PRÁCTICA: CRIPTOGRAFÍA

Objetivo:

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

Detalles:

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

Cualquier password que sea necesaria tendrá un valor 123456.

Evaluación

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

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

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

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

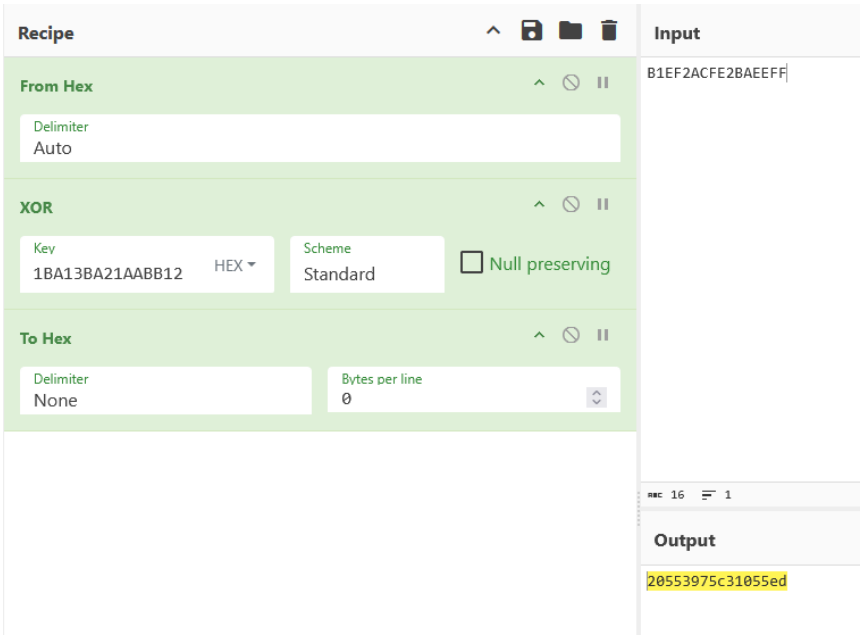
El objetivo principal de este módulo es adquirir conocimientos de criptografía y por ello se considera fundamental usar cualquier herramienta que pueda ayudar a su resolución, demostrando que no sólo se obtiene el dato sino que se tiene un conocimiento profundo del mismo. Si durante la misma no se indica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

Ejercicios:

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

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

   - ➔ Resultado: 20553975c31055ed
   - ➔ Ejecución:

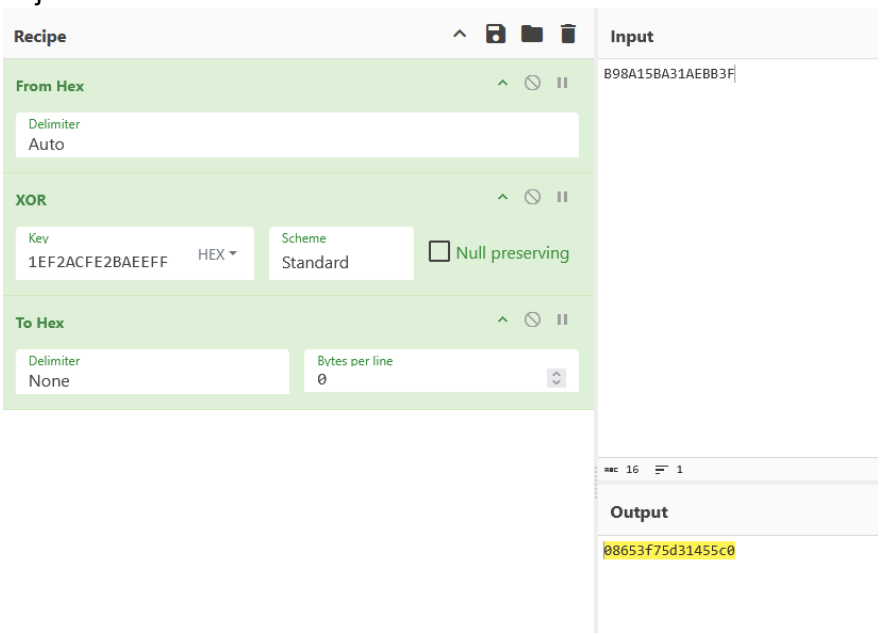



```python
# Ejercicio 1 de Ejercicios de Criptografia Finales

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


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

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

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

```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

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

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

- ➔ Resultado: 08653f75d31455c0
- ➔ Ejecución:



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

```
TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4US
t3aB/i50nvvJbBiG+le1ZhpR84oI=
```

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

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

- ➔ Código usado con importación de la librería usando mezcla de Python+Java por problemas de compatibilidad con C++ en KeyStore

C: > Users > d_gar > Desktop > Python VS Code > ejercicio2_final.py > descifrar_mensaje

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

def extraer_clave_con_test():
    """Usa el test que sabemos que funciona"""
    print("Ejecutando extractor probado...")

    # Código Java que SABEMOS funciona
    java_code = '''
import java.io.FileInputStream;
import java.security.KeyStore;
import javax.crypto.SecretKey;

public class TestExtract {
    public static void main(String[] args) throws Exception {
        // Ruta ABSOLUTA - MODIFICA ESTA LÍNEA
        String keystorePath = "C:\\\\Users\\\\d_gar\\\\Desktop\\\\KeyStore\\\\KeyStorePracticas";
        String keystorePass = "123456";
        String keyAlias = "cifrado-sim-aes-256";
        String keyPass = "123456";

        KeyStore ks = KeyStore.getInstance("JCEKS");
        ks.load(new FileInputStream(keystorePath), keystorePass.toCharArray());

        SecretKey key = (SecretKey) ks.getKey(keyAlias, keyPass.toCharArray());

        byte[] keyBytes = key.getEncoded();
        StringBuilder hex = new StringBuilder();
        for (byte b : keyBytes) {
            hex.append(String.format("%02x", b));
        }

        System.out.print(hex.toString());
    }
}
'''

    # Crear archivo
    with open('TestExtract.java', 'w', encoding='utf-8') as f:
        f.write(java_code)

    # Compilar
    compile_result = subprocess.run(['javac', 'TestExtract.java'],
                                    capture_output=True, text=True, shell=True)

```

C: > Users > d_gar > Desktop > Python VS Code > ejercicio2_final.py > descifrar_mensaje

```
7   def extraer_clave_con_test():
47                                  capture_output=True, text=True, shell=True)
48
49      if compile_result.returncode != 0:
50          print(f"Error compilando: {compile_result.stderr[:200]}")
51          # Limpiar y salir
52          if os.path.exists('TestExtract.java'):
53              os.remove('TestExtract.java')
54          return None
55
56      # Ejecutar
57      run_result = subprocess.run(['java', 'TestExtract'],
58                                  capture_output=True, text=True, shell=True)
59
60      # Limpiar
61      for f in ['TestExtract.java', 'TestExtract.class']:
62          if os.path.exists(f):
63              os.remove(f)
64
65      if run_result.returncode == 0 and run_result.stdout.strip():
66          clave = run_result.stdout.strip()
67          print(f"Clave extraída: {clave[:16]}...{clave[-16:]}")
68          return clave
69      else:
70          print(f"[!]Error ejecutando: {run_result.stderr[:200]}")
71          return None
72
73  def descifrar_mensaje(clave_hex):
74      """Descifra el mensaje"""
75      try:
76          clave_bytes = bytes.fromhex(clave_hex)
77
78          cifrado_base64 = "TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4US t3aB/i50nvvJbBiG+le1ZhpR84oI="
79          cifrado_base64 = cifrado_base64.replace(" ", "")
80
81          iv = b'\x00' * 16
82          texto_cifrado_bytes = base64.b64decode(cifrado_base64)
83
84          cipher = AES.new(clave_bytes, AES.MODE_CBC, iv)
85          mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size)
86
87          texto_claro = mensaje_des_bytes.decode('utf-8')
88
89          print(f"\n TEXTO DESCIFRADO:")
90          print(f"   {texto_claro}")
91
92          print(f"\n ANÁLISIS:")
93          print(f"   Longitud: {len(texto_claro)} caracteres")
```

C: > Users > d_gar > Desktop > Python VS Code > ejercicio2_final.py > descifrar_mensaje

```python
def descifrar_mensaje(clave_hex):
        print(f"\n ANÁLISIS:")
        print(f"   Longitud: {len(texto_claro)} caracteres")
        print(f"   Bytes UTF-8: {len(texto_claro.encode('utf-8'))}")

        padding = 16 - (len(texto_claro.encode('utf-8')) % 16)
        if padding == 16:
            padding = 0
        print(f"Padding PKCS7: {padding} byte(s)")

        return texto_claro

    except Exception as e:
        print(f"[!]Error descifrando: {e}")
        return None

# --- PROGRAMA PRINCIPAL ---
if __name__ == "__main__":
    print("Sistema de descifrado importando de libreria KeyStore")
    print("=" * 50)

    #Extraer clave automáticamente
    print("\n1. EXTRAYENDO CLAVE DEL KEYSTORE...")
    clave_hex = extraer_clave_con_test()
    # DESCIFRAR
    print(f"\n2. DESCIFRANDO CON CLAVE: {clave_hex[:16]}...")
    resultado = descifrar_mensaje(clave_hex)

    print("\n" + "=" * 50)
    if resultado:
        print("EJERCICIO 2 RESUELTO EXITOSAMENTE")
    else:
        print("[!]HUBO ERRORES EN LA RESOLUCIÓN")
    print("=" * 50)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER 364

```
Clave extra◆da: a2cff885901a5449...a0148fb3a426db72

2. DESCIFRANDO CON CLAVE: a2cff885901a5449...

TEXTO DESCIFRADO:
Esto es un cifrado en bloque t◆pico. Recuerda, vas por el buen camino. ◆nimo.

AN◆LISIS:
Longitud: 77 caracteres
Bytes UTF-8: 79
Padding PKCS7: 1 byte(s)
```

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

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

¿Cuánto padding se ha añadido en el cifrado?

→ Longitud texto claro: 79 bytes

➔ Padding PKCS7 añadido: 1 bytes

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

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

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

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

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

```python
# Ejercicio 3 ChaCha20

from Crypto.Cipher import ChaCha20
import base64

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

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

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

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

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

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

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Nonce decodificado: f5871c9ff7f99c926102dd92
Longitud del nonce: 12 bytes

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

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

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

```python
# Ejercicio 3 Chacha20 Poly305

from Crypto.Cipher import ChaCha20_Poly1305
import base64

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

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

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

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

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


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


```

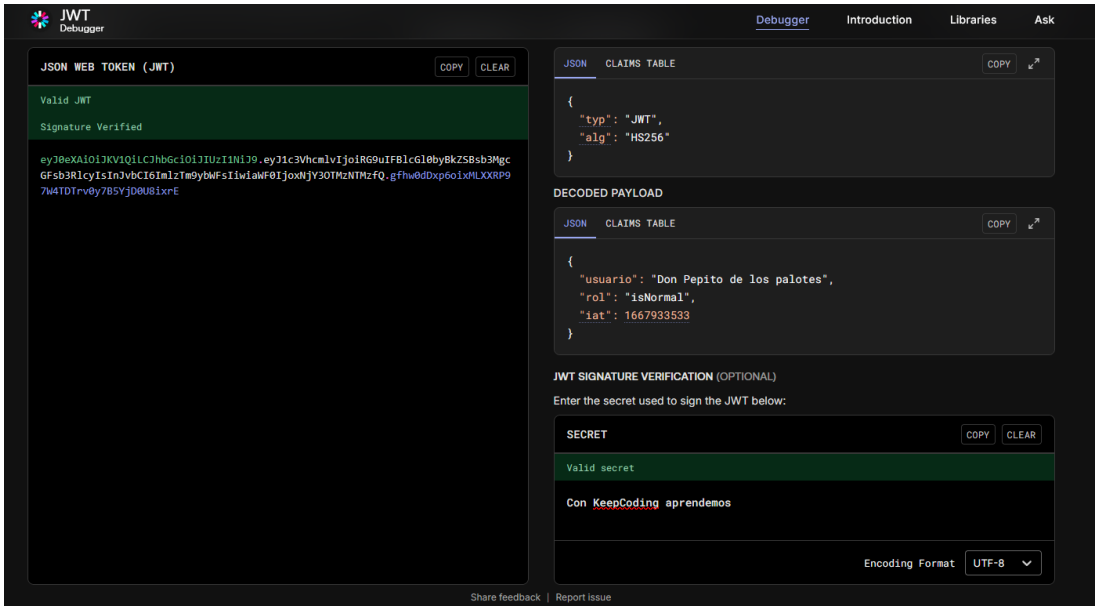
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

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

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


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

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0by BkZSB

sb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ .gfhw0

dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE



¿Qué algoritmo de firma hemos realizado?

➔ **Algoritmo: HS256** (HMAC con SHA-256)

¿Cuál es el body del jwt?

```
{
  "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? ¿Qué ocurre si intentamos validarlo con pyjwt?

➔ Se ha intentado cambiar el payload y reemplazar el “isNormal” por “isAdmin” para intentar escalar privilegios de administrador.

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

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?

➔ SHA3-256 (Keccak de 256 bits)

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

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f

6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

¿Qué hash hemos realizado?

➔ SHA-512 (SHA2 de 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?

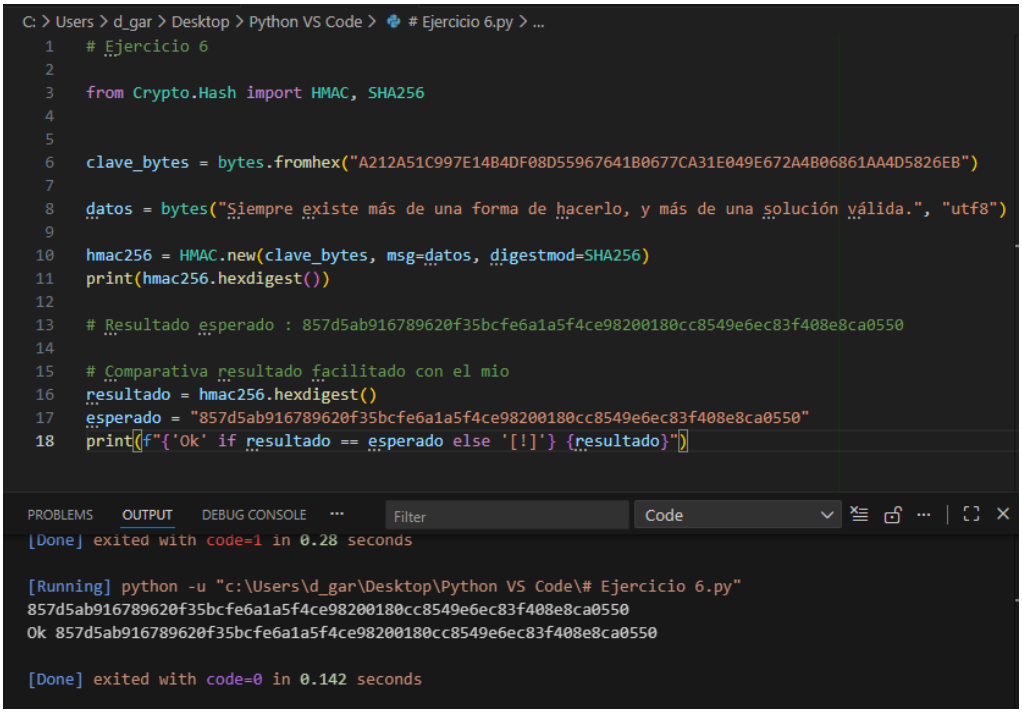
→ Efecto avalancha: Un cambio mínimo en la entrada (1 carácter: punto final) produce un cambio drástico en la salida

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:
8ad99ac91bac38320673143235d232057a3e216e2e56660f932795c54a48f195



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?

   → Es un sistema que se considera obsoleto actualmente e inseguro debido a vulnerabilidades descubiertas. Además tiene problemas de colisiones demostradas.

   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?

   → Para mejorar se podría añadir un “Salt”, que es único por usuario, dificultando el uso de diccionarios o rainbow tables para sacar claves por fuerza bruta. Se puede combinar con HMAC, usando una clave secreta del servidor.

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

   → Se podría añadir un “Pepper” que añadiría un punto más de resistencia.
   → Se podrían usar algoritmos especializados, como bcrypt, script y el más recomendado Argon2.

→ Implementar políticas de contraseñas fuertes.

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 el escenario propuesto, se entiende que puede ser transacciones con una tarjeta bancaria, por lo que además de un sistema robusto y con sistema de garantía de integridad, se necesita que sea rápido.
- ➔ Considero que lo mejor seria un sistema AES-CBC+MAC

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.

- ➔ KCV(SHA-256): DB7DF2
- ➔ KCV(AES): 5244DB

→Código aplicado:

```python
import hashlib
from Crypto.Cipher import AES

# Clave AES-256 proporcionada
clave_hex = "A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"
clave_bytes = bytes.fromhex(clave_hex)

print("\n CALCULO KCV - KEY CHECK VALUE")
print(f"Clave AES-256: {clave_hex}")
print(f"Longitud: {len(clave_bytes)} bytes = {len(clave_bytes)*8} bits\n")

# 1. KCV(SHA-256) - SHA-256 de la clave completa
print("1. KCV(SHA-256):")
sha256_hash = hashlib.sha256(clave_bytes).digest()  # bytes, no hexdigest
print(f"   SHA-256 completo: {sha256_hash.hex()}")
print(f"   Primeros 3 bytes: {sha256_hash[:3].hex()}")
print(f"   KCV(SHA-256): {sha256_hash[:3].hex().upper()}\n")

# 2. KCV(AES) - Cifrar 16 bytes de 0x00 con IV de 0x00
print("2. KCV(AES):")
# Bloque de 16 bytes de ceros (NO padding)
bloque_ceros = b'\x00' * 16  # Exactamente 16 bytes
iv_ceros = b'\x00' * 16

# Cifrar en CBC (sin padding - bloque exacto)
cipher = AES.new(clave_bytes, AES.MODE_CBC, iv_ceros)
texto_cifrado = cipher.encrypt(bloque_ceros)

print(f"   Texto plano: {bloque_ceros.hex()}")
print(f"   IV: {iv_ceros.hex()}")
print(f"   Texto cifrado completo: {texto_cifrado.hex()}")
print(f"   Primeros 3 bytes: {texto_cifrado[:3].hex()}")
print(f"   KCV(AES): {texto_cifrado[:3].hex().upper()}")

print("\n" + "="*60)
print("RESULTADOS FINALES:")
print("="*60)
print(f"KCV(SHA-256): {sha256_hash[:3].hex().upper()}")
print(f"KCV(AES):     {texto_cifrado[:3].hex().upper()}")
print("="*60)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER 259

```
2. KCV(AES):
============================================================
RESULTADOS FINALES:
============================================================
KCV(SHA-256): DB7DF2
KCV(AES):     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.

```
d_gar@MINI-Gaming-G1 MINGW64 ~/Desktop/PGP Bootcamp
$ gpg --verify MensajeRespoDeRaulARRHH.sig
gpg: Signature made Sun, Jun 26, 2022  1:47:01 PM RDT
gpg:                using EDDSA key 1BDE635E4EAE6E68DFAD2F7CD730BE196E466101
gpg:                issuer "pedro.pedrito.pedrc@empresa.com"
gpg: Good signature from "Pedro Pedrito Pedro <pedro.pedrito.pedrc@empresa.com>" [expired]
gpg: Note: This key has expired!
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.

➔ Como la firma estaba caducada he tenido que editarla para no tener problemas al firmar el documento de respuesta como RRHH.

```
d_gar@MINI-Gaming-G1 MINGW64 ~/Desktop/PGP Bootcamp
$ gpg --edit-key F2B1D0E8958DF2D3BDB6A1053869803C684D287B
gpg (GnuPG) 2.4.7-unknown; Copyright (C) 2024 g10 Code GmbH
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Secret key is available.

sec  ed25519/3869803C684D287B
     created: 2022-06-26  expired: 2024-06-25  usage: SC
     trust: unknown       validity: expired
ssb  cv25519/7C1A46EA20B0546F
     created: 2022-06-26  expired: 2024-06-25  usage: E
[ expired] (1). RRHH <RRHH@RRHH>

gpg> expire
Changing expiration time for the primary key.
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 2y
Key expires at Sun, Dec  5, 2027  6:54:43 PM RST
Is this correct? (y/N) y

sec  ed25519/3869803C684D287B
     created: 2022-06-26  expires: 2027-12-05  usage: SC
     trust: unknown       validity: unknown
ssb  cv25519/7C1A46EA20B0546F
     created: 2022-06-26  expired: 2024-06-25  usage: E
[ unknown] (1). RRHH <RRHH@RRHH>

gpg: WARNING: Your encryption subkey expires soon.
gpg: You may want to change its expiration date too.
gpg: WARNING: No valid encryption subkey left over.
gpg> save

d_gar@MINI-Gaming-G1 MINGW64 ~/Desktop/PGP Bootcamp
$ gpg --list-keys F2B1D0E8958DF2D3BDB6A1053869803C684D287B
gpg: checking the trustdb
gpg: no ultimately trusted keys found
pub   ed25519 2022-06-26 [SC] [expires: 2027-12-05]
      F2B1D0E8958DF2D3BDB6A1053869803C684D287B
uid           [ unknown] RRHH <RRHH@RRHH>
```

➔ Una vez la firma estaba correcta nuevamente he procedido con la firma del texto que tiene que enviar RRHH

```
d_gar@MINI-Gaming-G1 MINGW64 ~/Desktop/PGP Bootcamp
$ gpg --clearsign --output RRHH_firmado.asc --local-user F2B1D0E8958DF2D3BDB6A1053869803C684D287B  mensaje_rrhh.txt

d_gar@MINI-Gaming-G1 MINGW64 ~/Desktop/PGP Bootcamp
$ gpg --verify RRHH_firmado.asc
gpg: Signature made Fri, Dec  5, 2025  6:58:09 PM RST
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

d_gar@MINI-Gaming-G1 MINGW64 ~/Desktop/PGP Bootcamp
$ cat RRHH_firmado.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-----

iHUEARYKAB0WIQTysdDolY3y0722oQU4aYA8aE0oewUCaTMdMQAKCRA4aYA8aE0o
e906AQDb9kBWYehk57YqoBzqUI9iq2VYJUu5iq7EgLfP2/9cwgEA/lS3mxACkwOm
lJrVSOT7DJOjHBkQGiZPEvyOehPgBAQ=
=uZt3
-----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.

➔ Aqui al intentar crear y firmar el texto me daba problemas, porque aunque había actualizado la fecha de expiración, la subclase para firmar no se había actualizad, por lo que he tenido que meterme dentro para actualizar la fecha.

```
d_gar@MINI-Gaming-G1 MINGW64 ~/Desktop/PGP Bootcamp
$ gpg --edit-key F2B1D0E8958DF2D3BDB6A1053869803C684D287B
There is NO WARRANTY, to the extent permitted by law.

Secret key is available.

sec  ed25519/3869803C684D287B
     created: 2022-06-26  expires: 2027-12-05  usage: SC
     trust: ultimate      validity: ultimate
ssb  cv25519/7C1A46EA20B0546F
     created: 2022-06-26  expired: 2024-06-25  usage: E
[ultimate] (1). RRHH <RRHH@RRHH>

gpg> list

sec  ed25519/3869803C684D287B
     created: 2022-06-26  expires: 2027-12-05  usage: SC
     trust: ultimate      validity: ultimate
ssb  cv25519/7C1A46EA20B0546F
     created: 2022-06-26  expired: 2024-06-25  usage: E
[ultimate] (1). RRHH <RRHH@RRHH>

gpg> key 1 cv25519/7C1A46EA20B0546F

sec  ed25519/3869803C684D287B
     created: 2022-06-26  expires: 2027-12-05  usage: SC
     trust: ultimate      validity: ultimate
ssb* cv25519/7C1A46EA20B0546F
     created: 2022-06-26  expired: 2024-06-25  usage: E
[ultimate] (1). RRHH <RRHH@RRHH>

gpg> expire
Changing expiration time for a subkey.
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 2y
Key expires at Sun, Dec  5, 2027  7:20:30 PM RST
Is this correct? (y/N) y

sec  ed25519/3869803C684D287B
     created: 2022-06-26  expires: 2027-12-05  usage: SC
     trust: ultimate      validity: ultimate
ssb* cv25519/7C1A46EA20B0546F
     created: 2022-06-26  expires: 2027-12-05  usage: E
[ultimate] (1). RRHH <RRHH@RRHH>

gpg> save
```

→ Una vez esto estaba correcto en ambos, Pedro y RRHH, he procedido a crear el texto y probar que se había encriptado correctamente.

```
d_gar@MINI-Gaming-G1 MINGW64 ~/Desktop/PGP Bootcamp
$ gpg --encrypt --trust-model always \
    --recipient 1BDE635E4EAE6E68DFAD2F7CD730BE196E466101 \
    --recipient F2B1D0E8958DF2D3BDB6A1053869803C684D287B \
    --output FINAL.asc \
    mensaje_final.txt

d_gar@MINI-Gaming-G1 MINGW64 ~/Desktop/PGP Bootcamp
$ cat mensaje_final.txt
Estamos todos de acuerdo, el ascenso será el mes que viene, agosto, si no hay sorpresas.

d_gar@MINI-Gaming-G1 MINGW64 ~/Desktop/PGP Bootcamp
$ cat FINAL.asc
□^%□□)@5□P@Ċ□□Y□□□/□k□gj□4□M□
□bt□□Z□□Q0698t□□□p□a□lo/□~□□□zB§S□6
dO□□f□□%□□□^□□^|§F□ □Tc@□g□      \-□□K□□¡|s3;□□;□□%"□□0raÁq□Iy□□-$[□P□□3□Ṽ□4□□ʰ'□□□□□,□□□5□k□□□Rɳ          □□<□
□7□W□5□u4<i./□>□`□□''□□
□□□§□/"O□□ad□b□&□4□□□:□6Y□v□□□□\x%□vk<□□2□□a□□ □□=□c□□□6p□□□Cb□j□□~□□□□□ □\□ +U□□□□

□^□|嶗i
□□w
```

➔ Despues por checkear como siempre que está funcionando, he procedido a verificar.

```
d_gar@MINI-Gaming-G1 MINGW64 ~/Desktop/PGP Bootcamp
$ gpg --decrypt FINAL.asc
gpg: encrypted with cv25519 key, ID 7C1A46EA20B0546F, created 2022-06-26
      "RRHH <RRHH@RRHH>"
gpg: encrypted with cv25519 key, ID 25D6D0294035B650, created 2022-06-26
      "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>"
Estamos todos de acuerdo, el ascenso será el mes que viene, agosto, si no hay sorpresas.
```

11. Nuestra compañía tiene un contrato con una empresa que nos da un servicio de almacenamiento de información de videollamadas. Para lo cual, la misma nos envía la clave simétrica de cada videollamada cifrada usando un RSA-OAEP. El hash que usa el algoritmo interno es un SHA-256.

El texto cifrado es el siguiente:

```
b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c
96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d04e3d3d4ad629
793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae2c4f5ba670a4126f2f14
a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8624071be78ccef573d
896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f67d3f98b4cd907f27639f4b1
df3c50e05d5bf63768088226e2a9177485c54f72407fdf358fe64479677d8296ad38c6f
```

177ea7cb74927651cf24b01dee27895d4f05fb5c161957845cd1b5848ed64ed3b0372

2b21a526a6e447cb8ee

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

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

```
➔ Clave sim□trica recuperada:
   e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72
➔ Longitud: 32 bytes
```

```
============================================================
2. VOLVIENDO A CIFRAR...
============================================================
[OK] Re-cifrado funciona
Texto re-cifrado:
5ddbf23ee87f961d3208ab47f6b7516adc7e1b7d207eb8da763e91dd24e6924b0aec8b7bdfcf0e5fe06c215b23e5ca4889798194b59f8e5805452ccfab5c954884bccea2d2ef708e699c272a191ea2e589168d6d6436661fd94e217d735f6c20d45d717655f
8fa9fee9eff5ffb6c8a09e293b85b8e530d92e8c599cb14c6a0ae992464a461f1ecf9b1af2fa24821960e63415db34abbab6f066ab3a8ec104f121b2477def4c7b30ab892ce08c118bd8e809833835964eaee3bb5ed29716314ab033c018e9595839d999911
1519c87028caee5a4fea0bead83c419a1dfb6508f5bed1ab56d19b3250e22d94a46e4143a9a8a04c92029ebb6fb6b74a5ae88751c7...
Longitud: 256 bytes

3. COMPARACI□N:
Original:
b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d04e3d3d4ad629793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae2c4f5ba670a
4126f2f14a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8624071be78ccef573d896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f67d3f98b4cd907f27639f4b1df3c50e05d5bf63768088226e2a9177485c54f72407fdf358fe64
479677d8296ad38c6f177ea7cb74927651cf24b01dee27895d4f05fb5c161957845cd1b5848ed64ed3b03722b21a526a6e447cb8ee...
Nuevo:
5ddbf23ee87f961d3208ab47f6b7516adc7e1b7d207eb8da763e91dd24e6924b0aec8b7bdfcf0e5fe06c215b23e5ca4889798194b59f8e5805452ccfab5c954884bccea2d2ef708e699c272a191ea2e589168d6d6436661fd94e217d735f6c20d45d717655f
8fa9fee9eff5ffb6c8a09e293b85b8e530d92e8c599cb14c6a0ae992464a461f1ecf9b1af2fa24821960e63415db34abbab6f066ab3a8ec104f121b2477def4c7b30ab892ce08c118bd8e809833835964eaee3bb5ed29716314ab033c018e9595839d999911
1519c87028caee5a4fea0bead83c419a1dfb6508f5bed1ab56d19b3250e22d94a46e4143a9a8a04c92029ebb6fb6b74a5ae88751c7...
□Son iguales? NO
```

➔ Los textos cifrados son diferentes porque el sistema RSA-OAEP incluye componentes aleatorios cada vez que hace un cifrado por lo que cada vez que se genera uno nuevo siempre será diferente al anterior generado.

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

Key:E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A42

6DB74

Nonce:9Yccn/f5nJJhAt2S

¿Qué estamos haciendo mal?

➔ Inicialmente, según el texto, expone que la empresa esta “usando estos mismos datos en cada comunicación”, el problema viene en que el nonce debe ser

aleatorio y no usarse en mas de una ocasión, ya que si se usa más veces podría ser vulnerable si se descubre.

Cifra el siguiente texto:

He descubierto el error y no volveré a hacerlo mal

Usando para ello, la clave, y el nonce indicados. El texto cifrado presentalo en hexadecimal y en base64.

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

```python
# Ejercicio 12 - AES/GCM

from Crypto.Cipher import AES
import base64

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

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

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

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

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

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

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

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

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

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

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

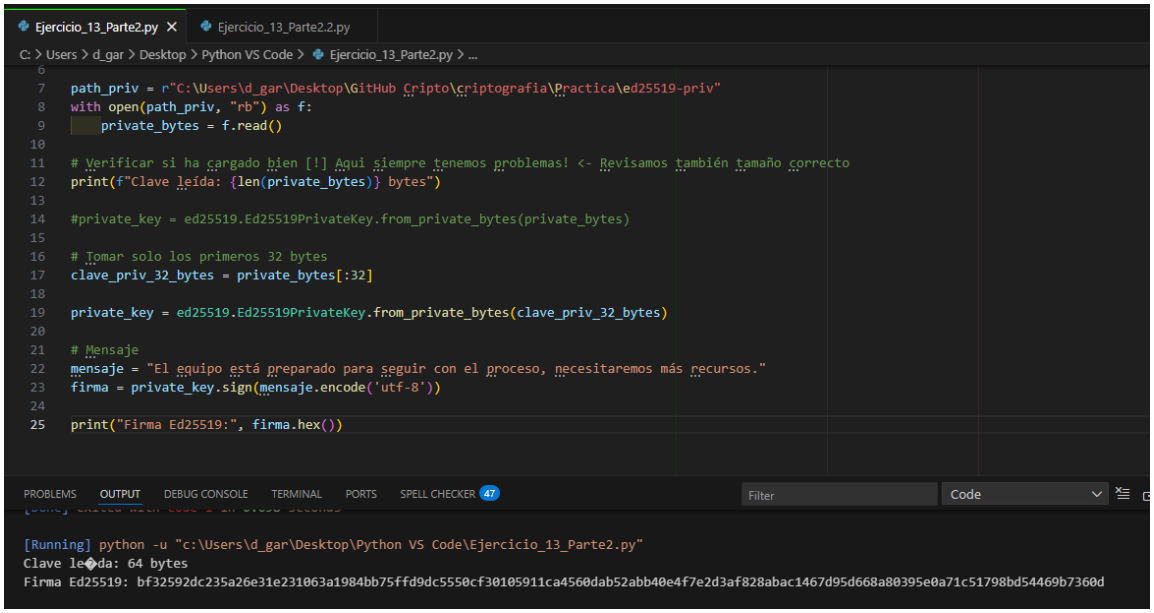
El equipo está preparado para seguir con el proceso, necesitaremos más recursos.

¿Cuál es el valor de la firma en hexadecimal?

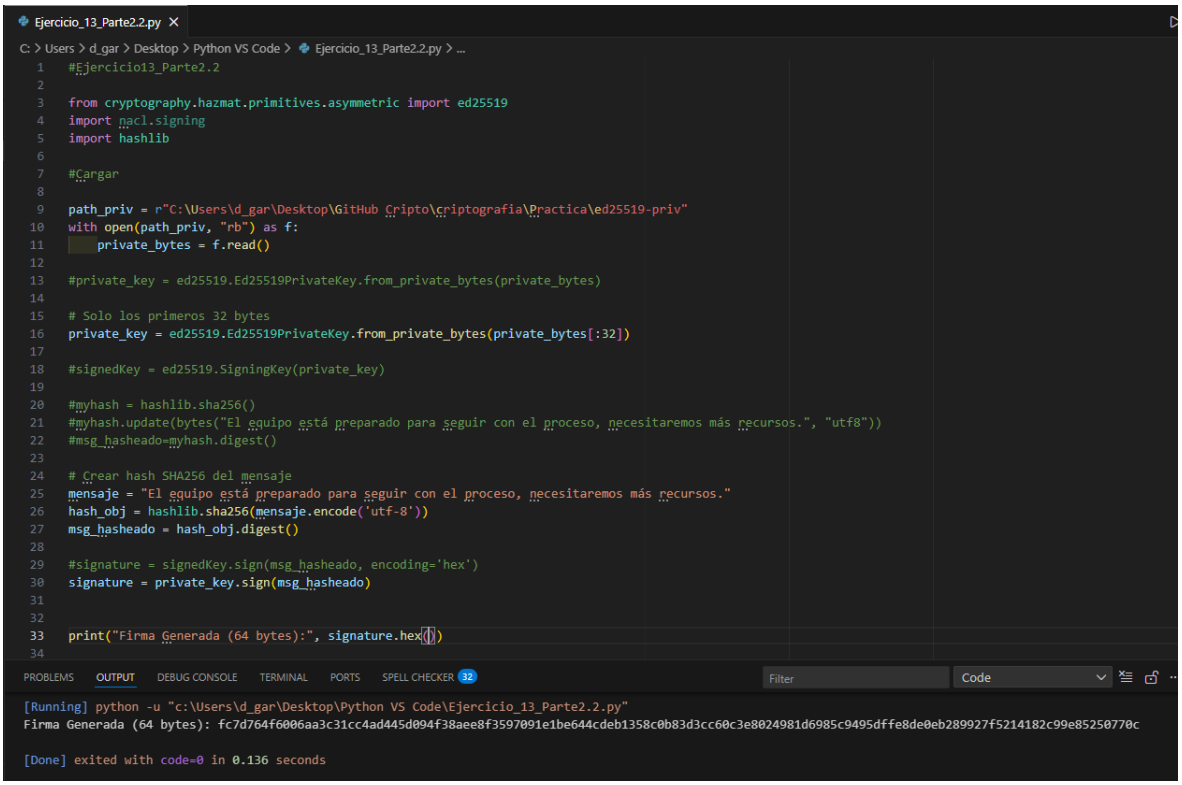
```
➔ Firma:
  a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dec
  e92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3
  a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb8736
  9d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe03185
  3277569f8383846d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9
  207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09
  663acb6b9519391b61891bf5e06699aa0a0dbae21f0aaaa6f9b9d59f41928d
➔
```

➔ Código usado:

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



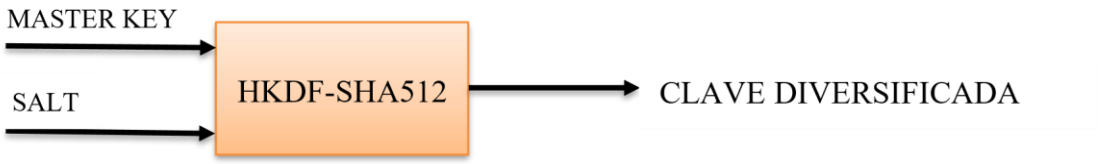
Alternativa 2?¿



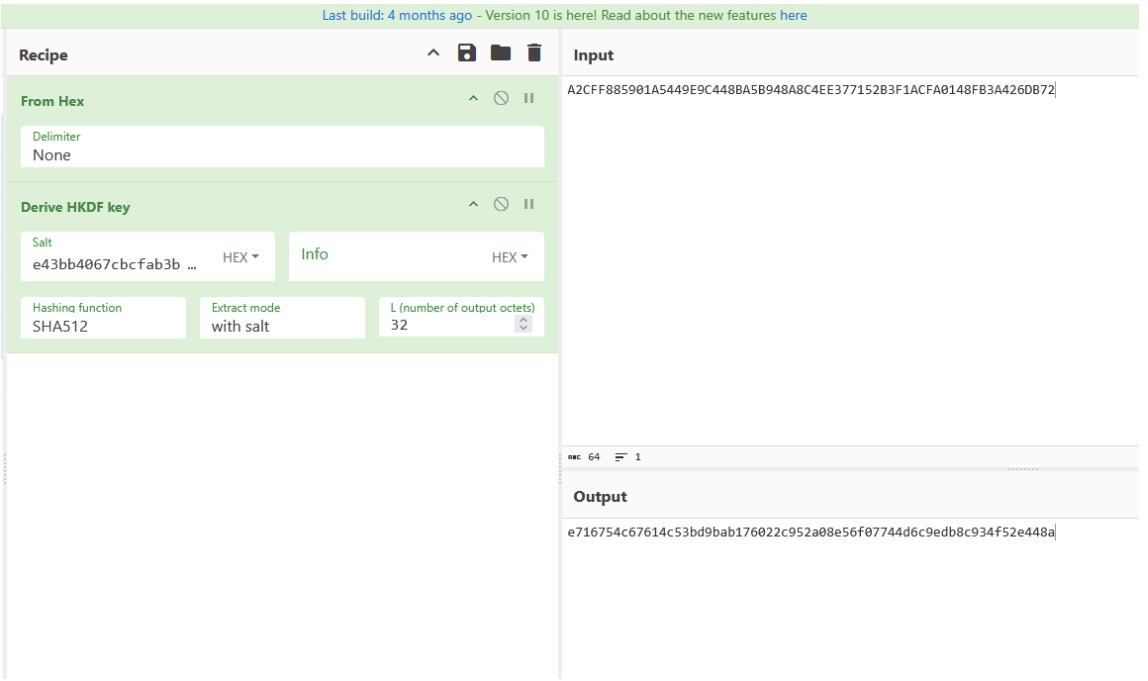
14. Necesitamos generar una nueva clave AES, usando para ello una HKDF (HMACbased Extractand-Expand key derivation function) con un hash SHA-512. La clave maestra requerida se encuentra en el keystore con la etiqueta "cifrado-sim-aes-

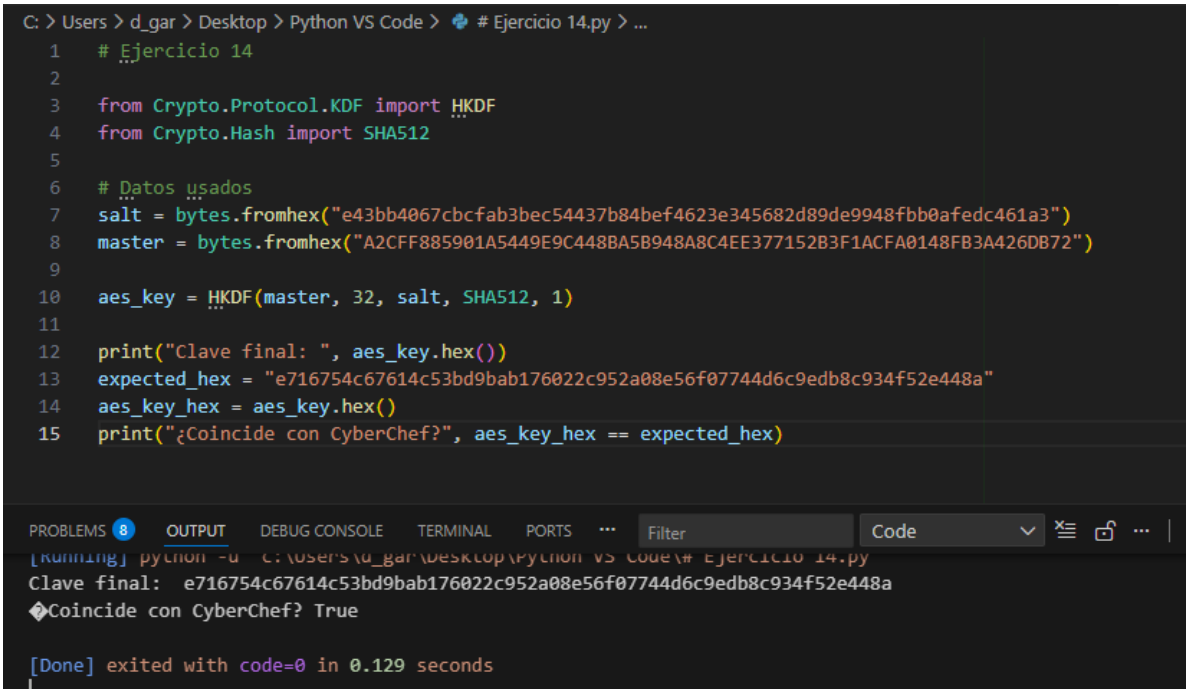
256". La clave obtenida dependerá de un identificador de dispositivo, en este caso tendrá el valor en hexadecimal:

```
e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3
```



¿Qué clave se ha obtenido?





15. Nos envían un bloque TR31:

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDB
E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E0
3CD857FD37018E111B

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

A1A10101010101010101010101010102

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

➔ AES Key Derivation Binding Method

¿Para qué algoritmo se ha definido la clave?

➔ AES

¿Para qué modo de uso se ha generado?

➔ Cifrado y descifrado

¿Es exportable?

➔ Sí, es exportable bajo clave no confiable

¿Para qué se puede usar la clave?

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

¿Qué valor tiene la clave?

➔ Clave desempaquetada = c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1