PRÁCTICA: 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?

```
Parte dinámica (properties) en desarrollo: 20553975c31055ed
```

La clave fija, recordemos es B1EF2ACFE2BAEEFF, mientras que en producción sabemos que la parte dinámica que se modifica en los ficheros de propiedades es B98A15BA31AEBB3F.

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

```
Clave final obtenida en producción: 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: 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.
```

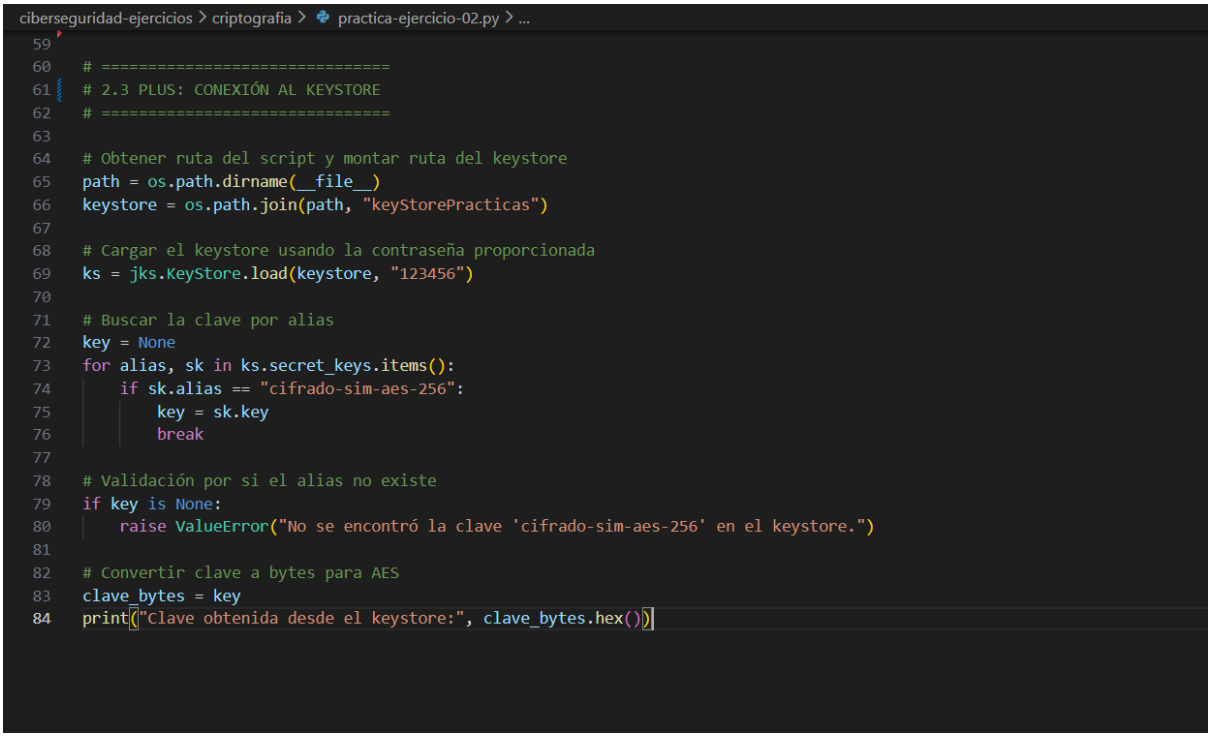
¿Qué ocurre si decidimos cambiar el padding a x923 en el descifrado?

```
Funciona perfectamente
```

¿Cuánto padding se ha añadido en el cifrado?

```
Se añadió 1 byte: \x01
```

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



---

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

```
69ac4ee7c4c552537a00a19bcaf7f0aaed7c9c8f769956a09bce6fadef6c3535f2
211c9467067cf5c4a842ab
```

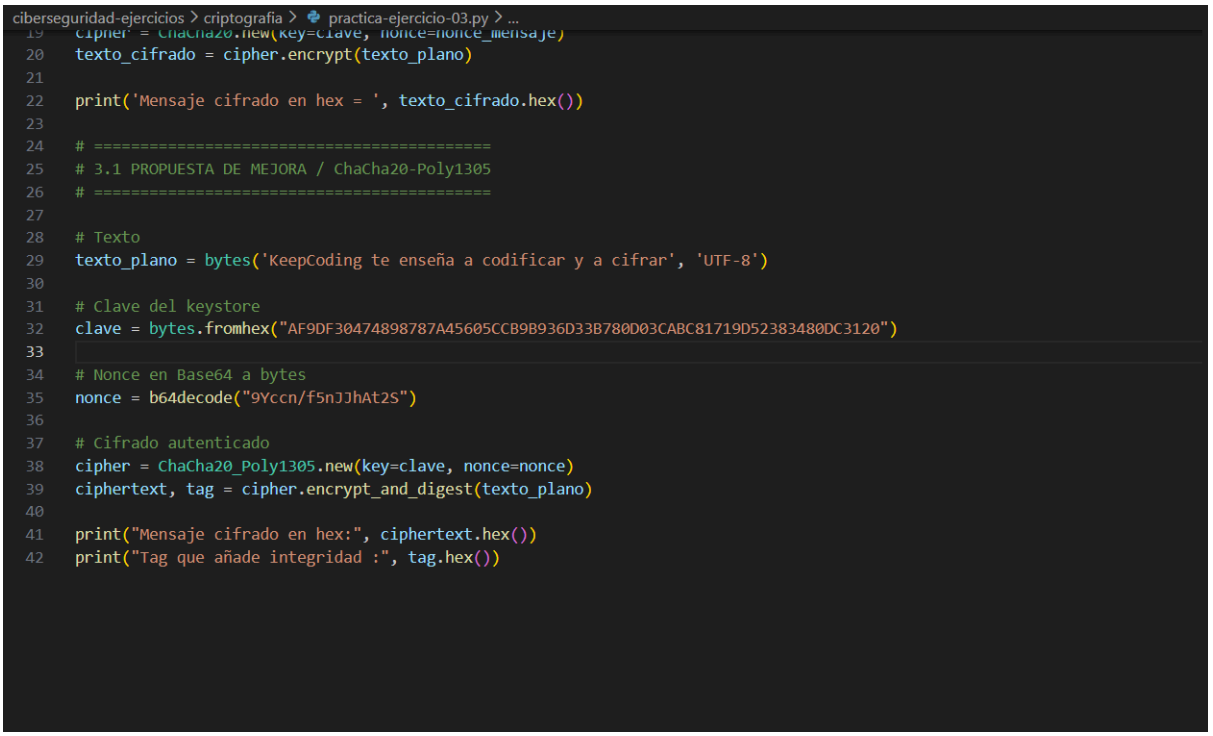
¿Cómo podríamos mejorar de forma sencilla el sistema, de tal forma, que no sólo garanticemos la confidencialidad sino, además, la integridad del mismo?

```
Una forma sencilla de mejorar el sistema es sustituir ChaCha20 por
ChaCha20-Poly1305, una variante que no solo cifra el mensaje sino
que además genera un tag de autenticación que permite verificar
que el contenido no ha sido modificado. Con este cambio se obtiene
un cifrado autenticado (AEAD), lo que garantiza simultáneamente
```

```
confidencialidad e integridad, evitando que un atacante pueda
alterar el ciphertext sin ser detectado.
```

Se requiere obtener el dato cifrado, demuestra, tu propuesta por código, así como añadir los datos necesarios para evaluar tu propuesta de mejora.

```
Mensaje cifrado en hex:
4ec95921ca8b757e2336605c7dbab8f4d40b5b4d220e66aa978f740d3e59b0cd70
e3217242991cc140bb1e1a
Tag que a◆ade integridad : 710cd4723da6d5ef37f23bee66285e57
```



```python
texto_cifrado = cipher.encrypt(texto_plano)

print('Mensaje cifrado en hex = ', texto_cifrado.hex())

# ========================================
# 3.1 PROPUESTA DE MEJORA / ChaCha20-Poly1305
# ========================================

# Texto
texto_plano = bytes('KeepCoding te enseña a codificar y a cifrar', 'UTF-8')

# Clave del keystore
clave = bytes.fromhex("AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120")

# Nonce en Base64 a bytes
nonce = b64decode("9Yccn/f5nJJhAt2S")

# Cifrado autenticado
cipher = ChaCha20_Poly1305.new(key=clave, nonce=nonce)
ciphertext, tag = cipher.encrypt_and_digest(texto_plano)

print("Mensaje cifrado en hex:", ciphertext.hex())
print("Tag que añade integridad :", tag.hex())
```

---

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0by BkZSB sb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ .gfhw0 dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

¿Qué algoritmo de firma hemos realizado?

```
Algoritmo HMAC-SHA256
Header original: {'typ': 'JWT', 'alg': 'HS256'}
```

¿Cuál es el body del jwt?

```
[Running] python -u
"c:\Proyectos\ciberseguridad-ejercicios\criptografia\practica-ejer
cicio-04.py"
Header original: {'typ': 'JWT', 'alg': 'HS256'}
Payload original (sin verificar): {'usuario': 'Don Pepito de los
palotes', 'rol': 'isNormal', 'iat': 1667933533}
Payload hacker (sin verificar): {'usuario': 'Don Pepito de los
palotes', 'rol': 'isAdmin', 'iat': 1667933533}
JWT original v◆lido: {'usuario': 'Don Pepito de los palotes',
'rol': 'isNormal', 'iat': 1667933533}
```

Un hacker está enviando a nuestro sistema el siguiente jwt:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBk
ZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9
.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODlHRI

¿Qué está intentando realizar?

```
Está intentando elevar privilegios para hacerse pasar por
administrador.
```

¿Qué ocurre si intentamos validarlo con pyjwt?

```
JWT del hacker rechazado: Signature verification failed
```

---

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?

```
64 caracteres hex
64 × 4 = 256 bits
```

```
Por tanto: Es un SHA3-256 (Keccak-256)
```

Y si hacemos un SHA2, y obtenemos el siguiente resultado:
4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f
6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

¿Qué hash hemos realizado?

```
128 caracteres hex
128 × 4 = 512 bits

Por tanto: Es un SHA-512
```

Genera ahora un SHA3 Keccak de 256 bits con el siguiente texto: “En KeepCoding aprendemos cómo protegernos con criptografía.”

```
Nuevo SHA3-256:
302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf
```

¿Qué propiedad destacarías del hash, atendiendo a los resultados anteriores?

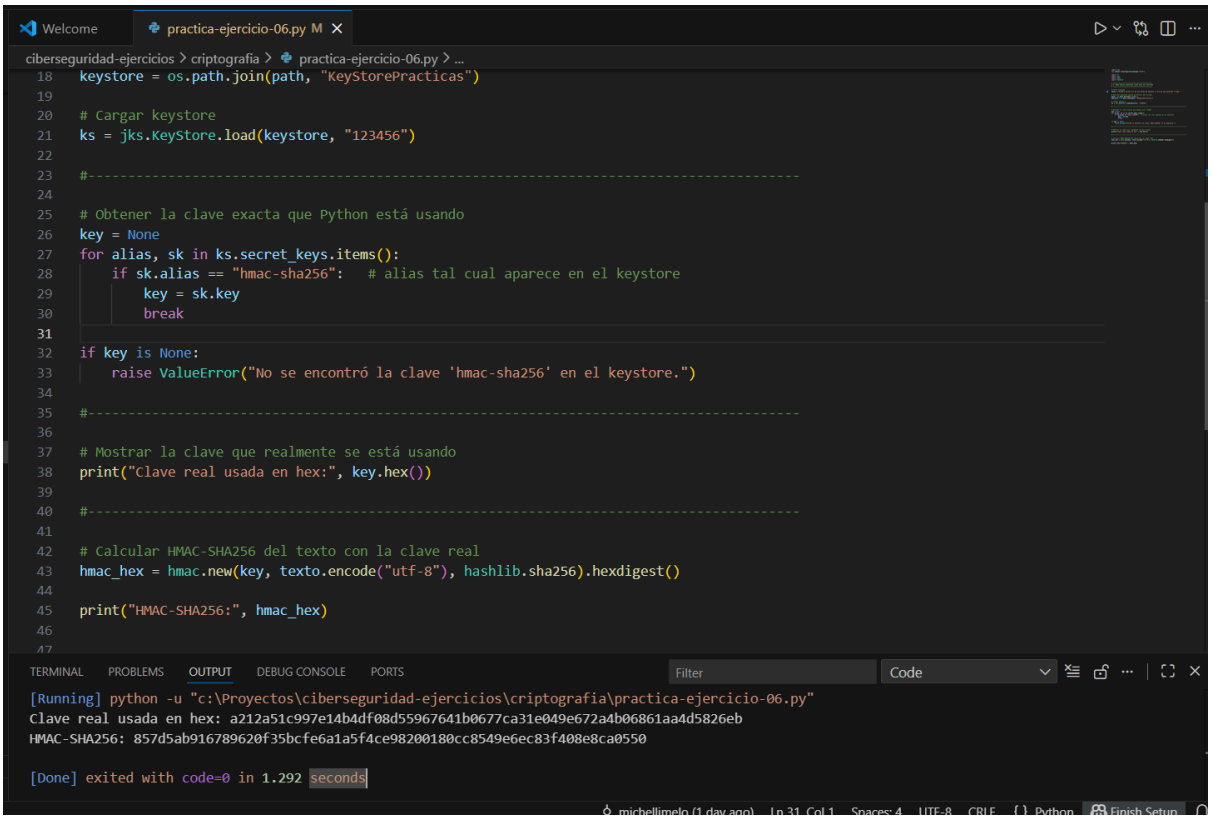
```
La propiedad más destacable es el efecto avalancha. Este principio
indica que un cambio mínimo en el mensaje original, incluso algo
tan pequeño como añadir un punto, produce un hash completamente
distinto y sin relación aparente con el anterior. Esto garantiza
que los hashes sean impredecibles y que cualquier modificación del
texto, por insignificante que sea, genere un resultado totalmente
nuevo, reforzando la integridad y la seguridad del sistema.
```

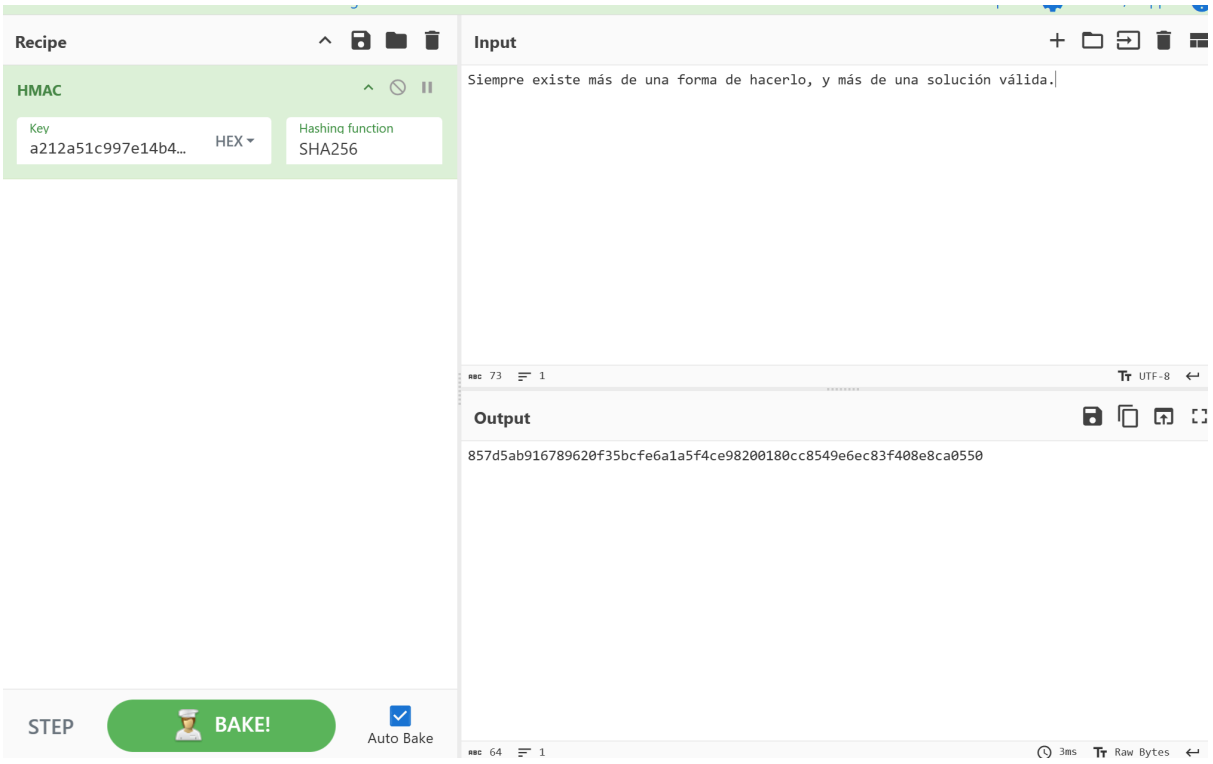
---

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.

```
HMAC-SHA256:
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 considerado inseguro desde hace años porque ya existen ataques prácticos capaces de encontrar colisiones, lo que significa que un atacante puede fabricar dos mensajes distintos con el mismo hash. Además, SHA-1 es extremadamente rápido, y eso lo convierte en una mala opción para almacenar contraseñas: un atacante puede probar millones de combinaciones por segundo, lo que hace muy viable un ataque por fuerza bruta o diccionario. En otras palabras, SHA-1 no está diseñado para proteger contraseñas y hoy en día se considera obsoleto y vulnerable.

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 mejor que SHA-1, sigue siendo un algoritmo muy rápido, y eso lo hace poco adecuado para guardar contraseñas por sí solo. Para fortalecerlo, lo mínimo sería añadir ‘salt’, que es un valor aleatorio diferente para cada usuario, de manera que dos personas con la misma contraseña no generen el mismo hash. Además, podríamos aplicar iteraciones, es decir, repetir el proceso del hash muchas veces para que calcularlo cueste más tiempo. Esto no afecta al usuario al iniciar sesión, pero sí hace mucho más difícil que un atacante pruebe combinaciones de contraseñas usando fuerza bruta. También podríamos añadir ‘pepper’, una clave secreta guardada en el servidor, para añadir una capa extra de protección si roban la base de datos.

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

Aunque usar SHA-256 con mecanismos de fortalecimento mejora mucho la seguridad respecto a un hash simple, todavía no es la mejor opción para guardar contraseñas. Hoy en día existen herramientas diseñadas específicamente para este propósito, que hacen que romper una contraseña sea mucho más difícil incluso si alguien consigue la base de datos. Lo más recomendable sería usar algoritmos como Argon2, bcrypt o PBKDF2, que están pensados para ser lentos y resistentes a ataques con máquinas potentes. Estos métodos añaden más protección porque obligan al atacante a invertir mucho más tiempo y recursos para intentar descifrar una sola contraseña.

---

8. Tenemos la siguiente API REST, muy simple. 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 correctamente los datos sensibles de esta API, usaría un método de cifrado moderno que no solo oculte la información, sino que también permita detectar si alguien ha intentado cambiar el mensaje.

Hoy en día lo más práctico es utilizar algoritmos que ya vienen preparados para ofrecer estas dos protecciones a la vez, como **AES-GCM** o **ChaCha20-Poly1305**. Ambos cifran los datos para que nadie pueda leerlos y generan un pequeño código de verificación que permite al receptor saber si el mensaje ha sido manipulado. Esto hace que el sistema sea más seguro y también más sencillo de implementar.

---

9. 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.

```
[Running] python -u
"c:\Proyectos\ciberseguridad-ejercicios\criptografia\practica-ejer
cicio-09.py"

SHA-256 COMPLETO:

db7df2f62c6f7d5fd13ebbc6c4fd79e95115c7c38d2a9fce8370eaac6cfd0631



KCV(SHA-256) (primeros 3 bytes): db7df2
```

```
----------------------------------------

AES-CBC(0x00...) BLOQUE CIFRADO COMPLETO:

5244dbd02d57d56ae08e064c56c7ca74



KCV(AES) (primeros 3 bytes): 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 Pedropriv.txt y Pedro-publ.txt, con las claves privada y pública. Las claves de los ficheros de RRHH son RRHH-priv.txt y RRHH-publ.txt que también se tendrán disponibles. Se requiere verificar la misma, y evidenciar dicha prueba.

```
miche@DESKTOP-TUF007 MINGW64 /c/Proyectos/ciberseguridad-ejercicios/criptografia (main)
$ gpg --import Pedro-publi.txt
gpg: directory '/c/Users/miche/.gnupg' created
gpg: /c/Users/miche/.gnupg/trustdb.gpg: trustdb created
gpg: key D730BE196E466101: public key "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa
.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1

miche@DESKTOP-TUF007 MINGW64 /c/Proyectos/ciberseguridad-ejercicios/criptografia (main)
$ gpg --verify MensajeRespoDeRaulARRHH.sig
gpg: Signature made Sun, Jun 26, 2022  1:47:01 PM
gpg:                using EDDSA key 1BDE635E4EAE6E68DFAD2F7CD730BE196E466101
gpg:                issuer "pedro.pedrito.pedro@empresa.com"
gpg: Good signature from "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>" [unkno
wn]
gpg: WARNING: The key's User ID 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
```

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.

```
miche@DESKTOP-TUF007 MINGW64 /c/Proyectos/ciberseguridad-ejercicios/criptografia (main)
$ gpg --import RRHH-priv.txt
gpg: key 3869803C684D287B: public key "RRHH <RRHH@RRHH>" imported
gpg: key 3869803C684D287B: secret key imported
gpg: Total number processed: 1
gpg:               imported: 1
gpg:       secret keys read: 1
gpg:   secret keys imported: 1
miche@DESKTOP-TUF007 MINGW64 /c/Proyectos/ciberseguridad-ejercicios/criptografia (main)
miche@DESKTOP-TUF007 MINGW64 /c/Proyectos/ciberseguridad-ejercicios/criptografia (main)
$ gpg --list-secret-keys
[keyboxd]
---------
sec   ed25519 2022-06-26 [SC] [expires: 2029-12-07]
      F2B1D0E8958DF2D3BDB6A1053869803C684D287B
uid           [ unknown] RRHH <RRHH@RRHH>
ssb   cv25519 2022-06-26 [E] [expires: 2029-12-07]


miche@DESKTOP-TUF007 MINGW64 /c/Proyectos/ciberseguridad-ejercicios/criptografia (main)
$ gpg --clearsign -u RRHH@RRHH MensajeRRHH.txt

miche@DESKTOP-TUF007 MINGW64 /c/Proyectos/ciberseguridad-ejercicios/criptografia (main)
$ gpg --verify MensajeRRHH.txt.asc
gpg: Signature made Thu, Dec 11, 2025 11:25:27 AM
gpg:                using EDDSA key F2B1D0E8958DF2D3BDB6A1053869803C684D287B
gpg: Good signature from "RRHH <RRHH@RRHH>" [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: F2B1 D0E8 958D F2D3 BDB6  A105 3869 803C 684D 287B
gpg: WARNING: not a detached signature; file 'MensajeRRHH.txt' was NOT verified!
```

```
miche@DESKTOP-TUF007 MINGW64 /c/Proyectos/ciberseguridad-ejercicios/criptografia (main)
$ cat MensajeRRHH.txt.asc
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA512

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

iHUEARYKAB0WIQTysdDolY3y0722oQU4aYA8aE0oewUCaTqcFwAKCRA4aYA8aE0o
ew6fAQDanbyIt2axCEkoMuvf/EK2xT8QotXjUPr5kCPIXMlMowEAzongMhqPtbtk
fXwnzv/sQIIYARH7My6Vz1otICsOeAg=
=oobj
-----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.

```
miche@DESKTOP-TUF007 MINGW64 /c/Proyectos/ciberseguridad-ejercicios/criptografia (main)
$ gpg --output MensajeFinal_Pedro_RRHH.gpg --encrypt -r "Pedro" -r "RRHH" 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@empre
sa.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
File 'MensajeFinal_Pedro_RRHH.gpg' exists. Overwrite? (y/N) y

miche@DESKTOP-TUF007 MINGW64 /c/Proyectos/ciberseguridad-ejercicios/criptografia (main)
$ ls
KeyStorePracticas            practica_ejercicio-12.py  private-rsa.pem
MensajeFinal.txt             practica-ejercicio-01.py  public-rsa.pem
MensajeFinal_Pedro_RRHH.gpg  practica-ejercicio-02.py  reto-01.py
MensajeRespoDeRaulARRHH.sig  practica-ejercicio-03.py  reto-02.py
MensajeRespoDeRaulARRHH.txt  practica-ejercicio-04.py  reto-03.py
MensajeRRHH.txt              practica-ejercicio-06.py  reto-04.py
MensajeRRHH.txt.asc          practica-ejercicio-09.py  reto-05.py
Pedro-priv.txt               practica-ejercicio-15.py  RRHH-priv.txt
Pedro-publi.txt              practica-ejericio-05.py   RRHH-publi.txt
```

11. Nuestra compañía tiene un contrato con una empresa que nos da un servicio de almacenamiento de información de videollamadas. Para lo cual, la misma nos envía la clave simétrica de cada videollamada cifrada usando un RSA-OAEP.

El hash que usa el algoritmo interno es un SHA-256.

El texto cifrado es el siguiente:

b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c
96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d04e3d3d4ad629
793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae2c4f5ba670a4126f2f14
a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8624071be78ccef573d
896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f67d3f98b4cd907f27639f4b1
df3c50e05d5bf63768088226e2a9177485c54f72407fdf358fe64479677d8296ad38c6f
177ea7cb74927651cf24b01dee27895d4f05fb5c161957845cd1b5848ed64ed3b0372
2b21a526a6e447cb8ee

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

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

Los textos cifrados son diferentes porque **RSA-OAEP no genera siempre el mismo resultado aunque se cifre exactamente el mismo mensaje**. Este algoritmo utiliza **valores aleatorios internos** cada vez que realiza un cifrado, de modo que dos ejecuciones nunca producen el mismo texto cifrado.

Esta variación es intencionada: sirve para evitar ataques y hacer que el cifrado sea más seguro. Por eso, aunque usemos la misma clave pública, el mismo mensaje, y el mismo algoritmo, el resultado cifrado cambia en cada operación.

```
[Running] python -u
"c:\Proyectos\ciberseguridad-ejercicios\criptografia\practica-ejer
cicio-11.py"

Clave simétrica recuperada (hex):

e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

----------------------------------------

Nuevo texto cifrado (hex):

48ebe6020ad2ed96d4cc372bf547e7595727dc22c398d7b178fd7f0b960397e860
f5d3e3215a7d5bcfd23bfe7efe12bebe0f4315ac08d7c927a87cb2d80a9c51d81f
69a769f16e2630fb6c96e3c76135d395358415e70d3ee2a2359de9f0da8af3d47b
d612869af7d911689e94bd338c3ed6e6d945bc3886bc101ace18b5ef960f5b3b9d
```

```
df042137fe340b5d27abc81688c15088a2352f815d54c7558c120498b03dd0f18b
3653c946b208d3057c30a1f644962ad164cd21e8066740a39e15b65a01cff21cf3
5b3255df779bcdd7b35d50fed26c66d653642478ec0972e9401c3f25620143be2c
338c7decde697dfc69894e6714a9b0459c8c050d696b578a32
```

---

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?

En AES-GCM el error está en reutilizar el mismo nonce junto con la misma clave. El algoritmo exige que, para una clave dada, cada mensaje use un nonce distinto. Si se mantiene siempre el mismo par clave-nonce en todas las comunicaciones, se pierde la seguridad del esquema: se pueden relacionar mensajes entre sí y comprometer tanto la confidencialidad como la integridad. Por eso, aunque la clave sea correcta, el nonce debe ser único en cada cifrado.

Cifra el siguiente texto:

He descubierto el error y no volveré a hacerlo mal Usando para ello, la clave, y el nonce indicados.

El texto cifrado presentalo en hexadecimal y en base64.

```
[Running] python -u
"c:\Proyectos\ciberseguridad-ejercicios\criptografia\practica_ejer
cicio-12.py"

Texto cifrado (hex):
5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2
abdd9d84bba6eb8307095f5078fbfc16256d

Texto cifrado (base64):
Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq92dhLum64MHCV9QePv8Fi
Vt
```

---

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?

```
[Running] python -u
"c:\Proyectos\ciberseguridad-ejercicios\criptografia\practica-ejer
cicio-13.py"

Firma RSA PKCS#1 v1.5 (hex):

a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a2
3885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8
f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9e
accd863bc924e70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b3
05c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf05d023
545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752
b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519
391b61891bf5e06699aa0a0dbae21f0aaaa6f9b9d59f41928d
```

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

```
Firma Ed25519 (hex):

bf32592dc235a26e31e231063a1984bb75ffd9dc5550cf30105911ca4560dab52a
bb40e4f7e2d3af828abac1467d95d668a80395e0a71c51798bd54469b7360d
```

---

14. Necesitamos generar una nueva clave AES, usando para ello una HKDF (HMACbased Extractand-Expand key derivation function) con un hash SHA-512. La clave maestra requerida se encuentra en el keystore con la etiqueta “cifrado-sim-aes-256”. La clave obtenida dependerá de un identificador de dispositivo, en este caso tendrá el valor en hexadecimal: e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3

¿Qué clave se ha obtenido?

```
Clave AES-256 derivada (HEX):
E716754C67614C53BD9BAB176022C952A08E56F07744D6C9EDB8C934F52E448A
```

---

15. Nos envían un bloque TR31:

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDB
E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E0
3CD857FD37018E111B

Donde la clave de transporte para desenvolver (unwrap) el bloque es:
A1A10101010101010101010101010102

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

Con **AES**, usando el formato TR-31 (versión D) para proteger el bloque de clave.

¿Para qué algoritmo se ha definido la clave?

Para el algoritmo **AES** (lo indica el campo A en la cabecera).

¿Para qué modo de uso se ha generado?

Para **Both (B)**, es decir, para ambos usos (cifrado/descifrado)

¿Es exportable?

Sí, **es exportable**.Se sabe porque en la cabecera del bloque TR-31 aparece la letra S en el campo de exportabilidad, que indica *Sensitive*: la clave puede exportarse, pero solo de forma controlada (por ejemplo, envuelta bajo otra clave).

¿Para qué se puede usar la clave?

La clave se puede usar para **cifrado y descifrado de datos**, porque el algoritmo es AES y el modo de uso indicado en la cabecera es Both (B).

¿Qué valor tiene la clave?

```
C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1
```