ADRIÁN LÓPEZ FERNÁNDEZ

CRIPTOGRAFÍA

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 B98A15BA31AEBB3F. ¿Qué clave será con la que se trabaje en memoria?

   Para calcular la clave que el Key Manager ha puesto en Properties es necesario **realizar un XOR** entre la clave fija y la clave final:

```python
#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())
```

   Por lo tanto el valor de Properties para desarrollo es: 20553975C31055ED

```
PS C:\Users\Adrián> & C:/Users/Adrián/AppData/Local/Microsoft/WindowsApps/python3.11.exe
20553975c31055ed
```

   Para calcular la clave con la que se trabaja en memoria se realiza un XOR de la clave fija (B1EF2ACFE2BAEEFF) y la clave Properties en production (B98A15BA31AEBB3F):

```python
#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("B98A15BA31AEBB3F")

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

```

```
PS C:\Users\Adrián> & C:/Users/Adrián/AppData/Local/Microsoft/WindowsApps/python3.11.exe
08653f75d31455c0
```

La clave final en memoria (producción) es: 08653F75D31455C0

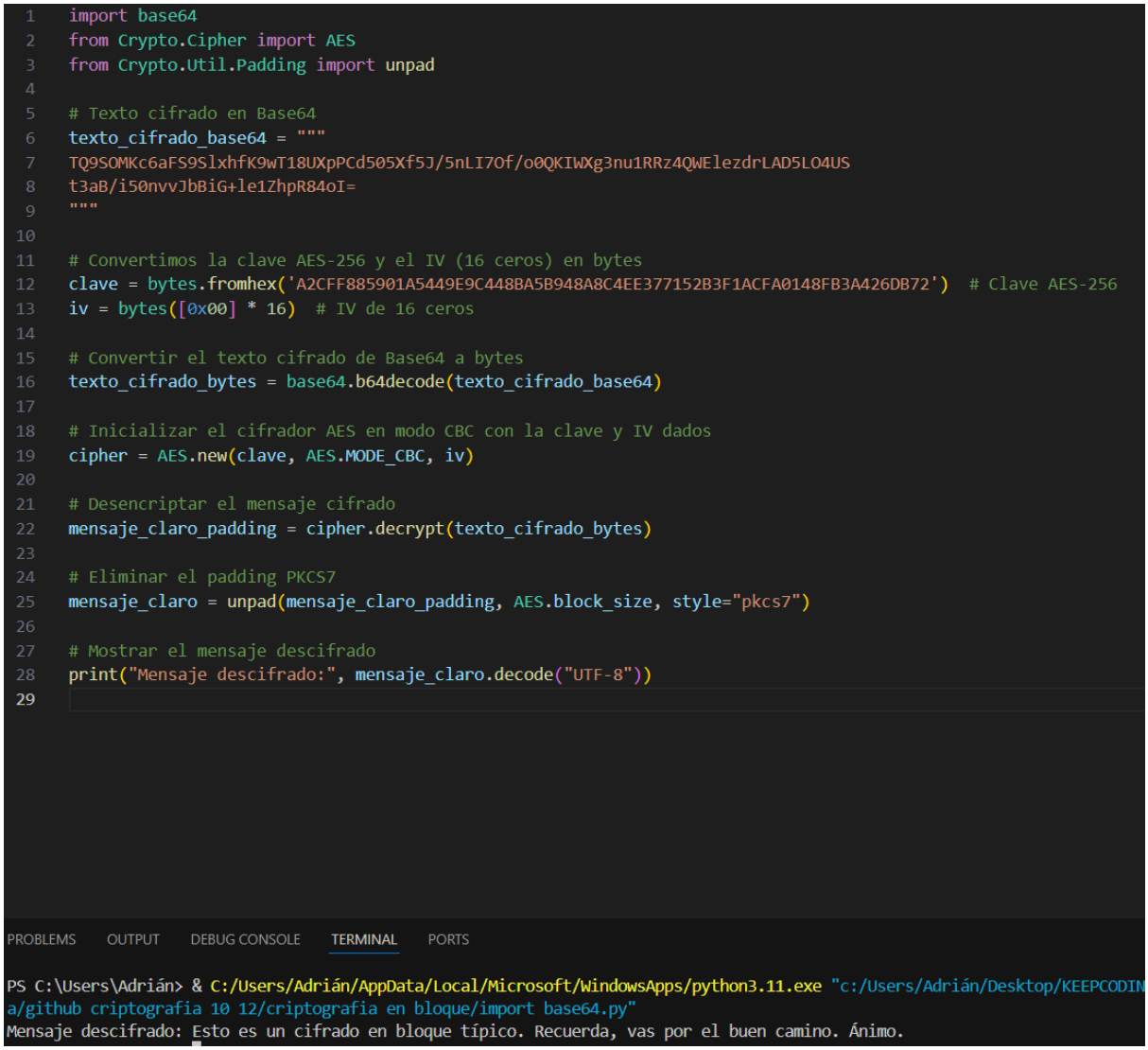
2. Dada la clave con etiqueta "cifrado-sim-aes-256" que contiene el keystore. El iv estará compuesto por el hexadecimal correspondiente a ceros binarios ("00"). Se requiere obtener el dato en claro correspondiente al siguiente dato cifrado:

TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5L O4US t3aB/i50nvvJbBiG+le1ZhpR84oI=

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



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

# Texto cifrado en Base64
texto_cifrado_base64 = """
TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4US
t3aB/i50nvvJbBiG+le1ZhpR84oI=
"""

# Convertimos la clave AES-256 y el IV (16 ceros) en bytes
clave = bytes.fromhex('A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72')  # Clave AES-256
iv = bytes([0x00] * 16)  # IV de 16 ceros

# Convertir el texto cifrado de Base64 a bytes
texto_cifrado_bytes = base64.b64decode(texto_cifrado_base64)

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

# Desencriptar el mensaje cifrado
mensaje_claro_padding = cipher.decrypt(texto_cifrado_bytes)

# Eliminar el padding PKCS7
mensaje_claro = unpad(mensaje_claro_padding, AES.block_size, style="pkcs7")

# Mostrar el mensaje descifrado
print("Mensaje descifrado:", mensaje_claro.decode("UTF-8"))
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Adrián> & C:/Users/Adrián/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/Adrián/Desktop/KEEPCODIN
a/github criptografia 10 12/criptografia en bloque/import base64.py"
Mensaje descifrado: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.
```

El tamaño del padding en ambos casos es e 1 byte, y aunque en este caso X923 no da error debido al esquema del padding (1 byte), **los esquemas de PKCS7 y X9.23 no son intercambiables**. En general, intentar descifrar un texto cifrado con PKCS7 usando X9.23 (o viceversa) **resultará en un error** o un texto incorrecto, especialmente si hay más de un byte de padding.

AES trabaja en bloques de 96 bytes y el texto tiene 84 en UTF-8, así que **se han añadido 12 bytes de padding al texto cifrado** para que su longitud total (96 bytes) sea un múltiplo de 16 bytes.

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-chacha20-256”. El nonce “9Yccn/f5nJJhAt2S”. El algoritmo que se debe usar es un Chacha20.

El texto cifrado resultante en base64 sería: aaxO58TFUlN6AKGbyvfwqu18nI92mVagm85vre9sNTXyIRyUZwZ89cSoQqs=

¿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.

Aunque el cifrado ChaCha20 confirma la confidencialidad del mensaje, no podemos saber con seguridad si el texto cifrado ha sido alterado por el camino.

Para añadir una capa extra de seguridad, podemos usar una **función de autenticación como el TAG**, un algoritmo HMAC que nos ayudaría a verificar si el mensaje ha sido modificado durante la transmisión.

A la hora de enviar el mensaje, el receptor recibe tanto el mensaje cifrado como el TAG para comprobar la integridad del mismo y que no ha sido modificado. (OK / KO)

```python
from Crypto.Protocol.KDF import scrypt
from Crypto.Random import get_random_bytes
from Crypto.Hash import SHA256
import hmac

# Datos
texto_plano = "KeepCoding te enseña a codificar y a cifrar"
clave = bytes.fromhex("AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120")  # 256 bits de clave
nonce = base64.urlsafe_b64decode("9Yccn/f5nJJhAt2S")  # Nonce en formato URL-safe Base64

# Cifrado con ChaCha20
cipher = ChaCha20.new(key=clave, nonce=nonce)
texto_cifrado = cipher.encrypt(texto_plano.encode())

# Calcular HMAC para la integridad
# Usamos SHA256 como función hash
hmac_hash = hmac.new(clave, texto_cifrado, SHA256).digest()

# Mostrar el resultado
print("Texto cifrado (Base64):", base64.b64encode(texto_cifrado).decode('utf-8'))
print("HMAC (Base64):", base64.b64encode(hmac_hash).decode('utf-8'))

# --- Verificación en el receptor ---

# El receptor recibe el texto cifrado y el HMAC
# Supongamos que los recibe en base64
texto_cifrado_recibido = base64.b64decode(base64.b64encode(texto_cifrado).decode('utf-8'))
hmac_recibido = base64.b64decode(base64.b64encode(hmac_hash).decode('utf-8'))

# Verificación del HMAC
hmac_verificacion = hmac.new(clave, texto_cifrado_recibido, SHA256).digest()

if hmac_verificacion == hmac_recibido:
    print("La integridad del mensaje está verificada. Este mensaje es auténtico.")
else:
    print("La integridad del mensaje no está garantizada. Alguien está intentando joderte.")

```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Adrián> & C:/Users/Adrián/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/Adrián/Desktop/KEEPCODING/Criptografía/Adrian python/chacha20hmac.py"
Texto cifrado (Base64): aaxO58TFUlN6AKGbyvfwqu18nI92mVagm85vre9sNTXyIRyUZwZ89cSoQqs=
HMAC (Base64): lCmx5yN6y0FRa2wBcXt2Ge2E4421THn2X/kPfyzSw3w=
La integridad del mensaje está verificada. El mensaje es auténtico.
PS C:\Users\Adrián>
```

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSB
sb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ.gfhw0
dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

¿Qué algoritmo de firma hemos realizado?

El valor “alg”: “HS256” en el header indica que el algoritmo de firma utilizado es HMAC256

¿Cuál es el body del jwt?

El body es {"usuario":"Don Pepito de los palotes","rol":"isNormal","iat":1667933533}

Un hacker está enviando a nuestro sistema el siguiente jwt:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBk
ZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9
.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2ClAODlHRI

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

El hacker está intentando escalar permisos al cambiar el rol de “Don Pepito de los palotes” de “isNormal” a “isAdmin”.

Si se intentase validar el JWT del hacker utilizando la biblioteca pyjwt, al utilizar la misma clave secreta ("Con KeepCoding aprendemos"), el token **no se validaría correctamente** porque la firma es incorrecta, ya que **el cuerpo ha sido modificado y la firma del hacker no correspondería con la esperada del body original**. La librería pyjwt lo detectaría y lanzaría un mensaje indicando que no la firma no puede verificarse.

5. El siguiente hash se corresponde con un SHA3 del texto “En KeepCoding aprendemos cómo protegernos con criptografía”.

Bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

¿Qué tipo de SHA3 hemos generado?

Hemos generado un SHA3-256 ya que el hash tiene 64 caracteres hexadecimales (32 bytes, porque los caracteres hexadecimales siempre son el doble de los bytes) lo que equivaldría a 256 bits (8 bits en cada byte x 32 caracteres, 8x32 = 256 bits)

Y si hacemos un SHA2, y obtenemos el siguiente resultado: 4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f 6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833 ¿Qué hash hemos realizado?

Hemos generado un SHA2-512, ya que el hash tiene 128 caracteres hexadecimales o 64 bytes. Teniendo en cuenta que cada byte son 8 bits (8 bits por byte × 64 = 512 bits).

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 texto resultante sería "302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf"

La propiedad que destacaría es la **diferencia en la longitud del hash entre SHA2-512 y SHA3-256**. SHA2-512 genera un hash de 512 bits (128 caracteres hexadecimales), mientras que SHA3-256 genera un hash de 256 bits (64 caracteres hexadecimales), ambos son muy seguros ante colisiones aunque SHA3 nace para evitar ataques de longitud sobre SHA2. Cabe destacar también el efecto avalancha que cambiaría radicalmente el hash resultante en caso de que haya un cambio mínimo en el mensaje, además de su capacidad de compresión, ya que la longitud del contenido no afecta a la longitud del hash resultante, que será siempre la misma tanto para hashear un palabra como un libro.

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.

El HMAC-SHA256 resultante es:
857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550

Además de CyberChef, también se ha comprobado a través de script:

7. Trabajamos en una empresa de desarrollo que tiene una aplicación web, la cual requiere un login y trabajar con passwords.

Nos preguntan qué mecanismo de almacenamiento de las mismas proponemos. Tras realizar un análisis, el analista de seguridad propone un hash SHA-1. Su responsable, le indica que es una mala opción.

¿Por qué crees que es una mala opción?

El SHA-1 a día de hoy no es un algoritmo seguro y es **vulnerable a ataques de colisión**, por lo que su uso no está recomendado por el NIST (especialmente para almacenar contraseñas).

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?

El problema de almacenar las contraseñas con un SHA-256 radica en que, aunque es un algoritmo seguro, **un atacante podría averiguar las contraseñas utilizando bibliotecas o Rainbow Tables** con el hash SHA-256 de múltiples contraseñas típicas, por lo que podría terminar encontrando una vulnerabilidad a través de este método de fuerza bruta.

Para prevenir el precálculo de hashes añadiría al hash **un valor aleatorio conocido como "Salt"**, de esta forma, te aseguras de cada hash es único:

Hash(salt + password) -> Evita las rainbow tables

También sería recomendable usar funciones que implementen hashing iterativo, como el Argon2.

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

Además del Salt, sería posible añadir **otra capa más de entropía conocida como "Pepper"** para

fortalecer el hash todavía más. Se trata de un valor secreto que se combina con la contraseña antes de hashearla y no se almacena en la base de datos, sino en un entorno seguro como un HSM, lo que añade una capa extra de seguridad ante una fuga de datos.

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?

En caso de no usar TLS 1.2 o 1.3 sería imprescindible rediseñar la comunicación para incluir las medidas de confidencialidad e integridad en la API.

Para proteger los datos sensibles (usuasrio, tarjeta, movimientos y saldo), utilizaría **AES-256-GCM**. Este algoritmo de cifrado simétrico garantiza la **confidencialidad** de los datos cifrados, y también incluye un **tag de autenticación** que asegura la **integridad** del mensaje. Esto elimina la necesidad de aplicar una capa separada para verificar integridad.

Proceso:

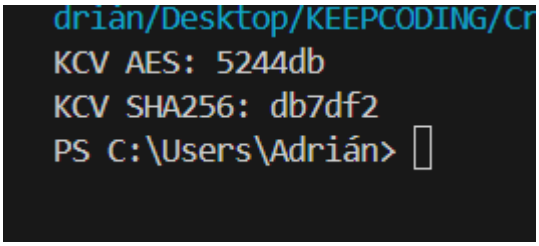
1. **Intercambio seguro de claves**:
   - Cliente y servidor intercambiarían una clave simétrica utilizando un protocolo seguro como **Diffie-Hellman** (por ejemplo DH basado en curvas elípticas).
   - Esto asegura que solo las partes legítimas (cliente y servidor) compartan la clave necesaria para cifrar y descifrar los mensajes.
2. **Cifrado de los datos sensibles**:
   - En el lado del cliente, se cifrarían los campos sensibles como usuario y tarjeta utilizando AES-256-GCM.
   - Los datos cifrados, junto con el tag de autenticación generado por AES-GCM, se incluirían en el mensaje enviado al servidor.
3. **Verificación en el servidor**:
   - El servidor descifra los datos utilizando la clave compartida y valida el tag de autenticación para asegurarse de que el mensaje no ha sido alterado en tránsito.

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.

El KCV(SHA-256) es db7df2

El KCV(AES) es 5244db



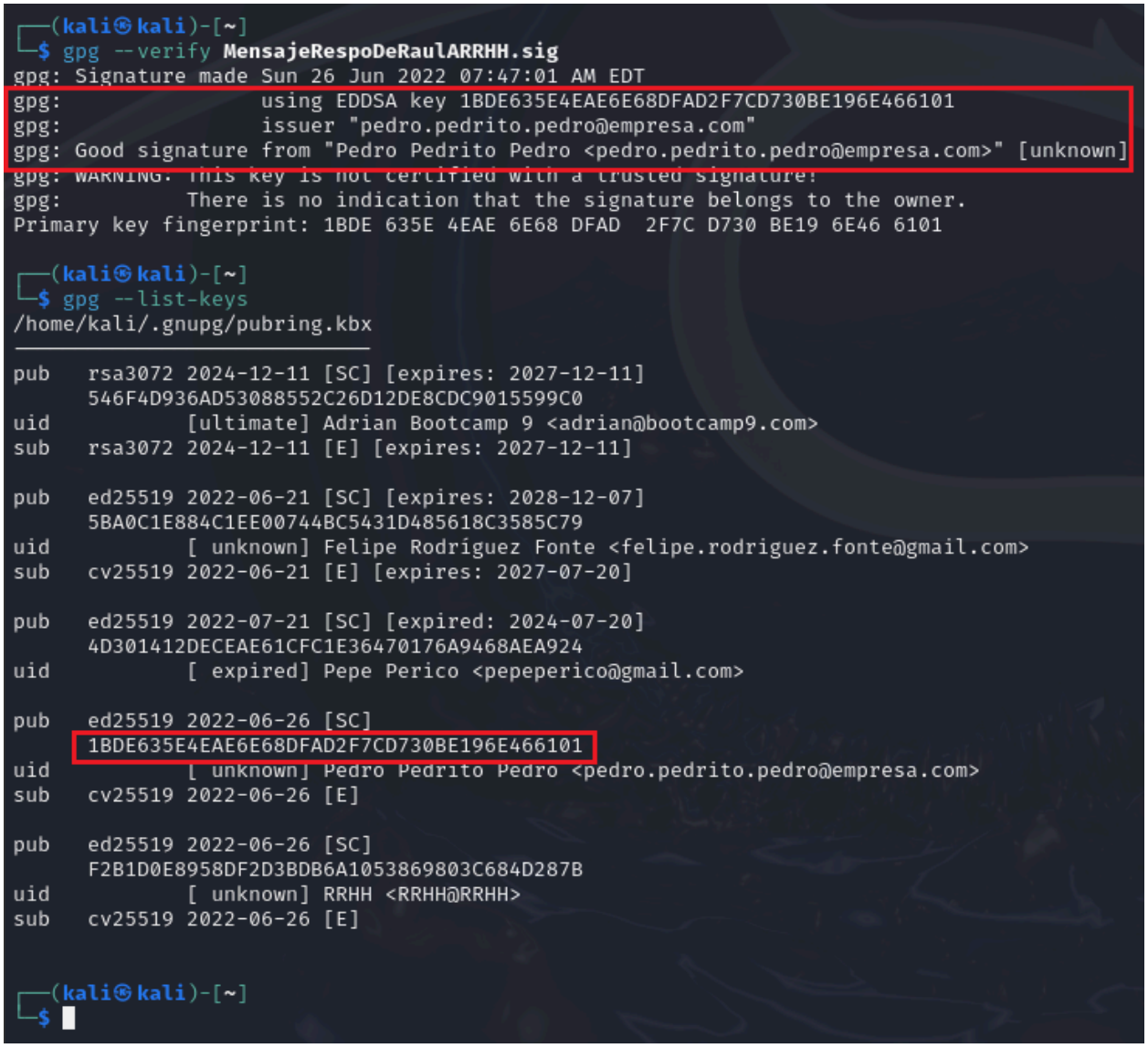
```python
import hashlib
import json
from base64 import b64encode, b64decode
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad



#Cifrar una cadena de 16 bytes de ceros binarios.
textoPlano_bytes = bytes.fromhex('00000000000000000000000000000000')
#Con la clave que quieres calcular el kcv
clave = bytes.fromhex('A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72')
iv_bytes = bytes.fromhex('00000000000000000000000000000000')
cipher = AES.new(clave, AES.MODE_CBC,iv_bytes)
texto_cifrado_bytes = cipher.encrypt(pad(textoPlano_bytes, AES.block_size,  style='pkcs7'))
print("KCV AES:", texto_cifrado_bytes.hex()[0:6])



m = hashlib.sha256()
m.update(bytes.fromhex("A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"))
print("KCV SHA256: " + m.digest().hex()[0:6])
```

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.

```
┌──(kali㉿kali)-[~]
└─$ sudo nano mensajerrhh.txt
[sudo] password for kali:

┌──(kali㉿kali)-[~]
└─$ gpg --list-secret-keys
/home/kali/.gnupg/pubring.kbx
------------------------------
sec   rsa3072 2024-12-11 [SC] [expires: 2027-12-11]
      546F4D936AD53088552C26D12DE8CDC9015599C0
uid           [ultimate] Adrian Bootcamp 9 <adrian@bootcamp9.com>
ssb   rsa3072 2024-12-11 [E] [expires: 2027-12-11]

sec   ed25519 2022-07-21 [SC] [expired: 2024-07-20]
      4D301412DECEAE61CFC1E36470176A9468AEA924
uid           [ expired] Pepe Perico <pepeperico@gmail.com>

sec   ed25519 2022-06-26 [SC]
      1BDE635E4EAE6E68DFAD2F7CD730BE196E466101
uid           [ unknown] Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>
ssb   cv25519 2022-06-26 [E]

sec   ed25519 2022-06-26 [SC]
      F2B1D0E8958DF2D3BDB6A1053869803C684D287B
uid           [ unknown] RRHH <RRHH@RRHH>
ssb   cv25519 2022-06-26 [E]


┌──(kali㉿kali)-[~]
└─$ BDE

┌──(kali㉿kali)-[~]
└─$ gpg --output mensaje-rrhh-firmado.sig --clearsign -u F2B1D0E8958DF2D3BDB6A1053869803C684D287B mensajerrhh.txt

┌──(kali㉿kali)-[~]
└─$ cat mensaje-rrhh-firmado.sig
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

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

iHUEARYIAB0WIQTysdDolY3y0722oQU4aYA8aE0oewUCZ2XMSAAKCRA4aYA8aE0o
eyLbAQCNNgWR/n4XYy2zOBPWJUDCn34UhUeL1FI3cNfZEO4ugAEA0nej40vQ/R4Z
uUMF9/UyjsiK27Th8ec6wlNcYQ1Ouwc=
=b7gg
-----END PGP SIGNATURE-----
```

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.


File Actions Edit View Help
└─$ sudo nano mensajefinal.txt
[sudo] password for kali:
┌──(kali㉿kali)-[~]
└─$ gpg --list-keys
/home/kali/.gnupg/pubring.kbx
pub rsa3072 2024-12-11 [SC] [expires: 2027-12-11]
546F4D936AD53088552C26D12DE8CDC9015599C0
uid [ultimate] Adrian Bootcamp 9 <adrian@bootcamp9.com>
sub rsa3072 2024-12-11 [E] [expires: 2027-12-11]
pub ed25519 2022-06-21 [SC] [expires: 2028-12-07]
5BA0C1E884C1EE00744BC5431D485618C3585C79
uid [unknown] Felipe Rodriguez Fonte <felipe.rodriguez.fonte@gmail.com>
sub cv25519 2022-06-21 [E] [expires: 2027-07-20]
pub ed25519 2022-07-21 [SC] [expired: 2024-07-20]
4D301412DECEAE61CFC1E36470176A9468AEA924
uid [expired] Pepe Perico <pepeperico@gmail.com>
pub ed25519 2022-06-26 [SC]
1BDE635E4EAE6E68DFAD2F7CD730BE196E466101
uid [unknown] Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>
sub cv25519 2022-06-26 [E]
pub ed25519 2022-06-26 [SC]
F2B1D0E8958DF2D3BDB6A1053869803C684D287B
uid [unknown] RRHH <RRHH@RRHH>
sub cv25519 2022-06-26 [E]
┌──(kali㉿kali)-[~]
└─$ gpg --output mensajefinal.gpg --encrypt --armor --recipient 1BDE635E4EAE6E68DFAD2F7CD730BE196E466101 --recipient F2B1D0E8958DF2D3BDB6A10538698
03C684D287B mensajefinal.txt
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.
Use this key anyway? (y/N) y
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
┌──(kali㉿kali)-[~]
└─$ cat mensajefinal.gpg
-----BEGIN PGP MESSAGE-----
hF4DJdbQKUA1tlASAQdAvmYNUeG4eW+ZwcFtNwFaZHTlvRwhsoj9n6QD14roAWgw
YUU2apBUWp5Xl4iEPrzJjOXzAv/RSz5xOYTXcOUATpbFpHQ24lk2ZkEJgpZBjhvr
hF4DfBpG6iCwVG8SAQdApB8BuQk+hs6l3RKNCTVTNhr5z+pFKMEK0+OXyZ2Yq3Aw
zynOff11w9rVcGEQxOR9UYdLgejfzs85RJYFGzqo3Hw0Sh/Zza6GiuSpCqTurcEq
0pgBRrJUQC2k+zd9nuyRHimTiRi45B/Kcx6DhicCdkZ7ue5JuLdbnbKLxfFTTsBN
v2WgwLCmb/I0pWW/R/8Opuh7ZahXdr+7rAOVpcCVhT9+P0CqL7wBk0/hGvXXF43M
MIIi6Yx8pRbUoLPGJ5VDYNBCcdtvuBi55o15cJSCn4QxWEFy6FllroqqbVouwgRz
+GeGTO0k9dhDkQ==
=MEnZ
-----END PGP MESSAGE-----
┌──(kali㉿kali)-[~]
└─$

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
df3c50e05d5bf63768088226e2a9177485c54f72407fdf358fe64479677d8296ad38c6f177ea7cb
74927651cf24b01dee27895d4f05fb5c161957845cd1b5848ed64ed3b0372
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?

```
PS C:\Users\Adrián> & C:/Users/Adrián/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/A
drián/Desktop/KEEPCODING/Criptografía/github criptografia 10 12/criptografia asimetrica e hibrida/RS
A-OAEP2.py"
Clave simétrica recuperada: e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72
Nuevo cifrado: 1cbb5a6a164af77fa6e21e8119fd4f507161bbdf245d62bca29a4cbf69d61b5360158f063069d7cfa253a
93bfad4e78e2d4907a7b5721485d2f04ca9feb20f66475fef9ff7a564edbf549b25b983876fed93fabd3135b201f879d49d3
74dd9ae845b8d3560faf024f004e205292f66bc3c659ece357c940b8c9dd003af0a60fcda474730bbd7af8b74f8d1ff4253f
2ce34e0d2348b6a5dd94c9b0a1ba5dc588c1be07152f7cd9dc16d17606a77d062f522388e95a9f8e1ea0576ea532237f2e30
6c318997ff3e5bfdc5061cf2a524a7731a2a14dced47fe8bbf43917bd0da2579520dbceaa3157694554b16676c8472112645
5418126f0ea19529396faecc726
PS C:\Users\Adrián>
```

Pese a usar la misma clave pública y privada, así como el mismo texto de origen, se crea un texto cifrado diferente debido a que **RSA-OAEP utiliza Padding aleatorio** en cada operación, **generando valores aleatorios que cambian el resultado** cada vez que se ejecuta, algo que le añade seguridad frente a ataques de análisis de patrones.

12. Nos debemos comunicar con una empresa, para lo cual, hemos decidido usar un algoritmo como el AES/GCM en la comunicación. Nuestro sistema, usa los siguientes datos en cada comunicación con el tercero:
Key:E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74

Nonce:9Yccn/f5nJJhAt2S

¿Qué estamos haciendo mal? Cifra el siguiente texto: He descubierto el error y noolveré a hacerlo mal Usando para ello, la clave, y el nonce indicados. El texto cifrado presentalo en hexadecimal y en base64.

```python
# Datos proporcionados
texto_original = "He descubierto el error y no volveré a hacerlo mal"
clave = bytes.fromhex('E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74')
nonce = b"9Yccn/f5nJJhAt2S"[:12]  # Ajustado a 12 bytes
datos_asociados = "Datos asociados para integridad y autenticación."

# CIFRADO
print("=== CIFRADO ===")
textoPlano_bytes = texto_original.encode('utf-8')
datos_asociados_bytes = datos_asociados.encode('utf-8')

# Crear objeto de cifrado
cipher = AES.new(clave, AES.MODE_GCM, nonce=nonce)
cipher.update(datos_asociados_bytes)
texto_cifrado_bytes, tag = cipher.encrypt_and_digest(textoPlano_bytes)

# Convertir a formatos legibles
texto_cifrado_hex = texto_cifrado_bytes.hex()
texto_cifrado_b64 = b64encode(texto_cifrado_bytes).decode('utf-8')
tag_b64 = b64encode(tag).decode('utf-8')
nonce_b64 = b64encode(nonce).decode('utf-8')

print("Texto cifrado (Hex):", texto_cifrado_hex)
print("Texto cifrado (Base64):", texto_cifrado_b64)
print("Tag (Base64):", tag_b64)
print("Nonce (Base64):", nonce_b64)

# DESCIFRADO
print("\n=== DESCIFRADO ===")
try:
    # Crear objeto de descifrado
    decipher = AES.new(clave, AES.MODE_GCM, nonce=nonce)
    decipher.update(datos_asociados_bytes)
    texto_descifrado_bytes = decipher.decrypt_and_verify(bytes.fromhex(texto_cifrado_hex),
    texto_descifrado = texto_descifrado_bytes.decode('utf-8')

    print("Texto descifrado:", texto_descifrado)
except Exception as e:
    print("Error al descifrar:", e)

```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

rto.py"
=== CIFRADO ===
Texto cifrado (Hex): 79f86701c1b017ad5903913bfe0ea33069c2c3064a50a0f6c023b12eadb5ff89c357eb26b826915
96f41a0c4db329f14222d97
Texto cifrado (Base64): efhnAcGwF61ZA5E7/g6jMGnCwwZKUKD2wCOxLq21/4nDV+smuCaRWW9BoMTbMp8UIi2X
Tag (Base64): QKA+GjEFJ67A/WOryz5Ngg==
Nonce (Base64): OVljY24vZjVuSkpo

=== DESCIFRADO ===
Texto descifrado: He descubierto el error y no volveré a hacerlo mal
```

Hexadecimal:
79f86701c1b017ad5903913bfe0ea33069c2c3064a50a0f6c023b12eadb5ff89c357eb26b826915
96f41a0c4db329f14222d97

Base64:
efhnAcGwF61ZA5E7/g6jMGnCwwZKUKD2wCOxLq21/4nDV+smuCaRWW9BoMTbMp8UIi2X
Tag (Base64): QKA+GjEFJ67A/WOryz5Ngg==

El problema se debe a que **nunca se debería usar el mismo nonce para cada comunicación y debería ser aleatorio**.

Es aceptable usar una clave fija en las comunicaciones durante un tiempo, siempre y cuando el nonce no se repita entre ellas, en caso de utilizar el mismo nonce y una clave fija **se exponen patrones en los datos cifrados lo que facilitaría un ataque**. Si se usase una clave fija es recomendable rotarla cada cierto tiempo, pero el nonce siempre tiene que cambiar para cada comunicación.

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?

a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92a
a3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959
bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a9ec79
8fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f838384
6d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa
752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b61891
bf5e06699aa0a0dbae21f0aaaa6f9b9d59f41928d

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

Se ha generado la firma utilizando la clave privada y se ha verificado con la pública

Firma Generada (Hexadecimal):
626633323539326463323335613236653331653233313036336131393834626237356666643964633535353063663330313035393131636134353630646162353261626234306534663765326433616638323861626163313436376439356436363861383033393565306137316335313739386264353434363962373336306 4

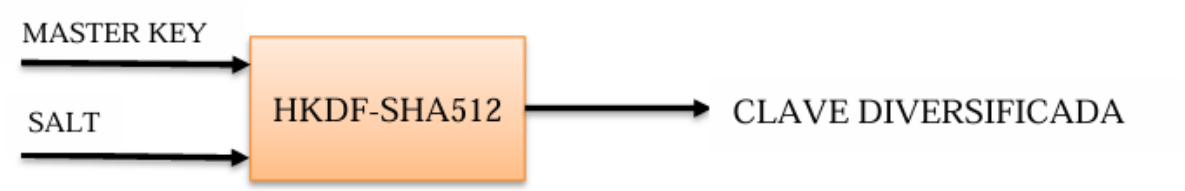
```python
import ed25519

publickey = open("C:/Users/Adrián/Desktop/KEEPCODING/Criptografía/github criptografia 10 12/Practic
privatekey = open("C:/Users/Adrián/Desktop/KEEPCODING/Criptografía/github criptografia 10 12/Practi


signedKey = ed25519.SigningKey(privatekey)
msg = bytes('El equipo está preparado para seguir con el proceso, necesitaremos más recursos.','utf
signature = signedKey.sign(msg, encoding='hex')
#signature = b'78613ae771b5b220d2a72008d7cf3bc6f232547692b02d38ffc671c70eb2015dc697252fbefa087995b2
print("Firma Generada (64 bytes):", signature)
signature_hex = signature.hex()
print("Firma Generada (Hexadecimal):", signature_hex)


try:
    verifyKey = ed25519.VerifyingKey(publickey.hex(),encoding="hex")
    verifyKey.verify(signature, msg, encoding='hex')
    print("La firma es válida")
except:
    print("Firma inválida!")

```

```
PROBLEMS 5  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Adrián> & C:/Users/Adrián/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/Adrián/Deskto
p/KEEPCODING/Criptografía/github criptografia 10 12/Practica/Ed25519-import.py"
Firma Generada (64 bytes): b'bf32592dc235a26e31e231063a1984bb75ffd9dc5550cf30105911ca4560dab52abb40e4f7e2d3af828
abac1467d95d668a80395e0a71c51798bd54469b7360d'
Firma Generada (Hexadecimal): 6266333235393264633233356132366533316532333130363361313938346262373566666439646335
3535306366333031303539313163613435363064616235326162623430653466376532643361663832386162616331343637643935643636
3861383033393565306137316335313739386264353434363962373336306 4
La firma es válida
```

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:

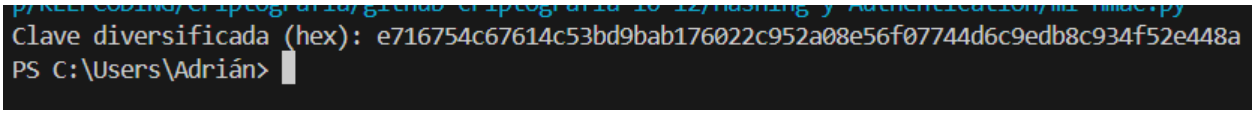
E43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3



¿Qué clave se ha obtenido?

Clave diversificada (hex):
e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a

15. Nos envían un bloque TR31:



- **¿Con qué algoritmo se ha protegido el bloque de clave?**
  - AES (Key block protected using the 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 (Both Encrypt & Decrypt / Wrap & Unwrap)
- **¿Es exportable?**
  - Sí (Sensitive, exportable under untrusted key)

- **¿Para qué se puede usar la clave?**
    - Clave simétrica para el cifrado de datos (Symmetric Key for Data Encryption)

- **¿Qué valor tiene la clave?**
    - C1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1

```
Clave: c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1
Key Version ID: D
Algoritmo: A
Modo de uso: B
Uso de la clave: D0
Exportabilidad: S
```

https://github.com/knovichikhin/psec/blob/master/psec/tr31.py

```python
from psec import tr31

#Documentado en este fichero
#https://github.com/knovichikhin/psec/blob/master/psec/tr31.py

#Luigy y Felipe hemos compartido una clave (KEK) -- KEK=A1A10101010101010101010101010103 --> ZMK
# TR31 (Key Block)= D0144D0AB00S000061653A93F145CA939753BAADE92C5BB69523F8D15EA97FE416BF3AA266F6962648882A66F2D21F44AEC1DF879068B802F13FB925CB4773D70D861DA2C9D75D30



header, key = tr31.unwrap( kbpk=bytes.fromhex("A1A10101010101010101010101010102"), key_block="D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37018E111B")
print("Clave=", 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)

header2, key2 = tr31.unwrap( kbpk=bytes.fromhex("A1A10101010101010101010101010102"), key_block="D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37018E111B")
print("Clave=", key2.hex())

print("Key Version ID: " + header2.version_id )
print("Algoritmo: " + header2.algorithm)
print("Modo de uso: " + header2.mode_of_use)
print("Uso de la clave: " + header2.key_usage)
print("Exportabilidad: " + header2.exportability)
```