PRÁCTICA: CRIPTOGRAFÍA

Objetivo:

Contestar un conjunto de preguntas / ejercicios relacionados con la materia aprendida en el curso demostrando la adquisición de conocimientos relacionados con la criptografía.

Detalles:

En esta práctica el alumno aplicará las técnicas y utilizará las diferentes herramientas vistas durante el módulo.

Cualquier password que sea necesaria tendrá un valor 123456.

Evaluación

Es obligatorio la entrega de un informe para considerar como APTA la práctica. Este informe ha de contener:

- Los enunciados seguido de las respuestas justificadas y evidenciadas.
- En el caso de que se hayan usado comandos / herramientas también se deben nombrar y explicar los pasos realizados.

El código escrito para la resolución de los problemas se entrega en archivos separados junto al informe.

Se va a valorar el proceso de razonamiento aunque no se llegue a resolver completamente los problemas. Si el código no funciona, pero se explica detalladamente la intención se valorará positivamente.

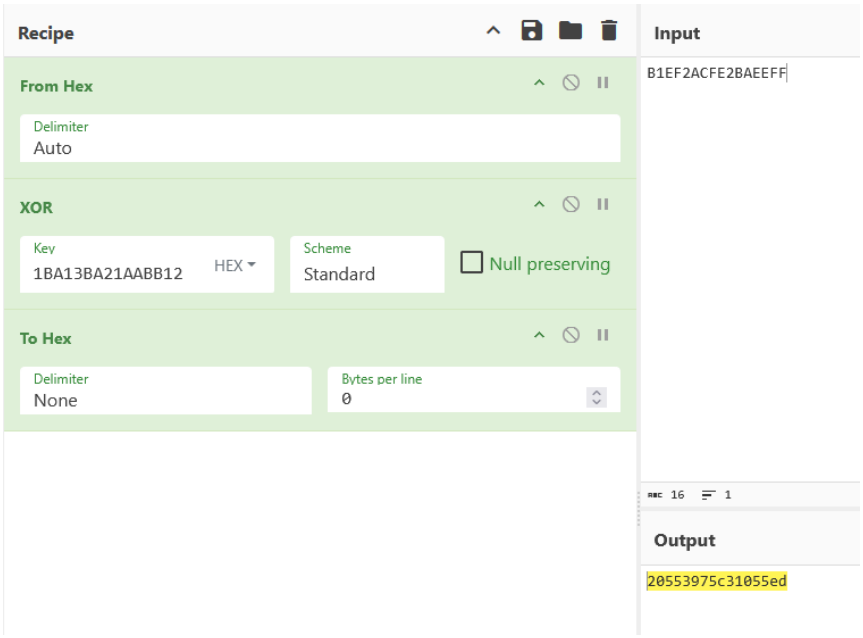
El objetivo principal de este módulo es adquirir conocimientos de criptografía y por ello se considera fundamental usar cualquier herramienta que pueda ayudar a su resolución, demostrando que no sólo se obtiene el dato sino que se tiene un conocimiento profundo del mismo. Si durante la misma no se indica claramente la necesidad de resolverlo usando programación, el alumno será libre de usar cualquier herramienta, siempre y cuando aporte las evidencias oportunas.

Ejercicios:

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?

   ➔ Resultado: 20553975c31055ed
   ➔ Ejecución:

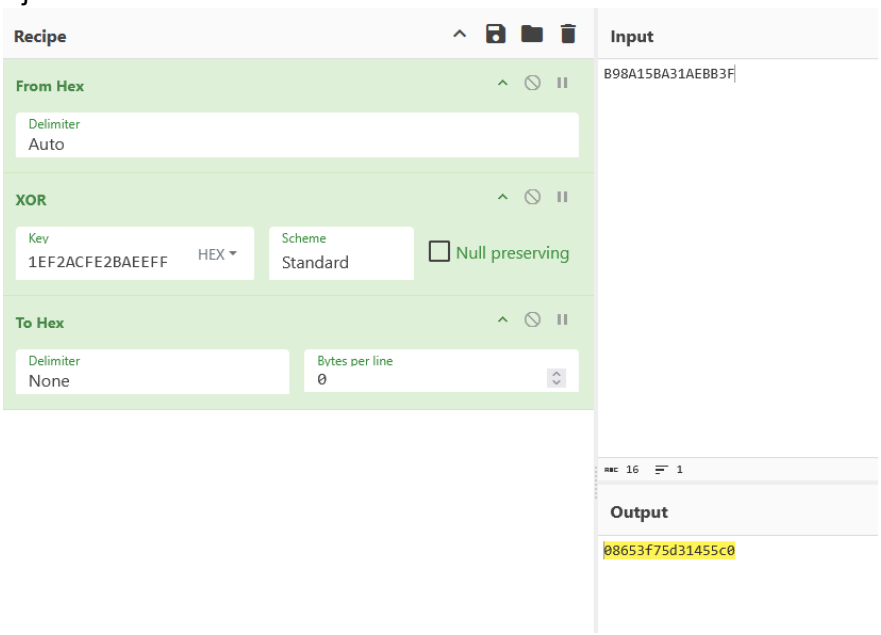



```python
# Ejercicio 1 de Ejercicios de Criptografia Finales

#XOR de datos binarios
def xor_data(binary_data_1, binary_data_2):
    return bytes([b1 ^ b2 for b1, b2 in zip(binary_data_1, binary_data_2)])


m = bytes.fromhex("B1EF2ACFE2BAEEFF")
k = bytes.fromhex("91BA13BA21AABB12")

print(xor_data(m,k).hex())

num1=0xB1EF2ACFE2BAEEFF
num2=0x91BA13BA21AABB12
num3=(hex(num1^num2))
print(num3[2:])

```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

[Running] python -u "c:\Users\d_gar\Desktop\GitHub Cripto\criptografia\tempCodeRunnerFile.python"
20553975c31055ed
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?

➔ Resultado: 08653f75d31455c0
➔ Ejecución:



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?

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

➔ Código usado manualmente:

```python
# Ejercicio 2 Criptografía

from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
import base64

# CLAVE MANUAL
CLAVE_HEX = "A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"
clave_bytes = bytes.fromhex(CLAVE_HEX)

# Descifrado
try:
    cifrado_base64 = "TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4US t3aB/i50nvvJbBiG+le1ZhpR84oI="
    cifrado_base64 = cifrado_base64.replace(" ", "")

    iv = b'\x00' * 16
    texto_cifrado_bytes = base64.b64decode(cifrado_base64)

    cipher = AES.new(clave_bytes, AES.MODE_CBC, iv)
    mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size)

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

except Exception as error:
    print('Error:', error)

# Analisis Padding
longitud_texto_claro = len(mensaje_des_bytes)
bloques = (longitud_texto_claro + 15) // 16  # división hacia arriba
padding = 16 - (longitud_texto_claro % 16)
print(f"Longitud texto claro: {longitud_texto_claro} bytes")
print(f"Padding PKCS7 añadido: {padding} bytes")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
[Running] python -u "c:\Users\d_gar\Desktop\GitHub Cripto\criptografía\tempCodeRunnerFile.python"
El texto en claro es: Esto es un cifrado en bloque t□pico. Recuerda, vas por el buen camino. □nimo.
Longitud texto claro: 79 bytes
Padding PKCS7 a□adido: 1 bytes
```

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

➔ En este caso como el padding es solo 1 byte el resultado es el mismo.

¿Cuánto padding se ha añadido en el cifrado?

→ Longitud texto claro: 79 bytes

➔ Padding PKCS7 añadido: 1 bytes

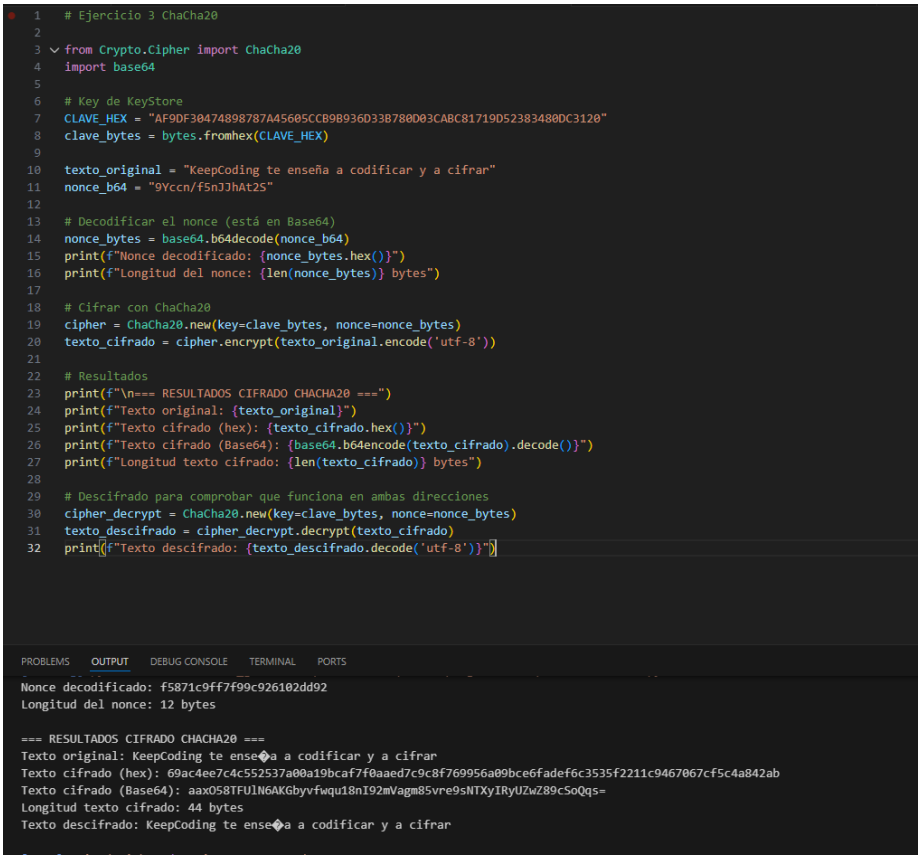
➔ Para resolverlo he añadido esto al final de mi comando original:

Analisis Padding
➔ longitud_texto_claro = len(mensaje_des_bytes)
➔ bloques = (longitud_texto_claro + 15) // 16 # división hacia arriba
➔ padding = 16 - (longitud_texto_claro % 16)
➔ print(f"Longitud texto claro: {longitud_texto_claro} bytes")
➔ print(f"Padding PKCS7 añadido: {padding} bytes")

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

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.

➔ Primero he comprobado que el ChaCha20 de manera normal funciona bien:



```python
# Ejercicio 3 ChaCha20

from Crypto.Cipher import ChaCha20
import base64

# Key de KeyStore
CLAVE_HEX = "AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120"
clave_bytes = bytes.fromhex(CLAVE_HEX)

texto_original = "KeepCoding te enseña a codificar y a cifrar"
nonce_b64 = "9Yccn/f5nJJhAt2S"

# Decodificar el nonce (está en Base64)
nonce_bytes = base64.b64decode(nonce_b64)
print(f"Nonce decodificado: {nonce_bytes.hex()}")
print(f"Longitud del nonce: {len(nonce_bytes)} bytes")

# Cifrar con ChaCha20
cipher = ChaCha20.new(key=clave_bytes, nonce=nonce_bytes)
texto_cifrado = cipher.encrypt(texto_original.encode('utf-8'))

# Resultados
print(f"\n=== RESULTADOS CIFRADO CHACHA20 ===")
print(f"Texto original: {texto_original}")
print(f"Texto cifrado (hex): {texto_cifrado.hex()}")
print(f"Texto cifrado (Base64): {base64.b64encode(texto_cifrado).decode()}")
print(f"Longitud texto cifrado: {len(texto_cifrado)} bytes")

# Descifrado para comprobar que funciona en ambas direcciones
cipher_decrypt = ChaCha20.new(key=clave_bytes, nonce=nonce_bytes)
texto_descifrado = cipher_decrypt.decrypt(texto_cifrado)
print(f"Texto descifrado: {texto_descifrado.decode('utf-8')}")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
Nonce decodificado: f5871c9ff7f99c926102dd92
Longitud del nonce: 12 bytes

=== RESULTADOS CIFRADO CHACHA20 ===
Texto original: KeepCoding te ense□a a codificar y a cifrar
Texto cifrado (hex): 69ac4ee7c4c552537a00a19bcaf7f0aaed7c9c8f769956a09bce6fadef6c3535f2211c9467067cf5c4a842ab
Texto cifrado (Base64): aaxO58TFUlN6AKGbyvfwqu18nI92mVagm85vre9sNTXyIRyUZwZ89cSoQqs=
Longitud texto cifrado: 44 bytes
Texto descifrado: KeepCoding te ense□a a codificar y a cifrar
```

¿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.

➔ La manera de mejorarlo sería usando el ChaCha20-Poly1305, ya que Poly1305 añade un tag de autenticación que verifica integridad.
➔ Este es el código que he empleado:

```python
# Ejercicio 3 Chacha20 Poly305

from Crypto.Cipher import ChaCha20_Poly1305
import base64

# Key de KeyStore
CLAVE_HEX = "AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120"
clave_bytes = bytes.fromhex(CLAVE_HEX)

texto_original = "KeepCoding te enseña a codificar y a cifrar"
nonce_b64 = "9Yccn/f5nJJhAt2S"

# Nonce
nonce_bytes = base64.b64decode(nonce_b64)
print(f"Nonce: {nonce_b64} -> {nonce_bytes.hex()} ({len(nonce_bytes)} bytes)")

# En esta parte cifra y autentica
cipher = ChaCha20_Poly1305.new(key=clave_bytes, nonce=nonce_bytes)
texto_cifrado, tag = cipher.encrypt_and_digest(texto_original.encode('utf-8'))

print(f"\n CIFRADO COMPLETADO:")
print(f"Texto cifrado (Base64): {base64.b64encode(texto_cifrado).decode()}")
print(f"Tag autenticación (hex): {tag.hex()}")
print(f"Nonce (Base64): {nonce_b64}")


# Descifrado
try:
    cipher_verif = ChaCha20_Poly1305.new(key=clave_bytes, nonce=nonce_bytes)
    texto_descifrado = cipher_verif.decrypt_and_verify(texto_cifrado, tag)
    print(f"\n\n Comprobacion desencriptado: {texto_descifrado.decode('utf-8')}")
except ValueError as e:
    print(f"Error desencriptado: {e}")


```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

[Running] python -u "c:\Users\d_gar\Desktop\GitHub Cripto\criptografia\tempCodeRunnerFile.python"
Nonce: 9Yccn/f5nJJhAt2S -> f5871c9ff7f99c926102dd92 (12 bytes)

 CIFRADO COMPLETADO:
Texto cifrado (Base64): TslZIcqLdX4jNmBcfbq49NQLW00iDmaql490DT5ZsM1w4yFyQpkcwUC7Hho=
Tag autenticaci▯n (hex): 710cd4723da6d5ef37f23bee66285e57
Nonce (Base64): 9Yccn/f5nJJhAt2S


 Comprobacion desencriptado: KeepCoding te ense▯a a codificar y a cifrar
```

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0by BkZSB

sb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ .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
.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODlHRI
```

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

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:

```
4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f
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?

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.

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?

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?

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.

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 Pedropriv.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.

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
793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae2c4f5ba670a4126f2f14
a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8624071be78ccef573d
896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f67d3f98b4cd907f27639f4b1
df3c50e05d5bf63768088226e2a9177485c54f72407fdf358fe64479677d8296ad38c6f
177ea7cb74927651cf24b01dee27895d4f05fb5c161957845cd1b5848ed64ed3b0372
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?

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?

➔ Inicialmente, según el texto, expone que la empresa esta "usando estos mismos datos en cada comunicación", el problema viene en que el nonce debe ser aleatorio y no usarse en mas de una ocasión, ya que si se usa más veces podría ser vulnerable si se descubre.

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.

➔ Este es el cifrado que se pide con sus variantes, como siempre incluyo al final el descifrado para comprobar que está funcionando correctamente:

```python
# Ejercicio 12 - AES/GCM

from Crypto.Cipher import AES
import base64

# Datos que vamos a usar
CLAVE_HEX = "E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74"
clave_bytes = bytes.fromhex(CLAVE_HEX)

nonce_b64 = "9Yccn/f5nJJhAt2S"
nonce_bytes = base64.b64decode(nonce_b64)

texto_original = "He descubierto el error y no volveré a hacerlo mal"

print("=== CIFRADO AES/GCM ===")
print(f"Key: {CLAVE_HEX}")
print(f"Nonce: {nonce_b64} -> {nonce_bytes.hex()} ({len(nonce_bytes)} bytes)")
print(f"Texto original: {texto_original}")

# Cifrado GCM
cipher = AES.new(clave_bytes, AES.MODE_GCM, nonce=nonce_bytes)
texto_cifrado, tag = cipher.encrypt_and_digest(texto_original.encode('utf-8'))

print(f"\n RESULTADOS:")
print(f"Texto cifrado (hex): {texto_cifrado.hex()}")
print(f"Texto cifrado (Base64): {base64.b64encode(texto_cifrado).decode()}")
print(f"Tag autenticación (hex): {tag.hex()}")

# Descifrado para verificar que funciona a la inversa
try:
    cipher_verif = AES.new(clave_bytes, AES.MODE_GCM, nonce=nonce_bytes)
    texto_descifrado = cipher_verif.decrypt_and_verify(texto_cifrado, tag)
    print(f"\n Descifrado funciona: {texto_descifrado.decode('utf-8')}")
except Exception as e:
    print(f"\n [!]Error: {e}")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
[Running] python -u "c:\Users\d_gar\Desktop\GitHub Cripto\criptografia\tempCodeRunnerFile.python"
=== CIFRADO AES/GCM ===
Key: E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74
Nonce: 9Yccn/f5nJJhAt2S -> f5871c9ff7f99c926102dd92 (12 bytes)
Texto original: He descubierto el error y no volver a hacerlo mal

 RESULTADOS:
Texto cifrado (hex): 5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d
Texto cifrado (Base64): Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq92dhLum64MHCV9QePv8FiVt
Tag autenticaci n (hex): 6120e37aa4c3ecfd9261640dcc46410d

 Descifrado funciona: He descubierto el error y no volver a hacerlo mal
```

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.

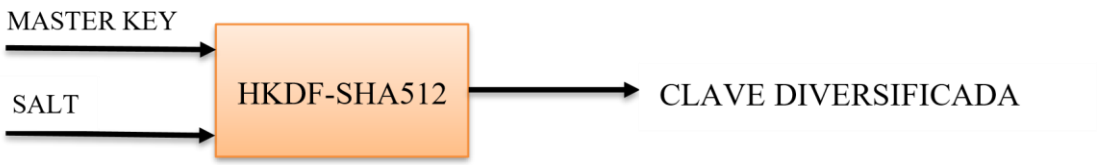
¿Cuál es el valor de la firma en hexadecimal?

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

14. Necesitamos generar una nueva clave AES, usando para ello una HKDF (HMACbased Extractand-Expand key derivation function) con un hash SHA-512. La

clave maestra requerida se encuentra en el keystore con la etiqueta "cifrado-sim-aes-256". La clave obtenida dependerá de un identificador de dispositivo, en este caso tendrá el valor en hexadecimal:

```
e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3
```



¿Qué clave se ha obtenido?

15. Nos envían un bloque TR31:

```
D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDB
E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E0
3CD857FD37018E111B
```

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

```
A1A10101010101010101010101010102
```

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

➔ AES Key Derivation Binding Method

¿Para qué algoritmo se ha definido la clave?

➔ AES

¿Para qué modo de uso se ha generado?

➔ Cifrado y descifrado

¿Es exportable?

➔ Sí, es exportable bajo clave no confiable

¿Para qué se puede usar la clave?

➔ **D0** = Data Encryption Key (Generic) | Clave de cifrado de datos genérica

¿Qué valor tiene la clave?

➔ Clave desempaquetada = c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1