

INFORME PRÁCTICA DE CRIPTOGRAFÍA

**ALEXANDRA
STEFANIA
OANE**

ÍNDICE

EJERCICIO 1.....	3
EJERCICIO 2.....	6
EJERCICIO 3.....	11
EJERCICIO 4.....	11
EJERCICIO 5.....	14
EJERCICIO 6.....	17
EJERCICIO 7.....	18
EJERCICIO 8.....	19
EJERCICIO 9.....	25
EJERCICIO 10.....	26
EJERCICIO 11.....	26

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 A1EF2ABFE1AAEEFF, mientras que en desarrollo sabemos que la clave final (en memoria) es B1AA12BA21AABB12. ¿Qué valor ha puesto el Key Manager en properties para forzar dicha clave final?

Se realiza un XOR con la clave fija y la clave final. El valor que nos devuelve en hexadecimal.

```
1  from operator import xor
2  from multiplicar import *
3  def xor_data(binary_data_1, binary_data_2):
4      return bytes([b1 ^ b2 for b1, b2 in zip(binary_data_1, binary_data_2)])
5
6
7  #K1 (key manager) ^ K2 (desarrollador) = K (keyStore)
8
9
10
11 K1= bytes.fromhex('A1EF2ABFE1AAEEFF')
12 K2= bytes.fromhex('B1AA12BA21AABB12')
13
14 K= xor_data(K1,K2)
15 K_hex = xor_data(K1,K2).hex()
16
17 print("K=K1^K2: ", K_hex)
18
19 K22 = xor_data(K,K1)
20
21 print("K2=K3 ^K1: ", K22.hex())
22
23 K1= xor_data(K2,K1)
24
25 print("K1=K2 ^ K: ", K1.hex())
26
27
28 #solucion
29 #b'\n\x07\x06\x14\x06\t\x03\x01\x07\r'
30 #0a07061406090301070d
31 #0a07061406090301070d
32 #K=K1^K2:  18653f05d00455dd
33 #K2=K3 ^K1:  b98a15ba31aebb22
34 #K1=K2 ^ K:  18653f05d00455dd
35 #K=K1^K2:  10453805c00055ed
36 #K2=K3 ^K1:  b1aa12ba21aabb12
37 #K1=K2 ^ K:  10453805c00055ed
38
```

SOLUCIÓN:

```
K1=K2 ^ K: 10453805c00055ed
```

La clave fija, recordemos es A1EF2ABFE1AAEEFF, mientras que en producción sabemos que la parte dinámica que se modifica en los ficheros de propiedades es B98A15BA31AE BB22. ¿Qué clave será con la que se trabaje en memoria?

SOLUCIÓN:

```
K1=K2 ^ K: 18653f05d00455dd
```

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

zcFJxR1fzaBj+gVWFRAah1N2wv+G2P01ifrKejICaGpQkPnZMiexn3WXlGYX5WnNlosyKfkNKG9GGSgG1awaZg==

Para este caso, se ha usado un AES/CBC/PKCS7.

Si lo desciframos, ¿qué obtenemos? ¿Qué ocurre si decidimos cambiar el padding a x923 en el descifrado? ¿Cuánto padding se ha añadido en el cifrado? Como truco, estudiar el resultado en hexadecimal, cuando lo imprimas en consola introducir un string para diferenciar el final, como “|*****|”.

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

```
home > kali > Desktop > EJERCICIOS_PRACTICA_ENTREGA > Ej2.py > ...
1  import json
2  from base64 import b64encode, b64decode
3  from Crypto.Cipher import AES
4  from Crypto.Util.Padding import pad, unpad
5  from Crypto.Random import get_random_bytes
6
7
8  clave_bytes = bytes.fromhex('e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72')
9
10 iv_bytes = bytes.fromhex('00000000000000000000000000000000')
11 dato_cifrado = ("zcFJxR1fzaBj+gVWFRAah1N2wv+G2P01ifrKejICaGpQkPnZMiexn3WXlGYX5WnNlosyKfkNKG9GGSgG1awaZg==")
12 dato_cifrado_bytes = b64decode(dato_cifrado)
13 cifrado = AES.new(clave_bytes, AES.MODE_CBC, iv_bytes)
14 # Desciframos
15 dato_desc_bytes = cifrado.decrypt(dato_cifrado_bytes)
16 # Quitar el padding
17 dato_descifrado_unpad_bytes = unpad(dato_desc_bytes, AES.block_size, style="pkcs7")
18
19 print("En hex: ", dato_desc_bytes.hex())
20 print("El texto en claro es: ", dato_desc_bytes.decode("utf-8"))
```

Solución:

```
[Running] python -u "/home/kali/Desktop/EJERCICIOS_PRACTICA_ENTREGA/Ej2.py"
En hex: 4573746f20657320756e206369667261646f20656e20626c6f7175652074c3ad7069636f2e20526563756572646120656c2070616464696e672e060606060606
El texto en claro es: Esto es un cifrado en bloque típico. Recuerda el padding.~~~~~
[Done] exited with code=0 in 0.179 seconds
```

En hex:

4573746f20657320756e206369667261646f20656e20626c6f7175652074c3ad7069636f2e20526563756572646120656c2070616464696e672e060606060606

El texto en claro es: Esto es un cifrado en bloque típico. Recuerda el padding.

Cambiar el padding a x923:

```
home > kali > Desktop > EJERCICIOS_PRACTICA_ENTREGA > Ej2.py > ...
1 import json
2 from base64 import b64encode, b64decode
3 from Crypto.Cipher import AES
4 from Crypto.Util.Padding import pad, unpad
5 from Crypto.Random import get_random_bytes
6
7
8 clave_bytes = bytes.fromhex('e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72')
9
10 iv_bytes = bytes.fromhex('00000000000000000000000000000000')
11 dato_cifrado = ("zcFJxR1fzaBj+gVWFRAah1N2wv+G2P01ifrKejICaGpQkPnZMiexn3WXLGYX5WnNIosyKfkNK6G9G6SgG1awaZg==")
12 dato_cifrado_bytes = b64decode(dato_cifrado)
13 cifrado = AES.new(clave_bytes, AES.MODE_CBC, iv_bytes)
14 # Desciframos
15 dato_desc_bytes = cifrado.decrypt(dato_cifrado_bytes)
16 # Quitar el padding
17 dato_descifrado_unpad_bytes = unpad(dato_desc_bytes, AES.block_size, style="x923")
18
19 print("En hex: ", dato_desc_bytes.hex())
20 print("El texto en claro es: ", dato_desc_bytes.decode("utf-8"))
21
```

```
[Running] python -u "/home/kali/Desktop/EJERCICIOS_PRACTICA_ENTREGA/Ej2.py"
Traceback (most recent call last):
  File "/home/kali/Desktop/EJERCICIOS_PRACTICA_ENTREGA/Ej2.py", line 17, in <module>
    dato_descifrado_unpad_bytes = unpad(dato_desc_bytes, AES.block_size, style="x923")
  File "/usr/local/lib/python3.10/dist-packages/Crypto/Util/Padding.py", line 98, in unpad
    raise ValueError("ANSI X.923 padding is incorrect.")
ValueError: ANSI X.923 padding is incorrect.
[Done] exited with code=1 in 0.19 seconds
```

Con padding X923 da error, debido a que al intentar quitar el padding, no corresponde con el formato generado con pkcs7.

¿Cuánto padding se ha añadido en el cifrado? Como truco, estudiar el resultado en hexadecimal, cuando lo imprimas en consola introducir un string para diferenciar el final, como “¡**¡”.***

```
060606060606
```

El padding que se ha añadido es de 6 bytes.

EJERCICIO 3. Se requiere cifrar el texto “Este curso es de lo mejor que podemos encontrar en el mercado”. La clave para ello, tiene la etiqueta en el keyStore “cifrado-sim-chacha-256”. El nonce “9Yccn/f5nJJhAt2S”. 979df30474898787a45605ccb9b36d33b780d03cab81719d52383480dc3120

¿Cómo podríamos mejorar de forma sencilla el sistema, de tal forma, que no sólo garanticemos la confidencialidad sino, además, la integridad del mismo? Demuestra, tu propuesta por código, así como añade los datos necesarios para evaluar tu solución.

Cifrando el texto con el algoritmo ChaCha 20, se obtiene el siguiente resultado:

```
[Running] python -u "/home/kali/Desktop/EJERCICIO5_PRACTICA_ENTREGA/EJERCICIO.3.2.py"
nonce = f5871c9ff7f99c926102dd92
Mensaje cifrado en HEX = 6be4dcbd3fb6b666eca8a41ecd92897c5a6d2162dbeeadd6ef0ef47c3ce8ddf8fcc41f3a13ff33bed4d1c94efd575e593fbb5ea9ce4e4f8e170724ebf
Mensaje cifrado en B64 = a+TcvT+2tmbsqKQezZKJfFptIWLb7q3W7w70fDzo3fj8xB86E/8zvtTRYU79V15ZP7vl6pzk5PjhchHJOvw==
Mensaje en claro = Este curso es de lo mejor que podemos encontrar en el mercado
[Done] exited with code=0 in 0.196 seconds
```

Solución:

```
nonce = f5871c9ff7f99c926102dd92
Mensaje cifrado en HEX =
6be4dcbd3fb6b666eca8a41ecd92897c5a6d2162dbeeadd6ef0ef47c3ce8ddf8fcc41f3
a13ff33bed4d1c94efd575e593fbb5ea9ce4e4f8e170724ebf
Mensaje cifrado en B64 =
a+TcvT+2tmbsqKQezZKJfFptIWLb7q3W7w70fDzo3fj8xB86E/8zvtTRYU79V15ZP7vl6p
zk5PjhchHJOvw==
Mensaje en claro = Este curso es de lo mejor que podemos encontrar en
el mercado
```

Realizando la mejora con el algoritmo ChaCha 20_poly1305

```

home > kali > Desktop > EJERCICIOS_PRACTICA_ENTREGA > EJERCICIO.3.1.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('Este curso es de lo mejor que podemos encontrar en el mercado', 'UTF-8')
9     clave_bytes = bytes.fromhex('979df38474898787a45605ccb9b936d33b780d03cab81719d52383480dc3120')
10    nonce_mensaje_bytes = get_random_bytes(12)
11    datos_asociados_bytes = bytes('Datos no cifrados sólo autenticados', 'utf-8')
12    cipher = ChaCha20_Poly1305.new(key=clave_bytes, nonce=nonce_mensaje_bytes)
13    cipher.update(datos_asociados_bytes)
14    texto_cifrado_bytes, tag_bytes = cipher.encrypt_and_digest(textoPlano_bytes)
15    mensaje_enviado = { 'nonce': b64encode(nonce_mensaje_bytes).decode(), "datos asociados": b64encode(datos_asociados_bytes).decode(), "texto cifrado": b64encode(texto_cifrado_bytes).decode(), "tag": tag_bytes.decode('utf-8')}
16    json_mensaje = json.dumps(mensaje_enviado)
17    print("Mensaje: ", json_mensaje)
18
19
20    #Descifrado...
21
22    tag_2 = b64decode("f5871c9ff7f99c926102dd92")
23    decipher = ChaCha20_Poly1305.new(key=clave_bytes, nonce=b64decode(mensaje_enviado["nonce"]))
24    decipher.update(b64decode(mensaje_enviado["datos asociados"]))
25    plaintext_bytes = decipher.decrypt_and_verify(b64decode(mensaje_enviado["texto cifrado"]), b64decode(mensaje_enviado["tag"]))
26    print('Datos cifrados en claro = ', plaintext_bytes.decode('utf-8'))
27 except (ValueError, KeyError) as error:
28     print("Problemas al descifrar....")
29     print("El motivo del error es: ", error)

```

```

[Running] python -u "/home/kali/Desktop/EJERCICIOS_PRACTICA_ENTREGA/EJERCICIO.3.1.py"
Mensaje: {"nonce": "B48lLKwUk4Q7N4Ft", "datos asociados": "RGF0b3Mgbm8gY2lmcmFkb3Mgc8OzbG8gYXV0ZW50aWNhZG9z", "texto cifrado": "ItQcunqQqQihNRCGiTE/VQj+WrEQEzG2s1x4XPREPUI1TN8hUVSZCMoRLVsdn4Jjz9BCbKvIeD6ep2Gp+ZPD9Bw==", "tag": "Qa/FExQjU+cL4k79A45Q3Q==" }
Datos cifrados en claro = Este curso es de lo mejor que podemos encontrar en el mercado"
[Done] exited with code=0 in 0.166 seconds

```

Solución:

```

Mensaje: {"nonce": "B48lLKwUk4Q7N4Ft", "datos asociados":
"RGF0b3Mgbm8gY2lmcmFkb3Mgc8OzbG8gYXV0ZW50aWNhZG9z", "texto cifrado":
"ItQcunqQqQihNRCGiTE/VQj+WrEQEzG2s1x4XPREPUI1TN8hUVSZCMoRLVsdn4Jjz9BCbKv
IeD6ep2Gp+ZPD9Bw==", "tag": "Qa/FExQjU+cL4k79A45Q3Q==" }
Datos cifrados en claro = Este curso es de lo mejor que podemos
encontrar en el mercado"

```

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VhcmIvIjoiRmVsaXBIIiJvZmVhZG9zAwZWRndWV6liwicz9sljoiaXNOOb3JtYWwifQ.-KiAA8cjkamjwRUiNVHgGeJU8k2wiErdxQP_iFXumM8

¿Qué algoritmo de firma hemos realizado?

El algoritmo realizado en este ejercicio es un HMAC-256.

¿Cuál es el body del jwt?

En el body del jwt se muestra los siguientes datos:

usuario: Felipe Rodríguez

rol: isNormal

Se han usado las herramientas de Cyberchef y Jwt.io.

Encoded

PASTE A TOKEN HERE

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3Vhcm1vIjoiaWVsaXB1IFJvZHLDrWd1ZXoiLCJyb2wiOiJpc05vcm1hbCJ9.vbZwgGWgbltdtCSutI1JwrtdZU1cthcovbC0dXb99s8
```

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

PAYLOAD: DATA

```
{
  "usuario": "Felipe Rodriguez",
  "rol": "isNormal"
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  KeepCoding
) ☐ secret base64 encoded
```

Last build: 20 days agoOptionsAbout / Support

Recipe

JWT Decode

Input

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3Vhcm1vIjoiaWVsaXB1IFJvZHLDrWd1ZXoiLCJyb2wiOiJpc05vcm1hbCJ9.vbZwgGWgbltdtCSutI1JwrtdZU1cthcovbC0dXb99s8

Output

{
 "usuario": "Felipe Rodríguez",
 "rol": "isNormal"
}

Un hacker está enviando a nuestro sistema el siguiente jwt:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3Vhcm1vIjoiaWVsaXB1IFJvZHLDrWd1ZXoiLCJyb2wiOiJpc05vcm1hbCJ9.vbZwgGWgbltdtCSutI1JwrtdZU1cthcovbC0dXb99s8

¿Qué está intentando realizar?

Está intentando cambiar el rol de usuario normal a rol de administrador.

¿Qué ocurre si intentamos validarlo con pyjwt?

Encoded

PASTE A TOKEN HERE

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmIvIjoiaWVsaXB1IFJvZHZHJcdTAwZWVndWV6Iiwicm9sIjoiaXNBZG1pb29uLWkiAA8cjkmajwRUiNVHGEJ08k2wiErdxQP_iFXumM8

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  "typ": "JWT",  "alg": "HS256"}
```

PAYLOAD: DATA

```
{  "usuario": "Felipe Rodriguez",  "rol": "isAdmin"}
```

VERIFY SIGNATURE

HMACSHA256(

base64UrlEncode(header) + "." + base64UrlEncode(payload),

KeepCoding

)☐ secret base64 encoded

The screenshot shows a web-based JWT decoding tool. The interface is divided into two main sections: 'Input' and 'Output'. The 'Input' section has a text area containing a JWT token: `eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3Vhcm91IjoiaWVsaXB1IFJvZHLDrWd1ZXoiLCJyb2wiOiJpc0FkbWluIn0.F4tTAt91rSWwBAkLtLNosyuf2s9ZJWsL57hix0KtRGY`. The 'Output' section displays the decoded payload as a JSON object: `{ "usuario": "Felipe Rodriguez", "rol": "isAdmin" }`. The tool also features a 'Recipe' tab, a 'JWT Decode' tab, and various icons for file operations and settings.

Si se intenta validar, nos dará error debido al cambio realizado.

EJERCICIO 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?

Se ha generado un SHA3-256.

```
home > kali > Desktop > ejerciciospractica > EJERCICIO5.py > ...
1  import hashlib
2
3
4  # initiating the "s" object to use the
5  # sha3_256 algorithm from the hashlib module.
6  s = hashlib.sha3_256()
7
8  # will output the name of the hashing algorithm currently in use.
9  print(s.name)
10
11 # will output the Digest-Size of the hashing algorithm being used.
12 print(s.digest_size)
13
14 # providing the input to the hashing algorithm.
15 s.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía", "UTF-8"))
16
17 print(s.hexdigest())
```

Solución:

```
[Running] python -u "/home/kali/Desktop/ejerciciospractica/EJERCICIO5.py"
sha3_256
32
bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

[Done] exited with code=0 in 0.051 seconds
```

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

***4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f646
883 3d77c07cfd69c488823b8d858283f1d05877120e8c5351c833***

¿Qué hash hemos realizado?

Con la herramienta Cyberchef se ha realizado un análisis del tipo de hash.

The screenshot shows the CyberChef web application interface. On the left, under the 'Recipe' tab, the 'Analyse hash' step is selected. The 'Input' pane on the right contains two lines of a hexadecimal hash: '4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f646' and '883 3d77c07cfd69c488823b8d858283f1d05877120e8c5351c833'. The 'Output' pane on the right displays the analysis results: 'Hash length: 128', 'Byte length: 64', and 'Bit length: 512'. Below this, it states: 'Based on the length, this hash could have been generated by one of the following hashing functions:'. A list of potential hash functions follows: SHA-512, SHA3-512, BLAKE-512, ECHO-512, FSB-512, Grøstl-512, JH, MD6, and Spectral Hash. At the bottom of the interface, there is a 'STEP' button, a green 'BAKE!' button with a chef icon, and an 'Auto Bake' checkbox which is checked.

Se observa que contiene 128 caracteres y que corresponde con 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?

```
home > kali > Desktop > ejerciciospractica > EJERCICIO5.py > ...
1  import hashlib
2
3
4  # initiating the "s" object to use the
5  # sha3_256 algorithm from the hashlib module.
6  s = hashlib.sha3_256()
7
8  # will output the name of the hashing algorithm currently in use.
9  print(s.name)
10
11 # will output the Digest-Size of the hashing algorithm being used.
12 print(s.digest_size)
13
14 # providing the input to the hashing algorithm.
15 s.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía.", "UTF-8"))
16
17 print(s.hexdigest())
```

Solución:

```
[Running] python -u "/home/kali/Desktop/ejerciciospractica/EJERCICIO5.py"
sha3_256
32
302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

[Done] exited with code=0 in 0.042 seconds
```

En comparación con el texto y resultado anterior, la propiedad a destacar es el signo de puntuación al final del texto “.”. Esto provoca que el nuevo resultado generado con un SHA3 Keccak de 256 bits sea distinto al anterior.

Cada vez que se añade un caracter nuevo en el texto inicial, automáticamente el hash cambia. Un pequeño cambio realizado genera un cambio drástico en el valor del hash, llamado efecto avalancha.

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.

```
home > kali > Desktop > EJERCICIOS_PRACTICA_ENTREGA > EJERCICIO.6..py > ...
1  from Crypto.Hash import HMAC, SHA256
2
3  secret = bytes.fromhex('2712A51C997E14B4DF08D55967641B0677CA31E049E672A4B06861AA4D5826EB')
4
5  #Generación
6  msg0= bytes('Siempre existe más de una forma de hacerlo, y más de una solución válida.', 'utf-8')
7  h = HMAC.new(secret, msg=msg0, digestmod=SHA256)
8
9  print(h.hexdigest())
10 mac = h.hexdigest()
11
12 #Verificación
13 h2 = HMAC.new(secret, digestmod=SHA256)
14 msg = bytes('Siempre existe más de una forma de hacerlo, y más de una solución válida.', 'utf-8')
15 h2.update(msg)
16 try:
17     h2.hexverify(mac)
18     print("Mensaje validado ok")
19 except ValueError:
20     print("Mensaje validado ko")
21
22
23 #Solución: d0b686743822793fb185263f1fa4ded09bd557bcd341ac53e06dad76238c8e09
24 # Mensaje validado ok
```

```
[Running] python -u "/home/kali/Desktop/EJERCICIOS_PRACTICA_ENTREGA/EJERCICIO.6..py"
d0b686743822793fb185263f1fa4ded09bd557bcd341ac53e06dad76238c8e09
Mensaje validado ok
[Done] exited with code=0 in 0.141 seconds
```

SOLUCIÓN:

```
d0b686743822793fb185263f1fa4ded09bd557bcd341ac53e06dad76238c8e09
Mensaje validado ok
```

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?

El SHA-1 es una mala opción para utilizarlo como mecanismo de almacenamiento debido a que se considera débil. Produce un valor de hash de 160 bits.

Se descubrió que este tipo de hash tiene varios fallos teóricos que podría provocar ataques de colisión. Es una situación muy complicada ya que permitiría que los atacantes cambiar archivos, sin que se observe.

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?

Como se comunica en el enunciado que se propone el almacenamiento de las contraseñas con SHA-256, una de las formas para fortalecer el hash, sería:
SHA-256 + salt.

Un salt es un valor generado por una función criptográficamente segura que se agrega a la entrada de funciones hash para crear hashes únicos para cada entrada, independientemente de que la entrada no sea única. Un salt hace que una función hash parezca no determinista, lo cual es bueno ya que no queremos revelar contraseñas duplicadas a través de nuestro hash.

Es importante utilizar sales únicas, por ejemplo si varios usuarios utilizarán la misma contraseña, ambas contraseñas que tuviesen salt tendrían el “mismo valor”, pero si se elige otro salt para la misma contraseña, se obtienen varias contraseñas únicas y más largas que tienen un valor diferente.

El almacenamiento de contraseñas sería:

Salt+ Hash + Nombre de usuario

Cuando el usuario inicie sesión se puede buscar por el nombre de usuario, agregar el salt a la contraseña proporcionada, codificarlo y verificar si el hash almacenado coincide con el hash calculado.

Es muy importante que el salt sea único para cada hash. Además si el almacenamiento lo permite usar Salt de 32 y 64 bytes con el tamaño real dependiendo de la función de protección que tenga. Cuanto más larga sea el salt aumenta la complejidad computacional de contraseñas de posible ataque.

Las contraseñas se codifican y se añade el salt con el algoritmo “bcrypt”.

Parece que el responsable se ha quedado conforme, tras mejorar la propuesta del SHA256, no obstante, hay margen de mejora. ¿Qué propondrías?

El margen de mejora sería añadir al salt, pepper. Y una mejor propuesta sería argon2.

EJERCICIO 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? ¿Serías capaz de hacer un ejemplo en Python de cómo resolverlo?

Importante, de cara a calcular cualquier cosa, se deben quitar retornos de carro, espacios intermedios, usar las mismas comillas, etc)

EJERCICIO 9. 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.

Para verificar el mensaje que ha enviado el responsable de Pedro, primero importamos las claves en gpg, con los siguientes comandos:

→ gpg --import RRHH-publ.txt

→ gpg --import RRHH-priv.txt

→ gpg --import Pedro-publ.txt

→ gpg --import Pedro-priv.txt

Nos devuelve que se han importado correctamente.

A continuación verificamos:

→ gpg --verify MensajeRespoDeRaulARRHH.txt.sig

Respuesta:

```
gpg: Firmado el dom 26 jun 2022 13:47:01 CEST
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: ATENCIÓN: ¡Esta clave no está certificada por una firma de confianza!
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
gpg: ADVERTENCIA: firma no separada; ¡el archivo MensajeRespoDeRaulARRHH.txt NO
ha sido verificado!
```

```
****
/ant@MOOVE2-15:~/Escritorio/BOOTCAMP_CIBER/Criptografia/criptografia-main/Práctica$ gpg --verify MensajeRespoDeRaulARRHH.txt.sig
gpg: Firmado el dom 26 jun 2022 13:47:01 CEST
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: ATENCIÓN: ¡Esta clave no está certificada por una firma de confianza!
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
gpg: ADVERTENCIA: firma no separada; ¡el archivo MensajeRespoDeRaulARRHH.txt NO ha sido verificado!
```

Si intento verificar:

- MensajeRespoDeRaulARRHH.txt.sig
- MensajeRespoDeRaulARRHH.txt

→ gpg --verify MensajeRespoDeRaulARRHH.txt.sig MensajeRespoDeRaulARRHH.txt

Me devuelve:

→ gpg: no es una firma separada

Para firmar el mensaje con la clave correspondiente siendo de RRHH:

Se crea el fichero con el mensaje desde RRHH: RRHHRespuesta.txt

Para firmar:

→ gpg --local-user F2B1D0E8958DF2D3BDB6A1053869803C684D287B --armor --output
dpc.sig --detach-sign RRHHRespuesta.txt

Se crea el archivo dpc.sig

```

dpc.sig
home > vant > Escritorio > BOOTCAMP_CIBER > Criptografía > criptografia-main > Práctica > EJERCICIO-9 > dpc.sig
1  -----BEGIN PGP SIGNATURE-----
2
3  iHUEABYIAB0WIQTysdDolY3y0722oQU4aYA8aE0ewUCY0aj7gAKCRA4aYA8aE0o
4  e1z+AQDfSmnpIPIGWLSQcsuFa7cxwXOMN6y502cowUUDn4W2HgEA+wtLcFG+f1Fa
5  wj2dw07YellIEceAohYIgbWSxcRp0Aw=
6  =FCK6
7  -----END PGP SIGNATURE-----
8
```

Para cifrar el mensaje con las dos claves públicas de RRHH y Pedro:

```
→ gpg --output FicheroCifradoPractica.gpg --encrypt --recipient
1BDE635E4EAE6E68DFAD2F7CD730BE196E466101 --recipient
F2B1D0E8958DF2D3BDB6A1053869803C684D287B FicheroParacifradoPractica.txt
```

Me devuelve:

```
→ 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

```
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
```

Y se genera el archivo:

```
→ FicheroCifradoPractica.gpg
```

EJERCICIO 10. 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:

7edee3ec0b808c440078d63ee65b17e85f0c1adbc0da1b7fa842f24fb06b332c1560
38062d9daa8ccfe83bace1dca475cfb7757f1f6446840044fe698a631fe882e1a6fc
00a2de30025e9dcc76e74f9d9d721e9664a6319eaa59dc9011bfc624d2a63eb0e449
ed4471ff06c9a303465d0a50ae0a8e5418a1d12e9392faaaf9d4046aa16e424ae1e2
6844bcf4abc4f8413961396f2ef9ffcd432928d428c2a23fb85b497d89190e3cfa49
6b6016cd32e816336cad7784989af89ff853a3acd796813eade65ca3a10bbf58c621
5fdf26ce061d19b39670481d03b51bb0eccc926c9d6e9cb05ba56082a899f9aa72f9
4c158e56335c5594fcc7f8f301ac1e15a938

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

```
home > kali > Desktop > EJERCICIOS_PRACTICA_ENTREGA > EJERCICIO.10..py > ...
1  from ast import Bytes
2  from Crypto.Cipher import PKCS1_OAEP
3  from Crypto.PublicKey import RSA
4  from Crypto.Hash import SHA256
5  import os
6
7  my_path = os.path.abspath(os.getcwd())
8  path_file_publ = my_path + "/clave-rsa-oaep-publ.pem"
9  path_file_priv = my_path + "/clave-rsa-oaep-priv.pem"
10
11 message_enc = bytes.fromhex("7edee3ec0b808c440078d63ee65b17e85f0c1adbc0da1b7fa842f24fb06b332c156038062d9daa8ccfe83bace1dca475cfb7757f1f6446840044fe698a631fe882e1a6fc00a2de30025e9dcc76e74f9d9d721e9664a6319eaa59dc9011bfc624d2a63eb0e449ed4471ff06c9a303465d0a50ae0a8e5418a1d12e9392faaaf9d4046aa16e424ae1e26844bcf4abc4f8413961396f2ef9ffcd432928d428c2a23fb85b497d89190e3cfa496b6016cd32e816336cad7784989af89ff853a3acd796813eade65ca3a10bbf58c6215fdf26ce061d19b39670481d03b51bb0eccc926c9d6e9cb05ba56082a899f9aa72f94c158e56335c5594fcc7f8f301ac1e15a938")
12 key = RSA.importKey(open(path_file_priv).read())
13 cipher = PKCS1_OAEP.new(key, hashAlgo=SHA256)
14 deciphertext = cipher.decrypt(message_enc)
15 print(deciphertext.hex().upper())
```

```
[Running] python -u "/home/kali/Desktop/EJERCICIOS_PRACTICA_ENTREGA/EJERCICIO.10..py"
E2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72
```

```
[Done] exited with code=0 in 0.396 seconds
```

Solución:

```
[Running] python -u
"/home/kali/Desktop/EJERCICIOS_PRACTICA_ENTREGA/EJERCICIO.10..py"
E2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72
```

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

Los textos cifrados son distintos debido a que cada vez que se vuelve a cifrar se añade un padding. En cada cifrado el padding cambia, haciendo que el cifrado sea totalmente distinto.

EJERCICIO 11. 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:E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB7
2 Nonce:9Yccn/f5nJJhAt2S***

¿Qué estamos haciendo mal?

Los datos de comunicación que se emplean no son los correctos.

Es un cifrado con autenticación.

El nonce debe ser distinto, no se debe de reutilizar, en todos los mensajes para evitar que nos puedan hacer un ataque.