

Aclaración: He tratado de usar código lo máximo posible y evitar páginas como cyberchef que esta más automatizado, como no soy muy experimentado en pyhton me he basado en los códigos que nos proporcionaste y creado por inteligencia artificial depende de lo que necesitase ya que todavía no sabría hacerlo por mi mismo.

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

Operación básica de bits en la informática y la criptografía, usada en una gran mayoría de los algoritmos de criptografía en alguna forma.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Commutativa : $A \oplus B = B \oplus A$

Asociativa : $A \oplus (B \oplus C) = (A \oplus B) \oplus C$

Identidad : $A \oplus 0 = A$

Auto-inversa : $A \oplus A = 0$

$A \oplus 1 = !A$

XOR Calculator

Thanks for using the calculator. [View help page.](#)

I. Input: [hexadecimal \(base 16\)](#) ▼

b1ef2acfe2baeef

II. Input: [hexadecimal \(base 16\)](#) ▼

91ba13ba21aabb12

Calculate XOR

III. Output: [hexadecimal \(base 16\)](#) ▼

20553975c31055ed

[Home](#)

[Help](#)

[Privacy](#)

Utilizando las propiedades del XOR y con una calculadora xor conseguimos el valor que dio el key manager que es **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 B98A15BA31AEBB3F. ¿Qué clave será con la que se trabaje en memoria?

Utilizando la misma propiedad asociativa del XOR conseguimos la clave (en memoria) que sería **8653f75d31455c0**

XOR Calculator

Thanks for using the calculator. [View help page.](#)

I. Input: [hexadecimal \(base 16\)](#) ▼

b1ef2acfe2baeeff

II. Input: [hexadecimal \(base 16\)](#) ▼

b98a15ba31aebb3f

Calculate XOR

III. Output: [hexadecimal \(base 16\)](#) ▼

8653f75d31455c0

[Home](#)

[Help](#)

[Privacy](#)

2.Dada la clave

A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72.

El iv estará compuesto por el hexadecimal correspondiente a ceros binarios ("00"). Se requiere obtener el dato en claro correspondiente al siguiente dato cifrado:

TQ9SOMKc6aFS9S!xhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4
US t3aB/i50nvVjbBiG+le1ZhpR84ol=

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?

```

1 from Crypto.Cipher import AES
2 from Crypto.Util.Padding import unpad
3 import base64
4
5
6 clave_hex = "A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"
7 clave_bytes = bytes.fromhex(clave_hex)
8 iv_ceros = b'\x00' * 16
9
10
11 dato_cifrado_base64 = ""TQ9SOMKc6aFS9S1xhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIwXg3nu1RRz4QwElezdrLAD5LO4US t3aB/i50envvJbBiG+le1ZhpR84oI=""
12 dato_cifrado_bytes = base64.b64decode(dato_cifrado_base64)
13
14 cipher = AES.new(clave_bytes, AES.MODE_CBC, iv_ceros)
15
16
17 dato_descifrado = unpad(cipher.decrypt(dato_cifrado_bytes), AES.block_size)
18 print(dato_descifrado.decode('UTF-8'))
19

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Code

[Running] python -u "c:\Users\cayud\OneDrive\Escritorio\Nube\antiguo ord\Documents\Github de criptografia\cripto-main\tempCodeRunnerFile.python"
 Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. 💎nimo.

[Done] exited with code=0 in 0.157 seconds

Tras descifrarlo descubrimos el mensaje en su interior.

Al cambiar el padding a x923 se puede comprobar que se añadió 1 byte de padding.

```

17
18 cipher = AES.new(key, AES.MODE_CBC, iv)
19
20
21 plaintext_pkcs7 = cipher.decrypt(ciphertext)
22
23
24 padding_length_pkcs7 = plaintext_pkcs7[-1]
25 plaintext_pkcs7_clean = plaintext_pkcs7[:-padding_length_pkcs7]
26
27
28 print(plaintext_pkcs7_clean.decode('utf-8', errors='ignore'), padding_length_pkcs7)
29
30

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

[Running] python -u "c:\Users\cayud\OneDrive\Escritorio\Nube\antiguo ord\Documents\Github de criptog
 Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. 💎nimo. 1

3. Se requiere cifrar el texto "KeepCoding te enseña a codificar y a cifrar". La clave para ello, tiene la etiqueta en el Keystore "La etiqueta es cifrado-sim-chacha20-256". 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.

Tuve problemas con “jks” y tuve que utilizar otro código base que no utilizara este y salió lo siguiente.

Lo que nos da:

El nonce: 39f7f0bb982e81b8

El mensaje cifrado en hexadecimal:

8c67eaf16b047fcde83f3e64d914b9f8f09b2e3c36df75f6df4d5d98c3a98b13bc5791e88f23
207ab44ac1d9709bfe7889

```
1  from Crypto.Cipher import ChaCha20
2  from base64 import b64decode, b64encode
3  import os
4
5
6  clave = b'cifrado-sim-chacha20-256'
7
8  if len(clave) != 32:
9      raise ValueError("La clave debe tener 256 bits (32 bytes).")
10
11 textoPlano = bytes('KeepCoding te enseña a codificar y a cifrar', 'UTF-8')
12
13 nonce_mensaje = b64decode("9Yccn/f5nJJhAt2S")
14
15 print('nonce = ', nonce_mensaje.hex())
16
17 cipher = ChaCha20.new(key=clave, nonce=nonce_mensaje)
18
19 # Cifrar el texto
20 texto_cifrado = cipher.encrypt(textoPlano)
21
22 # Mostrar el texto cifrado en formato hexadecimal y Base64
23 print('Mensaje cifrado en HEX = ', texto_cifrado.hex())
24 print('Mensaje cifrado en B64 = ', b64encode(texto_cifrado).decode())
25
26
```

Nombre	Fecha de modificación	tipo	tamaño
cripto-main	19/12/2024 16:27	Carpeta de archivos	
AES-CBC.py	19/12/2024 16:27	Archivo PY	2 KB
La etiqueta es cifrado-sim-chacha20-256	21/12/2024 23:30	Archivo	0 KB

Para aumentar la seguridad e integridad del cifrado le añadiría un MAC ya que este estaría asegurando que nadie a modificado ningún dato del mensaje cifrado, en este caso yo lo hare por python con un código de base.

```

1 from Crypto.Cipher import ChaCha20
2 from Crypto.Hash import SHA256
3 from Crypto.Random import get_random_bytes
4 import hmac
5 from base64 import b64decode, b64encode
6
7 clave = get_random_bytes(32)
8
9 if len(clave) != 32:
10     raise ValueError("La clave debe tener 256 bits (32 bytes).")
11
12 textoPlano = bytes('KeepCoding te enseña a codificar y a cifrar', 'UTF-8')
13
14 nonce_mensaje = b64decode("9Yccn/f5nJJhAt2S")
15
16 print('nonce = ', nonce_mensaje.hex())
17
18 cipher = ChaCha20.new(key=clave, nonce=nonce_mensaje)
19
20 texto_cifrado = cipher.encrypt(textoPlano)
21
22 hmac_obj = hmac.new(clave, texto_cifrado, SHA256)
23 hmac_codigo = hmac_obj.digest()
24
25
26 print('Mensaje cifrado en HEX = ', texto_cifrado.hex())
27 print('Mensaje cifrado en B64 = ', b64encode(texto_cifrado).decode())
28 print('HMAC en HEX = ', hmac_codigo.hex())
29
30
31 # Verificación de HMAC
32 hmac_obj_recibido = hmac.new(clave, texto_cifrado, SHA256)
33 hmac_calculado = hmac_obj_recibido.digest()
34
35 if hmac_codigo == hmac_calculado:
36     print("La integridad del mensaje ha sido verificada correctamente.")

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Code

```

[Running] python -u "c:\Users\cayud\OneDrive\Escritorio\Nube\antiguo ord\Documents\Github de criptografia\cripto-main\tempCodeRunnerFile.python"
nonce = f5871c9ff7f99c926102dd92
mensaje cifrado en HEX = 391709e9fe4a63b9362274321dfb5cac972973b67dc45d52583af404269e3388d9a92cfad1bd9e56f1347dac
mensaje cifrado en B64 = ORCj6f5KY7k2InQyHftcrJcpc7Z9xf15MDr0BCaeMjZqSz60b2eVVe0faw=
HMAC en HEX = d50b0cdbf13560d94cd8f0b750bb2d5ad29cb4bc057775263d381531c800d444
La integridad del mensaje ha sido verificada correctamente.
mensaje descifrado: KeepCoding te enseña a codificar y a cifrar

[Done] exited with code=0 in 0.159 seconds

```

```
25
26 print('Mensaje cifrado en HEX = ', texto_cifrado.hex())
27 print('Mensaje cifrado en B64 = ', b64encode(texto_cifrado).decode())
28 print('HMAC en HEX = ', hmac_codigo.hex())
29
30
31 # Verificación de HMAC
32 hmac_obj_recibido = hmac.new(clave, texto_cifrado, SHA256)
33 hmac_calculado = hmac_obj_recibido.digest()
34
35 if hmac_codigo == hmac_calculado:
36     print("La integridad del mensaje ha sido verificada correctamente.")
37     # Si la integridad es correcta, descifrar el mensaje
38     cipher_descifrado = ChaCha20.new(key=clave, nonce=nonce_mensaje)
39     mensaje_descifrado = cipher_descifrado.decrypt(texto_cifrado)
40     print("Mensaje descifrado:", mensaje_descifrado.decode('utf-8'))
41 else:
42     print("El mensaje ha sido alterado o el HMAC es incorrecto.")
43
```

Revisamos el MAC

```
from Crypto.Cipher import ChaCha20
from Crypto.Hash import SHA256
import hmac
from base64 import b64decode

# Clave de 32 bytes (256 bits)
clave = bytes.fromhex('b9c901c070ae61fa5a3959eafc79a50fbdcfcbc726b46549b428f51385a5a9f2')

# Nonce proporcionado (decodificado de Base64 a bytes)
nonce_mensaje = b64decode("9Yccn/f5nJJhAt2S") # El nonce debe ser de 12 bytes

# Mensaje cifrado (en formato hexadecimal, decodificado de hexadecimal a bytes)
texto_cifrado = bytes.fromhex("f4bd3517a657d0f757e31ee35c82fcf7a7991e7d162d272e0e1926cb3978e7899211b65f493224a5578ee973")

# HMAC proporcionado (en formato hexadecimal, decodificado a bytes)
hmac_codigo = bytes.fromhex("a5c46353d8260ee0b3ee6f246ba057acd8b6f2469eb65ebf7c94db4c82c89076")

# Verificación de HMAC
hmac_obj_recibido = hmac.new(clave, texto_cifrado, SHA256)
hmac_calculado = hmac_obj_recibido.digest()

# Verificar integridad
if hmac_codigo == hmac_calculado:
    print("La integridad del mensaje ha sido verificada correctamente.")
    # Si la integridad es correcta, descifrar el mensaje
    cipher_descifrado = ChaCha20.new(key=clave, nonce=nonce_mensaje)
    mensaje_descifrado = cipher_descifrado.decrypt(texto_cifrado)
    print("Mensaje descifrado:", mensaje_descifrado.decode('utf-8'))
else:
    print("El mensaje ha sido alterado o el HMAC es incorrecto.")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Code

La integridad del mensaje ha sido verificada correctamente.
Mensaje descifrado: KeepCoding te enseña a codificar y a cifrar

[Done] exited with code=0 in 0.173 seconds

[Running] python -u "c:\Users\cayud\OneDrive\Escritorio\Nube\antiguo ord\Documents\Github de criptografia\cripto-main\tempCodeRunnerFile.python"

La integridad del mensaje ha sido verificada correctamente.
Mensaje descifrado: KeepCoding te enseña a codificar y a cifrar

[Done] exited with code=0 in 0.168 seconds

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFB1cGl0byBkZSBsb3MgcGFsb3RlcylsInJvbCI6ImIzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

¿Qué algoritmo de firma hemos realizado?

¿Cuál es el body del jwt?

Utiliza un algoritmo HS256.

El body del jwt es:

"usuario": "Don Pepito de los palotes",

"rol": "isNormal",


"iat": 1667933533

Claves ejercicio 10 - KeepCodin


1. Tenemos un sistema que usa

JSON Web Tokens - jwt.io

Get an exclusive look at jwt.io v2 and help us shape its final form with your feedback. →

JWT

DebuggerLibrariesIntroductionAsk

Crafted by  Auth0 by Okta

Warning: JWTs are credentials, which can grant access to resources. Be careful where you paste them! We do not record tokens, all validation and debugging is done on the client side.

AlgorithmHS256

EncodedPASTE A TOKEN HERE

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmIvIjoIIG9uIFB1cGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbmCI6ImIzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzZfQ.09x-JKRexCUD1vaxXi_JzTghWGXcuRrZYcpy7pHu-0I

DecodedEDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256"
}
```

PAYLOAD: DATA

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

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." + |
  base64UrlEncode(payload),
  )
```

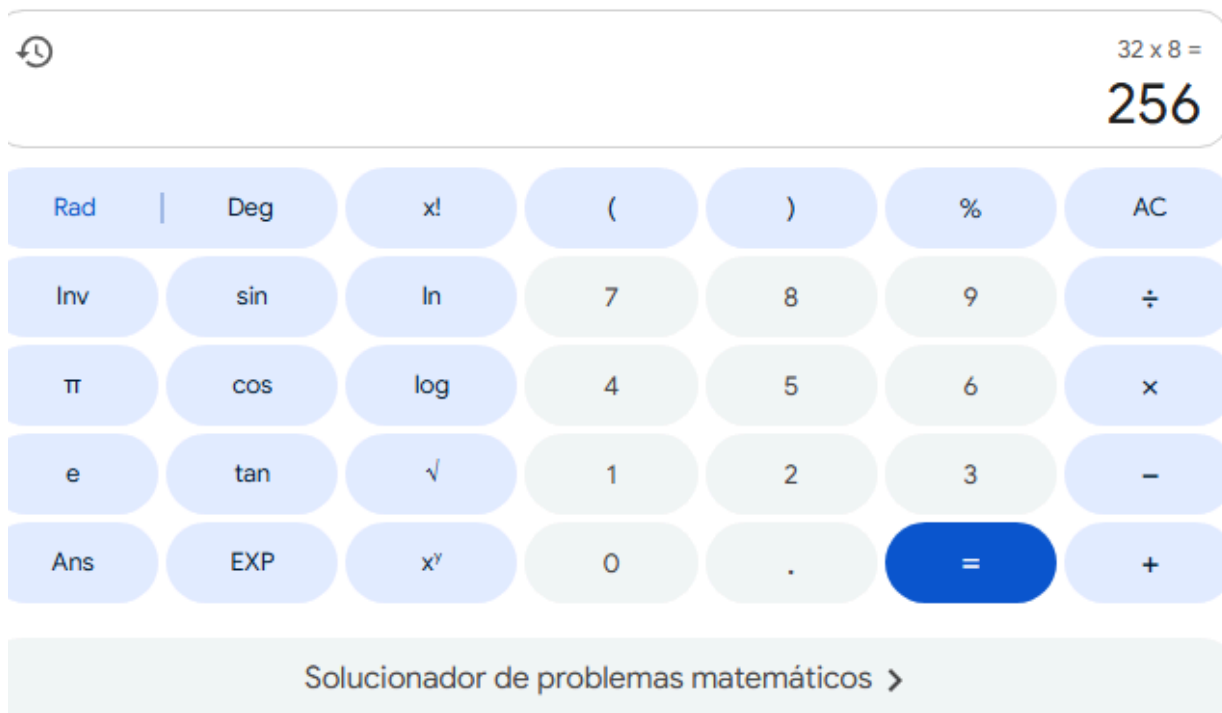
☐ secret base64 encoded

Un hacker está enviando a nuestro sistema el siguiente jwt:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmIvIjoIIG9uIFB1cGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbmCI6ImIzQWRtaW4iLCJpYXQiOiE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODIHRI

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

Si un hacker nos lo envía puede ser que este intentando un token malicioso o alterado, ya que al realizar el descryptado da error y sale como token invalido ya que podría haber sido modificado por el Mitm.



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

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

¿Qué hash hemos realizado?

Siguiendo la fórmula anterior descubrimos que el siguiente hash es un SHA2-512.

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

302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

Destacaría que al usar el mismo tipo de SHA3 y además el mismo mensaje este cambia totalmente.

```
1 from hashlib import sha3_256
2
3
4 texto = "En KeepCoding aprendemos cómo protegernos con criptografía."
5
6
7 hash_resultado = sha3_256(texto.encode()).hexdigest()
8 print(hash_resultado)
9
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

[Running] python -u "c:\Users\cayud\OneDrive\Escritorio\Nube\antiguo ord\Documents\Github de cr
302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

[Done] exited with code=0 in 0.083 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.

```
1 import hmac
2 import hashlib
3
4 clave_hex = "A212A51C997E14B4DF08D55967641B0677CA31E049E672A4B06861AA4D5826EB"
5 clave = bytes.fromhex(clave_hex) # Convertir clave a bytes
6
7
8 texto = "Siempre existe más de una forma de hacerlo, y más de una solución válida."
9
10 hmac_resultado = hmac.new(clave, texto.encode('utf-8'), hashlib.sha256).hexdigest()
11
12 print(hmac_resultado)
13
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

[Running] python -u "c:\Users\cayud\OneDrive\Escritorio\Nube\antiguo ord\Documents\Github de criptografia\c
857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550

[Done] exited with code=0 in 0.107 seconds

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?

Ya que es muy sencillo de desencriptar, yo recomendaría un SHA2 o 3 y además para más seguridad de 256.

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?

Para fortalecer yo añadiría un salt o hasta un pepper también.

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

Aplicaría lo mismo del salt y el pepper pero en un SHA3-256.

8.Tenemos la siguiente API REST, muy simple.

(Gráficos)

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?

Para empezar cambiaría los mensajes del sistema simétrico por un AES y con este intercambiaría las claves de manera asimétrica con un RSA en vez de utilizar un TLS. Además, habría que cifrar los datos sensibles de las personas como los números de tarjetas, y por último y de los puntos más destacables sería añadir un HMAC ya que este asegura la integridad de los mensajes intercambiados.

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.

De la clave se hashea en SHA-256 y AES, de la que podremos sacar el KCV.

```

1  from hashlib import sha256
2  from Crypto.Cipher import AES
3  from Crypto.Util.Padding import pad
4
5  clave_hex = "A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"
6  clave_bytes = bytes.fromhex(clave_hex)
7
8
9  hash_sha256 = sha256(clave_bytes).digest()
10 kcv_sha256 = hash_sha256[:3].hex()
11
12
13 bloque_ceros = b'\x00' * 16
14 iv_ceros = b'\x00' * 16
15
16
17 cipher = AES.new(clave_bytes, AES.MODE_CBC, iv_ceros)
18 cifrado = cipher.encrypt(pad(bloque_ceros, AES.block_size))
19
20
21 kcv_aes = cifrado[:3].hex()
22
23 print("El kcv con sha256 es:", kcv_sha256)
24 print(["Elkcv con aes es:", kcv_aes])

```

ROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Code

Running] python -u "c:\Users\cayud\OneDrive\Escritorio\Nube\antiguo ord\Documents\Github de criptografia\cripto-main\tempCodeRunnerFile.python"

l kcv con sha256 es: db7df2

Lkcv con aes es: 5244db

Done] exited with code=0 in 0.161 seconds

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.

Este ejercicio no paró de darme errores con las keys que se proporcionaron y de código.

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
df3c50e05d5bf63768088226e2a9177485c54f72407fdf358fe64479677d8296ad38c6f177e
a7cb74927651cf24b01dee27895d4f05fb5c161957845cd1b5848ed64ed3b0372
2b21a526a6e447cb8ee
```

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

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

Utilizo un código básico de descryptación con el RSA OAEP y con el sistema SHA-256 utilizando claves públicas y privadas que se proporcionaron.


```
32 oLjetp+Wb+8n/u60LNL85j52zG2uQq2/K3KVeSYrPF7uHdAdckMhKz9NB9WkWJk
33 ZzYV91I3DbwqF9N+bvW+oGZtHHKTbneSeoB+OEzoVzsys5RZ9fsMT3MwZQKBgAye
34 W/Kt+Kg1CBorPy2WHnxW28tm1HYXF5U8EH5L0St3dar0q7A16112UQqcBLHBVnZ/
35 ZAeodB/JoYNX+V5Gi0t3zSTiaHak02gCMRY7QJQBMMZpdonpSpw8v+1DM5jCvu4C
36 WPKRQ9A6WKFrKnqnURITbAXhAbtymMv57HtigZ/BAoGAdpmMRDQNKqai7aGbmbmF
37 Wy1GbLITkxWAOFSccQUYrFs8cuOGu79aB7PHwze0IHk/5ESj/gz7hoKJt0gi4ikx
38 zG2lYqqe11/Gg6wHendR1qR8VrbLBkpqylFTGusmLBuq7y4E/z9y2b4rMciU3OY2
39 X230g/Q6y6kMprauaCuxNSk=
40 -----END PRIVATE KEY-----"""
41
42 private_key = serialization.load_pem_private_key(
43     private_key_pem.encode(),
44     password=None,
45     backend=default_backend()
46 )
47
48 decrypted_key = private_key.decrypt(
49     ciphertext,
50     padding.OAEP(
51         mgf=padding.MGF1(algorithm=hashes.SHA256()),
52         algorithm=hashes.SHA256(),
53         label=None
54     )
55 )
56
57 print("Clave simétrica descifrada:")
58 print(binascii.hexlify(decrypted_key).decode())
59
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[Running] python -u "c:\Users\cayud\OneDrive\Escritorio\Nube\antiguo ord\Documents\Github de
Clave simétrica descifrada:
e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

[Done] exited with code=0 in 0.164 seconds
```

Se volvió a cifrar y efectivamente da diferente. Esto se debe al RSA-OAEP ya que este lo cifra con unos valores aleatorios dos veces añadiendo un padding con unos valores aleatorios que hacen que cada vez que se genere una clave aunque sea del mismo texto base dará diferente.

```
10 vtur8pHbYfcd3r8tZfZ3vSx1CbMb7YI7jRTflongratz0X7M43gTlns0R3xzyy
11 DSq+MWE0ZFpNE5qzdEdInsEppbqMgJP4U9gA1HZs62lai86rye+t0e/XDvkpro
12 KBBAXCpcSvsQhONr7VvTNHfZtmUhCYI1XlEcqWDisxciz5o5+nSmBTLg1FNSuDs
13 bQIDAQAB
14 -----END PUBLIC KEY-----
15
16
17
18 key_to_encrypt = b"e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72"
19
20
21 from cryptography.hazmat.primitives import serialization
22 public_key = serialization.load_pem_public_key(public_key_pem, backend=default_backend())
23
24
25 ciphertext = public_key.encrypt(
26     key_to_encrypt,
27     padding.OAEP(
28         mgf=padding.MGF1(algorithm=hashes.SHA256()),
29         algorithm=hashes.SHA256(),
30         label=None
31     )
32 )
33
34 print("Texto cifrado (hex):")
35 print(ciphertext.hex())
36
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Code

[Running] python -u "c:\Users\cayud\OneDrive\Escritorio\Wube\antiguo ord\Documents\Github de criptografia\cripto-main\tempCodeRunnerFile.python"

Texto cifrado (hex):

3f20957b268304e8f113d1576cb943acb41f30afe5e80029dadf59fa77d3ac456e444c4777437a7ceb7ca2f5f7ee7033594851c74e848593966651c0a9213bf8a8311f9142a9fc632b9a0162fb94b353ebabb83d57213b01a3a7f502b8aaa7f4ea31bb2a043deaa17f9812a7f503e9b385d454df3a87ca735c1a0e9ee625a81da7e1735cc6e656ce2210e78a9c68125557d774efb335712cd679c072b35d7647f26edddfc7550a2a9eb9b8624913f3362a8e065c0c3190c231668a9f6e0934c5d991df1c3ae3bc9026e94aee07c6be45a747f3238b08f56a107a726e1ea8803b5eae0e0bd0e734a885f8e7023114b4cc0875d9652a042929455a6f5f3256fc6

[Done] exited with code=0 in 0.15 seconds

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

Key:E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A42 6DB74
Nonce:9Yccn/f5nJjhAt2S

¿Qué estamos haciendo mal?

Que el Nonce debe ser de uso único y no debería compartirse ya que compromete la seguridad y veracidad del mensaje cifrado.

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.

Como ya se ha hecho anteriormente damos unos valores a las variables key, nonce y el texto a cifrar, se pasan a bytes y realizamos el cifrador.

```

1  from Crypto.Cipher import AES
2  from Crypto.Util.Padding import pad
3  from Crypto.Random import get_random_bytes
4  import base64
5
6
7  key_hex = "E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74"
8  nonce_b64 = "9Yccn/f5nJJhAt2S"
9  plaintext = "He descubierto el error y no volveré a hacerlo mal"
10
11 key = bytes.fromhex(key_hex.replace(" ", ""))
12
13
14 nonce = base64.b64decode(nonce_b64)
15
16 cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)
17
18 ciphertext, tag = cipher.encrypt_and_digest(pad(plaintext.encode(), AES.block_size))
19
20 ciphertext_hex = ciphertext.hex()
21 ciphertext_base64 = base64.b64encode(ciphertext).decode()
22
23 print(ciphertext_hex)
24 print(ciphertext_base64)
25

```

ROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Running] python -u "c:\Users\cayud\OneDrive\Escritorio\Nube\antiguo ord\Documents\Github de criptografia\cripto-main\tempCodeRunnerFil
5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256df06dbffcfb38008ff2ca76b5f5
Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWxiq92dhLum64MHCV9QePv8FiVt8G2//Ps4AI/yyna19Q==

Done] exited with code=0 in 0.159 seconds

La clave en:

Hex:

5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256df06dbffcfb38008ff2ca76b5f5

Base 64:

Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWxiq92dhLum64MHCV9QePv8FiVt8G2//Ps4AI/yyna19Q==

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?

Utilizamos el visual estudio para usar un código base de firma en hexadecimal con PKCS#1 que nos da:

a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21f0aaaa6f9b9d59f41928d

```
5
6  mensaje = "El equipo está preparado para seguir con el proceso, necesitaremos más recursos."
7
8  private_key_pem = '''-----BEGIN PRIVATE KEY-----
9  MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBAcwggSjAgEAAoIBAQC/absrLf79T7cz
10 tzjt/hHGJ+2LTBrZ90mJqVTwCtLU5xCd9heIf0iVQ+ZFZH5a1ewI3Q5hPA16R/Ad
11 g63clqWY4iRp4Jzt84GGw2XeLURQ60VNXlufQt1aC9oU0Qi1YksI1+LqNa6y5K0w
12 HqZFkoq+25EGkdN9h9zAPevy1kVne/lfUJsxvtgjuNFN+WieCtq3M5fszjeBM2ew
13 5HFHPLINKr5YpYTRkU80TmrN0R0iewSmlupaAk/hSL2ADUdmzraVqLzqvJ763R79
14 cO+SmugoEEDEKLxK+xCE42vu9W00d9m2ZSEJgiVeV5yDco0KzFyJnmhL6dKYFMuD
15 UU1K40xtAgMBAAECggEAPqkQGoyOIIsKLYkQ8QLyheOrtOmKJj70CF8yU/5ereQLD
16 T9KV3xjK0sJNiX3iVz4cbLJg2LfD+Z+/HQPUShg00c0GBBR/Y7MJPeKNYHQVHyBF
17 qbY7nCE5cRbcJ2Bep3Ir+hMiN2WncOykIS2HZNMafGywRyRMaUKGo3Ah43b9dWhx
18 RYhLee8CD1c9I1lkRZ2UycmeJdgWe+CmUOIWH87r0FBcqVI+6z1Kmk2IRh/HfVp
19 v646kOwKBF3XPT4YFjX+t2JSeLSaQbRQ+aVq3Twyz354GvPvaVsON91FsToQbj+1
20 f0JRzleWz/CU1X1bhnTvd8TCMv8+hPzf3weyOeTcgQKBgQDdm6wdu54/B/pfQ1UX
21 T3cyYHFKxDj5S/s96H1LcuUR89Sn7LQwwkkZYwki/osm9B5e54/rbSop/1+beCoO
22 0xKpSozc8/xms3ulbwpDxR3+xTMj9vAGjjp2hw5vy1TH6o7Kyq8kYyPjmGnqNpCf
23 ANv7mfDl8Jvw/kIAEpWxEo6+HQBKbQDdHm4jLWek8PZbBUmZajzzf1MddXMKX50u
24 dzhKOF2W57WutJpYRbg4/sz3Ty6qtulDexuWnw+feEkroq8cMfBB7FQaQPtq3nac
25 coWktSEUG7Tz1RQ318kslVWJ3Y93iSaoMtrThcaa18+FmVG3SwBefl0uEX8SpAVg
26 1iP0+pfWkQKBgQCTDDwuUpOT4ZhaY2qRGDLQ47vpT8E6cxeYocyqp+jxPcEIoyC
27 oLjetp+Wb+8n/u60LNWL85j52zG2uQq2/K3KVeSYrPF7uHdAdCkMhRz9NB9WKWJk
28 ZzYV9lI3Dbwqf9N+bvW+oGZtHHKTbneSeoB+OEzoVzsys5RZ9fsMT3MwZQKBgAye
29 W/Kt+Kg1CBoRpy2WHnxW28tmlHYXFsU8EH5L0St3darOq7A16l12UQQcBLHBVnZ/
30 ZAEodb/JoYNX+V5Gi0t3zStiaHak02gCMRY7QJQBMMZpdonpSpw8v+1DM5jCvu4C
31 WPKRQ9A6WKFrKnqnURITbAXhAbtymMv57HtigZ/BAoGAdpmMRDQNKqai7aGbmbmF
32 Wy1GbLITkxWAOFScQQUYrFs8cu0Gu79aB7PHwzeOIHk/5ESj/gz7hoKJtOgi4ikx
33 zG2lYqqe11/Gg6wHendR1qR8VrbLBkpqylFTGusmlBuq7y4E/z9y2b4rMciU30Y2
34 X230g/Q6y6kMprauaCuxNSk=
35 -----END PRIVATE KEY-----'''
36
37 private_key = RSA.import_key(private_key_pem)
38
39 h = SHA256.new(mensaje.encode())
40
41 signature = pkcs1_15.new(private_key).sign(h)
42
43 signature_hex = signature.hex()
44
45 print(f"Firma en hexadecimal: {signature_hex}")
46
```

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

“A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72”. La clave obtenida dependerá de un identificador de dispositivo, en este caso tendrá el valor en hexadecimal:

e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3

¿Qué clave se ha obtenido?

La clave que se ha obtenido en hexadecimal es:

e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a

```
1  from cryptography.hazmat.primitives import hashes
2  from cryptography.hazmat.primitives.kdf.hkdf import HKDF
3  from cryptography.hazmat.backends import default_backend
4
5  master_key_hex = "A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"
6  master_key = bytes.fromhex(master_key_hex)
7
8  device_id_hex = "e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3"
9  device_id = bytes.fromhex(device_id_hex)
10
11  hkdf = HKDF(
12      algorithm=hashes.SHA512(),
13      length=32, # AES-256: 32 bytes
14      salt=device_id,
15      info=None,
16      backend=default_backend()
17  )
18
19  derived_key = hkdf.derive(master_key)
20
21
22  print("Clave AES derivada (hex):", derived_key.hex())
23
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

[Running] python -u "c:\Users\cayud\OneDrive\Escritorio\Nube\antiguo ord\Documents\Github de criptografia\cripto-main\tempC
Clave AES derivada (hex): e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a

15. Nos envían un bloque TR31:

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDB
E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857F
D37018E111B

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

A1A10101010101010101010101010102

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

Debido a la clave de transporte que es de 24 bytes que generalmente se usa en 3DES pero no se puede averiguar con certeza si es ECB o CBC siendo esta primera muy mala opción ya que se puede romper sin mucha dificultad.

¿Para qué algoritmo se ha definido la clave?

EL TR31 se utiliza para algoritmos simétricos pudiendo ser el algoritmo de cifrado en bloque AES o DES.

¿Para qué modo de uso se ha generado?

No se puede saber con seguridad pero generalmente se utiliza para encriptación de claves.

¿Es exportable?

Al ser una clave de un sistema 3DES en principio no debería, dependerá de a quién y como se exportaría según la configuración.

¿Para qué se puede usar la clave?

Al igual que en otros sistemas para encriptar o desencriptar datos, mensajes,...

¿Qué valor tiene la clave?

A1A1010101010101010101010101010102 se utiliza para deshacer el bloque y conseguir la clave contenida dentro de la misma.