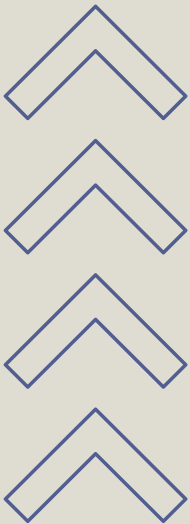




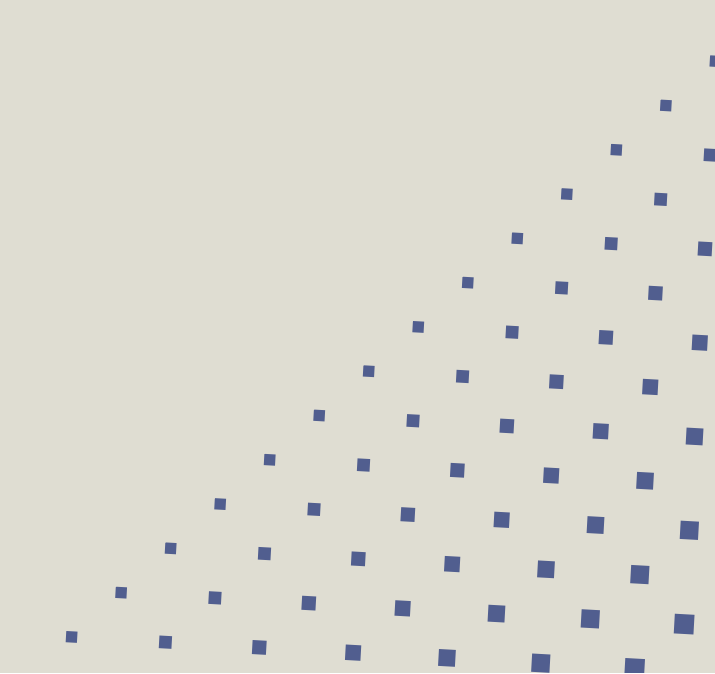
31-Julio-2022

CRIPTOGRAFIA

TRABAJO FINAL



Informe redactado por:
Diego Puchol Candel





INFORME DE PRÁCTICA FINAL

En este informe resolveré las preguntas de la práctica final con los conocimientos adquiridos en las clases.

En cada una de las preguntas explicaré como lo resolví y las herramientas usadas para resolverlas.

»»»» 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.

1.1. 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?

Usaré una función usado por el profesor en la clase que me ayudará a sacar el XOR. La función es la siguiente:

```
def xor_data(binary_data_1, binary_data_2):  
    return bytes([b1^b2 for b1,b2 in zip (binary_data_1, binary_data_2)])
```

Lo hice de la siguiente manera:

```
k2 =bytes.fromhex("A1EF2ABFE1AAEEFF")  
k =bytes.fromhex("B1AA12BA21AABB12")  
print(xor_data(k2,k).hex())
```

Y me da como resultado: 10453805c00055ed

1.2. 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?

Usaremos la misma formula para resolver esta segunda parte pero cambiando datos. La formula es la siguiente:

```
k2 =bytes.fromhex("A1EF2ABFE1AAEEFF")  
k1 =bytes.fromhex("B98A15BA31AEBB22")  
print(xor_data(k2,k).hex())
```

Y da como resultado: 18653f05d00455dd



EJERCICIO 2

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:

```
zcFJxRlfzaBj+gVWFRAah1N2wv+G2P01ifrKejICaGpQkPnZMiexn3WXLGYX5WnNIosy
KfkNKG9GGSgG1awaZg==
```

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

2.1. ¿qué obtenemos?

Para obtener el descifrado utilizaremos la siguientes funciones:

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

Como el profesor dijo para trabajar hay que pasar todo a bytes; tanto la clave, el IV, como el mensaje. El IV se puede saber gracias al enunciado principal, es un AES 256. Como el IV debe ocupar según el algoritmo utilizado y conociendo que 1 byte = a 2 caracteres en hex, podemos saber que el IV son 32 "0".

```
texto_cifrado_bytes=b64decode('zcFJxRlfzaBj+gVWFRAah1N2wv+G2P01ifrKejICaGpQkPnZMiexn
3WXLGYX5WnNIosyKfkNKG9GGSgG1awaZg==')
clave = bytes.fromhex('e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72')
iv_bytes = bytes.fromhex('00000000000000000000000000000000')
```

Cuanto todo esté en bytes, desciframos con las herramientas: el IV y la clave, con AES/CBC que lo sabemos gracias al enunciado. Después desciframos con las herramientas el mensaje. Y por ultimo le pido que lo muestre a hex y a texto:

```
try: cipher = AES.new(clave, AES.MODE_CBC, iv_bytes) mensaje_des_bytes =
unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size, style="pkcs7")
mensaje_des_bytes.hex()) print("El texto en claro es: ", mensaje_des_bytes.decode("utf-8"))
```

Con todo esto podemos saber que el texto y el hex es:

- En hex:

```
4573746f20657320756e206369667261646f20656e20626c6f7175652074c3ad7069636f2e205265637565
72646120656c2070616464696e672e
```

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

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

Para obtener la clave y poder resolver la primera pregunta hice lo siguiente:

```
import jks
import os

path=os.path.dirname(file)
print(path)

keystore=path + "/KeyStorePracticas/"
ks = jks.KeyStore.load(keystore, "123456")

for alias, sk in ks.secret_keys.items():
    print("Secret key: %s" % sk.alias)
    print(" Algorithm: %s" % sk.algorithm)
    print(" Key size: %d bits" % sk.key_size)
    print(" Key: %s" % "".join("{:02x}".format(b) for b in bytearray(sk.key)))
```

Primero se importan las funciones necesarias, después usaremos el "path" para escribir donde queremos que busque la clave. Después con el "path" definido le indicaremos la dirección que queremos que tome y también por supuesto tenemos que escribir la contraseña para que pueda ingresar y darnos la clave.

Con el "for" a continuación ya pasamos a pedir como tal todas las claves que esten en el "KeyStore" con la información de cada una de ellas.

Al poner en marcha este código podremos ver nuestra clave con la etiqueta que nos indica en el enunciado.

La clave es:

e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

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

Me sale error porque tiene que descifrarse con el mismo tipo de padding que se cifró en su origen. Lo averigüé probando y me salió error.

2.3. ¿Cuánto padding se ha añadido en el cifrado?

Para averiguar el padding añadido en el cifrado hice lo siguiente usando el código que hice en un principio:

try:

```
cipher = AES.new(clave, AES.MODE_CBC, iv_bytes)
#mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_bytes),
AES.block_size, style="pkcs7")
mensaje_des_bytes = cipher.decrypt(texto_cifrado_bytes)
print("En hex: ", mensaje_des_bytes.hex())
```

#mensaje_des_bytes = unpad(cipher.decrypt(texto_cifrado_bytes), AES.block_size, style="pkcs7") esta línea la oculté para comprobar el padding y añadí esta misma pero sin la última parte que es lo que hace referencia al padding y con esto conseguí descubrir cuánto padding se había añadido. El padding es el siguiente:

En hex:

4573746f20657320756e206369667261646f20656e20626c6f7175652074c3ad7069636f2e20526563756572

646120656c2070616464696e672e**060606060606**

Lo señalado en rojo vendría a ser el padding. 12 caracteres en hexadecimal o sea 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”. ¿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.

Utilizaremos las siguientes funciones:

```

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

```

Para cifrar el siguiente texto 'Este curso es de lo mejor que podemos encontrar en el mercado' haré lo siguiente:

```

try:
    textoPlano_bytes = bytes('Este curso es de lo mejor que podemos encontrar en el
mercado', 'UTF-8')
    clave_bytes=bytes.fromhex('979DF30474898787A45605CCB9B936D33B780D03CABC81719
D52383480DC3120')
    nonce_mensaje_bytes = b64decode('9Yccn/f5nJJhAt2S')
    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":
b64encode(texto_cifrado_bytes).decode(), "tag": b64encode(tag_bytes).decode()}
    json_mensaje = json.dumps(mensaje_enviado)
    print("Mensaje: ", json_mensaje)

except (ValueError, KeyError) as error:
    print("Problemas al cifrar....")
    print("El motivo del error es: ", error)

```

Paso el mensaje a bytes, consigo la clave en KeyStore con la etiqueta señalada en el enunciado y también el nonce.

La clave conseguida en en la KeyStore se pasa a bytes y el nonce. Después, creo unos datos asociados que servirán para que la persona que vaya a descifrar el mensaje pueda comprobar la integridad de este. Después, usare el cipher dándole lo que necesita para cifrar (nonce y key).

A este le añadiré los datos asociados mencionados antes. A continuación ciframos el texto y creamos al mismo tiempo el tag. En la siguiente línea creamos un mensaje que será json que contiene; el tag, datos asociados, el nonce y el mensaje cifrado.

Ya al fina estoy pidiendo que si hay algún error cifrando se muestre en pantalla.

Si lanzo el código recibiré lo siguiente:

```

Mensaje: {"nonce": "9Yccn/f5nJJhAt2S", "datos asociados":
"RGF0b3MgY2lmcFkb3Mgc8OzbG8gYXV0ZW50aWNhZG9z", "texto cifrado":
"EWGrU8t3Mr7pqLjrFz1hGgMkqxhZBXsbqK8nA1gqmkcMXkqzA5OdUsPYPO8asRk1nZo3
OzieQ56ggemkUQ=", "tag": "hDNXqvnIV89m6ClF4KLzWA=="}

```

>>>> EJERCICIO 4

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRmVsaXB1IFJvZ
HJcdT AwZWRndWV6Iiwicm9sIjoiaXNOY3JtYWwifQ.-
KiAA8cjkamjwRUiNVHgGeJU8k2wiErdxQP_iFXumM8

4.1. ¿Qué algoritmo de firma hemos realizado?

He usado jwt.io para obtener el tipo de algoritmo que es.
Es un HMAC Sha256.

```
HEADER: ALGORITHM & TOKEN TYPE

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

4.2. ¿Cuál es el body del jwt?

```
PAYLOAD: DATA

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

El body es el payload. Realizado con la misma pagina del apartado anterior.

Un hacker está enviando a nuestro sistema el siguiente jwt:

**eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRmVsaXB1IFJvZ
HJcdT AwZWRndWV6Iiwicm9sIjoiaXNBZG1pbiJ9.-
KiAA8cjkamjwRUiNVHgGeJU8k2wiErdxQP_iFXumM8**

4.3. ¿Qué está intentando realizar?

Esta intentando coger licencias de administrador, lo averigüé ingresando el jwt del hacker en "jwt.io".

```
PAYLOAD: DATA

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

4.4. ¿Qué ocurre si intentamos validarlo con pyjwt?

Intenté validarlo con 'pyjwt', con estos comandos:

```
import jwt

decode_jwt =
jwt.decode("eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRmVsaXB1IFJvZ  
HJcdT AwZWRndWV6Iiwicm9sIjoiaXNBZG1pbiJ9.-  
KiAA8cjkamjwRUiNVHgGeJU8k2wiErdxQP_iFXumM8","KeepCoding", algorithms="HS256")

print(decode_jwt)
```

Y me da el siguiente error:

InvalidSignatureError: Signature verification failed

Y da error porque no tiene la firma.

>>>> EJERCICIO 5

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

bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

5.1. ¿Qué tipo de SHA3 hemos generado?

SHA3 256 este es el tipo de hash, me he basado en la longitud del mismo hash, ya que contiene 64 caracteres en hexadecimal lo que es igual a 256 bits.

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

**4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d6373
9506f646883 3d77c07cfd69c488823b8d858283f1d05877120e8c5351c833**

5.2. ¿Qué hash hemos realizado?

Sabiendo que el SHA2 produce dos tipos de hashes, de 256 bits y de 512 bits al ver que este contiene 64 bytes sé que es un **SHA2 512 bits**

5.3. 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?

Estos son los comandos que realicé para sacar el hash del texto ; “En KeepCoding aprendemos cómo protegernos con criptografía.”

```
import hashlib
```

```
s = hashlib.sha3_256()
```

```
s.update(bytes("En KeepCoding aprendemos cómo protegernos con criptografía.", "UTF-8"))
```

```
print(s.hexdigest())
```

Y da como resultado el siguiente hash:

302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

Como propiedad diría que es que solo cambiando un carácter del mensaje cambia todo el hash.

>>>> 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.

Lo que hice fue lo siguiente, programé con las siguientes funciones.

Calculé el hmac-256 usando la clave de KeyStore y saqué el hash del texto indicado.

Pasé todo a bytes (el mensaje y la clave), utilicé la función HMAC para generarlo dándole la información que ya pase a bytes y diciendo que lo haga en SHA256. Después con el 'print' podré ver el HMAC que me ha generado y la ultima parte del código me servirá para verificar el mensaje.

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

```
secret=bytes.fromhex('2712A51C997E14B4DF08D55967641B0677CA31E049E672A4B06861AA4D5826EB')
```

```
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()
```

```
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")
```



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.

7.1. ¿Por qué crees que es una mala opción?

Porque en 2017 recibieron ataques de colisión. O sea no es seguro.

7.2. 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?

Necesitamos asegurar la integridad, autenticación y confidencialidad, entonces propondría un MAC para proteger los datos importantes y delicados de nuestros clientes. Y cifraría los datos delicados e importantes dentro del MAC.

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

Buscando opciones en todo lo aprendido creo que haría falta la verificación de alguna forma como algún OTP que son claves de un solo uso o también podríamos hacer uso de un HMAC. O creo también que vendría bien usar el KEY DERIVATION FUNCTIONS que están diseñados para el uso de passwords deliberadamente lentos para dificultar la fuerza bruta.



EJERCICIO 8

8. Tenemos la siguiente API REST, muy simple.

Request:

Post /movimientos

Campo	Tipo	Requiere Confidencialidad	Observaciones
idUsuario	Number	N	Identificador
Usuario	String	S	Nombre y Apellidos
Tarjeta	Number	S	

Petición de ejemplo que se desea enviar:

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

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?

Creo que resolvería todo esto en un primer lugar asegurándome que toda la información que sea valiosa/delicada este cifrada. También haría uso de alguna herramienta que nos ayude a verificar (HMAC). Estaría meter todo esto en un JSON para que con la firma cualquier cambio podamos detectarlo.

>>>> 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.

En este ejercicio utilizaremos la herramienta dicha por el profesor GnuPG y esta herramienta la usaré en consola.

Después importaré los archivos facilitados (le añadí antes '.txt') por medio de la plataforma GitHub.

`gpg --import + (el archivo)`

Y verifiqué el fichero recuperando la clave como se me indicaba. Con el siguiente comando:

```
PS C:\Users\User\Desktop\ejercicio9> gpg --verify .\MensajeRespoDeRaulARRHH.txt.sig
gpg: Firmado el 26/06/2022 13:47:01 Hora de verano romance
gpg: usando EDDSA clave 1BDE635E4EAE6E68DFAD2F7CD730BE196E466101
gpg: emisor "pedro.pedrito.pedro@empresa.com"
gpg: Firma correcta de "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>" [desconocido]
gpg: WARNING: The key's User ID is not certified with a trusted signature!
gpg: No hay indicios de que la firma pertenezca al propietario.
Huellas dactilares de la clave primaria: 1BDE 635E 4EAE 6E68 DFAD 2F7C D730 BE19 6E46 6101
gpg: ADVERTENCIA: firma no separada; ¡el archivo .\MensajeRespoDeRaulARRHH.txt NO ha sido verificado!
PS C:\Users\User\Desktop\ejercicio9>
```

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.

Para este ejercicio he guardado como txt el mensaje a enviar con nombre "ejer9", así que he usado el siguiente comando :

`gpg --local-user F2B1D0E8958DF2D3BDB6A1053869803C684D287B --armor --output firma.sig --detach-sign ejer9.txt`

Después de esto me ha pedido la contraseña para confirmar (123456) y he podido firmar el mensaje y al mismo tiempo crearla .

Así que igual al resto de los archivos que adjuntaré. Con eso también pondré el archivo "firma.sig" y el archivo del texto a firmar "ejer9.txt"

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 esta pregunta he guardado de nuevo el mensaje en un .txt y gracias a la explicación en clase he usado el comando :

```
gpg --output cifrado.gpg --encrypt --recipient F2B1D0E8958DF2D3BDB6A1053869803C684D287B --recipient 1BDE635E4EAE6E68DFAD2F7CD730BE196E466101 --armor Paracifrar.txt
```

lo que me hace confirmar dos veces que si confío en las claves. He puesto dos como me pide el ejercicio, la de Pedro y la de RRHH

Y ya si le pido que me lo muestre me da lo siguiente

-----BEGIN PGP MESSAGE-----

```
hF4DfBpG6iCwVG8SAQdALfIzAtOPcfZM0TC2Tw/U0DBsuoOeDF4fWx/riZBYoTAw
t2eAojAepReuuNFh/feVV1Ldj/aE3+UK6gBV3G9ChFBacj3OzxIAO94dj6+wLFgz
hF4DJdbQKUA1tIASAQdAMWyjTK7/sMFNH02i9jfW7XH5ccofFlsZqjS9MbRvTysw
erlrZ7o8BzC6YxIeqcK6IvJQGpYz+uU0a7JksUCt+bTL+jXSuG5B/38HLLDrQdhv
1JEBBCIQN5b+TLv75wPYKUM15aqVY94Z4sLfMVnk0nVr60//cX7at8eW+cLZKJTs
PT0ymbtZEUelGBf8ABak2GovveDmkzXglXjJFNMTH2HT0lDxudJopHeN4wnZzlav
lY09CrxiMpi7xZBNd7hHDg3w7yGNJfzLCsHa3ze9MqkPo7poPsyCRGjTaMG7Np8t
YaUH
=B69u
-----END PGP MESSAGE-----
```

```
PS C:\Users\User\Desktop\ejercicio9> gpg --output textoyacifrado.gpg --encrypt --recipient F2B1D0E8958DF2D3BDB6A1053869803C684D287B --recipient 1BDE635E4EAE6E68DFAD2F7CD730BE196E466101 --armor textoparacifrar.txt
gpg: 25D6D0294035B650: No hay seguridad de que esta clave pertenezca realmente al usuario que se nombra

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

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

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

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

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

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

```
¿Usar esta clave de todas formas? (s/N) s
PS C:\Users\User\Desktop\ejercicio9> cat .\textoyacifrado.gpg
-----BEGIN PGP MESSAGE-----

hF4DfBpG6iCwVG8SAQdALfIzAtOPcfZM0TC2Tw/U0DBsuoOeDF4fWx/riZBYoTAw
t2eAojAepReuuNFh/feVV1Ldj/aE3+UK6gBV3G9ChFBacj3OzxIAO94dj6+wLFgz
hF4DJdbQKUA1tIASAQdAMWyjTK7/sMFNH02i9jfW7XH5ccofFlsZqjS9MbRvTysw
erlrZ7o8BzC6YxIeqcK6IvJQGpYz+uU0a7JksUCt+bTL+jXSuG5B/38HLLDrQdhv
1JEBBCIQN5b+TLv75wPYKUM15aqVY94Z4sLfMVnk0nVr60//cX7at8eW+cLZKJTs
PT0ymbtZEUelGBf8ABak2GovveDmkzXglXjJFNMTH2HT0lDxudJopHeN4wnZzlav
lY09CrxiMpi7xZBNd7hHDg3w7yGNJfzLCsHa3ze9MqkPo7poPsyCRGjTaMG7Np8t
YaUH
=B69u
-----END PGP MESSAGE-----
PS C:\Users\User\Desktop\ejercicio9> |
```


>>>> 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.

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

Aquí indico por medio de comentarios en el mismo código como voy paso a paso resolviendo el ejercicio en Python.

```
ejercicio 10.py 1 X
C:\Users\> User > Desktop > ejercicio 10 > ejercicio 10.py > ...
1 import os
2 from Crypto.PublicKey import RSA
3 from Crypto.Hash import SHA256
4 from Crypto.Cipher import PKCS1_OAEP
5
6 #Indico de donde quiero que lea las claves
7
8 my_path = os.path.dirname(file)
9 path_file_publ = my_path + "/clave-rsa-oaep-publ.pem"
10 path_file_priv = my_path + "/clave-rsa-oaep-priv.pem"
11
12 #Importo las claves gracias al RSA import
13
14 keypub = RSA.import_key(open(path_file_publ).read())
15 keypriv = RSA.import_key(open(path_file_priv).read())
16
17 #Aquí descifro usando de guía la muestra de como descifrar con RSA OAEP
18
19 decryptor = PKCS1_OAEP.new(keypriv,SHA256)
20 textocifrado=bytes.fromhex('7edee3ec0b808c440078d63ee65b17e85f0c1adbc0da1b7fa842f24fb06b332c156038062d9daa8ccfe83bace1dca475cfb7757f1f6446840044fe698a631fe882e1a6fc00a2de30025e9dcc76e74f9d9d721e9664a6319eaa59dc9011bfc624d2a63eb0e449ed4471ff06c9a303465d0a50ae0a8e5418a1d12e9392faaaf9d4046aa16e424ae1e26844bcf4abc4f8413961396f2ef9ffcd432928d428c2a23fb85b497d89190e3cfa496b6016cd32e816336cad7784989af89ff853a3acd796813eade65ca3a10bbf58c6215fdf26ce061d19b39670481d03b51bb0eccc926c9d6e9cb05ba56082a899f9aa72f94c158e56335c5594fcc7f8f301ac1e15a938')
21 decrypted = decryptor.decrypt(textocifrado)
22
23 #Pido que me lo muestre en hexadecimal
24
25 print('El texto descifrado en hexadecimal:', decrypted.hex())
26
27 #Lo vuelvo a cifrar como me pide el ejercicio
28
29 cipher = PKCS1_OAEP.new(keypub,SHA256)
30 encrypted = cipher.encrypt(decrypted)
31
32 #De nuevo pido que lo enseñe en hexadecimal
33
34 print ('El texto que descifré , vuelto a cifrar : ',encrypted.hex())
```

Por el oaep, que usa valores aleatorios con los que hace un cifrado propio , entonces cada vez que lo ejecutamos produce cifrados distintos.
Es un algoritmo recomendable de usar ya que es seguro.

»»»» 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?

Por lo que veo el nonce mostrado en el ejercicio es fijo y por lo tanto se repetirá con lo cual es un gran error. El nonce es único e irrepetible.