Portada

CIBERSEGURIDAD

‘Bootcamp IX’

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# Informe Practica Criptografía.

Maximiliano Dariel Altamirano.

Academia Keep Coding.

Consigna

**PRÁCTICA: CRIPTOGRAFÍA**

**Objetivo**:

Contestar un conjunto de preguntas / ejercicios relacionados con la materia aprendida en el

curso demostrando la adquisición de conocimientos relacionados con la criptografía.

**Detalles:**

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

durante el módulo.

Cualquier password que sea necesario tendrá un valor **123456**.

**Evaluación**

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

ha de contener:

● Los enunciados seguido de las respuestas justificadas y evidenciadas.

● En el caso de que se hayan usado comandos / herramientas también se deben

nombrar y explicar los pasos realizados.

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

al informe.

Se va a valorar el proceso de razonamiento aunque no se llegue a resolver completamente

los problemas. Si el código no funciona, pero se explica detalladamente la intención se

valorará positivamente.

El objetivo principal de este módulo es adquirir conocimientos de criptografía y por ello se

considera fundamental usar cualquier herramienta que pueda ayudar a su resolución,

demostrando que no sólo se obtiene el dato sino que se tiene un conocimiento profundo

del mismo. Si durante la misma no se indica claramente la necesidad de resolverlo usando

programación, el alumno será libre de usar cualquier herramienta, siempre y cuando aporte

las evidencias oportunas.

Ejercicio 1

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.

* Consigna 1

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?

* Desarrollo 1

Con los datos conocidos implementamos el XOR

clave desarrollo= B1EF2ACFE2BAEEFF

clave properties = ?¿ **Respuesta = 20553975c31055ed**

clave final = 91BA13BA21AABB12

# Generamos el XOR de datos HEX

def xor\_hex(hex1, hex2):

bytes1 = bytes.fromhex(hex1)

bytes2 = bytes.fromhex(hex2)

# Realizar la operación XOR entre los bytes

xor\_result = bytes(a ^ b for a, b in zip(bytes1, bytes2))

# Convertir el resultado XOR a HEX

return xor\_result.hex()

# Claves consigna 1

clave\_desarrollo = ("B1EF2ACFE2BAEEFF")

clave\_final\_c1 = ("91BA13BA21AABB12")

# Calcular el valor XOR

clave\_properties\_c1 = xor\_hex(clave\_desarrollo, clave\_final\_c1)

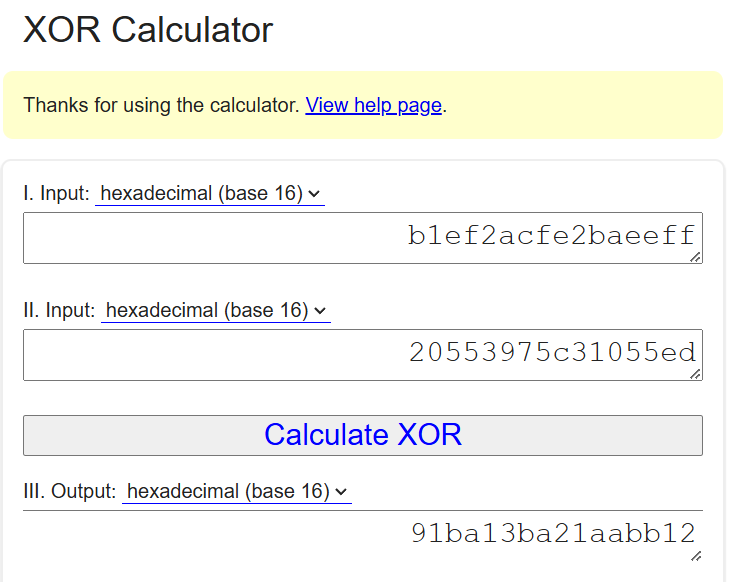
# Consigna 1

print("Clave asignada por Key Manager: ", clave\_properties\_c1)

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 1.py"

Clave en Memoria: 20553975c31055ed

Validamos respuesta en [“xor.pw/”](http://xor.pw/)



* Consiga 2

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?

* Desarrollo 2

clave fija = B1EF2ACFE2BAEEFF

clave properties = B98A15BA31AEBB3F

clave final = ?¿ **Respuesta = 08653f75d31455c0**

Reemplazamos valores en python, ya que utilizamos la misma función:

# Generamos el XOR de datos HEX

def xor\_hex(hex1, hex2):

bytes1 = bytes.fromhex(hex1)

bytes2 = bytes.fromhex(hex2)

# Realizar la operación XOR entre los bytes

xor\_result = bytes(a ^ b for a, b in zip(bytes1, bytes2))

# Convertir el resultado XOR a HEX

return xor\_result.hex()

# Claves consigna

clave\_desarrollo = ("B1EF2ACFE2BAEEFF")

clave\_properties\_c2 = ("B98A15BA31AEBB3F")

# Calcular el valor XOR

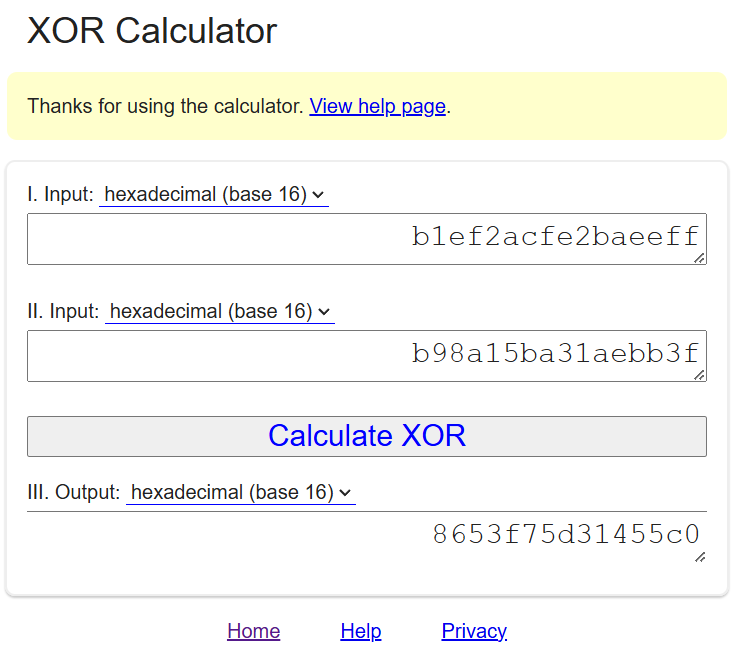
clave\_final\_c2 = xor\_hex(clave\_desarrollo, clave\_properties\_c2)

# Consigna 2

print("Clave en Memoria: ", clave\_final\_c2)

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 1.py"

Clave en Memoria: 08653f75d31455c0

Validamos respuesta en [“xor.pw/”](http://xor.pw/)

Herramientas utilizadas:

* Python
* Web http://xor.pw/

Ejercicio 2

2. Dada la clave con etiqueta “cifrado-sim-aes-256” que contiene el keystore. El iv estará

compuesto por el hexadecimal correspondiente a ceros binarios (“00”). Se requiere obtener

el dato en claro correspondiente al siguiente dato cifrado:

TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4US

t3aB/i50nvvJbBiG+le1ZhpR84oI=

* Consigna:

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?

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

Respuesta:

* La clave KeyStore: a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72
* Mensaje claro sin padding: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.
* Mensaje HEX con padding: 4573746f20657320756e206369667261646f20656e20626c6f7175652074c3ad7069636f2e2052656375657264612c2076617320706f7220656c206275656e2063616d696e6f2e20c3816e696d6f2e**01**
* Mensaje claro padding x923: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.
* La cantidad de padding agregado en el mensaje es: 1
* Desarrollo

Para ejecutar el algoritmo necesitamos obtener las clave del KeyStore, importamos desde py:

import jks

import os

#Obtenemos el path

path = os.path.dirname(r"C:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\KeyStorePracticas")

keystore = path + "\\KeyStorePracticas"

ks = jks.KeyStore.load(keystore, "123456")

for alias, sk in ks.secret\_keys.items():

if sk.alias == "cifrado-sim-aes-256":

key = sk.key

print("La clave es: ",key.hex())

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 2.py"

La clave es: a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

Con los datos conocidos ejecutamos la función en py:

* msg\_codificado = TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nvvJbBiG+le1ZhpR84oI=
* clave = a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72
* iv = 00000000000000000000000000000000

#Desciframos AES/CBC/PKCS7

from base64 import b64encode, b64decode

from Crypto.Cipher import AES

from Crypto.Util.Padding import pad, unpad

msg\_codificado = b64decode('TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nvvJbBiG+le1ZhpR84oI=')

clave = bytes.fromhex('a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72')

iv = bytes.fromhex('00000000000000000000000000000000')

cipher = AES.new(clave, AES.MODE\_CBC, iv)

msg\_dec\_bytes = cipher.decrypt(msg\_codificado)

msg\_dec\_claro = unpad(msg\_dec\_bytes, AES.block\_size, style= 'pkcs7')

msd\_dec\_padx923 = unpad(msg\_dec\_bytes, AES.block\_size, style= 'x923')

# Contamos padding

padding = len(msg\_dec\_bytes) - len(msg\_dec\_claro)

print("Mensaje claro con padding: ",msg\_dec\_bytes.decode('utf-8'))

print("Mensaje claro sin padding: ",msg\_dec\_claro.decode('utf-8'))

# Pintamos el mensaje cifrado en Hex para identificar el último par de digitos y conocer la cantidad de padding

print("Mensaje cifrado HEX con padding: ",msg\_dec\_bytes.hex())

# Simplemente validamos la observación anterior

print("La cantidad de padding agregado en el mensaje es: ", padding)

#Pintamos el mensaje descifrado con padding x923

print("Mensaje claro padding x923: ",msd\_dec\_padx923.decode('utf-8'))

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 2.py"

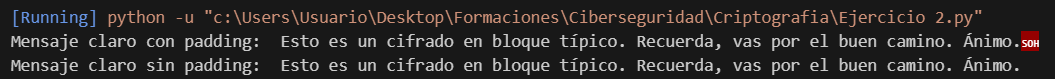
Mensaje claro con padding: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.

Mensaje claro sin padding: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.

Mensaje cifrado HEX con padding: 4573746f20657320756e206369667261646f20656e20626c6f7175652074c3ad7069636Af2e2052656375657264612c2076617320706f7220656c206275656e2063616d696e6f2e20c3816e696d6f2e01

La cantidad de padding agregado en el mensaje es: 1

Mensaje claro padding x923: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.



Herramientas utilizadas:

* Python

Ejercicio 3

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-chacha20-256**”. El nonce “9Yccn/f5nJJhAt2S”. El

El algoritmo que se debe usar es un Chacha20.

* Consigna

¿Cómo podríamos mejorar de forma sencilla el sistema, de tal forma, que no sólo

garantizamos 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.

* Desarrollo

Importamos la clave desde KeyStore

import jks

import os

#Obtenemos el path

path = os.path.dirname(r"C:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\KeyStorePracticas")

keystore = path + "\KeyStorePracticas"

ks = jks.KeyStore.load(keystore, "123456")

for alias, sk in ks.secret\_keys.items():

if sk.alias == "cifrado-sim-chacha20-256":

key = sk.key

print("La clave es: ",key.hex())

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 3.py"

La clave es: af9df30474898787a45605ccb9b936d33b780d03cabc81719d52383480dc3120

Con los datos conocidos ejecutamos la función en py:

mensaje claro = “KeepCoding te enseña a codificar y a cifrar”

clave = “af9df30474898787a45605ccb9b936d33b780d03cabc81719d52383480dc3120”

nonce = “9Yccn/f5nJJhAt2S”

Ciframos en Chacha 20:

# Ciframos en Chacha20

from Crypto.Cipher import ChaCha20

from base64 import b64decode, b64encode

msj\_claro = bytes('KeepCoding te enseña a codificar y a cifrar', 'utf-8')

clave = bytes.fromhex('af9df30474898787a45605ccb9b936d33b780d03cabc81719d52383480dc3120')

nonce = b64decode('9Yccn/f5nJJhAt2S')

cipher = ChaCha20.new(key=clave, nonce=nonce)

msj\_cifrado = cipher.encrypt(msj\_claro)

print('Mensaje cifrado en Hex: ', msj\_cifrado.hex())

print('Mensaje cifrado en base64: ', b64encode(msj\_cifrado).decode('utf-8'))

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 3.py"

Mensaje cifrado en Hex: 69ac4ee7c4c552537a00a19bcaf7f0aaed7c9c8f769956a09bce6fadef6c3535f2211c9467067cf5c4a842ab

Mensaje cifrado en base64: aaxO58TFUlN6AKGbyvfwqu18nI92mVagm85vre9sNTXyIRyUZwZ89cSoQqs=

Propuesta de mejora: cambiar al algoritmo de cifrado Chacha20poly1035, sumando una variable para garantizar la confidencialidad e integridad del dato. Sumamos “datos asociados”.

mensaje claro = “KeepCoding te enseña a codificar y a cifrar”

clave = “af9df30474898787a45605ccb9b936d33b780d03cabc81719d52383480dc3120”

nonce = “9Yccn/f5nJJhAt2S”

datos asociados = “Validamos con este texto”

# Propuesta de cifrado en Chacha20poly1035

from Crypto.Cipher import ChaCha20\_Poly1305

from base64 import b64decode, b64encode

msj\_claro = bytes('KeepCoding te enseÃ±a a codificar y a cifrar', 'utf-8')

clave = bytes.fromhex('af9df30474898787a45605ccb9b936d33b780d03cabc81719d52383480dc3120')

nonce = b64decode('9Yccn/f5nJJhAt2S')

datos\_asociados = bytes('Validamos con este texto', 'utf-8')

# Cifrado

cipher = ChaCha20\_Poly1305.new(key=clave, nonce=nonce)

cipher.update(datos\_asociados)

msj\_cifrado\_poly, tag = cipher.encrypt\_and\_digest(msj\_claro)

print('Mensaje cifrado en Chacha20poly1035 en Hex: ', msj\_cifrado\_poly.hex())

print('Mensaje cifrado en Chacha20poly1035 en base64: ', b64encode(msj\_cifrado\_poly).decode('utf-8'))

print('tag base64: ', b64encode(tag).decode('utf-8'))

# Descifrando

decipher = ChaCha20\_Poly1305.new(key=clave, nonce=nonce)

decipher.update(datos\_asociados)

msj\_descifrado\_poly = decipher.decrypt\_and\_verify(msj\_cifrado\_poly, tag)

print('Mensaje descifado en Chacha20poly1035: ', msj\_descifrado\_poly.decode('utf-8'))

**[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 3.py"**

**Mensaje cifrado en Chacha20poly1035 en Hex: 4ec95921ca8b757e2336605c7dbab8f4d40b5b4d220e66aa978f740d3e59b0cd70e3217242991cc140bb1e1a**

**Mensaje cifrado en Chacha20poly1035 en base64: TslZIcqLdX4jNmBcfbq49NQLW00iDmaql490DT5ZsM1w4yFyQpkcwUC7Hho=**

**tag base64: TM7SxeivgLmQLlAovLB2kw==**

**Mensaje descifado en Chacha20poly1035: KeepCoding te enseña a codificar y a cifrar**

Herramientas utilizadas:

* Python

Ejercicio 4

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

¿Qué algoritmo de firma hemos realizado?

¿Cuál es el body del jwt?

Un hacker está enviando a nuestro sistema el siguiente jwt:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODlHRI

¿Qué está intentando realizar?

¿Qué ocurre si intentamos validarlo con pyjwt?

* Desarrollo

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

*eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE*

*¿Qué algoritmo de firma hemos realizado?*

El algoritmo de firma que hemos utilizado es {"typ":"JWT","alg":"HS256"}, validamos esta información descifrando el header en Cyber Chef ([enlace](https://gchq.github.io/CyberChef/#recipe=From_Base64('A-Za-z0-9%2B/%3D',true,false)&input=ZXlKMGVYQWlPaUpLVjFRaUxDSmhiR2NpT2lKSVV6STFOaUo5&ienc=65001))

*¿Cuál es el body del jwt?*

El body del JWT es:

{

"usuario": "Don Pepito de los palotes",

"rol": "isNormal",

"iat": 1667933533

}

Lo validamos en Cyber Chef calculando el JWT Decode ([enlace](https://gchq.github.io/CyberChef/#recipe=JWT_Decode()&input=ZXlKMGVYQWlPaUpLVjFRaUxDSmhiR2NpT2lKSVV6STFOaUo5LmV5SjFjM1ZoY21sdklqb2lSRzl1SUZCbGNHbDBieUJrWlNCc2IzTWdjR0ZzYjNSbGN5SXNJbkp2YkNJNkltbHpUbTl5YldGc0lpd2lhV0YwSWpveE5qWTNPVE16TlRNemZRLmdmaHcwZER4cDZvaXhNTFhYUlA5N1c0VERUcnYweTdCNVlqRDBVOGl4ckU&ienc=65001))

Verificamos el mensaje aplicando la clave en Cyber Chef ([enlace](https://gchq.github.io/CyberChef/#recipe=JWT_Verify('Con%20KeepCoding%20aprendemos')&input=ZXlKMGVYQWlPaUpLVjFRaUxDSmhiR2NpT2lKSVV6STFOaUo5LmV5SjFjM1ZoY21sdklqb2lSRzl1SUZCbGNHbDBieUJrWlNCc2IzTWdjR0ZzYjNSbGN5SXNJbkp2YkNJNkltbHpUbTl5YldGc0lpd2lhV0YwSWpveE5qWTNPVE16TlRNemZRLmdmaHcwZER4cDZvaXhNTFhYUlA5N1c0VERUcnYweTdCNVlqRDBVOGl4ckU&ienc=65001))

*Un hacker está enviando a nuestro sistema el siguiente jwt:*

*eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODlHRI*

*¿Qué está intentando realizar?*

Revisando el Body del mensaje identificamos que el hacker está intentando insertar un usuario admin para vulnerar el acceso con privilegios:

{

"usuario": "Don Pepito de los palotes",

"rol": "isAdmin",

"iat": 1667933533

}

Validamos esta información en Cyber Chef ([enlace](https://gchq.github.io/CyberChef/#recipe=JWT_Decode()&input=ZXlKMGVYQWlPaUpLVjFRaUxDSmhiR2NpT2lKSVV6STFOaUo5LmV5SjFjM1ZoY21sdklqb2lSRzl1SUZCbGNHbDBieUJrWlNCc2IzTWdjR0ZzYjNSbGN5SXNJbkp2YkNJNkltbHpRV1J0YVc0aUxDSnBZWFFpT2pFMk5qYzVNek0xTXpOOS5rcmdCa3pDQlE1V1o4Sm5aSHVSdm1uQVpkZzRaTWVSTnYyQ0lBT0RsSFJJ&ienc=65001))

*¿Qué ocurre si intentamos validarlo con pyjwt?*

Reemplazamos los valores en la función para revisar el comportamiento de pyjwt, por supuesto entendiendo que el mensaje no dispone de contraseña (o contraseña inválida):

import jwt

# Reemplazamos los valores para revisar el comportamiento de pyjwt

encoded\_jwt = jwt.encode({"usuario": "Don Pepito de los palotes", "rol": "isAdmin", "iat": 1667933533}, "", algorithm="HS256")

print(encoded\_jwt)

decode\_jwt = jwt.decode(encoded\_jwt,"Con KeepCoding aprendemos", algorithms="HS256")

print(decode\_jwt)

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 4.py"

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.kZXp\_c\_ULqmTu5DMLzvFX0ExvjPdS2D6OqjMbOSwoqo

Traceback (most recent call last):

File "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 4.py", line 7, in <module>

decode\_jwt = jwt.decode(encoded\_jwt,"Con KeepCoding aprendemos", algorithms="HS256")

File "C:\Users\Usuario\AppData\Local\Programs\Python\Python310\lib\site-packages\jwt\api\_jwt.py", line 222, in decode

decoded = self.decode\_complete(

File "C:\Users\Usuario\AppData\Local\Programs\Python\Python310\lib\site-packages\jwt\api\_jwt.py", line 156, in decode\_complete

decoded = api\_jws.decode\_complete(

File "C:\Users\Usuario\AppData\Local\Programs\Python\Python310\lib\site-packages\jwt\api\_jws.py", line 220, in decode\_complete

self.\_verify\_signature(signing\_input, header, signature, key, algorithms)

File "C:\Users\Usuario\AppData\Local\Programs\Python\Python310\lib\site-packages\jwt\api\_jws.py", line 328, in \_verify\_signature

raise InvalidSignatureError("Signature verification failed")

jwt.exceptions.InvalidSignatureError: Signature verification failed

La validación es “Signature verification failed”, evita el acceso sin autorización.

Herramientas utilzadas:

* Python
* Web https://gchq.github.io/CyberChef/

Ejercicio 5

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?

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

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f

6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

¿Qué hash hemos realizado?

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?

* Desarrollo

Para generar el sha3, se utilizó el algoritmo SHA3\_256

Para generar el sha 2, se utilizó el algoritmo SHA2\_512

import hashlib

# Identificamos el tipo de algoritmo

m\_256 = hashlib.sha3\_256()

m\_256.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía", "utf8"))

print("sha3\_256: " + m\_256.digest().hex())

m\_512 = hashlib.sha512()

m\_512.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía", "utf8"))

print("sha256\_512: " + m\_512.digest().hex())

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 5.py"

sha3\_256: bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

sha256\_512: 4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

Texto 1: “En KeepCoding aprendemos cómo protegernos con criptografía”

Texto 2: “En KeepCoding aprendemos cómo protegernos con criptografía.”

*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?*

# Ciframos los texto con el mismo algoritmo sumando "solo" en el texto 2

m1 = hashlib.sha3\_256()

m1.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía", "utf8"))

print("Texto cifrado 1:" + m1.digest().hex())

m2 = hashlib.sha3\_256()

m2.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía.", "utf8"))

print("Texto cifrado 1:" + m2.digest().hex())

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 5.py"

Texto cifrado 1: bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

Texto cifrado 1: 302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

Cuando calculamos el hash en un texto modificando solo un carácter del mismo, el hash resultante es totalmente diferente, dejando en evidencia su “**propiedad de no predictibilidad”**.

Herramientas utilizadas:

* Python

Ejercicio 6

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.

* Desarrollo

Importamos la clave desde KeyStore:

path = os.path.dirname(r"C:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\KeyStorePracticas")

keystore = path + "\\KeyStorePracticas"

ks = jks.KeyStore.load(keystore, "123456")

for alias, sk in ks.secret\_keys.items():

if sk.alias == "hmac-sha256":

key = sk.key

print("La clave es: ",key.hex())

La clave es: a212a51c997e14b4df08d55967641b0677ca31e049e672a4b06861aa4d5826eb

Datos conocidos:

clave = a212a51c997e14b4df08d55967641b0677ca31e049e672a4b06861aa4d5826eb

mensaje a calcular = “Siempre existe más de una forma de hacerlo, y más de una solución válida.”

from Crypto.Hash import HMAC, SHA256

# Generamos el HMAC con los siguientes datos

def getHMAC(key\_bytes,data\_bytes):

hmac256 = HMAC.new(key\_bytes, msg=data\_bytes, digestmod=SHA256)

return hmac256.hexdigest()

def validateHMAC(key\_bytes,data\_bytes,hmac):

hmac256 = HMAC.new(key\_bytes,msg=data\_bytes,digestmod=SHA256)

result = "KO"

try:

hmac256.hexverify(hmac)

result = "OK"

except ValueError:

result = "KO"

print("result: " + result)

return result

clave = bytes.fromhex('a212a51c997e14b4df08d55967641b0677ca31e049e672a4b06861aa4d5826eb')

msj\_cod\_bytes = bytes('Siempre existe más de una forma de hacerlo, y más de una solución válida.', 'utf-8')

hmac256 = getHMAC(clave, msj\_cod\_bytes)

print(hmac256)

print(validateHMAC(clave, msj\_cod\_bytes,hmac256))

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 6.py"

857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550

result: OK

OK

Hemos calculado el HMAC con la función ***def getHMAC(key\_bytes,data\_bytes):***  y hemos verificado el cálculo con ***validateHMAC(key\_bytes,data\_bytes,hmac):***.

Herramientas utilizadas:

* Python

Ejercicio 7

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?

* Desarrollo

El analista de seguridad no fue asertivo con la propuesta ya que el SHA1 es un algoritmo vulnerable y propenso a ataques de colisión en criptografía, actualmente disponemos de algoritmos más fuertes como lo son SHA-2 y SHA-3.

Considero correcta la implementación de SHA2 ya que es un algoritmo confiable, es por ello que pasa a ser el más utilizado para este tipo de tareas.

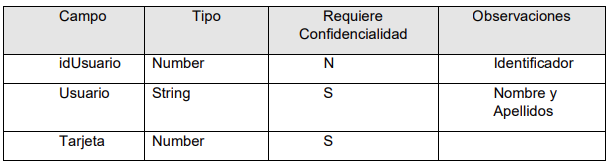
A pesar de que no tiene un gran despliegue actualmente, puede mejorarse el mecanismo de almacenamiento utilizando SHA3 por sus características basadas en el algoritmo Kaccak y diseño o funciones esponja. Es una gran opción para mejorar la seguridad en almacenamientos de información sensible.

Ejercicio 8

8. Tenemos la siguiente API REST, muy simple.

Request:

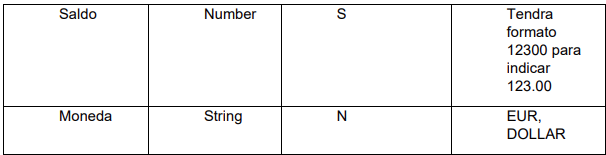
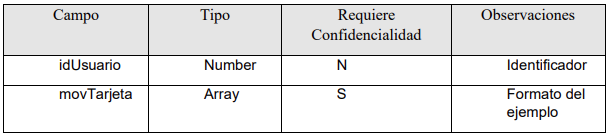
Post /movimientos



Petición de ejemplo que se desea enviar:

{"idUsuario":1,"usuario":"José Manuel Barrio Barrio","tarjeta":4231212345676891}

Response:



{

"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?

* Desarrollo

Asumiendo que el sistema E2E no utiliza TLS, la primera recomendación es implementar el protocolo TLS 1.3 para garantizar la seguridad de conexión de los datos.

En todo caso avanzaremos con las siguientes sugerencias, definimos los parámetros del protocolo:

Utilizaremos cifrados en bloque, asimétricos o híbridos.

La información se compartirá en el formato Json.

ECDHE\_RSA = se recomienda el uso de curva elíptica diffie hellman con firma RSA para garantizar la confidencialidad de los datos en el caso de que un tercero esté observando la red o mensaje (MitM) para el intercambio de claves.

AES-128\_GCM = se sugiere el cifrado con AES 128 bytes, algoritmo suficientemente seguro para el tipo de transacción (consulta temporal de movimientos de cuenta) con el MODO GCM para autenticar el cifrado del mensaje.

SHA256 = HMAC con sha256 para autenticar el mensaje HASH.

Ciframos implementando los sugerencias (py), datos sensibles:

import json

from base64 import b64encode, b64decode

from Crypto.Cipher import AES

from Crypto.Random import get\_random\_bytes

# Creamos los aleatorios para cifrar los datos sensibles

clave = get\_random\_bytes(16)

nonce = get\_random\_bytes(16)

datos\_asociados = bytes('','utf-8')

datos\_confidenciales = b'{"idUsuario":1,"tarjeta":4231212345676891}'

cipher = AES.new(key=clave, mode=AES.MODE\_GCM, nonce=nonce)

cipher.update(datos\_asociados)

texto\_cifrado, tag = cipher.encrypt\_and\_digest(datos\_confidenciales)

texto\_cifrado\_b64 = b64encode(texto\_cifrado).decode('utf-8')

nonce\_b64 = b64encode(nonce).decode('utf-8')

tag\_b64 = b64encode(tag).decode('utf-8')

datos\_asociados\_b64 = b64encode(datos\_asociados).decode('utf-8')

texto\_cifrado\_json = json.dumps({'texto cifrado':texto\_cifrado\_b64,'nonce':nonce\_b64,'tag':tag\_b64,'datos asociados':datos\_asociados\_b64})

print(texto\_cifrado\_json)

# Validamos descifrando el mensaje

decipher = AES.new(key=clave, mode=AES.MODE\_GCM, nonce=nonce)

decipher.update(datos\_asociados)

texto\_descifrado = decipher.decrypt\_and\_verify(texto\_cifrado,tag)

print('=====Validación=====')

print('Validamos descifrando:',texto\_descifrado)

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 10.py"

{"texto cifrado": "GFNaYn63mMrzX3NTPDnc/I6xEzZNR2PmGtMnetzeb1PCMZswU4EagBhL", "nonce": "f0B8eyR7/Yg7ZaV2UlMY3Q==", "tag": "bvuE05W3bvkDKMt5bjhTqg==", "datos asociados": ""}

=====Validación=====

Validamos descifrando: b'{"idUsuario":1,"tarjeta":4231212345676891}'

Ciframos implementando los sugerencias (py), firma RSA:

# Generamos la firma RSA para el mensaje confidencial

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

from Crypto.Hash import SHA256

keyPair = RSA.generate(2048)

pubKey = keyPair.publickey()

pubKeyPEM = pubKey.exportKey()

print(pubKeyPEM.decode('utf8'))

print("--------------------------------------------------")

#Generamos la clave privada.

privKeyPEM = keyPair.exportKey()

print(privKeyPEM.decode('utf8'))

print("--------------------------------------------------")

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 10.py"

-----BEGIN PUBLIC KEY-----

MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAtQeI+Gd50xNT5bI1qSKS

C0x7X4ixjzIf3iMPOo2C0Kl7gmmvahmvgqMdir/1OWcG4CSP9hZnwSGLtt/YqWFx

zfylinQeoKWMFQBgYqU0RMN6fNA3kLAkD8oqiBj0jWy8Ev+ztmNQfWuHr0SpFWDu

sOhBfisXmVJZroM+pIPV80ZAhXHa8gXqrSkmf6JaH7dC8CCbrXgVAFbj/JhYaNPo

xwIV3wCkuyeHpps9i7E3CqY/llKAdvyOr09YR4PDObW0aRe+juaTQ92BxBtS3a+6

Qf2VZCyvQTSRaH6X5jw1OINRROjYY2rs1jWXE8Cvsh8aJBMBxskOvqnAuKBRK2r1

7wIDAQAB

-----END PUBLIC KEY-----

--------------------------------------------------

-----BEGIN RSA PRIVATE KEY-----

MIIEogIBAAKCAQEAtQeI+Gd50xNT5bI1qSKSC0x7X4ixjzIf3iMPOo2C0Kl7gmmv

ahmvgqMdir/1OWcG4CSP9hZnwSGLtt/YqWFxzfylinQeoKWMFQBgYqU0RMN6fNA3

kLAkD8oqiBj0jWy8Ev+ztmNQfWuHr0SpFWDusOhBfisXmVJZroM+pIPV80ZAhXHa

8gXqrSkmf6JaH7dC8CCbrXgVAFbj/JhYaNPoxwIV3wCkuyeHpps9i7E3CqY/llKA

dvyOr09YR4PDObW0aRe+juaTQ92BxBtS3a+6Qf2VZCyvQTSRaH6X5jw1OINRROjY

Y2rs1jWXE8Cvsh8aJBMBxskOvqnAuKBRK2r17wIDAQABAoIBABhQ6coMBCYhC3zo

AiM03I8d99huQQNuBS/OX8ZKqc6jZU6wKXrD9npCtL4laKgMObUz1A6HFeqwOdcX

nD+sMiAsFoRRcqe7dOSBR3EYpbuy0iofwcbykfV4/1uGlprEh0K6DAassmwWXIRj

YZ7YhVdH6zwqR6I1mIxDXs+0UbG6fQC1/o291jhCq3uuMimoNO+YtO8TMMXCOjei

9v7TrQNuYnsaehyX9kncyqr4F33mcfun79G+V/OcqmchI7x20QxrOdErF6xOHDV4

1UysEEZcoK9pfOEWp1gTab7wtJeRl+O8iVDwoIuuom0aD3DRxcF5pNjXk+bUnHX7

VSM8zKECgYEAx0OdzKNwEg8rV4f2oyNdu/BR2uM/2/oIsE85926jpJWWqJdUqwN6

vnQUtD6+KATyORTTppQLOrlNrV8iwpsM6315tV4JVj8JYwO380taRU56CBEpuyi+

QIsMe2gle6TPTFgKF9I45SLRD+kWsJ2VMGLtIPUmyWvy+uUuPyyo6H8CgYEA6JLK

OARNFi3FZhdZa43NVvaR/MpVE1ZOqviu4r4lVil4DPMTjkLavMF8QVx05FTkgb+H

ETU1RSGF1x3muIKq1a4aYR7Ou0boK8q9YroMQvLZDH6KbvItE82ENHHrd0sclMeE

ipuqjoLhXps48VuE7Gw4Ru/shaNtidCLInetupECgYBxPicmzld5m+9YzmdHbWju

zvPl/rIjFOTtNglOai1L8sttQpuWmOKf7B58FRBCdBUohYKHL6l2FStk9SUvDaj5

CBUsEqYhwq9V+x4GgXNq4d+OSBq8R7pfueY9GAH9U7uPupT51roI1pGThjcj5lqL

5hy3VztUyymKlsaClqHjhwKBgFHzb4gfk94t0LHxplOsyt9K0RRHY4UF04CRUmaZ

Y+Shhijh+lEVo6gBcbxCTPosI7zIdVqN6jBq6JNteOApQvM6fxcb2PLKwIK11NgW

AL4SfoLVkAgaE+I3d89XGQaXFAIk78I7qfS/j5Rbk8wQWJ1KYOY2IpzQgoeeUXia

vT/RAoGAMKb1n+yY+e7iqaXQDdX5yOi8QhzeoX2Tv9ZQbvhlIvNNjoacxRaWWgpy

qbYgUaP9xxwgN3a1BB3TpiG1p2iMpJbe5zw9v6XlWIC4d2toE0A5VKWUHSjuOgA4

ZKngsy8Zoiw9MHRu2X8KfHmq6CB/0VdKR21jEjUBYFMWDLWgcdQ=

-----END RSA PRIVATE KEY-----

--------------------------------------------------

Ya con esta información ciframos con la firma privada generada:

# Generamos la firma RSA para el mensaje confidencial

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

from Crypto.Hash import SHA256

# Par de claves

keyPair = RSA.generate(2048)

# Publica

pubKey = keyPair.publickey()

pubKeyPEM = pubKey.exportKey()

# print(pubKeyPEM.decode('utf8'))

# print("--------------------------------------------------")

# Privada

privKeyPEM = keyPair.exportKey()

# print(privKeyPEM.decode('utf8'))

# print("--------------------------------------------------")

mensaje = bytes(texto\_cifrado\_json,'utf-8')

cipher\_RSA = PKCS1\_OAEP.new(pubKey,SHA256)

text\_cifrado = cipher\_RSA.encrypt(mensaje)

print("Cifrado RSA:", text\_cifrado.hex())

print("--------------------------------------------------")

# Validamos descifrando el mensaje

decryptor = PKCS1\_OAEP.new(keyPair,SHA256)

descifrado\_RSA= decryptor.decrypt(text\_cifrado)

print("Descifrado RSA: ", descifrado\_RSA)

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 10.py"

Cifrado RSA: 

--------------------------------------------------

Descifrado RSA: b'{"texto cifrado": "hRoiHafohApvj49K1pqHqnvYm4abXxrUa7kwdpyAh9QZTEMfRYBlXI60", "nonce": "UIJ7AbJ13Y0lTDEdks+g2w==", "tag": "yXSSlSj2EQ+6S8n0nfl1Xw==", "datos asociados": ""}'

Conclusión:

Identificamos la necesidad de conectar datos del usuario con el servidor de forma segura, ya que corresponde a información confidencial.

Recomendamos protocolos estandarizados con nivel alto de protección de información, garantizando integridad y confidencialidad de los mismos.

Implementamos los algoritmos recomendados en py.

En un estadío más avanzado de la API REST se puede implementar JWT formado de mensaje.

Herramientas utilizadas:

* Python

Ejercicio 9

9. Se requiere calcular el KCV de las siguiente clave AES:

A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72

Para lo cual, vamos a requerir el KCV(SHA-256) así como el KCV(AES). El KCV(SHA-256) se

corresponderá con los 3 primeros bytes del SHA-256. Mientras que el KCV(AES) se

corresponderá con cifrar un texto del tamaño del bloque AES (16 bytes) compuesto con

ceros binarios (00), así como un iv igualmente compuesto de ceros binarios. Obviamente, la clave usada será la que queremos obtener su valor de control.

* Desarrollo

Con los datos conocidos ejecutamos el algoritmo en py:

Texto plano = 00000000000000000000000000000000

iv = 00000000000000000000000000000000

clave = A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72

import hashlib

from base64 import b64encode, b64decode

from Crypto.Cipher import AES

from Crypto.Util.Padding import pad, unpad

textoPlano\_bytes = bytes.fromhex('00000000000000000000000000000000')

iv\_bytes = bytes.fromhex('00000000000000000000000000000000')

clave = bytes.fromhex('A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72')

# Calculamos el KCV AES

cipher = AES.new(clave, AES.MODE\_CBC,iv\_bytes)

texto\_cifrado\_bytes = cipher.encrypt(pad(textoPlano\_bytes, AES.block\_size, style='pkcs7'))

print("KCV AES:", texto\_cifrado\_bytes.hex()[0:6])

# Calculamoso clave KVCsha256

m = hashlib.sha256()

m.update(bytes.fromhex("A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"))

print("KCV SHA256: " + m.digest().hex()[0:6])

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 9.py"

KCV AES: 5244db

KCV SHA256: db7df2

Obtenemos los siguientes KCV:

**KCV AES: 5244db**

**KCV SHA256: db7df2**

Herramientas utilizadas:

* Python

Ejercicio 10

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.

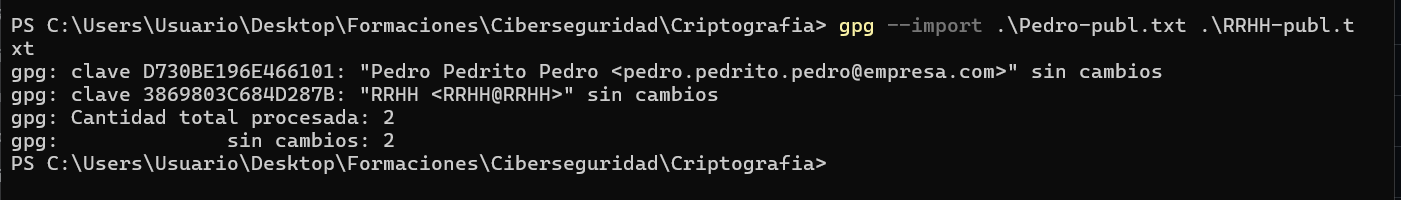
**“Viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario. Saludos.”**

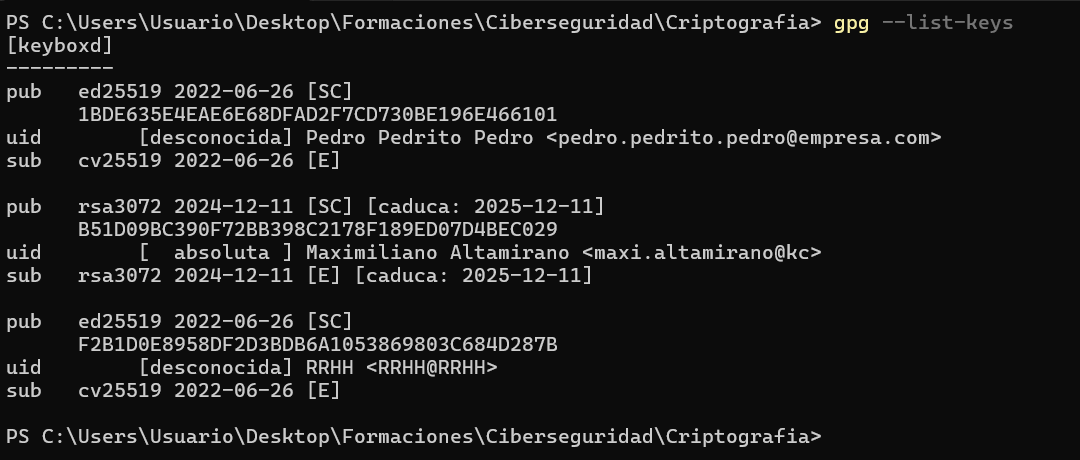
Por último, cifra el siguiente mensaje tanto con la clave pública de RRHH como la de Pedro y adjunta el fichero con la práctica.

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

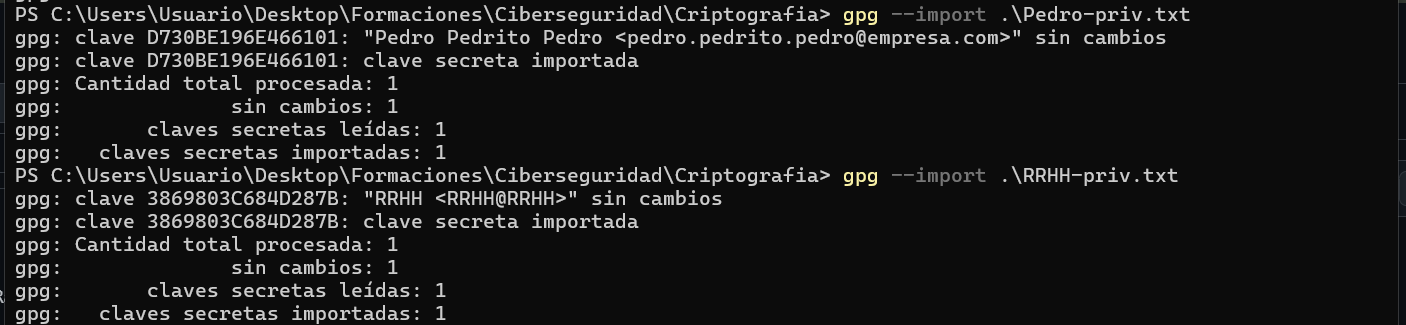
* Desarrollo

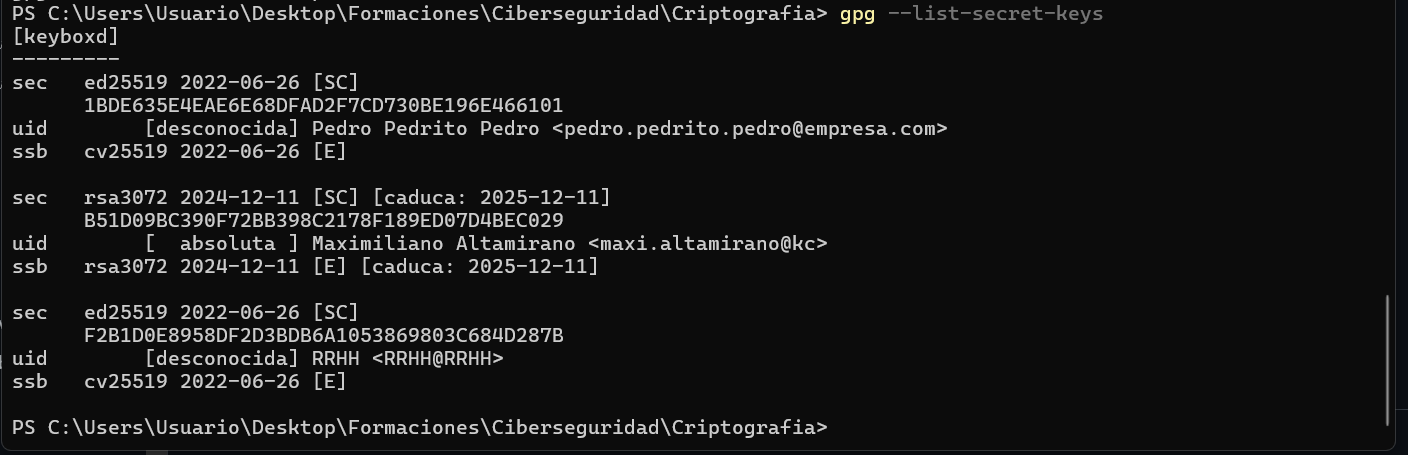
Importamos las claves públicas.



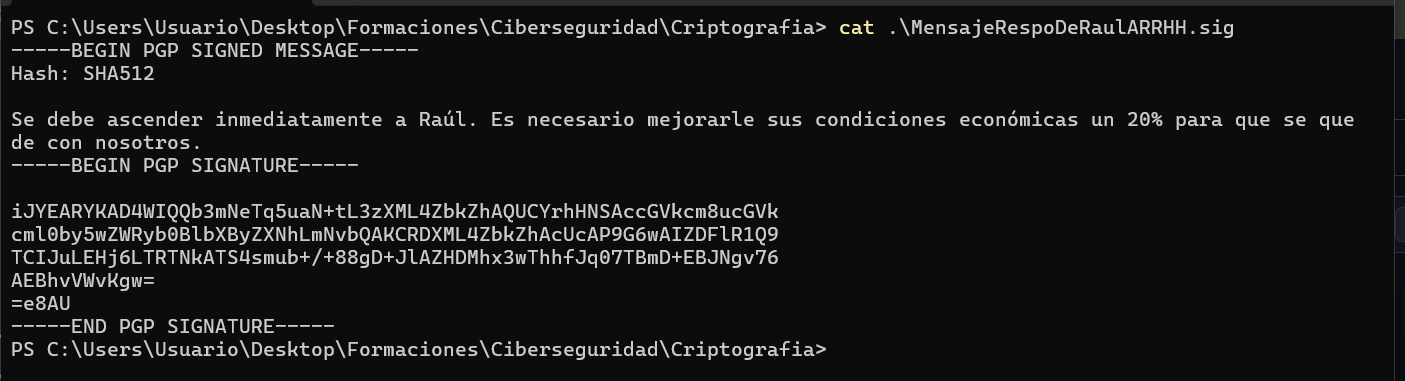


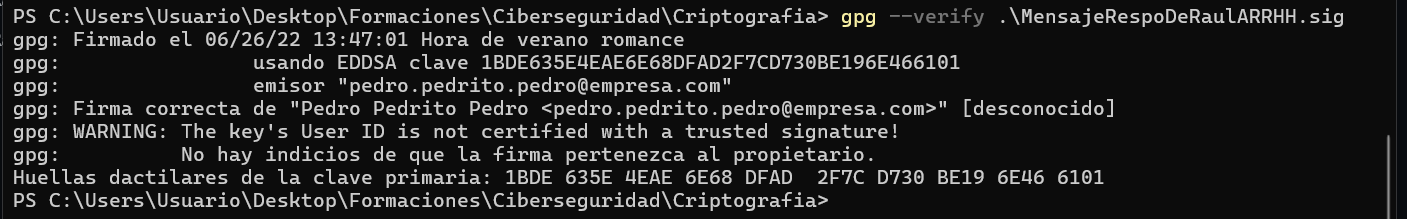
Importamos las claves privadas:





* *Comprobamos la firma del fichero* ***“MensajeRespoDeRaulARRHH.sig”***





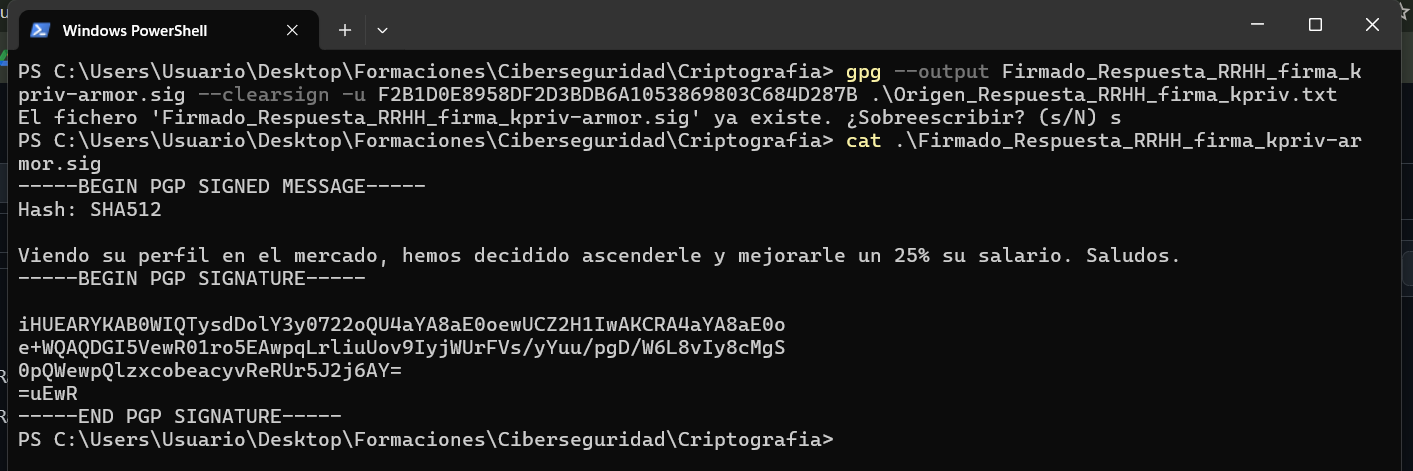
Confirmamos que al mensaje lo ha firmado Pedro “**gpg: Firma correcta de "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>" “**

* *Debemos enviar el siguiente mensaje:*

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

*firmado por RRHH.*  
  
Creamos el fichero “**Origen\_Respuesta\_RRHH\_firma\_kpriv.txt**” y lo firmamos con la clave privada “**RRHH-priv.txt**”

Lanzamos el código “**gpg --output Firmado\_Respuesta\_RRHH\_firma\_kpriv-armor.sig --clearsign -u F2B1D0E8958DF2D3BDB6A1053869803C684D287B .\Origen\_Respuesta\_RRHH\_firma\_kpriv.txt**”



Hemos generado el nuevo fichero “**Firmado\_Respuesta\_RRHH\_firma\_kpriv-armor.sig**” con el mensaje y la firma correspondiente.

* *Por último, cifra el siguiente mensaje tanto con la clave pública de RRHH como la de Pedro:*

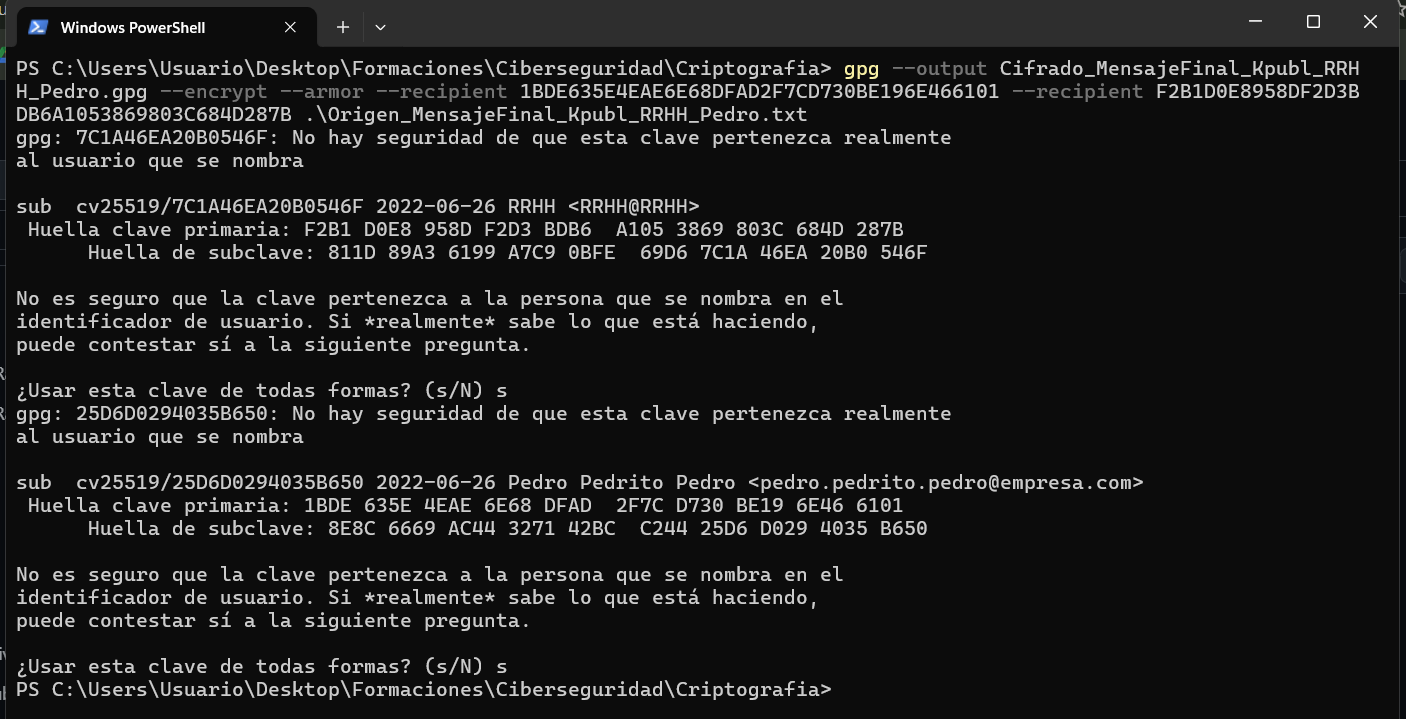
*“Estamos todos de acuerdo, el ascenso será el mes que viene, agosto, si no hay sorpresas.”*

Creamos el fichero “**Origen\_MensajeFinal\_Kpubl\_RRHH\_Pablo.txt**” con el mensaje y formamos con las claves públicas:

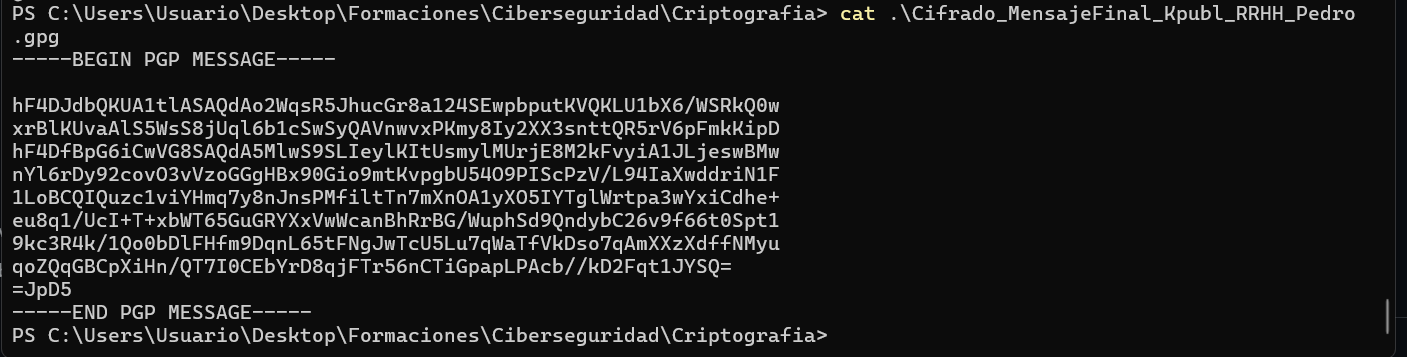
1BDE635E4EAE6E68DFAD2F7CD730BE196E466101 (Pedro)

F2B1D0E8958DF2D3BDB6A1053869803C684D287B (RRHH)

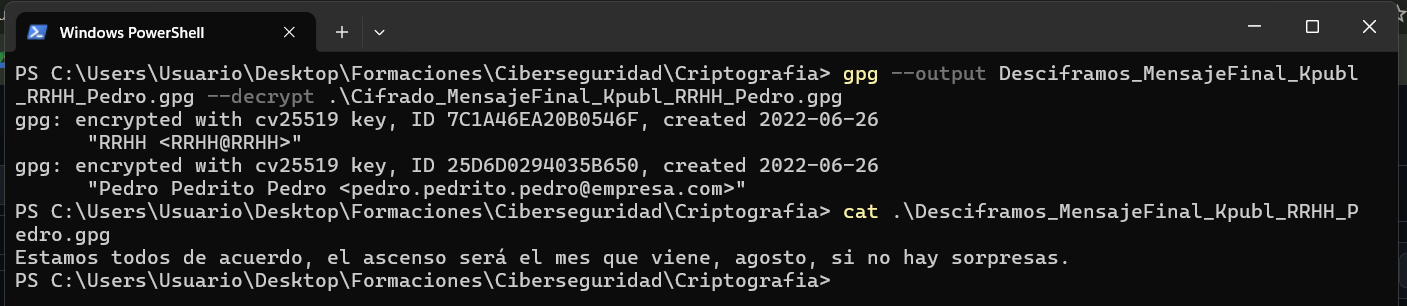
Ejecutamos el comando “**gpg --output Cifrado\_MensajeFinal\_Kpubl\_RRHH\_Pedro.gpg --encrypt --armor --recipient 1BDE635E4EAE6E68DFAD2F7CD730BE196E466101 --recipient F2B1D0E8958DF2D3BDB6A1053869803C684D287B .\Origen\_MensajeFinal\_Kpubl\_RRHH\_Pedro.txt**”



Revisamos el cifrado



Desciframos el mensaje para validar que hicimos el proceso correcto:



Herramientas utilizadas:

* GPG

Ejercicio 11

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

El texto cifrado es el siguiente:



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

Si has recuperado la clave, vuelve a cifrar con el mismo algoritmo. ¿Por qué son diferentes

los textos cifrados?

* Desarrollo

Importamos la clave privada del fichero y definimos el descifrador:

Descifrado: **e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72**

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

from Crypto.Hash import SHA256,SHA512

import os

my\_path = os.path.abspath(os.getcwd())

fichero\_fpriv = my\_path + "\\clave-rsa-oaep-priv.pem"

fpriv=open(fichero\_fpriv)

keypr= RSA.import\_key(fpriv.read())

msj\_cifrado = bytes.fromhex("")

descifrador = PKCS1\_OAEP.new(keypr, SHA256)

try:

# Intentar descifrar el mensaje

decrypted = descifrador.decrypt(msj\_cifrado)

print("Descifrado: ", decrypted.hex())

except Exception as e:

print("Error al descifrar:", e)

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 11.py"

Descifrado: e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

[Done] exited with code=0 in 0.757 seconds

Importamos la clave pública del fichero para cifrar y definimos el cifrador:

Cifrado:

****

**# Ciframos nuevamente el mensaje con clave pública**

**my\_path = os.path.abspath(os.getcwd())**

**fichero\_fpub = my\_path + "\\clave-rsa-oaep-publ.pem"**

**fpub=open(fichero\_fpub)**

**keypub= RSA.import\_key(fpub.read())**

**mensaje = bytes.fromhex("e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72")**

**cipher = PKCS1\_OAEP.new(keypub,SHA256)**

**text\_cifrado = cipher.encrypt(mensaje)**

**print("Cifrado:", text\_cifrado.hex())**

**[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 11.py"**

**Cifrado: **

El algoritmo de RSA OAEP garantiza un esquema aleatorio de relleno para garantizar un nivel de seguridad más eficiente, esto genera que cada mensaje cifrado sea único, a pesar de estar cifrando el mismo mensaje cada vez.

Herramientas utilizadas:

* Python

Ejercicio 12

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?

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.

* Desarrollo

Datos conocidos:

mensaje claro = He descubierto el error y no volveré a hacerlo mal

clave = E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74

nonce = 9Yccn/f5nJJhAt2S

Texto cifrado en Hex: **5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d**

Texto cifrado en base64: **Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq92dhLum64MHCV9QePv8FiVt**

Tag: **6120e37aa4c3ecfd9261640dcc46410d**

**from base64 import b64encode, b64decode**

**from Crypto.Cipher import AES**

**from Crypto.Util.Padding import pad, unpad**

**# Datos conocidos para cifrar**

**msj\_claro = bytes('He descubierto el error y no volveré a hacerlo mal', 'utf-8')**

**clave = bytes.fromhex('E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74')**

**nonce = b64decode('9Yccn/f5nJJhAt2S')**

**datos\_asociados = bytes('','utf-8')**

**cipher = AES.new(key=clave, mode=AES.MODE\_GCM, nonce=nonce)**

**cipher.update(datos\_asociados)**

**msj\_cifrado, tag = cipher.encrypt\_and\_digest(msj\_claro)**

**print('Texto cifrado en Hex:', msj\_cifrado.hex())**

**print('Texto cifrado en base64:', b64encode(msj\_cifrado).decode('utf-8'))**

**print('Tag:', tag.hex())**

**[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 12.py"**

**Texto cifrado en Hex: 5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d**

**Texto cifrado en base64: Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq92dhLum64MHCV9QePv8FiVt**

**Tag: 6120e37aa4c3ecfd9261640dcc46410d**

# Validamos descifrando la información

cifrado\_en\_hex = bytes.fromhex('5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d')

decipher = AES.new(key=clave, mode=AES.MODE\_GCM, nonce=nonce)

decipher.update(datos\_asociados)

msj\_claro\_descifrado = decipher.decrypt\_and\_verify(cifrado\_en\_hex,tag)

print('Mensaje validado:', msj\_claro\_descifrado.decode('utf-8'))

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 12.py"

Mensaje validado: He descubierto el error y no volveré a hacerlo mal

El error principal en este cifrado es utilizar un nonce “estático”, ya que para garantizar la seguridad del cifrado es necesario utilizar un nonce aleatorio. Aunque no es imprescindible, pudo sumarse un dato asociado específico.

Herramientas utilizadas:

* Python

Ejercicio 13

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.

* Desarrollo

La firma del mensaje en HEX es:



from Crypto.PublicKey import RSA

from Crypto.Signature.pkcs1\_15 import PKCS115\_SigScheme

from Crypto.Hash import SHA256

import binascii

import os

# Importamos la clave privada para firmar

my\_path = os.path.abspath(os.getcwd())

path\_file\_priv = my\_path + "/clave-rsa-oaep-priv.pem"

keypriv = RSA.importKey(open(path\_file\_priv).read())

mensaje\_bytes = bytes("El equipo está preparado para seguir con el proceso, necesitaremos más recursos.","utf-8")

hash = SHA256.new(mensaje\_bytes)

signer=PKCS115\_SigScheme(keypriv)

firma = signer.sign(hash)

print("Firma: ", firma.hex())

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 13.py"

Firma: 

Validamos la firma para garantizar que el proceso es correcto:

#Importamos la clave publica para validar la firma

my\_path = os.path.abspath(os.getcwd())

fichero\_pub = my\_path + "/clave-rsa-oaep-publ.pem"

f=open(fichero\_pub,'r')

keypub= RSA.import\_key(f.read())

firma= bytes.fromhex("")

verifier=PKCS115\_SigScheme(keypub)

try:

verifier.verify(hash,firma)

print("Firma válida")

except:

print("La firma no es válida")

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 13.py"

Firma válida

Calculamos la firma con la curva elíptica ed25519.

# Calculamos la firma con la curva elíptica ed25519

import ed25519

publickey = open("ed25519-publ","rb").read()

privatekey = open("ed25519-priv","rb").read()

signedKey = ed25519.SigningKey(privatekey)

msg = bytes('El equipo está preparado para seguir con el proceso, necesitaremos más recursos.','utf8')

signature = signedKey.sign(msg, encoding='hex')

print("Firma en Hex:", signature)

try:

verifyKey = ed25519.VerifyingKey(publickey.hex(),encoding="hex")

verifyKey.verify(signature, msg, encoding='hex')

print("La firma es válida")

except:

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 13.py"

Firma en Hex: b'bf32592dc235a26e31e231063a1984bb75ffd9dc5550cf30105911ca4560dab52abb40e4f7e2d3af828abac1467d95d668a80395e0a71c51798bd54469b7360d'

La firma es válida

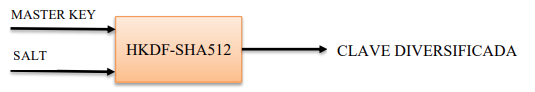
Herramientas utilizadas:

* Python

Ejercicio 14

14. Necesitamos generar una nueva clave AES, usando para ello una HKDF (HMAC-based Extractand-Expand key derivation function) con un hash SHA-512. La clave maestra requerida se encuentra en el keystore con la etiqueta “cifrado-sim-aes-256”. La clave obtenida dependerá de un identificador de dispositivo, en este caso tendrá el valor en hexadecimal:

e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3



¿Qué clave se ha obtenido?

* Desarrollo

Datos conocidos:

Clave maestra = **cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72**

Salt **= e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3**

Clave diversificada = **e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a**

**import jks**

**import os**

**# Importamos la clave desde KeyStore**

**path = os.path.dirname(r"C:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\KeyStorePracticas")**

**keystore = path + "\\KeyStorePracticas"**

**ks = jks.KeyStore.load(keystore, "123456")**

**for alias, sk in ks.secret\_keys.items():**

**if sk.alias == "cifrado-sim-aes-256":**

**key = sk.key**

**print("La clave es: ",key.hex())**

**# Generamos la clave con las condiciones informada**

**from Crypto.Protocol.KDF import HKDF**

**from Crypto.Hash import SHA512**

**salt = bytes.fromhex('e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3')**

**master\_secret = key**

**keyAES = HKDF(master\_secret,32,salt, SHA512, 1)**

**print("Clave AES nueva: ", keyAES.hex())**

**[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 14.py"**

**La clave es: a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72**

**Clave AES nueva: e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a**

Herramientas utilizadas:

* Python

Ejercicio 15

15. Nos envían un bloque TR31:

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37018E111B

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

A1A10101010101010101010101010102

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

¿Para qué algoritmo se ha definido la clave?

¿Para qué modo de uso se ha generado?

¿Es exportable?

¿Para qué se puede usar la clave?

¿Qué valor tiene la clave?

* Desarrollo

Desciframos el TR31 obteniendo la siguiente información:

Clave= c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1

Key Version ID: D

Algoritmo: A

Modo de uso: B

Uso de la clave: D0

Exportabilidad: S

from psec import tr31

header, key = tr31.unwrap( kbpk=bytes.fromhex("A1A10101010101010101010101010102"), key\_block="D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37018E111B")

print("Clave=", key.hex())

print("Key Version ID: " + header.version\_id )

print("Algoritmo: " + header.algorithm)

print("Modo de uso: " + header.mode\_of\_use)

print("Uso de la clave: " + header.key\_usage)

print("Exportabilidad: " + header.exportability)

[Running] python -u "c:\Users\Usuario\Desktop\Formaciones\Ciberseguridad\Criptografia\Ejercicio 15.py"

Clave= c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1

Key Version ID: D

Algoritmo: A

Modo de uso: B

Uso de la clave: D0

Exportabilidad: S

Con el apoyo de la siguiente guía <https://github.com/knovichikhin/psec/blob/master/psec/tr31.py> revisamos la siguiente información:

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

Key Version ID: D

Bloque de claves protegido mediante el método de vinculación de derivación de claves AES.

**¿Para qué algoritmo se ha definido la clave?**

Algoritmo: A  
La clave se definió para utilizar con el algoritmo AES

**¿Para qué modo de uso se ha generado?**

Modo de uso: B

Se ha generado tanto para cifrar como descifrar/envolver y desenvolver

**¿Es exportable?**

Exportabilidad: S

Sensible, exportable bajo clave no confiable

**¿Para qué se puede usar la clave?**

Uso de la clave: D0

Clave simétrica para cifrado de datos

**¿Qué valor tiene la clave?**

Clave= c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1

El valor contenido en este bloque de claves es un componente de una clave. Las normas locales dictarán el uso adecuado de un componente.

Herramientas utilizadas:

* Python
* Web https://github.com/knovichikhin/psec/blob/master/psec/tr31.py

Resumen

Resumen de la práctica:

* Repasamos los procedimientos más utilizados en el rol de analista de seguridad
* Abordamos las consignas identificando la solución más eficiente
* Aplicamos los conocimientos adquiridos en el módulo de Criptografía
* Marcamos la solución fundamentando el procedimiento utilizado

Junto con esta documentación se adjuntan los siguientes ficheros:

* Ejercicio 1.py
* Ejercicio 2.py
* Ejercicio 3.py
* Ejercicio 4.py
* Ejercicio 5.py
* Ejercicio 6.py
* Ejercicio 9.py
* Ejercicio 10.py
* Ejercicio 11.py
* Ejercicio 12.py
* Ejercicio 13.py
* Ejercicio 14.py
* Ejercicio 15.py
* Cifrado\_MensajeFinal\_Kpubl\_RRHH\_Pedro.gpg
* Desciframos\_MensajeFinal\_Kpubl\_RRHH\_Pedro.gpg
* Firmado\_Respuesta\_RRHH\_firma\_kpriv-armor.sig
* MensajeRespoDeRaulARRHH.sig
* MensajeRespoDeRaulARRHH.txt
* Origen\_MensajeFinal\_Kpubl\_RRHH\_Pedro.txt
* Origen\_Respuesta\_RRHH\_firma\_kpriv.txt
* Pedro-priv.txt
* Pedro-publ.txt
* clave-rsa-oaep-priv.pem
* clave-rsa-oaep-publ.pem
* ed25519-priv
* ed25519-publ
* KeyStorePracticas
* RRHH-priv.txt
* RRHH-publ.txt

Herramientas utilizadas:

* Python
* Web <https://gchq.github.io/CyberChef/>
* Web <http://xor.pw/>
* Web <https://github.com/knovichikhin/psec/blob/master/psec/tr31.py>