PRÁCTICA CRIPTOGRAFÍA

ENRIQUE LÓPEZ PASCUAL

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?

La cifra proporcionada por el Key manager es *20553975c31055ed*, obtenida mediante una operación XOR en formato hexadecimal, que representa la combinación de ambas claves. Este valor lo hemos obtenido a traves del código que podemos apreciar en la captura de pantalla. La obtención se realizó al dividir el código de la clave fija *B1EF2ACFE2BAEEFF* entre la clave final en memoria *91BA13BA21AABB12*.

La clave utilizada en la memoria es *8653f75d31455c0*, obtenida mediante una operación XOR en formato hexadecimal como podemos apreciar en la captura de pantalla adjunta. Esta clave se derivó al dividir el código de la clave fija *B1EF2ACFE2BAEEFF* entre la clave de los archivos de propiedades *B98A15BA31AEBB3F*.

```python
#Clave fija en código
kfija_desarrollador = 0xB1EF2ACFE2BAEEFF
#Parte dinámica que modifica el fichero de propiedades
kdinamica_fichero = 0xB98A15BA31AEBB3F
#Clave final (en memoria)
kfinal_memoria = 0x91BA13BA21AABB12



#XOR entre Clave fija y Clave final en memoria
k_en_properties = (hex(kfija_desarrollador^kfinal_memoria))
k_en_memoria = (hex(kfija_desarrollador^kdinamica_fichero))
print("Clave en properties:", k_en_properties)
print("Clave en meoria:", k_en_memoria)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                    Code
[Running] python -u "c:\Users\Usuario\OneDrive\criptografia\tempCodeRunnerFile.python"
Clave en properties: 0x20553975c31055ed
Clave en meoria: 0x8653f75d31455c0

[Done] exited with code=0 in 0.094 seconds
```

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?
¿Qué ocurre si decidimos cambiar el padding a x923 en el descifrado?
¿Cuánto padding se ha añadido en el cifrado?

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

Cifrado-sim.aes-256:

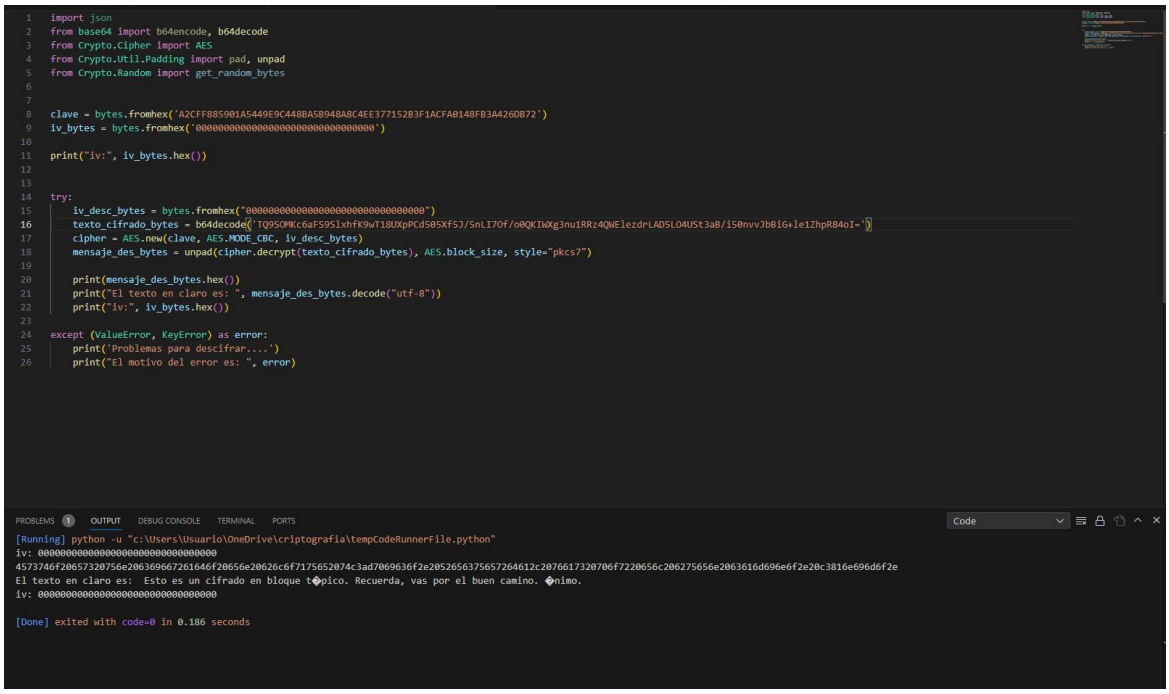
A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72

RESPUESTA 1.

Mensaje en claro: *“Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.”*

iv: 00000000000000000000000000000000

Mensaje descifrado quitando el padding añadido:
4573746f20657320756e206369667261646f20656e20626c6f7175652074c3ad7069636f2e20526563756572646 12c2076617320706f7220656c206275656e2063616d696e6f2e20c3816e696d6f2e.



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


clave = bytes.fromhex('A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72')
iv_bytes = bytes.fromhex('00000000000000000000000000000000')

print("iv:", iv_bytes.hex())


try:
    iv_desc_bytes = bytes.fromhex("00000000000000000000000000000000")
    texto_cifrado_bytes = b64decode('TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4U5t3aB/150nvvJbBiG+le1ZhpR84oI=')
    cipher = AES.new(clave, AES.MODE_CBC, iv_desc_bytes)
    mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size, style="pkcs7")

    print(mensaje_des_bytes.hex())
    print("El texto en claro es: ", mensaje_des_bytes.decode("utf-8"))
    print("iv:", iv_bytes.hex())

except (ValueError, KeyError) as error:
    print('Problemas para descifrar....')
    print("El motivo del error es: ", error)
```

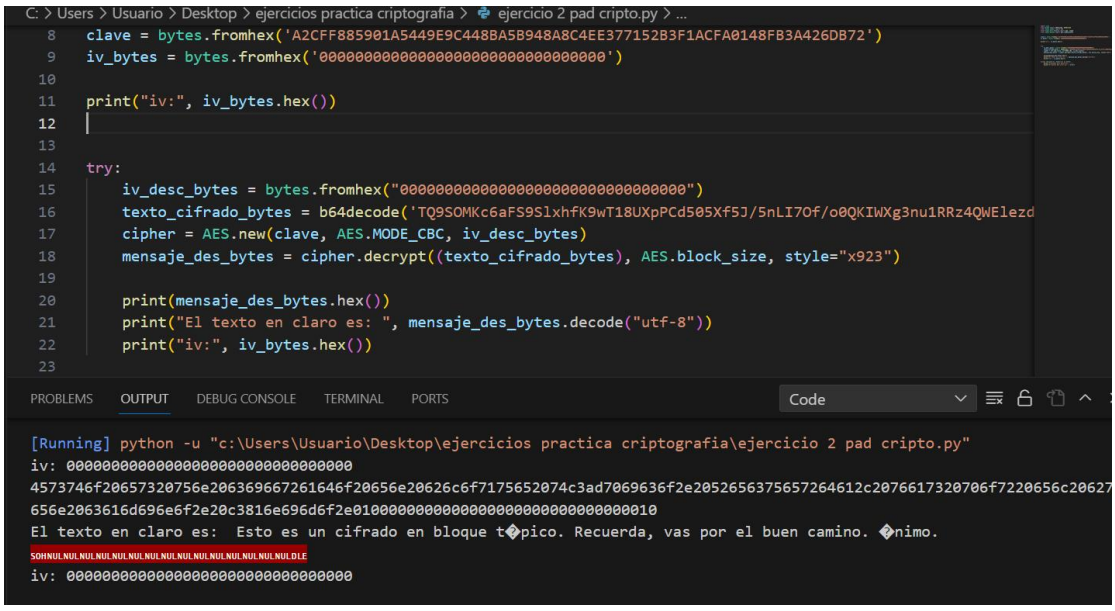
```
PROBLEMS 1  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                Code
[Running] python -u "c:\Users\Usuario\OneDrive\criptografia\tempCodeRunnerFile.python"
iv: 00000000000000000000000000000000
4573746f20657320756e20636966726164f20656e20626c6f7175652074c3ad7069636f2e2052656375657264612c207661732070f7220656c206275656e2063616d696e6f2e20c3816e696d6f2e
El texto en claro es:  Esto es un cifrado en bloque t◆pico. Recuerda, vas por el buen camino. ◆nimo.
iv: 00000000000000000000000000000000

[Done] exited with code=0 in 0.186 seconds
```

RESPUESTA 2:

Mensaje descifrado con padding:

4573746f20657320756e20636966726164f20656e20626c6f7175652074c3ad7069636f2e2052656375657264612c207661732070f7220656c206275656e2063616d696e6f2e20c3816e696d6f2e0100000000000000000000000000000010

Añade 10 bits en hexadecimal



```python
C: > Users > Usuario > Desktop > ejercicios practica criptografia > ejercicio 2 pad cripto.py > ...
clave = bytes.fromhex('A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72')
iv_bytes = bytes.fromhex('00000000000000000000000000000000')

print("iv:", iv_bytes.hex())


try:
    iv_desc_bytes = bytes.fromhex("00000000000000000000000000000000")
    texto_cifrado_bytes = b64decode('TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezd
    cipher = AES.new(clave, AES.MODE_CBC, iv_desc_bytes)
    mensaje_des_bytes = cipher.decrypt((texto_cifrado_bytes), AES.block_size, style="x923")

    print(mensaje_des_bytes.hex())
    print("El texto en claro es: ", mensaje_des_bytes.decode("utf-8"))
    print("iv:", iv_bytes.hex())
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                Code
[Running] python -u "c:\Users\Usuario\Desktop\ejercicios practica criptografia\ejercicio 2 pad cripto.py"
iv: 00000000000000000000000000000000
4573746f20657320756e20636966726164f20656e20626c6f7175652074c3ad7069636f2e2052656375657264612c207661732070f7220656c20627
656e2063616d696e6f2e20c3816e696d6f2e0100000000000000000000000000000010
El texto en claro es:  Esto es un cifrado en bloque t◆pico. Recuerda, vas por el buen camino. ◆nimo.
SOHNULNULNULNULNULNULNULNULNULNULNULNULNULNULDLE
iv: 00000000000000000000000000000000
```

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 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

nonce en HEX

f5871c9ff7f99c926102dd92

Mensaje cifrado en HEX

69ac4ee7c4c552537a00a19bcaf7f0aaed7c9c8f769956a09bce6fadef6c3535f2211c94
67067cf5c4a842ab

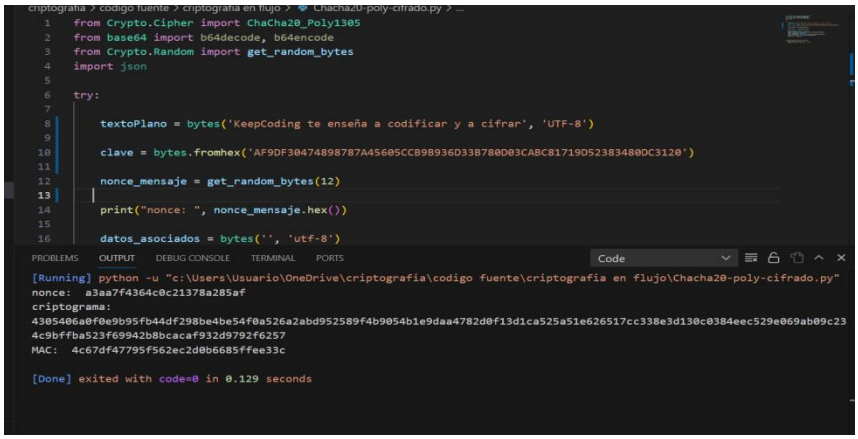
Mensaje cifrado en B64

aaxO58TFUlN6AKGbyvfwqu18nI92mVagm85vre9sNTXyIRyUZwZ89cSoQqs=

Mensaje en claro

KeepCoding te enseña a codificar y a cifrar

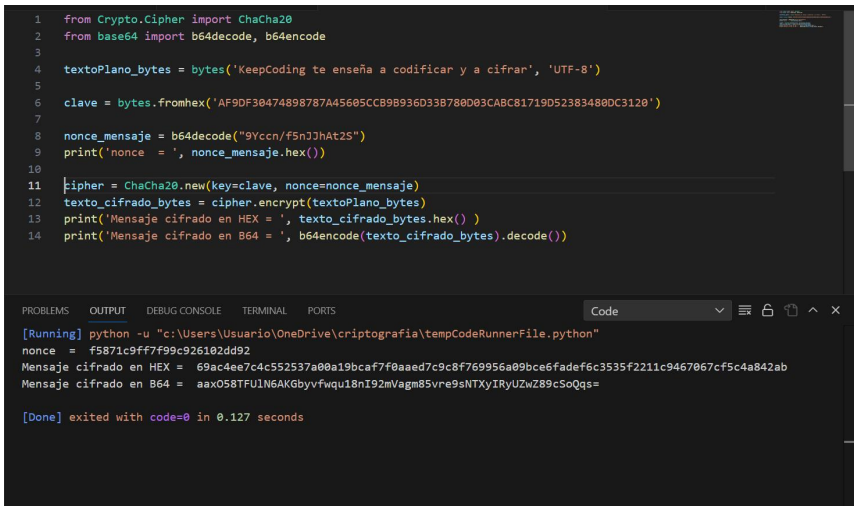
Para mejorar el sistema se debe generar un nonce aleatorio y aparte utilizar el algoritmo chacha-20.poly ya que es una versión mejorada que implementa un código de autentificación por lo que es más seguro.



```python
from Crypto.Cipher import ChaCha20_Poly1305
from base64 import b64decode, b64encode
from Crypto.Random import get_random_bytes
import json

try:

    textoPlano = bytes('KeepCoding te enseña a codificar y a cifrar', 'UTF-8')

    clave = bytes.fromhex('AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120')

    nonce_mensaje = get_random_bytes(12)
    print("nonce: ", nonce_mensaje.hex())

    datos_asociados = bytes('', 'utf-8')
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                                    Code

[Running] python -u "c:\Users\Usuario\OneDrive\criptografia\codigo fuente\criptografia en flujo\Chacha20-poly-cifrado.py"
nonce:  a3aa7f4364c0c21378a285af
criptograma:
4305406a0f0e9b95fb44df298be4be54f0a526a2abd952589f4b9054b1e9daa4782d0f13d1ca525a51e626517cc338e3d130c0384eec529e069ab09c23
4c9bffba523f69942b8bcacaf932d9792f6257
MAC:  4c67df47795f562ec2d0b6685ffee33c

[Done] exited with code=0 in 0.129 seconds
```

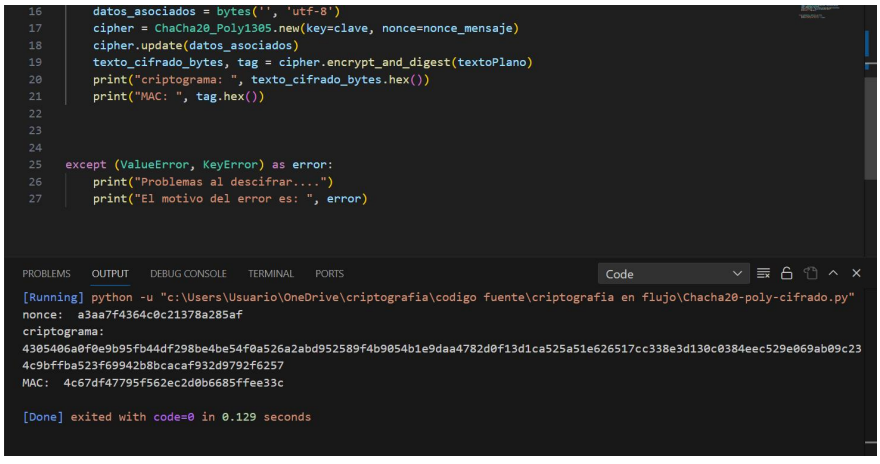
Captura de imagen x ejercicio x



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

textoPlano_bytes = bytes('KeepCoding te enseña a codificar y a cifrar', 'UTF-8')

clave = bytes.fromhex('AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120')

nonce_mensaje = b64decode("9Yccn/f5nJJhAt2S")
print('nonce  = ', nonce_mensaje.hex())

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())
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                                    Code

[Running] python -u "c:\Users\Usuario\OneDrive\criptografia\tempCodeRunnerFile.python"
nonce  =  f5871c9ff7f99c926102dd92
Mensaje cifrado en HEX =  69ac4ee7c4c552537a00a19bcaf7f0aaed7c9c8f769956a09bce6fadef6c3535f2211c9467067cf5c4a842ab
Mensaje cifrado en B64 =  aaxO58TFUlN6AKGbyvfwqu18nI92mVagm85vre9sNTXyIRyUZwZ89cSoQqs=

[Done] exited with code=0 in 0.127 seconds
```

Captura de imagen x ejercicio x



```python
    datos_asociados = bytes('', 'utf-8')
    cipher = ChaCha20_Poly1305.new(key=clave, nonce=nonce_mensaje)
    cipher.update(datos_asociados)
    texto_cifrado_bytes, tag = cipher.encrypt_and_digest(textoPlano)
    print("criptograma: ", texto_cifrado_bytes.hex())
    print("MAC: ", tag.hex())


except (ValueError, KeyError) as error:
    print("Problemas al descifrar....")
    print("El motivo del error es: ", error)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                                    Code

[Running] python -u "c:\Users\Usuario\OneDrive\criptografia\codigo fuente\criptografia en flujo\Chacha20-poly-cifrado.py"
nonce:  a3aa7f4364c0c21378a285af
criptograma:
4305406a0f0e9b95fb44df298be4be54f0a526a2abd952589f4b9054b1e9daa4782d0f13d1ca525a51e626517cc338e3d130c0384eec529e069ab09c23
4c9bffba523f69942b8bcacaf932d9792f6257
MAC:  4c67df47795f562ec2d0b6685ffee33c

[Done] exited with code=0 in 0.129 seconds
```

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ.gfhw0 dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

¿Qué algoritmo de firma hemos realizado?
¿Cuál es el body del jwt?

Un hacker está enviando a nuestro sistema el siguiente jwt:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBk
ZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9 .krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2ClAODlHRI

¿Qué está intentando realizar?
¿Qué ocurre si intentamos validarlo con pyjwt?

Respuesta 1:

Utilizamos el algoritmo H256

Respuesta 2:

Body del jwt:
{
"usuario": "Don Pepito de los palotes",
"rol": "isNormal",
"iat": 1667933533
}

Respuesta 3:

El hacker está modificando el rol del body de "isNormal" a "IsAdmin"

Respuesta 4:

Si intentamos validarlo nos da un código de verificación distinto.
[FRF] Debería de fallar.

Enlaces ejercicio 4:

Respuesta 1.

https://gchq.github.io/CyberChef/#recipe=JWT_Sign('Con%20KeepCoding%20aprendemos','HS256')&input=ew0KICAgICJ1c3VhcmlvIjogIkRvbiBQZXBpdG8gZGUgbG9zIHBhbG90ZXMiLA0KICAgICJyb2wiOiAiaXNOb3JtYWwiLA0KICAgICJpYXQiOiAxNjY3OTMzNTMzDQp9DQo

Respuesta 2.

https://gchq.github.io/CyberChef/#recipe=JWT_Decode()&input=ZXlKMGVYQWlPaUpLVjFRaUxDSmhiR2NpT2lKSVV6STFOaUo5LmV5SjFjM1ZoY21sdklqb2lSRzl1SUZCbGNHbDBieUJrWlNCc2IzTWdjR0ZzYjNSbGN5SXNJbkp2YkNJNkltbHpUbTl5YldGc0lpd2lhV0YwSWpveE5qWTNPVE16TlRNemZRLmdmaHcwZER4cDZvaXhNTFhYUlA5N1c0VERUcnYweTdCNVlqRDBVOGl4ckU

Respuesta 3.

https://gchq.github.io/CyberChef/#recipe=JWT_Decode()&input=ZXlKMGVYQWlPaUpLVjFRaUxDSmhiR2NpT2lKSVV6STFOaUo5LmV5SjFjM1ZoY21sdklqb2lSRzl1SUZCbGNHbDBieUJrWlNCc2IzTWdjR0ZzYjNSbGN5SXNJbkp2YkNJNkltbHpSV1J0YVc0aUxDSnBZWFFpT2pFMk5qYzVNek0xTXpOOS5rcmdCa3pDQlE1V1o4Sm5aSHVSdm1uQVpkZzRaTWVSTnYyQ0lBT0RsSFJJ

Respuesta 4.

https://gchq.github.io/CyberChef/#recipe=JWT_Verify('Con%20KeepCoding%20aprendemos')&input=ZXlKMGVYQWlPaUpLVjFRaUxDSmhiR2NpT2lKSVV6STFOaUo5LmV5SWlPaUpFYjI0Z1VHVndhWFJ2SUdSbElHeHZjeUJ3WVd4dmRHVnpJaXdpY205c0lqb2lhWE5CWkcxcGJpSjkua25WUG1qRHlHM3dvVTNPN194Z21TcUxUbHlHTDVTVEIzelptNmNIZ1A4cw

https://gchq.github.io/CyberChef/#recipe=JWT_Verify('Con%20KeepCoding%20aprendemos')&input=ZXlKMGVYQWlPaUpLVjFRaUxDSmhiR2NpT2lKSVV6STFOaUo5LmV5SWlPaUpFYjI0Z1VHVndhWFJ2SUdSbElHeHZjeUJ3WVd4dmRHVnpJaXdpY205c0lqb2lhWE5PYjNKdFlXd2lmUS42YXlLOFdKNFhiTUF6d3ctMnVpakY2S0Vrb3hBc082LXRNaEY1bkd2MG93

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?

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

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63 739506f 6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

¿Qué hash hemos realizado?

Genera ahora un SHA3 Keccak de 256 bits con el siguiente texto: “En KeepCoding aprendemos cómo protegernos con criptografía.” ¿Qué propiedad destacarías del hash, atendiendo a los resultados anteriores?

Respuesta 1.

Hemos generado un SHA3 256

```python
import hashlib


m = hashlib.sha3_224()
m.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía","utf8"))
print("sha3 - 224:    " + m.digest().hex())

m = hashlib.sha3_256()
m.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía","utf8"))
print("sha3 - 256:    " + m.digest().hex())
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                  Code
[Running] python -u "c:\Users\Usuario\OneDrive\criptografia\tempCodeRunnerFile.python"
sha3 - 224:    5f12ac6c044097c694bf740504679ef78e38a4a8fca86eb4ef9e05ae
sha3 - 256:    bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

[Done] exited with code=0 in 0.249 seconds
```

Respuesta 2.

Hemos generado un SHA2 512

```python
import hashlib


m = hashlib.sha256()
m.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía", "utf8"))
print("SHA256: " + m.digest().hex())

m = hashlib.sha256()
m.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía", "utf8"))
print("SHA256: " + m.digest().hex())

m = hashlib.sha512()
m.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía", "utf8"))
print("SHA512: " + m.digest().hex())

```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                  Code

[Running] python -u "c:\Users\Usuario\OneDrive\criptografia\tempCodeRunnerFile.python"
SHA256: 13067f558aed141a490bf95775e0c6fc583a09178ae7a0fefe93a8336be81237
SHA256: 13067f558aed141a490bf95775e0c6fc583a09178ae7a0fefe93a8336be81237
SHA512:
4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c53
51c833

[Done] exited with code=0 in 0.262 seconds
```

Respuesta 3.

302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

Observamos que la longitud del algoritmo es similar al primero de SHA3, pero el resultado es diferente. Esto se debe a que la frase utilizada para calcular este último contiene un punto final, lo que genera una variación en el resultado. En comparación con SHA-2, notamos que la longitud varía aún más, ya que se trata de un código con más bytes.

```python
import hashlib

s = hashlib.sha3_256()


print(s.name)

print(s.digest_size)

s.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía.","UTF-8"))

print(s.hexdigest())
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                  Code

[Running] python -u "c:\Users\Usuario\OneDrive\criptografia\tempCodeRunnerFile.python"
sha3_256
32
302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

[Done] exited with code=0 in 0.262 seconds
```

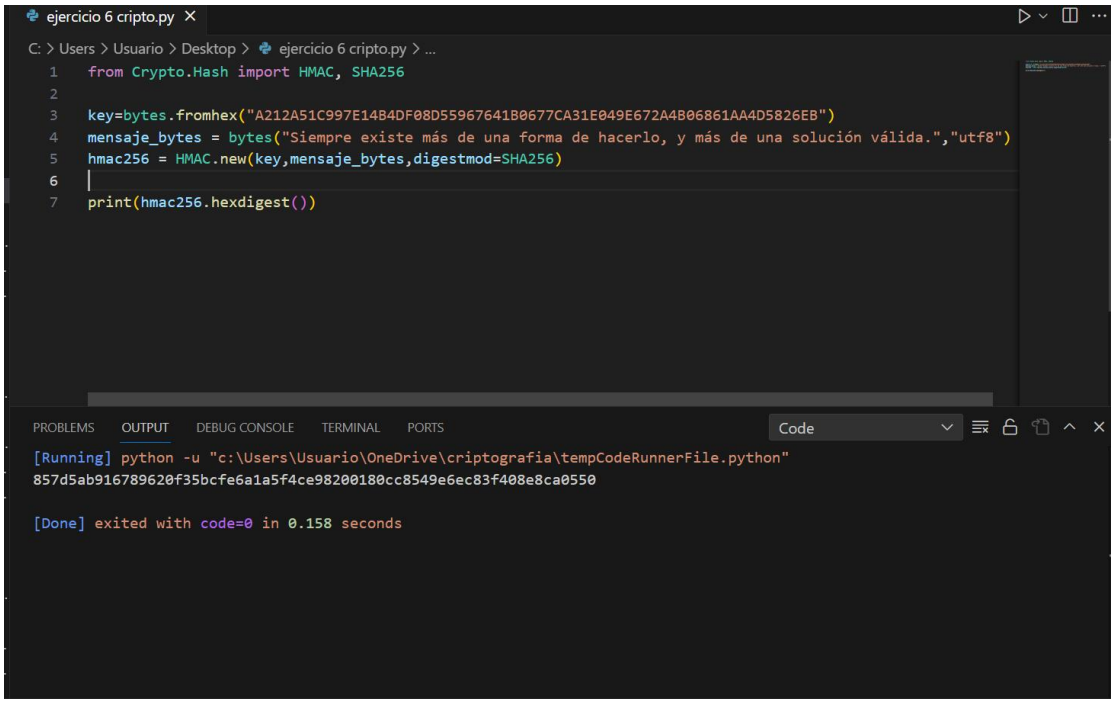
6. Calcula el hmac-256 (usando la clave contenida en el Keystore) del siguiente texto:

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.

Clave:
A212A51C997E14B4DF08D55967641B0677CA31E049E672A4B06861AA4D5826EB

Hmac:
857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550



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?

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?

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

La razón principal es que SHA-1 es considerado obsoleto y vulnerable a ataques de colisión. Esto significa que dos entradas diferentes podrían generar el mismo hash, lo cual es un riesgo de seguridad, que puede provocar falsificación de firmas digitales y la potencial interceptación y descifrado de comunicaciones cifradas que utilizan este algoritmo.

Una mejora común es agregar "sal" (salt) a las contraseñas antes de aplicar el hash. La sal es un valor aleatorio único para cada usuario que se concatena con su contraseña antes de aplicar la función hash.

Aunque SHA-256 con sal es una mejora, es más recomendable utilizar funciones específicamente diseñadas para el almacenamiento seguro de contraseñas.
Algunas de estas funciones son PBKDF2 (Password-Based Key Derivation Function 2), bcrypt, scrypt, Argon…
Hoy por hoy lo ideal es utilizar la función Argon.

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?

Optaríamos por la implementación de AES-GCM. Dado que AES es el método de cifrado simétrico más confiable para salvaguardar las comunicaciones, la combinación con GCM nos permite conferir a AES tanto confidencialidad como autenticidad de los datos (integridad) en una sola operación. De esta manera, podemos verificar que los datos transferidos permanecen inalterados, asegurando la autenticidad de la información.
Datos sensibles como el "idUsuario", "usuario", y "tarjeta" deberían cifrarse antes de ser enviados.

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

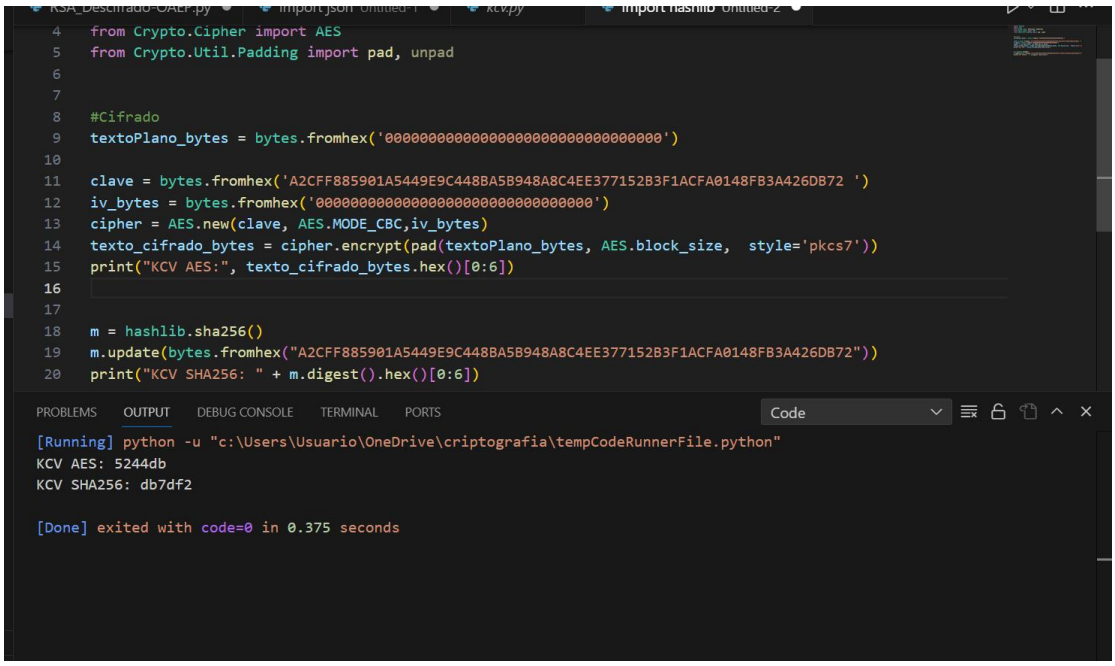
A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72

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.

Respuestas.

KCV (AES): 5244db

KCV SHA256: db7df2



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.

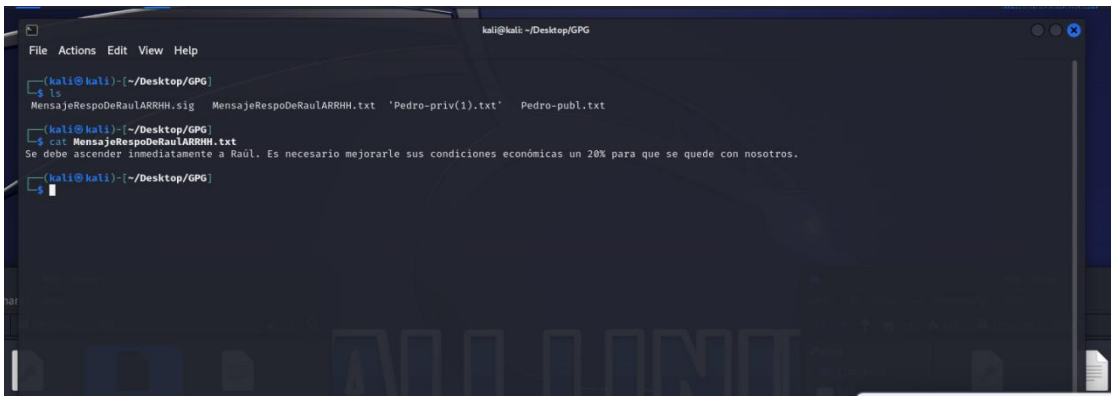
Así mismo, se requiere firmar el siguiente mensaje con la clave correspondiente de las anteriores, simulando que eres personal de RRHH.

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.

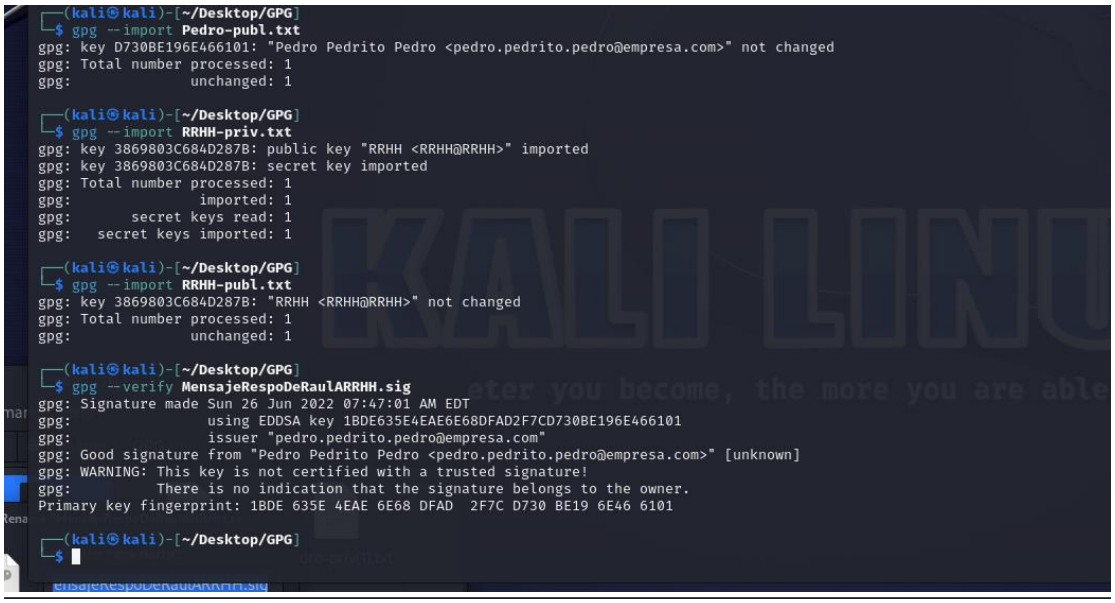
Estamos todos de acuerdo, el ascenso será el mes que viene, agosto, si no hay sorpresas.

LECTURA DEL MENSAJE



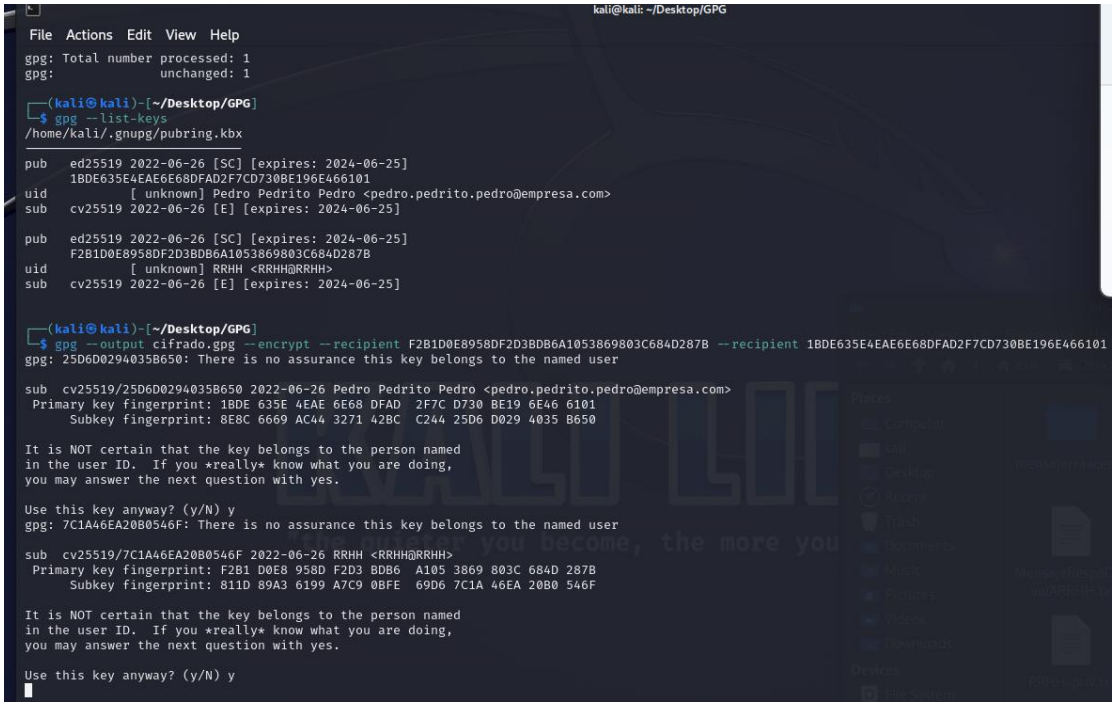
IMPORTACION DE FIRMAS

VERIFICACION DE FIRMA



FIRMA MENSAJE

MENSAJE CIFRADO



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:

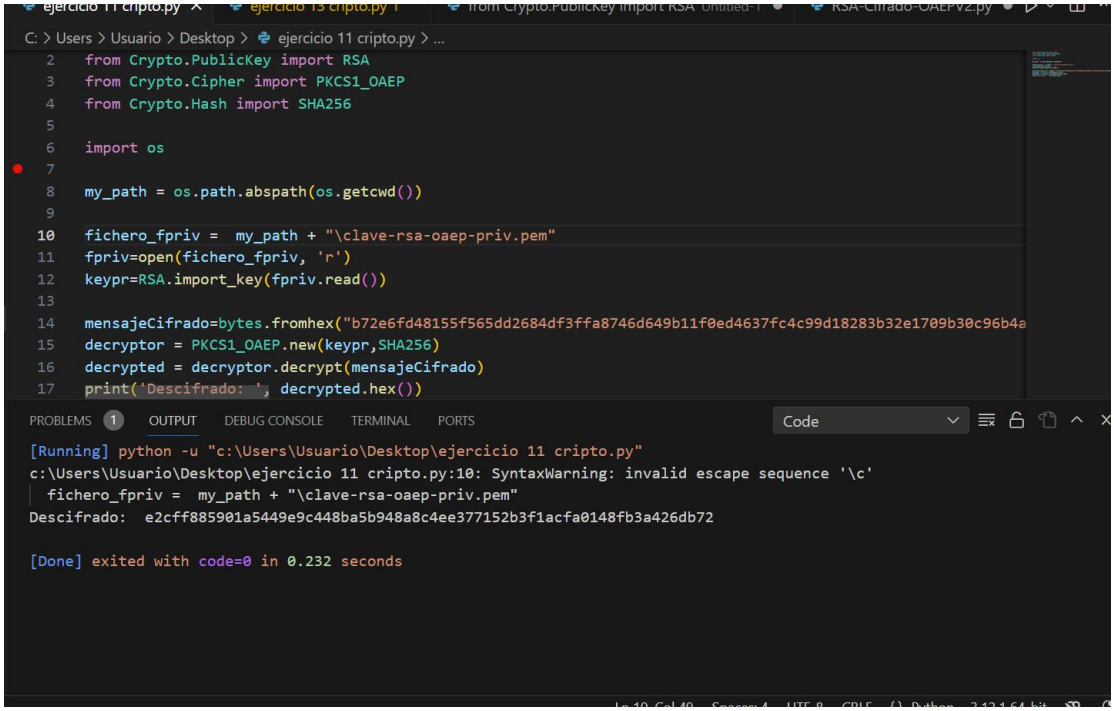
b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e17 09b30c96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d0 4e3d3d4ad629793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae 2c4f5ba670a4126f2f14a9f4b6f41aa2edba01b4bd586624659fca82f5b497018 6502de8624071be78ccef573d896b8eac86f5d43ca7b10b59be4acf8f8e0498a 455da04f67d3f98b4cd907f27639f4b1df3c50e05d5bf63768088226e2a917748 5c54f72407fdf358fe64479677d8296ad38c6f177ea7cb74927651cf24b01dee2 7895d4f05fb5c161957845cd1b5848ed64ed3b0372 2b21a526a6e447cb8ee

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. ¿Por qué son diferentes los textos cifrados?

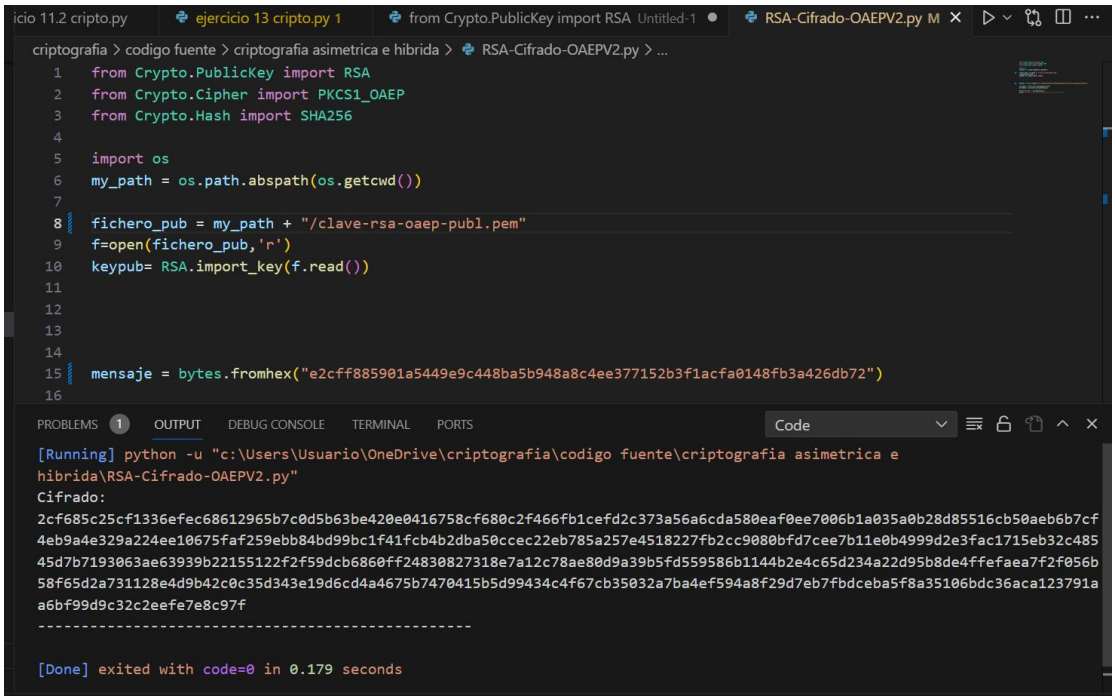
Clave:

e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72



MENSAJE CIFRADO:

```
2cf685c25cf1336efec68612965b7c0d5b63be420e0416758cf680c2f466fb1cefd2c373a56a6cda580eaf0ee7006b1a035a0b28d85
516cb50aeb6b7cf4eb9a4e329a224ee10675faf259ebb84bd99bc1f41fcb4b2dba50ccec22eb785a257e4518227fb2cc9080bfd7cee
7b11e0b4999d2e3fac1715eb32c48545d7b7193063ae63939b22155122f2f59dcb6860ff24830827318e7a12c78ae80d9a39b5fd559
586b1144b2e4c65d234a22d95b8de4ffefaea7f2f056b58f65d2a731128e4d9b42c0c35d343e19d6cd4a4675b7470415b5d99434c4f
67cb35032a7ba4ef594a8f29d7eb7fbdceba5f8a35106bdc36aca123791aa6bf99d9c32c2eefe7e8c97f
```



RESPUESTA:

Los textos cifrados son distintos por el padding del RSA-OAEP.

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:E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148F B3A42 6DB74

Nonce:9Yccn/f5nJJhAt2S

¿Qué estamos haciendo mal?

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.

El problema radica en el nonce, que debe ser siempre un valor aleatorio y nunca repetirse. En el caso del AES, la clave también debe generarse de manera segura y con aleatoriedad, aunque no necesariamente debe generarse en cada operación de cifrado.

TEXTO CIFRADO EN HEXADECIMAL:

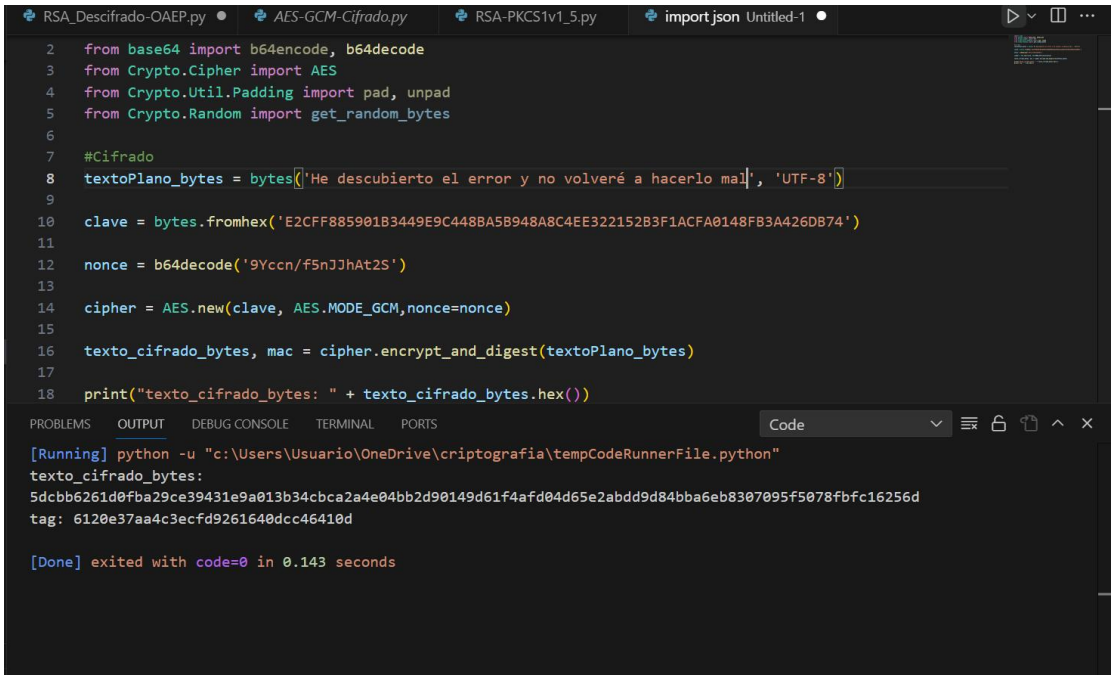
```
5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d
```

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

#Cifrado
textoPlano_bytes = bytes('He descubierto el error y no volveré a hacerlo mal', 'UTF-8')

clave = bytes.fromhex('E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74')

nonce = b64decode('9Yccn/f5nJJhAt2S')
datos_asociados_bytes = bytes("", "UTF-8")
cipher = AES.new(clave, AES.MODE_GCM,nonce=nonce)

cipher.update(datos_asociados_bytes)
#Vamos a cifrar y autenticar.
texto_cifrado_bytes, mac = cipher.encrypt_and_digest(textoPlano_bytes)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                Code

[Running] python -u "c:\Users\Usuario\OneDrive\criptografia\tempCodeRunnerFile.python"
texto_cifrado_bytes:
5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d
tag: 6120e37aa4c3ecfd9261640dcc46410d

[Done] exited with code=0 in 0.376 seconds
```

TEXTO CIFRADO EN B64:

```
5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d
tag: 6120e37aa4c3ecfd9261640dcc46410d
```



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:

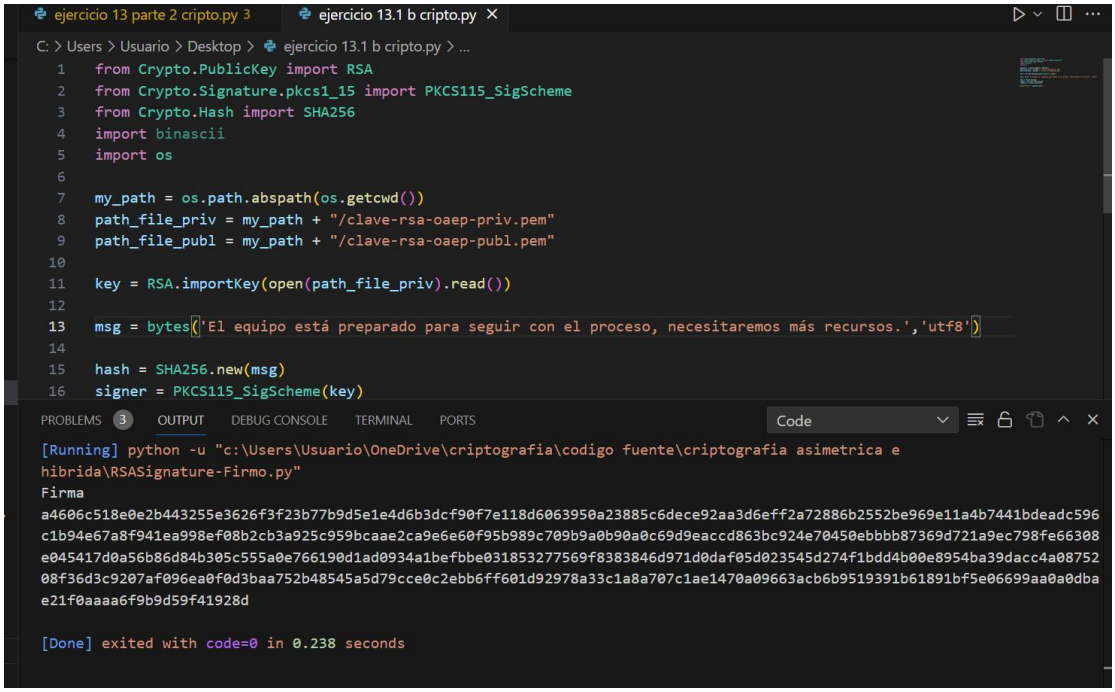
El equipo está preparado para seguir con el proceso, necesitaremos más recursos.

¿Cuál es el valor de la firma en hexadecimal?

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

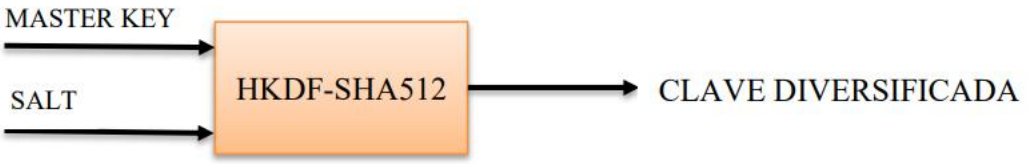
VALOR DE LA FIRMA

a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21f0aaaa6f9b9d59f41928d

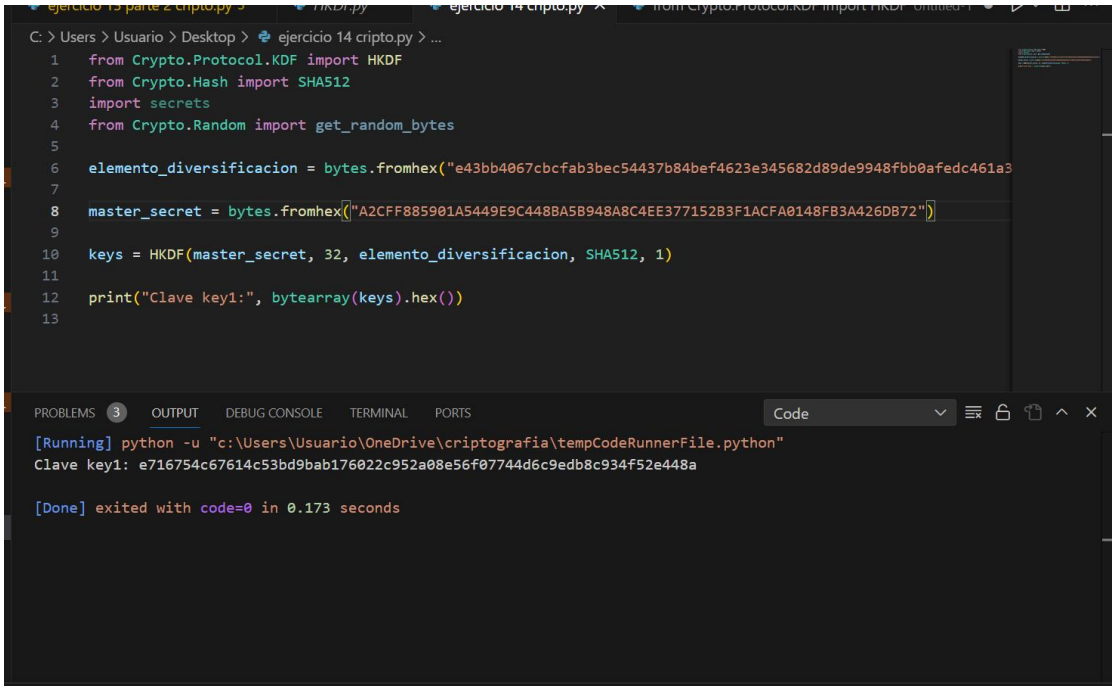


14. Necesitamos generar una nueva clave AES, usando para ello una HKDF (HMAC-based Extractand-Expand key derivation function) con un hash SHA-512. La clave maestra requerida se encuentra en el keystore con la etiqueta “cifrado-sim-aes-256”. La clave obtenida dependerá de un identificador de dispositivo, en este caso tendrá el valor en hexadecimal:

e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3



¿Qué clave se ha obtenido?



15. Nos envían un bloque TR31:

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D51 5ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B 2E5657495E03CD857FD37018E111B

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

A1A10101010101010101010101010102

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

Bloque de claves protegido mediante el método de vinculación de derivación de claves AES.

¿Para qué algoritmo se ha definido la clave?

AES.

¿Para qué modo de uso se ha generado?

Clave simétrica para cifrado de datos.

¿Es exportable?

Si, es exportable bajo clave no fiable.

¿Para qué se puede usar la clave?

Para encriptar y desencriptar.

¿Qué valor tiene la clave?

c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1

```
from psec import tr31

#Documentado en este fichero
#https://github.com/knovichikhin/psec/blob/master/psec/tr31.py

header, key = tr31.unwrap( kbpk=bytes.fromhex("A1A10101010101010101010101010102"), key_block="D0144D0AB00S
print(key.hex())

print("Key Version ID: " + header.version_id )
print("Algoritmo: " + header.algorithm)
print("Modo de uso: " + header.mode_of_use)
print("Uso de la clave: " + header.key_usage)
print("Exportabilidad: " + header.exportability)
```

```
PROBLEMS 1  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                Code

[Running] python -u "c:\Users\Usuario\OneDrive\criptografia\codigo fuente\Gestión de Claves\cabeceraTR31.py"
c2c1c1c1c1c1c1c1c1c1c1c1c1c1c1c2
Key Version ID: D
Algoritmo: A
Modo de uso: B
Uso de la clave: D0
Exportabilidad: S

[Done] exited with code=0 in 0.209 seconds
```