

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 necesario 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 seguidos 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?

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?

```
def hex_to_bytes(hex_str):
    return bytes.fromhex(hex_str)

def bytes_to_hex(byte_data):
    return byte_data.hex().upper()

def xor_bytes(bytes1, bytes2):
    return bytes(a ^ b for a, b in zip(bytes1, bytes2))

# Claves en hexadecimal
clave_fija = "B1EF2ACFE2BAEEFF"
clave_final_desarollo = "91BA13BA21AABB12"
clave_fichero_produccion = "B98A15BA31AE3B3F"

# Convertir las claves de hexadecimal a bytes
clave_fija_bytes = hex_to_bytes(clave_fija)
clave_final_desarollo_bytes = hex_to_bytes(clave_final_desarollo)
clave_fichero_produccion_bytes = hex_to_bytes(clave_fichero_produccion)

# Calcular la clave del fichero de propiedades en desarrollo
clave_fichero_desarollo_bytes = xor_bytes(clave_fija_bytes,
clave_final_desarollo_bytes)
clave_fichero_desarollo = bytes_to_hex(clave_fichero_desarollo_bytes)

# Calcular la clave final en memoria en producción
clave_final_produccion_bytes = xor_bytes(clave_fija_bytes,
clave_fichero_produccion_bytes)
clave_final_produccion = bytes_to_hex(clave_final_produccion_bytes)

print(clave_fichero_desarollo, "clave fichero de desarrollo")
print(clave_final_produccion, "clave final de produccion")
```

20553975C31055ED clave fichero de desarrollo

08653F75D31455C0 clave **final** de producción

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=

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

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

# Datos proporcionados
key_hex = 'A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72'
cipher_text_b64 =
'TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4US
t3aB/i50nvvJbBiG+le1ZhpR84oI='
iv_hex = '00000000000000000000000000000000' # iv de ceros binarios

# Decodificar los datos
key = bytes.fromhex(key_hex)
iv = bytes.fromhex(iv_hex)
cipher_text = base64.b64decode(cipher_text_b64.replace(' ', ''))

# Configurar el descifrador
cipher = AES.new(key, AES.MODE_CBC, iv)

# Descifrar el texto cifrado con PKCS7
try:
    plain_text_pkcs7 = unpad(cipher.decrypt(cipher_text), AES.block_size,
style='pkcs7')
    print("Texto descifrado con PKCS7:", plain_text_pkcs7.decode('utf-8'))
except (ValueError, UnicodeDecodeError) as e:
    print(f"Error descifrando con PKCS7: {e}")

# Descifrar el texto cifrado con x923 padding
try:
    plain_text_x923 = unpad(cipher.decrypt(cipher_text), AES.block_size,
style='x923')
    try:
        print("Texto descifrado con x923:", plain_text_x923.decode('utf-8'))
    except UnicodeDecodeError as e:
        print("Texto descifrado con x923 (bytes):", plain_text_x923)
        print(f"Error decodificando con x923: {e}")
except ValueError as e:
    print(f"Error descifrando con x923: {e}")

# Verificar el padding añadido en PKCS7
padding_len_pkcs7 = AES.block_size - (len(plain_text_pkcs7) % AES.block_size)
print(f"Padding añadido (PKCS7): {padding_len_pkcs7} bytes")
```

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

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

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

Ocurre un error debemos recordar que el padding debe coincidir con el que se utilizó en el cifrado. Si el texto fue cifrado usando PKCS7, descifrarlo con x923 producirá resultados incorrectos y posiblemente errores, como ya observamos. Podemos intentar descifrar el texto cifrado utilizando x923 y observar el resultado. Sin embargo, es importante notar que este resultado no será correcto debido a la discrepancia en el padding.

¿Cuánto padding se ha añadido en el cifrado?

Para determinar cuánto padding se ha añadido en el cifrado, podemos utilizar la función `unpad` para eliminar el padding y luego calcular la diferencia entre el tamaño del bloque AES y el tamaño del texto descifrado. Dado que PKCS7 y x923 añaden bytes de padding según la cantidad de bytes necesarios para completar el tamaño del bloque (en este caso, 16 bytes), podemos verificar el último byte del texto descifrado, que indica cuántos bytes de padding se añadieron.

El último byte del texto descifrado proporciona la cantidad de bytes de padding añadidos, que en este caso es 1 byte.

```
Texto descifrado con PKCS7: Esto es un cifrado en bloque típico. Recuerda, vas
por el buen camino. Ánimo.

Traceback (most recent call last):
  File
"C:\Users\User\Documents\COD\BOOTCAMP\KEEPCODING\crypto\criptografia-main\Pract
ica\2.py", line 27, in <module>
    print("Texto descifrado con x923:", plain_text_x923.decode('utf-8'))
                                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xd8 in position 0: invalid
continuation byte
```

```
Texto descifrado con PKCS7: Esto es un cifrado en bloque típico. Recuerda, vas
por el buen camino. Ánimo.

Texto descifrado con x923 (bytes):
b'\xd8T\xca\x9d{cR\x9e\xe0\x83y\xe5\xfd\x1a\x90\xe3do en bloque t\xc3\xadco.
Recuerda, vas por el buen camino. \xc3\xadimo.'
Error decodificando con x923: 'utf-8' codec can't decode byte 0xd8 in position 0:
invalid continuation byte

Padding añadido (PKCS7): 1 bytes
```

```
(proyectoit) C:\Users\User\Documents\CODE\BOOTCAMPS\KEEPCODING\crypto\criptografia-main\Practica>python 2.py
Texto descifrado con PKCS7: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.
Traceback (most recent call last):
  File "C:\Users\User\Documents\CODE\BOOTCAMPS\KEEPCODING\crypto\criptografia-main\Practica\2.py", line 27, in <modul
e>
    print("Texto descifrado con x923:", plain_text_x923.decode('utf-8'))
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xd8 in position 0: invalid continuation byte

(proyectoit) C:\Users\User\Documents\CODE\BOOTCAMPS\KEEPCODING\crypto\criptografia-main\Practica>python 2.py
Texto descifrado con PKCS7: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.
Texto descifrado con x923 (bytes): b'\xd8T\xca\x9d\xcR\x9e\xe0\x83y\xe5\xfd\x1a\x90\xe3do en bloque t\xc3\xadadpico. Rec
uerda, vas por el buen camino. \xc3\x81nimo.'
Error decodificando con x923: 'utf-8' codec can't decode byte 0xd8 in position 0: invalid continuation byte
Padding añadido (PKCS7): 1 bytes
```

```
# Desencriptar el texto
decrypted_data = cipher.decrypt(nonce, ciphertext, None)

# Mostrar el texto descifrado
print(f"Texto descifrado: {decrypted_data.decode('utf-8')}")
```

```
4ec95921ca8b757e2336605c7dbab8f4d40b5b4d220e66aa978f740d3e59b0cd70e3217242991cc14
0bb1e1a710cd4723da6d5ef37f23bee66285e57
```

```
(proyectoit) C:\Users\User\Documents\CODE\BOOTCAMP5\KEEPCODING\crypto\criptografia-main\Pr
actica>python 3.py
Cifrado (hex): 4ec95921ca8b757e2336605c7dbab8f4d40b5b4d220e66aa978f740d3e59b0cd70e32172429
91cc140bb1e1a710cd4723da6d5ef37f23bee66285e57
Texto descifrado: KeepCoding te enseña a codificar y a cifrar
```

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

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmVlIjoiaWJlRG9uIFBlcGl0byBkZSB
sb3MgcGFsb3RlcylsInJvbCI6ImIzTm9ybWVzIiwiaWF0IjoxNjY3OTMzMzQgZGhwO
dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE
```

¿Qué algoritmo de firma hemos realizado?

El JWT especifica el algoritmo de firma en su encabezado (header). En este caso, el header codificado en base64 es:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
```

Decodificando obtenemos:

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

Por lo tanto, el algoritmo de firma es **HS256** (HMAC-SHA256).

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmIvIjoiRG9uIFB1cG10byBkZSBsb3MgcGFsb3RlcysInJvbCI6ImIzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzZQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE
```

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

¿Cuál es el body del JWT?

El body (payload) del JWT es la segunda parte del token. Codificado en base64 es:

```
eyJ1c3VhcmIvIjoiRG9uIFB1cG10byBkZSBsb3MgcGFsb3RlcysInJvbCI6ImIzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzZQ
```

Decodificando obtenemos:

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

PAYLOAD: DATA

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

VERIFY SIGNATURE

Un hacker está enviando a nuestro sistema el siguiente jwt:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmIvIjoiRG9uIFB1cG10byBkZSBsb3MgcGFsb3RlcysInJvbCI6ImIzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzZQ
```

¿Qué está intentando realizar?

El hacker ha modificado el campo rol de `isNormal` a `isAdmin`, intentando obtener permisos de administrador.

ZSBsb3MgcGFsb3RlcYIsInJvbCI6ImIzQWRt

iLCjPYYQi0jE2Njc5MzM1MzN9.krgBkzCBQ5WZ8

JnZHuvrmnAZdg4ZMeRNv2CIAOD1HRI

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

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

```
import jwt

# Clave secreta
secret_key = "Con KeepCoding aprendemos"

# JWT original
jwt_original =
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiriRG9uIFB1cGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImIzTm9ybWFsIiwiaWF0IjoxNjY3OTMzMNTMzfQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE"

# JWT modificado por el hacker
jwt_hacker =
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiriRG9uIFB1cGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImIzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHURvmnAZdg4ZMeRNv2CIAODlhRI"

try:
    # Decodificar y validar el JWT original
    payload_original = jwt.decode(jwt_original, secret_key, algorithms=["HS256"])
    print("JWT original es válido.")
    print("Payload:", payload_original)
except jwt.InvalidTokenError as e:
    print("JWT original no es válido:", str(e))

try:
    # Decodificar y validar el JWT del hacker
    payload_hacker = jwt.decode(jwt_hacker, secret_key, algorithms=["HS256"])
    print("JWT del hacker es válido.")
    print("Payload:", payload_hacker)
```



```
except jwt.InvalidTokenError as e:  
    print("JWT del hacker no es válido:", str(e))
```

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?

Dado que el hash tiene una longitud de 64 caracteres hexadecimales, se trata de un SHA3-256 (SHA3 Keccak de 256 bits).

```
main.py +
1 import hashlib
2
3 # Texto dado
4 texto = "En KeepCoding aprendemos cómo protegernos con criptografía"
5
6 # Generar SHA3-256
7 hash_sha3_256 = hashlib.sha3_256(texto.encode()).hexdigest()
8 print(hash_sha3_256)
```

Ln: 4, Col: 68

 Run  Share Command Line Arguments

 bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

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

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07
cfd69c488823b8d858283f1d05877120e8c5351c833

¿Qué hash hemos realizado?

SHA2-512

Hash Analyzer

Tool to identify hash types. Enter a hash to be identified.

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f64

Analyze

| | |
|-------------------|--|
| Hash: | 4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833 |
| Salt: | Not Found |
| Hash type: | SHA2-512 |
| Bit length: | 512 |
| Character length: | 128 |
| Character type: | hexidecimal |

main.py

+

```
1 import hashlib
2
3 # Texto dado
4 texto = "En KeepCoding aprendemos cómo protegernos con criptografía"
5
6 # Generar SHA-512
7 hash_sha512 = hashlib.sha512(texto.encode()).hexdigest()
8 print(hash_sha512)
```

Ln: 8, Col: 19

Run

Share

Command Line Arguments



4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

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?



The screenshot shows a Python IDE with a file named 'main.py'. The code is as follows:

```
1 import hashlib
2
3 # Texto dado
4 texto = "En KeepCoding aprendemos cómo protegernos con criptografía."
5
6 # Generar SHA3-256
7 hash_sha3_256 = hashlib.sha3_256(texto.encode()).hexdigest()
8 print(hash_sha3_256)
```

The status bar at the bottom indicates 'Ln: 8, Col: 21'. Below the code editor, there are buttons for 'Run' (a green play icon) and 'Share' (a share icon), followed by a text input field labeled 'Command Line Arguments'. The output of the script is displayed in a terminal window at the bottom, showing the hexadecimal hash: '302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf'.

```
import hashlib

# Texto dado
texto = "En KeepCoding aprendemos cómo protegernos con criptografía."

# Generar SHA3-256
hash_sha3_256 = hashlib.sha3_256(texto.encode()).hexdigest()
print(hash_sha3_256)

//302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf
```

La propiedad destacada del hash es su sensibilidad a pequeños cambios en el texto de entrada. A pesar de que el texto original y el texto nuevo son casi idénticos (la diferencia es solo un punto adicional), los hashes generados son completamente diferentes.

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.

```
import hmac
import hashlib

# Clave en hexadecimal
hex_key = "A212A51C997E14B4DF08D55967641B0677CA31E049E672A4B06861AA4D5826EB"
# Convertir la clave a bytes
key = bytes.fromhex(hex_key)
```

```
# Texto para el que queremos calcular el HMAC-SHA256
text = "Siempre existe más de una forma de hacerlo, y más de una solución válida."
# Convertir el texto a bytes
message = text.encode()

# Crear el objeto HMAC con la clave y el hash SHA256
hmac_obj = hmac.new(key, message, hashlib.sha256)

# Obtener el resultado del HMAC en hexadecimal
hmac_result = hmac_obj.hexdigest()

print(hmac_result)
```

857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550

```
(proyectoit) C:\Users\User\Documents\CODE\BOOTCAMP\KEEPCODING\crypto\criptografia-main\Practica>python 6.py
857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550
```

7. Trabajamos en una empresa de desarrollo que tiene una aplicación web, la cual requiere un login y trabajar con passwords. Nos preguntan qué mecanismo de almacenamiento de las mismas proponemos.

Tras realizar un análisis, el analista de seguridad propone un hash SHA-1. Su responsable, le indica que es una mala opción. ¿Por qué crees que es una mala opción?

SHA-1 es un algoritmo de hash considerado inseguro desde hace varios años. En 2005 se descubrieron varias debilidades criptográficas en SHA-1 y, en 2017, Google y CWI Amsterdam demostraron una colisión práctica para SHA-1. Esto significa que dos diferentes conjuntos de datos pueden producir el mismo hash, comprometiendo la integridad y seguridad de los datos.

Los algoritmos de hash como SHA-1 están diseñados para ser rápidos. Esta característica es deseable en muchas aplicaciones, pero no es adecuada para almacenar contraseñas, ya que permite a los atacantes realizar ataques de fuerza bruta y adivinar muchas contraseñas por segundo.

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?

Aunque SHA-256 es más seguro que SHA-1 y aún no ha sido comprometido, todavía es rápido y, por lo tanto, susceptible a ataques de fuerza bruta. Para aumentar la seguridad, necesitamos hacer que el proceso de hashing sea más lento y complicado para dificultar los ataques.

Para fortalecer SHA-256, se pueden usar técnicas como añadir **salting y pepper**.

Salting: Un salt es un valor aleatorio que se añade a la contraseña antes de hashearla. Cada contraseña tiene un salt único, lo que significa que incluso si dos usuarios tienen la misma contraseña, los hashes serán diferentes. Esto ayuda a prevenir ataques de rainbow table.

Pepper: Además del salting, se puede usar pepper para agregar una capa adicional de seguridad. Un pepper es un valor secreto (una cadena corta de caracteres) que se añade a la contraseña antes de hashearla, similar a un salt, pero a diferencia de éste, el pepper no se almacena junto al hash en la base de datos. En cambio, se mantiene en un lugar seguro, separado del almacenamiento principal (por ejemplo, en una variable de entorno). Esto significa que incluso si un atacante obtiene acceso a la base de datos de hashes, no podrá

realizar ataques de fuerza bruta sin conocer el pepper.

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

bcrypt (Ya no recomendado): Era una opción popular que incorpora salting y key stretching. Específicamente diseñado para proteger contraseñas, bcrypt adapta la velocidad del algoritmo para que sea lento, dificultando los ataques de fuerza bruta.

scrypt: Similar a bcrypt, pero también requiere una gran cantidad de memoria, lo que lo hace resistente a ataques con hardware especializado como GPUs y ASICs.

Argon2 (Recomendada) : El ganador del Password Hashing Competition (PHC). Es considerado uno de los algoritmos más seguros y modernos para hashing de contraseñas, diseñado para ser resistente a ataques de fuerza bruta y optimizado para la seguridad y eficiencia.

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?

Para asegurar la confidencialidad y la integridad de los mensajes en la API REST sin TLS, podemos usar el algoritmo ChaCha20-Poly1305. Este enfoque garantiza que los mensajes están cifrados y que cualquier alteración sea detectada, cumpliendo con los requisitos de seguridad de la API. Así también podremos mantener confidenciales ciertos parámetros como el nombre, la tarjeta, los movimientos e incluso el saldo, ya que estos temas son muy delicados.

ChaCha20-Poly1305 es un algoritmo actual y el más recomendado en la actualidad, es una opción excelente para asegurar tanto la confidencialidad (cifrado) como la integridad (autenticación) de los mensajes. Este algoritmo combina el cifrado ChaCha20 y el código de autenticación de mensajes Poly1305. ChaCha20 se encarga de cifrar el contenido del mensaje, garantizando que sólo los destinatarios autorizados puedan leer el mensaje. Poly1305 se encarga de generar un código de autenticación de mensajes (MAC), garantizando que el mensaje no haya sido modificado durante la transmisión.

Para poder aplicarlo podemos seguir los siguientes pasos:

- Generar una clave secreta: Tanto el cifrado como la autenticación requieren una clave secreta compartida entre el cliente y el servidor.
- Cifrar el mensaje: Usa ChaCha20 para cifrar el mensaje JSON. Incluye un nonce (número único de uso único) con cada mensaje para evitar ataques de repetición.
- Generar un código de autenticación: Usa Poly1305 para crear un MAC sobre el mensaje cifrado y el nonce.
- Incluir el MAC en el mensaje: Envía el mensaje cifrado junto con el MAC en la petición.
- Desencriptar y verificar: Al recibir el mensaje, el destinatario debe verificar el MAC con Poly1305 para

asegurarse de que el mensaje no ha sido alterado y luego descifrar el mensaje con ChaCha20.

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.

```
import hashlib
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad

# Clave AES proporcionada
clave_aes =
bytes.fromhex("A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72")

# Calcular SHA-256 de la clave
hash_sha256 = hashlib.sha256(clave_aes).digest()

# Obtener los 3 primeros bytes del SHA-256
kcv_sha256 = hash_sha256[:3]

text_block = bytes(16)

iv = bytes(16)

# Crear el cifrador AES en modo CBC con la clave y IV
cipher = AES.new(clave_aes, AES.MODE_CBC, iv)

# Cifrar el bloque de texto
ciphertext = cipher.encrypt(pad(text_block, AES.block_size))

# Obtener los 3 primeros bytes del ciphertext
kcv_aes = ciphertext[:3]

print(kcv_sha256.hex(), kcv_aes.hex())
```

('db7df2', '5244db')

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.

Verificar la firma

```
gpg --import Pedro-publ.txt
```

```
(kali@kali)-[~/Desktop]
$ gpg --import Pedro-publ.txt

gpg: key D730BE196E466101: "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>" not changed
gpg: Total number processed: 1
gpg:      unchanged: 1
```

```
gpg --verify MensajeRespoDeRaulARRHH.sig
```

```
(kali@kali)-[~/Desktop]
$ gpg --verify MensajeRespoDeRaulARRHH.sig

gpg: Signature made Sun 26 Jun 2022 07:47:01 AM EDT
gpg:      using EDDSA key 1BDE635E4EAE6E68DFAD2F7CD730BE196E466101
gpg:      issuer "pedro.pedrito.pedro@empresa.com"
gpg: Good signature from "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:      There is no indication that the signature belongs to the owner.
Primary key fingerprint: 1BDE 635E 4EAE 6E68 DFAD 2F7C D730 BE19 6E46 6101
```

```
gpg --import RRHH-priv.txt
```

```
(kali㉿kali)-[~/Desktop]
$ gpg --import RRHH-priv.txt

gpg: key 3869803C684D287B: "RRHH <RRHH@RRHH>" not changed
gpg: key 3869803C684D287B: secret key imported
gpg: Total number processed: 1
gpg:      unchanged: 1
gpg:      secret keys read: 1
gpg:      secret keys unchanged: 1
```

```
echo "Viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario. Saludos." > mensaje_rrhh.txt
```

```
(kali㉿kali)-[~/Desktop]
$ echo "Viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario. Saludos." > mensaje_rrhh.txt

(kali㉿kali)-[~/Desktop]
$ cat mensaje_rrhh.txt
Viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario. Saludos.
```

Firmar el mensaje

```
gpg --clearsign --default-key 3869803C684D287B mensaje_rrhh.txt
```

```
(kali㉿kali)-[~/Desktop]
$ gpg --clearsign --default-key 3869803C684D287B mensaje_rrhh.txt
gpg: using "3869803C684D287B" as default secret key for signing
```

Cifrar el mensaje para RRHH y Pedro

Importar la clave pública de RRHH

```
gpg --import RRHH-publ.txt
```

```
(kali㉿kali)-[~/Desktop]
$ gpg --import RRHH-publ.txt

gpg: key 3869803C684D287B: "RRHH <RRHH@RRHH>" not changed
gpg: Total number processed: 1
gpg:      unchanged: 1
```

```
echo "Estamos todos de acuerdo, el ascenso será el mes que viene, agosto, si no hay sorpresas." > mensaje_a_cifrar.txt
```

```
(kali@kali)-[~/Desktop]
$ echo "Estamos todos de acuerdo, el ascenso será el mes que viene, agosto, si no hay sorpresas." > mensaje_a_cifrar.txt

(kali@kali)-[~/Desktop]
$ cat mensaje_a_cifrar.txt
Estamos todos de acuerdo, el ascenso será el mes que viene, agosto, si no hay sorpresas.
```

Cifrar el mensaje con las claves públicas de RRHH y Pedro

```
gpg --encrypt --recipient 3869803C684D287B --recipient D730BE196E466101 mensaje_a_cifrar.txt
```

```
(kali@kali)-[~/Desktop]
$ gpg --encrypt --recipient 3869803C684D287B --recipient D730BE196E466101 mensaje_a_cifrar.txt
gpg: 25D6D0294035B650: There is no assurance this key belongs to the named user

sub cv25519/25D6D0294035B650 2022-06-26 Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>
Primary key fingerprint: 1BDE 635E 4EAE 6E68 DFAD 2F7C D730 BE19 6E46 6101
Subkey fingerprint: 8E8C 6669 AC44 3271 42BC C244 25D6 D029 4035 B650

It is NOT certain that the key belongs to the person named
in the user ID. If you *really* know what you are doing,
you may answer the next question with yes.

Use this key anyway? (y/N) y
gpg: 7C1A46EA20B0546F: There is no assurance this key belongs to the named user

sub cv25519/7C1A46EA20B0546F 2022-06-26 RRHH <RRHH@RRHH>
Primary key fingerprint: F2B1 D0E8 958D F2D3 BDB6 A105 3869 803C 684D 287B
Subkey fingerprint: 811D 89A3 6199 A7C9 0BFE 69D6 7C1A 46EA 20B0 546F

It is NOT certain that the key belongs to the person named
in the user ID. If you *really* know what you are doing,
you may answer the next question with yes.
```

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.

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

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.backends import default_backend
import binascii

# Leer la clave privada desde el archivo
with open('clave-rsa-oaep-priv.pem', 'rb') as key_file:
    private_key = serialization.load_pem_private_key(
        key_file.read(),
        password=None,
        backend=default_backend()
    )

# Leer la clave pública desde el archivo
with open('clave-rsa-oaep-publ.pem', 'rb') as key_file:
    public_key = serialization.load_pem_public_key(
        key_file.read(),
        backend=default_backend()
    )

# Texto cifrado proporcionado
ciphertext_hex = (
    "b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1"
    "709b30c96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d"
    "04e3d3d4ad629793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae"
    "2c4f5ba670a4126f2f14a9f4b6f41aa2edba01b4bd586624659fca82f5b49701"
    "86502de8624071be78cce5f573d896b8eac86f5d43ca7b10b59be4acf8f8e0498"
    "a455da04f67d3f98b4cd907f27639f4b1df3c50e05d5bf63768088226e2a9177"
    "485c54f72407fdf358fe64479677d8296ad38c6f177ea7cb74927651cf24b01d"
    "ee27895d4f05fb5c161957845cd1b5848ed64ed3b03722b21a526a6e447cb8ee"
)

# Convertir el texto cifrado de hexadecimal a bytes
ciphertext = binascii.unhexlify(ciphertext_hex)

# Descifrar el texto cifrado con la clave privada
plaintext = private_key.decrypt(
    ciphertext,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

try:
    print("Texto descifrado:", plaintext.decode('utf-8'))
except UnicodeDecodeError:
```

```

    print("El contenido descifrado no es texto UTF-8. Imprimiendo en formato hexadecimal.")
    print(binascii.hexlify(plaintext).decode('utf-8'))

# Cifrar la clave simétrica nuevamente con la clave pública
new_ciphertext = public_key.encrypt(
    plaintext,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

# Convertir el nuevo texto cifrado a hexadecimal para mostrarlo
new_ciphertext_hex = binascii.hexlify(new_ciphertext).decode('utf-8')
print("Nuevo cifrado:", new_ciphertext_hex)

```

```
e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72
```

Paste hex numbers or drop file

```
e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72
```

Character encoding

ASCII

Convert

Reset

Swap

```
âïø??TIéÄH°[??Nãw?+?ï ??³x&Ûr
```

Si has recuperado la clave, vuelvela a cifrar con el mismo algoritmo

Resultado:

```
8e978385fc43536bcd3b7c071f3c0829d08738a5a853a0eefe0237d4f1441fa0d3596c52541c7ea42
98bde5ab62fd81dbdb14f3546736d73dcb08d76efa4747d180e326c002de70c18c407657f4c8f3f58
0bdd5176d1a1171a2272481c57ea1e39c2777f002dd32e2a745724344a7b9540c1204c81f5b01d33f
9f201c800c786558f419c32c35a85ac4cfe4b3153075ae6eee1fb94532d80b966761de583df11f
c2a52b49ea3eb8064d50031bb2df91947ee84be403382cd74c87bc1b2080f0ca3764a53fffa76c299
c247178d55b7b1306bffe84b18772a1606b82ea72d390ba41c3d72d4050c309056c815b351e4ee53a
2ae32ad96659c9cad8d4864988
```

```
(proyectoit) C:\Users\User\Documents\CODE\BOOTCAMP\KEEPCODING\crypto\criptografia-main\Pr
ctica>python 11.py
8e978385fc43536bcd3b7c071f3c0829d08738a5a853a0eefe0237d4f1441fa0d3596c52541c7ea4298bde5ab6
2fd81dbdb14f3546736d73dcb08d76efa4747d180e326c002de70c18c407657f4c8f3f580bdd5176d1a1171a22
72481c57ea1e39c2777f002dd32e2a745724344a7b9540c1204c81f5b01d33f9f201c800c786558f419c32c35a
85ac4cfe4b3153075ae6eee1fb94532d80b966761de583df11fc2a52b49ea3eb8064d50031bb2df91947ee8
4be403382cd74c87bc1b2080f0ca3764a53fffa76c299c247178d55b7b1306bffe84b18772a1606b82ea72d390
ba41c3d72d4050c309056c815b351e4ee53a2ae32ad96659c9cad8d4864988
```

¿Por qué son diferentes los textos cifrados?

El algoritmo RSA-OAEP (Optimal Asymmetric Encryption Padding) incluye una parte aleatoria en su esquema de padding. Esto significa que cada vez que se cifra el mismo mensaje (la misma clave simétrica en este caso) con la misma clave pública, el texto cifrado resultante será diferente. La aleatoriedad en el padding asegura que el texto cifrado sea único incluso para el mismo mensaje, proporcionando una mayor seguridad y dificultando los ataques de criptoanálisis.

Este comportamiento se debe a que OAEP utiliza un padding basado en un generador de máscara que incorpora valores aleatorios, garantizando que cada cifrado produce un resultado distinto.

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?

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.

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
import base64
import binascii

key_hex = 'E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74'
nonce_base64 = '9Yccn/f5nJJhAt2S'
```

```

plaintext = 'He descubierto el error y no volveré a hacerlo mal'

# Convertir clave de hexadecimal a bytes
key_bytes = binascii.unhexlify(key_hex)

# Convertir nonce de Base64 a bytes
nonce_bytes = base64.b64decode(nonce_base64)

# Convertir texto a bytes
plaintext_bytes = plaintext.encode()

# Crear el cifrador AES/GCM
cipher = Cipher(algorithms.AES(key_bytes), modes.GCM(nonce_bytes),
backend=default_backend())
encryptor = cipher.encryptor()

# Cifrar el texto
ciphertext_bytes = encryptor.update(plaintext_bytes) + encryptor.finalize()

# Obtener el tag de autenticación
tag = encryptor.tag

# Convertir a formato hexadecimal y Base64
ciphertext_hex = binascii.hexlify(ciphertext_bytes).decode('utf-8')
ciphertext_base64 = base64.b64encode(ciphertext_bytes).decode('utf-8')
tag_hex = binascii.hexlify(tag).decode('utf-8')
tag_base64 = base64.b64encode(tag).decode('utf-8')

print("Texto en hexadecimal:", ciphertext_hex)
print("Texto en base64:", ciphertext_base64)
print("Tag hex:", tag_hex)
print("Tag base64:", tag_base64)

# Descifrar para verificar
cipher = Cipher(algorithms.AES(key_bytes), modes.GCM(nonce_bytes, tag),
backend=default_backend())
decryptor = cipher.decryptor()
decrypted_bytes = decryptor.update(ciphertext_bytes) + decryptor.finalize()

# Convertir bytes a texto
decrypted_text = decrypted_bytes.decode()
print("-----Verificacion-----")
print("Texto descifrado:", decrypted_text)

# Verificar longitud del nonce
print("Longitud del nonce en bytes:", len(nonce_bytes))

```

Texto **en** hexadecimal:

5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d

Texto **en** base64:


```
(proyectoit) C:\Users\User\Documents\CODE\B00TCAMPS\KEEPCODING\crypto\criptografia-main\Practica>python 12.py
Texto en hexadecimal: 5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f
5078fbfc16256d
Texto en base64: Xcu2Jh0Puin00UMemgE7NMvKKk4Euy2QFJ1h9K/QTWxiq92dhLum64MHCv9QePv8FiVt
Tag hex: 6120e37aa4c3ecfd9261640dcc46410d
Tag base64: YSDjeqTD7P2SYWQNzEZBDQ==
-----Verificacion-----
Texto descifrado: He descubierto el error y no volveré a hacerlo mal
Longitud del nonce en bytes: 12
```

13. Se desea calcular una firma con el algoritmo PKCS#1 v1.5 usando las claves contenidas en los ficheros clave-rsa-oaep-priv y clave-rsa-oaep-publ.pem del mensaje siguiente:

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

¿Cuál es el valor de la firma en hexadecimal?

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

```
from Crypto.Signature import pkcs1_15
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA

# Leer la clave privada RSA desde el archivo
with open('clave-rsa-oaep-priv.pem', 'r') as priv_file:
    private_key_rsa = RSA.import_key(priv_file.read())

# Crear el hash del mensaje
message = "El equipo está preparado para seguir con el proceso, necesitaremos más recursos."
hash = SHA256.new(message.encode('utf-8'))

# Calcular la firma
signature_rsa = pkcs1_15.new(private_key_rsa).sign(hash)
signature_rsa_hex = signature_rsa.hex()

print(f"Firma RSA (hexadecimal): {signature_rsa_hex}")
```

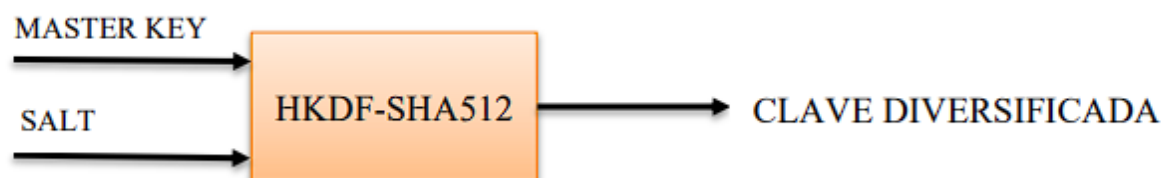
```
Firma RSA (hexadecimal):
a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92aa3
d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959bcaa
e2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a9ec798fe6630
8e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf
05d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d
79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0d
```



```
(proyectoit) C:\Users\User\Documents\CODE\BOOTCAMP\KEEP CODING\crypto\criptografia-main\Practica>python 13_2.py
Firma RSA (hexadecimal): a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21f0aaaa6f9b9d59f41928d
```

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



¿Qué clave se ha obtenido?

```
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend

# Datos iniciales
clave_maestra_hex = "E377152B3F1ACFA0148FB3A426DB72"
device_id_hex = "e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3"

# Convertir datos hexadecimales a bytes
clave_maestra = bytes.fromhex(clave_maestra_hex)
device_id = bytes.fromhex(device_id_hex)

# Usar HKDF para derivar una nueva clave
hkdf = HKDF(
    algorithm=hashes.SHA512(),
    length=32, # 256 bits
    salt=None,
    info=device_id,
    backend=default_backend()
)
```

```
clave_aes = hkdf.derive(clave_maestra)

# Imprimir la clave AES en formato hexadecimal
print(f"Clave AES derivada: {clave_aes.hex()}")
```

Clave AES derivada:
5494f4c2e50994df80f3470a30b922d544652a8bdb707262597387f725f313b8

```
(proyectoit) C:\Users\User\Documents\CODE\BOOTCAMP\KEEPCODING\crypto\criptografia-main\Practica>python 14.py
Clave AES derivada: 5494f4c2e50994df80f3470a30b922d544652a8bdb707262597387f725f313b8
```

15. Nos envían un bloque TR31:

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDB
E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495EO
3CD857FD37018E111B

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

A1A10101010101010101010101010102

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

El bloque de clave ha sido protegido usando el algoritmo 3DES (Triple DES). La clave de transporte proporcionada es utilizada con 3DES en modo ECB para descryptar el bloque TR31.

¿Para qué algoritmo se ha definido la clave?

Para determinar para qué algoritmo se ha definido la clave, debemos inspeccionar el contenido del bloque TR31 una vez descryptado. En este caso, dado que el bloque TR31 se ha descryptado con 3DES, la clave extraída probablemente se utiliza para un algoritmo de cifrado de datos o para otros propósitos criptográficos definidos en el bloque TR31.

¿Para qué modo de uso se ha generado?

El modo de uso de la clave se especifica en el encabezado del bloque TR31. Dado que el bloque descryptado proporciona una clave de 48 bytes, es probable que la clave sea utilizada para cifrado de datos o MAC. La etiqueta en el encabezado del bloque TR31 define el propósito exacto de la clave. Sin el encabezado exacto, no podemos confirmar el modo de uso específico, pero generalmente se utilizará para cifrado o autenticación.

¿Es exportable?

La exportabilidad de la clave está especificada en el encabezado del bloque TR31. La información en el encabezado indica si la clave es exportable. Como la clave descryptada es significativa en longitud y formato, la exportabilidad es una característica que se define en el bloque y no puede ser confirmada sin el encabezado completo.

¿Para qué se puede usar la clave?

La clave descriptada probablemente se utiliza para cifrado de datos o generación de códigos de autenticación de mensajes (MAC). La longitud de la clave sugiere que puede estar destinada a un uso específico dentro del contexto del bloque TR31.

¿Qué valor tiene la clave?

Para obtener el valor de la clave, debemos desempaquetar (unwrap) la clave protegida usando la clave de transporte proporcionada.

```
from Crypto.Cipher import DES3
from binascii import unhexlify

# Clave de transporte
key_transport = unhexlify("A1A101010101010101010101010102")

# Clave protegida (en formato hexadecimal)
wrapped_key =
unhexlify("42766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C
6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37018E111B")

# Crear el objeto de descriptación (unwrap) de la clave usando 3DES
cipher = DES3.new(key_transport, DES3.MODE_ECB)

# Descriptar la clave (unwrap)
unwrapped_key = cipher.decrypt(wrapped_key)

# Mostrar la clave descriptada
print("Clave descriptada:", unwrapped_key.hex())
```

Clave descriptada:

6e86ab7c01580385afd82500a41d996e616589b80e2f71b7e00ea01193f339f49078171042ea466a986c052dfbb3
d5e0567be3a2d6301ed6c23488a645ba932d