

# PRÁCTICA DE CRIPTOGRAFÍA Y SEGURIDAD DE LA INFORMACIÓN

TITULO:

Ejercicios de Cifrado y Descifrado

CURSO:

Ciberseguridad / Criptografía y Seguridad de la Información

PROFESOR:

Felipe Rodríguez Fonte

ALUMNO:

Jordan Andres Diaz Sanchez

## Ejercicios prácticos:

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?

- **RESPUESTA:** El valor es **20553975C31055ED**

## Calculadora XOR

Gracias por utilizar la calculadora. [Ver página de ayuda](#).

I. Entrada : [hexadecimal \(base 16\)](#) ▼

b1ef2acfe2baeeff

II. Entrada : [hexadecimal \(base 16\)](#) ▼

91ba13ba21aabb12

Calcular XOR

III. Salida : [hexadecimal \(base 16\)](#) ▼

20553975c31055ed

[Hogar](#)

[Ayuda](#)

[Privacidad](#)

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?

- **RESPUESTA:** La clave es **08653F75D31455C0**

## Calculadora XOR

Gracias por utilizar la calculadora. [Ver página de ayuda](#) .

I. Entrada : [hexadecimal \(base 16\)](#) ▼

b1ef2acfe2baeeff

II. Entrada : [hexadecimal \(base 16\)](#) ▼

b98a15ba31aebb3f

Calcular XOR

III. Salida : [hexadecimal \(base 16\)](#) ▼

8653f75d31455c0

[Hogar](#)

[Ayuda](#)

[Privacidad](#)

---

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:

**TQ9SOMKc6aFS9SIxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWEle  
zdrLAD5LO4USt3aB/i50nvvJbBiG+le1ZhpR84ol=**

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

- **RESPUESTA:** Obtenemos el siguiente texto en claro: **Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.**

```
AES-CBC practica.py U X
criptografia en bloque > AES-CBC practica.py > ...
1 from base64 import b64decode
2 from Crypto.Cipher import AES
3 from Crypto.Util.Padding import unpad
4 import json
5
6
7 # Texto cifrado en base64
8 texto_cifrado_b64 = "TQ9SOMKc6aFS9S1xhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIwXg3nu1RRz4QWleZdrLAD5L04USt3aB/150nvVJbBiG+le1ZhpR84o"
9 clave = bytes.fromhex("A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72") # keystore
10 iv = bytes([0] * 16) # Ceros binarios
11
12 # Decodificar texto cifrado desde Base64
13 texto_cifrado = b64decode(texto_cifrado_b64)
14 cipher = AES.new(clave, AES.MODE_CBC, iv)
15
16 # Descifrar los datos
17 texto_descifrado_bytes = cipher.decrypt(texto_cifrado)
18 texto_descifrado = unpad(texto_descifrado_bytes, AES.block_size, style='pkcs7')
19
20 print(f"Texto en claro: {texto_descifrado.decode('UTF-8')}")
21
22 # Descifrar los datos usando AES-256-CBC con X.923 padding
23 cipher = AES.new(clave, AES.MODE_CBC, iv)
24 texto_descifrado_x923 = unpad(cipher.decrypt(texto_cifrado), AES.block_size, style='x923')
25
26 print("Texto descifrado en claro (X.923):", texto_descifrado_x923.decode('utf-8'))
27
```

b000430300010337

[Done] exited with code=0 in 0.367 seconds

[Running] python -u "c:\Users\jorda\OneDrive\Documentos\GitHub\cripto\criptografia en bloque\AES-CBC practica.py"

Texto en claro: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. ánimo.

Texto descifrado en claro (X.923): Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. ánimo.

[Done] exited with code=0 in 0.359 seconds

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

- **RESPUESTA:** Si cambiamos el padding a x923 en este caso nos está dando el mismo texto en claro debido a que se parecen la cantidad de bytes que tienen.

```
[Running] python -u "c:\Users\jorda\OneDrive\Documentos\GitHub\cripto\criptografia en bloque\AES-CBC practica Eje2.py"
```

Texto en claro: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. ánimo.

Texto descifrado en claro (X.923): Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. ánimo.

[Done] exited with code=0 in 0.36 seconds

¿Cuánto padding se ha añadido en el cifrado?

- **RESPUESTA:** Se añadieron 02/02 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-chacha-256**”. El nonce “**9Yccn/f5nJJhAt2S**”. El algoritmo que se debe usar es un Chacha20.

- **RESPUESTA:** El resultado del mensaje cifrado es:
  - nonce: 9Yccn/f5nJJhAt2S
  - Encrypt Text:  
TslZlCqLdX4jNmBcfbq49NQLW00iDmaql490DT5ZsM1w4yFyQpkcwUC7Hh  
o=
  - Datos asociados:
  - Tag: cQzUcj2m1e838jvuZiheVw==

```
Chacha20-practica-cif_Eje3.py U X
criptografia en flujo > Chacha20-practica-cif_Eje3.py > ...
1 from Crypto.Cipher import ChaCha20_Poly1305
2 from base64 import b64decode, b64encode
3 from Crypto.Random import get_random_bytes
4 import json
5
6 try:
7
8     textoPlano_bytes = bytes('KeepCoding te enseña a codificar y a cifrar', 'UTF-8')
9     clave = bytes.fromhex('AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120') #Clave hmac256 del keyStore
10
11     #Importante NUNCA debe fijarse el nonce
12     nonce_mensaje_bytes=b64decode("9Yccn/f5nJJhAt2S")
13
14     #Con la clave y con el nonce se cifra. El nonce debe ser único por mensaje
15     #Hoy decido que no tenga datos asociados
16     datos_asociados = bytes('', 'utf-8')
17
18     cipher = ChaCha20_Poly1305.new(key=clave, nonce=nonce_mensaje_bytes)
19     #Por ser cifrado autenticado hacemos un update (lo mismo ocurría en AES-GCM)
20     cipher.update(datos_asociados)
21     texto_cifrado, tag = cipher.encrypt_and_digest(textoPlano_bytes)
22     print("nonce:", b64encode(nonce_mensaje_bytes).decode())
23     print("Encrypt Text:", b64encode(texto_cifrado).decode())
24     print("Datos asociados:", b64encode(datos_asociados).decode())
25     print("Tag:", b64encode(tag).decode())
26
27
PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS Code
[Running] python -u "c:\Users\jorda\OneDrive\Documentos\GitHub\cripto\criptografia en flujo\Chacha20-practica-cif_Eje3.py"
nonce: 9Yccn/f5nJJhAt2S
Encrypt Text: TslZlCqLdX4jNmBcfbq49NQLW00iDmaql490DT5ZsM1w4yFyQpkcwUC7Hh
Datos asociados:
Tag: cQzUcj2m1e838jvuZiheVw==
Datos cifrados en claro = KeepCoding te enseña a codificar y a cifrar
[Done] exited with code=0 in 0.402 seconds
```

¿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.

- **RESPUESTA:** Mi propuesta es utilizar ChaCha20\_Poly1305 con datos asociados para darle más confidencialidad.
  - nonce:9Yccn/f5nJJhAt2S
  - datos asociados: RGF0b3MgY2lmcmFkb3MgeSBhdXRlbnRpY2Fkb3M=
  - texto cifrado:  
TslZlclqLdX4jNmBcfbq49NQLW00iDmaql490DT5ZsM1w4yFyQpkcwUC7Hh  
o=
  - tag: h2QsEZwsPnEO8PGR5EiYFg==

```
Chacha20-poly_Eje3.py x
criptografia en flujo > Chacha20-poly_Eje3.py > ...
1 from Crypto.Cipher import ChaCha20_Poly1305
2 from base64 import b64decode, b64encode
3 from Crypto.Random import get_random_bytes
4 import json
5
6 try:
7
8     textoPlano = bytes('KeepCoding te enseña a codificar y a cifrar', 'UTF-8')
9     clave = bytes.fromhex('AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480C3120')
10    #Importante NUNCA debe fijarse el nonce
11    nonce_mensaje=b64decode("9Yccn/f5nJJhAt2S")
12    #Con la clave y con el nonce se cifra. El nonce debe ser único por mensaje
13    datos_asociados = bytes('Datos cifrados y autenticados', 'utf-8')
14    cipher = ChaCha20_Poly1305.new(key=clave, nonce=nonce_mensaje)
15    cipher.update(datos_asociados)
16    texto_cifrado, tag = cipher.encrypt_and_digest(textoPlano)
17    #Simulamos el mensaje que se debe enviar, en este caso lo enviaremos todo el contenido en base64
18    mensaje_enviado = { "nonce": b64encode(nonce_mensaje).decode(), "datos asociados": b64encode(datos_asociados).decode(), "texto cifrado": b64encode(texto_cifrado).decode(), "tag": tag.decode() }
19    json_mensaje = json.dumps(mensaje_enviado)
20    print("Mensaje: ", json_mensaje)
21
22
23    #Descifrado...
24
25    decipher = ChaCha20_Poly1305.new(key=clave, nonce=b64decode(mensaje_enviado["nonce"]))
26    decipher.update(datos_asociados)
27    plaintext = decipher.decrypt_and_verify(b64decode(mensaje_enviado["texto_cifrado"]), b64decode(mensaje_enviado["tag"]))
28    print("Mensaje descifrado: ", plaintext.decode())
29
30 except Exception as e:
31    print("Error: ", e)
32
33 if __name__ == '__main__':
34    main()
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS Code

[Done] exited with code=0 in 0.337 seconds

[Running] python -u "c:\Users\jorda\OneDrive\Documents\GitHub\cripto\criptografia en flujo\Chacha20-poly\_Eje3.py"

Mensaje: { "nonce": "9Yccn/f5nJJhAt2S", "datos asociados": "RGF0b3MgY2lmcmFkb3MgeSBhdXRlbnRpY2Fkb3M=", "texto cifrado": "TslZlclqLdX4jNmBcfbq49NQLW00iDmaql490DT5ZsM1w4yFyQpkcwUC7Hh", "tag": "h2QsEZwsPnEO8PGR5EiYFg=" }

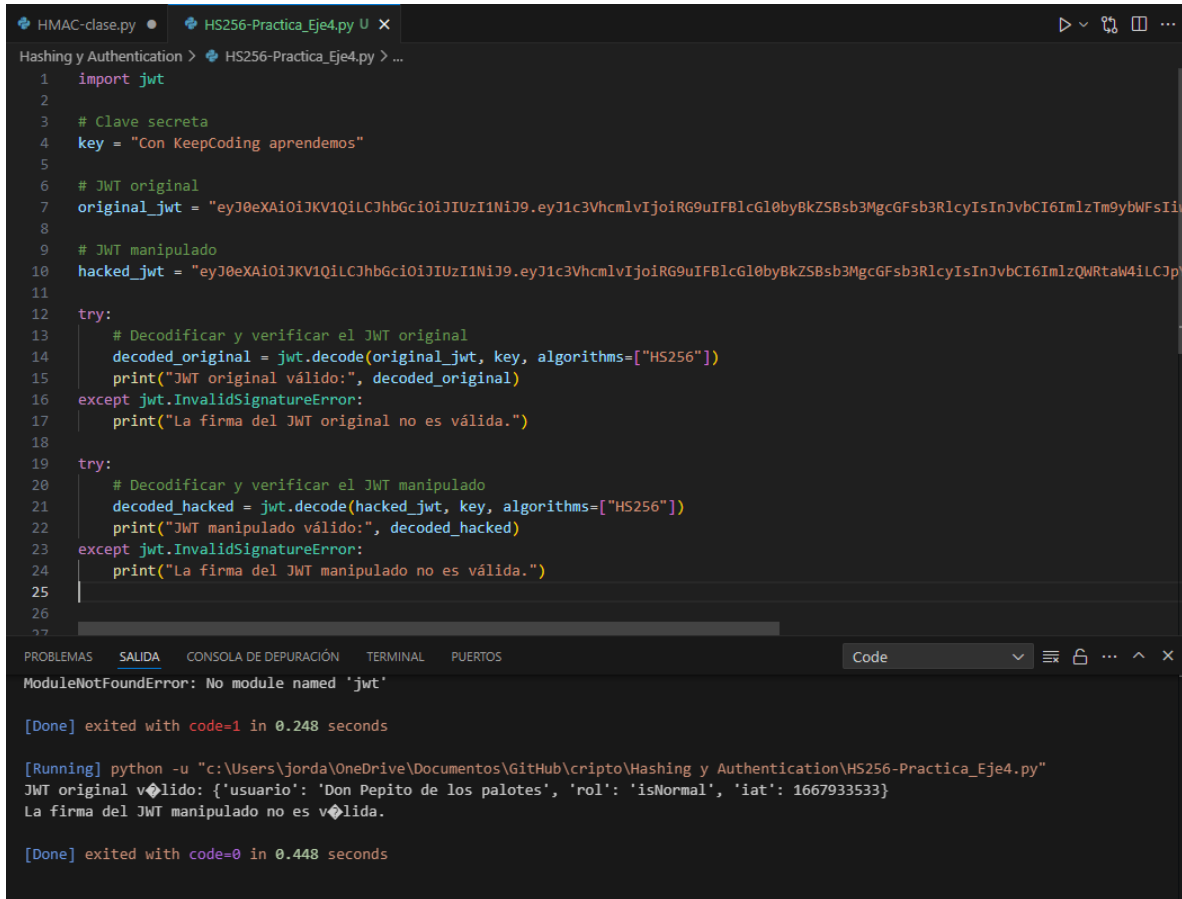
Datos cifrados en claro = KeepCoding te enseña a codificar y a cifrar

[Done] exited with code=0 in 0.44 seconds

4. Tenemos el siguiente jwt, cuya clave es “Con KeepCoding aprendemos”.  
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmVlIjoibG9uIFB1cGl0byBkZSBsb3MgcGFsb3RlcylsInJvbmCI6ImIzTm9ybWFSliwWF0ljoXNjY3OTMzNTMzZfQ.gfhW0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

¿Qué algoritmo de firma hemos realizado?

- **RESPUESTA:** El algoritmo de firma utilizado es HMAC con SHA-256



```
1 import jwt
2
3 # Clave secreta
4 key = "Con KeepCoding aprendemos"
5
6 # JWT original
7 original_jwt = "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmVlIjoibG9uIFB1cGl0byBkZSBsb3MgcGFsb3RlcylsInJvbmCI6ImIzTm9ybWFSliwWF0ljoXNjY3OTMzNTMzZfQ.gfhW0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE"
8
9 # JWT manipulado
10 hacked_jwt = "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmVlIjoibG9uIFB1cGl0byBkZSBsb3MgcGFsb3RlcylsInJvbmCI6ImIzTm9ybWFSliwWF0ljoXNjY3OTMzNTMzZfQ.gfhW0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE"
11
12 try:
13     # Decodificar y verificar el JWT original
14     decoded_original = jwt.decode(original_jwt, key, algorithms=["HS256"])
15     print("JWT original válido:", decoded_original)
16 except jwt.InvalidSignatureError:
17     print("La firma del JWT original no es válida.")
18
19 try:
20     # Decodificar y verificar el JWT manipulado
21     decoded_hacked = jwt.decode(hacked_jwt, key, algorithms=["HS256"])
22     print("JWT manipulado válido:", decoded_hacked)
23 except jwt.InvalidSignatureError:
24     print("La firma del JWT manipulado no es válida.")
25
26
27
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS Code

ModuleNotFoundError: No module named 'jwt'

[Done] exited with code=1 in 0.248 seconds

[Running] python -u "c:\Users\jorda\OneDrive\Documentos\GitHub\cripto\Hashing y Authentication\HS256-Practica\_Eje4.py"

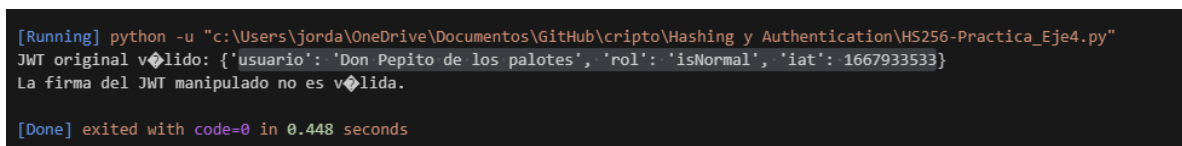
JWT original válido: {'usuario': 'Don Pepito de los palotes', 'rol': 'isNormal', 'iat': 1667933533}

La firma del JWT manipulado no es válida.

[Done] exited with code=0 in 0.448 seconds

¿Cuál es el body del jwt?

- **RESPUESTA:** Es el siguiente:
  - usuario: Don Pepito de los palotes
  - rol: isNormal
  - iat: 1667933533



```
[Running] python -u "c:\Users\jorda\OneDrive\Documentos\GitHub\cripto\Hashing y Authentication\HS256-Practica_Eje4.py"
JWT original válido: {'usuario': 'Don Pepito de los palotes', 'rol': 'isNormal', 'iat': 1667933533}
La firma del JWT manipulado no es válida.
[Done] exited with code=0 in 0.448 seconds
```

Un hacker está enviando a nuestro sistema el siguiente jwt:

**eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmIvIjoiRG9uIFB1cGl0byBkZSBsb3MgcGFsb3RlcylsInJvbCI6ImIzQWRtaW4iLCJpYXQiOiE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODIHRI**

¿Qué está intentando realizar?

- **RESPUESTA:** El hacker esta intentando enviar un JWT que podría ser malicioso para suplantar a un usuario y obtener acceso no autorizado, al intentarlo sin la firma correcta nos enviara una alerta.

```
[Running] python -u "c:\Users\jorda\OneDrive\Documentos\GitHub\cripto\Hashing y Authentication\HS256-Practica_Eje4.py"
JWT original v❖lido: {'usuario': 'Don Pepito de los palotes', 'rol': 'isNormal', 'iat': 1667933533}
❖a firma del JWT manipulado no es v❖lida.

[Done] exited with code=0 in 0.448 seconds
```

¿Qué ocurre si intentamos validarlo con pyjwt?

- **RESPUESTA:** Si intentamos validar el JWT con la clave secreta incorrecta pyjwt levantara un error por que la firma no es valida con la clave correcta decodificara y lo devolverá de la manera correcta.

---

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?

- **RESPUESTA:** SHA3 keccak de 512 bits (el largo del hash es 128 caracteres hexadecimales)

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

**4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833**

¿Qué hash hemos realizado?

- **RESPUESTA:** SHA de 512 bits (el hash tiene una longitud de 256 caracteres hexadecimales)



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

- **RESPUESTA:** El hash SHA3 Keccak :  
**302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf**

```
[Running] python -u "c:\Users\jorda\OneDrive\Documentos\GitHub\cripto\Hashing y Authentication\SHA3_keccak256_Eje5.py"
El hash SHA3 Keccak de 256 bits es: 302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

[Done] exited with code=0 in 0.348 seconds
```

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

- **RESPUESTA:** Destacaría la unidireccionalidad del hash, esto quiere decir que al ir en una sola dirección no se puede obtener el mensaje original con el hash.

---

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.

- **RESPUESTA:** HMAC-SHA256:  
**857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550**

```
HMAC-SHA256_practica_Eje6.py X
Hashing y Authentication > HMAC-SHA256_practica_Eje6.py > ...
1 import hmac
2 import hashlib
3
4 clave_hex = 'A212A51C997E14B4DF08D55967641B0677CA31E049E672A4B06861AA4D5826EB' #KeyStorePracticas Hmac-SHA256
5 clave = bytes.fromhex(clave_hex)
6 mensaje_en_claro = "Siempre existe más de una forma de hacerlo, y más de una solución válida."
7
8
9 hmac_result = hmac.new(clave, mensaje_en_claro.encode('utf-8'), hashlib.sha256)
10 hmac_hex = hmac_result.hexdigest()
11
12 print("HMAC-SHA256:", hmac_hex)
13

[Running] python -u "c:\Users\jorda\OneDrive\Documentos\GitHub\cripto\Hashing y Authentication\HMAC-SHA256_practica_Eje6.py"
HMAC-SHA256: 857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550

[Done] exited with code=0 in 0.391 seconds
```

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?

- **RESPUESTA:** Para mi consideración SHA-1 no es adecuado para almacenar contraseñas por que se sabe que es vulnerable a ataques de fuerza bruta, además de no haber sido diseñado para proteger contraseñas sino para integridad de datos.

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?

- **RESPUESTA:** Aunque SHA-256 es mas seguro que SHA-1 propondría las siguientes mejoras según mis conocimientos:
  - Utilizar un “**salt**” podría ser único por usuario.  
hash = SHA256(salt + contraseña)
  - También se podría aplicar iteraciones múltiples para dificultar ataques de fuerza bruta.

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

¿Qué propondrías?

- **RESPUESTA:** Consideraría algoritmos que ya implementan estas estrategias mencionadas anteriormente de manera segura y eficiente, como lo son **Aragon2**, **BCrypt** o **script**.

---

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?

- **RESPUESTA:** Yo utilizaría **AES-256-GCM** para la confidencialidad y la integridad de la API.

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.

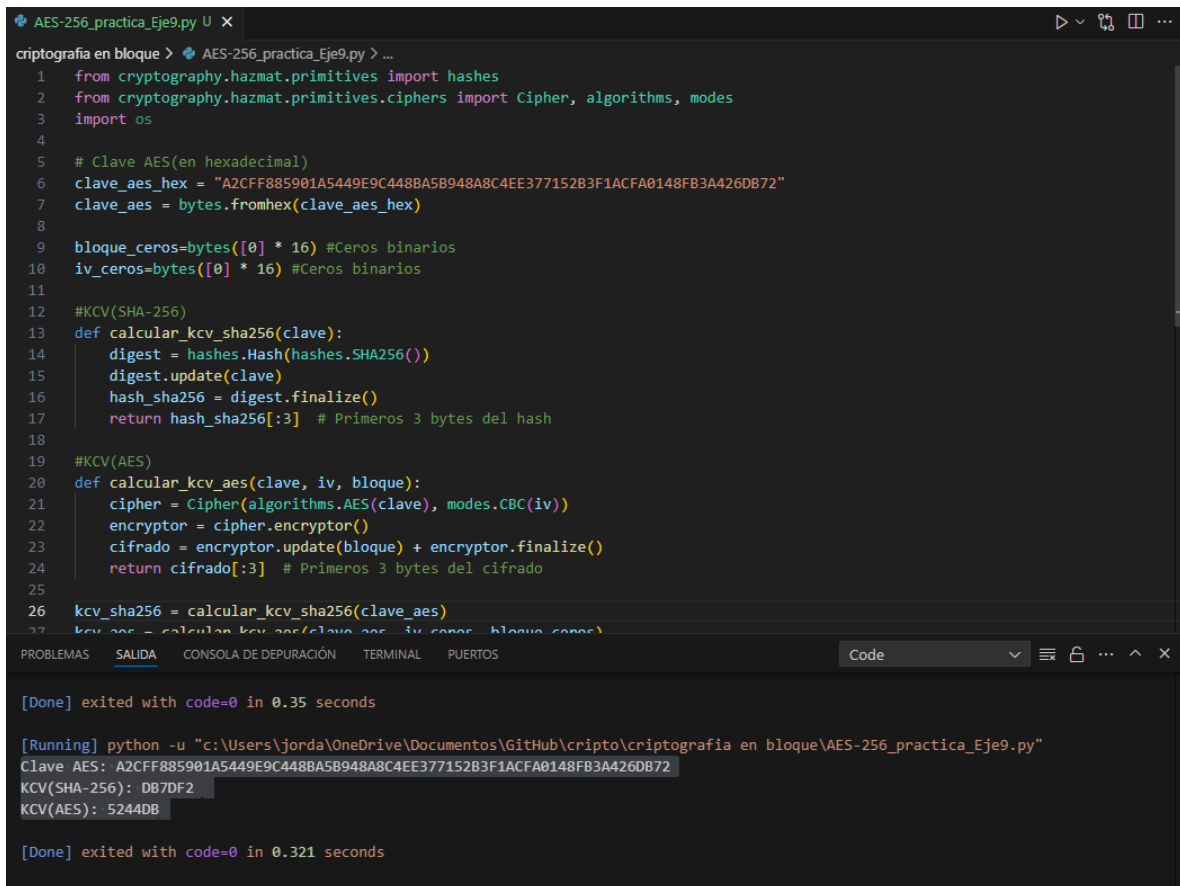
- **RESPUESTA:** Los datos son los siguientes:

- Clave AES:

**A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72**

- KCV(SHA-256): **DB7DF2**

- KCV(AES): **5244DB**



```
AES-256_practica_Eje9.py U X
criptografia en bloque > AES-256_practica_Eje9.py > ...
1 from cryptography.hazmat.primitives import hashes
2 from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
3 import os
4
5 # Clave AES(en hexadecimal)
6 clave_aes_hex = "A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"
7 clave_aes = bytes.fromhex(clave_aes_hex)
8
9 bloque_ceros=bytes([0] * 16) #Ceros binarios
10 iv_ceros=bytes([0] * 16) #Ceros binarios
11
12 #KCV(SHA-256)
13 def calcular_kcv_sha256(clave):
14     digest = hashes.Hash(hashes.SHA256())
15     digest.update(clave)
16     hash_sha256 = digest.finalize()
17     return hash_sha256[:3] # Primeros 3 bytes del hash
18
19 #KCV(AES)
20 def calcular_kcv_aes(clave, iv, bloque):
21     cipher = Cipher(algorithms.AES(clave), modes.CBC(iv))
22     encryptor = cipher.encryptor()
23     cifrado = encryptor.update(bloque) + encryptor.finalize()
24     return cifrado[:3] # Primeros 3 bytes del cifrado
25
26 kcv_sha256 = calcular_kcv_sha256(clave_aes)
27 kcv_aes = calcular_kcv_aes(clave_aes, iv_ceros, bloque_ceros)

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS Code
[Done] exited with code=0 in 0.35 seconds

[Running] python -u "c:\Users\jorda\OneDrive\Documentos\GitHub\cripto\criptografia en bloque\AES-256_practica_Eje9.py"
Clave AES: A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72
KCV(SHA-256): DB7DF2
KCV(AES): 5244DB

[Done] exited with code=0 in 0.321 seconds
```

**10.** El responsable de Raúl, Pedro, ha enviado este mensaje a RRHH:

Se debe ascender inmediatamente a Raúl. Es necesario mejorarle sus condiciones económicas un 20% para que se quede con nosotros.

Lo acompaña del siguiente fichero de firma PGP (MensajeRespoDeRaulARRHH.txt.sig). Nosotros, que pertenecemos a RRHH vamos al directorio a recuperar la clave para verificarlo. Tendremos los ficheros (Pedro-priv.txt y Pedro-publ.txt), con las claves privada y pública. Las claves de los ficheros de RRHH son (RRHH-priv.txt y RRHH-publ.txt) que también se tendrán disponibles. Se requiere verificar la misma, y evidenciar dicha prueba. Así mismo, se requiere firmar el siguiente mensaje con la clave correspondiente de las anteriores, simulando que eres personal de RRHH.

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

Por último, cifra el siguiente mensaje tanto con la clave pública de RRHH como la de Pedro y adjunta el fichero con la práctica.

Estamos todos de acuerdo, el ascenso será el mes que viene, agosto, si no hay sorpresas.

---

**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:

```
b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c96b4a
8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dfa76a329d04e3d3d4ad629793eb00cc
76d10fc00475eb76bfbcb1273303882609957c4c0ae2c4f5ba670a4126f2f14a9f4b6f41aa2ed
ba01b4bd586624659fca82f5b4970186502de8624071be78ccef573d896b8eac86f5d43ca7
b10b59be4acf8f8e0498a455da04f67d3f98b4cd907f27639f4b1df3c50e05d5bf6376808822
6e2a9177485c54f72407fdf358fe64479677d8296ad38c6f177ea7cb74927651cf24b01dee2
7895d4f05fb5c161957845cd1b5848ed64ed3b03722b21a526a6e447cb8ee
```

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?

- **RESPUESTA:** Se genera un texto cifrado diferente al del original debido a la aleatoriedad introducida por RSA-OAEP

```
RSA_OAEP_Desc-practica_Eje11.py M X
Practica > RSA_OAEP_Desc-practica_Eje11.py > ...
1 from Crypto.PublicKey import RSA
2 from Crypto.Cipher import PKCS1_OAEP
3 from Crypto.Hash import SHA256,SHA512
4 import os
5
6 my_path = os.path.abspath(os.getcwd())
7
8 fichero_fpriv = my_path + "\\clave-rsa-oaep-priv.pem"
9 fpriv=open(fichero_fpriv,'r')
10 keypr= RSA.import_key(fpriv.read())
11
12 mensajeCifrado=bytes.fromhex("b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c96b4a8a20d5dbc639e9d83a536")
13 descifrador = PKCS1_OAEP.new(keypr, SHA256)
14 decrypted = descifrador.decrypt(mensajeCifrado)
15 print("Descifrado: ", decrypted.hex())
16
17
18
19
20
21
22
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN **TERMINAL** PUERTOS

(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\jorda\OneDrive\Documents\GitHub\cripto>cd c:/Users/jorda/OneDrive/Documents/GitHub/cripto/Practica

c:\Users\jorda\OneDrive\Documents\GitHub\cripto\Practica>C:/Users/jorda/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/jorda/OneDrive/Documents/GitHub/cripto/Practica/RSA\_OAEP\_Desc-practica\_Eje11.py

Descifrado: e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

c:\Users\jorda\OneDrive\Documents\GitHub\cripto\Practica>

**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?

- **RESPUESTA:** En este caso para mí lo que estamos haciendo mal es reutilizar el nonce ya que debe ser aleatorio nunca debe ser fijo esto nos permite evitar posibles vulnerabilidades.

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 preséntalo en hexadecimal y en base64.

- **RESPUESTA:**

- Texto cifrado en hexadecimal:  
**5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256df06dbffcfb38008ff2ca76b5f5**
- Texto cifrado en base64:  
**Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq92dhLum64MHCV9QePv8FiVt8G2//Ps4AI/yyna19Q==**
- Tag de autenticación en hexadecimal:  
**5807f6deb7009b97e363b8d2cd0a1448**
- Tag de autenticación en base64:  
**Waf23rcAm5fjY7jSzQoUSA==**

```
AES-GCM-practica_Eje12.py X HKDF.py
criptografia en bloque > AES-GCM-practica_Eje12.py > ...
1 from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
2 from cryptography.hazmat.backends import default_backend
3 from cryptography.hazmat.primitives import padding
4 import base64
5 import binascii
6
7
8 key = bytes.fromhex("E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74")
9 nonce = base64.b64decode("9Yccn/f5nJhAt2S")
10 texto_para_cifrar = "He descubierto el error y no volveré a hacerlo mal"
11
12 padder = padding.PKCS7(128).padder()
13 padded_data = padder.update(texto_para_cifrar.encode()) + padder.finalize()
14
15
16 cipher = Cipher(algorithms.AES(key), modes.GCM(nonce), backend=default_backend())
17 encryptor = cipher.encryptor()
18 ciphertext = encryptor.update(padded_data) + encryptor.finalize()
19 tag = encryptor.tag
20
21 ciphertext_hex = binascii.hexlify(ciphertext).decode()
22 ciphertext_base64 = base64.b64encode(ciphertext).decode()
23
24 tag_hex = binascii.hexlify(tag).decode()
25 tag_base64 = base64.b64encode(tag).decode()
26
27 print("Texto cifrado en hexadecimal: ", ciphertext_hex)
28
PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS Code
[Running] python -u "c:\Users\jorda\OneDrive\Documents\GitHub\cripto\criptografia en bloque\AES-GCM-practica_Eje12.py"
Texto cifrado en hexadecimal: 5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256df06dbffcfb38008ff2ca76b5f5
Texto cifrado en base64: Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq92dhLum64MHCV9QePv8FiVt8G2//Ps4AI/yyna19Q==
Tag de autenticación en hexadecimal: 5807f6deb7009b97e363b8d2cd0a1448
Tag de autenticación en base64: Waf23rcAm5fjY7jSzQoUSA==
Texto descifrado: He descubierto el error y no volveré a hacerlo mal

[Done] exited with code=0 in 0.321 seconds
Lín. 37, col. 1 Espacios: 4 UTF-8 CRLF {} Python
```

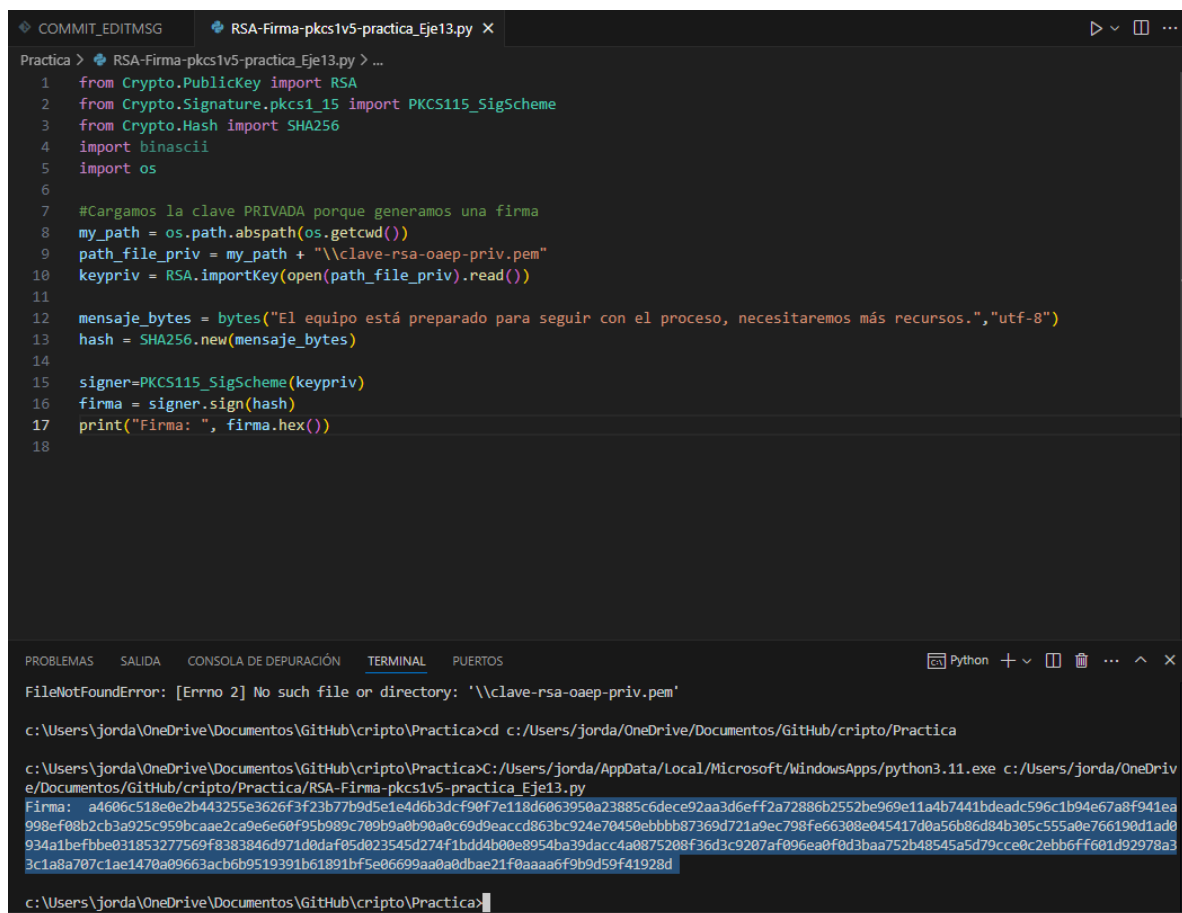
**13.** Se desea calcular una firma con el algoritmo PKCS#1 v1.5 usando las claves contenidas en los ficheros clave-rsa-oaep-priv y clave-rsa-oaep-publ.pem del mensaje siguiente:  
**El equipo está preparado para seguir con el proceso, necesitaremos más recursos.**

¿Cuál es el valor de la firma en hexadecimal?

- **RESPUESTA:**

- Firma:

- a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d60639  
50a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c  
1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9  
a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a9ec798fe66308e0  
45417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569  
f8383846d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a0875208f  
36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33  
c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21f  
0aaaa6f9b9d59f41928d



```
COMMIT_EDITMSG  RSA-Firma-pkcs1v5-practica_Eje13.py X
Practica > RSA-Firma-pkcs1v5-practica_Eje13.py > ...
1 from Crypto.PublicKey import RSA
2 from Crypto.Signature.pkcs1_15 import PKCS115_SigScheme
3 from Crypto.Hash import SHA256
4 import binascii
5 import os
6
7 #Cargamos la clave PRIVADA porque generamos una firma
8 my_path = os.path.abspath(os.getcwd())
9 path_file_priv = my_path + "\\clave-rsa-oaep-priv.pem"
10 keypriv = RSA.importKey(open(path_file_priv).read())
11
12 mensaje_bytes = bytes("El equipo está preparado para seguir con el proceso, necesitaremos más recursos.", "utf-8")
13 hash = SHA256.new(mensaje_bytes)
14
15 signer=PKCS115_SigScheme(keypriv)
16 firma = signer.sign(hash)
17 print("Firma: ", firma.hex())
18

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS
FileNotFoundError: [Errno 2] No such file or directory: '\\clave-rsa-oaep-priv.pem'

c:\Users\jorda\OneDrive\Documents\GitHub\cripto\Practica>cd c:/Users/jorda/OneDrive/Documentos/GitHub/cripto/Practica

c:\Users\jorda\OneDrive\Documents\GitHub\cripto\Practica>c:/Users/jorda/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/jorda/OneDrive/Documentos/GitHub/cripto/Practica/RSA-Firma-pkcs1v5-practica_Eje13.py
Firma: a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21f0aaaa6f9b9d59f41928d

c:\Users\jorda\OneDrive\Documents\GitHub\cripto\Practica>
```

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



```
COMMIT_EDITMSG RSA-Firma-pkcs1v5-practica_Eje13.py RSA-Firma-curvaeliptica-practica_Eje13.py x
Practica > RSA-Firma-curvaeliptica-practica_Eje13.py > ...
1 from cryptography.hazmat.primitives.asymmetric import ed25519
2 from cryptography.hazmat.primitives import hashes
3 import binascii
4 import os
5
6
7 my_path = os.path.abspath(os.getcwd())
8 path_file_priv = my_path + "\\ed25519-priv.gpg"
9
10 with open(path_file_priv, "rb") as f:
11     private_key_bytes = f.read()
12
13 private_key = ed25519.Ed25519PrivateKey.from_private_bytes(private_key_bytes)
14 mensaje_bytes = bytes("El equipo está preparado para seguir con el proceso, necesitaremos más recursos.", "utf-8")
15
16 hash = hashes.Hash(hashes.SHA256())
17 hash.update(mensaje_bytes)
18 mensaje_hash = hash.finalize()
19
20 firma = private_key.sign(mensaje_bytes)
21
22 print("Firma Ed25519: ", firma.hex())
23
```

**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?

- **RESPUESTA:** La clave obtenida es la siguiente.
  - Clave derivada:  
**f6fbc6204bd24b43c42fe1be7d970eeecbcee87481711a64433ea1b7ef655ffa**

```
HKDF-practica_Eje14.py X
Hashing y Authentication > HKDF-practica_Eje14.py > ...
1 from cryptography.hazmat.primitives.kdf.hkdf import HKDF
2 from cryptography.hazmat.primitives.hashes import SHA512
3 from cryptography.hazmat.backends import default_backend
4 import binascii
5
6 # Clave del keystore "cifrado-sim-aes-256"
7 key = bytes.fromhex("A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72")
8 id = bytes.fromhex("e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3")
9 salt = None
10
11 hkdf = HKDF(
12     algorithm=SHA512(),
13     length=32,
14     salt=salt,
15     info=id,
16     backend=default_backend()
17 )
18
19 key_derivada = hkdf.derive(key)
20 key_derivada_hex = binascii.hexlify(key_derivada).decode()
21
22 print("Clave derivada:", key_derivada_hex)
23
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS Code

```
[Running] python -u "c:\Users\jorda\OneDrive\Documentos\GitHub\cripto\Hashing y Authentication\HKDF-practica_Eje14.py"
Clave derivada: f6fbc6204bd24b43c42fe1be7d970eeecbcee87481711a64433ea1b7ef655ffa

[Done] exited with code=0 in 0.357 seconds
```

15. Nos envían un bloque TR31:  
D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A  
5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857F  
D37018E111B

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

A1A10101010101010101010101010102

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

- **RESPUESTA:** El algoritmo que se utiliza comúnmente para este tipo de protección es Triple DES (3DES) o AES, ya que ambos algoritmos son frecuentemente usados en el ámbito de la criptografía bancaria y de tarjetas de pago.

¿Para qué algoritmo se ha definido la clave?

- **RESPUESTA:** Teniendo en cuenta mis conocimientos adquiridos en clase el algoritmo de cifrado es simétrico.

¿Para qué modo de uso se ha generado?

- **RESPUESTA:** Es probable que se use el modo seguro ECB, ya que cifra bloques de manera independiente.

¿Es exportable?

- **RESPUESTA:** No es exportable ya que las claves protegidas con un bloque TR31 no son exportables de manera directa.

¿Para qué se puede usar la clave?

- **RESPUESTA:** Esta clave puede tener varios fines como lo son (Cifrados de datos, Autenticación de transacciones, Generación de claves secundarias).

¿Qué valor tiene la clave?

- **RESPUESTA:** Clave: c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1

```
TR31-practica_Eje15.py U x
Practica > TR31-practica_Eje15.py > ...
1  from psec import tr31
2
3  # Documentado en este fichero
4  # https://github.com/knovichikhin/psec/blob/master/psec/tr31.py
5
6  # Luigi y Felipe hemos compartido una clave (KEK) -- KEK:
7  # A1A10101010101010101010101010103 --> ZMK
8  # TR31 (Key Block): D0144D0AB005000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B
9
10 header, key = tr31.unwrap(
11     kpbk=bytes.fromhex("A1A10101010101010101010101010102"),
12     key_block="D0144D0AB005000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B"
13 )
14 print("Clave=", key.hex())
15
16 print("Key Version ID: " + header.version_id)
17 print("Algoritmo: " + header.algorithm)
18 print("Modo de uso: " + header.mode_of_use)
19 print("Uso de la clave: " + header.key_usage)
20 print("Exportabilidad: " + header.exportability)
21
22 header2, key2 = tr31.unwrap(
23     kpbk=bytes.fromhex("A1A10101010101010101010101010102"),
24     key_block="D0144D0AB005000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B"
25 )
26 print("Clave=", key2.hex())
27
28 print("Key Version ID: " + header2.version_id)
29 print("Algoritmo: " + header2.algorithm)
30 print("Modo de uso: " + header2.mode_of_use)
31 print("Uso de la clave: " + header2.key_usage)
32 print("Exportabilidad: " + header2.exportability)
33
34
35
```