Práctica criptografía Daniel Landeira

1. Primer ejercicio.

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.

1) 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: Al haberse generado la clave final a través de un XOR entre la clave en código y la Key Manager, se puede deducir que, conociendo la clave final y la escrita en código, haciendo un XOR entre estas dos últimas, se podría obtener la Key Manager que nos es desconocida, dando el resultado: **20553975c31055ed** .

2) 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?

Respuesta: Aplicando el mismo proceso que en el apartado 1 del ejercicio, usando la tabla lógica XOR, obtenemos que la clave en memoria será: **8653f75d31455c0 .**

2. Segundo Ejercicio.

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/o0QKlWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nvvJbBiG+le1ZhpR84oI=”

Para este caso, se ha usado un AES/CBC/PKCS7. Si lo desciframos, ¿qué obtenemos?

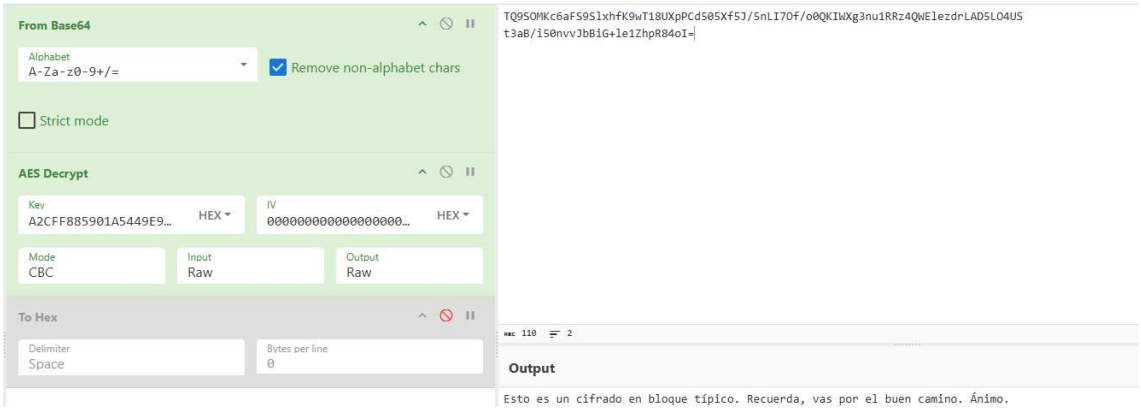
Respuesta: “Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.” Realizado la obtención de la clave por Python y el desencriptado del mensaje a través de la página “Cyberchef” de forma

rápida. Adicionalmente también se puede obtener por Python con el siguiente código.

```python
import jks
import os

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

keystore = path + "/KeyStorePracticas"


ks = jks.KeyStore.load(keystore, store_password: "123456")

for alias, sk in ks.secret_keys.items():
    if sk.alias == "cifrado-sim-aes-256":
        key = sk.key

print("La clave es:", key.hex())
```

```
C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:\Users\Usuario\Desktop\Bootcamp\Criptografia\criptografia\codigo fuente\Gestión de Claves\Ejemplo KeyStore.py"
La clave es: a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

Process finished with exit code 0
```

Código y respuesta de Python para la obtención de la clave.



Obtención del desencriptado en CyberChef.

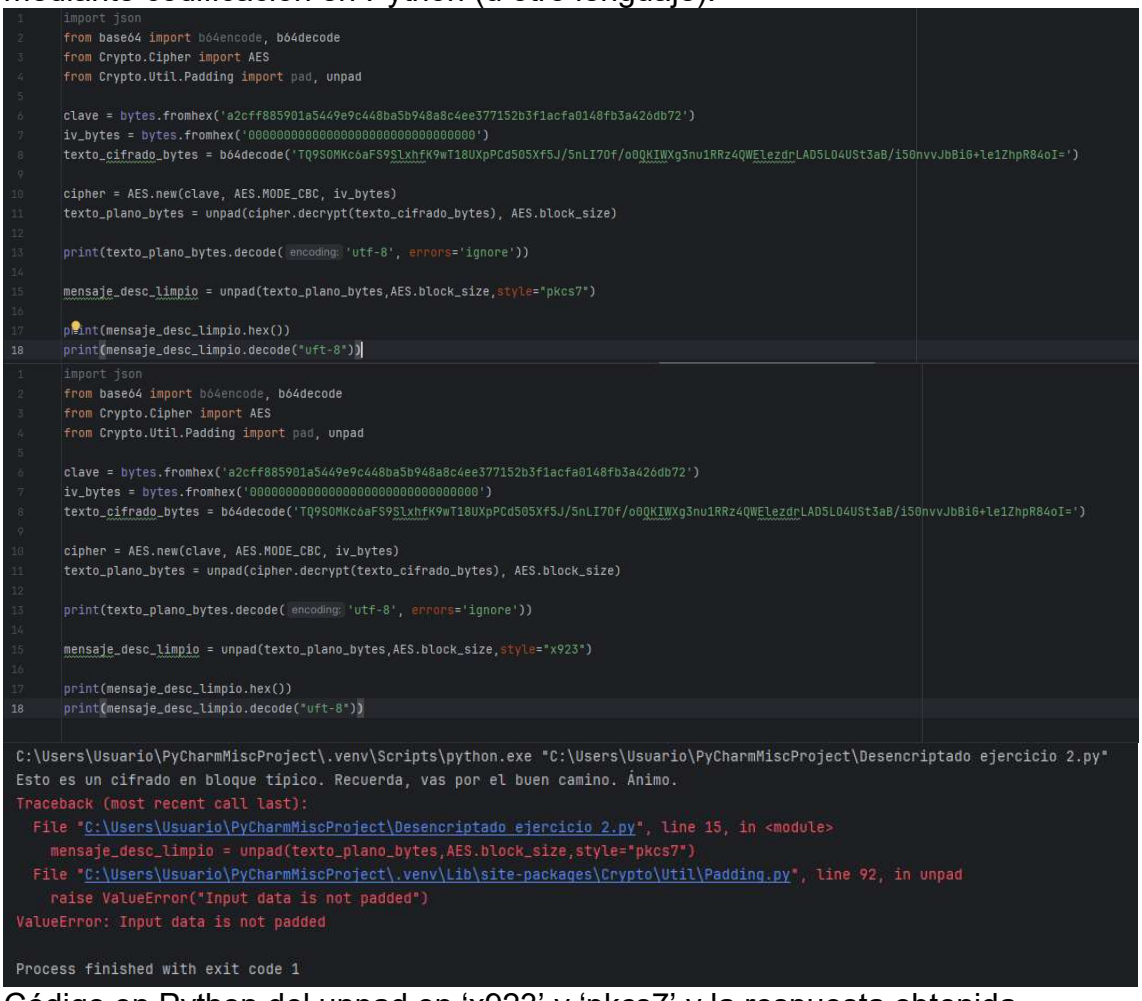
```python
import json
from base64 import b64encode, b64decode
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

clave = bytes.fromhex('a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72')
iv_bytes = bytes.fromhex('00000000000000000000000000000000')
texto_cifrado_bytes = b64decode('TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nvvJbBiG+le1ZhpR84oI=')

cipher = AES.new(clave, AES.MODE_CBC, iv_bytes)
texto_plano_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size)

print(texto_plano_bytes.decode( encoding: 'utf-8', errors='ignore'))
```

```
C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:\Users\Usuario\PyCharmMiscProject\Desencriptado ejercicio 2.py"
Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.

Process finished with exit code 0
```

Código y resultado en Python del desencriptado.

¿Qué ocurre si decidimos cambiar el padding a x923 en el descifrado?
Respuesta: tanto en el formato de padding ‘x923’ como con el formato ‘pkcs7’, la respuesta en Python es que “Los datos no están padeados”, por tanto esto resuelve también la siguiente pregunta, puesto que no existe padding.
¿Cuánto padding se ha añadido en el cifrado?
Respuesta: Ninguno.

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



```
import json
from base64 import b64encode, b64decode
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

clave = bytes.fromhex('a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72')
iv_bytes = bytes.fromhex('00000000000000000000000000000000')
texto_cifrado_bytes = b64decode('TQ9SOMKcóaFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nvvJbBi6+le1ZhpR84oI=')

cipher = AES.new(clave, AES.MODE_CBC, iv_bytes)
texto_plano_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size)

print(texto_plano_bytes.decode( encoding: 'utf-8', errors='ignore'))

mensaje_desc_limpio = unpad(texto_plano_bytes,AES.block_size,style="pkcs7")

print(mensaje_desc_limpio.hex())
print(mensaje_desc_limpio.decode("uft-8"))
```

```
import json
from base64 import b64encode, b64decode
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

clave = bytes.fromhex('a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72')
iv_bytes = bytes.fromhex('00000000000000000000000000000000')
texto_cifrado_bytes = b64decode('TQ9SOMKcóaFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nvvJbBi6+le1ZhpR84oI=')

cipher = AES.new(clave, AES.MODE_CBC, iv_bytes)
texto_plano_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size)

print(texto_plano_bytes.decode( encoding: 'utf-8', errors='ignore'))

mensaje_desc_limpio = unpad(texto_plano_bytes,AES.block_size,style="x923")

print(mensaje_desc_limpio.hex())
print(mensaje_desc_limpio.decode("uft-8"))
```

```
C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:\Users\Usuario\PyCharmMiscProject\Desencriptado ejercicio 2.py"
Esto es un cifrado en bloque tipico. Recuerda, vas por el buen camino. Ánimo.
Traceback (most recent call last):
  File "C:\Users\Usuario\PyCharmMiscProject\Desencriptado ejercicio 2.py", line 15, in <module>
    mensaje_desc_limpio = unpad(texto_plano_bytes,AES.block_size,style="pkcs7")
  File "C:\Users\Usuario\PyCharmMiscProject\.venv\Lib\site-packages\Crypto\Util\Padding.py", line 92, in unpad
    raise ValueError("Input data is not padded")
ValueError: Input data is not padded

Process finished with exit code 1
```

Código en Python del unpad en ‘x923’ y ‘pkcs7’ y la respuesta obtenida.

3. Tercer Ejercicio

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 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: El cifrado en este caso se podría mejorar de forma sencilla estableciendo un 'nonce' aleatorio, generando mayor integridad, ya que cada encriptado tendría un 'nonce' único generado en el momento específicamente para ese encriptado. En este caso, con los datos suministrados, el mensaje codificado sería:

- Mensaje cifrado en HEX = 69ac4ee7c4c552537a00a19bcaf7f0aaed7c9c8f769956a09bce6fadef6c3535f2211c9467067cf5c4a842ab
- Mensaje cifrado en B64 = aaxO58TFUlN6AKGbyvfwqu18nI92mVagm85vre9sNTXyIRyUZwZ89cSoQqs=

```
C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:\Users\Usuario\Desktop\Bootcamp\Criptografia\criptografia\codigo
Mensaje cifrado en HEX =  69ac4ee7c4c552537a00a19bcaf7f0aaed7c9c8f769956a09bce6fadef6c3535f2211c9467067cf5c4a842ab
Mensaje cifrado en B64 =  aaxO58TFUlN6AKGbyvfwqu18nI92mVagm85vre9sNTXyIRyUZwZ89cSoQqs=

Process finished with exit code 0
```

```
from Crypto.Cipher import ChaCha20
from base64 import b64decode, b64encode
from Crypto.Random import get_random_bytes

textoPlano = 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('AF9DF30474898787A456056CCB9B936D33B780D03CABC81719D52383480DC3120')
#Importante NUNCA debe fijarse el nonce, en este caso lo hacemos para mostrar el mismo resultado en cualquier lenguaje.
#nonce_mensaje = get_random_bytes(8)
#print('nonce  = ', nonce_mensaje.hex())
nonce = b64decode('9Yccn/f5nJJhAt2S')


#Con la clave y con el nonce se cifra. El nonce debe ser único por mensaje
cipher = ChaCha20.new(key=clave, nonce=nonce)
texto_cifrado = cipher.encrypt(textoPlano)

print('Mensaje cifrado en HEX = ', texto_cifrado.hex())
print('Mensaje cifrado en B64 = ', b64encode(texto_cifrado).decode())
```

Código para obtener el cifrado del texto.

Con la variación del lenguaje de programación para generar un 'nonce' aleatorio sería:

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


textoPlano = 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 = get_random_bytes(8)
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 = cipher.encrypt(textoPlano)

print('Mensaje cifrado en HEX = ', texto_cifrado.hex())
print('Mensaje cifrado en B64 = ', b64encode(texto_cifrado).decode())


 C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:\Users\Usuario\Desktop\Bootcamp\Criptografia\criptografia\co
 nonce  =  e5dde62a52c9010d
 Mensaje cifrado en HEX =  f8e676997812cb61719c08ca1b8e447767f3a44644bfacfb4445b06dd57c6420759918f0455c7410aab5160a
 Mensaje cifrado en B64 =  +OZ2mXgSy2FxnAjKG45Ed2fzpEZEv6z7REWwbdV8ZCB1mRjwRVxOEKq1Fgo=

 Process finished with exit code 0
```

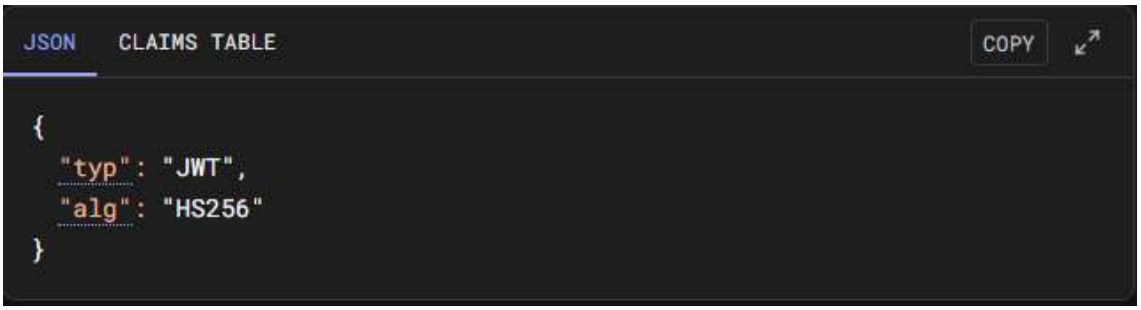
Código y solución con la implementación del ‘nonce’ aleatorio.

4. Cuarto ejercicio

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNMzfQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

¿Qué algoritmo de firma hemos realizado? Respuesta:



¿Cuál es el body del jwt?
Respuesta: El body del jwt es la parte del código separada entre puntos ‘.’, por tanto corresponde a:
“eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNMzfQ”

Un hacker está enviando a nuestro sistema el siguiente jwt:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODlHRI

¿Qué está intentando realizar?
Respuesta: El hacker estaría intentando realizar un ataque de "sustitución de algoritmo" al intentar manipular el token del jwt para cambiar entre algoritmos simétricos (HS256) y asimétricos (SHA256).

¿Qué ocurre si intentamos validarlo con pyjwt?
Respuesta: El código se caería/fallaría al estar el token manipulado y no coincidir la verificación de las credenciales.

5. Quinto ejercicio

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: por la cantidad de bytes del hash obtenido, se ha realizado un SHA3 256.

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

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

¿Qué hash hemos realizado? Respuesta: siguiendo el mismo método que en la anterior pregunta, por comparación de cantidad de bytes obtenidos, se debe realizar un SHA2 512.
Genera ahora un SHA3 Keccak de 256 bits con el siguiente texto: "En KeepCoding aprendemos como protegernos con criptografia." ¿Qué propiedad destacarías del hash, atendiendo a los resultados anteriores?

Respuesta: El hash obtenido sería "5fa324631a34e0aab1419c2702158c91a625b8d12fb973892f4cf1cf065975ff". Una propiedad a destacar muy importante, es que cada has para encriptación genera un código y tamaño únicos.

6. Sexto

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.

Respuesta: El hmac es:
“857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550”

```
from Crypto.Hash import HMAC, SHA256

#HMAC clave y datos --- hmac=17938c1b46db10b099e3d0ccc96b685b82a793481b20a931f6e1df7711b8e785
clave_bytes = bytes.fromhex('A212A51C997E14B4DF08D55967641B0677CA31E049E672A4B06861AA4D5826EB')
datos = bytes("Siempre existe más de una forma de hacerlo, y más de una solución válida.", "utf8")
hmac256 = HMAC.new(clave_bytes, msg=datos, digestmod=SHA256)
print("El hmac es:",hmac256.hexdigest())

C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:\Users\Usuario\Desktop\Bootcamp\Crip
El hmac es: 857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550

Process finished with exit code 0
```

Código y respuesta de Python para la obtención del “HMAC”.

7. Séptimo

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: El SHA-1 se considera una mala opción debido a la “fragilidad” de los hashes que produce, siendo relativamente cortos y “fáciles” de descifrar, teniendo una longitud de apenas 20 bytes, que serían 40 caracteres en hexadecimal. Esto podría ser descifrado relativamente fácil con un terminal de ataque adecuado.

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: La forma de poder fortalecer el SHA-256 (se entiende que es un SHA2 al no llevar número de versión), sería implementarle un ‘MAC’, que es un código de autentificación que te asegura que el mensaje no ha sido modificado de ninguna manera, ya que, al descifrar el mensaje enviado, debe coincidir al 100%, sino el MAC tiraría el proceso de descifrado por un fallo de concordancias.

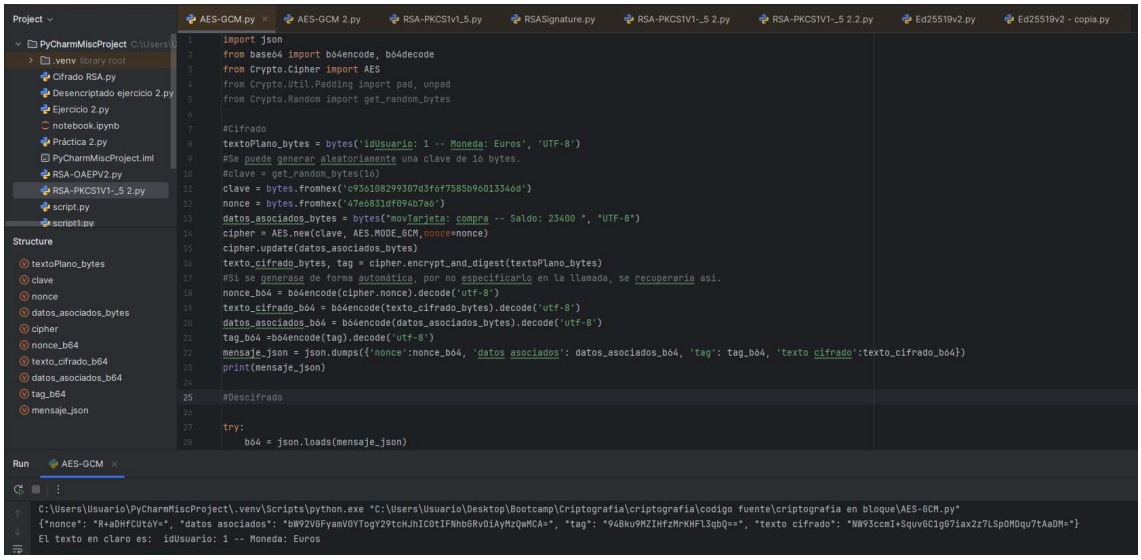
Parece que el responsable se ha quedado conforme, tras mejorar la propuesta del SHA-256, no obstante, hay margen de mejora. ¿Qué propondrías?

- Respuesta: La mejor opción es usar Argon2, que es un algoritmo moderno y muy seguro para proteger contraseñas. Argon2 está diseñado para ser resistente a ataques, también es configurable para ajustar su seguridad y es considerado el estándar actual para almacenamiento seguro de contraseñas.

8. Octavo

En este ejercicio, para mantener los parámetros necesarios de manera confidencial, se podrían utilizar los métodos de encriptado de ChaCha20 Poly o también el AES-GCM, que nos permiten tratar varios campos (que serían nuestros datos a mantener confidenciales y los otros datos irrelevantes) de manera distinta. Adicionalmente, en el caso del AES-GCM, incluye la

generación de un ‘Tag’ en el momento del cifrado, que es una etiqueta pequeña de 16 bytes que funciona como firma digital que garantiza la integridad y autenticidad del mensaje.
Teniendo esto presente, un ejemplo de codificación de esos datos sería cambiar dentro de la programación (en AES-GCM) el nombre de la impresión por los conceptos a modificar, por ejemplo, el texto asociado a codificar el *Saldo y movTarjeta*, mientras que el texto plano lo mantendremos como *idUsuario* y *Moneda.*



9. Noveno

Se requiere calcular el KCV de la 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.
Respuesta: El KCV es: Parte AES “5244db”; parte SHA-256 “db7df2” siendo este último el KCV de la clave solicitado.

```python
import hashlib
import json
from base64 import b64encode, b64decode
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad


#Cifrado
textoPlano_bytes = bytes.fromhex('00000000000000000000000000000000')

clave = bytes.fromhex('A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72')
iv_bytes = bytes.fromhex('00000000000000000000000000000000')
cipher = AES.new(clave, AES.MODE_CBC,iv_bytes)
texto_cifrado_bytes = cipher.encrypt(pad(textoPlano_bytes, AES.block_size,  style='pkcs7'))
print("KCV AES:", texto_cifrado_bytes.hex()[0:6])
print("texto_cifrado con padding: ", texto_cifrado_bytes.hex())

#cipher2 = AES.new(clave, AES.MODE_CBC,iv_bytes)
#texto_cifrado_bytes = cipher2.encrypt(textoPlano_bytes)
#print("KCV AES:", texto_cifrado_bytes.hex()[0:6])
#print("texto_cifrado sin padding: ", texto_cifrado_bytes.hex())

m = hashlib.sha256()
m.update(bytes.fromhex("A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"))
print("KCV SHA256: " + m.digest().hex()[0:6])
```

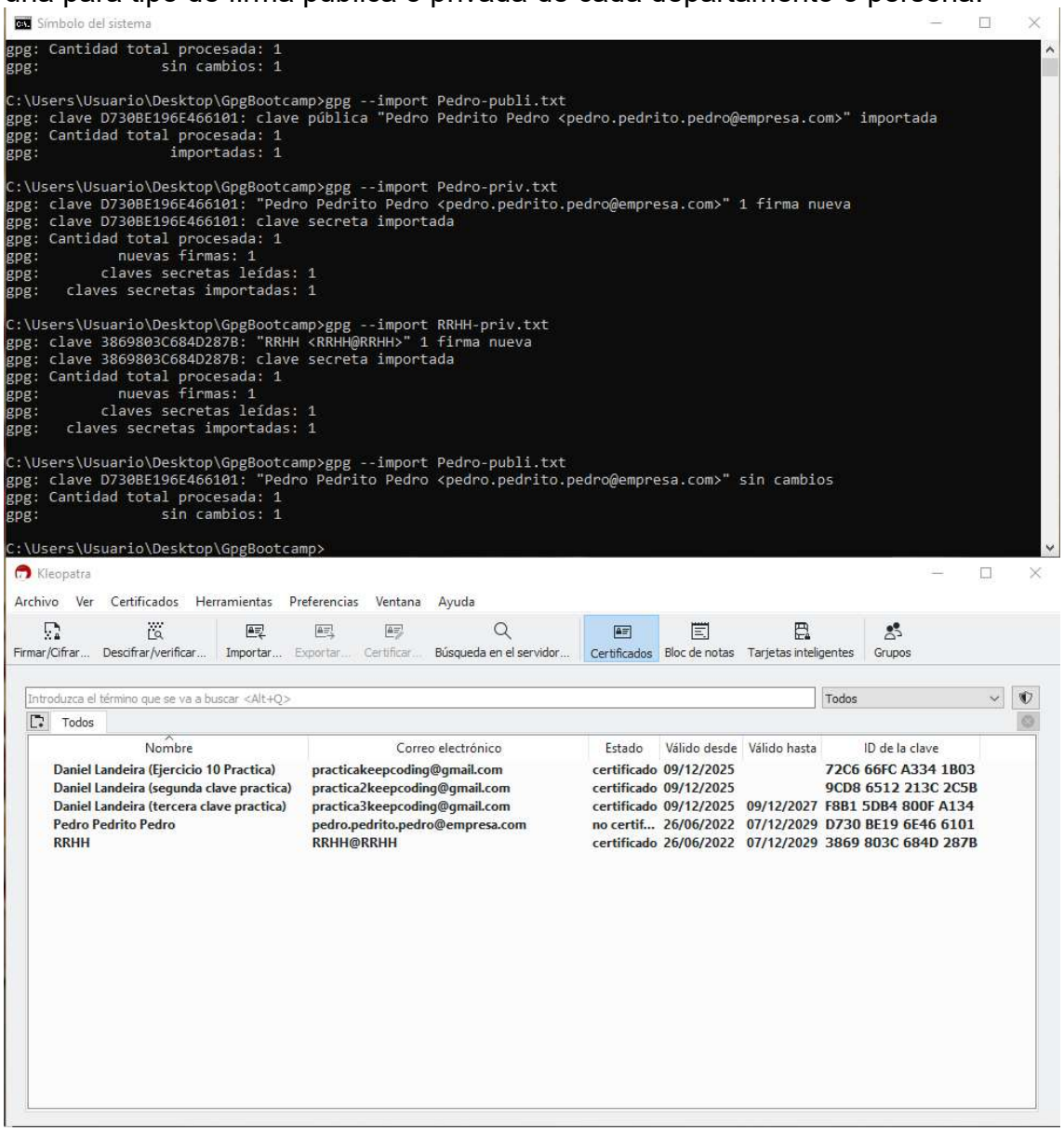
```
C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:\Users\Usuario\Desktop\Bootcamp\Criptografia\
KCV AES: 5244db
texto_cifrado con padding:  5244dbd02d57d56ae08e064c56c7ca74a35eccad6db31f05841bde3d4e3ada4a
KCV SHA256: db7df2

Process finished with exit code 0
```

Código y Respuesta de Python de la obtención del KCV(SHA-256) y del KCV(AES)

10. Decimo

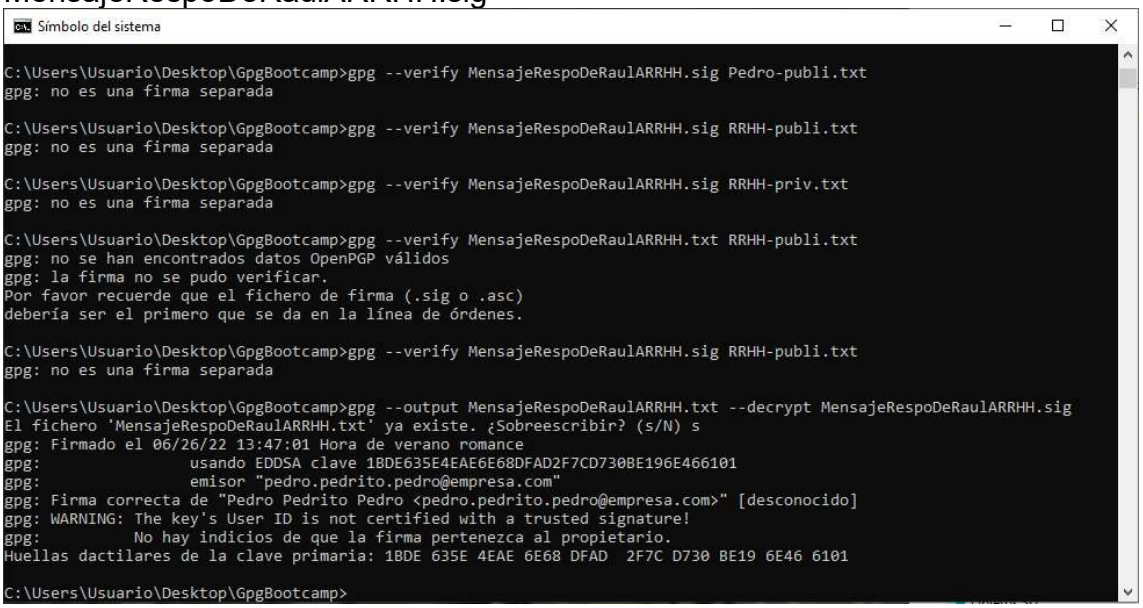
Primero, importamos todas las claves con el comando:

“gpg –import RRHH-priv/publi.txt // Pedro-priv/publi.txt”. En cuatro tandas, cada una para tipo de firma pública o privada de cada departamento o persona.

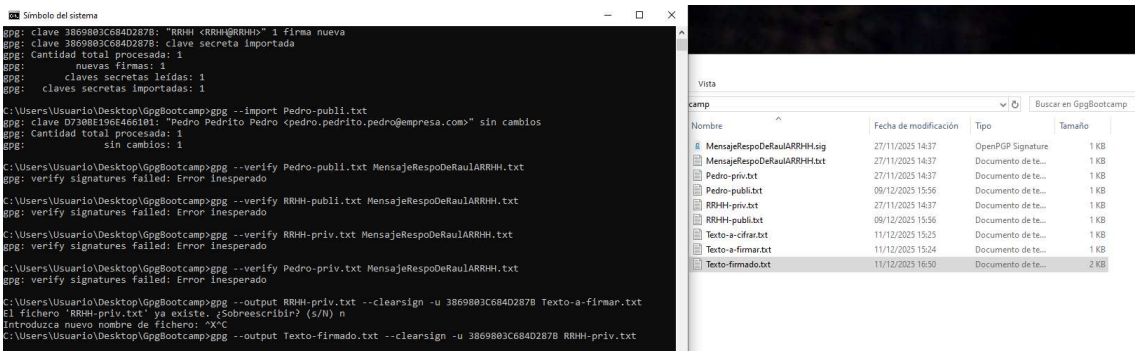


Seguido, verificamos la firma del archivo enviado por Pedro con el comando:

“--output MensajeRespoDeRaulARRHH.txt --decrypt MensajeRespoDeRaulARRHH.sig”

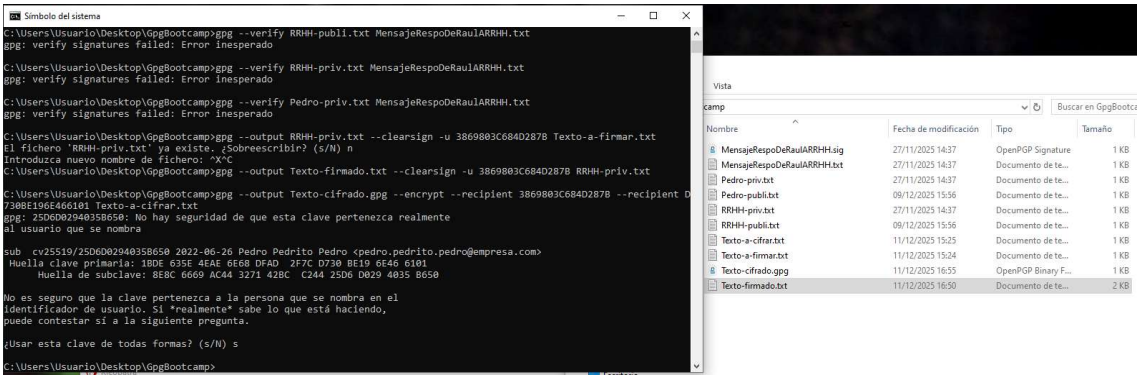


Una vez verificado la autenticidad de la firma, pasamos a firmar el siguiente documento, de parte de recursos humanos, con el comando:
“--output RRHH-priv.txt --clearsign -u 3869803C684D287B Texto-a-firmar.txt”.



Hecho esto, procedemos a cifrar, con la clave de RRHH y Pedro, el siguiente documento solicitado, mediante el comando:

“--output Texto-cifrado.gpg --encrypt --recipient 3869803C684D287B --recipient D730BE196E466101 Texto-a-cifrar.txt”.



11. Decimoprimero

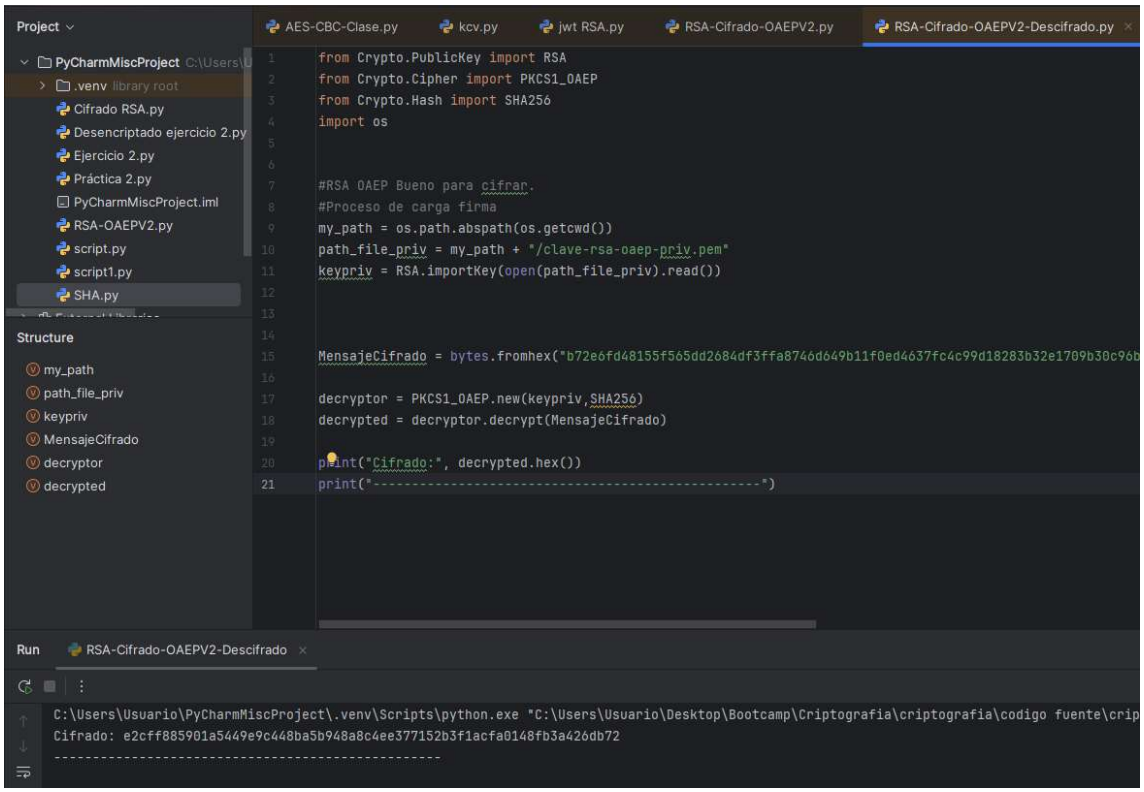
Nuestra compañía tiene un contrato con una empresa que nos da un servicio de almacenamiento de información de videollamadas. Para lo cual, la misma nos envía la clave simétrica de cada videollamada cifrada usando un RSA-OAEP. El hash que usa el algoritmo interno es un SHA-256.

El texto cifrado es el siguiente:

b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709
b30c96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d04e3d
3d4ad629793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae2c4f5b
a670a4126f2f14a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8
624071be78ccef573d896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f6
7d3f98b4cd907f27639f4b1df3c50e05d5bf63768088226e2a9177485c54f72407f
df358fe64479677d8296ad38c6f177ea7cb74927651cf24b01dee27895d4f05fb5c
161957845cd1b5848ed64ed3b03722b21a526a6e447cb8ee

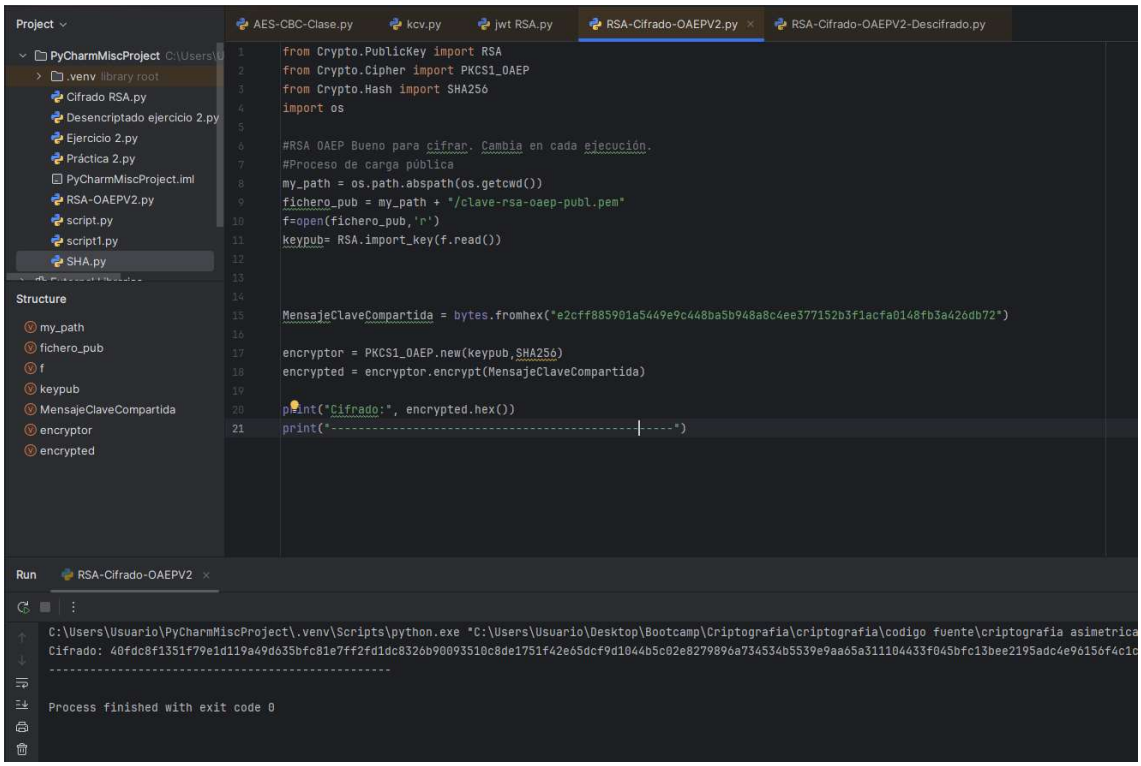
Las claves pública y privada las tenemos en los ficheros clave-rsa-oaep-publ.pem y clave-rsaoaep-priv.pem.

Si has recuperado la clave, vuelve a cifrarla con el mismo algoritmo.
Respuesta: La clave obtenida es:
“e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72”



Código de descifrado de la clave.
La clave re-cifrada es:
“40fdc8f1351f79e1d119a49d635bfc81e7ff2fd1dc8326b90093510c8de1751f42e
65dcf9d1044b5c02e8279896a734534b5539e9aa65a311104433f045bfc13bee2
195adc4e96156f4c1c1b8b283981eb889fe084e9a25e1a47c690c5d6958133b09
0cfea8fe9a00d5cbea56e13a34a2e76d5316500dfb29ebb5154283a4ed7ee720f
9a2435ec2ed18bacc3ed6c0c4123f668bd3bb832af0ed486f40e14d7df0e620825

17c93a815f6a5780fb583c5afeb1b71533be63a0a6fb6945b803c53e54c29c9e61e66b10b44bd1ea040ed167b2b168e8b61461a8bae02bf5d7389134e0b217b3893bfb7ebfb729e9d4335d2be72507382273c892ec2953a0461925bd” .



Código de obtención del nuevo cifrado de la clave.

¿Por qué son diferentes los textos cifrados?
Respuesta: La clave sale con un cifrado distinto debido a que ahora quien cifra somos nosotros como empresa, debido a que la clave pública de firma es distinta, generando así otro cifrado distinto al incluir un nuevo relleno aleatorio generado en cada cifrado.

12. Decimosegundo

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: De inicio, el ‘nonce’ debería ser único y aleatorio para cada encriptado.

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:

- Base 64: Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq92dhLum64MHCV9QePv8FiVt
- Hex: 5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d

```python
from Crypto.Cipher import AES
from base64 import b64encode, b64decode
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

# AES-GCM --> (Datos Asociados + Datos a cifrar) + key + nonce

texto_gcm_bytes = bytes("He descubierto el error y no volveré a hacerlo mal", "utf-8")
key_bytes  = bytes.fromhex('E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74')
nonce_bytes = b64decode('9Yccn/f5nJJhAt2S')
#print("Nonce hex=", nonce_bytes(base64))
datos_asociados_bytes = bytes("Id usuario","utf-8")
cifrador = AES.new(key_bytes,AES.MODE_GCM,nonce=nonce_bytes)
cifrador.update(datos_asociados_bytes)
texto_cifrado_bytes,mac_bytes = cifrador.encrypt_and_digest(texto_gcm_bytes)
print("Texto cifrado:", b64encode(texto_cifrado_bytes).decode('utf-8'))
print("MAC:", mac_bytes.hex())

```

Código para la obtención del cifrado en Base64.

AES-CBC-Clase.py | kcv.py | jwt RSA.py | RSA-Cifrado-OAEPV2.py | RSA-Cifrado-OAEPV2-Descifrado.py | AES-GCM-Ci

```python
from Crypto.Cipher import AES
from base64 import b64encode, b64decode
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

# AES-GCM --> (Datos Asociados + Datos a cifrar) + key + nonce

texto_gcm_bytes = bytes("He descubierto el error y no volveré a hacerlo mal", "utf-8")
key_bytes  = bytes.fromhex('E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74')
nonce_bytes = b64decode('9Yccn/f5nJJhAt2S')
#print("Nonce hex=", nonce_bytes(base64))
datos_asociados_bytes = bytes("Id usuario","utf-8")
cifrador = AES.new(key_bytes,AES.MODE_GCM,nonce=nonce_bytes)
cifrador.update(datos_asociados_bytes)
texto_cifrado_bytes,mac_bytes = cifrador.encrypt_and_digest(texto_gcm_bytes)
print("Texto cifrado:", texto_cifrado_bytes.hex())
print("MAC:", mac_bytes.hex())

```

Código para la obtención del cifrado en hexadecimal.

13. Decimotercero

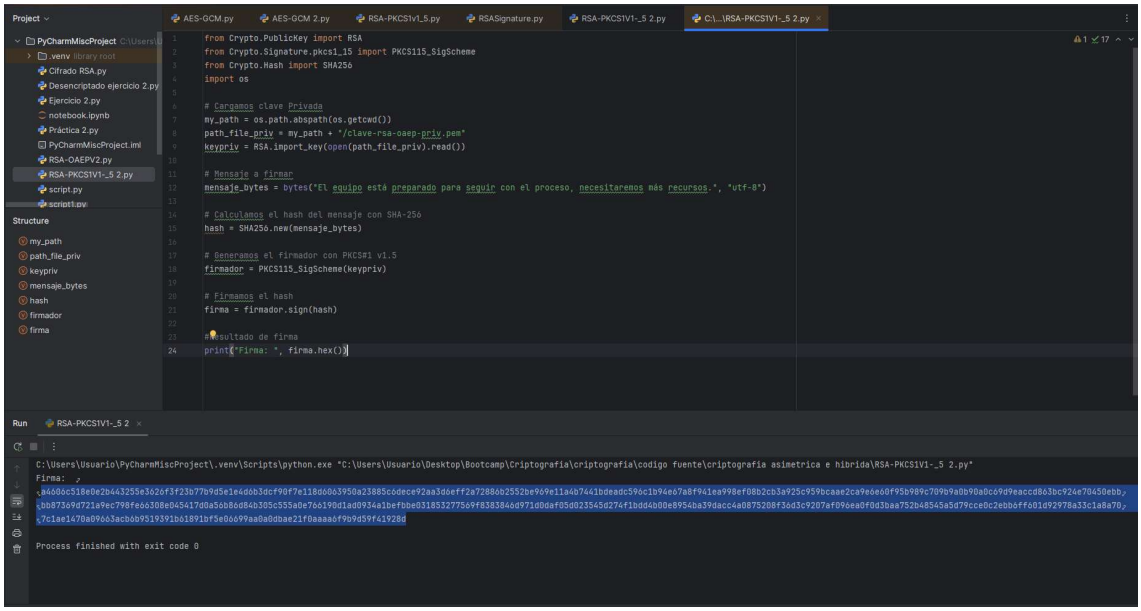
Se desea calcular una firma con el algoritmo PKCS#1 v1.5 usando las claves contenidas en los ficheros clave-rsa-oaep-priv y clave-rsa-oaep-publ.pem del mensaje siguiente:

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

¿Cuál es el valor de la firma en hexadecimal?

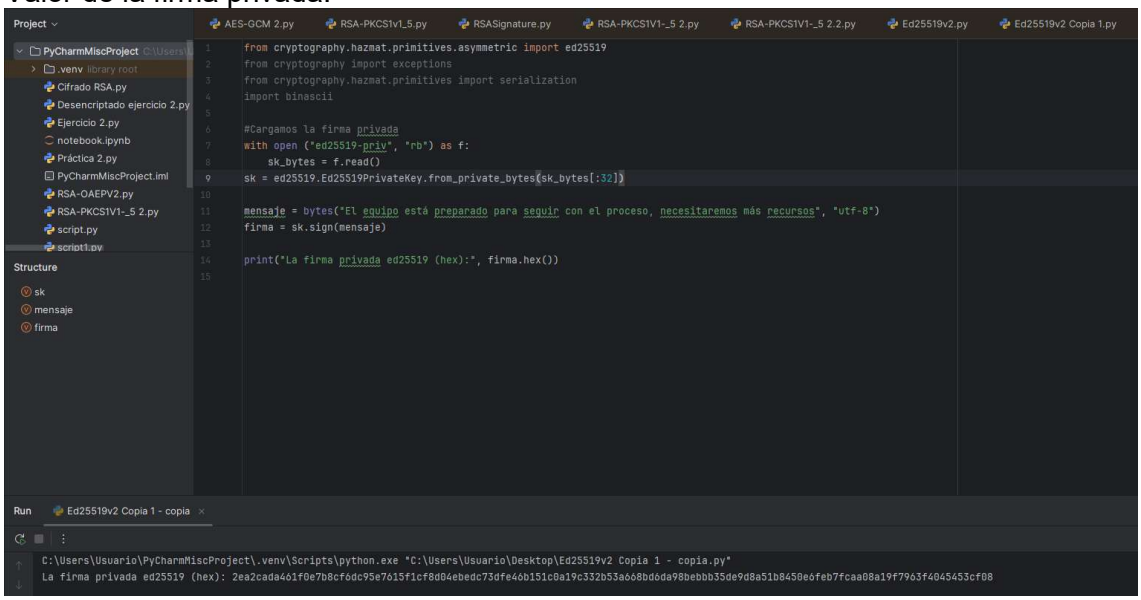
Valor de la firma:

a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21f0aaaa6f9b9d59f41928d

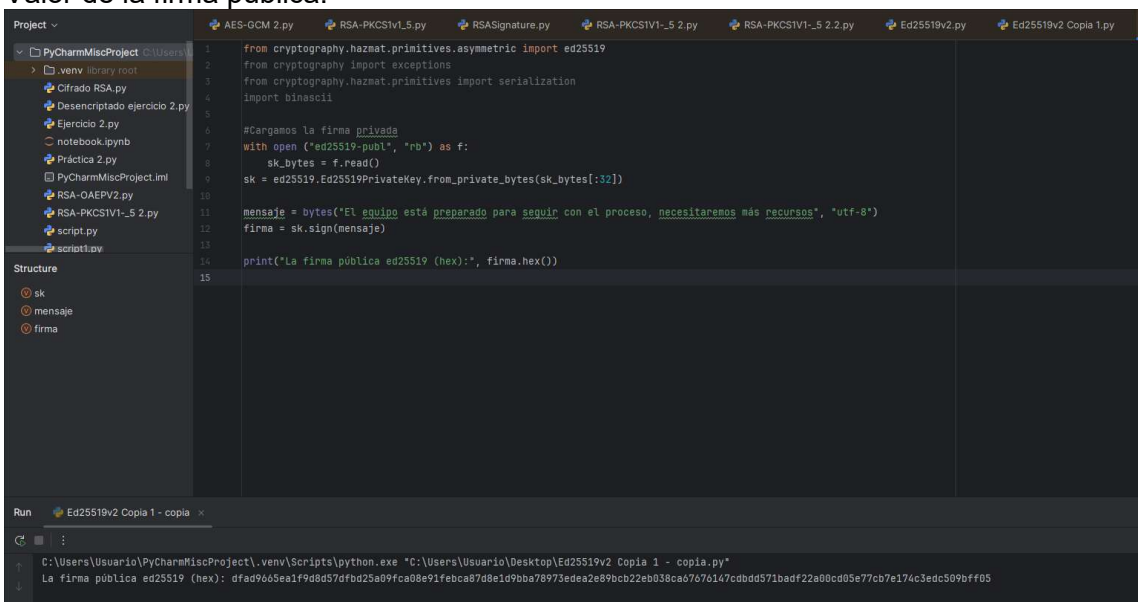


Calcula la firma (en hexadecimal) con la curva elíptica ed25519, usando las claves ed25519priv y ed25519-publ.

Valor de la firma privada:



Valor de la firma pública:



14. Decimocuarto

Necesitamos generar una nueva clave AES, usando para ello una HKDF (HMACbased 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:

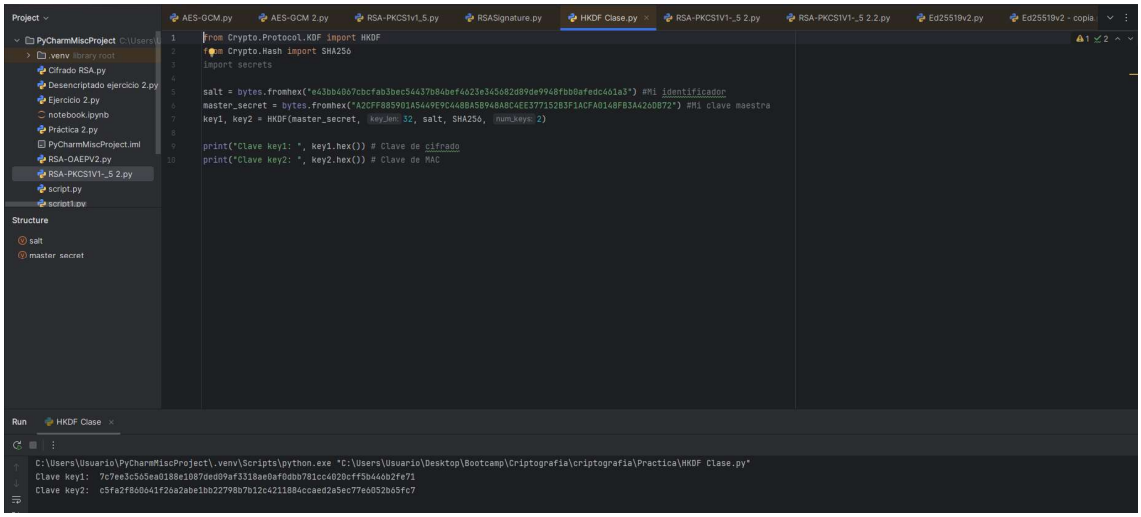
e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3

¿Qué clave se ha obtenido?

Se obtienen los valores de “Clave 1” y “Clave 2”, siendo la primera la clave de cifrado y la segunda la clave del ‘MAC’:

Clave key1:
7c7ee3c565ea0188e1087ded09af3318ae0af0dbb781cc4020cff5b446b2fe71

Clave key2:
c5fa2f860641f26a2abe1bb22798b7b12c4211884ccaed2a5ec77e6052b65fc7



15. Decimoquinto

Nos envían un bloque TR31:

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515
ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E
5657495E03CD857FD37018E111B

MASTER KEY
HKDF - SHA512 CLAVE SALT DIVERSIFICADA

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

A1A10101010101010101010101010102

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

- Respuesta: El algoritmo se ha protegido mediante el método de derivación de clave AES.

¿Para qué algoritmo se ha definido la clave?

- Respuesta: La clave se ha definido para un algoritmo AES.

¿Para qué modo de uso se ha generado?

- Respuesta: Se ha generado para el modo de uso “B”, que se refiere a que sirve tanto como para cifrado como para descifrado.

¿Es exportable?

- Respuesta: SI, pero bajo una clave no confiable.

¿Para qué se puede usar la clave?

- Respuesta: La clave se puede usar para el cifrado de datos, es genérica.

¿Qué valor tiene la clave?

- Respuesta: La clave importada tiene un valor de “C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1” en Hexadecimal.

```python
from psec import tr31


def importar():
    """Importa y muestra información de un Key Block TR-31"""
    kbpk_b = bytes.fromhex("A1A10101010101010101010101010102")
    kb_string = "D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965

    # Crear KeyBlock con el KBPK
    kb = tr31.KeyBlock(kbpk_b)

    # Cargar el header desde el Key Block string
    kb.header.load(kb_string)

    # Unwrappear la clave
    clave_unwrapped = kb.unwrap(kb_string)

    print("=" * 60)
    print("INFORMACIÓN DEL KEY BLOCK TR-31")
    print("=" * 60)
    print(f"Clave importada (hex):  {clave_unwrapped.hex().upper()}")
    print(f"Versión:                {kb.header.version_id}")
    print(f"Uso de clave:           {kb.header.key_usage}")
    print(f"Algoritmo:              {kb.header.algorithm}")
    print(f"Modo de uso:            {kb.header.mode_of_use}")
    print(f"Exportabilidad:         {kb.header.exportability}")
    print(f"Número de versión:      {kb.header.version_num}")
    print("=" * 60)
```

Código para la obtención de datos del bloque TR31

```
C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:\Users\Usuario\Desktop\Bootcamp\Criptografia\criptografia\codigo fuente\Gestión de Claves\Impo
============================================================
INFORMACIÓN DEL KEY BLOCK TR-31
============================================================
Clave importada (hex):  C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1
Versión:                D
Uso de clave:           D0
Algoritmo:              A
Modo de uso:            B
Exportabilidad:         S
Número de versión:      00
============================================================
============================================================
EXPORTACIÓN TR-31
============================================================
Key Block: D0144C0AB00S0000065B7E137592153E7555B018E7E16B79A4797F110F288EA3951C69850FBB64C16FBE30EBA31A741EC5CDA42806CC56A04CAD8E35AAD2209867A963E1D3C509BD
============================================================

Process finished with exit code 0
```

Respuesta de Python del bloque TR31.

Bloque de teclas

```
D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDB
E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E0
3CD857FD37018E111B
```

Descodificar

Encabezado (16 bytes)

Compensar	Campo	Valor	Significado
0	ID de versión	D	Bloque de clave TR-31 protegido mediante el método de derivación de clave AES
1-4	Longitud del bloque clave	0144	Longitud total del bloque de clave
5-6	Uso de la clave	D0	Clave de cifrado de datos (genérica)
7	Algoritmo	A	AES
8	Modo de uso	B	Tanto el cifrado como el descifrado
9-10	Número de versión de clave	00	No se utiliza el control de versiones de clave para esta clave
11	Exportabilidad	S	Sensible, exportable bajo una clave no confiable
12-13	Número de bloques opcionales	00	No hay bloques opcionales
14-15	Reservado para uso futuro	00	

Datos de clave cifrados (120 bytes)

```
42766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37
```

Datos de la nomenclatura del bloque TR31 obtenidos mediante la página web https://paymentcardtools.com/key-block .