

# 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 llenará 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/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nnvJbBiG+le1ZhpR84ol=”

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.

```
1 import jks
2 import os
3
4 # Obteniendo el path
5 path = os.path.dirname(__file__)
6
7 keystore = path + "/KeyStorePracticas"
8
9
10 ks = jks.KeyStore.load(keystore, store_password: "123456")
11
12 for alias, sk in ks.secret_keys.items():
13     if sk.alias == "cifrado-sim-aes-256":
14         key = sk.key
15
16 print("La clave es:", key.hex())
17
```

C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:\Users\Usuario\Desktop\Bootcamp\Criptografia\criptografia 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.

The screenshot shows the CyberChef interface with two tabs: 'From Base64' and 'AES Decrypt'. In the 'From Base64' tab, a long hex string is pasted into the input field. In the 'AES Decrypt' tab, the 'Key' field contains 'A2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72', the 'Mode' field is set to 'CBC', and the 'Input' field is set to 'Raw'. The 'Output' field shows the decrypted data in raw hex format. A note at the bottom states: 'Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.'

Obtención del desencriptado en CyberChef.

```
1 import json
2 from base64 import b64encode, b64decode
3 from Crypto.Cipher import AES
4 from Crypto.Util.Padding import pad, unpad
5
6 clave = bytes.fromhex('a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72')
7 iv_bytes = bytes.fromhex('00000000000000000000000000000000')
8 texto_cifrado_bytes = b64decode('TQ9S0MKc6aFS9SlxfK9wT18UXpPCd505XF5J/5nLI70f/o0QKIWXg3nu1RRz4QWElezdrLAD5L04US
t3aB/i50nvvJbBi6+le1ZhpR84oI=')
9
10 cipher = AES.new(clave, AES.MODE_CBC, iv_bytes)
11 texto_plano_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size)
12
13 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).

```
1 import json
2 from base64 import b64encode, b64decode
3 from Crypto.Cipher import AES
4 from Crypto.Util.Padding import pad, unpad
5
6 clave = bytes.fromhex('a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72')
7 iv_bytes = bytes.fromhex('00000000000000000000000000000000')
8 texto_cifrado_bytes = b64decode('TQ9S0MKc6aFS95lxhfK9wT18UXpPCd505Xf5J/5nLI70f/o00KIWXg3nu1RRz4QWElezdrLAD5L04UST3aB/i50nvvJbBi6+le1ZhpR84oI=')
9
10 cipher = AES.new(clave, AES.MODE_CBC, iv_bytes)
11 texto_plano_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size)
12
13 print(texto_plano_bytes.decode(encoding='utf-8', errors='ignore'))
14
15 mensaje_desc_limpio = unpad(texto_plano_bytes, AES.block_size, style="pkcs7")
16
17 print(mensaje_desc_limpio.hex())
18 print(mensaje_desc_limpio.decode("utf-8"))
19
20 import json
21 from base64 import b64encode, b64decode
22 from Crypto.Cipher import AES
23 from Crypto.Util.Padding import pad, unpad
24
25 clave = bytes.fromhex('a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72')
26 iv_bytes = bytes.fromhex('00000000000000000000000000000000')
27 texto_cifrado_bytes = b64decode('TQ9S0MKc6aFS95lxhfK9wT18UXpPCd505Xf5J/5nLI70f/o00KIWXg3nu1RRz4QWElezdrLAD5L04UST3aB/i50nvvJbBi6+le1ZhpR84oI=')
28
29 cipher = AES.new(clave, AES.MODE_CBC, iv_bytes)
30 texto_plano_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size)
31
32 print(texto_plano_bytes.decode(encoding='utf-8', errors='ignore'))
33
34 mensaje_desc_limpio = unpad(texto_plano_bytes, AES.block_size, style="x923")
35
36 print(mensaje_desc_limpio.hex())
37 print(mensaje_desc_limpio.decode("utf-8"))
38
39
C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:\Users\Usuario\PyCharmMiscProject\Desencriptado ejercicio 2.py"
Este 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 =  
69ac4ee7c4c552537a00a19bcraf7f0aaed7c9c8f769956a09bce6fadef6c3535f221  
1c9467067cf5c4a842ab
- Mensaje cifrado en B64 =  
aaxO58TFUIN6AKGbyvfwqu18nI92mVagm85vre9sNTXyIRyUZwZ89cSoQqs=

```
C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:/Users/Usuario/Desktop/Bootcamp/Criptografia/criptografia/codigo.py"
Mensaje cifrado en HEX = 69ac4ee7c4c552537a00a19bcraf7f0aaed7c9c8f769956a09bce6fadef6c3535f2211c9467067cf5c4a842ab
Mensaje cifrado en B64 = aaxO58TFUIN6AKGbyvfwqu18nI92mVagm85vre9sNTXyIRyUZwZ89cSoQqs=

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('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())
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:

```

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('AF9DF30474898787A45605CCB9B936D33B780003CABC817190523834800C3120')
#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 = +0Z2mXgSy2FxnaAjKG45Ed2FzpEZEv6z7REWbdV8ZCB1mRjwRVx0EKq1Fgo=
Process finished with exit code 0

```

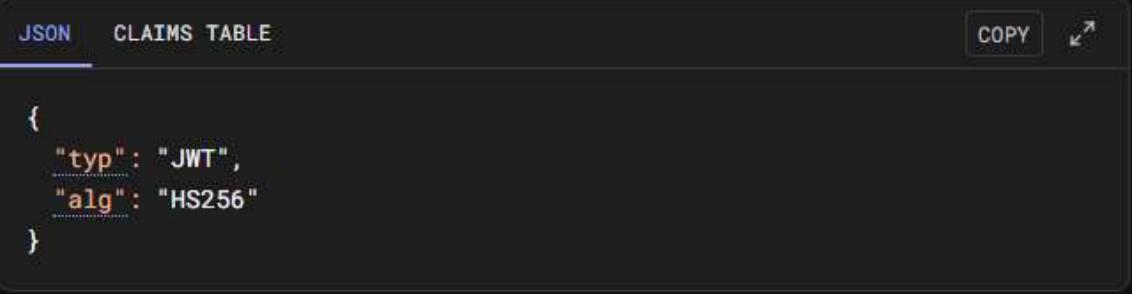
Código y solución con la implementación del ‘nonce’ aleatorio.

## 4. Cuarto ejercicio

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBIcGI0byBkZSBsb3MgcGFsb3RlcylsInJvbCI6ImlzTm9ybWFsliwiaWF0ljoxNjY3OTMzNMzfQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

¿Qué algoritmo de firma hemos realizado? Respuesta:



The screenshot shows a JSON viewer interface with two tabs: 'JSON' and 'CLAIMS TABLE'. The 'JSON' tab is selected, displaying the following JSON object:

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

There is a 'COPY' button in the top right corner of the JSON viewer.

¿Cuál es el body del jwt?

Respuesta: El body del jwt es la parte del código separada entre puntos ‘.’, por tanto corresponde a:

“eyJ1c3VhcmlvIjoiRG9uIFBIcGI0byBkZSBsb3MgcGFsb3RlcylsInJvbCI6ImlzTm9ybWFsliwiaWF0ljoxNjY3OTMzNMzfQ”

Un hacker está enviando a nuestro sistema el siguiente jwt:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGI0b  
yBkZSBsb3MgcGFsb3RlcylsInJvbCI6ImIzQWRtaW4iLCJpYXQiOjE2Njc5Mz  
M1MzN9.krgBkzCBQ5WZ8JnZHURvmnAZdg4ZMeRNv2CIAODIHRI

¿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 validarla 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”.

bced1be95fb85d2ffcce9c85434d79aa26f24ce82fb4439517ea3f072d56fe

¿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:

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6df836459551a1044f4f2908aa5d6  
3739506f6468833d77c07cf69c488823b8d858283f1d05877120e8c5351c83  
3

¿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 hash 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\Criptografía\HMAC\HMAC.py"  
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?

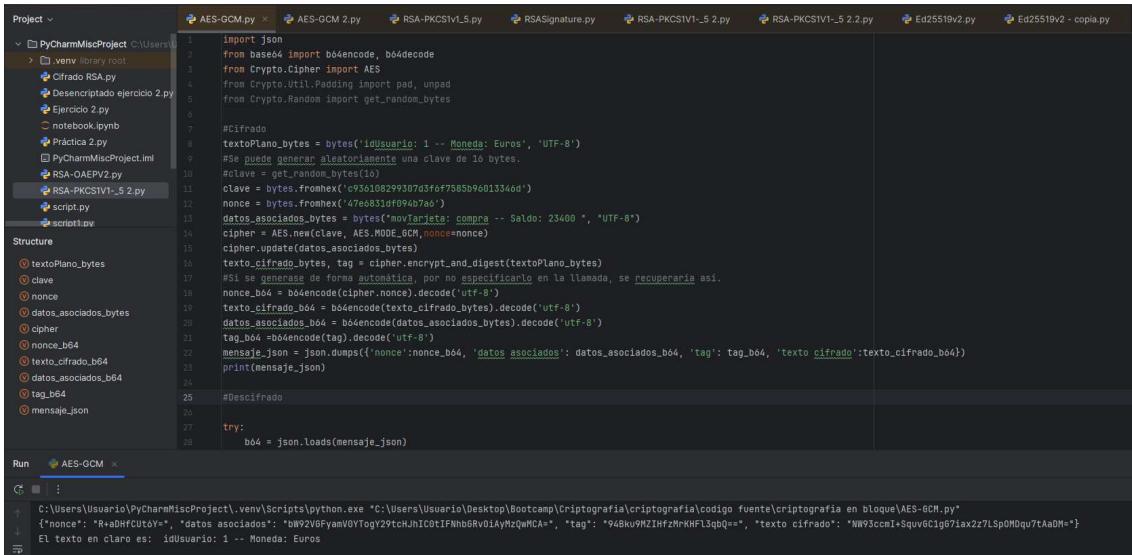
- Respuesta: La mejor opción es usar Argon2, que es un algoritmo moderno y muy seguro para proteger contraseñas. Argon2 está diseñado para ser resistente a ataques, también es configurable para ajustar su seguridad y es considerado el estándar actual para almacenamiento seguro de contraseñas.

## 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 *idUser* y *Moneda*.



```

Project ▾
  PyCharmMiscProject C:\Users\...
    venv library root
      Cifrado RSA.py
      Desencriptado ejercicio 2.py
      Ejercicio 2.py
      notebook.ipynb
      Práctica 2.py
      PyCharmMiscProject.iml
      RSA-OAEPV2.py
      RSA-PKCSIV1-5 2.py
      script.py
      scriptiov
Structure
  textoPlano_bytes
  clave
  nonce
  datos_asociados_bytes
  cipher
  nonce_b64
  texto_cifrado_b64
  datos_asociados_b64
  tag_b64
  mensaje_json
Run AES-GCM
C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:\Users\Usuario\Desktop\Bootcamp\Criptografia\criptografia fuente\criptografia en bloque\AES-GCM.py"
{"nonce": "R+ahFcUtb0=", "datos_asociados": "WNYV0FyamW0YlogZ9tchJni0tIbhnb6Rv0IayMzQwMCAs", "tag": "94Bku9MZIMfzMrKhFlsqbQ==", "texto_cifrado": "NW93ccmI+Squv6C1g6IaxZz7LSpOMDqu7tAaDM="}
El texto en claro es: idUsuario: 1 -- Moneda: Euros

```

## 9. Noveno

Se requiere calcular el KCV de la siguiente clave AES:

A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A42  
6DB72

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.

```

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.

The terminal window shows four commands being run in sequence to import keys from files:

```
C:\Users\Usuario\Desktop\GpgBootcamp>gpg --import Pedro-publi.txt
gpg: Cantidad total procesada: 1
gpg:     sin cambios: 1

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --import Pedro-priv.txt
gpg: clave D730BE196E466101: clave pública "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>" importada
gpg: Cantidad total procesada: 1
gpg:     importadas: 1

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --import RRHH-priv.txt
gpg: clave 3869803C684D287B: "RRHH <RRHH@RRHH>" 1 firma nueva
gpg: clave 3869803C684D287B: clave secreta importada
gpg: Cantidad total procesada: 1
gpg:     nuevas firmas: 1
gpg:         claves secretas leídas: 1
gpg:         claves secretas importadas: 1

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --import Pedro-publi.txt
gpg: clave D730BE196E466101: "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>" sin cambios
gpg: Cantidad total procesada: 1
gpg:     sin cambios: 1
```

The Kleopatra interface shows a list of certificates:

Nombre	Correo electrónico	Estado	Válido desde	Válido hasta	ID de la clave
Daniel Landeira (Ejercicio 10 Practica)	practicakeepcoding@gmail.com	certificado	09/12/2025		72C6 66FC A334 1B03
Daniel Landeira (segunda clave practica)	practica2keepcoding@gmail.com	certificado	09/12/2025		9CD8 6512 213C 2C5B
Daniel Landeira (tercera clave practica)	practica3keepcoding@gmail.com	certificado	09/12/2025	09/12/2027	F8B1 5D84 800F A134
Pedro Pedrito Pedro	pedro.pedrito.pedro@empresa.com	no certif...	26/06/2022	07/12/2029	D730 BE19 6E46 6101
RRHH	RRHH@RRHH	certificado	26/06/2022	07/12/2029	3869 803C 684D 287B

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

--output MensajeRespoDeRaulARRHH.txt --decrypt  
MensajeRespoDeRaulARRHH.sig"

```
C:\Users\Usuario\Desktop\GpgBootcamp>gpg --verify MensajeRespoDeRaulARRHH.sig Pedro-publi.txt
gpg: no es una firma separada

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --verify MensajeRespoDeRaulARRHH.sig RRHH-publi.txt
gpg: no es una firma separada

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --verify MensajeRespoDeRaulARRHH.sig RRHH-priv.txt
gpg: no es una firma separada

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --verify MensajeRespoDeRaulARRHH.txt RRHH-publi.txt
gpg: no se han encontrados datos OpenPGP válidos
gpg: la firma no se pudo verificar.
Por favor recuerde que el fichero de firma (.sig o .asc)
debería ser el primero que se da en la línea de órdenes.

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --verify MensajeRespoDeRaulARRHH.sig RRHH-publi.txt
gpg: no es una firma separada

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --output MensajeRespoDeRaulARRHH.txt --decrypt MensajeRespoDeRaulARRHH.sig
El fichero "MensajeRespoDeRaulARRHH.txt" ya existe. ¿Sobreescribir? (s/N) s
gpg: Firmado el 06/26/22 13:47:01 Hora de verano romance
gpg:           usando EDDSA clave 1BDE635E4AE6E68DFAD2F7CD730BE196E466101
gpg:           emisor "pedro.pedrito.pedro@empresa.com"
gpg: Firma correcta de "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>" [desconocido]
gpg: WARNING: The key's User ID is not certified with a trusted signature!
gpg:           No hay indicios de que la firma pertenezca al propietario.
Huellas dactilares de la clave primaria: 1BDE 635E 4AE6 E688 DFAD 2F7C D730 BE19 6E46 6101

C:\Users\Usuario\Desktop\GpgBootcamp>
```

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".

```
RPG: Clave 3869803C684D287B: "RRHH <RRHH@RRHH>" firma nueva
RPG: Clave 3869803C684D287B: clave secreta importada
gpg: Cantidad total procesada: 1
gpg:     nuevas firmas: 1
gpg:     claves secretas leídas: 1
gpg:     claves secretas importadas: 1

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --import Pedro-publi.txt
gpg: clave D730BE196E466101: "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>" sin cambios
gpg: Cantidad total procesada: 1
gpg:     sin cambios: 1

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --verify Pedro-publi.txt MensajeRespoDeRaulARRHH.txt
gpg: verify signatures failed: Error inesperado

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --verify RRHH-publi.txt MensajeRespoDeRaulARRHH.txt
gpg: verify signatures failed: Error inesperado

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --verify RRHH-priv.txt MensajeRespoDeRaulARRHH.txt
gpg: verify signatures failed: Error inesperado

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --verify Pedro-priv.txt MensajeRespoDeRaulARRHH.txt
gpg: verify signatures failed: Error inesperado

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --output RRHH-priv.txt --clearsign -u 3869803C684D287B Texto-a-firmar.txt
El fichero "RRHH-priv.txt" ya existe. ¿Sobreescribir? (s/N) n
Introduzca nuevo nombre de fichero: "X"
C:\Users\Usuario\Desktop\GpgBootcamp>gpg --output Texto-firmado.txt --clearsign -u 3869803C684D287B RRHH-priv.txt

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --output Texto-cifrado.gpg --encrypt --recipient 3869803C684D287B Texto-a-cifrar.txt
El fichero "RRHH-priv.txt" ya existe. ¿Sobreescribir? (s/N) n
Introduzca nuevo nombre de fichero: "X"
C:\Users\Usuario\Desktop\GpgBootcamp>gpg --output Texto-firmado.gpg --encrypt --recipient 3869803C684D287B --recipient D730BE196E466101 Texto-a-cifrar.txt
gpg: 25D6D02940358650: No hay seguridad de que esta clave pertenezca realmente
al usuario que se nombra
sub c72519/25D6D02940358650 2022-06-26 Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>
    Huella de subclave: 8ERC 6669 AC44 3211 42BC C244 25D6 0029 4035 8650

No es seguro que la clave pertenezca a la persona que se nombra en el
identificador de usuario. Si *realmente* sabe lo que está haciendo,
puede contestar s a la siguiente pregunta.

¿Usa esta clave de todas formas? (s/N) s
C:\Users\Usuario\Desktop\GpgBootcamp>
```

Vista	Nombre	Fecha de modificación	Tipo	Tamaño
	MensajeRespoDeRaulARRHH.sig	27/11/2025 14:37	OpenPGP Signature	1 KB
	MensajeRespoDeRaulARRHH.txt	27/11/2025 14:37	Documento de te...	1 KB
	Pedro-priv.txt	27/11/2025 14:37	Documento de te...	1 KB
	Pedro-publi.txt	09/12/2025 15:56	Documento de te...	1 KB
	RRHH-priv.txt	27/11/2025 14:37	Documento de te...	1 KB
	RRHH-publi.txt	09/12/2025 15:56	Documento de te...	1 KB
	Texto-a-cifrar.txt	11/12/2025 15:25	Documento de te...	1 KB
	Texto-a-firmar.txt	11/12/2025 15:24	Documento de te...	1 KB
	Texto-firmado.txt	11/12/2025 16:50	Documento de te...	2 KB

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".

```
C:\Users\Usuario\Desktop\GpgBootcamp>gpg --verify RRHH-publi.txt MensajeRespoDeRaulARRHH.txt
gpg: verify signatures failed: Error inesperado

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --verify RRHH-priv.txt MensajeRespoDeRaulARRHH.txt
gpg: verify signatures failed: Error inesperado

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --verify Pedro-priv.txt MensajeRespoDeRaulARRHH.txt
gpg: verify signatures failed: Error inesperado

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --output RRHH-priv.txt --clearsign -u 3869803C684D287B Texto-a-cifrar.txt
El fichero "RRHH-priv.txt" ya existe. ¿Sobreescribir? (s/N) n
Introduzca nuevo nombre de fichero: "X"
C:\Users\Usuario\Desktop\GpgBootcamp>gpg --output Texto-firmado.txt --clearsign -u 3869803C684D287B RRHH-priv.txt

C:\Users\Usuario\Desktop\GpgBootcamp>gpg --output Texto-cifrado.gpg --encrypt --recipient 3869803C684D287B --recipient D730BE196E466101 Texto-a-cifrar.txt
gpg: 25D6D02940358650: No hay seguridad de que esta clave pertenezca realmente
al usuario que se nombra
sub c72519/25D6D02940358650 2022-06-26 Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>
    Huella de subclave: 8ERC 6669 AC44 3211 42BC C244 25D6 0029 4035 8650

No es seguro que la clave pertenezca a la persona que se nombra en el
identificador de usuario. Si *realmente* sabe lo que está haciendo,
puede contestar s a la siguiente pregunta.

¿Usa esta clave de todas formas? (s/N) s
C:\Users\Usuario\Desktop\GpgBootcamp>
```

Vista	Nombre	Fecha de modificación	Tipo	Tamaño
	MensajeRespoDeRaulARRHH.sig	27/11/2025 14:37	OpenPGP Signature	1 KB
	MensajeRespoDeRaulARRHH.txt	27/11/2025 14:37	Documento de te...	1 KB
	Pedro-priv.txt	27/11/2025 14:37	Documento de te...	1 KB
	Pedro-publi.txt	09/12/2025 15:56	Documento de te...	1 KB
	RRHH-priv.txt	27/11/2025 14:37	Documento de te...	1 KB
	RRHH-publi.txt	09/12/2025 15:56	Documento de te...	1 KB
	Texto-a-cifrar.txt	11/12/2025 15:25	Documento de te...	1 KB
	Texto-a-firmar.txt	11/12/2025 15:24	Documento de te...	1 KB
	Texto-cifrado.gpg	11/12/2025 16:55	OpenPGP Binary F...	1 KB
	Texto-firmado.txt	11/12/2025 16:50	Documento de te...	2 KB

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:

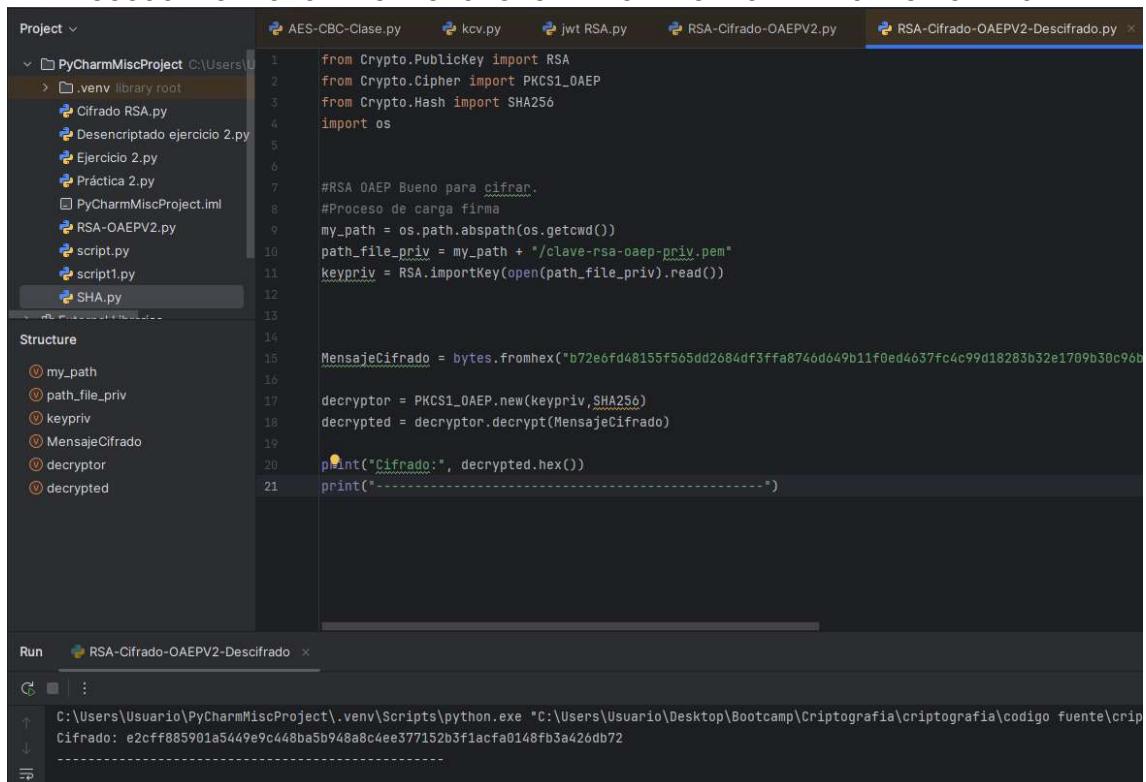
```
b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709  
b30c96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dff76a329d04e3d  
3d4ad629793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae2c4f5b  
a670a4126f2f14a9f4b6f41aa2edba01b4bd586624659fc82f5b4970186502de8  
624071be78cce573d896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f6  
7d3f98b4cd907f27639f4b1df3c50e05d5bf63768088226e2a9177485c54f72407f  
df358fe64479677d8296ad38c6f177ea7cb74927651cf24b01dee27895d4f05fb5c  
161957845cd1b5848ed64ed3b03722b21a526a6e447cb8ee
```

Las claves pública y privada las tenemos en los ficheros clave-rsa-oeap-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"
```



```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Hash import SHA256
import os

#RSA OAEP Bueno para cifrar.
#Proceso de carga firma
my_path = os.path.abspath(os.getcwd())
path_file_priv = my_path + "/clave-rsa-oeap-priv.pem"
keypriv = RSA.importKey(open(path_file_priv).read())

MensajeCifrado = bytes.fromhex("b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c96b  
624071be78cce573d896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f6  
7d3f98b4cd907f27639f4b1df3c50e05d5bf63768088226e2a9177485c54f72407f  
df358fe64479677d8296ad38c6f177ea7cb74927651cf24b01dee27895d4f05fb5c  
161957845cd1b5848ed64ed3b03722b21a526a6e447cb8ee")

decryptor = PKCS1_OAEP.new(keypriv, SHA256)
decrypted = decryptor.decrypt(MensajeCifrado)

print("Cifrado:", decrypted.hex())
print("-----")
```

Código de descifrado de la clave.

La clave re-cifrada es:

```
"40fdc8f1351f79e1d119a49d635bfc81e7ff2fd1dc8326b90093510c8de1751f42e  
65dcf9d1044b5c02e8279896a734534b5539e9aa65a311104433f045bfc13bee2  
195adc4e96156f4c1c1b8b283981eb889fe084e9a25e1a47c690c5d6958133b09  
0cfea8fe9a00d5cbea56e13a34a2e76d5316500dfb29ebb5154283a4ed7ee720f  
9a2435ec2ed18bacc3ed6c0c4123f668bd3bb832af0ed486f40e14d7df0e620825
```

17c93a815f6a5780fb583c5afeb1b71533be63a0a6fb6945b803c53e54c29c9e61  
e66b10b44bd1ea040ed167b2b168e8b61461a8bae02bf5d7389134e0b217b389  
3fbf7ebfb729e9d4335d2be72507382273c892ec2953a0461925bd".

The screenshot shows the PyCharm IDE interface. The top navigation bar has tabs for AES-CBC-Close.py, kcv.py, jwt RSA.py, RSA-Cifrado-OAEPV2.py (which is the active tab), and RSA-Cifrado-OAEPV2-Descifrado.py. The left sidebar shows a project structure for 'PyCharmMiscProject' with files like RSA.py, SHA.py, and various exercise scripts. The main code editor displays the following Python script:

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Hash import SHA256
import os

#RSA OAEP Bueno para cifrar. Cambia en cada ejecución.
#Proceso de carga pública
my_path = os.path.abspath(os.getcwd())
fichero_pub = my_path + "/clave-rsa-oaep-publ.pem"
f=open(fichero_pub,'r')
keypub= RSA.import_key(f.read())

MensajeClaveCompartida = bytes.fromhex("e2cff885901a5449e9c448ba5b948a8c4ee377152b5f1acfa0148fb3a426db72")

encryptor = PKCS1_OAEP.new(keypub,SHA256)
encrypted = encryptor.encrypt(MensajeClaveCompartida)

print("cifrado:", encrypted.hex())
print("-----")
```

The bottom panel shows the terminal window with the command 'python RSA-Cifrado-OAEPV2.py' run, displaying the encrypted hex string: 'Cifrado: 40fdcc8f1351f79e1d119a49d635bfc81e7ff2fd1dc8320b90093510c8de1751f42e65dcf9d1044b5c02e8279896a734534b5539e9aa65a311104433f045bfc13bee2195adc4e96156f4c1c'. The message 'Process finished with exit code 0' is shown at the bottom.

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/QTWXiq92dhLum64MHC  
V9QePv8FiVt
- Hex:  
5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4af04d65e  
2abdd9d84bba6eb8307095f5078fbfc16256d

```
1  from Crypto.Cipher import AES
2  from base64 import b64encode, b64decode
3  from Crypto.Util.Padding import pad, unpad
4  from Crypto.Random import get_random_bytes
5
6  # AES-GCM --> (Datos Asociados + Datos a cifrar) + key + nonce
7
8  texto_gcm_bytes = bytes("He descubierto el error y no volveré a hacerlo mal", "utf-8")
9  key_bytes = bytes.fromhex('E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74')
10 nonce_bytes = b64decode('9Yccnf5nJJhAt2S')
11 #print("Nonce hex=", nonce_bytes(base64))
12 datos_asociados_bytes = bytes("Id usuario", "utf-8")
13 cifrador = AES.new(key_bytes,AES.MODE_GCM,nonce=nonce_bytes)
14 cifrador.update(datos_asociados_bytes)
15 texto_cifrado_bytes,mac_bytes = cifrador.encrypt_and_digest(texto_gcm_bytes)
16 print("Texto cifrado:", b64encode(texto_cifrado_bytes).decode('utf-8'))
17 print("MAC:", mac_bytes.hex())
18
19 |
```

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-C
1  from Crypto.Cipher import AES
2  from base64 import b64encode, b64decode
3  from Crypto.Util.Padding import pad, unpad
4  from Crypto.Random import get_random_bytes
5
6  # AES-GCM --> (Datos Asociados + Datos a cifrar) + key + nonce
7
8  texto_gcm_bytes = bytes("He descubierto el error y no volveré a hacerlo mal", "utf-8")
9  key_bytes = bytes.fromhex('E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74')
10 nonce_bytes = b64decode('9Yccnf5nJJhAt2S')
11 #print("Nonce hex=", nonce_bytes(base64))
12 datos_asociados_bytes = bytes("Id usuario", "utf-8")
13 cifrador = AES.new(key_bytes,AES.MODE_GCM,nonce=nonce_bytes)
14 cifrador.update(datos_asociados_bytes)
15 texto_cifrado_bytes,mac_bytes = cifrador.encrypt_and_digest(texto_gcm_bytes)
16 print("Texto cifrado:", texto_cifrado_bytes.hex())
17 print("MAC:", mac_bytes.hex())
18
19 |
```

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:**

a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c  
6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998e  
f08b2cb3a925c959bc当地2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70  
450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad  
0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8954ba3  
9dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d929  
78a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21f0aaa  
a6f9b9d59f41928d

```

Project: PyCharmMiscProject
File: RSA-PKCS1V1-5_2.py
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 # Cargamos clave Privada
7 my_path = os.path.abspath(os.getcwd())
8 path_file_priv = my_path + '/clave-rsa-oaep-priv.pem'
9 keypriv = RSA.importKey(open(path_file_priv).read())
10
11 # Mensaje a firmar
12 mensaje_bytes = bytes("El equipo está preparado para seguir con el proceso, necesitaremos más recursos.", "utf-8")
13
14 # Calculamos el hash del mensaje con SHA-256
15 hash = SHA256.new(mensaje_bytes)
16
17 # Generamos el firmador con PKCS#1 v1.5
18 firmador = PKCS115_SigScheme(keypriv)
19
20 # Firmamos el hash
21 firma = firmador.sign(hash)
22
23 # Resultado de firma
24 print("Firma: ", firma.hex())

```

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

Valor de la firma privada:

```

Project: PyCharmMiscProject
File: Ed25519v2.py
1 from cryptography.hazmat.primitives.asymmetric import ed25519
2 from cryptography import exceptions
3 from cryptography.hazmat.primitives.serialization import binascii
4
5 #Cargamos la firma privada
6 with open ("ed25519-priv", "rb") as f:
7     sk_bytes = f.read()
8 sk = ed25519.Ed25519PrivateKey.from_private_bytes(sk_bytes[:32])
9
10 mensaje_bytes = bytes("El equipo está preparado para seguir con el proceso, necesitaremos más recursos.", "utf-8")
11 firma = sk.sign(mensaje)
12
13 print("La firma privada ed25519 (hex):", firma.hex())
14
15

```

## Valor de la firma pública:

The screenshot shows a PyCharm interface with a project named 'PyCharmMiscProject'. The code editor displays a Python script named 'script1.py' which contains the following code:

```
from cryptography.hazmat.primitives.asymmetric import ed25519
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.serialization import PublicFormat
import binascii

#Cargamos la firma privada
with open ("ed25519-publ", "rb") as f:
    sk_bytes = f.read()
sk = ed25519.Ed25519PrivateKey.from_private_bytes(sk_bytes[:32])

mensaje = bytes("El equipo está preparado para seguir con el proceso, necesitaremos más recursos", "utf-8")
firma = sk.sign(mensaje)

print("La firma pública ed25519 (hex):", firma.hex())
```

The run output window shows the command run and the resulting output:

```
C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:\Users\Usuario\Desktop\Ed25519v2 Copia 1 - copia.py"
La firma pública ed25519 (hex): dfad9605ea1f9d8d57fdbd25a09fc08e91febc8a78973edea2e89cb22eb038ca67676147cdbdd571badf22a00cd05e77cb7e174c3edc509bfff05
```

## 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:

e43bb4067cbc fab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3

¿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

The screenshot shows a PyCharm interface with a project named 'PyCharmMiscProject'. The code editor displays a Python script named 'HKDF Clase.py' which contains the following code:

```
from Crypto.Protocol.KDF import HKDF
from Crypto.Hash import SHA256
import secrets

salt = bytes.fromhex("e43bb4067cbc fab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3") #Mi identificador
master_secret = bytes.fromhex("A2CF885901A5449E9C448BAS948ABC4EE3771523F1ACFA0148FB3A420B072") #Mi clave maestra
key1, key2 = HKDF(master_secret, keylen=32, salt=salt, numkeys=2)

print("Clave key1: ", key1.hex()) # Clave de cifrado
print("Clave key2: ", key2.hex()) # Clave de MAC
```

The run output window shows the command run and the resulting output:

```
C:\Users\Usuario\PyCharmMiscProject\.venv\Scripts\python.exe "C:\Users\Usuario\Desktop\Bootcamp\Criptografia\criptografia\Practica\HKDF Clase.py"
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: Sí, 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 “C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1” en Hexadecimal.

```
from psec import tr31

def importar():
    """Importa y muestra información de un Key Block TR-31"""
    kb_pk_b = bytes.fromhex("A1A1010101010101010101010102")
    kb_string = "D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965

    # Crear KeyBlock con el KBPK
    kb = tr31.KeyBlock(kb_pk_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\Importación de clave TR-31.py"
=====
INFORMACIÓN DEL KEY BLOCK TR-31
=====
Clave importada (hex): C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1
Versión: D
Uso de clave: D0
Algoritmo: A
Modo de uso: B
Exportabilidad: S
Número de versión: 00
=====
EXPORTACIÓN TR-31
=====
Key Block: D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDB
E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37
3CD857FD37018E111B
=====

Process finished with exit code 0
```

## Respuesta de Python del bloque TR31.

Bloque de teclas

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDB  
E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37  
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>.