
texto_cifrado_hexadecimal: 0fff75c264ef54986089c43d39eadab567da719af3567a7b4a83119d524ae0005b4fc45eb752b746f4e59f91a1008ae2a8
texto_en_base_64: D/9hwtVWZhpicQY96tirVh2Kga8126dT0PwVnZK6AFtPxf63LrdGcNfkaEAluko
tag: ad737aea100b866f2417f7cd00771c9
(venv) PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_12> python.exe .\aes_gcm_descifrado.py
El texto en claro es: He descubierto el error y no volveré a hacerlo mal
(venv) PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_12>
```

Figura 1. Cadena cifrada mostrada en Hexadecimal y en Base64

### Ejercicio 13.

Se desea calcular una firma con el algoritmo PKCS#1 v1.5 usando las claves contenidas en los ficheros **clave-rsa-oaep-priv** y **clave-rsa-oaep-publ.pem** del mensaje siguiente:

**El equipo está preparado para seguir con el proceso, necesitaremos más recursos.**

¿Cuál es el valor de la firma en hexadecimal?

**Respuesta:** El valor es:

**a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063  
950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596  
c1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709  
b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a9ec798fe66308  
e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe0318532775**

69f8383846d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a08752  
08f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d9297  
8a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dba  
e21f0aaaa6f9b9d59f41928d

Calcula la firma (en hexadecimal) con la curva elíptica  
ed25519, usando las claves **ed25519-priv** y **ed25519-publ**.

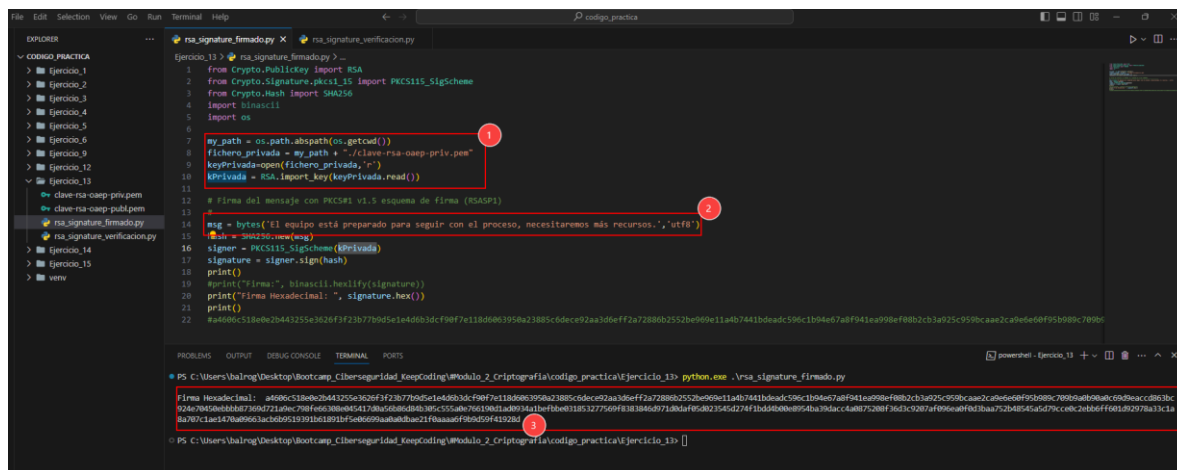
**Respuesta:** El valor es:

bf32592dc235a26e31e231063a1984bb75ffd9dc5550cf30105911ca4560d  
ab52abb40e4f7e2d3af828abac1467d95d668a80395e0a71c51798bd54469  
b7360d

**Nota:** Si me fue posible validar la firma 😊

### Procedimiento de la solución (Pregunta 1, ejercicio 13)

En la siguiente imagen se muestra la ejecución del programa en Python, podemos apreciar que se está realizando el firmado de un texto que se nos ha compartido. En el punto 1 se hace la importación de la llave privada, en el punto 2 se hace referencia al texto que debemos ocupar, en el punto 3 finalmente obtenemos la firma en formato Hexadecimal.



```
1 from Crypto.PublicKey import RSA
2 from Crypto.Signature.pkcs1_15 import PKCS115_SigScheme
3 from Crypto.Hash import SHA256
4 import os
5
6 my_path = os.path.abspath(os.getcwd())
7 fichero_privada = my_path + '/../clave-ria-oeap-priv.pem'
8 keyPrivada=open(fichero_privada, 'r')
9 kPrivada = RSA.import_key(keyPrivada.read())
10
11 # Firma del mensaje con PKCS1 v1.5 esquema de firma (RSAPSS)
12
13 msg = bytes('El equipo está preparado para seguir con el proceso, necesitaremos más recursos.', 'utf8')
14 hsh = SHA256.new(msg)
15 signer = PKCS115_SigScheme(kPrivada)
16 signature = signer.sign(hsh)
17 print()
18 #verificamos la firma
19 print('Firma Hexadecimal: ', signature.hex())
20 print()
21 print()
22 #a406c518b9c20441255a3626f3f23b779d5e1e40b53dcf98f7e1806063958a2385c6dce92aa3d0eff2a72808b2552be90e11a4b7441bdad596c1b94e67a0f941ea998e98b2c33a925c959bc0a02ca9ede60f95980c78900
```

Problems OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\balrog\Desktop\laptop\cap_ciberseguridad_keepcoding\Modulo_2_Criptografia\codigo_practica\ejercicio_13> python.exe .\rsa_signature_firmado.py
Firma Hexadecimal: a406c518b9c20441255a3626f3f23b779d5e1e40b53dcf98f7e1806063958a2385c6dce92aa3d0eff2a72808b2552be90e11a4b7441bdad596c1b94e67a0f941ea998e98b2c33a925c959bc0a02ca9ede60f95980c78900
PS C:\Users\balrog\Desktop\laptop\cap_ciberseguridad_keepcoding\Modulo_2_Criptografia\codigo_practica\ejercicio_13>
```

Figura 1. Se calcula la firma de un texto usando el algoritmo PKCS#1 v1.5



En la siguiente imagen se implementó la parte que corresponde a la verificación de la firma usando la llave pública, para así demostrar al receptor del mensaje que, el emisor del mensaje es quien dice ser. En los puntos marcados se muestra la carga de la llave publica, en el punto 2 se usa el texto original y la firma obtenida al inicio, en el punto 3 nos arroja el resultado de la validación siendo exitosa.

```

1 from Crypto.PublicKey import RSA
2 from Crypto.Signature.pkcs1_5 import PKCS1_5_SigScheme
3 from Crypto.Hash import SHA256
4 import binascii
5 import os
6
7 my_path = os.path.abspath(os.getcwd())
8 fichero_publica = my_path + "/clave-rsa-osep-publ.pem"
9 keyPublica=open(fichero_publica,"r")
10 kPublica = RSA.import_key(keyPublica.read())
11
12 # Verificar la firma con PKCS1_5 v1.5 (R4096)
13
14 msg = bytes("El equipo está preparado para seguir con el proceso, necesitamos más recursos.", "utf8")
15 hash = SHA256.new(msg)
16 signature = bytes.fromhex("a606c51be02644325e3626f3f22b7706d5e1e4d03dcf907e11d0d603958a2385c5dec92aa38eff2a72886b2552be90e11a4b7441bdeac596c1b946e7abf941ea998ef8b2cb34925c959b")
17 verifier = PKCS1_5_SigScheme(kPublica)
18 try:
19     verifier.verify(hash, signature)
20     print()
21     print("Firma válida.")
22     print()
23 except:
24     print()
25     print("Firma inválida.")
26

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\ejercicio_13> python.exe .\rsa_signature_verificacion.py
Firma válida.
PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\ejercicio_13>

```

Figura 2. Se ha verificado la firma del texto original para corroborar la autenticidad  
**Procedimiento de la solución (Pregunta 2, ejercicio 13)**

En la siguiente imagen se puede observar que se ha obtenido la firma usando la curva elíptica ed25519 del texto que nos han proporcionado, en el punto 1 cargamos la clave privada, en el punto 2 se muestra el mensaje a firmar, finalmente en el punto 3 obtenemos la firma en formato hexadecimal.

```

1 import ed25519
2 import os
3
4 my_path = os.path.abspath(os.getcwd())
5 fichero_privada = my_path + "/ed25519-priv"
6 keyPrivada=open(fichero_privada,"r").read()
7 #privatkey = open("./ed25519-priv","r").read()
8
9 #signedkey = ed25519.SigningKey(privatkey)
10 #signedkey = ed25519.SigningKey(keyPrivada)
11 msg = bytes("El equipo está preparado para seguir con el proceso, necesitamos más recursos.", "utf8")
12 signature = signedkey.sign(msg, encoding="hex")
13 print()
14 print("Firma Generada (64 bytes):", signature)
15 print("Firma Hexadecimal:", signature.hex())
16 print()
17

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

(nvm) PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\ejercicio_13> python.exe .\ed25519_firmado.py
Firma Generada (64 bytes): b'\xf3252d2c235a26e31e231061a1084bb75ff40dc5506c780189911c40500a52ab04bf7e2d1af82b0ac146149560004081950eb71c31708b24460b700d'
Firma Hexadecimal: f3252d2c235a26e31e231061a1084bb75ff40dc5506c780189911c40500a52ab04bf7e2d1af82b0ac146149560004081950eb71c31708b24460b700df32603373161351373938626435343639623733363064
(nvm) PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\ejercicio_13>

```

Figura 1. Se obtiene la firma hexadecimal usando curvas elípticas

Se ha utilizado el valor resultante de la línea 15 como se me indicó en el chat. Se ha intentado verificar nuevamente la firma de origen, podemos observar que la firma es válida ver punto 3 de la imagen siguiente.

```
File Edit Selection View Go Run Terminal Help
EXPLORER
CODIGO_PRACTICA
  Ejercicio_1
  Ejercicio_2
  Ejercicio_3
  Ejercicio_4
  Ejercicio_5
  Ejercicio_6
  Ejercicio_9
  Ejercicio_10
  Ejercicio_11
  Ejercicio_12
  Ejercicio_13
    ed25519
      ed25519_firmado.py
      ed25519_verificacion.py
    ed25519_priv
    ed25519_publ
    rsa_signature_firmado.py
    rsa_signature_verificacion.py
  Ejercicio_14
  Ejercicio_15
  venv

Ejercicio_13 > ed25519_verificacion.py x
1 import ed25519
2 import os
3
4 my_path = os.path.abspath(os.getcwd())
5 fichero_publica = my_path + "/ed25519-publ"
6 keyPublica=open(fichero_publica,'rb').read()
7
8 firma = bytes.fromhex("bf32592dc235a26e31e231063a1984bb75ff9dc5550cf30105911ca4560dab52abb40e4f7e2d3af828abac1467d95d668a08395e0a71c51798bd54469b7360d")
9 keyPublica=open("ed25519-publ","r").read()
10 msg = bytes('El equipo está preparado para seguir con el proceso, necesitaremos más recursos.','utf8')
11
12 try:
13     verifyKey = ed25519.VerifyingKey(keyPublica.hex(),encoding="hex")
14     verifyKey.verify(firma.hex(), msg, encoding='hex')
15     print()
16     print("La firma es Válida")
17     print()
18 except:
19     print()
20     print("Firma Inválida")
21     print()

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(venv) PS C:\Users\bairog\Desktop\bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_13> python.exe .\ed25519_verificacion.py
La firma es Válida
(venv) PS C:\Users\bairog\Desktop\bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_13>
```

Figura 2. Verificación correcta de la firma con curva elíptica

## Ejercicio 14.

Necesitamos generar una nueva clave AES, usando para ello una HKDF (HMAC-based Extract-and-Expand key derivation function) con un hash SHA-512.

La clave maestra requerida se encuentra en el keystore con la etiqueta "cifrado-sim-aes-256". La clave obtenida dependerá de un identificador de dispositivo, en este caso tendrá el valor en hexadecimal:

**e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3**

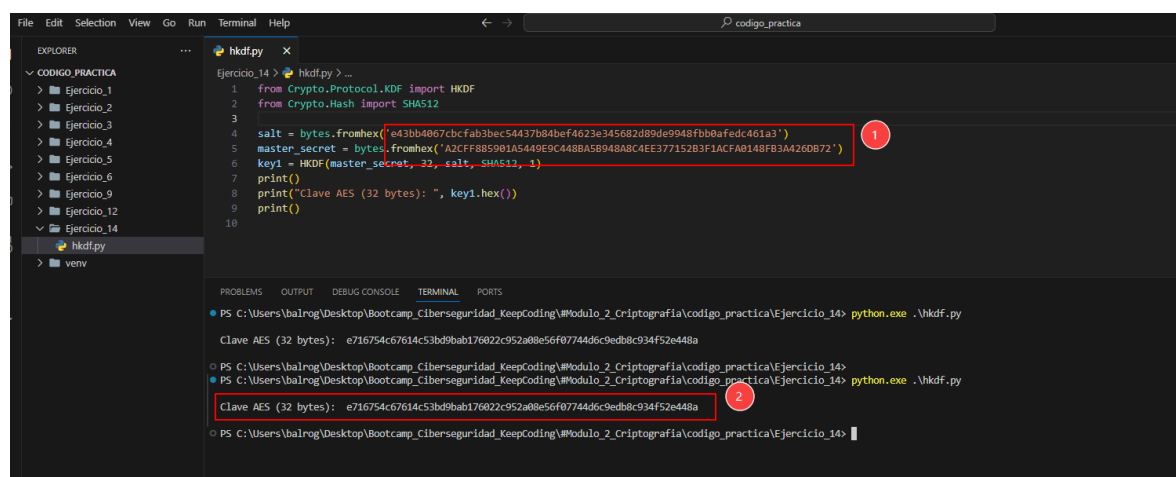
¿Qué clave se ha obtenido?

**Respuesta:** La clave que se obtuvo es:

**e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a**

## Procedimiento de la solución (Pregunta 1 - Ejercicio 14)

En la siguiente imagen se observa el uso de HKDF, para este ejercicio se usó tanto la llave maestra que está alojada en el KeyStore así como un Salt que en este caso es el identificador de dispositivo que nos han dado, una vez teniendo estos valores se procede a calcular la clave que se ha solicitado, ver punto 2.



```
File Edit Selection View Go Run Terminal Help
codigo_practica
EXPLORER
CODIGO_PRACTICA
  Ejercicio_1
  Ejercicio_2
  Ejercicio_3
  Ejercicio_4
  Ejercicio_5
  Ejercicio_6
  Ejercicio_9
  Ejercicio_12
  Ejercicio_14
  hkdf.py
  veriv
TERMINAL
Ejercicio_14 > hkdf.py ...
1 from Crypto.Protocol.KDF import HKDF
2 from Crypto.Hash import SHA512
3
4 salt = bytes.fromhex('e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3')
5 master_secret = bytes.fromhex('A2CFF885981A5449E9C448BA5B948A8C4EE377152B3F1ACFA8148FB3A426D872')
6 key1 = HKDF(master_secret, salt, SHA512, 1)
7 print()
8 print("Clave AES (32 bytes): ", key1.hex())
9 print()
10
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_14> python.exe .\hkdf.py
Clave AES (32 bytes): e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a
PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_14>
PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_14> python.exe .\hkdf.py
Clave AES (32 bytes): e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a
PS C:\Users\balrog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_14> |
```

Figura 1. Se obtiene una clave a partir de una llave maestra y un salt

## Ejercicio 15.

Nos envían un bloque TR31:

**D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37018E111B**

Donde la clave de transporte para desenvolver (unwrap) el bloque es: **A1A1010101010101010101010101010102**

¿Con qué algoritmo se ha protegido el bloque de clave?

**Respuesta:** Valor A y es un algoritmo AES

¿Para qué algoritmo se ha definido la clave?

**Respuesta:** Valor D, se ha definido para un algoritmo AES

¿Para qué modo de uso se ha generado?

**Respuesta:** Valor B y puede cifrar/descifrar además puede de envolver/desenvolver

¿Es exportable?

**Respuesta:** Valor S, es sensible, si es exportable

¿Para qué se puede usar la clave?

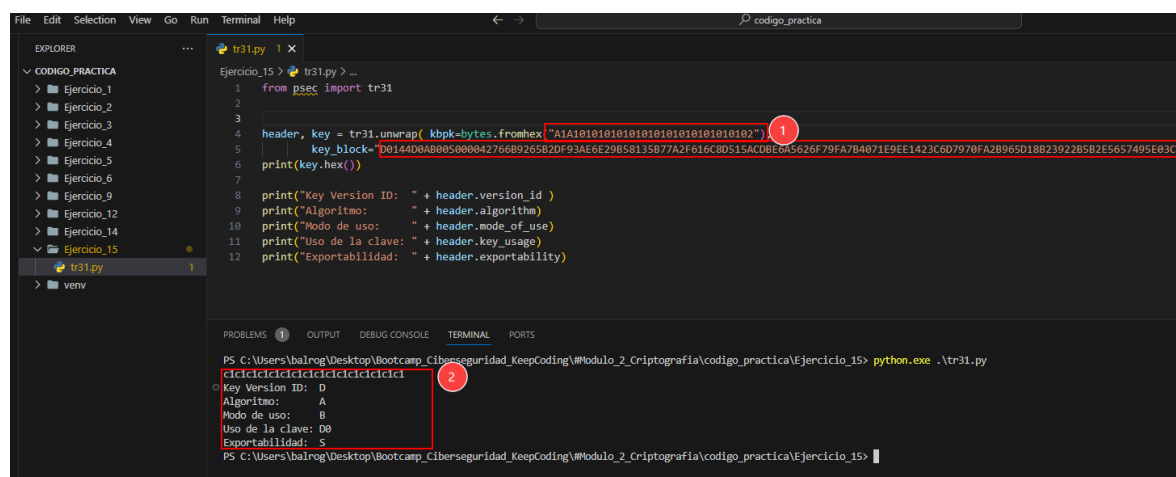
**Respuesta:** Se puede usar para cifrar y descifrar

¿Qué valor tiene la clave?

**Respuesta:** El valor de la clave es c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1

## Procedimiento de la solución (Pregunta 1 - Ejercicio 15)

En la siguiente figura se ha logrado desenvolver el bloque con los datos que nos han dado ver punto 1, en la parte del punto 2 obtenemos información muy diversa como el tipo de algoritmo, uso de la clave entre otros, una vez teniendo estos valores debemos buscar la documentación de GitHub para ver el detalle preciso de los valores que nos ha arrojado y poder determinar por ejemplo: **el Modo de uso** de esta misma.



```
File Edit Selection View Go Run Terminal Help
codigo_practica

EXPLORER
CODIGO_PRACTICA
  Ejercicio_1
  Ejercicio_2
  Ejercicio_3
  Ejercicio_4
  Ejercicio_5
  Ejercicio_6
  Ejercicio_9
  Ejercicio_12
  Ejercicio_14
  Ejercicio_15
  tr31.py
  venv

tr31.py
1 from pycryptodome import tr31
2
3
4 header, key = tr31.unwrap(kbpbk-bytes.fromhex("A1A10101010101010101010101010102"))
5 key_block="D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37018E111B"
6 print(key.hex())
7
8 print("Key Version ID: " + header.version_id)
9 print("Algoritmo: " + header.algorithm)
10 print("Modo de uso: " + header.mode_of_use)
11 print("Uso de la clave: " + header.key_usage)
12 print("Exportabilidad: " + header.exportability)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\bairog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_15> python.exe .\tr31.py
c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1
Key Version ID: D
Algoritmo: A
Modo de uso: B
Uso de la clave: D0
Exportabilidad: S
PS C:\Users\bairog\Desktop\Bootcamp_Ciberseguridad_KeepCoding\Modulo_2_Criptografia\codigo_practica\Ejercicio_15>
```

Figura 1. Se desenvuelve el bloque TR31 desde Python