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.

home > kali > Desktop > ejercicospractica > EJERCICIO.1.1.py > ...

```python
from operator import xor
from multiplicar import *
def xor_data(binary_data_1, binary_data_2):
    return bytes([b1 ^ b2 for b1, b2 in zip(binary_data_1, binary_data_2)])


#K1 (key manager) ^ K2 (desarrollador) = K (keyStore)



K1= bytes.fromhex('A1EF2ABFE1AAEEFF')
K2= bytes.fromhex('B1AA12BA21AABB12')

K= xor_data(K1,K2)
K_hex = xor_data(K1,K2).hex()

print("K=K1^K2: ", K_hex)

K22 = xor_data(K,K1)

print("K2=K3 ^K1: ", K22.hex())

K1= xor_data(K2,K1)

print("K1=K2 ^ K: ", K1.hex())
```

```
[Running] python -u "/home/kali/Desktop/ejercicospractica/EJERCICIO.1.1.py"
b'\n\x07\x06\x14\x06\t\x03\x01\x07\r'
0a07061406090301070d
0a07061406090301070d
K=K1^K2:  18653f05d00455dd
K2=K3 ^K1:  b98a15ba31aebb22
K1=K2 ^ K:  18653f05d00455dd
K=K1^K2:  10453805c00055ed
K2=K3 ^K1:  b1aa12ba21aabb12
K1=K2 ^ K:  10453805c00055ed

[Done] exited with code=0 in 0.032 seconds
```

La clave fija, recordemos es A1EF2ABFE1AAEEFF, mientras que en producción sabemos que la parte dinámica que se modifica en los ficheros de propiedades es B98A15BA31AEBB22. ¿Qué clave será con la que se trabaje en memoria?

home > kali > Desktop > ejercicospractica > EJERCICIO.1.2.py > ...

```python
from operator import xor
from multiplicar import *
def xor_data(binary_data_1, binary_data_2):
    return bytes([b1 ^ b2 for b1, b2 in zip(binary_data_1, binary_data_2)])


#K1 (key manager) ^ K2 (desarrollador) = K (keyStore)



K1= bytes.fromhex('A1EF2ABFE1AAEEFF')
K2= bytes.fromhex('B98A15BA31AEBB22')

K= xor_data(K1,K2)
K_hex = xor_data(K1,K2).hex()

print("K=K1^K2: ", K_hex)

K22 = xor_data(K,K1)

print("K2=K3 ^K1: ", K22.hex())

K1= xor_data(K2,K1)

print("K1=K2 ^ K: ", K1.hex())
```

```
[Running] python -u "/home/kali/Desktop/ejercicospractica/EJERCICIO.1.2.py"
b'\n\x07\x06\x14\x06\t\x03\x01\x07\r'
0a07061406090301070d
0a07061406090301070d
K=K1^K2:  18653f05d00455dd
K2=K3 ^K1:  b98a15ba31aebb22
K1=K2 ^ K:  18653f05d00455dd
K=K1^K2:  18653f05d00455dd
K2=K3 ^K1:  b98a15ba31aebb22
K1=K2 ^ K:  18653f05d00455dd

[Done] exited with code=0 in 0.077 seconds
```

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+G2P01ifrKejICaGpQkPnZMiexn3WXlGYX5WnNIosy KfkNKG9GGSgG1awaZg==

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 > ...

```python
import json
from base64 import b64encode, b64decode
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes


clave_bytes = bytes.fromhex('e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72')

iv_bytes = bytes.fromhex('00000000000000000000000000000000')
dato_cifrado = ("zcFJxR1fzaBj+gVWFRAah1N2wv+G2P01ifrKejICaGpQkPnZMiexn3WXlGYX5WnNIosyKfkNKG9GGSgG1awaZg==")
dato_cifrado_bytes = b64decode(dato_cifrado)
cifrado = AES.new(clave_bytes, AES.MODE_CBC, iv_bytes)
# Desciframos
dato_desc_bytes = cifrado.decrypt(dato_cifrado_bytes)
# Quitar el padding
dato_descifrado_unpad_bytes = unpad(dato_desc_bytes, AES.block_size,style="pkcs7")

print("En hex: ", dato_desc_bytes.hex())
print("El texto en claro es: ", dato_desc_bytes.decode("utf-8"))
```

```
[Running] python -u "/home/kali/Desktop/EJERCICIOS_PRACTICA_ENTREGA/Ej2.py"
En hex:  4573746f20657320756e206369667261646f20656e20626c6f7175652074c3ad7069636f2e2052656375657264612065 6c2070616464696e672e060606060606
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:
4573746f20657320756e206369667261646f20656e20626c6f7175652074c3ad7069636f2e2052656375657264612065 6c2070616464696e672e060606060606
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 > ...

```python
import json
from base64 import b64encode, b64decode
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes


clave_bytes = bytes.fromhex('e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72')

iv_bytes = bytes.fromhex('00000000000000000000000000000000')
dato_cifrado = ("zcFJxR1fzaBj+gVWFRAah1N2wv+G2P01ifrKejICaGpQkPnZMiexn3WXlGYX5WnNIosyKfkNKG9GGSgG1awaZg==")
dato_cifrado_bytes = b64decode(dato_cifrado)
cifrado = AES.new(clave_bytes, AES.MODE_CBC, iv_bytes)
# Desciframos
dato_desc_bytes = cifrado.decrypt(dato_cifrado_bytes)
# Quitar el padding
dato_descifrado_unpad_bytes = unpad(dato_desc_bytes, AES.block_size,style="x923")

print("En hex: ", dato_desc_bytes.hex())
print("El texto en claro es: ", dato_desc_bytes.decode("utf-8"))
```

```
[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.

¿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*". 979df30474898787a45605ccb9b36d33b780d03cabc81719d52383480dc3120

¿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/EJERCICIOS_PRACTICA_ENTREGA/EJERCICIO.3.2.py"
nonce  =  f5871c9ff7f99c926102dd92
Mensaje cifrado en HEX =  6be4dcbd3fb6b666eca8a41ecd92897c5a6d2162dbeeadd6ef0ef47c3ce8ddf8fcc41f3a13ff33bed4d1c94efd575e593fbbe5ea9ce4e4f8e170724ebf
Mensaje cifrado en B64 =  a+TcvT+2tmbsqKQezZKJfFptIWLb7q3W7w70fDzo3fj8xB86E/8zvtTRyU79V15ZP7vl6pzk5PjhcHJOvw==
Mensaje en claro =  Este curso es de lo mejor que podemos encontrar en el mercado

[Done] exited with code=0 in 0.196 seconds
```

Resultado:

```
nonce  =  f5871c9ff7f99c926102dd92
Mensaje cifrado en HEX =
6be4dcbd3fb6b666eca8a41ecd92897c5a6d2162dbeeadd6ef0ef47c3ce8ddf8fcc41f3
a13ff33bed4d1c94efd575e593fbbe5ea9ce4e4f8e170724ebf
Mensaje cifrado en B64 =
a+TcvT+2tmbsqKQezZKJfFptIWLb7q3W7w70fDzo3fj8xB86E/8zvtTRyU79V15ZP7vl6pz
k5PjhcHJOvw==
Mensaje en claro =  Este curso es de lo mejor que podemos encontrar en
el mercado
```

Realizando la mejora con el algoritmo ChaCha 20_poly1305

```python
from Crypto.Cipher import ChaCha20_Poly1305
from base64 import b64decode, b64encode
from Crypto.Random import get_random_bytes
import json

try:

    textoPlano_bytes = bytes('Este curso es de lo mejor que podemos encontrar en el mercado”', 'UTF-8')
    clave_bytes = bytes.fromhex('979df30474898787a45605ccb9b936d33b780d03cabc81719d52383480dc3120')
    nonce_mensaje_bytes = get_random_bytes(12)
    datos_asociados_bytes = bytes('Datos no cifrados sólo autenticados', 'utf-8')
    cipher = ChaCha20_Poly1305.new(key=clave_bytes, nonce=nonce_mensaje_bytes)
    cipher.update(datos_asociados_bytes)
    texto_cifrado_bytes, tag_bytes = cipher.encrypt_and_digest(textoPlano_bytes)
    mensaje_enviado = { "nonce": b64encode(nonce_mensaje_bytes).decode(),"datos asociados": b64encode(datos_asociados_bytes).decode(), "texto cifrado": b64enc
    json_mensaje = json.dumps(mensaje_enviado)
    print("Mensaje: ", json_mensaje)


    #Descifrado...

    tag_2 = b64decode("f5871c9ff7f99c926102dd92")
    decipher = ChaCha20_Poly1305.new(key=clave_bytes, nonce=b64decode(mensaje_enviado["nonce"]))
    decipher.update(b64decode(mensaje_enviado["datos asociados"]))
    plaintext_bytes = decipher.decrypt_and_verify(b64decode(mensaje_enviado["texto cifrado"]),b64decode(mensaje_enviado["tag"]))
    print('Datos cifrados en claro = ',plaintext_bytes.decode('utf-8'))
except (ValueError, KeyError) as error:
    print("Problemas al descifrar....")
    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
+WrEQEzG2s1x4XPREPU1TN8hUVSZCMoRLVsdn4Jjz9BCbKvieD6ep2Gp+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
```

```
Mensaje:  {"nonce": "B48lLKwUk4Q7N4Ft", "datos asociados":
"RGF0b3Mgbm8gY2lmcmFkb3Mgc8OzbG8gYXV0ZW50aWNhZG9z", "texto cifrado":
"ItQcunqQqQihNRCGiTE/VQj+WrEQEzG2s1x4XPREPU1TN8hUVSZCMoRLVsdn4Jjz9BCbKv
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.eyJ1c3VhcmlvIjoiRmVsaXBlIFJvZHJcdTAwZWRndWV6Iiwicm9sIjoiaXNOb3JtYWwifQ.-KiAA8cjkamjwRUiNVHgGeJU8k2wiErdxQP_iFXumM8

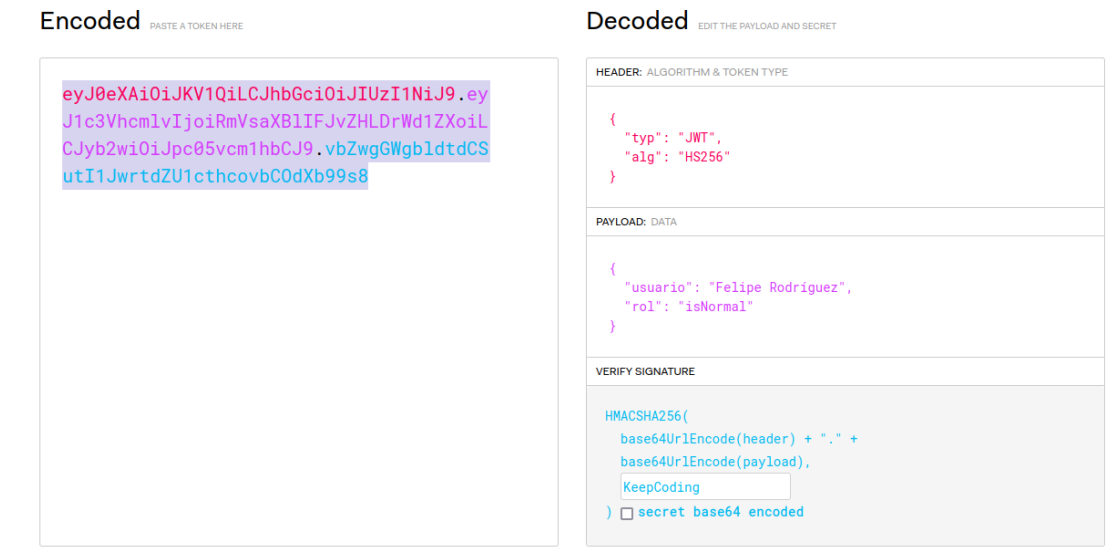
¿Qué algoritmo de firma hemos realizado?

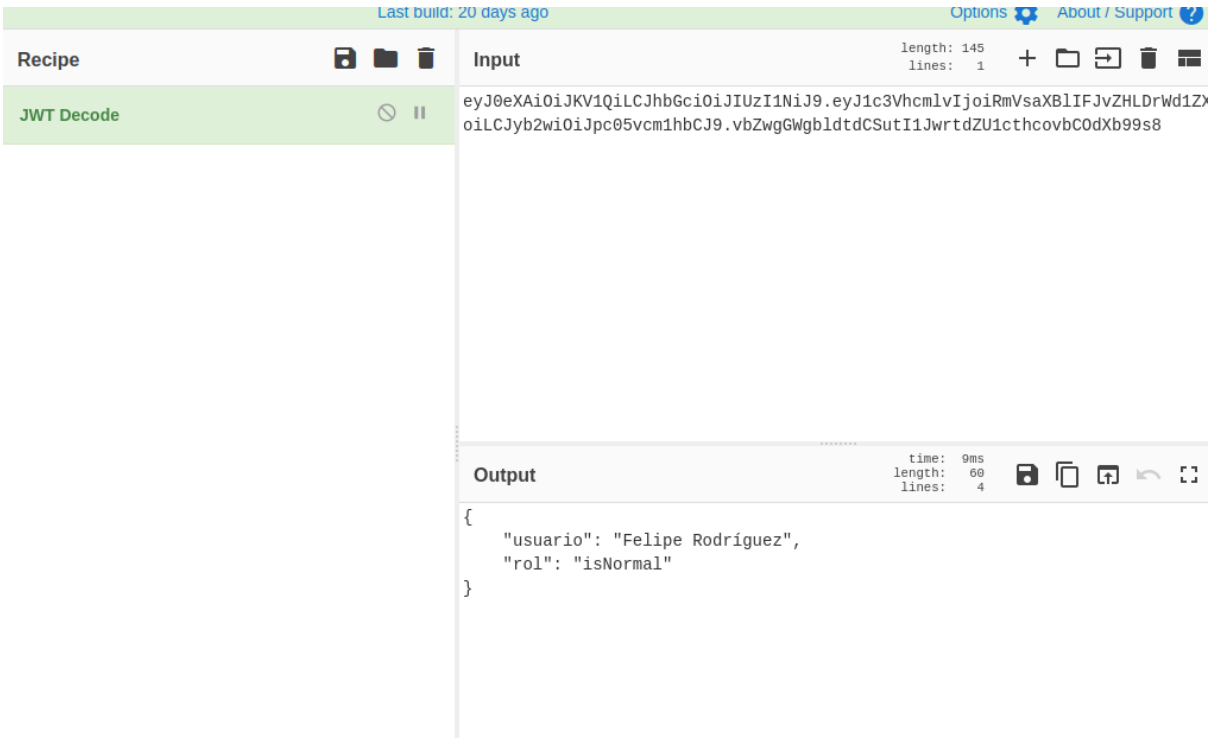
El algoritmo realizado en este ejercicio es un HMAC-256.

¿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.





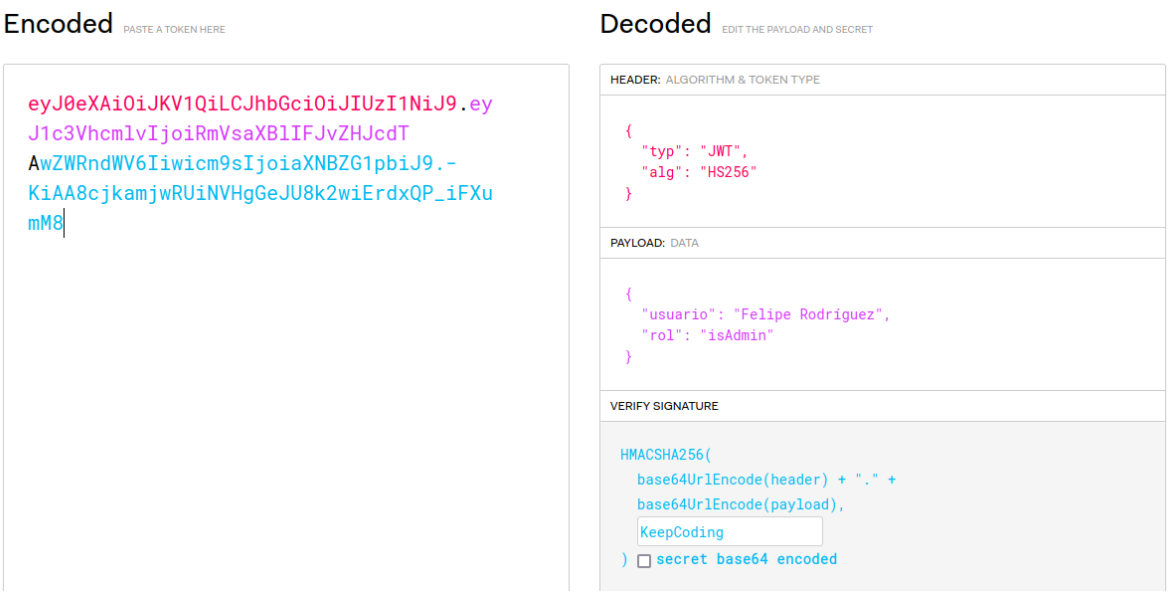
Un hacker está enviando a nuestro sistema el siguiente jwt:

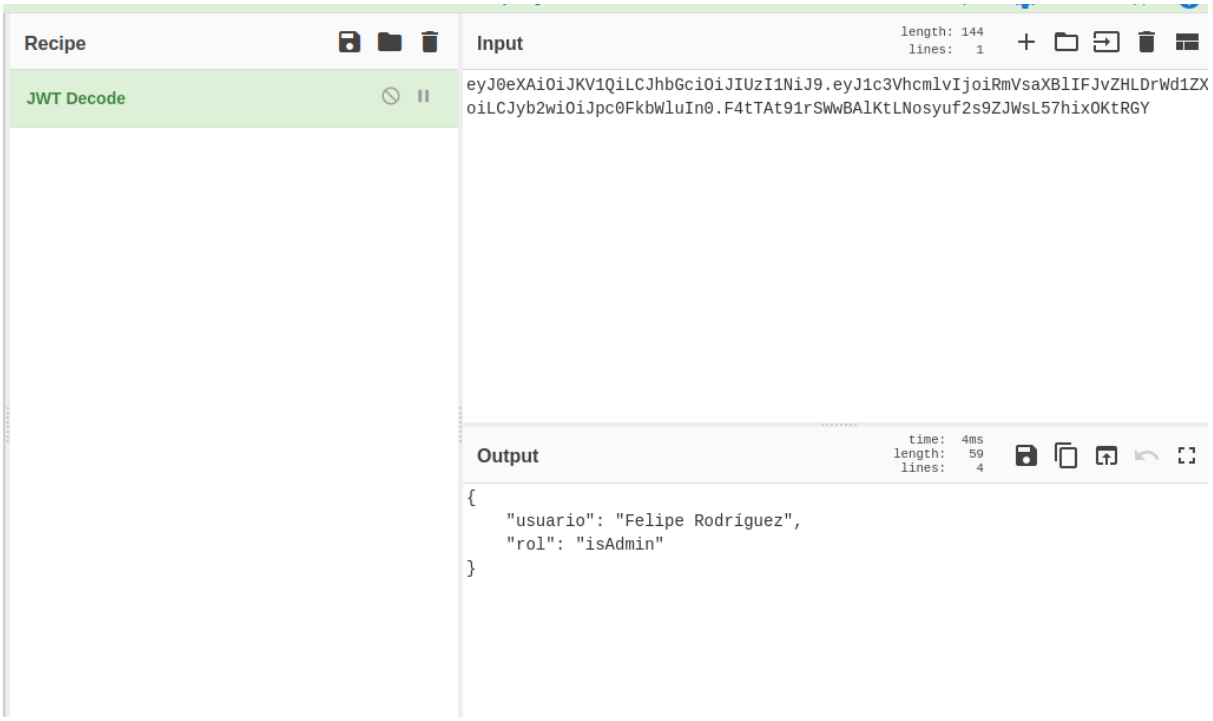
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRmVsaXBlIFJvZHJcdTAwZWRndWV6Iiwicm9sIjoiaXNBZG1pbiJ9.-KiAA8cjkamjwRUiNVHgGeJU8k2wiErdxQP_iFXumM8

¿Qué está intentando realizar?

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

¿Qué ocurre si intentamos validarlo con pyjwt?





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 > ejercicospractica > EJERCICIO5.py > ...

```python
import hashlib


# initiating the "s" object to use the
# sha3_256 algorithm from the hashlib module.
s = hashlib.sha3_256()

# will output the name of the hashing algorithm currently in use.
print(s.name)

# will output the Digest-Size of the hashing algorithm being used.
print(s.digest_size)

# providing the input to the hashing algorithm.
s.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía","UTF-8"))

print(s.hexdigest())
```

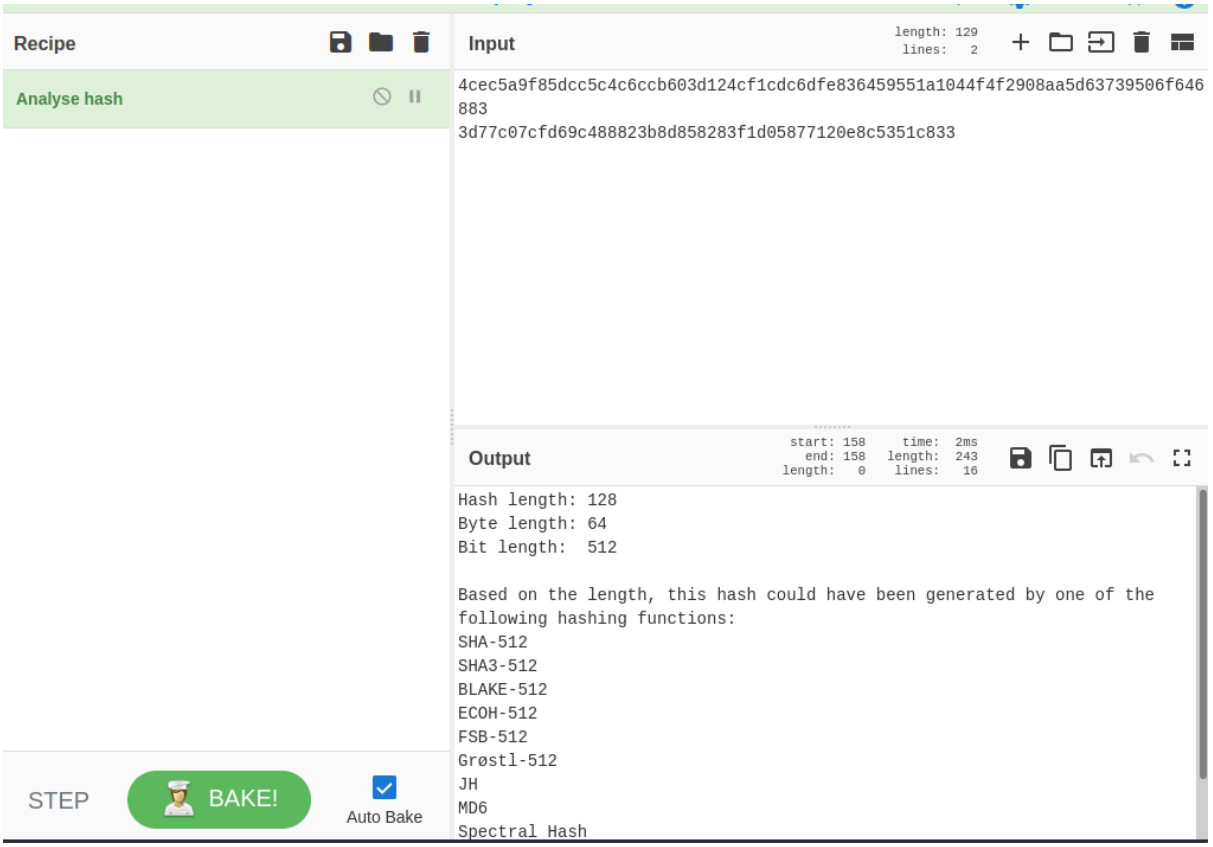
```
[Running] python -u "/home/kali/Desktop/ejercicospractica/EJERCICIO5.py"
sha3_256
32
bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

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

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

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f646883 3d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

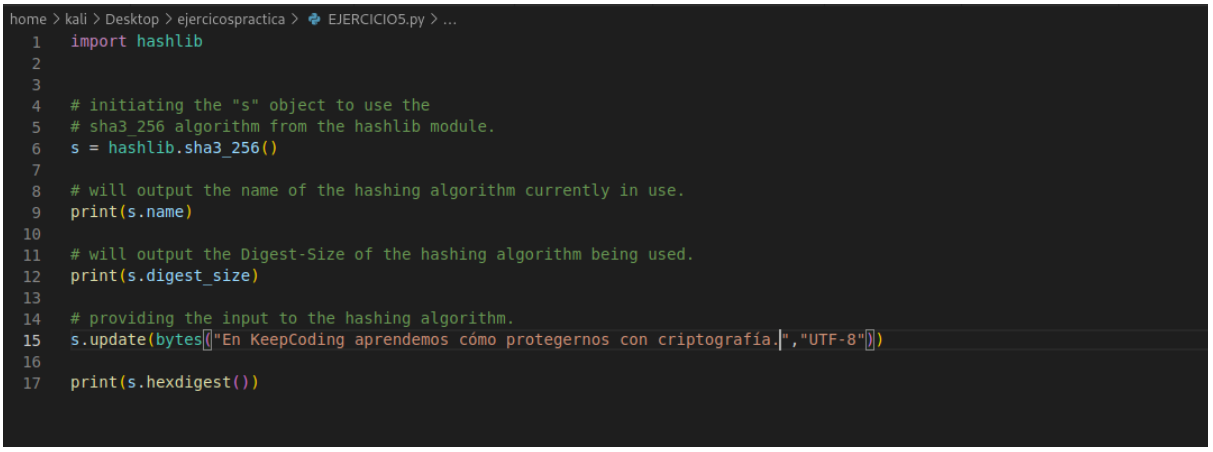
¿Qué hash hemos realizado?

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

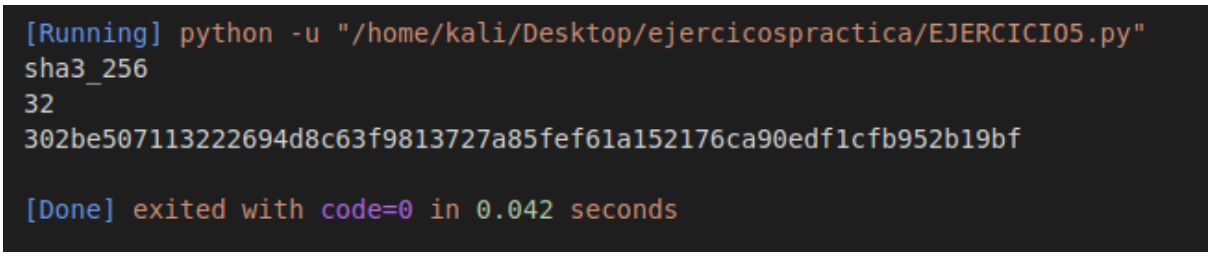


Se observa que el tamaño del hash es de 128 bits y es SHA3-512.

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 > ejercicospractica > 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())
```



```
[Running] python -u "/home/kali/Desktop/ejercicospractica/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.

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 > ...

```python
from Crypto.Hash import HMAC, SHA256

secret = bytes.fromhex('2712A51C997E14B4DF08D55967641B0677CA31E049E672A4B06861AA4D5826EB')

#Generación
msg0= bytes('Siempre existe más de una forma de hacerlo, y más de una solución válida.', 'utf-8')
h = HMAC.new(secret, msg=msg0, digestmod=SHA256)

print(h.hexdigest())
mac = h.hexdigest()

#Verificación
h2 = HMAC.new(secret, digestmod=SHA256)
msg = bytes('Siempre existe más de una forma de hacerlo, y más de una solución válida.', 'utf-8')
h2.update(msg)
try:
  h2.hexverify(mac)
  print("Mensaje validado ok")
except ValueError:
  print("Mensaje validado ko")


  #Solución: d0b686743822793fb185263f1fa4ded09bd557bcd341ac53e06dad76238c8e09
  # 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.

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
5fdf26ce061d19b39670481d03b51bb0eecc926c9d6e9cb05ba56082a899f9aa72f9
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 > ...

```python
from ast import Bytes
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA256
import os

my_path = os.path.abspath(os.getcwd())
path_file_publ = my_path + "/clave-rsa-oaep-publ.pem"
path_file_priv = my_path + "/clave-rsa-oaep-priv.pem"

message_enc = bytes.fromhex("7edee3ec0b808c440078d63ee65b17e85f0c1adbc0da1b7fa842f
key = RSA.importKey(open(path_file_priv).read())
cipher = PKCS1_OAEP.new(key, hashAlgo=SHA256)
deciphertext = cipher.decrypt(message_enc)
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:E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB72 Nonce:9Yccn/f5nJJhAt2S

¿Qué estamos haciendo mal?

Los datos de comunicación que se emplean no son los correctos.
Es un cifrado con autentificación.
El nonce debe ser distinto, no se debe de reutilizar, en todos los mensajes para evitar que nos puedan hacer un ataque.