



Practica: Criptografía

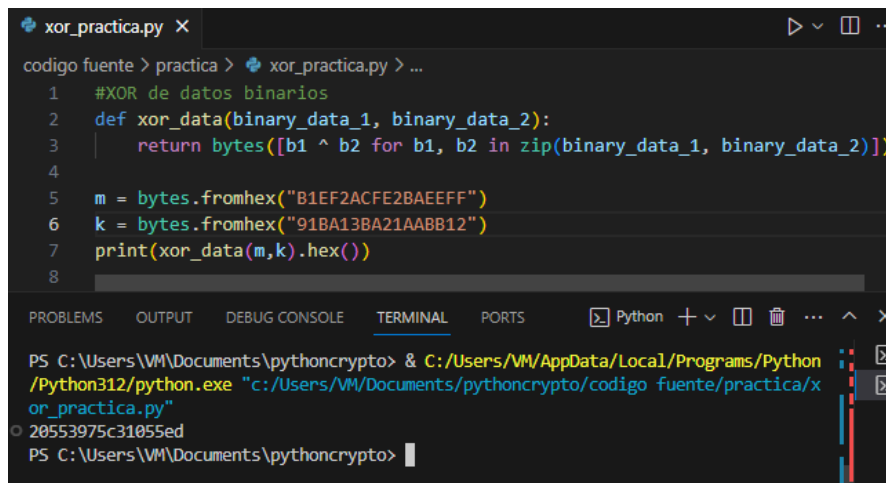
Índice

Ejercicio 1	Pag. 3
Ejercicio 2	Pag. 4
Ejercicio 3	Pag. 6
Ejercicio 4	Pag. 7
Ejercicio 5	Pag. 9
Ejercicio 6	Pag. 10
Ejercicio 7	Pag. 10
Ejercicio 8	Pag. 11
Ejercicio 9	Pag. 13
Ejercicio 10	Pag. 14
Ejercicio 11	Pag. 16
Ejercicio 12	Pag. 17
Ejercicio 13	Pag. 18
Ejercicio 14	Pag. 20
Ejercicio 15	Pag. 21
Ficheros Adjuntos	Pag. 21

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?

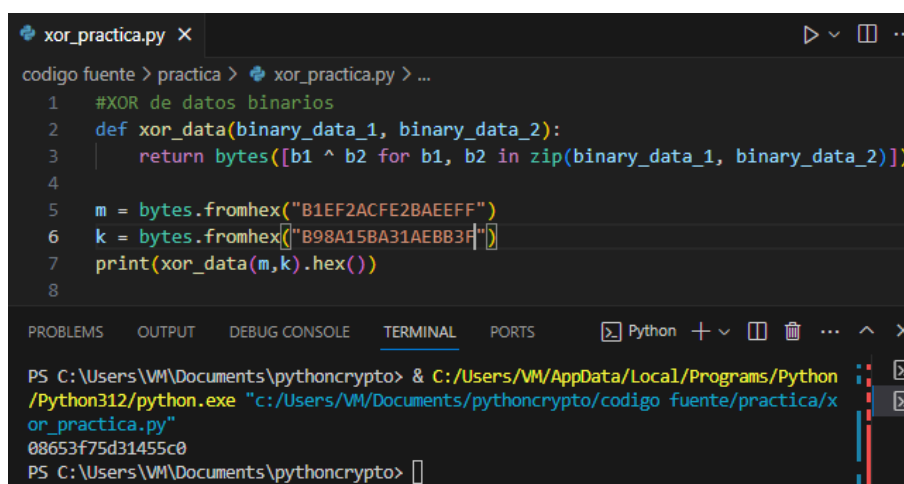


```
xor_practica.py X
codigo fuente > practica > xor_practica.py > ...
1  #XOR de datos binarios
2  def xor_data(binary_data_1, binary_data_2):
3      return bytes([b1 ^ b2 for b1, b2 in zip(binary_data_1, binary_data_2)])
4
5  m = bytes.fromhex("B1EF2ACFE2BAEEFF")
6  k = bytes.fromhex("91BA13BA21AABB12")
7  print(xor_data(m,k).hex())
8
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\VM\Documents\pythoncrypto> & C:/Users/VM/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/VM/Documents/pythoncrypto/codigo fuente/practica/xor_practica.py"
20553975c31055ed
PS C:\Users\VM\Documents\pythoncrypto>
```

20553975c31055ed

La clave fija, recordemos es B1EF2ACFE2BAEEFF, mientras que en producción sabemos que la parte dinámica que se modifica en los ficheros de propiedades es B98A15BA31AE3B3F.

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



```
xor_practica.py X
codigo fuente > practica > xor_practica.py > ...
1  #XOR de datos binarios
2  def xor_data(binary_data_1, binary_data_2):
3      return bytes([b1 ^ b2 for b1, b2 in zip(binary_data_1, binary_data_2)])
4
5  m = bytes.fromhex("B1EF2ACFE2BAEEFF")
6  k = bytes.fromhex("B98A15BA31AE3B3F")
7  print(xor_data(m,k).hex())
8
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\VM\Documents\pythoncrypto> & C:/Users/VM/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/VM/Documents/pythoncrypto/codigo fuente/practica/xor_practica.py"
08653f75d31455c0
PS C:\Users\VM\Documents\pythoncrypto>
```

08653f75d31455c0

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/o0QKIWXg3nu1RRz4QWEIezdrLAD5L
O4US t3aB/i50nvvJbBiG+le1ZhpR84ol=

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

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

```
> AES-CBC practica > AES-CBC practica > ...  
1 import sys  
2 from base64 import b4encode, b4decode  
3 from Crypto.Cipher import AES  
4 from Crypto.Util.Padding import pad, unpad  
5 from Crypto.Random import get_random_bytes  
6  
7  
8 clave = bytes.fromhex('A2CFB85901A544E9C4ABBA504848CEE377152B31ACFA014FB3A4260D72')  
9 iv_bytes = bytes.fromhex('0000000000000000000000000000000000000000000000000000')  
10 texto_cifrado_b64 = b4decode('TQVSOHkcbafS9S1xhfKwXlBUxpckdSXfSj/SnL7Iof/qOQClNgSmuRRzq4WElzdrLD0SL4DSUJabf/iSmmvYjbIG4leizhpR4oi=  
11  
12 #Descifrado  
13  
14 try:  
15  
16     cipher = AES.new(clave, AES.MODE_CBC, iv_bytes)  
17     mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_b64), AES.block_size, style='pkcs7')  
18     mensaje_des_bytes_verpad = cipher.decrypt(texto_cifrado_b64)  
19  
20     print(mensaje_des_bytes.hex())  
21     print(b4encode(mensaje_des_bytes).decode('utf-8'))  
22     print("El texto en claro es: ", mensaje_des_bytes.decode("utf-8"))  
23     print("Mensaje viendo el padding: ",mensaje_des_bytes.verpad.hex())  
24  
25 except (ValueError, KeyError) as error:  
26     print("Problema para descifrar...")  
27     print("El motivo del error es:", error)
```

PROBLEMAS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
4573744e2f68572875ee20eb36967262164626556e20c26cf7175652074c7ad7086936f2e20c26565175652162c207661732076f77220656c206275656e206163616d96ef2e20c316e696de  
f2e  
20b0b71c9b3b16c9c794bf10a10103f21580636e46f1b524a1020c70e9gfhZJBheJilIClAw10G4blldbydygeufdvstvg=  
El texto en claro es: Esto es un cifrado en bloque tipo CBC, usa por el buen camino. Am\u00fas.  
Mensaje viendo el padding: d83caad7b67525ee081795fd1a9c364f20856e20c26cf7175652074c7ad7086936f2e20c26565175652162c207661732076f77220656c206275656e2  
083616969ef2e20c316e69df2ed1  
[5] C:\Users\Alfonso> python test.py
```

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

```

AES-CRC_practica.py ×
codigo fuente > practica > AES-CRC_practica.py ...
1 import sys
2 from base64 import b64encode, b64decode
3 from Crypto.Cipher import AES
4 from Crypto.Util.Padding import pad, unpad
5 from Crypto.Random import get_random_bytes
6
7
8 clave = bytes.fromhex("A2CFF8B5981A5449E9CA48B80B4B8BCAE377152B1F1ACFA0148F81B46160B72")
9 iv_bytes = bytes.fromhex("8000000000000000000000000000000000000000000000000000000000000000")
10 texto_cifrado_b64 = b64decode("IQ5059rC6a59S1xhFCwN1RtXpPCd50K5T5/sNt170f/qRQcXhGmJm1Rk2AQezldRLADSL04US13ab/15hmvY3BtG4ie1Zp8k4oi-")
11
12 #Descifrado
13
14 try:
15
16     cipher = AES.new(clave, AES.MODE_CBC, iv_bytes)
17     mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_b64), AES.block_size, style='x92')
18     mensaje_des_bytes_verpad = cipher.decrypt(texto_cifrado_b64)
19
20     print(mensaje_des_bytes.hex())
21     print(b64encode(mensaje_des_bytes).decode('utf-8'))
22     print("El texto en claro es: ", mensaje_des_bytes.decode("utf-8"))
23     print("Mensaje viendo el padding: ", mensaje_des_bytes_verpad.hex())
24
25 except (ValueError, KeyError) as error:
26     print("Problemas para descifrar...")
27     print("El motivo del error es: ", error)

```

Comprobamos que con pkcs7 el padding es 01 por tanto al cambio de x923, se queda igual y no sucede nada porque ya esta usando 1 byte de padding

¿Cuánto padding se ha añadido en el cifrado?

1byte (01)

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

```
AES-CBC_usando_keystore_practica.py X
codigo fuente > practica > AES-CBC_usando_keystore_practica.py > ...
1 import json
2 from base64 import b64encode, b64decode
3 from Crypto.Cipher import AES
4 from Crypto.Util.Padding import pad, unpad
5 from Crypto.Random import get_random_bytes
6 import jks
7 import os
8
9 # Obteniendo el path
10 path = os.path.dirname(__file__)
11 keystore = path + "/KeyStorePracticas"
12 ks = jks.KeyStore.load(keystore, "123456")
13
14 for alias, sk in ks.secret_keys.items():
15     if sk.alias == "cifrado-sim-aes-256":
16         key = sk.key
17
18 clave=bytes.fromhex(key.hex())
19 iv_bytes = bytes.fromhex('00000000000000000000000000000000')
20 texto_cifrado_b64 = b64decode('TQ9SOWKc6aF59S1xhfK9wT18UXpPCd505XF5J/5nLI70f/o0QKIWg3nu1RRz4QNEIezdrLAD5L04UST3aB/150nvvJb81G+1e1ZhpR84oI=')
21
22 #Descifrado
23
24 try:
25     cipher = AES.new(clave, AES.MODE_CBC, iv_bytes)
26     mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_b64), AES.block_size, style="x923")
27     mensaje_des_bytes_verpad = cipher.decrypt(texto_cifrado_b64)
28
29     print(mensaje_des_bytes.hex())
30     print(b64encode(mensaje_des_bytes).decode('utf-8'))
31     print("El texto en claro es: ", mensaje_des_bytes.decode("utf-8"))
32     print("Mensaje viendo el padding: ",mensaje_des_bytes_verpad.hex())
33
34 except (ValueError, KeyError) as error:
35     print("Problemas para descifrar....")
36     print("El motivo del error es: ", error)
37
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\VM\Documents\pythondcrypto> & C:\Users\VM\AppData\Local\Programs\Python\Python312\python.exe "c:/Users/VM/Documents/pythondcrypto/codigo_fuente/practica/AES-CBC_usando_keystore_practica.py"
4573746f20957320756e206369667261646f20656e20626c6f7175652074c3ad7069636f202052656375657264612c2076617320706f7220656c206275656e2063616d696e6f2020c3816e696d6f2e
RXN8y8lcy81b1b1j4ZyYwRvIGVlIG13b3F1ZS90w61w6WVlBS2M1ZCkYsagdfZIH8vc1BlbcBldwVlG0HbWluby4qW4Fuak1vlg==
El texto en claro es: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.
Mensaje viendo el padding: d854ca9d7b63529ee88379e5fd1a90e3646f20656e20626c6f7175652074c3ad7069636f202052656375657264612c2076617320706f7220656c206275656e2063616d696e6f2020c3816e696d6f2e01
PS C:\Users\VM\Documents\pythondcrypto>
```

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.

```
ChaCha20-Cifrado_practica.py X
codigo fuente > criptografia en flujo > ChaCha20-Cifrado_practica.py > ...
1 from Crypto.Cipher import ChaCha20
2 from base64 import b64decode, b64encode
3
4 textoPlano_bytes = bytes('KeepCoding te enseña a codificar y a cifrar', 'UTF-8')
5 clave = bytes.fromhex('AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120')
6 nonce_mensaje = b64decode('9Yccn/f5nJJhAt2S')
7 print('nonce = ', nonce_mensaje.hex())
8
9
10 cipher = ChaCha20.new(key=clave, nonce=nonce_mensaje)
11 texto_cifrado_bytes = cipher.encrypt(textoPlano_bytes)
12 print('Mensaje cifrado en HEX = ', texto_cifrado_bytes.hex() )
13 print('Mensaje cifrado en B64 = ', b64encode(texto_cifrado_bytes).decode())
14
15

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\VM\Documents\pythoncrypto> & C:/Users/VM/AppData/Local/Programs/Python/Python39-32/python.exe "c
:/Users/VM/Documents/pythoncrypto/codigo fuente/criptografia en flujo/ChaCha20-Cifrado_practica.py"
nonce = f5871c9ff7f99c926102dd92
Mensaje cifrado en HEX = 69ac4ee7c4c552537a00a19bcaf7f0aaed7c9c8f769956a09bce6f4def6c3535f2211c9467067cf5c4
a842ab
Mensaje cifrado en B64 = aaxO58TFUJN6AKGbyvfwqu18nI92mVagm8Svre9sNTXyIRyUZwZ89cSoQqs=
PS C:\Users\VM\Documents\pythoncrypto>
```

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

Haciendo que el nonce sea aleatorio

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

```
ChaCha20-Cifrado_practica.py X
codigo fuente > criptografia en flujo > ChaCha20-Cifrado_practica.py > ...
1 from Crypto.Cipher import ChaCha20
2 from base64 import b64decode, b64encode
3 from Crypto.Random import get_random_bytes
4
5 textoPlano_bytes = bytes('KeepCoding te enseña a codificar y a cifrar', 'UTF-8')
6 clave = bytes.fromhex('AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120')
7 |
8 #Hacemos que el nonce sea aleatorio
9 nonce_mensaje = get_random_bytes(12)
10 print('nonce = ', nonce_mensaje.hex())
11
12 #Ciframos
13 cipher = ChaCha20.new(key=clave, nonce=nonce_mensaje)
14 texto_cifrado_bytes = cipher.encrypt(textoPlano_bytes)
15 print('Mensaje cifrado en HEX = ', texto_cifrado_bytes.hex() )
16 print('Mensaje cifrado en B64 = ', b64encode(texto_cifrado_bytes).decode())
17
18

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\VM\Documents\pythoncrypto> & C:/Users/VM/AppData/Local/Programs/Python/Python39-32/python.exe "c
:/Users/VM/Documents/pythoncrypto/codigo fuente/criptografia en flujo/ChaCha20-Cifrado_practica.py"
nonce = 77436a8416b12f911c89384b
Mensaje cifrado en HEX = 61375b1ec838ebf6dad27209d6df019f6dac38181ebec1cd824eb88227fe535f182bfea883218338bf
b71ad5
Mensaje cifrado en B64 = YTdbHsg46/ba0nI3it8Bn22s0BgevsHlkg6gif+U18YK/6ogyGDOL+3GtU=
PS C:\Users\VM\Documents\pythoncrypto>
```

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmVlIjoRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcylsInJvbCI6ImIzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzFQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

¿Qué algoritmo de firma hemos realizado?

HS256

¿Cuál es el body del jwt?

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

A screenshot of a Python IDE window titled 'jwt simetrico_practica.py'. The code in the editor is as follows:

```
1 import jwt
2
3
4 encoded_jwt="eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmVlIjoRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcylsInJvbCI6ImIzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzFQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE"
5 decode_jwt = jwt.decode(encoded_jwt,"Con KeepCoding aprendemos", algorithms="HS256")
6
7 print(decode_jwt)
8
9
```

The bottom panel of the IDE shows the 'TERMINAL' output with the following text:

```
PS C:\Users\WM\Documents\pythondocuments> & C:/Users/WM/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/WM/Documents/pythondocuments/codigo fuente/Hashing y Authentication/jwt simetrico_practica.py"
Password correcta
{'usuario': 'Don Pepito de los palotes', 'rol': 'isNormal', 'iat': 1667933533}
PS C:\Users\WM\Documents\pythondocuments>
```

Un hacker está enviando a nuestro sistema el siguiente jwt:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmVlIjoRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcylsInJvbCI6ImIzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRnV2CIAODIHRI

¿Qué está intentando realizar?

Está intentando hacer un cambio de rol

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

¿Qué ocurre si intentamos validarlo con pyjwt?

```
jwt simetrico.practica.py •
```

```
codigo fuente > Hashing y Authentication > jwt simetrico.practica.py > [!] encoded_jwt
```

```
1 import jwt
```

```
2
```

```
3 [encoded_jwt="eyJ0eXAiOiJKV1QiLCBhbGkiOiJIUzI1NiIsInR5cCI6IzcwmlmVoiJG9uIjB1CglibyBKZS83MgcOfsb3RlcYIsInR5bCI6ImlzQWltaw4lClpwYXQlOjE2Njc3MzH9.krgBkzCBQ5MZ83nZHuRvnmA2dg42MeRNv2CIAODjHR1"]
```

```
4 decode_jwt = jwt.decode(encoded_jwt,"Con KeepCoding aprendemos", algorithms="HS256")
```

```
5
```

```
6 print(decode_jwt)
```

```
7
```

```
8
```

```
9
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
```

```
File "C:\Users\WM\AppData\Local\Programs\Python\Python312\Lib\site-packages\jwt\api_jwt.py", line 218, in decode  
    decoded = self.decode_complete(  
              ^^^^^^^^^^^^^^^^^  
File "C:\Users\WM\AppData\Local\Programs\Python\Python312\Lib\site-packages\jwt\api_jwt.py", line 151, in decode_complete  
    decoded = api_jws.decode_complete(  
              ^^^^^^^^^^^^^^^^^  
File "C:\Users\WM\AppData\Local\Programs\Python\Python312\Lib\site-packages\jwt\api_jws.py", line 289, in decode_complete  
    self.verify_signature(signing_input, header, signature, key, algorithms)  
File "C:\Users\WM\AppData\Local\Programs\Python\Python312\Lib\site-packages\jwt\api_jws.py", line 310, in _verify_signature  
    raise InvalidSignatureError("Signature verification failed")  
jwt.exceptions.InvalidSignatureError: Signature verification failed  
PS C:\Users\WM\Documents>pythoncrypto.py
```

La verificación de la firma, falla. Así que evita cualquier cambio en el payload

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?

Un SHA3-256

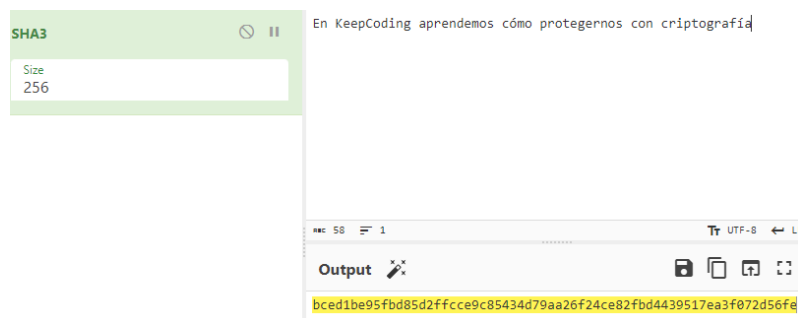
Y si hacemos un SHA2, y obtenemos el siguiente resultado:

4cec5a9f85dcc5c4c6ccb603d124cf1cd6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

¿Qué hash hemos realizado?

Un SHA-512

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?

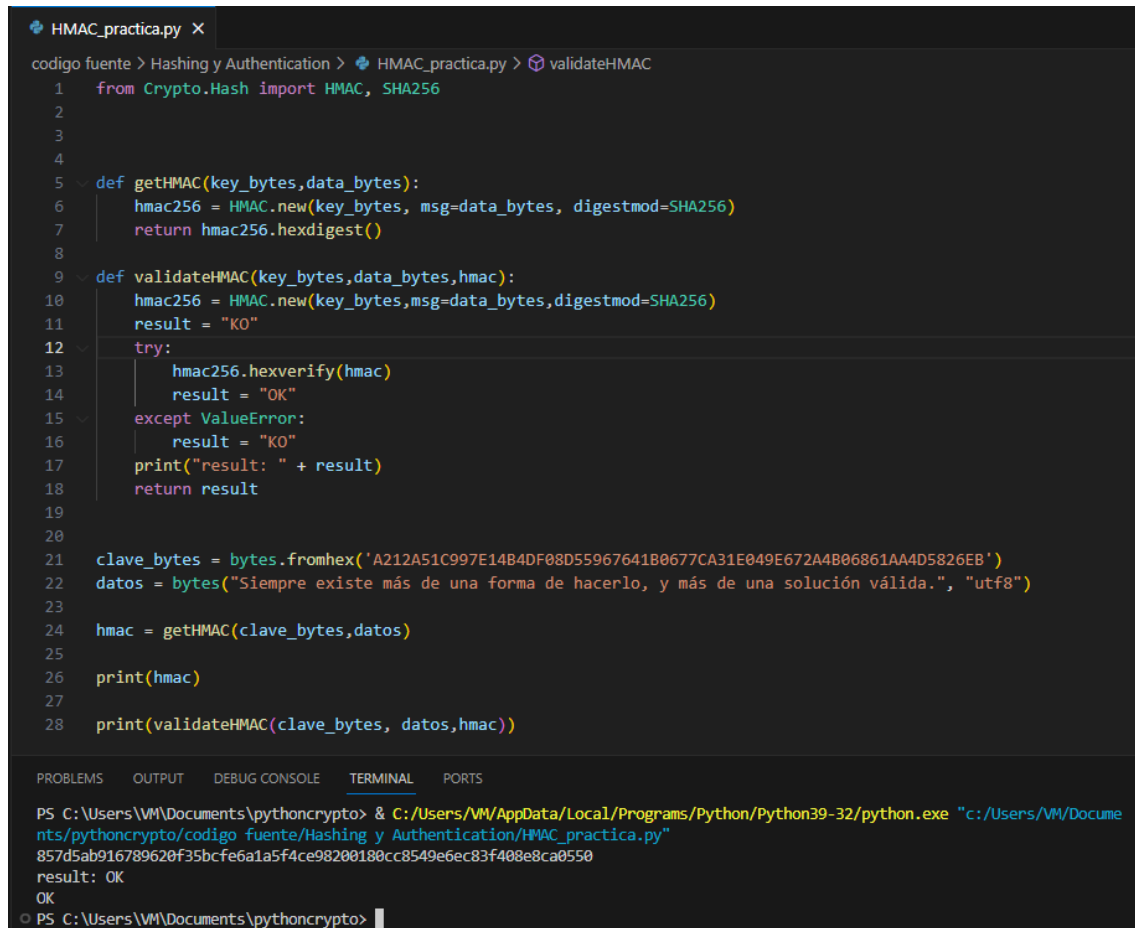


La propiedad de difusión, cualquier cambio en el texto de entrada, el resultado del hash será totalmente distinto

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.



The screenshot shows a Python script named `HMAC_practica.py` in a code editor. The script defines two functions: `getHMAC` and `validateHMAC`. `getHMAC` takes a key and data in bytes and returns the HMAC256 hexdigest. `validateHMAC` takes the same key and data, calculates the HMAC, and compares it with the provided HMAC, returning "OK" or "KO". The script then defines a key in hexadecimal, a message in UTF-8, calculates the HMAC, and prints it. Finally, it calls `validateHMAC` with the key, message, and calculated HMAC, printing the result.

```
1 from Crypto.Hash import HMAC, SHA256
2
3
4
5 def getHMAC(key_bytes, data_bytes):
6     hmac256 = HMAC.new(key_bytes, msg=data_bytes, digestmod=SHA256)
7     return hmac256.hexdigest()
8
9 def validateHMAC(key_bytes, data_bytes, hmac):
10    hmac256 = HMAC.new(key_bytes, msg=data_bytes, digestmod=SHA256)
11    result = "KO"
12    try:
13        hmac256.hexverify(hmac)
14        result = "OK"
15    except ValueError:
16        result = "KO"
17    print("result: " + result)
18    return result
19
20
21 clave_bytes = bytes.fromhex('A212A51C997E14B4DF08D55967641B0677CA31E049E672A4806861AA4D5826EB')
22 datos = bytes("Siempre existe más de una forma de hacerlo, y más de una solución válida.", "utf8")
23
24 hmac = getHMAC(clave_bytes, datos)
25
26 print(hmac)
27
28 print(validateHMAC(clave_bytes, datos, hmac))
```

The terminal output shows the execution of the script:

```
PS C:\Users\VM\Documents\pythoncrypto> & C:/Users/VM/AppData/Local/Programs/Python/Python39-32/python.exe "c:/Users/VM/Docume
nts/pythoncrypto/codigo fuente/Hashing y Authentication/HMAC_practica.py"
857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550
result: OK
OK
PS C:\Users\VM\Documents\pythoncrypto>
```

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?

El SHA-1 es fácil de romper con rainbow tables (ataque de fuerza bruta)

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?

Uno de los hashes más seguros es Argon2 para almacenar datos en base de datos, así que propondría utilizar ese

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.

```
JWT RSA_ej8_practica.py x
codigo fuente > practica > jwt RSA_ej8_practica.py ...
1 from ssl import VerifyFlags
2 import jwt
3
4 #Clave publica y privada
5 public_key = open("C:\\Users\\WM\\Documents\\pythoncrypto\\codigo fuente\\Hashing y Authentication\\public.pem").read()
6 private_key = open("C:\\Users\\WM\\Documents\\pythoncrypto\\codigo fuente\\Hashing y Authentication\\private.pem").read()
7
8
9 #Codificamos el POST
10 print("=====")
11 encoded = jwt.encode("idUsuario": "1", "usuario": "José Manuel Barrio Barrio", "tarjeta": "4231212345676891", private_key, algorithm="RS256")
12 print(encoded)
13 print("=====")
14
15 #Comprovamos la firma con la clave publica
16 decode = jwt.decode(encoded, public_key, algorithms="RS256", options={"verify_signature": True})
17 print(decode)
18 print("=====")
19
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```

=====
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZiZvdWkiOmYwODI0IiwidXN1YXJpbSI6Ikpvc1x1M0B1OSBNYW51ZWwQcmFmcmlvIEhcnjpbYiIsInRhcmlkGE0iOiJHbWJyMjEyMzQ1Njc2ODRkInR0L3Vn
2lpXU8PXRwCshkyO3G6VpVDtFeQggGT1MXTB9L2d37MadN8atvgRBUGVRGc39dL5sr_t27LRcNtBqJsuH5WuZKCe-h08X8vNyn-gAU01A82GeywQ0m0V1HCddvnm9e4Dm0YxShqdyCacUcVdu_BRDk2-j8Ro
wLxhm7g8I3Jbw7CzBxm1g4SA00FL9PCmkmk8RUKa2wOyBuUjJ-asoFw0Sk33CBPrh9RbxCc16B4lMuXdyYdJzccLFQny-dotN8MvHsVHRPZzxUxMqVXiHbBt4CSmgZyYJfGu_wGZXQTj9wVbuvQ3nczcwFt1
2smYb3JebZxrsRw
=====
{'idUsuario': '1', 'usuario': 'José Manuel Barrio Barrio', 'tarjeta': '4231212345676891'}
PS C:\Users\WM\Documents\pythoncrypto>
```

¿Qué algoritmos usarías?

Para garantizar la robustez del contenido se utilizaría el algoritmo SHA256 con una verificación de firma publica

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.

```
kcv_practica.py X
codigo fuente > practica > kcv_practica.py > ...
1  import hashlib
2  import json
3  from base64 import b64encode, b64decode
4  from Crypto.Cipher import AES
5  from Crypto.Util.Padding import pad, unpad
6
7
8  #Cifrado
9  textoPlano_bytes = bytes.fromhex('00000000000000000000000000000000')
10
11  clave = bytes.fromhex('A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72')
12  iv_bytes = bytes.fromhex('00000000000000000000000000000000')
13  cipher = AES.new(clave, AES.MODE_CBC, iv_bytes)
14  texto_cifrado_bytes = cipher.encrypt(pad(textoPlano_bytes, AES.block_size, style='pkcs7'))
15  #Calculamos la clave KVC AES
16  print("KCV AES:", texto_cifrado_bytes.hex()[0:6])
17
18  #Calculamos con el hash SHA256
19  m = hashlib.sha256()
20  m.update(bytes.fromhex("A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"))
21  print("KCV SHA256: " + m.digest().hex()[0:6])

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
● PS C:\Users\VM\Documents\pythoncrypto> & C:/Users/VM/AppData/Local/Programs/Python/Python39-32/python.exe "c:/Users/VM/Documents/pythoncrypto/codigo fuente/practica/kcv_practica.py"
KCV AES: 5244db
KCV SHA256: db7df2
○ PS C:\Users\VM\Documents\pythoncrypto> 
```

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.

```
Administrador: Windows PowerShell
PS C:\practica_gpg> gpg --output .\verificacion_pedro_pub.txt --decrypt -u 1BDE635E4EAE6E68DFAD2F7CD7308E196E466101 .\MensajeRespoDeRaulARRHH.sig
gpg: Firmado el 26/06/2022 13:47:01 Hora de verano romance
gpg: usando EDDSA clave 1BDE635E4EAE6E68DFAD2F7CD7308E196E466101
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 4EAE 6E68 DFAD 2F7C D730 8E19 6E46 6101
PS C:\practica_gpg>
```

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.

```
Administrador: Windows PowerShell
PS C:\practica_gpg> gpg --list-keys
[keyboxd]
-----
pub   ed25519 2022-06-26 [SC] [caduca: 2024-06-25]
      1BDE635E4EAE6E68DFAD2F7CD7308E196E466101
uid   [desconocida] Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>
sub   cv25519 2022-06-26 [E] [caduca: 2024-06-25]

pub   ed25519 2022-06-26 [SC] [caduca: 2024-06-25]
      F2B1D0E8958DF2D38DB6A1053869803C684D287B
uid   [desconocida] RRHH <RRHH@RRHH>
sub   cv25519 2022-06-26 [E] [caduca: 2024-06-25]

PS C:\practica_gpg> gpg --output .\verificacion_rhh_pub.txt --decrypt -u F2B1D0E8958DF2D38DB6A1053869803C684D287B .\MensajeRespoDeRaulARRHH.sig
gpg: Firmado el 26/06/2022 13:47:01 Hora de verano romance
gpg: usando EDDSA clave 1BDE635E4EAE6E68DFAD2F7CD7308E196E466101
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 4EAE 6E68 DFAD 2F7C D730 8E19 6E46 6101
PS C:\practica_gpg>
```

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.

```
Administrador: Windows PowerShell
PS C:\practica_gpg> gpg --output firma_msj_personalderrhh.sig --clearsign .\msj_personal_rrhh.txt
PS C:\practica_gpg> cat .\firma_msj_personalderrhh.sig
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA512

Viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario. Saludos.
-----BEGIN PGP SIGNATURE-----

iHUEARYKAB0WlQQb3mNeTq5uaN+tL3zXML4ZbkZhaQUcZaBBnAAKCRDXML4ZbkZh
AbEGAQDN9byGJmX8zbaVXi8EuQkusAhONsgXiR+2nEG7JkzujgD/Scu26XDgvBmz
qesCyCpheJwz05ow+03E6w5srPydlAw=
=Ri0Q
-----END PGP SIGNATURE-----
PS C:\practica_gpg>
```

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.

```
Administrador: Windows PowerShell
PS C:\practica_gpg> gpg --output msj_cifrado.gpg --encrypt --armor --recipient 1BDE635E4EA6E68DFAD2F7CD7308E196E466101 --recipient F2B1D0E8958DF2D38DB6A1053869803C6840287B .\msj_cifrado.txt
gpg: 7C1A46EA20B0546F: No hay seguridad de que esta clave pertenezca realmente al usuario que se nombra
sub cv25519/7C1A46EA20B0546F 2022-06-26 RRHH <RRHH@RRHH>
Huella clave primaria: F2B1 D0E8 958D F2D3 B0B6 A105 3869 803C 684D 287B
Huella de subclave: 811D 89A3 6199 A7C9 08FE 69D6 7C1A 46EA 20B0 546F
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 sí a la siguiente pregunta.
¿Usar esta clave de todas formas? (s/N) s
gpg: 25D6D0294035B650: No hay seguridad de que esta clave pertenezca realmente al usuario que se nombra
sub cv25519/25D6D0294035B650 2022-06-26 Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>
Huella clave primaria: 1BDE 635E 4EAE 6E68 DFAD 2F7C D730 8E19 6E46 6101
Huella de subclave: 8EBC 6669 AC44 3271 42BC C244 25D6 D029 4035 B650
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 sí a la siguiente pregunta.
¿Usar esta clave de todas formas? (s/N) s
PS C:\practica_gpg> cat .\msj_cifrado.gpg
-----BEGIN PGP MESSAGE-----
hF4DjddQKUA1t1ASAgdAU5k/owTIhpvezD9ayatt/ckDc0xhH39ixqAL6F3vGRsw
Qz37nUQUUCX0T9pLltB2sydcisPtiTSazHm+2ae40paeDuoZ0xOQ+12D14x8Zj
hF4DFp66s1CwG8SAQdAdC30+Ejqr0XELip18N/B7Jt+xcQX9Y+30ymw8LIZx8w
k/rX0rGd81BPLY61QmslGpAgC2GeEqZcmcVHfX8KF+EXZf0iVrWbVRRiZX4dhe
1KEBCQ1Qm1jpmW4W9XeqmFKA1aN7GdyLxcKKBZM1RAOrxP56cs2BPzwE707Lmf
vrtuyXJAp1f8uK0931r02H4318Ppge2i0on/SV1tySCnMc2EfdqgPpZn22xN
/nLSrQY4TnyCeaq7VTJ35dI7tgFuisWssmbLZ0y39xi-Vp2UMD+AK5GFEE7r
Kwrv1HwKjg7lUYnKIPAftzGT1A==
=xakM
-----END PGP MESSAGE-----
PS C:\practica_gpg>
```

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

b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c
96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d04e3d3d4ad629
793eb00cc76d10fc00475eb76bfbcb1273303882609957c4c0ae2c4f5ba670a4126f2f14
a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8624071be78ccef573d
896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f67d3f98b4cd907f27639f4b1
df3c50e05d5bf63768088226e2a9177485c54f72407fdf358fe64479677d8296ad38c6f
177ea7cb74927651cf24b01dee27895d4f05fb5c161957845cd1b5848ed64ed3b0372
2b21a526a6e447cb8ee

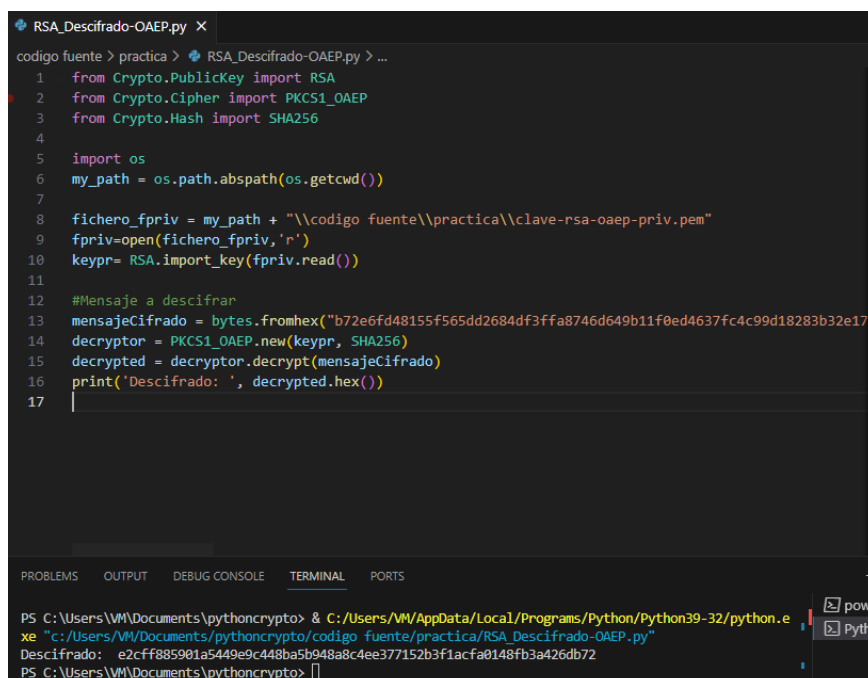
Las claves pública y privada las tenemos en los ficheros clave-rsa-oaep-publ.pem y clave-rsa-oaep-priv.pem.

La clave cifrada da:

e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

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

Si la ciframos y la volvemos a cifrar con el mismo algoritmo debido a la función *RSA.generate(2048)* va generando hashes de números aleatorios, por tanto las claves que genere serán totalmente diferentes.



```

RSA_Descifrado-OAEP.py x
codigo fuente > practica > RSA_Descifrado-OAEP.py > ...
1  from Crypto.PublicKey import RSA
2  from Crypto.Cipher import PKCS1_OAEP
3  from Crypto.Hash import SHA256
4
5  import os
6  my_path = os.path.abspath(os.getcwd())
7
8  fichero_fpriv = my_path + "\\codigo fuente\\practica\\clave-rsa-oaep-priv.pem"
9  fpriv=open(fichero_fpriv,'r')
10 keypr= RSA.import_key(fpriv.read())
11
12 #Mensaje a descifrar
13 mensajeCifrado = bytes.fromhex("b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e17
14 decryptor = PKCS1_OAEP.new(keypr, SHA256)
15 decrypted = decryptor.decrypt(mensajeCifrado)
16 print('Descifrado: ', decrypted.hex())
17 |

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\VM\Documents\pythoncrypto> & C:/Users/VM/AppData/Local/Programs/Python/Python39-32/python.exe "c:/Users/VM/Documents/pythoncrypto/codigo fuente/practica/RSA_Descifrado-OAEP.py"
Descifrado: e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72
PS C:\Users\VM\Documents\pythoncrypto>

```

El OAP va cambiando cada cifrado, va generando hashes de números aleatorios, para esa generación hay que indicarle un hash

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

Key: E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74

Nonce: 9Yccn/f5nJJhAt2S

¿Qué estamos haciendo mal?

No admite que nonce sea hexadecimal, por tanto habrá que decodificarlo en base64

Cifra el siguiente texto: He descubierto el error y no volveré a hacerlo mal

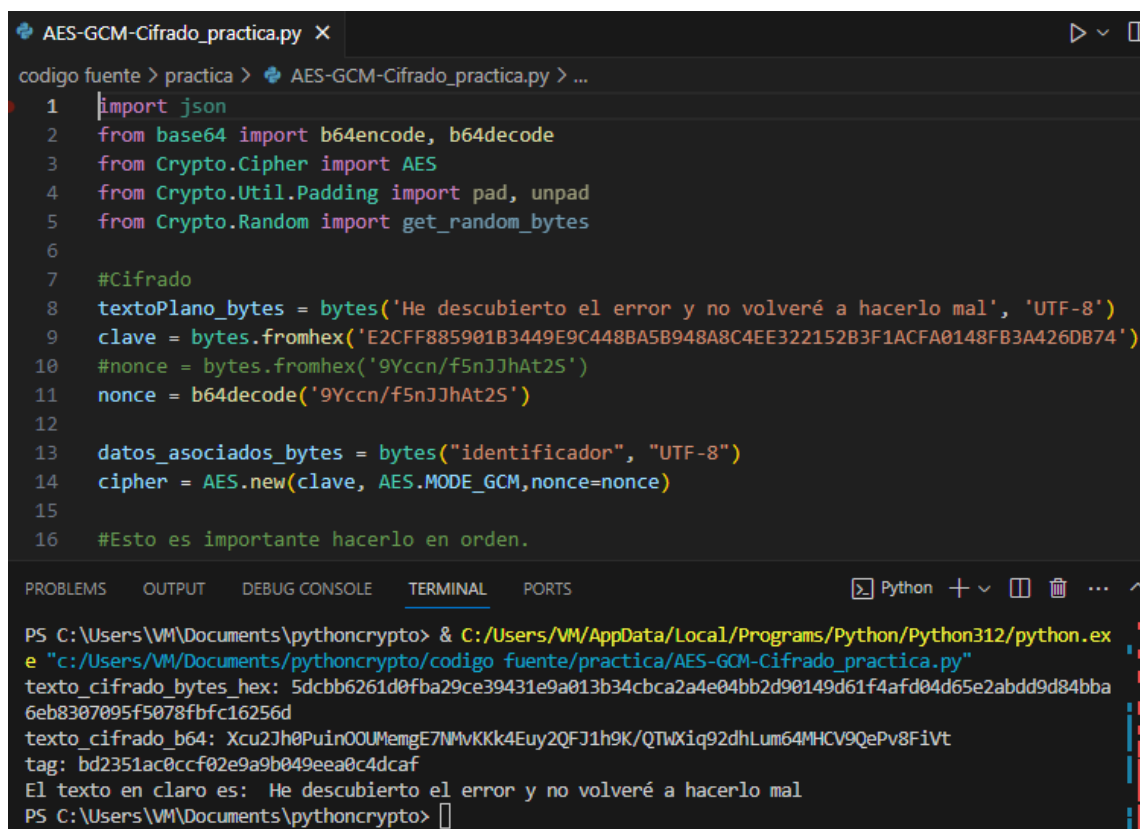
Usando para ello, la clave, y el nonce indicados. El texto cifrado presentalo en hexadecimal y en base64.

Texto cifrado en hexadecimal:

5dcb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d

Texto cifrado en base64

Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWxiq92dhLum64MHCv9QePv8FiVt



```
AES-GCM-Cifrado_practica.py X
codigo fuente > practica > AES-GCM-Cifrado_practica.py > ...
1 import json
2 from base64 import b64encode, b64decode
3 from Crypto.Cipher import AES
4 from Crypto.Util.Padding import pad, unpad
5 from Crypto.Random import get_random_bytes
6
7 #Cifrado
8 textoPlano_bytes = bytes('He descubierto el error y no volveré a hacerlo mal', 'UTF-8')
9 clave = bytes.fromhex('E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74')
10 #nonce = bytes.fromhex('9Yccn/f5nJJhAt2S')
11 nonce = b64decode('9Yccn/f5nJJhAt2S')
12
13 datos_asociados_bytes = bytes("identificador", "UTF-8")
14 cipher = AES.new(clave, AES.MODE_GCM, nonce=nonce)
15
16 #Esto es importante hacerlo en orden.

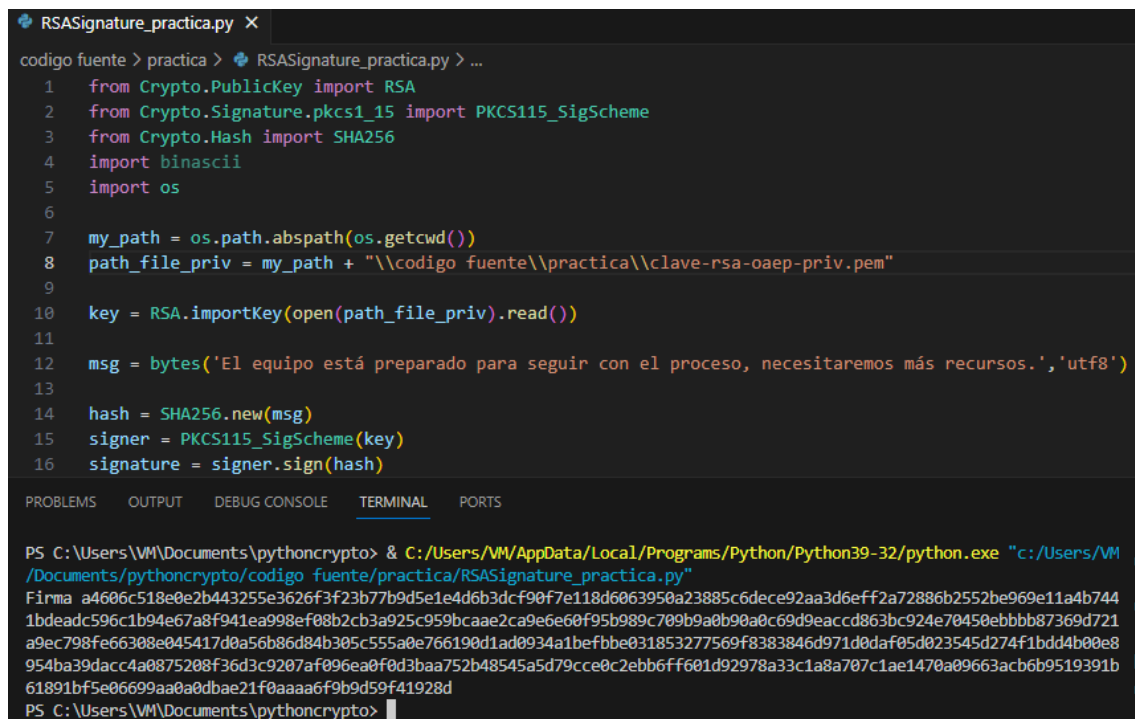
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python + - [ ] [ ] ... ^
PS C:\Users\VM\Documents\pythoncrypto> & C:/Users/VM/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/VM/Documents/pythoncrypto/codigo fuente/practica/AES-GCM-Cifrado_practica.py"
texto_cifrado_bytes_hex: 5dcb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d
texto_cifrado_b64: Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWxiq92dhLum64MHCv9QePv8FiVt
tag: bd2351ac0ccf02e9a9b049eea0c4dc4f
El texto en claro es: He descubierto el error y no volveré a hacerlo mal
PS C:\Users\VM\Documents\pythoncrypto>
```

13. Se desea calcular una firma con el algoritmo PKCS#1 v1.5 usando las claves contenidas en los ficheros clave-rsa-oeap-priv y clave-rsa-oeap-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?

a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721_oeap.pem'a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8/Documents/pythoncrypto/codigo fuente/practica/RSASignature_practica.py"954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b1bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a61891bf5e06699aa0a0dbae21f0aaaa6f9b9d59f41928d



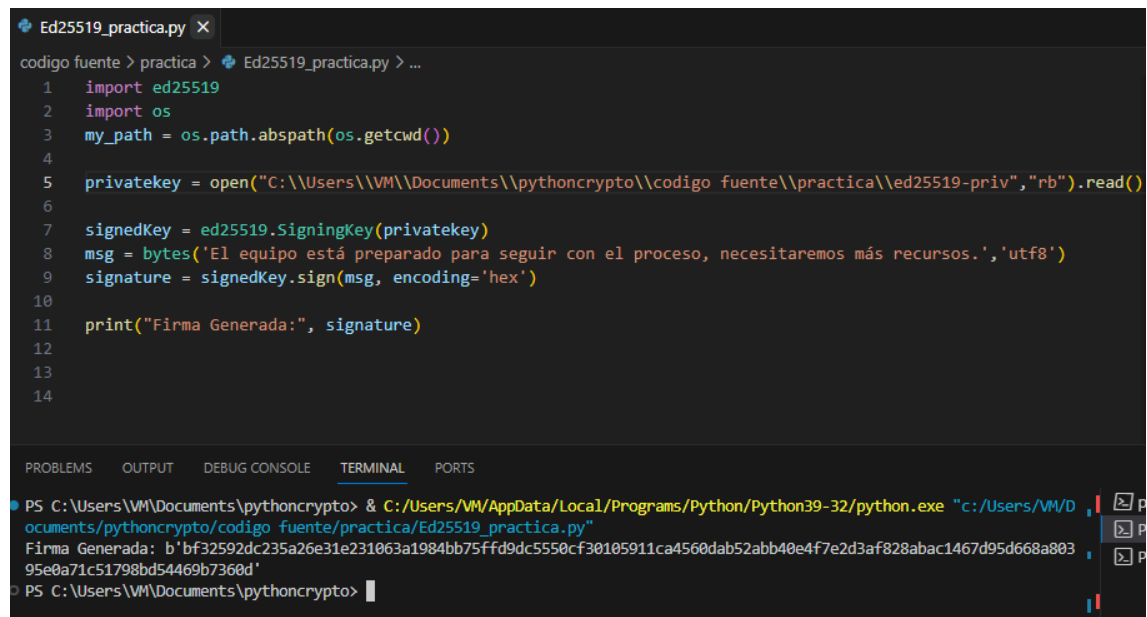
```
RSASignature_practica.py X
codigo fuente > practica > RSASignature_practica.py > ...
1 from Crypto.PublicKey import RSA
2 from Crypto.Signature.pkcs1_15 import PKCS115_SigScheme
3 from Crypto.Hash import SHA256
4 import binascii
5 import os
6
7 my_path = os.path.abspath(os.getcwd())
8 path_file_priv = my_path + "\\codigo fuente\\practica\\clave-rsa-oeap-priv.pem"
9
10 key = RSA.importKey(open(path_file_priv).read())
11
12 msg = bytes('El equipo está preparado para seguir con el proceso, necesitaremos más recursos.','utf8')
13
14 hash = SHA256.new(msg)
15 signer = PKCS115_SigScheme(key)
16 signature = signer.sign(hash)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\VM\Documents\pythoncrypto> & C:/Users/VM/AppData/Local/Programs/Python/Python39-32/python.exe "c:/Users/VM
/Documents/pythoncrypto/codigo fuente/practica/RSASignature_practica.py"
Firma a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b744
1bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721
a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8
954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b
61891bf5e06699aa0a0dbae21f0aaaa6f9b9d59f41928d
PS C:\Users\VM\Documents\pythoncrypto>
```

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

bf32592dc235a26e31e231063a1984bb75ffd9dc5550cf30105911ca4560dab52abb40e4f7e2d3af828abac1467d95d668a80395e95e0a71c51798bd54469b7360d



The image shows a screenshot of a Python IDE with a dark theme. The editor window displays a file named `Ed25519_practica.py` with the following code:

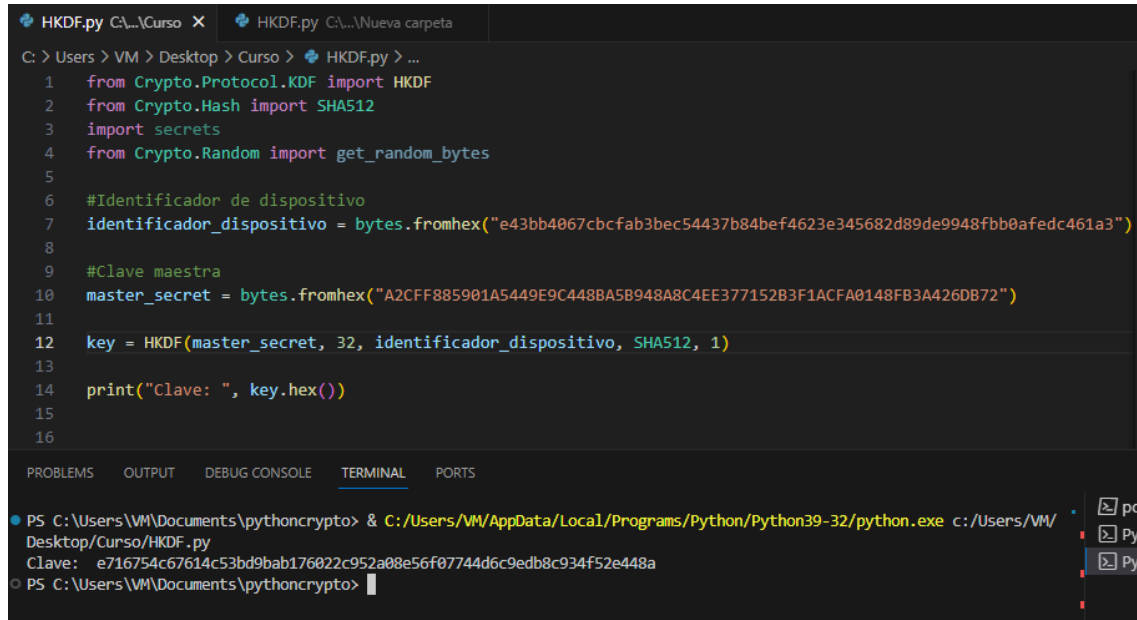
```
1 import ed25519
2 import os
3 my_path = os.path.abspath(os.getcwd())
4
5 privatekey = open("C:\\Users\\VM\\Documents\\pythoncrypto\\codigo fuente\\practica\\ed25519-priv", "rb").read()
6
7 signedKey = ed25519.SigningKey(privatekey)
8 msg = bytes('El equipo está preparado para seguir con el proceso, necesitaremos más recursos.', 'utf8')
9 signature = signedKey.sign(msg, encoding='hex')
10
11 print("Firma Generada:", signature)
12
13
14
```

Below the editor, the **TERMINAL** tab is active, showing the command to run the script and its output:

```
PS C:\Users\VM\Documents\pythoncrypto> & C:/Users/VM/AppData/Local/Programs/Python/Python39-32/python.exe "c:/Users/VM/Documents/pythoncrypto/codigo fuente/practica/Ed25519_practica.py"
Firma Generada: b'bf32592dc235a26e31e231063a1984bb75ffd9dc5550cf30105911ca4560dab52abb40e4f7e2d3af828abac1467d95d668a80395e0a71c51798bd54469b7360d'
PS C:\Users\VM\Documents\pythoncrypto>
```

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

e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3



```
HKDF.py C:\...Curso X HKDF.py C:\...Nueva carpeta
C: > Users > VM > Desktop > Curso > HKDF.py > ...
1  from Crypto.Protocol.KDF import HKDF
2  from Crypto.Hash import SHA512
3  import secrets
4  from Crypto.Random import get_random_bytes
5
6  #Identificador de dispositivo
7  identificador_dispositivo = bytes.fromhex("e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3")
8
9  #Clave maestra
10 master_secret = bytes.fromhex("A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72")
11
12 key = HKDF(master_secret, 32, identificador_dispositivo, SHA512, 1)
13
14 print("Clave: ", key.hex())
15
16
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\VM\Documents\pythoncrypto> & C:/Users/VM/AppData/Local/Programs/Python/Python39-32/python.exe c:/Users/VM/Desktop/Curso/HKDF.py
Clave: e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a
PS C:\Users\VM\Documents\pythoncrypto>
```

15. Nos envían un bloque TR31:

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDB
E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E0
3CD857FD37018E111B

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

A1A1010101010101010101010102

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

¿Para qué algoritmo se ha definido la clave? Algoritmo AES

¿Para qué modo de uso se ha generado? Para cifrar o descifrar

¿Es exportable? Es sensitiva, exportable si tiene una clave no confiable

¿Para qué se puede usar la clave? Encripta datos de manera simétrica, se usa para medios de pago o PIN

¿Qué valor tiene la clave? c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1

```
cabeceraR31_practica.py X
codigo fuente > Gestión de Claves > cabeceraR31_practica.py > ...
1 from psec import tr31
2
3 #Documentado en este fichero
4 https://github.com/knovichikhin/psec/blob/master/psec/tr31.py
5
6 header, key = tr31.unwrap( b'kbpc-bytes.fromhex("A1A1010101010101010101010101010102")', key_block='D0144D0AB00500004276689265B2DF93AE6E29B58135B77A2F616C8D0515ACDBEGA5626F79F7B407')
7 print(key.hex())
8
9 print("Key Version ID: " + header.version_id )
10 print("Algoritmo: " + header.algorithm)
11 print("Modo de uso: " + header.mode_of_use)
12 print("Uso de la clave: " + header.key_usage)
13 print("Exportabilidad: " + header.exportability)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
P5 C:\Users\NM\Documents\pythoncrypto> & C:/Users/NM/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/NM/Documents/pythoncrypto/codigo fuente/Gestión de Claves/cabeceraR31_practica.py"
ccccccccccccccccc
Key Version ID: D
Algoritmo: A
Modo de uso: B
Uso de la clave: 00
Exportabilidad: S
P5 C:\Users\NM\Documents\pythoncrypto>
```

Ficheros adjuntos practica

Enlace ficheros adjuntos practica