Alumno: José Benito Ibarria Topete

PRÁCTICA: CRIPTOGRAFÍA

Objetivo:

Contestar un conjunto de preguntas / ejercicios relacionados con la materia aprendida en el curso demostrando la adquisición de conocimientos relacionados con la criptografía.

Detalles:

En esta práctica el alumno aplicará las técnicas y utilizará las diferentes herramientas vistas durante el módulo.

Cualquier password que sea necesaria tendrá un valor 123456.

Evaluación

Es obligatorio la entrega de un informe para considerar como APTA la práctica. Este informe ha de contener:

- Los enunciados seguido de las respuestas justificadas y evidenciadas.
- En el caso de que se hayan usado comandos / herramientas también se deben nombrar y explicar los pasos realizados.

El código escrito para la resolución de los problemas se entrega en archivos separados junto al informe.

Se va a valorar el proceso de razonamiento aunque no se llegue a resolver completamente los problemas. Si el código no funciona, pero se explica detalladamente la intención se valorará positivamente.

El objetivo principal de este módulo es adquirir conocimientos de criptografía y por ello se considera fundamental usar cualquier herramienta que pueda ayudar a su resolución, demostrando que no sólo se obtiene el dato sino que se tiene un conocimiento profundo del mismo. Si durante la misma no se indica claramente la necesidad de resolverlo usando programación, el alumno será libre de usar cualquier herramienta, siempre y cuando aporte las evidencias oportunas.

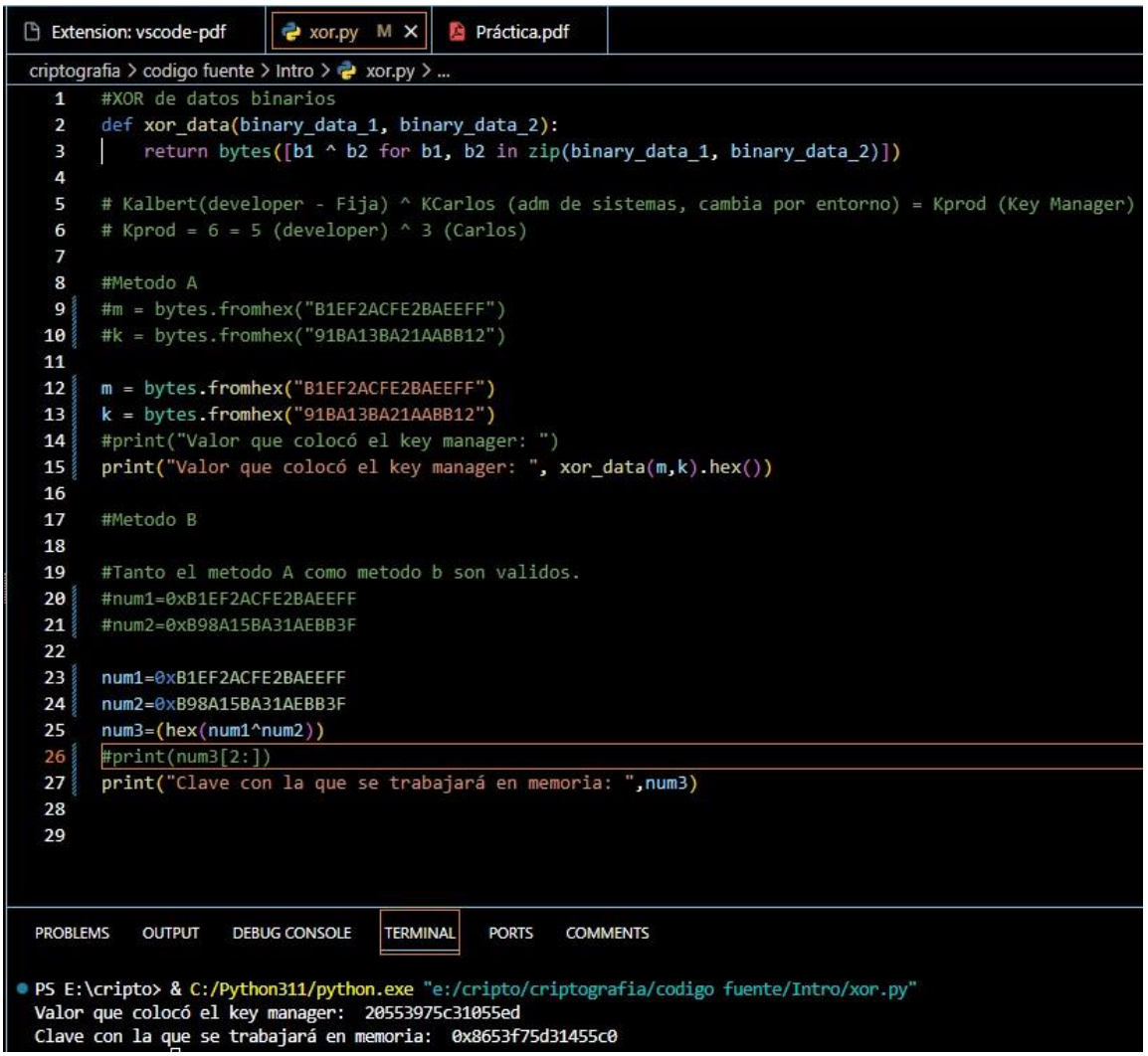
Ejercicios:

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?

La clave fija, recordemos es B1EF2ACFE2BAEEFF, mientras que en producción sabemos que la parte dinámica que se modifica en los ficheros de propiedades es B98A15BA31AEBB3F. ¿Qué clave será con la que se trabaje en memoria?

R.- En este caso nos apoyaremos con el programa xor.py para realizar ambas operaciones XOR al mismo tiempo:



Los resultados obtenidos mediante el programa xor.py son los siguientes:

a) El key manager ha puesto el siguiente valor en properties para forzar la clave final: ***20553975c31055ed***

b) La clave en producción, con la que se trabajará en memoria es: ***8653f75d31455c0***

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:

```
TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4US
t3aB/i50nvvJbBiG+le1ZhpR84oI=
```

Para este caso, se ha usado un AES/CBC/PKCS7. Si lo desciframos, ¿qué obtenemos?
R.- Obtenemos el siguiente mensaje: "Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo."

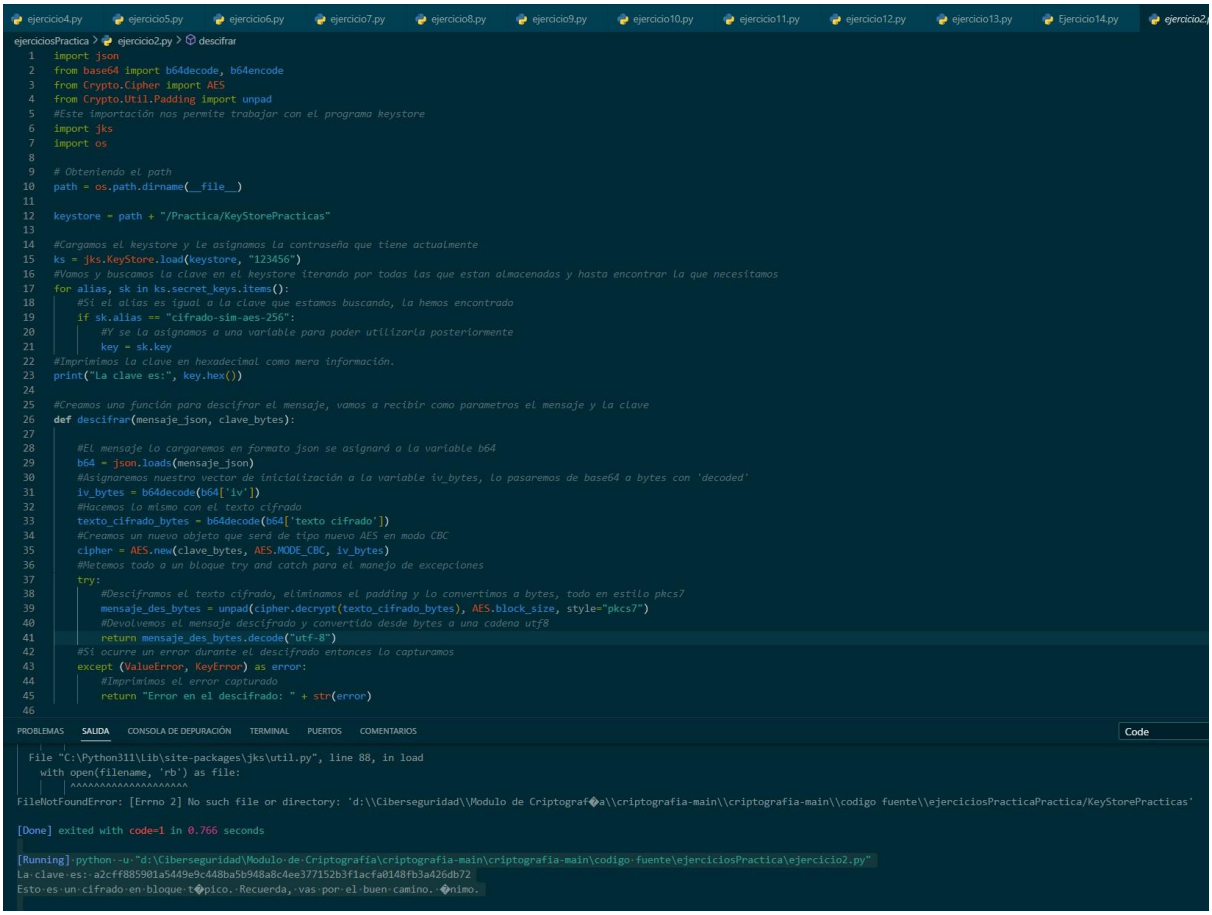
¿Qué ocurre si decidimos cambiar el padding a x923 en el descifrado?

R.- Aunque lo lógico sería que tuviéramos un error ya que el cifrado se ha realizado bajo el estilo "PKCS7", en realidad no tenemos ningún problema ya que, en esencia son de tipo iguales puesto que al final se añade el número del padding adicionado en el formato 01 en el tipo x923, dado nuestro ejemplo de práctica, pasaría lo mismo si usáramos el tipo de padding ISO 101 26. No debemos de olvidar que solo en el caso de nuestro ejemplo de practica funcionaría(cuando termina en 01), en el resto de los casos mandaría un error y se debería de colocar forzosamente el mismo estilo de padding en el cifrado y en el descifrado.

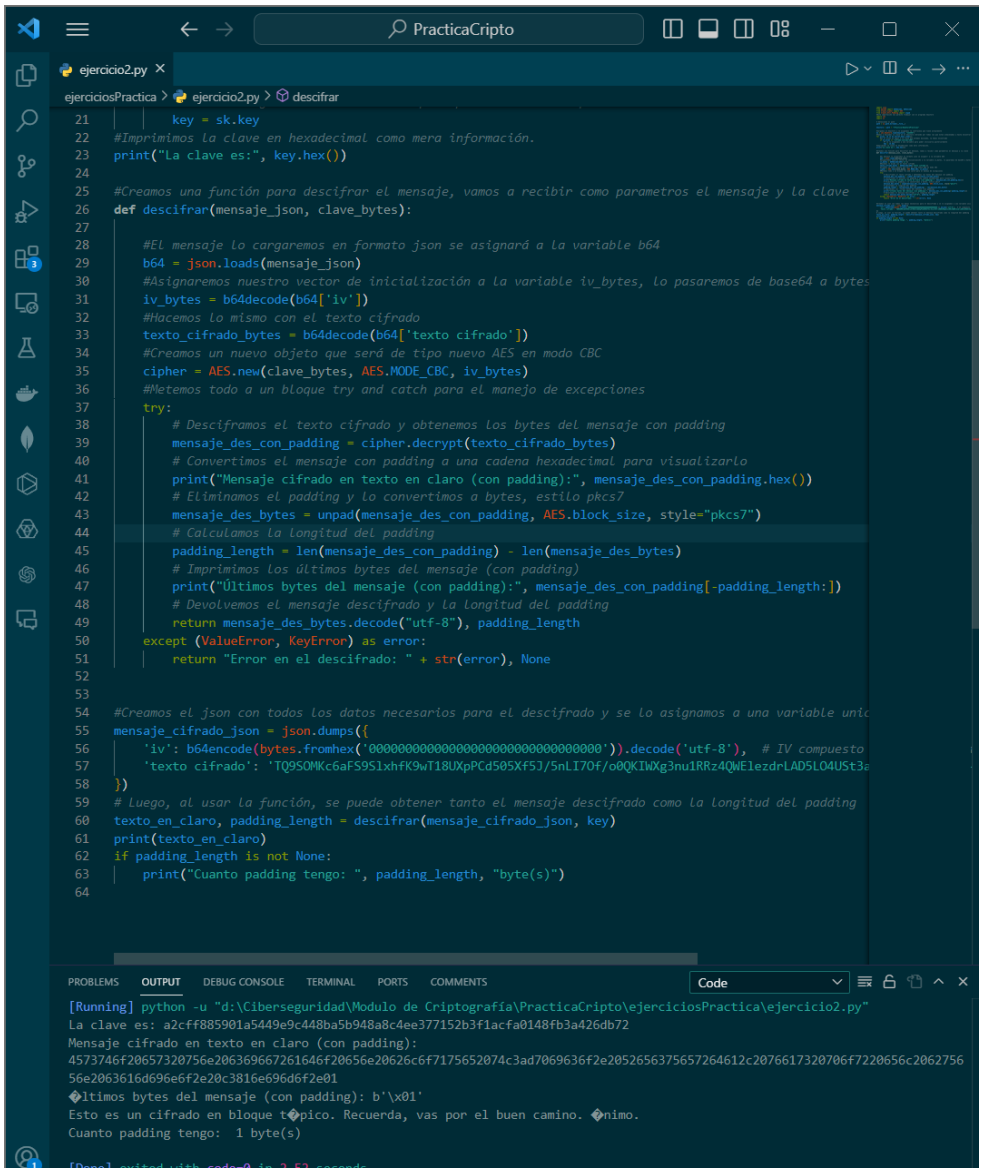
¿Cuánto padding se ha añadido en el cifrado?

R.- Se ha añadido un padding de 1 byte, esto porque hemos añadido al flujo de ejecución del programa la impresión del mensaje cifrado en texto en claro antes de añadirle el padding, mostrándome en pantalla "x01" que fundamentalmente en el estilo de padding pkcs significa 1 byte de padding, ademas hemos realizado una resta del mensaje con padding – los bytes que tiene el mensaje dando un total de 1 byte y sumado a lo anterior, hacemos en cybercheff la conversión de base64 del texto cifrado a hexadecimal y nos dá un total de 80 bytes, eso nos sirve porque entonces podemos determinar que el tamaño del bloque es 16 bytes (El algoritmo AES utiliza tamaños de bloque de 16 bytes, por ejemplo 3DES utiliza un tamaño de bloque de 8 bytes) y para confirmar, al seleccionar el bloque cifrado en hexadecimal solo se seleccionan los caracteres del texto cifrado desde la "T" y hasta la "l" sobrando el signo "=", siendo este el padding de 1 byte.

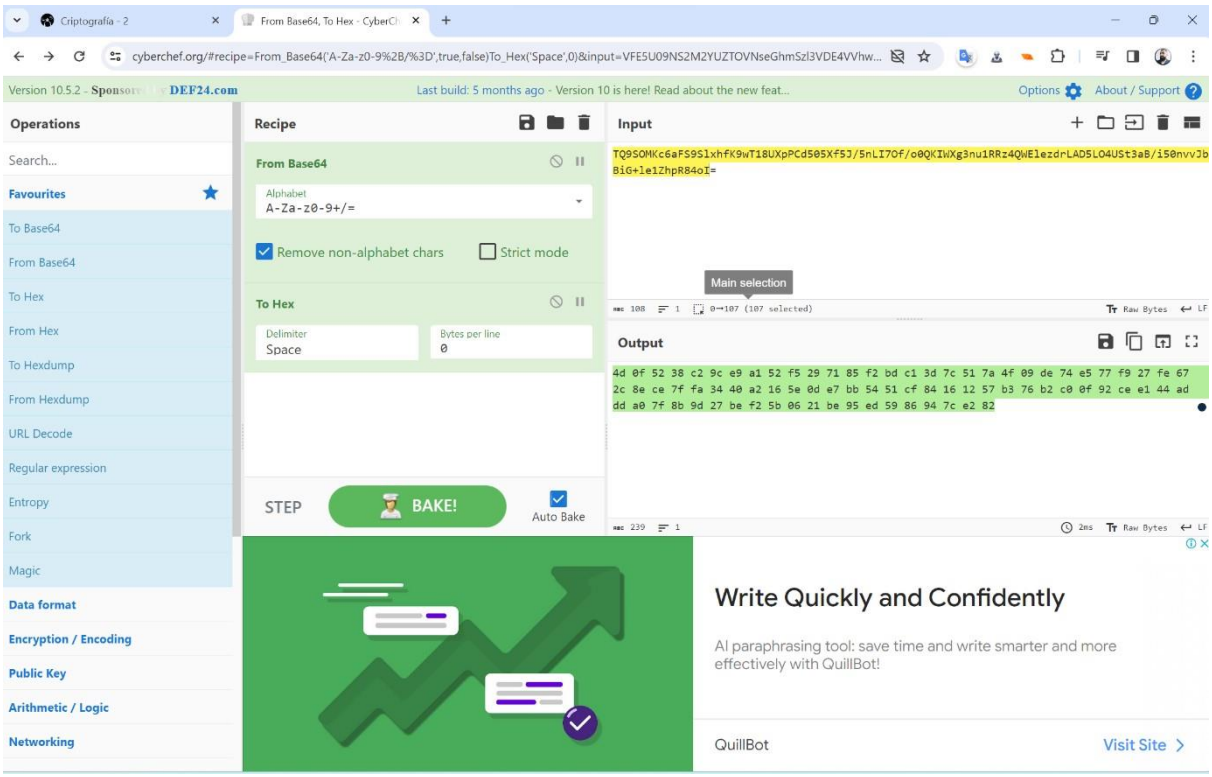
Se valorará positivamente, obtener el dato de la clave desde el keystore mediante codificación en Python (u otro lenguaje).



```
ejercicio4.py  ejercicio5.py  ejercicio6.py  ejercicio7.py  ejercicio8.py  ejercicio9.py  ejercicio10.py  ejercicio11.py  ejercicio12.py  ejercicio13.py  Ejercicio14.py  ejercicio2.
ejerciciosPractica > ejercicio2.py > descifrar
1   import json
2   from base64 import b64decode, b64encode
3   from Crypto.Cipher import AES
4   from Crypto.Util.Padding import unpad
5   #Este importación nos permite trabajar con el programa keystore
6   import jks
7   import os
8
9   # Obteniendo el path
10  path = os.path.dirname(__file__)
11
12  keystore = path + "/Practica/KeyStorePracticas"
13
14  #Cargamos el keystore y le asignamos la contraseña que tiene actualmente
15  ks = jks.KeyStore.load(keystore, "123456")
16  #Vamos y buscamos la clave en el keystore iterando por todas las que estan almacenadas y hasta encontrar la que necesitamos
17  for alias, sk in ks.secret_keys.items():
18      #Si el alias es igual a la clave que estamos buscando, la hemos encontrado
19      if sk.alias == "cifrado-sim-aes-256":
20          #Y se la asignamos a una variable para poder utilizarla posteriormente
21          key = sk.key
22  #Imprimimos la clave en hexadecimal como mera información.
23  print("La clave es:", key.hex())
24
25  #Creamos una función para descifrar el mensaje, vamos a recibir como parametros el mensaje y la clave
26  def descifrar(mensaje_json, clave_bytes):
27
28      #El mensaje lo cargaremos en formato json se asignará a la variable b64
29      b64 = json.loads(mensaje_json)
30      #Asignaremos nuestro vector de inicialización a la variable iv_bytes, lo pasaremos de base64 a bytes con 'decoded'
31      iv_bytes = b64decode(b64['iv'])
32      #Hacemos lo mismo con el texto cifrado
33      texto_cifrado_bytes = b64decode(b64['texto cifrado'])
34      #Creamos un nuevo objeto que será de tipo nuevo AES en modo CBC
35      cipher = AES.new(clave_bytes, AES.MODE_CBC, iv_bytes)
36      #Metemos todo a un bloque try and catch para el manejo de excepciones
37      try:
38          #Desciframos el texto cifrado, eliminamos el padding y lo convertimos a bytes, todo en estilo pkcs7
39          mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size, style="pkcs7")
40          #Devolvemos el mensaje descifrado y convertido desde bytes a una cadena utf8
41          return mensaje_des_bytes.decode("utf-8")
42      #Si ocurre un error durante el descifrado entonces lo capturamos
43      except (ValueError, KeyError) as error:
44          #Imprimimos el error capturado
45          return "Error en el descifrado: " + str(error)
46

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS  COMENTARIOS                                    Code

  File "C:\Python311\Lib\site-packages\jks\util.py", line 88, in load
    with open(filename, 'rb') as file:
         ^^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'd:\\Ciberseguridad\\Modulo de Criptograf\ufffda\\criptografia-main\\criptografia-main\\codigo fuente\\ejerciciosPracticaPractica/KeyStorePracticas'

[Done] exited with code=1 in 0.766 seconds

[Running] python -u "d:\Ciberseguridad\Modulo de Criptografía\criptografia-main\criptografia-main\codigo fuente\ejerciciosPractica\ejercicio2.py"
La clave es: a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72
Esto es un cifrado en bloque t\ufffdpico. Recuerda, vas por el buen camino. \ufffdnimo.
```



```
PracticaCripto
ejercicio2.py
ejerciciosPractica > ejercicio2.py > descifrar
21          key = sk.key
22  #Imprimimos la clave en hexadecimal como mera información.
23  print("La clave es:", key.hex())
24
25  #Creamos una función para descifrar el mensaje, vamos a recibir como parametros el mensaje y la clave
26  def descifrar(mensaje_json, clave_bytes):
27
28      #El mensaje lo cargaremos en formato json se asignará a la variable b64
29      b64 = json.loads(mensaje_json)
30      #Asignaremos nuestro vector de inicialización a la variable iv_bytes, lo pasaremos de base64 a bytes
31      iv_bytes = b64decode(b64['iv'])
32      #Hacemos lo mismo con el texto cifrado
33      texto_cifrado_bytes = b64decode(b64['texto cifrado'])
34      #Creamos un nuevo objeto que será de tipo nuevo AES en modo CBC
35      cipher = AES.new(clave_bytes, AES.MODE_CBC, iv_bytes)
36      #Metemos todo a un bloque try and catch para el manejo de excepciones
37      try:
38          # Desciframos el texto cifrado y obtenemos los bytes del mensaje con padding
39          mensaje_des_con_padding = cipher.decrypt(texto_cifrado_bytes)
40          # Convertimos el mensaje con padding a una cadena hexadecimal para visualizarlo
41          print("Mensaje cifrado en texto en claro (con padding):", mensaje_des_con_padding.hex())
42          # Eliminamos el padding y lo convertimos a bytes, estilo pkcs7
43          mensaje_des_bytes = unpad(mensaje_des_con_padding, AES.block_size, style="pkcs7")
44          # Calculamos la longitud del padding
45          padding_length = len(mensaje_des_con_padding) - len(mensaje_des_bytes)
46          # Imprimimos los últimos bytes del mensaje (con padding)
47          print("Últimos bytes del mensaje (con padding):", mensaje_des_con_padding[-padding_length:])
48          # Devolvemos el mensaje descifrado y la longitud del padding
49          return mensaje_des_bytes.decode("utf-8"), padding_length
50      except (ValueError, KeyError) as error:
51          return "Error en el descifrado: " + str(error), None
52
53
54  #Creamos el json con todos los datos necesarios para el descifrado y se lo asignamos a una variable unic
55  mensaje_cifrado_json = json.dumps({
56      'iv': b64encode(bytes.fromhex('00000000000000000000000000000000')).decode('utf-8'),  # IV compuesto
57      'texto cifrado': 'TQ9SOMKc6aFS9S1xhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3a
58  })
59  # Luego, al usar la función, se puede obtener tanto el mensaje descifrado como la longitud del padding
60  texto_en_claro, padding_length = descifrar(mensaje_cifrado_json, key)
61  print(texto_en_claro)
62  if padding_length is not None:
63      print("Cuanto padding tengo: ", padding_length, "byte(s)")
64

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS                                    Code
[Running] python -u "d:\Ciberseguridad\Modulo de Criptografía\PracticaCripto\ejerciciosPractica\ejercicio2.py"
La clave es: a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72
Mensaje cifrado en texto en claro (con padding):
4573746f20657320756e20636966726164f20656e20626c6f7175652074c3ad7069636f2e2052656375657264612c2076617320706f7220656c2062756
56e2063616d696e6f2e20c3816e696d6f2e01
\ufffdltimos bytes del mensaje (con padding): b'\x01'
Esto es un cifrado en bloque t\ufffdpico. Recuerda, vas por el buen camino. \ufffdnimo.
Cuanto padding tengo:  1 byte(s)
```



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.

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

R. – Considero que podríamos mejorar de forma sencilla el sistema haciéndolo más robusto extrayendo la clave directamente desde el Keystore y que no la coloquemos de forma manual en el código fuente ya que esta podría ser vista de forma directa por un actor malicioso, por lo que considero que esta clave debería de estar oculta bajo otra aplicación y esa aplicación con una contraseña que también sea robusta, así mismo, podríamos añadir al bloque de código un nuevo bloque que realice la conversión del “nounce” desde base64 a hexadecimal y pasar ese valor de forma directa al cifrado en la variable “cipher”, sumado a todo lo anterior podríamos también calcular un HMAC para el mensaje cifrado y así poder proveer de una mayor integridad puesto que si alguien nos cambie cualquier cosa en el mensaje se calcule un HMAC distinto confirmando así una posible corrupción o ataque a la integridad de nuestro mensaje.

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

ChaCha20-descifrado-reto.py | ChaCha20-Cifrado.py M | Conversiones.py M

criptografia > codigo fuente > criptografia en flujo > ChaCha20-Cifrado.py > ...

```python
from Crypto.Cipher import ChaCha20
from base64 import b64decode, b64encode

textoPlano_bytes = bytes('KeepCoding te enseña a codificar y a cifrar', 'UTF-8')
#Se requiere o 256 o 128 bits de clave, por ello usamos 256 bits que se transforman en 64 caracteres hexadecimales
clave = bytes.fromhex('AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120')
#Importante NUNCA debe fijarse el nonce, en este caso lo hacemos para mostrar el mismo resultado en cualquier lenguaje.
nonce_mensaje = bytes.fromhex('f5871c9ff7f99c926102dd92')
print('nonce  = ', nonce_mensaje.hex())

#Con la clave y con el nonce se cifra. El nonce debe ser único por mensaje
cipher = ChaCha20.new(key=clave, nonce=nonce_mensaje)
texto_cifrado_bytes = cipher.encrypt(textoPlano_bytes)
print('Mensaje cifrado en HEX = ', texto_cifrado_bytes.hex() )
print('Mensaje cifrado en B64 = ', b64encode(texto_cifrado_bytes).decode())

```

Archivo Editar Selección Ver Ir Ejecutar ... codigo fuente

ejercicio3.py | ejercicio2.py | ejercicio4.py | ejercicio5.py | ejercicio6.py | ejercicio7.py | ejercicio8.py | ejercicio9.py | ejercicio10.py

ejerciciosPractica > ejercicio3.py > ...

```python
from Crypto.Cipher import ChaCha20
from base64 import b64decode, b64encode
import base64
#Este importación nos permite trabajar con el programa keystore
import jks
import os
textoPlano_bytes = bytes('KeepCoding te enseña a codificar y a cifrar', 'UTF-8')

#Importamos la clave desde el keystore para garantizar la confidencialidad y la integridad

# Obteniendo el path
path = os.path.dirname(__file__)

keystore = path + "/Practica/KeyStorePracticas"

#Cargamos el keystore y le asignamos la contraseña que tiene actualmente
ks = jks.KeyStore.load(keystore, "123456")
#Vamos y buscamos la clave en el keystore iterando por todas las que estan almacenadas y hasta encontrar la que necesitamos
for alias, sk in ks.secret_keys.items():
    #Si el alias es igual a la clave que estamos buscando, la hemos encontrado
    if sk.alias == "cifrado-sim-chacha20-256":
        #Y se la asignamos a una variable para poder utilizarla posteriormente
        key = sk.key

#Se requiere o 256 o 128 bits de clave, por ello usamos 256 bits que se transforman en 64 caracteres hexadecimales
clave = key
#convertimos directamente el nonce desde base64 a hexadecimal
b64_string= '9Yccn/f5nJJhAt2S'
decoded_bytes = base64.b64decode(b64_string)
hex_string = decoded_bytes.hex()
#Importante NUNCA debe fijarse el nonce, en este caso lo hacemos para mostrar el mismo resultado en cualquier lenguaje.
nonce_mensaje = hex_string

#Con la clave y con el nonce se cifra. El nonce debe ser único por mensaje
cipher = ChaCha20.new(key=clave, nonce=bytes.fromhex(nonce_mensaje))
texto_cifrado_bytes = cipher.encrypt(textoPlano_bytes)
print('Mensaje cifrado en HEX = ', texto_cifrado_bytes.hex() )
print('Mensaje cifrado en B64 = ', b64encode(texto_cifrado_bytes).decode())
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS COMENTARIOS

```
[Running] python -u "d:\Ciberseguridad\Modulo de Criptografía\criptografia-main\criptografia-main\codigo fuente\ejerciciosPractica\ejercicio3.py"
Mensaje cifrado en HEX =  69ac4ee7c4c552537a00a19bcaf7f0aaed7c9c8f769956a09bce6fadef6c3535f2211c9467067cf5c4a842ab
Mensaje cifrado en B64 =  aaxO58TFU1N6AKGbyvfwqu18nI92mVagm85vre9sNTXyIRyUZwZ89cSoQqs=

[Done] exited with code=0 in 4.269 seconds
```

Mejora con el cálculo del HMAC

Un punto al final del mensaje que cambié y tenemos un HMAC distinto, vale, con esta mejora podemos garantizar la integridad al mismo tiempo.

4. Tenemos el siguiente jwt, cuya clave es “Con KeepCoding aprendemos”.

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSB

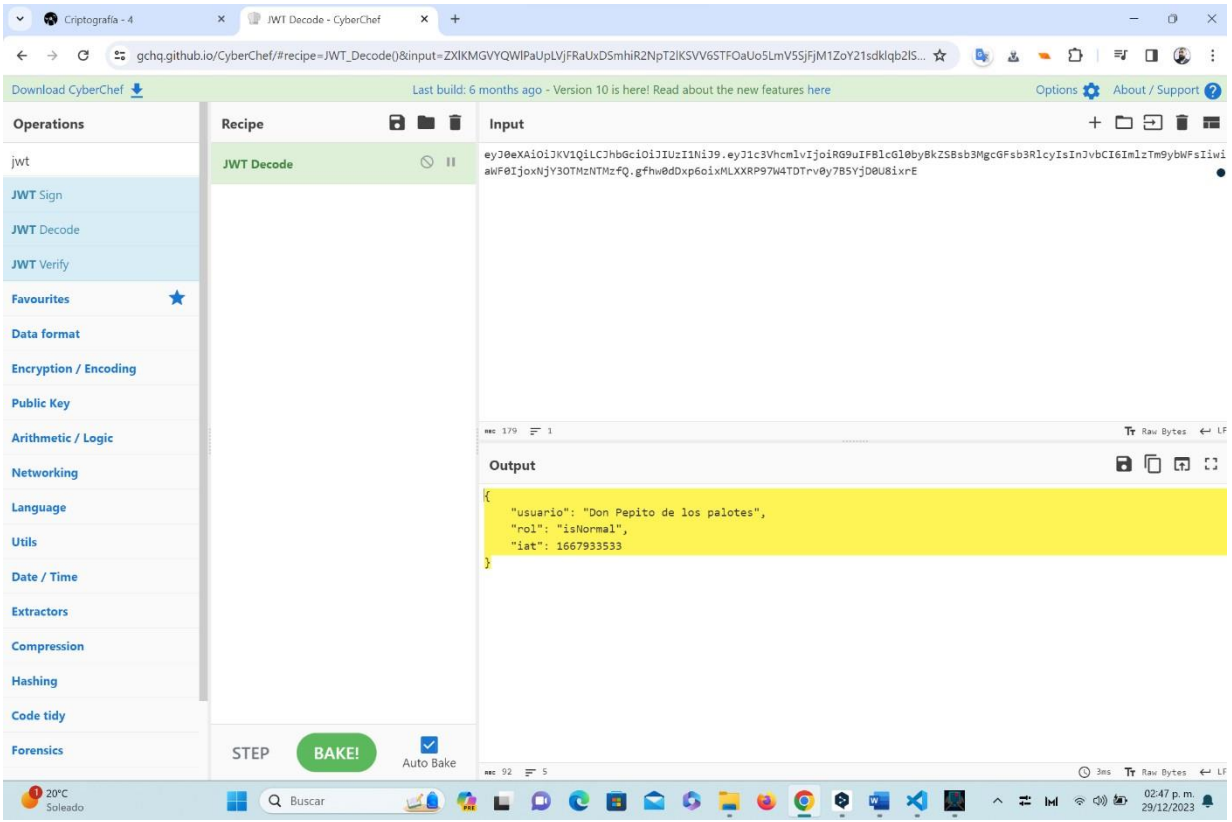
sb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ.gfhw0

dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

¿Qué algoritmo de firma hemos realizado?
{"typ":"JWT","alg":"HS256"}
R.- De acuerdo a lo extraído desde ciberchef del header desde un “Frombase64” obtenemos que el algoritmo de firma usado es un HMAC + SHA256

¿Cuál es el body del jwt?
R.- El body del JWT es el siguiente:

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



Un hacker está enviando a nuestro sistema el siguiente jwt:

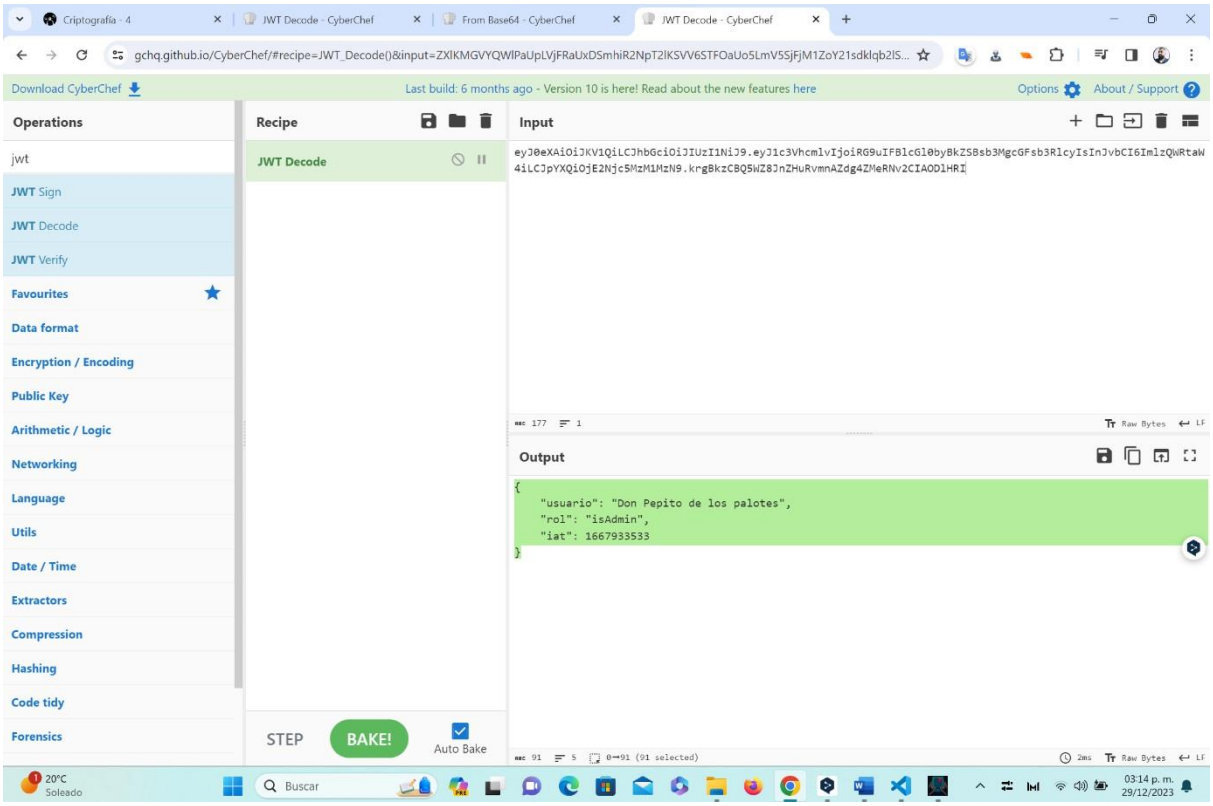
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBk

ZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9
.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODlHRI

¿Qué está intentando realizar?

R.- Está intentando básicamente el hacker autentificarse como administrador para poder quizás ejecutar un ataque de escalada de privilegios porque de acuerdo a la captura de pantalla obtenida desde ciberchef el payload es el siguiente:

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



¿Qué ocurre si intentamos validarlo con pyjwt?

R.- Va a enviar un error porque con las versiones actuales de hacen una validación con la clave que se ha dado, si el hacker no tiene la clave Con KeepCoding aprendemos” Entonces no se va a poder autenticar

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?
R.- Hemos generado un tipo de SHA3_256 bits porque la cadena de hash lograda (bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe) tiene 64 caracteres que divididos entre 2 son 32 y estos 32 multiplicados por 8 nos da 256 bits

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

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f

6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

¿Qué hash hemos realizado?
R.- Hemos realizado un SHA-512 ya que el hash tiene 128 caracteres que divididos entre 2 da 64 y multiplicado por 8 encontramos un valor de 512 bits de salida.

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?

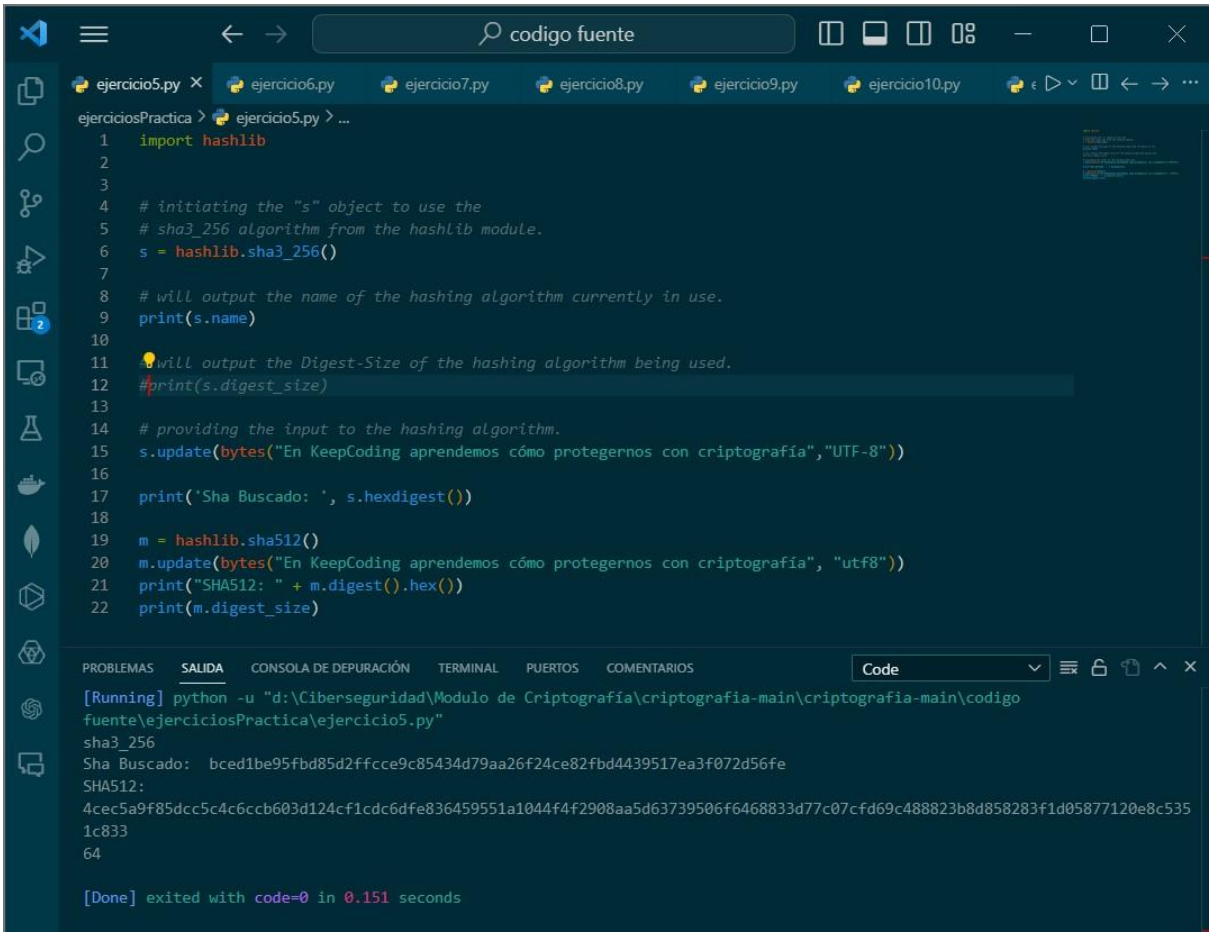
R.- Destacaría el efecto difusión y el efecto avalancha atendiendo los resultados anteriores ya que el texto en este ejercicio lleva un carácter más en forma de un punto al final del texto utf8, esto lo hace que se comporte de manera muy distinta a los resultados obtenidos anteriormente, al menos con el SHA3 Keccak de 256 bits que hicimos al principio del ejercicio.

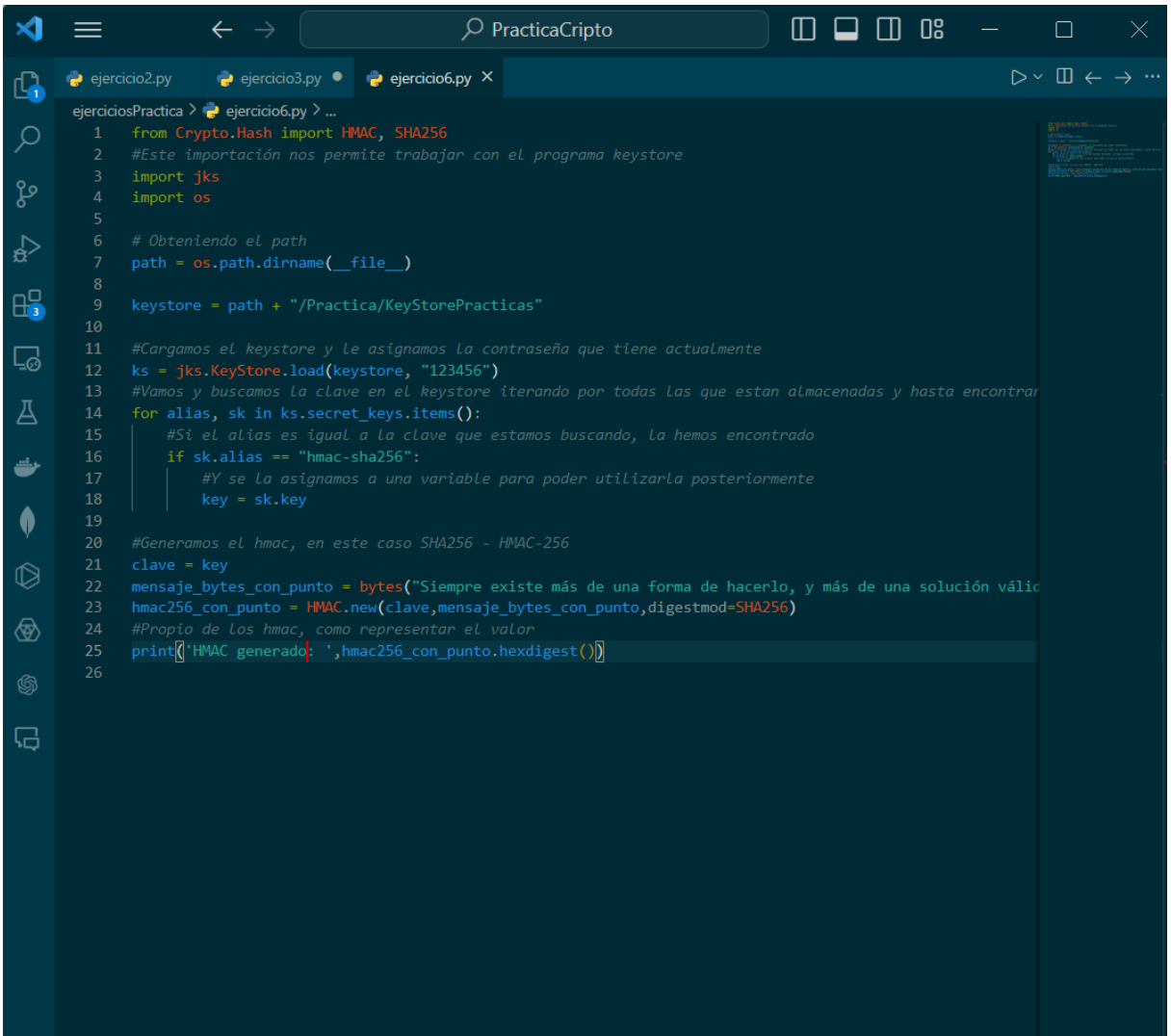
6. Calcula el hmac-256 (usando la clave contenida en el Keystore) del siguiente texto:

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

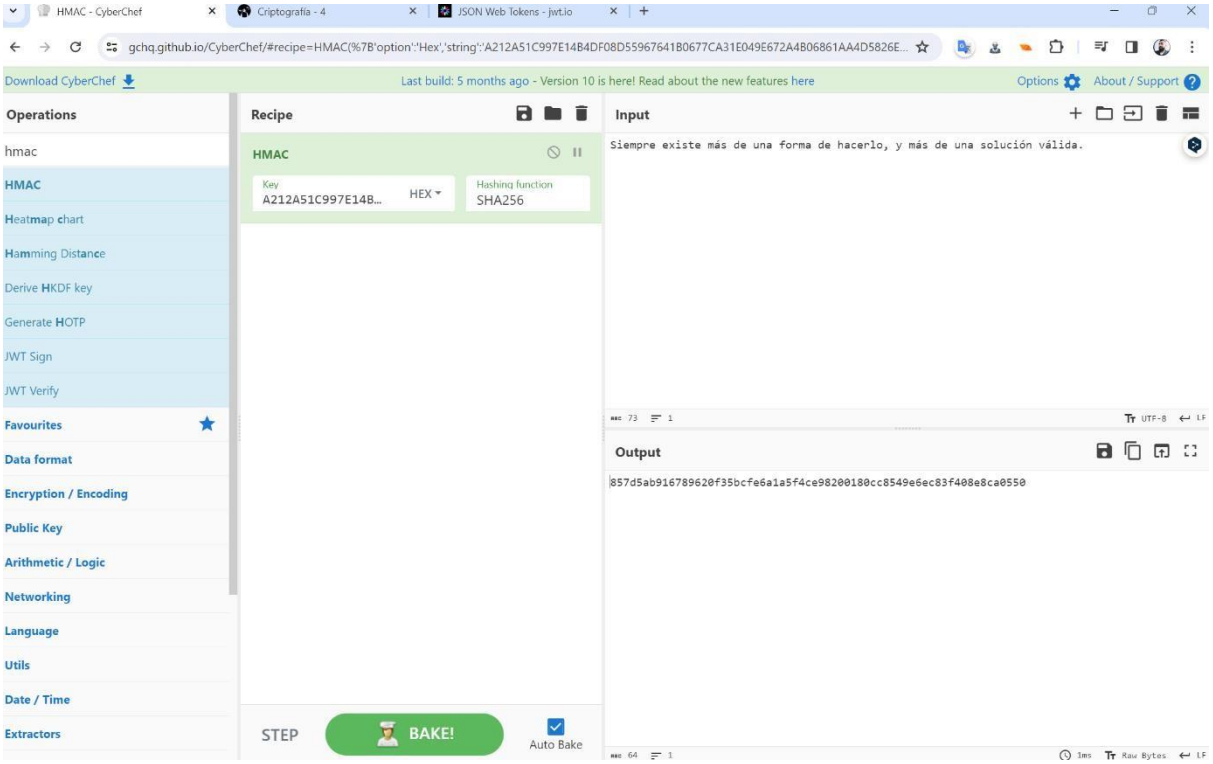
Se debe evidenciar la respuesta. Cuidado si se usan herramientas fuera de los lenguajes de programación, por las codificaciones es mejor trabajar en hexadecimal.

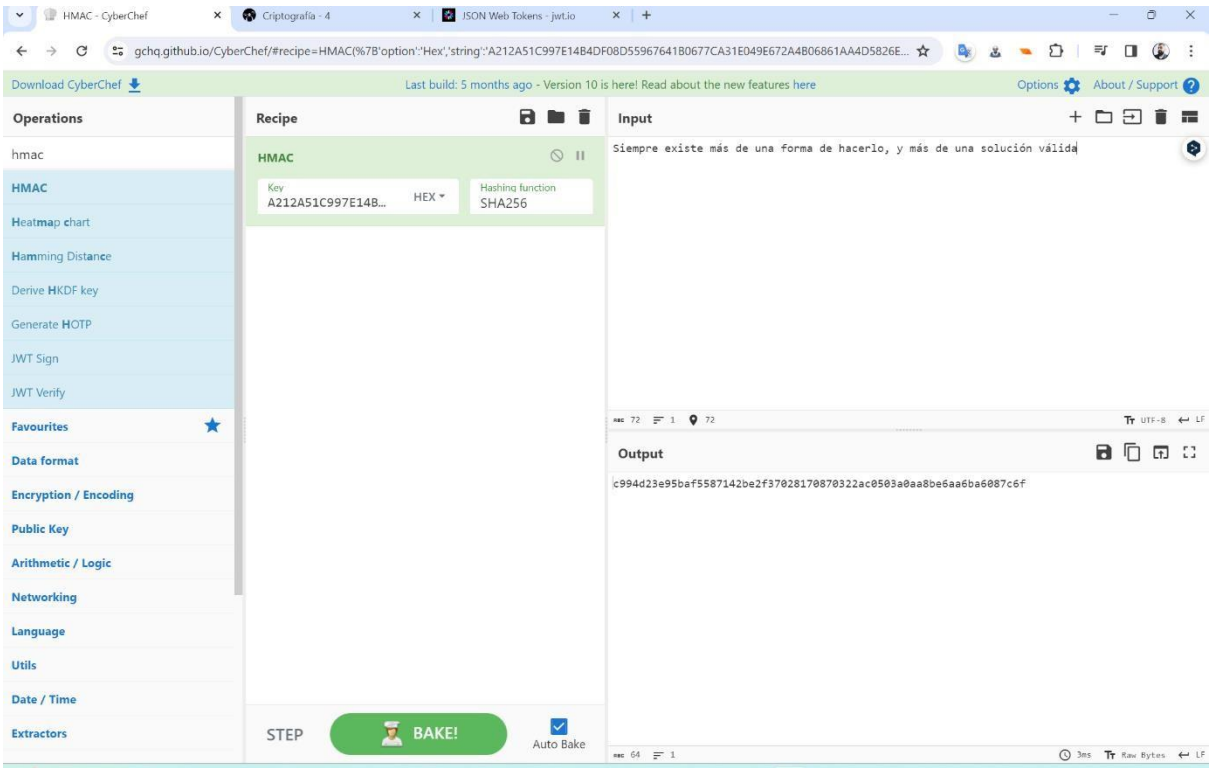
R.- En esencia tenemos el siguiente HMAC con el texto + punto final: HMAC :857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550


codigo fuente
ejercicio5.py
ejercicio6.py
ejercicio7.py
ejercicio8.py
ejercicio9.py
ejercicio10.py
ejerciciosPractica > ejercicio5.py > ...
1 import hashlib
2
3
4 # initiating the "s" object to use the
5 # sha3_256 algorithm from the hashlib module.
6 s = hashlib.sha3_256()
7
8 # will output the name of the hashing algorithm currently in use.
9 print(s.name)
10
11 #will output the Digest-Size of the hashing algorithm being used.
12 #print(s.digest_size)
13
14 # providing the input to the hashing algorithm.
15 s.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía","UTF-8"))
16
17 print('Sha Buscado: ', s.hexdigest())
18
19 m = hashlib.sha512()
20 m.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía", "utf8"))
21 print("SHA512: " + m.digest().hex())
22 print(m.digest_size)
PROBLEMAS
SALIDA
CONSOLA DE DEPURACIÓN
TERMINAL
PUERTOS
COMENTARIOS
Code
[Running] python -u "d:\Ciberseguridad\Modulo de Criptografía\criptografia-main\criptografia-main\codigo fuente\ejerciciosPractica\ejercicio5.py"
sha3_256
Sha Buscado: bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe
SHA512:
4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833
64
[Done] exited with code=0 in 0.151 seconds


PracticaCripto
ejercicio2.py
ejercicio3.py
ejercicio6.py
ejerciciosPractica > ejercicio6.py > ...
1 from Crypto.Hash import HMAC, SHA256
2 #Este importación nos permite trabajar con el programa keystore
3 import jks
4 import os
5
6 # Obteniendo el path
7 path = os.path.dirname(__file__)
8
9 keystore = path + "/Practica/KeyStorePracticas"
10
11 #Cargamos el keystore y le asignamos la contraseña que tiene actualmente
12 ks = jks.KeyStore.load(keystore, "123456")
13 #Vamos y buscamos la clave en el keystore iterando por todas las que estan almacenadas y hasta encontrar
14 for alias, sk in ks.secret_keys.items():
15 #Si el alias es igual a la clave que estamos buscando, la hemos encontrado
16 if sk.alias == "hmac-sha256":
17 #Y se la asignamos a una variable para poder utilizarla posteriormente
18 key = sk.key
19
20 #Generamos el hmac, en este caso SHA256 - HMAC-256
21 clave = key
22 mensaje_bytes_con_punto = bytes("Siempre existe más de una forma de hacerlo, y más de una solución válic
23 hmac256_con_punto = HMAC.new(clave,mensaje_bytes_con_punto,digestmod=SHA256)
24 #Propio de los hmac, como representar el valor
25 print('HMAC generado: ',hmac256_con_punto.hexdigest())
26

Realizado con ciberchef:





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?

R.- Es una mala opción porque el algoritmo SHA-1 está deprecado a nivel de criptografía debido a que, hoy en día con el poder computacional actual podemos realizar el cálculo desde el valor del hash en hexadecimal a utf8, es decir, podemos ver el valor de forma reversible.

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?

R.- Definitivamente se me ocurriría proponer agregar un valor aleatorio a los passwords de cada uno de los usuarios este valor aleatorio sería lo equivalente a agregar un valor de SALT + PEPPER a nuestro hash SHA-256.

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

R.- Como propuesta de mejora yo propondría añadir otro número aleatorio, único para la base de datos que contiene a todos los valores de los passwords. Dicho número sería, igual, un numero aleatorio pero en forma de “Pepper” que guardaríamos lejos de los hashes y passwords del usuario, y además, para mejorar la resistencia utilizaría Argon2id como Hash.

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.

¿Qué algoritmos usarías?

R.- Considero como una excelente opción para la protección de contraseñas y los nombres de usuarios en la base de datos al algoritmo de hashing ARGON, fundamentalmente bajo el contexto de almacenar y verificar contraseñas de manera segura. Para el caso del cifrado de los campos como los números de las tarjetas propongo utilizar un algoritmo de cifrado simétrico como AES con GCM.

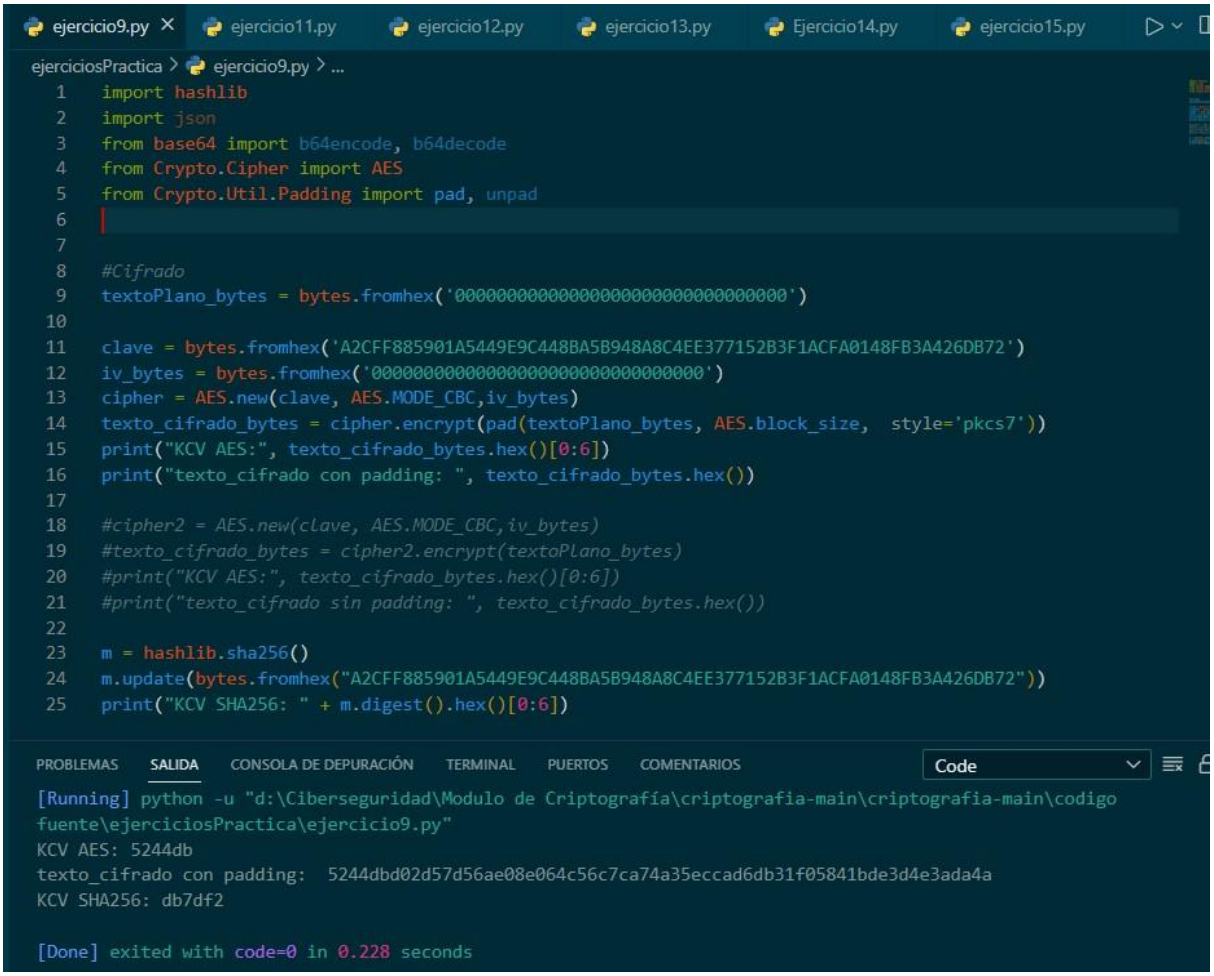
9. Se requiere calcular el KCV de las siguiente clave AES:

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

R.- El KCV(SHA256) es db7df2

R.- El KCV(AES) es 5244db



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.

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.

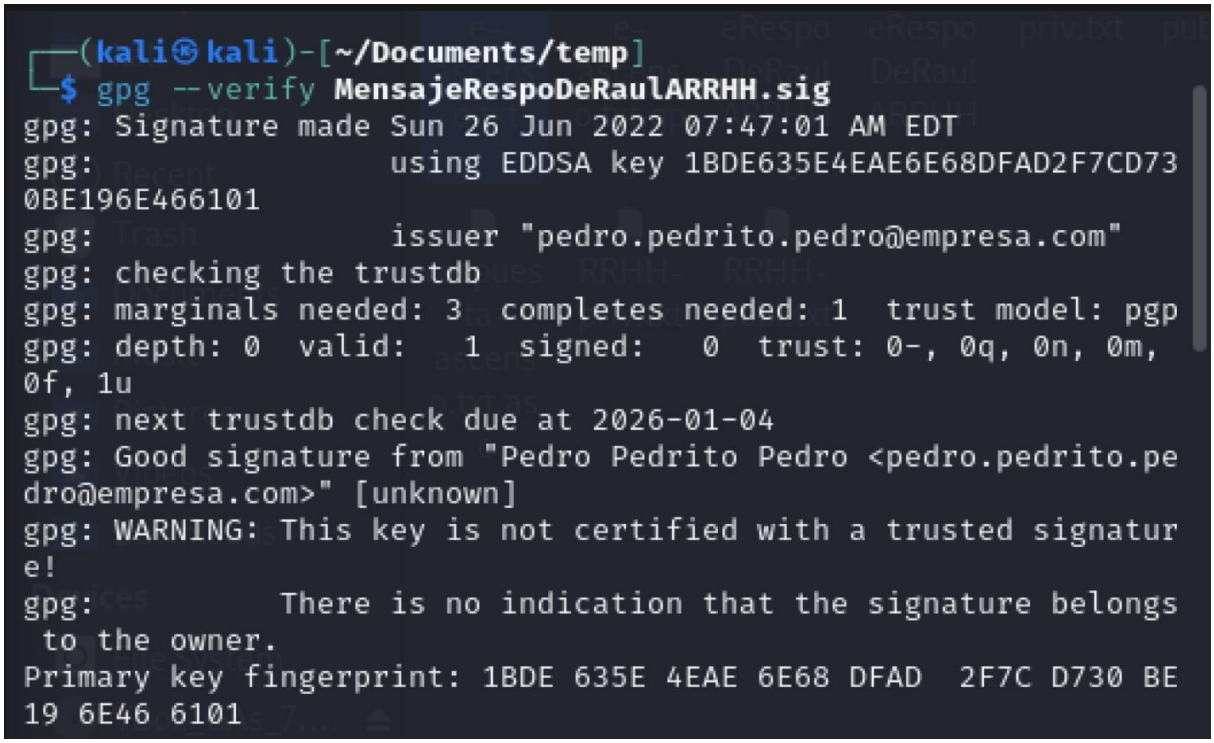
Por último, cifra el siguiente mensaje tanto con la clave pública de RRHH como la de Pedro y adjunta el fichero con la práctica.

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

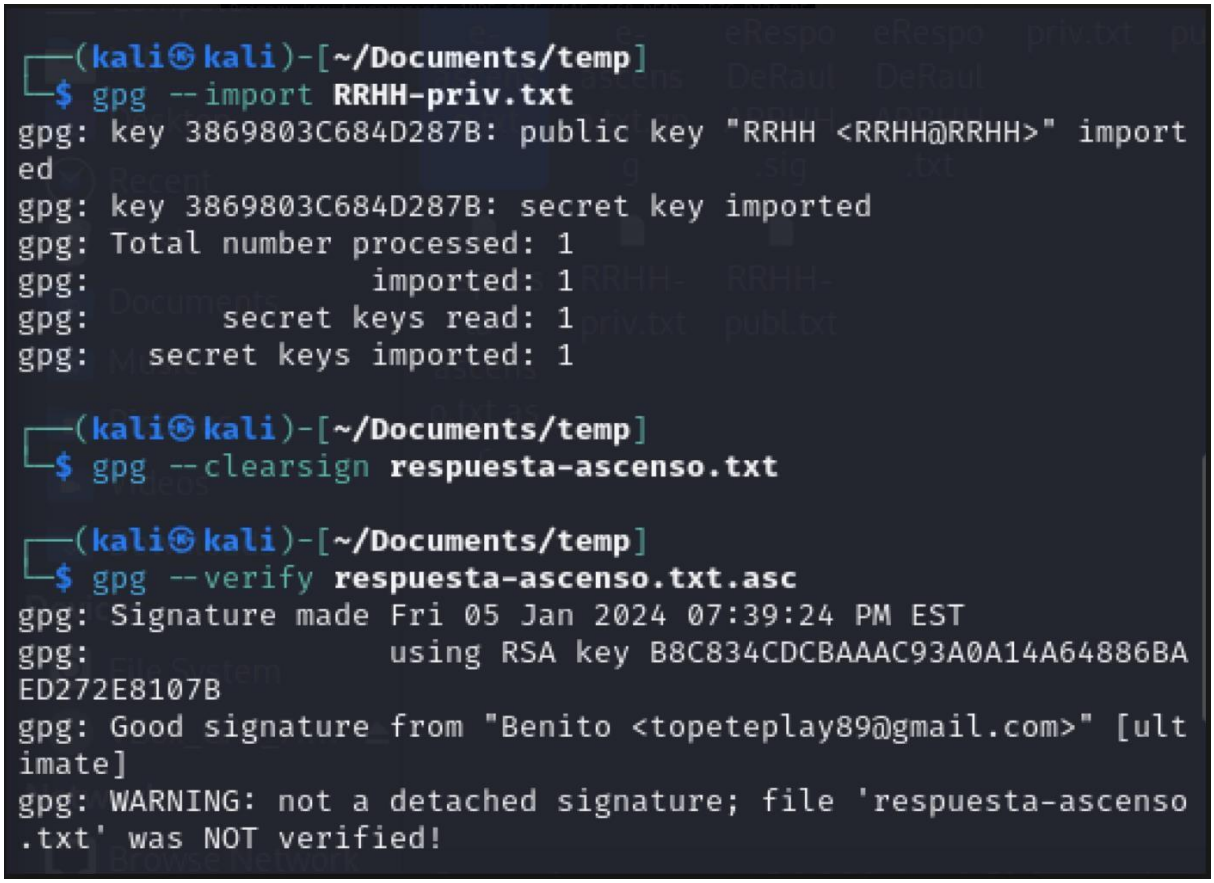
R.- Fundamentalmente hemos realizado el cifrado y la firma de los mensajes mediante gpg desde Kali Linux, la evidencia es la siguiente:
Importamos la clave pública de Pedro

```
┌──(kali㉿kali)-[~/Documents/temp]
└─$ gpg --import Pedro-publ.txt
gpg: key D730BE196E466101: public key "Pedro Pedrito Pedro <ped
ro.pedrito.pedro@empresa.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

Verificamos que el mensaje haya sido efectivamente firmado por pedro



Importamos la clave privada de recursos humanos



Firmamos el mensaje de RRHH, verificamos la firma y confirmamos que efectivamente el mensaje viene desde recurso humanos

```
┌──(kali㉿kali)-[~/Documents/temp]
└─$ gpg --encrypt --recipient 'RRHH' mensaje-ascenso.txt
gpg: 7C1A46EA20B0546F: There is no assurance this key belongs t
o the named user

sub  cv25519/7C1A46EA20B0546F 2022-06-26 RRHH <RRHH@RRHH>
 Primary key fingerprint: F2B1 D0E8 958D F2D3 BDB6  A105 3869 8
03C 684D 287B
      Subkey fingerprint: 811D 89A3 6199 A7C9 0BFE  69D6 7C1A 4
6EA 20B0 546F

It is NOT certain that the key belongs to the person named
in the user ID.  If you *really* know what you are doing,
you may answer the next question with yes.

Use this key anyway? (y/N) y
```

Ciframos el mensaje con la clave publica de Pedro

```
┌──(kali㉿kali)-[~/Documents/temp]
└─$ gpg --encrypt --recipient 'Pedro' mensaje-ascenso.txt
gpg: 25D6D0294035B650: There is no assurance this key belongs t
o the named user

sub  cv25519/25D6D0294035B650 2022-06-26 Pedro Pedrito Pedro <p
edro.pedrito.pedro@empresa.com>
 Primary key fingerprint: 1BDE 635E 4EAE 6E68 DFAD  2F7C D730 B
E19 6E46 6101
      Subkey fingerprint: 8E8C 6669 AC44 3271 42BC  C244 25D6 D
029 4035 B650

It is NOT certain that the key belongs to the person named
in the user ID.  If you *really* know what you are doing,
you may answer the next question with yes.

Use this key anyway? (y/N) y
File 'mensaje-ascenso.txt.gpg' exists. Overwrite? (y/N) y
```

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.

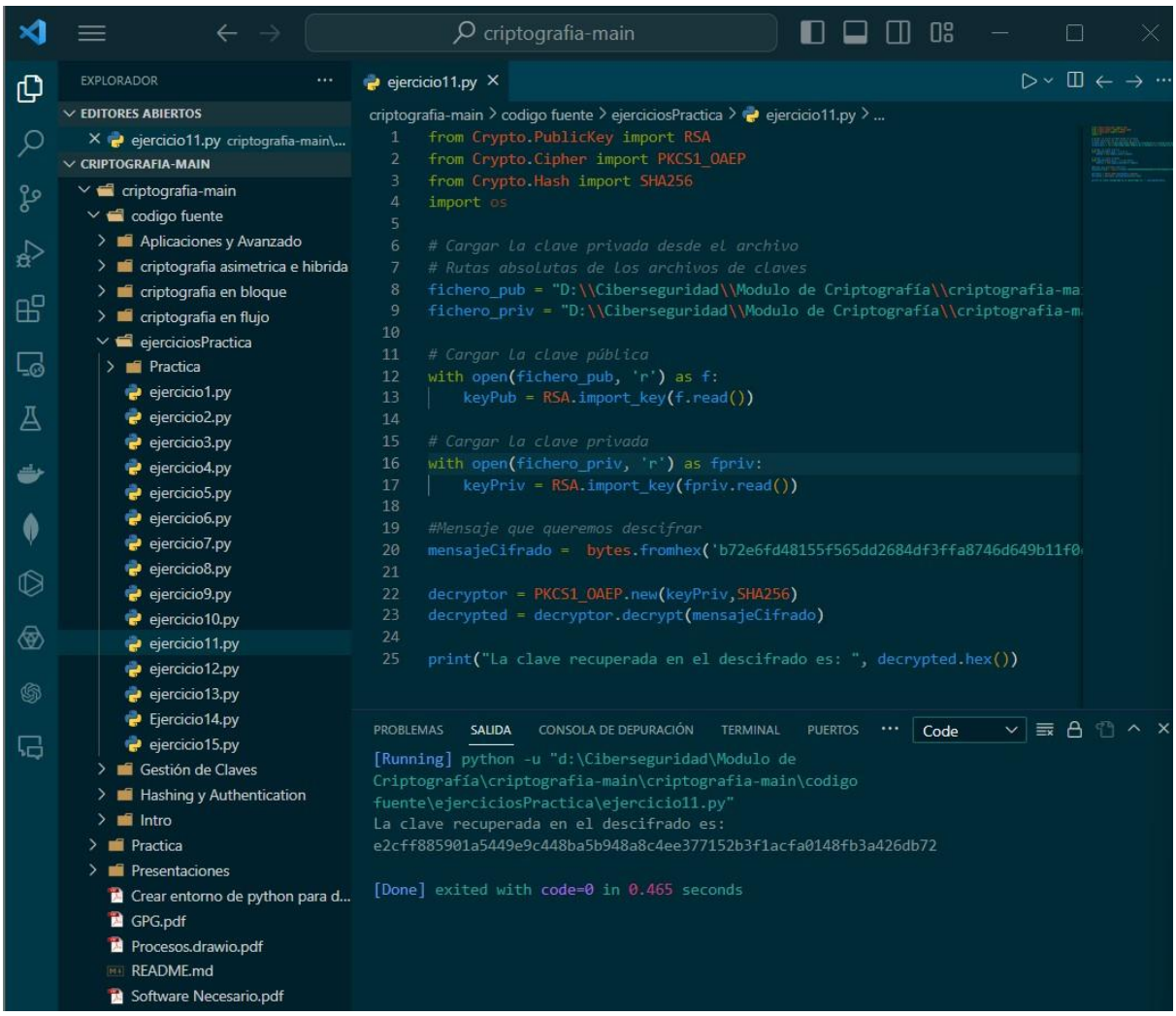
El texto cifrado es el siguiente:

b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c
96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d04e3d3d4ad629
793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae2c4f5ba670a4126f2f14
a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8624071be78ccef573d
896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f67d3f98b4cd907f27639f4b1
df3c50e05d5bf63768088226e2a9177485c54f72407fdf358fe64479677d8296ad38c6f
177ea7cb74927651cf24b01dee27895d4f05fb5c161957845cd1b5848ed64ed3b0372

2b21a526a6e447cb8ee

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

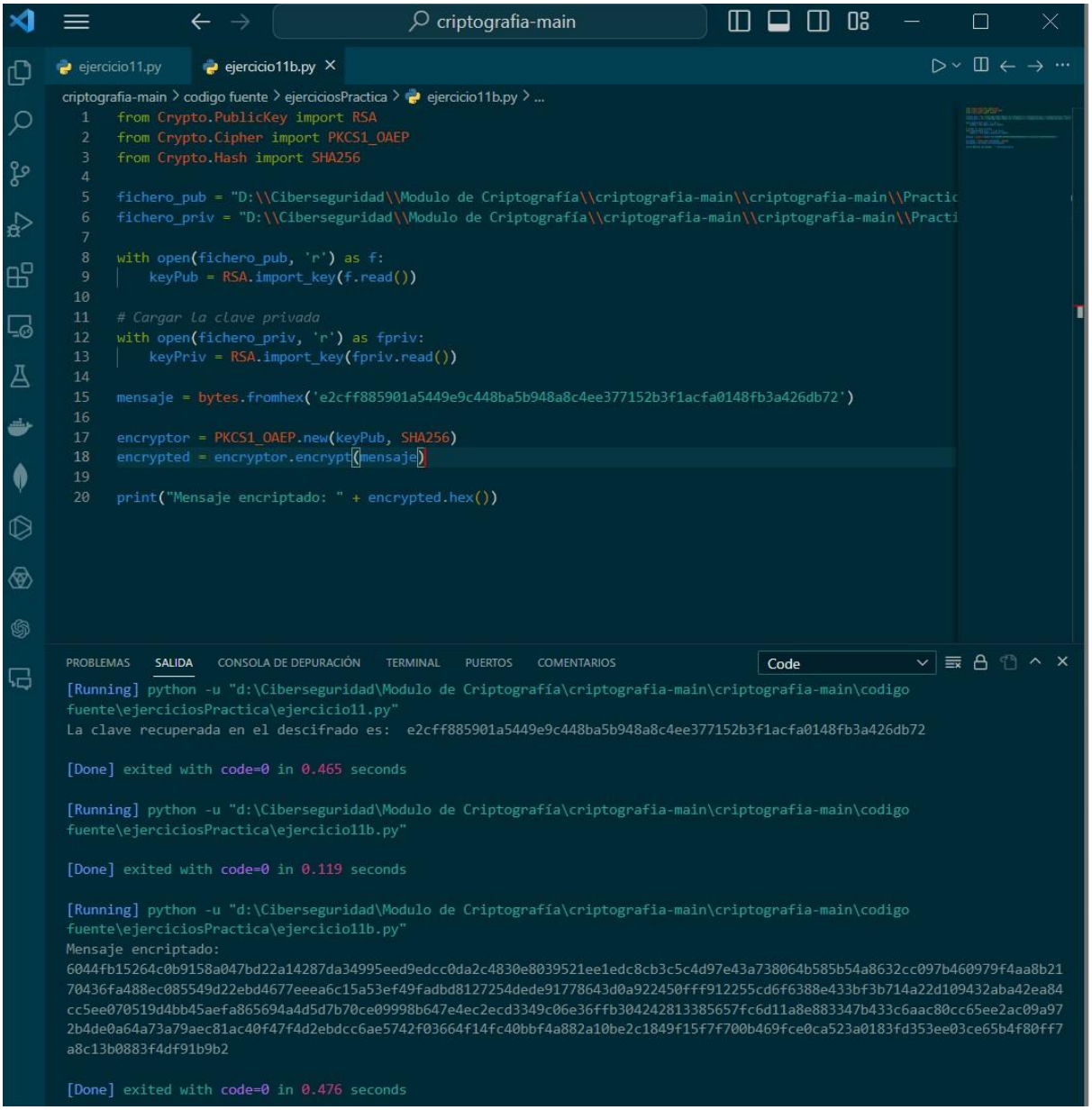
Primero desciframos bajo el algoritmo de criptografía asimétrica RSA-OAEP



Clave Recuperada:

```
e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72
```

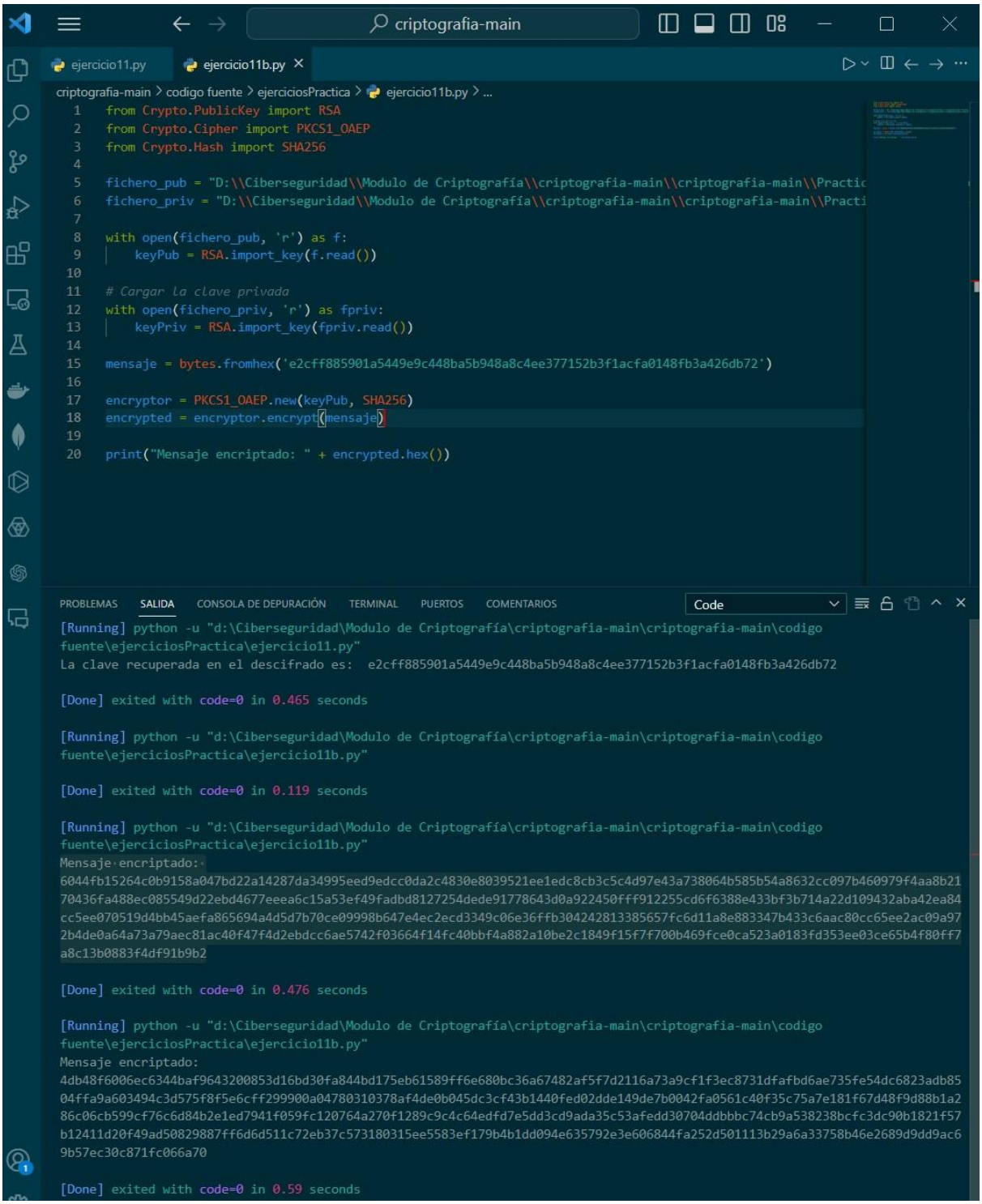
Segundo, ciframos bajo el algoritmo de criptografía asimétrica RSA-OAEP utilizando la clave obtenida en el descifrado anterior:



Obteniendo el siguiente mensaje en la primera ejecución:

```
Mensaje encriptado:
6044fb15264c0b9158a047bd22a14287da34995eed9edcc0da2c4830e8039521ee1edc8cb3c5c4
d97e43a738064b585b54a8632cc097b460979f4aa8b2170436fa488ec085549d22ebd4677eeea6
c15a53ef49fadbd8127254dede91778643d0a922450fff912255cd6f6388e433bf3b714a22d109
432aba42ea84cc5ee070519d4bb45aefa865694a4d5d7b70ce09998b647e4ec2ecd3349c06e36f
fb304242813385657fc6d11a8e883347b433c6aac80cc65ee2ac09a972b4de0a64a73a79aec81a
c40f47f4d2ebdcc6ae5742f03664f14fc40bbf4a882a10be2c1849f15f7f700b469fce0ca523a0
183fd353ee03ce65b4f80ff7a8c13b0883f4df91b9b2
```

Tercero, lo volvemos a ejecutar utilizando las mismas claves (Tanto pública como privada y la obtenida en el descifrado)



Obteniendo el siguiente mensaje cifrado:

```
4db48f6006ec6344baf9643200853d16bd30fa844bd175eb61589ff6e680bc36a67482af5f7d21
16a73a9cf1f3ec8731dfafbd6ae735fe54dc6823adb8504ffa9a603494c3d575f8f5e6cff29990
0a04780310378af4de0b045dc3cf43b1440fed02dde149de7b0042fa0561c40f35c75a7e181f67
d48f9d88b1a286c06cb599cf76c6d84b2e1ed7941f059fc120764a270f1289c9c4c64edfd7e5dd
```

```
3cd9ada35c53afedd30704ddbbbc74cb9a538238bcfc3dc90b1821f57b12411d20f49ad5082988
7ff6d6d511c72eb37c573180315ee5583ef179b4b1dd094e635792e3e606844fa252d501113b29
a6a33758b46e2689d9dd9ac69b57ec30c871fc066a70
```

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

R.- Si, he recuperado la clave y resulta que son diferentes los mensajes encriptados por el fenómeno de aleatoriedad que, en esencia sucede cada que volvamos a cifrar el texto debido a que vamos a tener un nuevo padding producto de un valor aleatorio llamado "SEED" que genera un nuevo “padding” para el texto cifrado cada que ejecutemos de nueva cuenta el cifrado.

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:E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A42

6DB74

Nonce:9Yccn/f5nJJhAt2S

¿Qué estamos haciendo mal?
En esencia el nonce el problema lo tenemos en el nonce porque este no satisface los requerimientos de aleatoriedad necesarios, Por definición debemos de generar este nonce de manera aleatoria para que así sea mucho más difícil para un actor malicioso lograr atacarnos.

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 presentalo en hexadecimal y en base64.

Texto cifrado en hexadecimal:

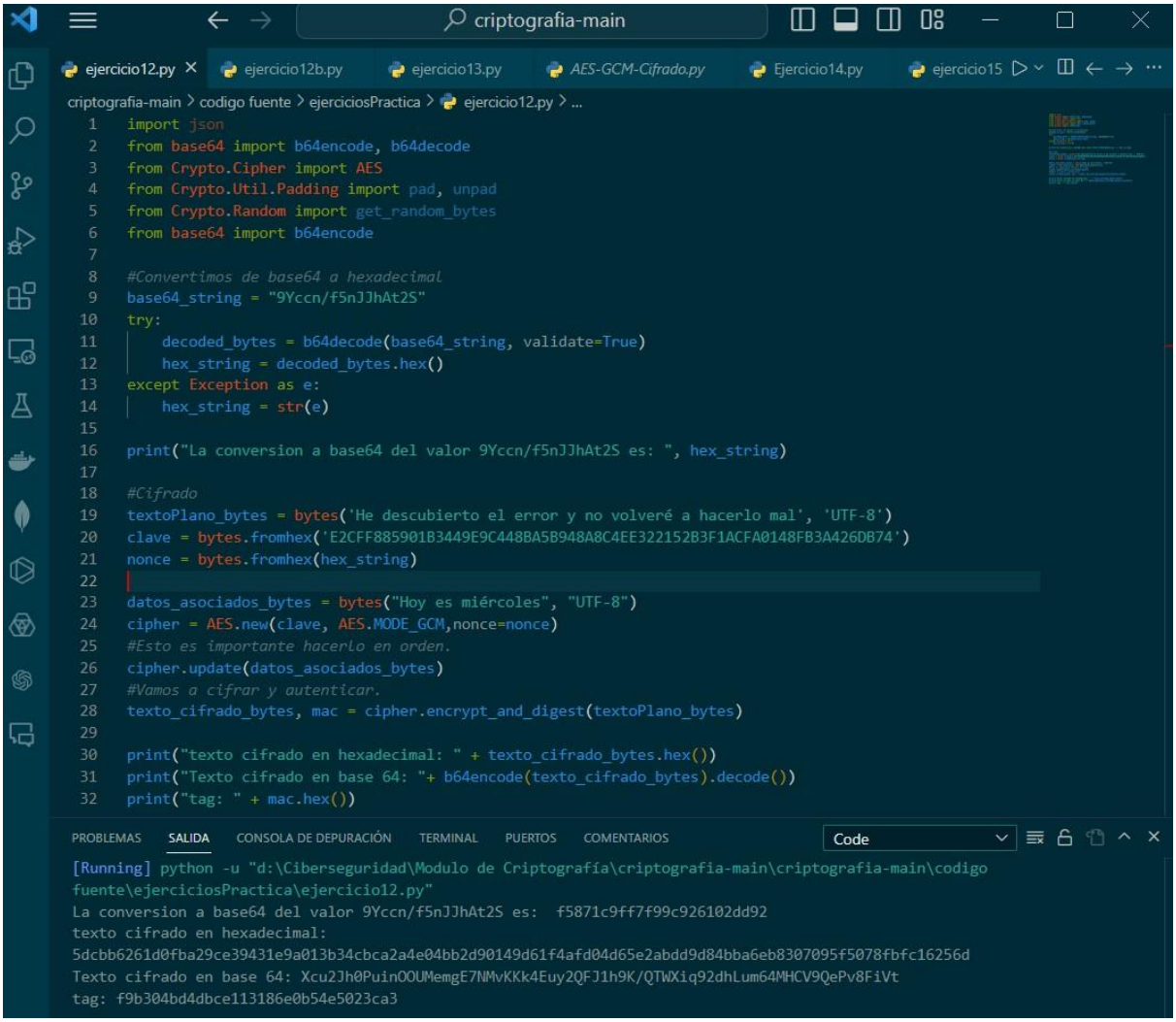
```
texto cifrado en hexadecimal:
5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6
eb8307095f5078fbfc16256d
```

Texto cifrado en base64:

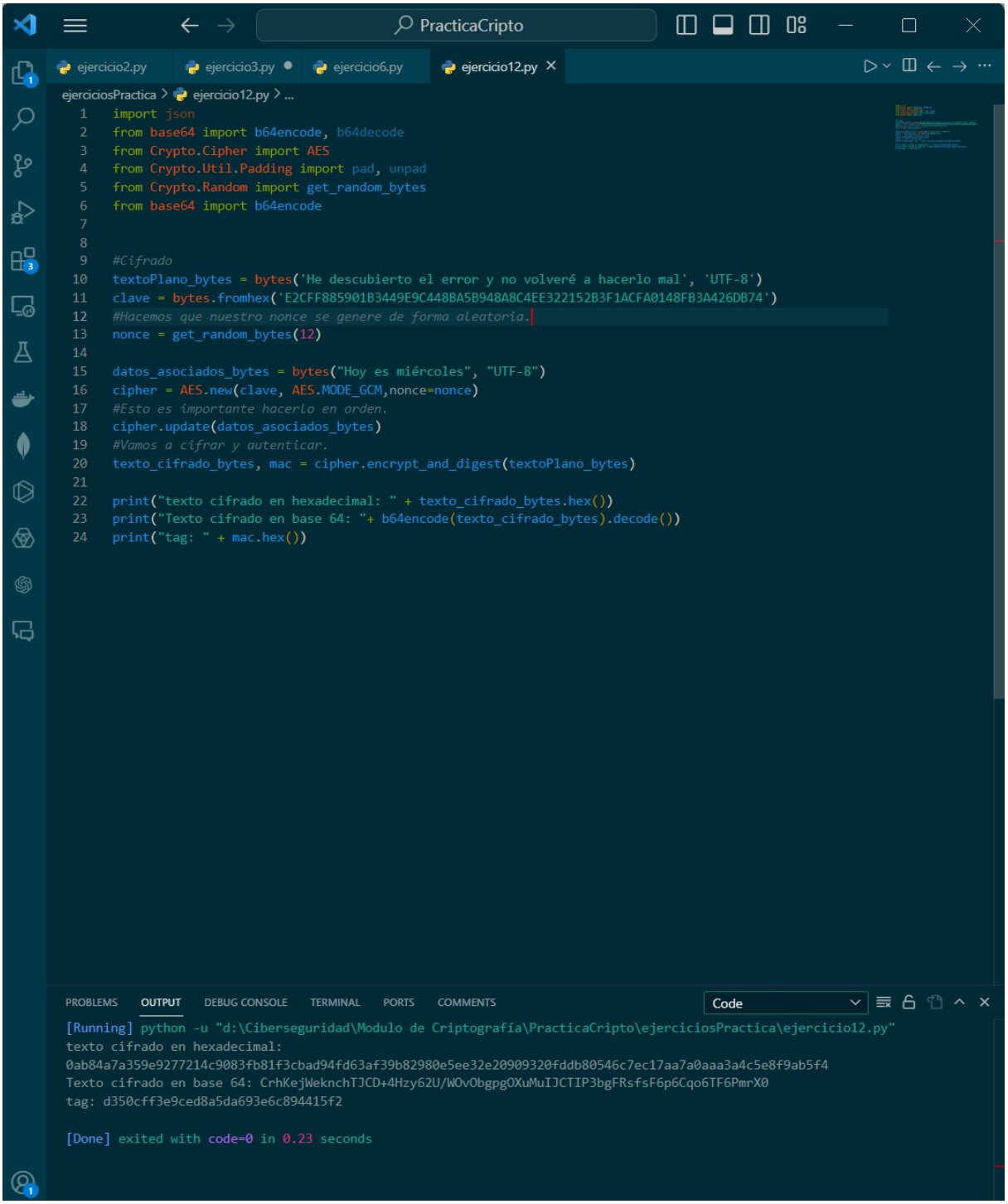
```
Texto cifrado en base 64:
Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq92dhLum64MHCV9QePv8FiVt
```

nonce en hexadecimal:

```
f5871c9ff7f99c926102dd92
```



Descifrado para comprobar que las claves en hexadecimal y el nonce aleatorio fueron bien logradas:



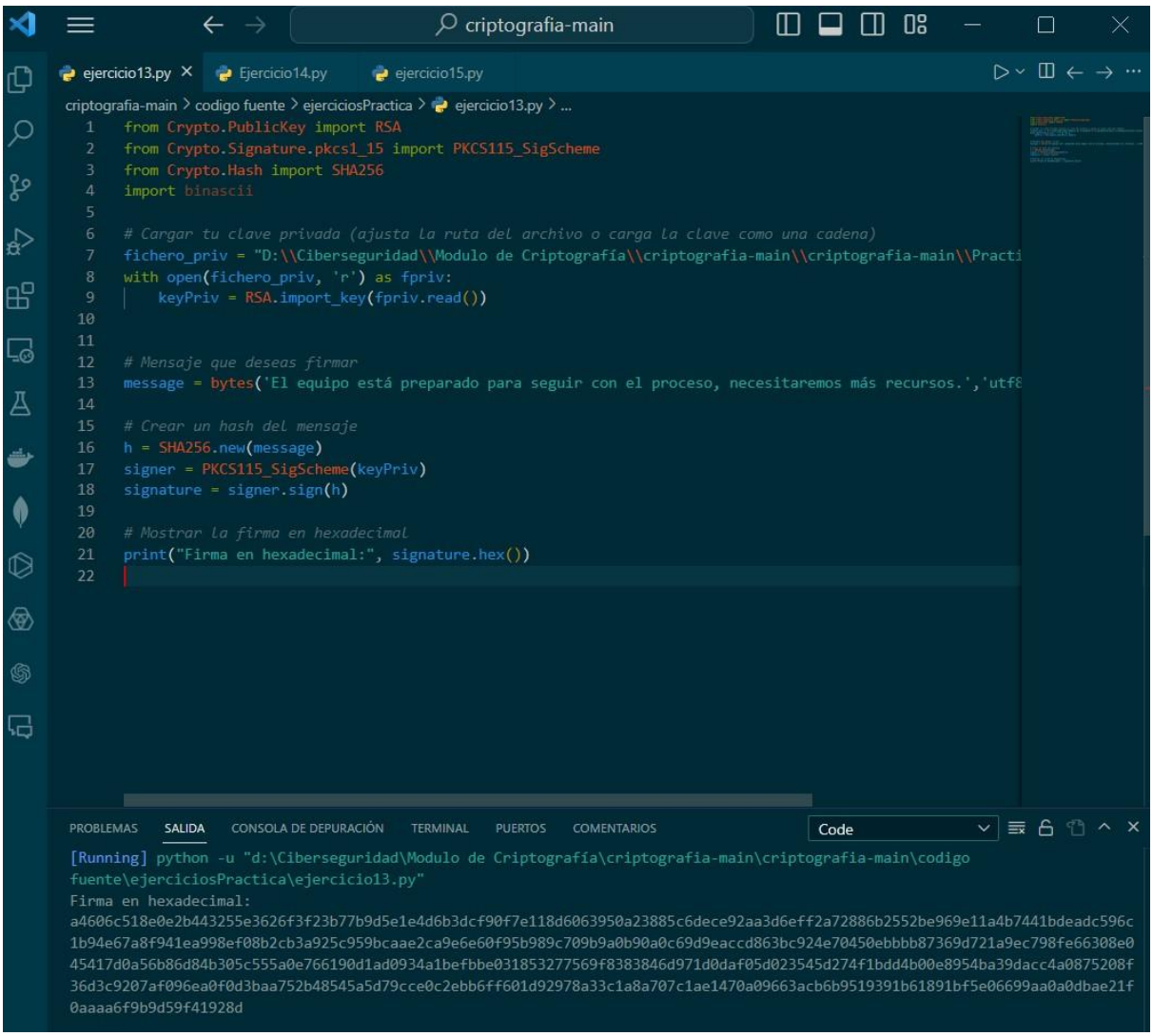
13. Se desea calcular una firma con el algoritmo PKCS#1 v1.5 usando las claves contenidas en los ficheros clave-rsa-oaep-priv y clave-rsa-oaep-publ.pem del mensaje siguiente:

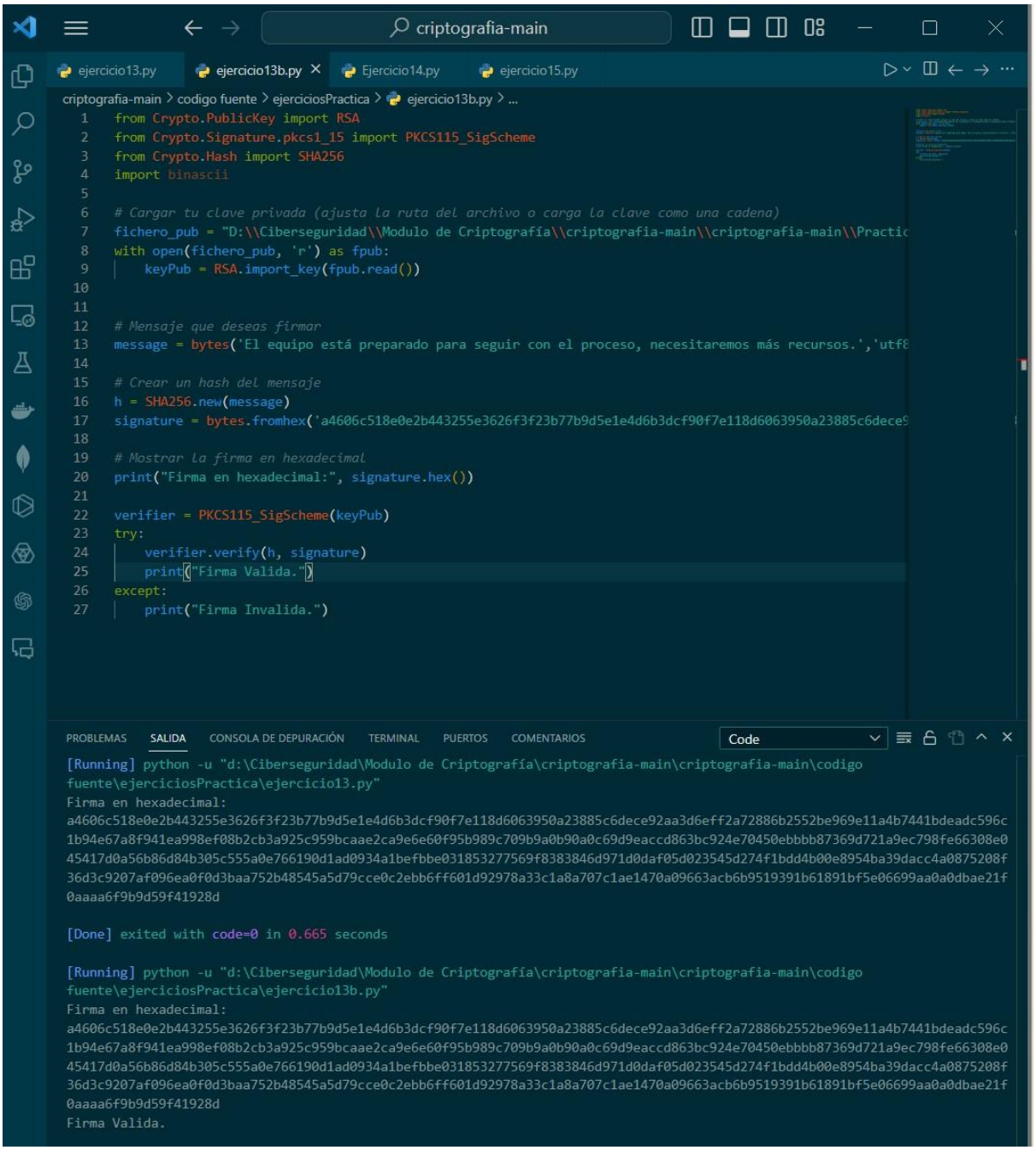
El equipo está preparado para seguir con el proceso, necesitaremos más recursos.

¿Cuál es el valor de la firma en hexadecimal?

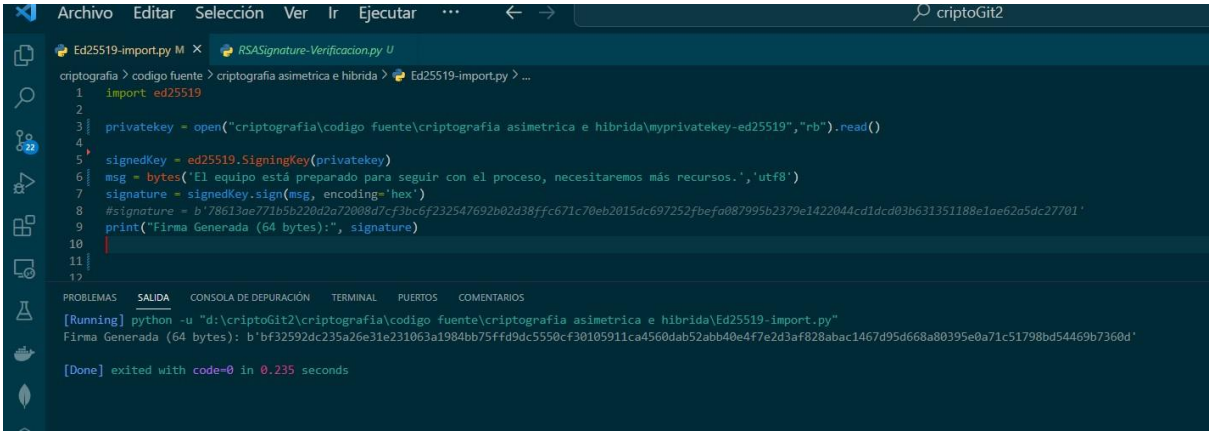
R.-

a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21f0aaaa6f9b9d59f41928d





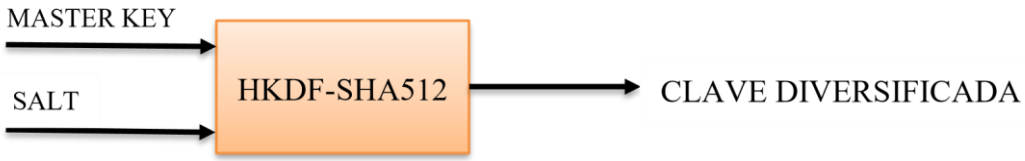
Calcula la firma (en hexadecimal) con la curva elíptica ed25519, usando las claves ed25519priv y ed25519-publ.



R.- Firma Generada (64 bytes):
bf32592dc235a26e31e231063a1984bb75ffd9dc5550cf30105911ca4560dab52abb40e4f7e2d3af828abac1467d95d668a80395e0a71c51798bd54469b7360d

14. Necesitamos generar una nueva clave AES, usando para ello una HKDF (HMAC-based Extractand-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



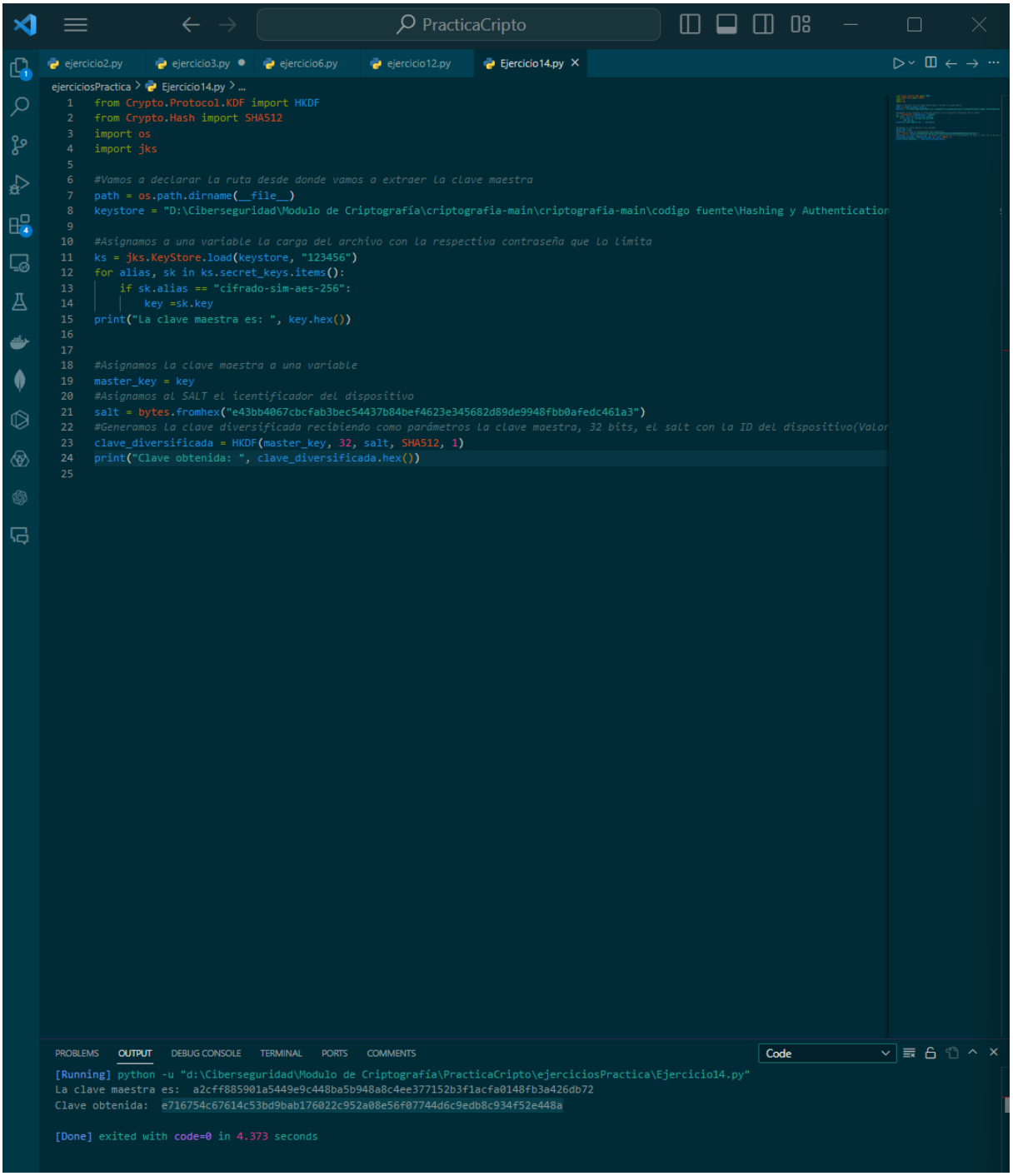
¿Qué clave se ha obtenido?

La clave maestra es
a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

Y

Clave Diversificada
es: e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a

En este caso en particular hemos utilizado el SALT como elemento diversificador, pasandole la clave en hexadecimal para realizar el proceso de obtener nuestra clave diversificada, un SHA512 le acompaña.



15. Nos envían un bloque TR31:

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDB
E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E0
3CD857FD37018E111B

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

A1A10101010101010101010101010102

¿Con qué algoritmo se ha protegido el bloque de clave? ***R.-*** Con un algoritmo de cifrado 'AES'
¿Para qué algoritmo se ha definido la clave? ***R.-***Para Un algoritmo de cifrado 'AES'
¿Para qué modo de uso se ha generado? ***R.-*** Nos permite cifrar y descifrar
¿Es exportable? ***R.-*** Si, Si es exportable
¿Para qué se puede usar la clave? ¿Qué valor tiene la clave? ***R.-*** Clave de cifrado simétrico para encriptar datos