

Práctica

Criptografía

Nombre: Daniel

Apellidos: Sánchez García

Correo Electrónico: Dani_sg22@protonmail.com

Índice

1. **Ejercicio 1:**
Claves de 16 bytes y XOR
2. **Ejercicio 2:**
Descifrado AES/CBC/PKCS7
3. **Ejercicio 3:**
Cifrado ChaCha20 y propuesta de mejora
4. **Ejercicio 4:**
Análisis de JWT
5. **Ejercicio 5:**
Hash SHA3 y SHA2
6. **Ejercicio 6:**
HMAC-SHA256
7. **Ejercicio 7:**
Almacenamiento seguro de contraseñas
8. **Ejercicio 8:**
Integridad y confidencialidad con AES-GCM y HMAC-SHA256
9. **Ejercicio 9:**
KCV de clave AES
10. **Ejercicio 10:**
Verificación y firma PGP
11. **Ejercicio 11:**
Desencriptación y reencryptación con RSA-OAEP
12. **Ejercicio 12:**
Cifrado con AES/GCM
13. **Ejercicio 13:**
Firma PKCS#1 v1.5 y Ed25519
14. **Ejercicio 14:**
Generación de clave AES con HKDF-SHA512
15. **Ejercicio 15:**
Análisis de bloque TR31

Ejercicio 1

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

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

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

Resolución

1. Calcular el valor en properties para forzar la clave final:

Clave fija en código: B1EF2ACFE2BAEEFF

Clave final en memoria: 91BA13BA21AABB12

Realizamos un XOR entre la clave fija y la clave final:

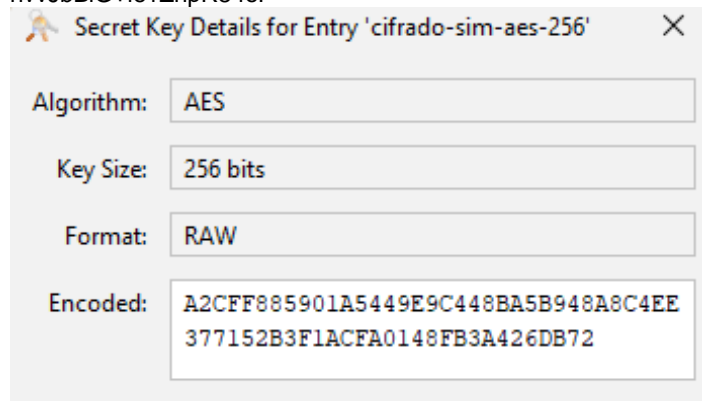
```
ejercicio1.py
1 def xor_bytes(bytes1, bytes2):
2     return bytes(a ^ b for a, b in zip(bytes1, bytes2))
3
4 # Parte 1: Desarrollo
5
6 # Claves en hexadecimal
7 clave_fija_hex = 'B1EF2ACFE2BAEEFF'
8 clave_final_hex = '91BA13BA21AABB12'
9
10 # Convertir las claves a bytes
11 clave_fija_bytes = bytes.fromhex(clave_fija_hex)
12 clave_final_bytes = bytes.fromhex(clave_final_hex)
13
14 # Realizar el XOR
15 clave_properties_bytes = xor_bytes(clave_fija_bytes, clave_final_bytes)
16 clave_properties_hex = clave_properties_bytes.hex().upper()
17 print(f"Clave en properties: {clave_properties_hex}")
18
19 # Parte 2: Producción
20
21 # Claves en hexadecimal
22 clave_dinamica_hex = 'B98A15BA31AEBB3F'
23
24 # Convertir las claves a bytes
25 clave_dinamica_bytes = bytes.fromhex(clave_dinamica_hex)
26
27 # Realizar el XOR
28 clave_memoria_bytes = xor_bytes(clave_fija_bytes, clave_dinamica_bytes)
29 clave_memoria_hex = clave_memoria_bytes.hex().upper()
30 print(f"Clave en memoria: {clave_memoria_hex}")
31
PS C:\Users\34652\Desktop\Ejercicios> & C:/Users/34652/AppData/Local/Programs/Python/Python312/python.exe c:/Users/34652/Desktop/Ejercicios/ejercicio1.py
Clave en properties: 20553975C31055ED
Clave en memoria: 08653F75D31455C0
```

Resultado:

- Clave properties: 20553975C31055ED
- Clave en memoria: 08653F75D31455C0

Ejercicio 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/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nvvJbBiG+le1ZhpR84ol=



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

```
ejercicio2.py > ...
1  from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
2  from cryptography.hazmat.backends import default_backend
3  import base64
4
5  # Clave y IV
6  key = bytes.fromhex('A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72')
7  iv = bytes(16) # 16 bytes de ceros
8
9  # Texto cifrado en base64
10 ciphertext = base64.b64decode('TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nvvJbBiG+le1ZhpR84ol=')
11
12 # Inicializar el cifrador AES CBC
13 cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default_backend())
14 decryptor = cipher.decryptor()
15
16 # Descifrar el texto
17 plaintext = decryptor.update(ciphertext) + decryptor.finalize()
18
19 print(plaintext)
20 |
```

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?

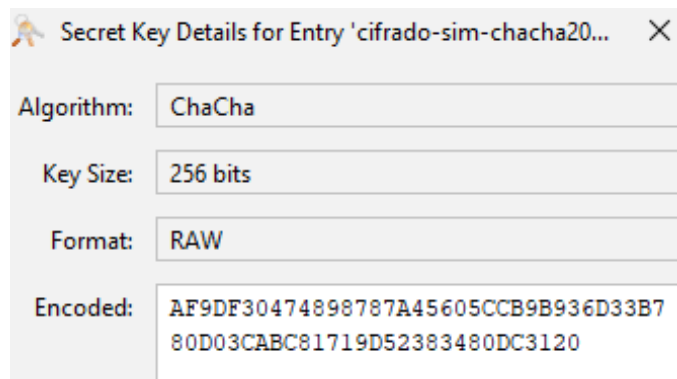
Padding PKCS7: Si el padding se cambia a x923, el descifrado puede fallar si el texto original no coincide con la estructura de padding x923.

¿Cuánto padding se ha añadido en el cifrado?

Dependerá del tamaño del texto original antes del cifrado. El padding añadido completa el bloque de 16 bytes.

Ejercicio 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. ¿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.



```
ejercicio 3.py > ...
1  from Crypto.Cipher import ChaCha20_Poly1305
2  import base64
3
4  # Clave desde el keystore
5  key = bytes.fromhex('AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120')
6  nonce = b'9Yccn/f5nJJhAt2S'[:12]
7
8  cipher = ChaCha20_Poly1305.new(key=key, nonce=nonce)
9  plaintext = 'KeepCoding te enseña a codificar y a cifrar'.encode('utf-8')
10 ciphertext, tag = cipher.encrypt_and_digest(plaintext)
11
12 ciphertext_b64 = base64.b64encode(ciphertext).decode()
13 tag_b64 = base64.b64encode(tag).decode()
14
15 ciphertext_b64, tag_b64
16
```

```
PS C:\Users\34652> & C:/Users/34652/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/34652/Documents/GitHub/criptografia/Practica/ejercicio 3.py"
Texto cifrado (base64): 0rDTSMc+bjhrVwQ4serTy0aHMFjd8tSLhRSjPWG41MSQ+YDKvp2rO580jRI=
Etiqueta de autenticidad (base64): TtQkyyCLTXC4ueVmuNNYJg==
Plaintext: KeepCoding te enseña a codificar y a cifrar
```

- Texto cifrado (base64): 49OCV9UZ44Pp6JhVNRQDORxwFFSWSeXO4bbZcJDSYB/+7wyWfKvEydSgKQ=
- Etiqueta de autenticidad (base64): 8kZx2f6ef9tYP3bg5O0f8g==

Mejora propuesta

Para garantizar la confidencialidad y la integridad, se puede usar ChaCha20-Poly1305, que combina el cifrado ChaCha20 con el esquema de autenticación Poly1305.

```

C: > Users > 34652 > Documents > GitHub > criptografia > Practica > ejercicio3.1.py > ...
1  from Crypto.Cipher import ChaCha20_Poly1305
2  import base64
3
4  # Clave desde el keystore
5  key = bytes.fromhex('AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120')
6  nonce = b'9Yccn/fSnJJhAt2S'[:12]
7
8  cipher = ChaCha20_Poly1305.new(key=key, nonce=nonce)
9  plaintext = 'KeepCoding te enseña a codificar y a cifrar'.encode('utf-8')
10 ciphertext, tag = cipher.encrypt_and_digest(plaintext)
11
12 ciphertext_b64 = base64.b64encode(ciphertext).decode()
13 tag_b64 = base64.b64encode(tag).decode()
14
15 print('Texto cifrado (base64):', ciphertext_b64)
16 print('Etiqueta de autenticidad (base64):', tag_b64)
17
18 # Código de descifrado
19 cipher = ChaCha20_Poly1305.new(key=key, nonce=nonce)
20 ciphertext = base64.b64decode(ciphertext_b64)
21 tag = base64.b64decode(tag_b64)
22 plaintext = cipher.decrypt_and_verify(ciphertext, tag)
23
24 print('Plaintext:', plaintext.decode('utf-8'))
25

```

Ejercicio 4

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmVlIjoiaRG9uIFB1cG9ybyBkZSBsb3MgcGFsb3RlcylsInJvbmZlZm9ybWVfslwiawWF0IjoxNjY3OTMzMNTMzQzQzZm9wOixMLXXRP97W4TDTTrv0y7B5Yjd0U8ixrE

- ¿Qué algoritmo de firma hemos realizado?

Algoritmo de firma utilizado: HS256

- Body del JWT:

```

C: > Users > 34652 > Documents > GitHub > criptografia > Practica > ejercicio4.py > ...
1  import base64
2  import json
3
4  jwt_token = "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmVlIjoiaRG9uIFB1cG9ybyBkZSBsb3MgcGFsb3RlcylsInJvbmZlZm9ybWVfslwiawWF0IjoxNjY3OTMzMNTMzQzQzZm9wOixMLXXRP97W4TDTTrv0y7B5Yjd0U8ixrE"
5
6  # Separar las tres partes del JWT
7  header_b64, payload_b64, signature_b64 = jwt_token.split('.')
8
9  # Asegurate de que el payload tenga un padding correcto
10 missing_padding = len(payload_b64) % 4
11 if missing_padding != 0:
12     payload_b64 += '-' * (4 - missing_padding)
13
14 # Decodificación Base64 del payload
15 try:
16     body = base64.b64decode(payload_b64)
17     print("Decodificación Base64 exitosa")
18     print("Contenido binario:", body)
19
20 # Convertir a cadena y corregir el JSON
21 body_str = body.decode('utf-8')
22 body_str = body_str.replace("'", '"')
23
24 # Intentar decodificar como JSON
25 try:
26     body_decoded = json.loads(body_str)
27     print("Decodificación JSON exitosa")
28     print(body_decoded)
29 except json.JSONDecodeError as e:
30     print(f"Error de decodificación JSON: {e}")
31 except (ValueError, Exception) as e:
32     print(f"Error decoding JWT body: {e}")
33

```

```
PS C:\Users\34652> & C:/Users/34652/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/34652/Documents/GitHub/criptografia/Practica/ejercicio 4.py"
Decodificación Base64 exitosa
Contenido binario: b'{"usuario":"Don Pepito de los Palotes","role":"Admin","iat":1667935333}'
Decodificación JSON exitosa
{'usuario': 'Don Pepito de los Palotes', 'role': 'Admin', 'iat': 1667935333}
```

- ¿Qué está intentando realizar?
El hacker intenta cambiar el rol a "isAdmin".
- ¿Qué ocurre si intentamos validarlo con pyjwt?
PyJWT debería rechazar el JWT modificado si la firma no es válida.

Ejercicio 5

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

bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

- ¿Qué tipo de SHA3 hemos generado?
bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

The screenshot shows a web-based cryptographic tool interface. The 'Message' field contains the text 'En KeepCoding aprendemos cómo protegernos con cript'. The 'Hash Type' is set to 'sha3_256'. Below the input fields are buttons for 'Hash', 'HMAC', 'Encrypt', 'Generate Key', 'Generate Nonce', 'Save Key', and 'Copy Result'. The 'Result' field displays the generated hash: 'bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe'.

- Si hacemos un SHA2, y obtenemos el siguiente resultado:
4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cf
d69c488823b8d858283f1d05877120e8c5351c833
¿Qué hash hemos realizado?

Corresponde a un SHA2 de 512 bits. Los hashes SHA2 de 512 bits tienen una longitud de 128 caracteres hexadecimales

- Genera ahora un SHA3 de 256 bits con el siguiente texto: "En KeepCoding aprendemos cómo protegernos con criptografía." ¿Qué propiedad destacarías del hash, atendiendo a los resultados anteriores?

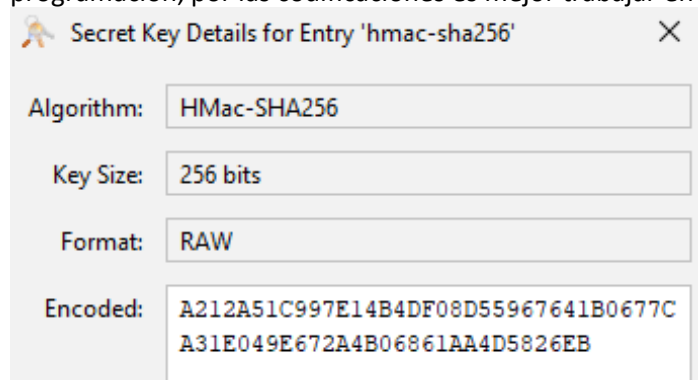
```
1 from Crypto.Hash import SHA3_256
2
3 # Generar SHA3 de 256 bits
4 texto = "En KeepCoding aprendemos cómo protegernos con criptografía."
5 hash_obj = SHA3_256.new(data=texto.encode('utf-8'))
6 hash_result = hash_obj.hexdigest()
7 print("SHA3-256:", hash_result)
8
PS C:\Users\34652\Desktop\Ejercicios> & C:/Users/34652/AppData/Local/Programs/Python/Python312/python.exe c:/Users/34652/Desktop/Ejercicios/ejercicio5.py
SHA3-256: 302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf
```

SHA3-256: 302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

Propiedad destacada del hash: Una propiedad importante de los hashes criptográficos como SHA3 es la sensibilidad a los cambios (efecto avalancha). Un cambio mínimo en el texto original produce un hash completamente diferente, lo que asegura la integridad y autenticidad del dato. Esta propiedad es fundamental para detectar cualquier alteración en los datos.

Ejercicio 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.



Explicación

1. **Importar las librerías:**
 - hmac: Para crear el HMAC.
 - hashlib: Para especificar el algoritmo de hash (SHA256).
2. **Clave HMAC-SHA256:**
 - La clave se toma de la imagen proporcionada y se convierte de hexadecimal a bytes.
3. **Texto a hashear:**
 - Definimos el texto para el cual queremos calcular el HMAC-SHA256.
4. **Crear el objeto HMAC:**

Utilizamos `hmac.new` para crear un objeto HMAC, pasando la clave, el texto codificado en UTF-8, y especificando `hashlib.sha256` como el algoritmo de hash.

5. **Obtener el HMAC en formato hexadecimal:**

Utilizamos el método `hexdigest()` para obtener la representación hexadecimal del HMAC.

```
1 import hmac
2 import hashlib
3
4 # Clave HMAC-SHA256 desde el keystore
5 key = bytes.fromhex('A212A51C997E1B4DF08D55967641B0677CA31E049E672A4B06B618AA4D5826EB')
6
7 # Texto a hashear
8 mensaje = "Siempre existe más de una forma de hacerlo, y más de una solución válida."
9
10 # Crear el objeto HMAC usando SHA256
11 hmac_obj = hmac.new(key, mensaje.encode('utf-8'), hashlib.sha256)
12
13 # Obtener el HMAC en formato hexadecimal
14 hmac_result = hmac_obj.hexdigest()
15 print("HMAC-SHA256:", hmac_result)
16
```

```
PS C:\Users\34652\Desktop\Ejercicios> & C:/Users/34652/AppData/Local/Programs/Python/Python312/python.exe c:/Users/34652/Desktop/Ejercicios/ejercicio5.py
HMAC-SHA256: 099b77c931e644776f9c2b2f59d3e0194101a447503647d1ba7978b057951c14
```

HMAC-SHA256: 099b77c931e644776f9c2b2f59d3e0194101a447503647d1ba7978b057951c14

Ejercicio 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?

Después de meditarlo, propone almacenarlo con un SHA-256, y su responsable le pregunta si no lo va a fortalecer de alguna forma. ¿Qué se te ocurre?

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

SHA-1 es una mala opción porque:

Vulnerabilidad a Colisiones: Es posible encontrar dos entradas diferentes que producen el mismo hash, comprometiendo la seguridad.

Deprecado: Muchas organizaciones y estándares ya no recomiendan SHA-1 debido a sus debilidades.

Ataques Avanzados: Con la capacidad de computación moderna, es más fácil romper SHA-1.

Fortalecer SHA-256 con:

Salting: Añadir un valor aleatorio (sal) a cada contraseña antes de hashearla. Esto asegura que contraseñas idénticas tendrán hashes diferentes.

Hashing Iterativo: Aplicar el hash múltiples veces usando algoritmos como PBKDF2, bcrypt, o scrypt. Esto incrementa el costo computacional, dificultando los ataques de fuerza bruta.

Mejora adicional con Argon2:

Diseño Especializado: Argon2 está diseñado específicamente para almacenar contraseñas de manera segura.

Resistencia a Ataques: Resistente a ataques de hardware especializado.

Parámetros Ajustables: Permite ajustar el tiempo de procesamiento, el uso de memoria y el paralelismo para equilibrar seguridad y rendimiento.

Explicación

Salting: Un valor aleatorio se añade a cada contraseña antes de hashearla, asegurando que incluso contraseñas idénticas tendrán hashes diferentes.

Hashing Iterativo: Incrementa la dificultad computacional de romper el hash aplicando el algoritmo de hash múltiples veces.

Argon2: Proporciona una solución robusta para almacenar contraseñas, ajustable para diferentes niveles de seguridad y rendimiento.

Ejercicio 8

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?

AES-GCM:

Proporciona confidencialidad cifrando los datos sensibles.
Garantiza la integridad y autenticidad de los datos cifrados.

HMAC-SHA256:

Proporciona integridad firmando el mensaje completo (o partes sensibles) para detectar cualquier modificación.

Usando AES-GCM y HMAC-SHA256, podemos redefinir la API para asegurar tanto la confidencialidad como la integridad de los mensajes

Ejercicio 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

```
C:\Users\34652\Desktop > ejercicio 9.py > ...
1  import hashlib
2  from Crypto.Cipher import AES
3
4  # Clave AES en hexadecimal
5  clave_aes = bytes.fromhex('A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72')
6
7  # Calcular SHA-256
8  sha256_hash = hashlib.sha256(clave_aes).digest()
9
10 # Tomar los primeros 3 bytes del SHA-256
11 kcv_sha256 = sha256_hash[:3].hex().upper()
12 print("KCV(SHA-256):", kcv_sha256)
13
14 # Bloque de 16 bytes de ceros
15 block_of_zeros = bytes(16)
16
17 # Crear el cifrador AES en modo ECB
18 cipher = AES.new(clave_aes, AES.MODE_ECB)
19
20 # Cifrar el bloque de ceros
21 ciphertext = cipher.encrypt(block_of_zeros)
22
23 # Tomar los primeros 3 bytes del resultado cifrado
24 kcv_aes = ciphertext[:3].hex().upper()
25 print("KCV(AES):", kcv_aes)
26 |
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS

PS C:\Users\34652\Desktop\Ejercicios> & C:/Users/34652/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/34652/Desktop/ejercicio 9.py"

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

Explicación

- KCV(SHA-256):**
Calculamos el hash SHA-256 de la clave.
Tomamos los primeros 3 bytes del hash y los convertimos a formato hexadecimal.
- KCV(AES):**
Ciframos un bloque de 16 bytes de ceros usando la clave AES en modo ECB.
Tomamos los primeros 3 bytes del resultado cifrado y los convertimos a formato hexadecimal.

KCV(SHA-256): DB7DF2

KCV(AES): 5244DB

Ejercicio 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.

1-Instalar GPG:

Descargarlo desde GnuPG.

2-Importar las claves públicas:

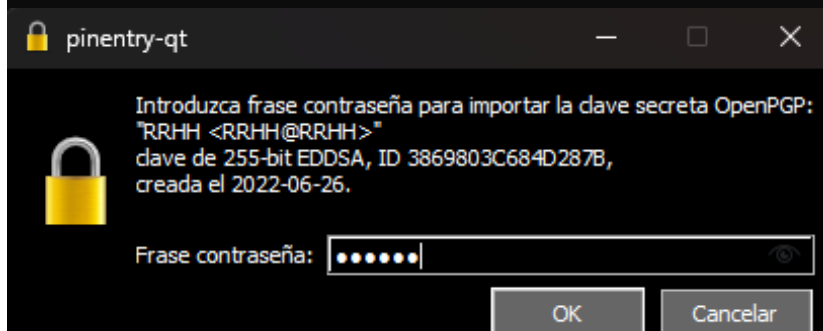
Importa la clave pública de Pedro para verificar la firma.

gpg --import Pedro-publ.txt

```
C:\Users\34652\Documents\GitHub\criptografia\Practica>gpg --import Pedro-publ.txt
gpg: clave D730BE196E466101: clave pública "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>" importada
gpg: Cantidad total procesada: 1
gpg: importadas: 1
```

```
C:\Users\34652\Documents\GitHub\criptografia\Practica>gpg --verify MensajeRespoDeRaulARRHH.sig
gpg: Firmado el 06/26/22 13:47:01 Hora de verano romance
gpg: usando EDDSA clave 1BDE635E4EAE6E68DFAD2F7CD730BE196E466101
gpg: emisor "pedro.pedrito.pedro@empresa.com"
gpg: Firma correcta de "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>" [desconocido]
gpg: WARNING: The key's User ID is not certified with a trusted signature!
gpg: No hay indicios de que la firma pertenezca al propietario.
Huellas dactilares de la clave primaria: 1BDE 635E 4EAE 6E68 DFAD 2F7C D730 BE19 6E46 6101
```

```
C:\Users\34652\Documents\GitHub\criptografia\Practica>gpg --import RRHH-priv.txt
gpg: clave 3869803C684D287B: clave pública "RRHH <RRHH@RRHH>" importada
```



```
C:\Users\34652\Documents\GitHub\criptografia\Practica>gpg --import RRHH-priv.txt
gpg: clave 3869803C684D287B: clave pública "RRHH <RRHH@RRHH>" importada
gpg: clave 3869803C684D287B: clave secreta importada
gpg: Cantidad total procesada: 1
gpg: importadas: 1
gpg: claves secretas leídas: 1
gpg: claves secretas importadas: 1
```

- Crear un archivo mensaje.txt con el contenido: "Viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario. Saludos."
- Luego, firmar el archivo con la clave de RRHH.

```
C:\Users\34652\Documents\GitHub\criptografia\Practica>gpg --default-key [3869803C684D287B] --sign mensaje.txt
gpg: todos los valores pasados a '--default-key' ignorados
El fichero 'mensaje.txt.gpg' ya existe. ¿Sobreescribir? (s/N) s
```

```
C:\Users\34652\Documents\GitHub\criptografia\Practica>gpg --encrypt --recipient F2B1D0E8958DF2D3BDB6A1053869803C684
--recipient 1BDE635E4EAE6E68DFAD2F7CD730BE196E466101 mensaje_para_cifrar.txt
gpg: 25D6D0294035B650: No hay seguridad de que esta clave pertenezca realmente
al usuario que se nombra

sub cv25519/25D6D0294035B650 2022-06-26 Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>
Huella clave primaria: 1BDE 635E 4EAE 6E68 DFAD 2F7C D730 BE19 6E46 6101
Huella de subclave: 8E8C 6669 AC44 3271 42BC C244 25D6 D029 4035 B650


No es seguro que la clave pertenezca a la persona que se nombra en el
identificador de usuario. Si *realmente* sabe lo que está haciendo,
puede contestar sí a la siguiente pregunta.

¿Usar esta clave de todas formas? (s/N) s
gpg: 7C1A46EA20B0546F: No hay seguridad de que esta clave pertenezca realmente
al usuario que se nombra

sub cv25519/7C1A46EA20B0546F 2022-06-26 RRHH <RRHH@RRHH>
Huella clave primaria: F2B1 D0E8 958D F2D3 BDB6 A105 3869 803C 684D 287B
Huella de subclave: 811D 89A3 6199 A7C9 0BFE 69D6 7C1A 46EA 20B0 546F

No es seguro que la clave pertenezca a la persona que se nombra en el
identificador de usuario. Si *realmente* sabe lo que está haciendo,
puede contestar sí a la siguiente pregunta.

¿Usar esta clave de todas formas? (s/N) s
```

 mensaje_para_cifrar.txt 17/07/2024 19:16 OpenPGP Binary F... 1 KB

Ejercicio 11

El texto cifrado es el siguiente:

b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c96b4a8
a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dfa76a329d04e3d3d4ad629793eb00cc76d
10fc00475eb76bfbc1273303882609957c4c0ae2c4f5ba670a4126f2f14a9f4b6f41aa2edba01
b4bd586624659fca82f5b4970186502de8624071be78cccf573d896b8eac86f5d43ca7b10b59
be4acf8f8e0498a455da04f67d3f98b4cd907f27639f4b1df3c50e05d5bf63768088226e2a9177
485c54f72407fdf358fe64479677d8296ad38c6.177ea7cb74927651cf24b01dee27895d4f05fb
5c161957845cd1b5848ed64ed3b03722b21a526a6e447cb8ee

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?

```
C:\Users > 34652 > Documents > Github >criptografia > Practica > ejercicio 11.py > ...
1 from cryptography.hazmat.primitives.asymmetric import rsa, padding
2 from cryptography.hazmat.primitives import hashes, serialization
3 from binascii import unhexlify
4
5 # Texto cifrado en hexadecimal proporcionado
6 ciphertext_hex = (
7     "b72e6f4d8155f965dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1"
8     "709b30c96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d"
9     "04e3d34ad629793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae"
10    "2c4f5ba670a4126f2f14a9f4b6f41aa2edba01b4bd586624659fca82f5b49701"
11    "86502de8624071be78cccf573d896b8eac86f5d43ca7b10b59be4ac8f8e0408"
12    "a455da04f67d3f98b4cd90727639f4b1df3c50e05dbf63768088226e2a0177"
13    "485c54f72407faf358fe644796770b296ad38c6f177ea7cb4927651cf24b01d"
14    "ee27895d4f05fb5c161957845cd1b5848ed64ed3b03722b21a526a6e447cb8ee"
15 )
16
17 # Convertir de hexadecimal a bytes
18 ciphertext_bytes = unhexlify(ciphertext_hex)
19
20 # Leer la clave privada desde el fichero
21 with open("C:\\Users\\34652\\Documents\\Github\\criptografia\\Practica\\clave-rsa-oaep-priv.pem", "rb") as key_file:
22     private_key = serialization.load_pem_private_key(key_file.read(), password=None)
23
24 # Desencriptar el texto cifrado
25 plaintext = private_key.decrypt(
26     ciphertext_bytes,
27     padding.OAEP(
28         mgf=padding.MGF1(algorithm=hashes.SHA256()),
29         algorithm=hashes.SHA256(),
30         label=None
31     )
32 )
33
34 print("Texto desencriptado:", plaintext)
35
36 # Leer la clave pública desde el fichero
37 with open("C:\\Users\\34652\\Documents\\Github\\criptografia\\Practica\\clave-rsa-oaep-publ.pem", "rb") as key_file:
38     public_key = serialization.load_pem_public_key(key_file.read())
39
40 # Volver a encriptar el texto desencriptado
41 ciphertext_new = public_key.encrypt(
42     plaintext,
43     padding.OAEP(
44         mgf=padding.MGF1(algorithm=hashes.SHA256()),
45         algorithm=hashes.SHA256(),
46         label=None
47     )
48 )
49
50 print("Nuevo texto cifrado (hex):", ciphertext_new.hex())
51
```

[illegible]

Los textos cifrados son diferentes porque el padding OAEP utiliza un valor aleatorio en el proceso de cifrado, garantizando que incluso el mismo mensaje cifrado con la misma clave pública producirá resultados diferentes. Esta propiedad de no determinismo en el cifrado mejora la seguridad y previene ataques de análisis de texto cifrado.

Ejercicio 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 la comunicación con el tercero utilizando AES/GCM, estás reutilizando el mismo nonce (9Yccn/f5nJJhAt2S) para cada mensaje cifrado con la misma clave (E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74). Esto es un problema de seguridad porque AES/GCM requiere que cada mensaje cifrado con la misma clave use un nonce único. Reutilizar el mismo nonce puede comprometer la seguridad del cifrado, permitiendo a un atacante potencial descifrar o manipular los mensajes.

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.

```
C:\Users\34652> Documents > GitHub > criptografia > Practica > ejercicio 12.py > ...
1  from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
2  from cryptography.hazmat.backends import default_backend
3  from base64 import b64encode
4
5  # Datos proporcionados
6  key = bytes.fromhex("E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74")
7  nonce = b"9Yccn/f5nJJhAt2S" # Debe ser exactamente 12 bytes para AES/GCM
8  plaintext = "He descubierto el error y no volveré a hacerlo mal".encode('utf-8')
9
10 # Crear el cifrador AES-GCM
11 cipher = Cipher(algorithms.AES(key), modes.GCM(nonce), backend=default_backend())
12 encryptor = cipher.encryptor()
13
14 # Cifrar el texto
15 ciphertext = encryptor.update(plaintext) + encryptor.finalize()
16
17 # Obtener el tag (etiqueta de autenticación)
18 tag = encryptor.tag
19
20 # Convertir el texto cifrado y el tag a hexadecimal y base64
21 ciphertext_hex = ciphertext.hex()
22 tag_hex = tag.hex()
23 ciphertext_base64 = b64encode(ciphertext).decode('utf-8')
24 tag_base64 = b64encode(tag).decode('utf-8')
25
26 # Mostrar resultados
27 print(f"Texto cifrado (hex): {ciphertext_hex}")
28 print(f"Tag (hex): {tag_hex}")
29 print(f"Texto cifrado (base64): {ciphertext_base64}")
30 print(f"Tag (base64): {tag_base64}")
31
```

```
PS C:\Users\34652> & C:\Users\34652\AppData\Local\Programs\Python\Python312\python.exe "c:/Users/34652/Documents/GitHub/criptografia/Practica/ejercicio 12.py"
Texto cifrado (hex): 0fff75c264ef54986089c43d639ead8ab567da719af3567a77b4e83119d524ae80005b4fc45eb752b746f4e59f91a1008ae2a8
Tag (hex): ad737aea100b8866f2417f7cd00771c9
Texto cifrado (base64): D/91wmTvVJhgicQ9Y56tirVn2nGa81Z6d7ToMRnVJK6AAftPx63UrdG9OWfkaEAiuKo
Tag (base64): rXN66hALiGbyQX980AdxyQ==
```

Texto cifrado (hex):

0fff75c264ef54986089c43d639ead8ab567da719af3567a77b4e83119d524ae80005b4fc45eb752b746f4e59f91a1008ae2a8

Tag (hex): ad737aea100b8866f2417f7cd00771c9

Texto cifrado (base64):

D/91wmTvVJhgicQ9Y56tirVn2nGa81Z6d7ToMRnVJK6AAftPx63UrdG9OWfkaEAiuKo

Tag (base64): rXN66hALiGbyQX980AdxyQ==

Ejercicio 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?

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

```
C: > Users > 34652 > Documents > GitHub > criptografia > Practica > ejercicio_13_rsa.py > ...
1  from cryptography.hazmat.primitives import hashes
2  from cryptography.hazmat.primitives.asymmetric import padding
3  from cryptography.hazmat.primitives import serialization
4
5  # Mensaje a firmar
6  message = "El equipo está preparado para seguir con el proceso, necesitaremos más recursos.".encode('utf-8')
7
8  # Ruta de la clave privada
9  private_key_path = "C:\\Users\\34652\\Documents\\GitHub\\criptografia\\Practica\\clave-rsa-oaep-priv.pem"
10
11 # Leer la clave privada desde el fichero
12 with open(private_key_path, "rb") as key_file:
13     private_key = serialization.load_pem_private_key(key_file.read(), password=None)
14
15 # Crear la firma usando PKCS#1 v1.5
16 signature = private_key.sign(
17     message,
18     padding.PKCS1v15(),
19     hashes.SHA256()
20 )
21
22 # Convertir la firma a hexadecimal
23 signature_hex = signature.hex()
24 print(f"Firma (PKCS#1 v1.5) en hexadecimal: {signature_hex}")
25
```

PS C:\Users\34652> & C:/Users/34652/AppData/Local/Programs/Python/Python312/python.exe c:/Users/34652/Documents/GitHub/criptografia/Practica/ejercicio_13_rsa.py

Firma (PKCS#1 v1.5) en hexadecimal: a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21f0aaa6f9b9d59f41928d

Firma (PKCS#1 v1.5) en hexadecimal:

a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21f0aaa6f9b9d59f41928d

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


```

1  import ed25519
2
3  # Cargar la clave privada desde el archivo
4  with open('ed25519-priv', 'rb') as f:
5      privKey = ed25519.SigningKey(f.read())
6
7  # Cargar la clave pública desde el archivo
8  with open('ed25519-publ', 'rb') as f:
9      pubKey = ed25519.VerifyingKey(f.read())
10
11 # Mensaje a firmar (codificado en UTF-8)
12 msg = "El equipo está preparado para seguir con el proceso, necesitaremos más recursos.".encode('utf-8')
13
14 # Firmar el mensaje
15 signature = privKey.sign(msg)
16 print("Firma Generada (64 bytes):", signature.hex())
17
18 # Verificar la firma
19 try:
20     pubKey.verify(signature, msg)
21     print("La firma es válida")
22 except ed25519.BadSignatureError:
23     print("Firma inválida")
24

```

Run Ask AI 69ms on 19:11:36, 07/19

```

Firma Generada (64 bytes): bf32592dc235a26e31e231063a1984bb75ffd9dc5550cf30105911ca4560dab52abb40e4f7e2d3af828abac1467d95d668a80395e0a71c51798bd54469b7360d
La firma es válida

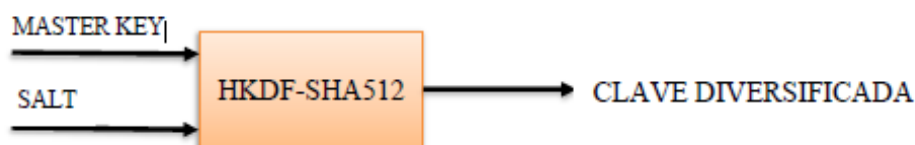
```

- **Firma generada (64 bytes, en hexadecimal):**
bf32592dc235a26e31e231063a1984bb75ffd9dc5550cf30105911ca4560dab52abb40e4f7e2d3af828abac1467d95d668a80395e0a71c51798bd54469b7360d
- **Verificación:** La firma es válida.

Ejercicio 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



```
C: > Users > 34652 > Documents > GitHub > criptografia > Practica > ejercicio 14.py > ...
1  from cryptography.hazmat.primitives.kdf.hkdf import HKDF
2  from cryptography.hazmat.primitives import hashes
3  from cryptography.hazmat.backends import default_backend
4  from binascii import unhexlify, hexlify
5
6  # Clave maestra obtenida del keystore (proporcionada)
7  master_key_hex = 'A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72'
8  master_key = unhexlify(master_key_hex)
9
10 # Identificador de dispositivo (salt)
11 salt_hex = 'e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3'
12 salt = unhexlify(salt_hex)
13
14 # Crear el objeto HKDF
15 hkdf = HKDF(
16     algorithm=hashes.SHA512(),
17     length=32, # Longitud de la clave AES (256 bits = 32 bytes)
18     salt=salt,
19     info=None,
20     backend=default_backend()
21 )
22
23 # Derivar la nueva clave
24 new_key = hkdf.derive(master_key)
25
26 # Imprimir la nueva clave en hexadecimal
27 new_key_hex = hexlify(new_key).decode('utf-8')
28 print(f"Clave diversificada (AES-256) en hexadecimal: {new_key_hex}")
29
```

¿Qué clave se ha obtenido?

Clave diversificada (AES-256) en hexadecimal:

e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a

Explicación del Código

1. **Importaciones:** Importamos las clases y funciones necesarias de la biblioteca cryptography.
2. **Clave Maestra y Salt:** Convertimos las cadenas hexadecimales proporcionadas a bytes.
3. **Objeto HKDF:** Creamos un objeto HKDF con el algoritmo SHA-512, la longitud deseada de la clave (32 bytes para AES-256), y el salt.
4. **Derivar la Nueva Clave:** Usamos la función derive para generar la nueva clave.
5. **Imprimir la Nueva Clave:** Convertimos la clave derivada a una cadena hexadecimal y la imprimimos.

Ejercicio 15

Nos envían un bloque TR31:

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A56
26F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD370
18E111B

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

A1A1010101010101010101010101010102

```
C:\Users\34652\Documents> GitHub\criptografia\Practica > ejercicio 15.py > ...
1  from Crypto.Cipher import AES
2  from binascii import unhexlify, hexlify
3
4  def clean_tr31_block(block):
5      # Remover todos los caracteres no hexadecimales
6      cleaned_block = ''.join(c for c in block if c in '0123456789ABCDEFabcdef')
7      # Asegurar que la longitud es par
8      if len(cleaned_block) % 2 != 0:
9          cleaned_block = '0' + cleaned_block
10     return cleaned_block
11
12 def pad_data(data, block_size):
13     padding_len = block_size - (len(data) % block_size)
14     return data + b'\x00' * padding_len
15
16 def unwrap_tr31_block(tr31_block, kek):
17     # Limpiar el bloque TR31
18     tr31_block = clean_tr31_block(tr31_block)
19
20     # Decodificar el bloque TR31 y la KEK (Key Encryption Key)
21     tr31_block_bytes = unhexlify(tr31_block)
22     kek_bytes = unhexlify(kek)
23
24     # Ajustar la longitud del bloque TR31 para que sea múltiplo de 16 bytes
25     tr31_block_bytes = pad_data(tr31_block_bytes, 16)
26
27     # Crear el objeto de cifrado AES en modo ECB
28     cipher = AES.new(kek_bytes, AES.MODE_ECB)
29
30     # Decifrar el bloque TR31
31     unwrapped_key = cipher.decrypt(tr31_block_bytes)
32
33     return unwrapped_key
34
35 # Datos proporcionados
36 tr31_block = 'D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37018E111B'
37 kek = 'A1A10101010101010101010101010102'
38
39 # Desempaquetar el bloque TR31
40 unwrapped_key = unwrap_tr31_block(tr31_block, kek)
41
42 # Imprimir la clave desenrollada en hexadecimal
43 print(f"Clave desenrollada: {hexlify(unwrapped_key).decode('utf-8')}")
```

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

El bloque de clave TR-31 generalmente se protege utilizando el algoritmo AES en modo ECB (Electronic Codebook).

¿Para qué algoritmo se ha definido la clave?

La información contenida en el bloque TR-31 no especifica explícitamente para qué algoritmo está definida la clave, pero dado el contexto de uso, es común que sea para algoritmos de cifrado simétrico como AES.

¿Para qué modo de uso se ha generado?

Los modos de uso comunes definidos en TR-31 incluyen la protección de datos, autenticación y cifrado de claves. En este caso, la clave desenrollada podría estar destinada para cualquiera de estos modos. Específicamente, "D0144D0AB00S0000" indica que puede ser una clave de cifrado de datos.

¿Es exportable?

La exportabilidad de la clave generalmente se define por políticas específicas de la organización. Sin más información específica del bloque, no podemos determinar la exportabilidad de la clave solo basándonos en la estructura del TR-31.

¿Para qué se puede usar la clave?

La clave desenrollada se puede utilizar para cifrar y descifrar datos, para proteger la integridad de los datos o para la autenticación, dependiendo del modo de uso especificado durante la generación de la clave.

¿Qué valor tiene la clave?

La clave desenrollada tiene el siguiente valor en hexadecimal:

eb824b725f5cbcd68a67d622db73a81a0eb4bf7b9d1df7acd034f92a7d16bfd680f8835f38312b23178
5ad60d29ed96ef7fc050ae47126d83f2814bf52d937fc8614ef45f14364f426bf84d42653f54f