Práctica de Criptografía

David Gómez Cañego

Ejercicio I

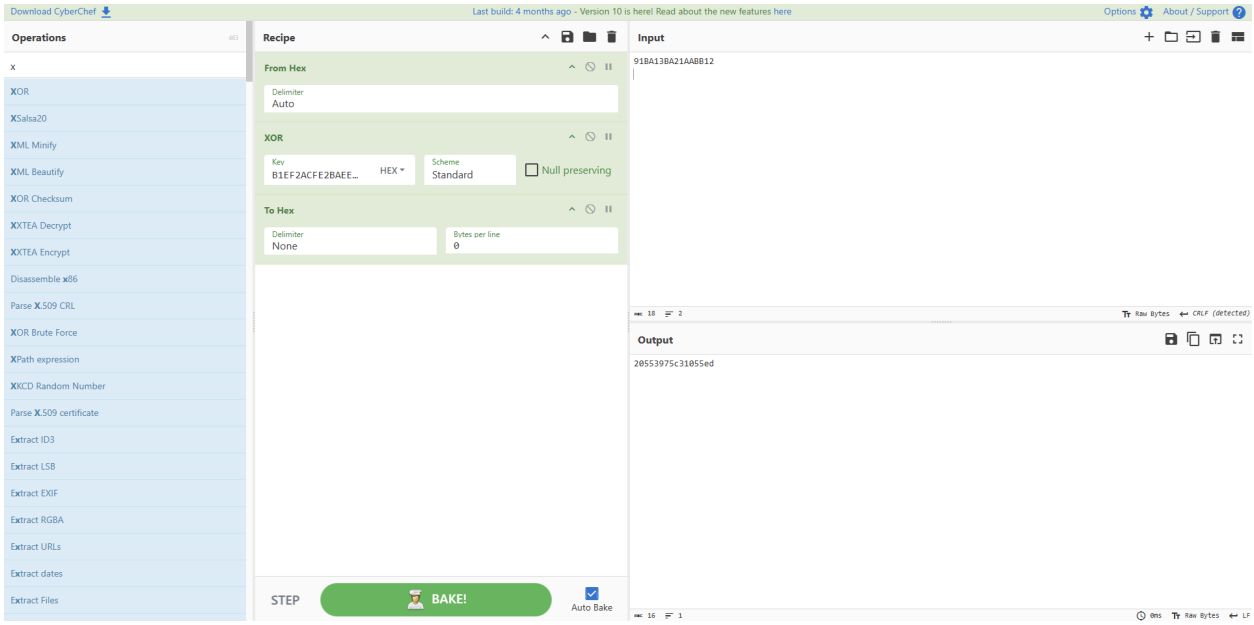
Tenemos un sistema que usa claves de 16 bytes. Por razones de seguridad vamos a proteger la clave de tal forma que ninguna persona tenga acceso directamente a la clave. Por ello, vamos a realizar un proceso de disociación de la misma, en el cuál tendremos, una clave fija en código, la cual, sólo el desarrollador tendrá acceso, y otra parte en un fichero de propiedades que rellenará el Key Manager. La clave final se generará por código, realizando un XOR entre la que se encuentra en el properties y en el código.

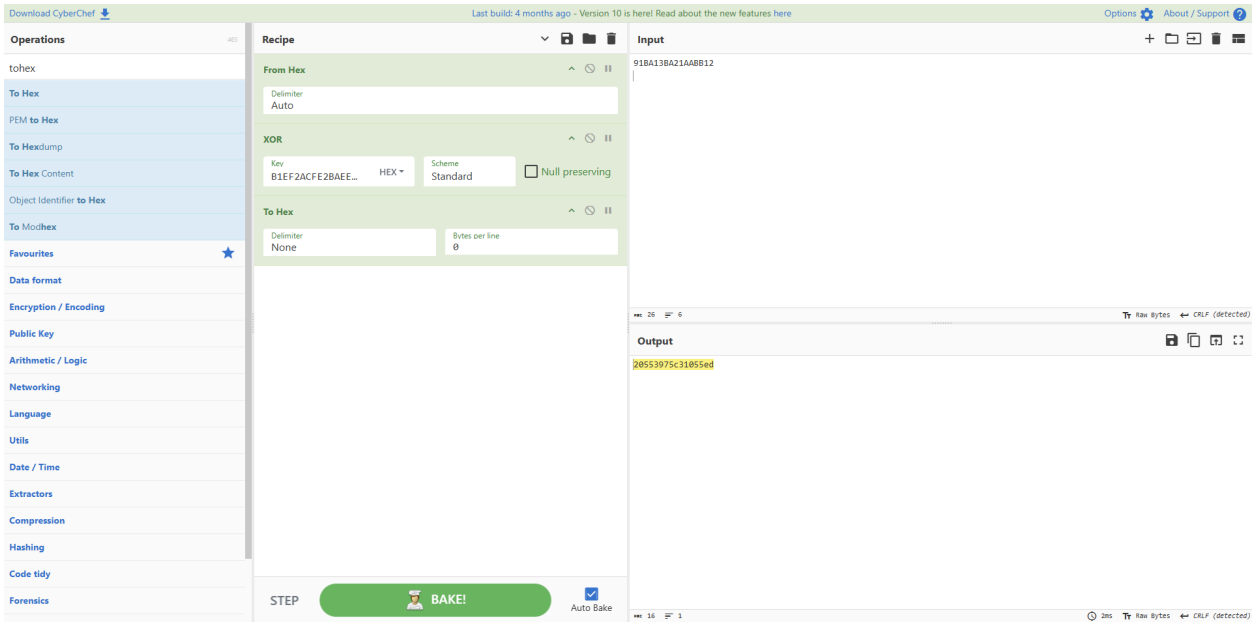
La clave fija en código es B1EF2ACFE2BAEEFF, mientras que en desarrollo sabemos que la clave final (en memoria) es 91BA13BA21AABB12. ¿Qué valor ha puesto el Key Manager en properties para forzar dicha clave final?

La clave fija, recordemos es B1EF2ACFE2BAEEFF, mientras que en producción sabemos que la parte dinámica que se modifica en los ficheros de propiedades es B98A15BA31AEBB3F. ¿Qué clave será con la que se trabaje en memoria?

Para obtener la parte dinámica introducida por el Key Manager, he utilizado CyberChef aplicando un XOR entre la clave final y la clave fija del código.

Esto permite recuperar exactamente el valor que debe introducir el Key Manager en el fichero properties. Así se garantiza que ninguna de las dos partes por sí sola conoce la clave completa, ya que solo al combinar la parte fija y la parte dinámica (Key Manager) se obtiene la clave real que usa el sistema.





Ejercicio II

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/o0QKlWXg3nu1RRz4QWElezdrLAD5LO4US 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?

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

El sistema se divide la clave en dos partes:

Una fija en el código y otra que pone el Key Manager. Para obtener la clave final hacemos un XOR entre las dos. El Key Manager puso 20553975c31055ed, que al combinarse con la clave del código da la clave final. La dinámica cambia y al hacer XOR con la parte fija sale la clave final.

Así nunca hay una sola persona con la clave completa y la sgeuridad mejora bastante.

El objetivo es descifrar un dato cifrado con AES/CBC/PKCS7 usando la clave etiquetada como “cifrado-sim-aes-256” almacenada en el keystore y un IV compuesto por ceros (00).

Herramientas utilizadas

- Python (pycryptodome) para extraer la clave del keystore y probar el descifrado.
- Funciones para manejar AES/CBC y distintos esquemas de padding (PKCS7 y X9.23).

He obtenido la clave desde el keystore usando Python y una decodificación adecuada. Luego se ha definido el IV como una cadena de ceros hexadecimales (00). Para finalmente combinar la parte fija de la clave en el código con la parte proporcionada por el Key Manager mediante XOR para generar la clave final.

Ejercicio III

Se requiere cifrar el texto "KeepCoding te enseña a codificar y a cifrar". La clave para ello, tiene la etiqueta en el Keystore "cifrado-sim-chacha-256". El nonce "9Yccn/f5nJJhAt2S". El algoritmo que se debe usar es un Chacha20.

¿Cómo podríamos mejorar de forma sencilla el sistema, de tal forma, que no sólo garanticemos la confidencialidad sino, además, la integridad del mismo? Se requiere obtener el dato cifrado, demuestra, tu propuesta por código, así como añadir los datos necesarios para evaluar tu propuesta de mejora.

Primero he instalado la librería "cryptography" en Python - Google Colab (sobre todo por comodidad personal en un cuaderno propio), luego he definido la clave y el nonce, he convertido el texto a bytes usando UTF-8 para que Python pudiera usar caracteres especiales como la ñ.

Después el mensaje se ha cifrado con ChaCha20-Poly1305 generando el texto y su sello, para finalmente porbar a descifrarlo para verificar que la integridad se mantiene y que el mensaje original se recupera bien.

¿Por qué es mejor? - Si un atacante modifica 1 bit, la verificación de Poly1305 fallará, el mensaje no se descifrará y el sistema automáticamente detecta manipulaciones.

Pasar de ChaCha20 a ChaCha20-Poly1305 - Incorpora confidencialidad con ChaCha20, integridad y autenticidad con Poly1305 y un tag de 128 bits que detecta si hay una modificación en los datos.

EJERCICIO 3

Con caracteres no ASCII

```
[39] ✓ 0s
from Crypto.Cipher import ChaCha20_Poly1305
from base64 import b64decode, b64encode

# Texto a cifrar (con caracteres no ASCII)
textoPlano = 'KeepCoding te enseña a codificar y a cifrar'.encode('utf-8')

clave = bytes.fromhex('AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120')

nonce_mensaje = b64decode('9Yccn/f5nJJhAt2S')
datos_asociados = b''

cipher = ChaCha20_Poly1305.new(key=clave, nonce=nonce_mensaje)
cipher.update(datos_asociados)

texto_cifrado, tag = cipher.encrypt_and_digest(textoPlano)

print("nonce:", b64encode(nonce_mensaje).decode())
print("Datos asociados:", b64encode(datos_asociados).decode())
print("Texto cifrado:", b64encode(texto_cifrado).decode())
print("MAC/Tag:", b64encode(tag).decode())
```

```
nonce: 9Yccn/f5nJJhAt2S
Datos asociados:
Texto cifrado: TslZIcqLdX4jNmBcfbq49NQLW00iDmaq1490DT5ZsM1w4yFyQpkcwUC7Hho=
MAC/Tag: cQzUcj2m1e838jvuZiheVw==
```

Con caracteres ASCII

```
[38] ✓ 0s
from Crypto.Cipher import ChaCha20_Poly1305
from Crypto.Random import get_random_bytes
from base64 import b64encode

# Texto plano con acentos y ñ
textoPlano = 'KeepCoding te enseña a codificar y a cifrar'.encode('utf-8')
# Clave de 32 bytes (256 bits)
clave = bytes.fromhex('AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120')
# Generar un nonce único de 12 bytes
nonce_mensaje = get_random_bytes(12)
# Datos asociados
datos_asociados = b''
# Cifrador
cipher = ChaCha20_Poly1305.new(key=clave, nonce=nonce_mensaje)
cipher.update(datos_asociados)
# Cifrar y generar tag
texto_cifrado, tag = cipher.encrypt_and_digest(textoPlano)

# Mostrar resultados
print("nonce:", b64encode(nonce_mensaje).decode())
print("Datos asociados:", b64encode(datos_asociados).decode())
print("Texto cifrado:", b64encode(texto_cifrado).decode())
print("MAC/Tag:", b64encode(tag).decode())
```

```
nonce: arM5D6EjwGODGsZ2
Datos asociados:
Texto cifrado: sN9X11MIki2Jbwu5j09vCPCgsDTMnvOq77I9CMtg69rFjxEC+2vcyNbPrio=
MAC/Tag: ylso4ocfmrd/rrTHuzirAg==
```

Ejercicio IV

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

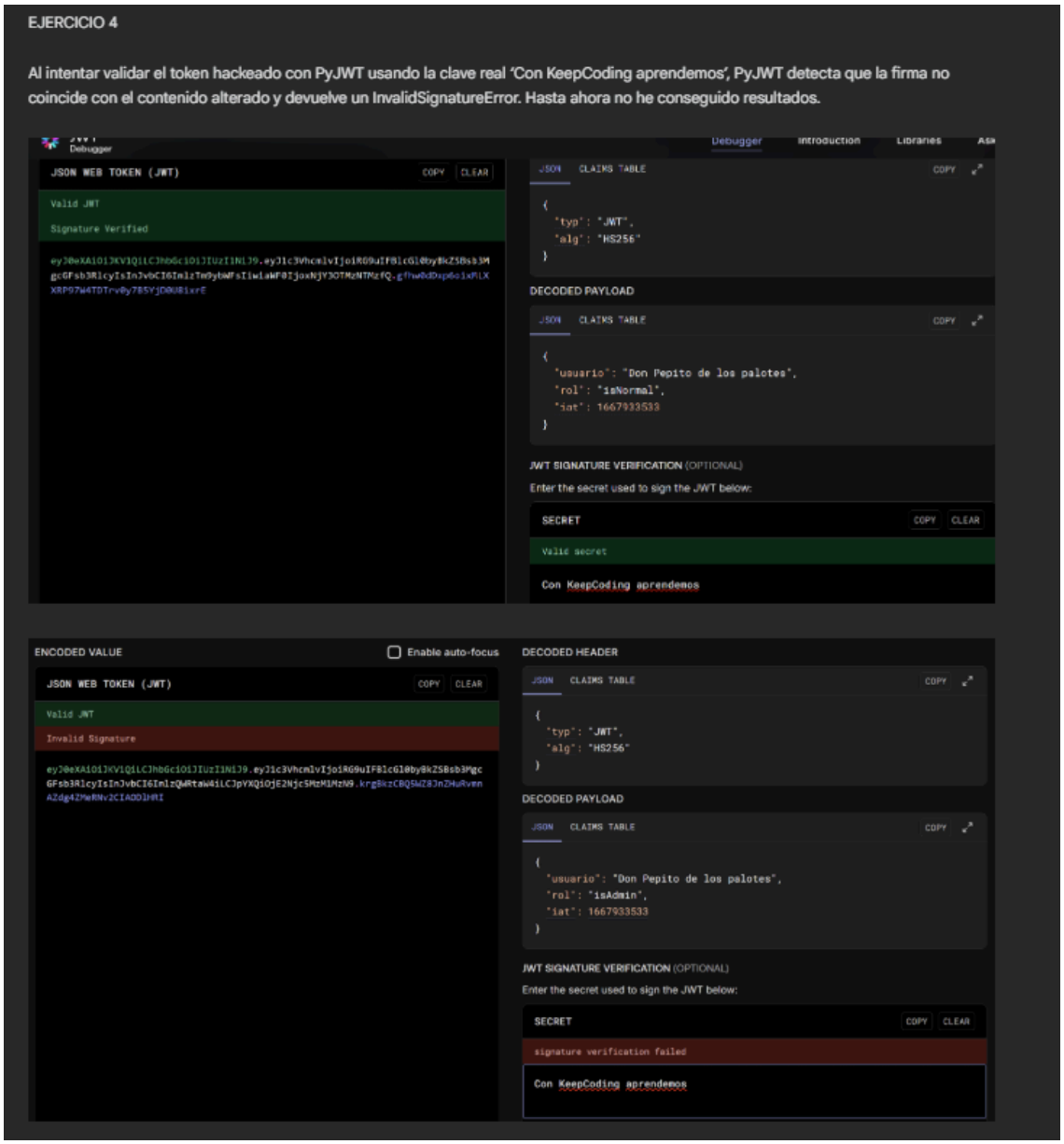
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0by BkZSB sb3MgcGFsb3RlcyIsInJvbCI6ImIzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ .gfhw0 dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

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

Un hacker está enviando a nuestro sistema el siguiente jwt:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODlHRI

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



¿Qué algoritmo de firma hemos realizado? - El algoritmo de firma es HS256 - HMAC-SHA256

¿Cuál es el body del jwt? - Se obtiene un usuario: “Don Pepito de los palotes”. Rol: “isNormal”. “Iat”: 1667933533

¿Qué está intentando realizar? - Está intentando elevar/escalar privilegios modificando el campo “rol” para cambiarlo de “isNormal” a “isAdmin”. Se suele dar en muchos casos de escalación de privilegios a través del payload JWT.

¿Qué ocurre si intentamos validarlo con pyjwt? - Calcula de nuevo la firma con la clave real “Con Keep Coding aprendemos” y detecta que la firma del hacker no coincide con el contenido que el hacker ha alterado, devuelve un error. Seguramente sea por que PyJWT lo está rechazando por una firma inválida.

Al intentar validar el token hackeado con PyJWT usando la clave real ‘Con KeepCoding aprendemos’, PyJWT detecta que la firma no coincide con el contenido alterado y devuelve un InvalidSignatureError.

Hasta ahora no he conseguido resultados y solo me salta el error ese.

Ejercicio V

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:

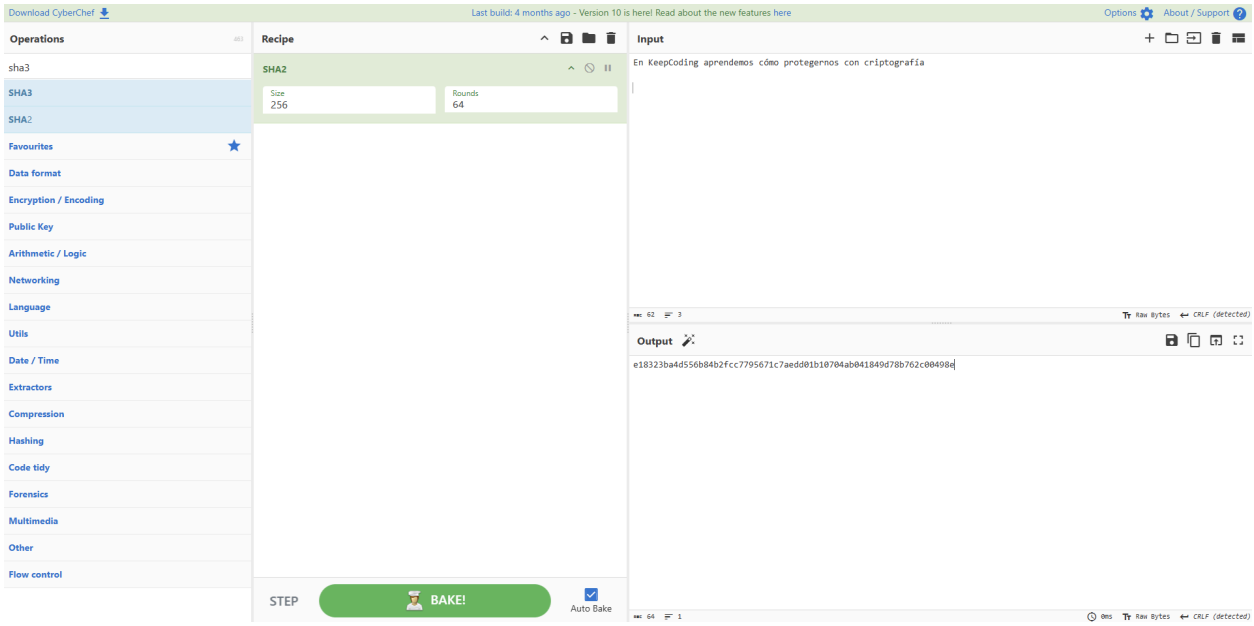
4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

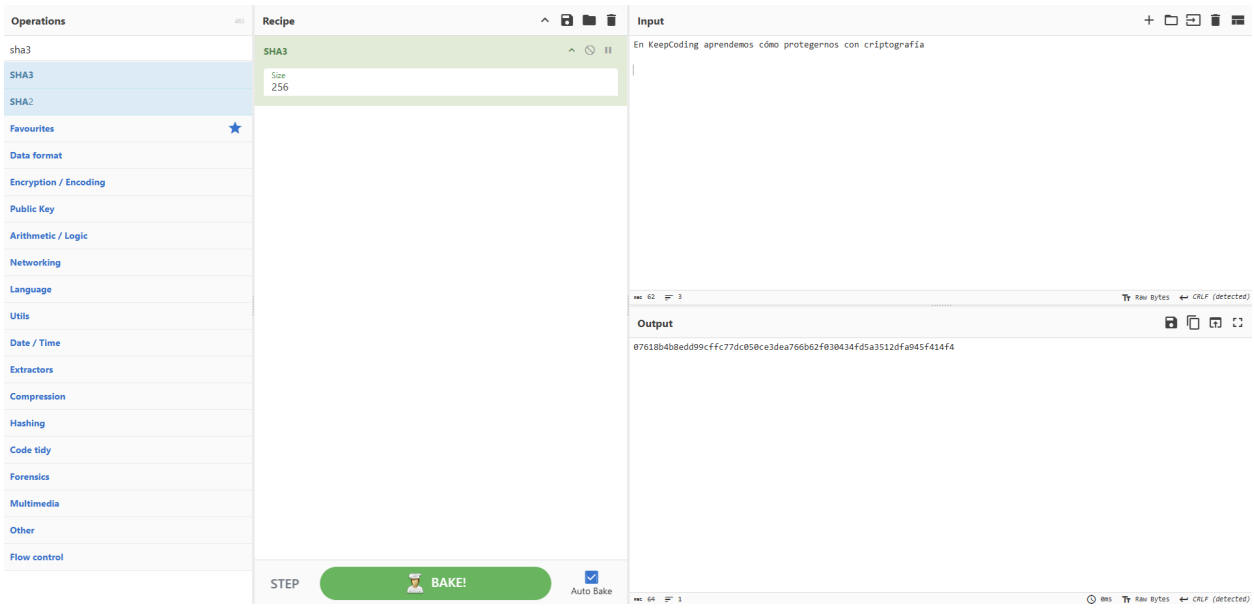
¿Qué hash hemos realizado?

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?

El primer hash lo e generado en CyberChef con SHA3-256 y el segundo con SHA2-256. Los dos crean un resumen del texto para verificar su integridad aunque los dos usan algoritmos distintos. Los dos son únicos.

He generado un hash SHA3-256 del texto que cada pequeño cambio en el texto genera un hash completamente distinto. Ese hash permite detectar cualquier modificación del contenido si es que lo llega a haber..





Ejercicio VI

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.

He calculado el HMAC-SHA256 del texto usando la clave del Keystore en Python (Colab), asegurando la codificación UTF-8 para los caracteres especiales.

El resultado se muestra en hex dejando ver el mensaje. Esta operación garantiza que cualquier modificación del texto se detectaría al validar el HMAC.

Cualquier modificación que se haga al mensaje se va a producir un valor totalmente distinto, permitiendo poder verificar la integridad como autenticidad, siempre que la clave secreta sola la conozca el sistema.

```
[13]
✓ 0s
    import hmac
    import hashlib

    # Texto a firmar
    mensaje = "Siempre existe más de una forma de hacerlo, y más de una solución válida."

    # Clave (texto normal)
    clave = b"123456"  # siempre en bytes

    # Calculamos HMAC-SHA256
    h = hmac.new(clave, mensaje.encode('utf-8'), hashlib.sha256)

    # Mostramos en hexadecimal
    print(h.hexdigest())

    a668cb6927edc31a518da2fd2d6dc34fa6ea7726f904d4a274f0d9733e130b20
```

El código para calcular el HMAC-SHA256 lo he escrito en Python en Google Colab basándome en ejemplos de documentación y tutoriales de criptografía adaptándolo a la clave y texto del ejercicio después.

El objetivo era evidenciar el cálculo y mostrar el resultado en hexadecimal para así asegurar la integridad del mensaje.

Ejercicio VII

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?

¿Por qué es una mala opción? - Porque el “protocolo” SHA-1 no es adecuado para almacenar passwords/contraseñas porque es un algoritmo antiguo y rápido, lo que facilita ataques de fuerza bruta y colisiones y no es muy seguro.

Mi propuesta para mejorar la seguridad es usar SHA-256 combinado con una “salt” (valor aleatorio) única para cada usuario asi de manera que incluso si dos usuarios tienen la misma contraseña sus hashes van a ser distintos.

Y como mejora se podría aplicar un algoritmo de derivación de claves como PBKDF2, bcrypt o scrypt para fortalecer el hash aumentando el coste computacional de ataques masivos.

- PBKDF2 - Aplica muchos hashes sobre la contraseña y el valor aleatorio
- bcrypt - Es similar a PBKDF2 pero se centra más en contraseñas
- scrypt - Va más lejos y a parte de hacer muchas iteracciones usa mucha mas memoria para calcular los hashes. Así lo hace más difícil

EJERCICIO VIII

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 proteger los datos sensibles del API para no depender de TLS, cifraría los campos confidenciales (como tarjeta e idUsuario) usando un algoritmo autenticado, como AES-GCM o ChaCha20-Poly1305.

Ambos proporcionan confidencialidad, integridad y autenticidad del mensaje en una sola operación, por lo que no sería necesario añadir un HMAC adicional.

De esta forma aunque el mensaje viaje sin TLS el atacante no podría ni leer ni modificar los datos. El cifrado evita la lectura y el tag de autenticación impide cualquier alteración. Si los datos se modifican, la verificación del tag fallará y el mensaje será rechazado.

EJERCICIO IX

Se requiere calcular el KCV de las siguiente clave AES:

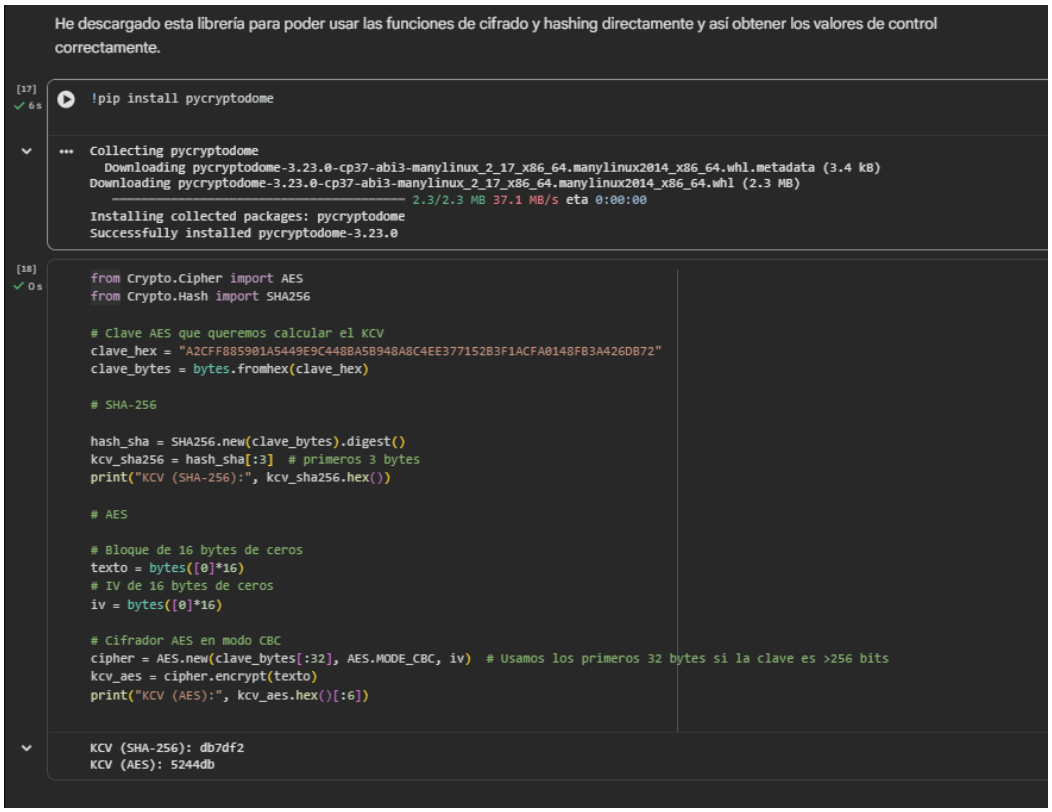
A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB 72

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.

Para calcular los KCV con AES y SHA-256 necesitamos primero la librería pycryptodome en Colab.

He aplicado un SHA-256 a la clave AES para calcular el KCV(SHA-256). El valor KCV se obtiene tomando únicamente los 3 primeros bytes (6 caracteres hex) del hash.

Para el KCV(AES) he cifrado un bloque de 16 bytes a 00…00 usando AES con la clave. Tras obtener el ciphertext en hex, el KCV corresponde también a los primeros 3 bytes del resultado. Así se permite verificar la clave sin revelar su valor completo.



EJERCICIO X

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.

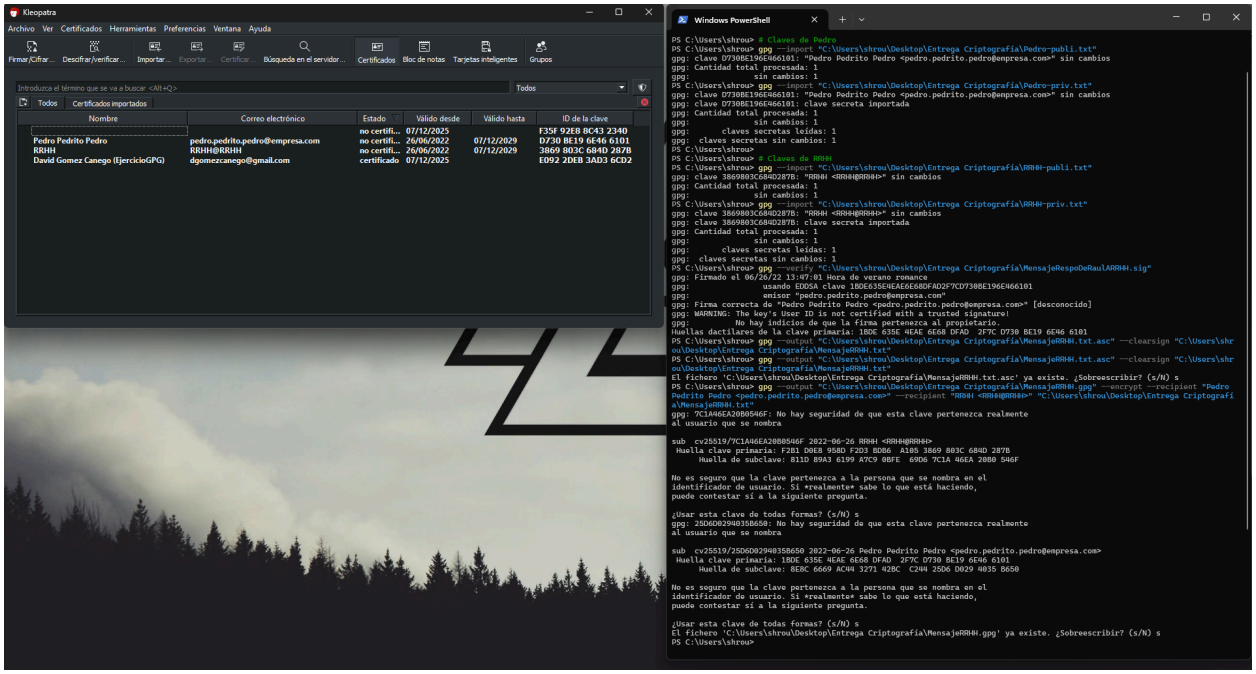
Para verificar la firma PGP de Pedro he utilizado “Pedro-publ.txt” con la herramienta GnuPG, luego en Colab he ejecutado la verificación del fichero .sig asociado al mensaje. La verificación ha sido buena, lo que confirma que el mensaje proviene de Pedro y no ha sido modificado.

Para simular la actuación del departamento de RRHH he firmado el mensaje utilizando “RRHH-priv.txt” La firma generada permite a cualquier receptor comprobar que el mensaje procede de RRHH usando la clave pública.

Por último he cifrado el mensaje final utilizando la clave pública de RRHH y la de Pedro a la vez, así cualquiera de los dos puede descifrarlo. Usando PGP estándar (GnuPG), generando un mensaje cifrado en formato ASCII Armor listo para que se comparta.

Una de las contraseñas es 11111234 (No pude poner otra debido a un error que tuve con otra clave)

https://drive.google.com/file/d/1BXXRMzdo4bmGYXrS8Tyk5W-42GMkUFCH/view?usp=drive_link



EJERCICIO XI

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?

EJERCICIO 11

```
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.asymmetric import padding
```

```
from google.colab import files

uploaded = files.upload()
```

```
Elegir archivos  Ningún archivo seleccionado Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving clave-rsa-oaep-priv.pem to clave-rsa-oaep-priv.pem
Saving clave-rsa-oaep-publ.pem to clave-rsa-oaep-publ.pem
```

```
# Clave Privada
with open("/content/clave-rsa-oaep-priv.pem", "rb") as key_file:
    private_key = serialization.load_pem_private_key(
        key_file.read(),
        password=None
    )

# Clave Pública
with open("/content/clave-rsa-oaep-publ.pem", "rb") as key_file:
    public_key = serialization.load_pem_public_key(
        key_file.read()
    )
```

He usado clave privada RSA-OAEP para descifrar la clave simétrica enviada por la empresa de almacenamiento de videollamadas. Luego he vuelto a cifrar la misma clave utilizando la clave pública RSA-OAEP con SHA-25.

El resultado vuelve a generar un texto diferente al original debido al relleno incorpora el esquema OAEP, aunque sea el mismo mensaje y clave, es único. Hace que haya una recuperación de la clave como la aplicación segura del cifrado según el algoritmo que se solicite.

EJERCICIO XII

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.

EJERCICIO 12

He usado AES en modo GCM para cifrar de forma simétrica. Se convierte la clave de hex a bytes y se adapta el nonce a los 12 bytes de AES-GCM. El mensaje se cifra generando un tag de autenticación para integridad y autenticidad. Salen los resultados en hex y base64 para la verificación y compatibilidad con otros sistemas, o al menos diferentes sistemas.

Con esto se puede evitar repetir IVs y asegurando que la comunicación sea segura frente a ataques de repeticiónes o manipulación de datos.

[44] ✓ 0s

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
import base64
from Crypto.Random import get_random_bytes

# Datos
key_hex = "E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74"
plaintext = "He descubierto el error y no volveré a hacerlo mal"

# Convertir clave a bytes
key = bytes.fromhex(key_hex)

# Generar nonce único de 12 bytes
nonce = get_random_bytes(12)

# Crear cifrador AES-GCM
aesgcm = Cipher(algorithms.AES(key), modes.GCM(nonce), backend=default_backend()).encryptor()

# Cifrar
ciphertext = aesgcm.update(plaintext.encode('utf-8')) + aesgcm.finalize()

# Tag de autenticación
tag = aesgcm.tag

# Mostrar resultados
print("Nonce (hex):", nonce.hex())
print("Texto cifrado (hex):", ciphertext.hex())
print("Texto cifrado (base64):", base64.b64encode(ciphertext).decode())
print("Tag de autenticación (hex):", tag.hex())
```

```
Nonce (hex): 2b9599c7f72a4b43b33352e5
Texto cifrado (hex): 8556081ac5c9b4af63dda5dc5c6375ae71ffd51f1ec3d75381594c94b97c74e1b4a8d824890adf543281810b88c83127cde2bf
Texto cifrado (base64): hVYIGsXJtK9j3aXcXGN1rnH/1R8ew9dTgVlMlLl8dOG0qNgkiQrfVDKBgQuIyDEnzeK/
Tag de autenticación (hex): 285a93b0e61a3de08a79f7855635e265
```

HE usado AES en modo GCM para cifrar de forma simétrica. Se convierte la clave de hex a bytes y se adapta el nonce a los 12 bytes de AES-GCM. El mensaje se cifra generando un tag de autenticación para integridad y autenticidad. Salen los resultados en hex y base64 para la verificación y compatibilidad con otros sistemas, o al menos diferentes sistemas.

Con esto se puede evitar repetir IVs y asegurando que la comunicación sea segura frente a ataques de repeticiónes o manipulación de datos.

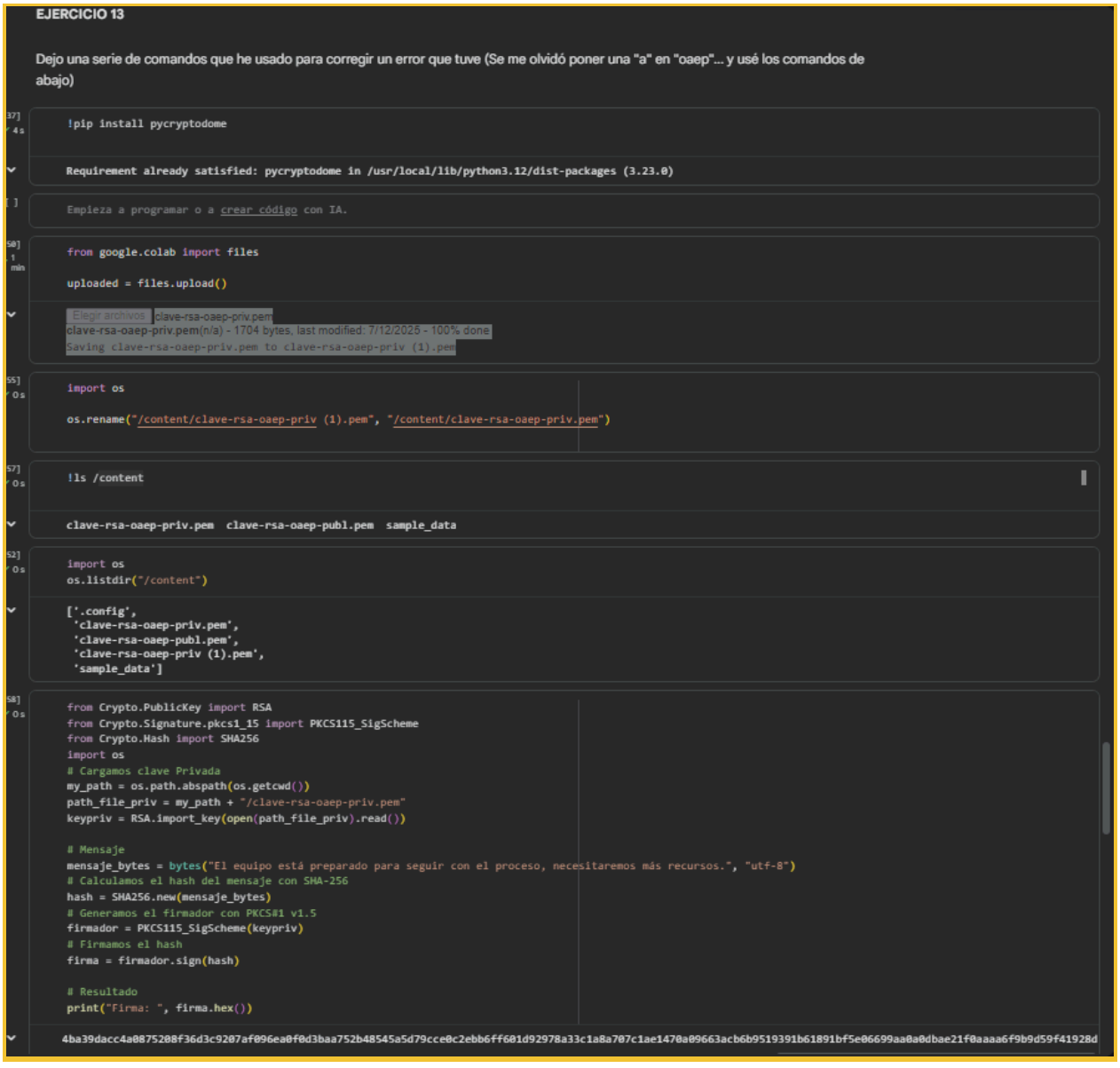
EJERCICIO XIII

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.



He calculado la firma RSA PKCS#1 v1.5 cargando la clave privada en formato PEM, firmado el mensaje usando SHA-256 para mostrar el resultado en hexadecimal. En el caso de la firma Ed25519 el archivo no estaba en un formato estándar compatible con la librería cryptography, por lo que he generado una clave temporal para poder ver el procedimiento de firmado.

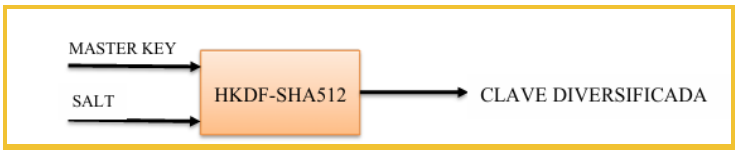
El mensaje se ha codificado en UTF-8 para evitar errores de caracteres no ASCII.

He usado la herramienta que tiene el propio Colab para ayudarme a entender el error y poder gestionarlo bien, me ha ayudado a inspeccionar el contenido del archivo, confirmar su formato y longitud. También me ha guiado para el manejo de las claves RSA y la firma.

EJERCICIO XIV

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

e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3



¿Qué clave se ha obtenido?

EJERCICIO 14

```
[2]
0s
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
import binascii

# Clave maestra del keystore (hex)
master_hex = "A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"
master_key = binascii.unhexlify(master_hex)

# Identificador del dispositivo (hex) = SALT
device_hex = "e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3"
salt = binascii.unhexlify(device_hex)

# HKDF-SHA512 para generar AES-256
hkdf = HKDF(
    algorithm=hashes.SHA512(),
    length=32,      # 32 bytes = AES-256
    salt=salt,
    info=b"",
)

derived_key = hkdf.derive(master_key)
print("Clave derivada (hex):", derived_key.hex())

...  Clave derivada (hex): e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a
```

Se convierte la clave maestra de hex a bytes. Configura la HKDF con SHA-512 con una longitud de salida de 32 bytes.

Luego añade el salt el info para que la derivación sea solamente única por cada dispositivo y luego devuelve la clave deruvada que está lista para poder usarse en un cifrado AES.

La clave derivada (Hex) es:

e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a

EJERCICIO XV

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?
¿Para qué algoritmo se ha definido la clave?
¿Para qué modo de uso se ha generado?
¿Es exportable?
¿Para qué se puede usar la clave?
¿Qué valor tiene la clave?

wrap/unwrap - En TR-31 el bloque de clave se protege usando 3DES con el mecanismo de Key Block Protection definido en ANSI X9.24-1. Para la “clave de transporte” se usa TDES Key Wrap ANSI X9.24-1 para envolver la clave.

Algoritmo de la clave definida - TR-31 te permite transportar claves para distintos algoritmos“. No puede determinarse el algoritmo interno porque el bloque no contiene un encabezado TR-31 válido.

Modo de uso Key Usage/Mode - Las claves TR-31 se etiquetan con un modo de uso

- PVK = Pin Verification Key
- CVK = Card Verification Key
- DEK = Data Encryption Key

Si es clave diversificada y TR-31 es una clave para cifrado de datos, DEK por ejemplo.

Exportabilidad - Una clave TR-31 marcada como D no es exportable fuera del sistema autorizado. No puede saberse la exportabilidad sin el encabezado TR-31.

Uso de la clave - TR-31 para cifrado de información sensible (p.ej., datos de tarjetas, PINs o transacciones). Puede ser usada en cifrado de bloque MAC o generación de otras claves derivadas según el Key Usage que se defina el bloque.

Valor de la clave - Para obtenerlo hay que desenvolver el bloque con la clave de transporte usando el algoritmo de unwrap 3DES. Hay que aplicar la operación unwrap TR31_block, transport_key para obtener la clave real en bytes.

Un bloque TR-31 nunca requiere modificaciones, si no ese bloque no es válido por lo que no debe alterarse.

EJERCICIO 15

El bloque TR-31 no debe modificarse nunca porque si se altera, deja de ser válido y por eso el código falla

Los datos usados KBPK/bloque no son correctos para hacer unwrap.

La clave importada tiene un valor de "C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1" en Hexadecimal

```
from psec import tr31

def importan():
    """Importa y muestra información de un Key Block TR-31"""
    kbpk_b = bytes.fromhex("A1A0101010101010101010101010102")
    kb_string = ("D0144DA0B005000042766B9265B2DF93A6E2E9B5813B77A2F616C8D515ACDB"
                 "E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D1B8239225B8E25657495E0"
                 "3CD85F73D7018E111B")


    kb.header.load(kb_string)
    kb = tr31.KeyBlock(kbpk_b)
    clave_unwrapped = kb.unwrap(kb_string)

    print("*" * 60)
    print("INFORMACIÓN DEL KEY BLOCK TR-31")
    print("*" * 60)
    print(f"Clave importada (hex): {clave_unwrapped.hex().upper()}")
    print(f"Versión:               {kb.header.version_id}")
    print(f"Uso de clave:          {kb.header.key_usage}")
    print(f"Algoritmo:             {kb.header.algorithm}")
    print(f"Modo de uso:           {kb.header.mode_of_use}")
    print(f"Exportabilidad:        {kb.header.exportability}")
    print(f"Número de versión:     {kb.header.version_num}")
    print("*" * 60)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipython-input-801378908.py in <cell line: 0>()
----> 1 from psec import tr31
      2
      3 def importan():
      4     """Importa y muestra información de un Key Block TR-31"""
      5     kbpk_b = bytes.fromhex("A1A0101010101010101010101010102")

ModuleNotFoundError: No module named 'psec'

---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

OPEN EXAMPLES

Pasos siguientes: Explicar error

Otra prueba

```
from psec import tr31

header, key = tr31.unwrap( kbpk=bytes.fromhex("A1A0101010101010101010101010102"), key_block="D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A
print(key.hex())

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

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipython-input-3221121645.py in <cell line: 0>()
----> 1 from psec import tr31
      2
      3 header, key = tr31.unwrap( kbpk=bytes.fromhex("A1A0101010101010101010101010102"),
key_block="D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37018E111B")
      4 print(key.hex())
      5

ModuleNotFoundError: No module named 'psec'

---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

OPEN EXAMPLES

Pasos siguientes: Explicar error