PRÁCTICA DE CRIPTOGRAFÍA

Gerard Díaz Hoyos

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 *A1EF2ABFE2BAEEFF*, mientras que en desarrollo sabemos que la clave final (en memoria) es *F1BA12BA21AABB12*. ¿Qué valor ha puesto el Key Manager en properties para forzar dicha clave final?

   La clave fija, recordemos es *A1EF2ABFE2BAEEFF*, 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?

   En este caso en concreto, he hecho toda la explicación pertinente en los mismos comentarios del código fuente. Se explica los datos que nos el enunciado y la operación XOR a realizar en ambos supuestos, sabiendo las propiedades que existen con esta misma operación.
   Dejo también el archivo Ejercicio01.py subido por si las capturas no te permiten ver el texto suficientemente grande.

Código Fuente > Intro > Ejercicio01.py > ...

```python
# sabemos que la clave fija en código es A1EF2ABFE2BAEEFF (codificación hexadecimal) --> K1
# sabemos también que la clave final F1BA12BA21AABB12 (hexadecimal) --> K
# por tanto, nos falta saber la clave de propiedades. --> K2

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

# primero pasamos las claves a binario para poder realizar las operaciones XOR necesarias:
K = bytes.fromhex("F1BA12BA21AABB12")
K1 = bytes.fromhex("A1EF2ABFE2BAEEFF")

# teniendo en cuenta las propiedades de XOR, sabemos que K2 = K ^ K1: (resultado después de ejecutar el código: 50553805c31055ed)

K2 = xor_data(K,K1).hex()
print("La clave de propiedades dinámica es = "+ K2)

# en el 2o apartado del ejercicio sabemos que:
# la clave fija sigue siendo la misma: A1EF2ABFE2BAEEFF; la llamaremos ahora K11
# La clave dinámica ahora es B98A15BA31AEBB3F; la llamaremos ahora K22
# Como la clave fija final es el resultado de una disociación en las 2 partes expuestas, deberemos hacer la operación XOR de ambos: la llamaremos KK

# volvemos a pasar las claves que tenemos a binario primeramente:
K11 = bytes.fromhex("A1EF2ABFE2BAEEFF")
K22 = bytes.fromhex("B98A15BA31AEBB3F")

KK = xor_data(K11,K22).hex()   # el resultado, tras la ejecución es: 18653f05d31455c0
print("La clave final será: " + KK)
```

Y estos serían los resultados tras ejecutar el código anterior:

```
(Criptologia) D:\Ciberseguridad\Criptografia\criptografia-main>C:/Users/Gerard/anaconda3/envs/Criptologia/python.exe "d:/Ciberseguridad/Criptografia/criptografia-main/Código Fuente/Intro/Ejercicio01.py"
La clave de propiedades dinámica es = 50553805c31055ed
La clave final será: 18653f05d31455c0
```

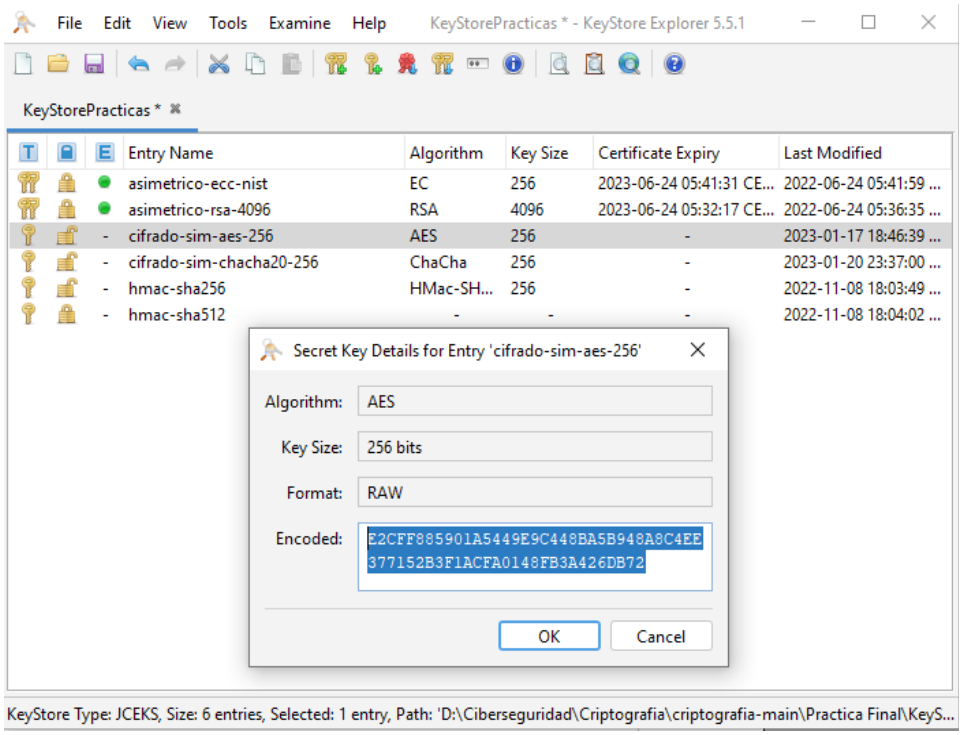
2) **Dada la clave con etiqueta "*cifrado-sim-aes-256*" que contiene el keystore. El iv estará compuesto por el hexadecimal correspondiente a *ceros binarios ("00")*. Se requiere obtener el dato en claro correspondiente al siguiente dato cifrado:**

zcFJxR1fzaBj+gVWFRAah1N2wv+G2P01ifrKejICaGpQkPnZMiexn3WXlGYX5WnNgpZs3h0N4jLXi2xlV02D1g==

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).
(observar archivo Ejercicio02.py)

Antes de nada, accederemos al software keystore para obtener la clave correspondiente a la etiqueta "cifrado-sim-aes-256:
E2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72



Aunque en este caso, vamos a aplicar código para que se rescate directamente la clave del keystore, sin tener que dejarla visible en código (cuestión que no se recomienda).

El IV en hexadecimal y compuesto por ceros binarios será:
`00000000000000000000000000000000`
(32 ceros en hexadecimal, que divido por 2, obtenemos un número compuesto por 16 bytes en hexadecimal)

Pasaremos todos los datos de entrada que tenemos a binario (teniendo en cuenta que el IV y la clave los tenemos en hexadecimal y el dato cifrado que nos da el propio enunciado en

base64), tal y como exigen prácticamente todos los algoritmos criptográficos y ejecutaremos las líneas de código correspondientes al descifrado de un AES/CBC/PKCS7 (observar código fuente del archivo Ejercicio02).

El texto obtenido, como resultado del descifrado es:
Esto es un cifrado en bloque típico. Recuerda el padding...

O en hexadecimal:

4573746f20657320756e206369667261646f20656e20626c6f7175652074c3ad7069636f2e20
52656375657264612065c2070616464696e672e2e2e

Si probamos de cambiar el padding a x923 veremos como obtenemos un error en la operación de descifrado:
"Problemas para descifrar...."
"El motivo del error es: ANSI X.923 padding is incorrect."
Esto es totalmente normal, puesto que la operación de cifrado del texto en claro se hico utilizado el tipo de padding PKCS7 y, al descifrarlo, deberíamos respetar este formato para poder realizar la operación inversa de descifrado.

Recordemos que el texto en claro ya lo habíamos sacado en hexadecimal:

4573746f20657320756e206369667261646f20656e20626c6f7175652074c3ad7069636f2e20
52656375657264612065c2070616464696e672e2e2e

Si ahora ejecutamos el código de descifrado sin especificar qué padding estamos usando, obtenemos los siguientes resultados (en utf-8 y hexadecimal):

El texto en claro es: Esto es un cifrado en bloque típico. Recuerda el padding...♦♦♦♦
4573746f20657320756e206369667261646f20656e20626c6f7175652074c3ad7069636f2e20
*52656375657264612065c2070616464696e672e2e2e**04040404***
Vemos que hay 4 bytes de padding; en utf-8 se nos representa como 4 rombos y en hexadecimal como 04040404 (formato de padding estándar o PKCS7).

Resultados obtenidos por consola:

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*". El algoritmo que se debe usar es un Chacha20. ¿Cómo podríamos mejorar de forma sencilla el sistema, de tal forma, que no sólo garanticemos la confidencialidad sino, además, la integridad del mismo? Se requiere obtener el dato cifrado, demuestra, tu propuesta por código, así como añadir los datos necesarios para evaluar tu propuesta de mejora.**
(Observar archivo *Ejercicio03.py)*

Ciframos los datos de entrada con el algoritmo ChaCha20 (observar en código fuente).

Para mejorar este cifrado, deberíamos implementar el **ChaCha20-Poly1305**, en vez del que hemos usado anteriormente: ChaCha20. De este modo, estamos añadiendo un TAG que nos permite autenticar el mensaje.
A parte, utilizaremos un nonce generado aleatoriamente, puesto que es uno de los requisitos necesarios para añadir seguridad a los cifrados que utilicen este parámetro de entrada.
Observar código fuente del archivo Ejercicio03.py para ver todo lo implementado y comentarios extras explicativos

4) **Tenemos el siguiente jwt, cuya clave es "Con KeepCoding aprendemos".**

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c3VhcmlvIjoiRG9uIE1pZ3VlbCBGZXJveiIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ.VeFJ0HsEawLsqiMUkZlJtDjeAjhygTiyqMzJr3o8yqQ

¿Qué algoritmo de firma hemos realizado? ¿Cuál es el body del jwt? Un hacker está enviando a nuestro sistema el siguiente jwt:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c3VhcmlvIjoiRG9uIE1pZ3VlbCBGZXJveiIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.-KiAA8cjkamjwRUiNVHgGeJU8k2wiErdxQP_iFXumM8

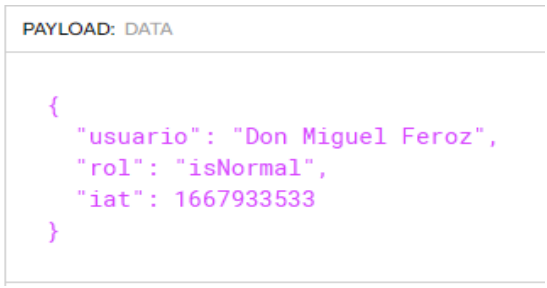
¿Qué está intentando realizar? ¿Qué ocurre si intentamos validarlo con pyjwt?
(Observar código del archivo Ejercicio04.py)

Siguiendo la estructura típica de un Jason Token Web: **XXX.YYY.ZZZ**
El XXX sería el Header o cabecera y correspondería a la primera parte del código del enunciado, hasta el primer punto:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
(utilizo cualquier página de Internet para descodificarlo)

Vemos que el algoritmo de firma que estamos utilizando es el **SHA256**.

Posteriormente, nos encontramos con el Body o Payload:

eyJ1c3VhcmlvIjoiRG9uIE1pZ3VlbCBGZXJveiIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ



Y por último, tenemos la firma:
VeFJ0HsEawLsqiMUkZlJtDjeAjhygTiyqMzJr3o8yqQ
(la firma no nos muestra datos legibles)

El supuesto hacker ha intentado cambiar el Payload o Body del JWT, cambiando los permisos de usuario para tener el perfil de admin:

de **{'usuario': 'Don Miguel Feroz', 'rol': 'isNormal', 'iat': 1667933533}**
a
{'usuario': 'Don Miguel Feroz', 'rol': 'isAdmin', 'iat': 1667933533}

En el caso de que hubiera interceptado nuestra contraseña, podría realizar el cambio y se seguiría verificando correctamente la firma.

```
(Criptologia) D:\Ciberseguridad\Criptografia\criptografia-main>C:/Users/Gerard/anaconda3/envs/Criptologia/python.exe "d:/Ciberseguridad/Criptografia/criptografia-main/Código Fuente/Hashing y Authentication/Ejercicio04.py"
b64pwd: 3476695663786337535O4a776b6a2b67384963355673683362 6a3376734c476568612b5178596235377453d
2432612431322472645 54d336e4274667a38464e6d55664a566578442e6532413371497477516232 4e5a44616e74326442305a3779434d3253582f6d
Password correcta
Codigo normal: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIE1pZ3VlbCBGZXJveiIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ.snCUE066pVyiDstmWwQAkvDizxB_XXhxcWaVZmUHZbU
Codigo modificado: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIE1pZ3VlbCBGZXJveiIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.MX_AeJhPo3xNK_Be-i4Ddp8FRpewOVQwkZ6oTlnJtec
{'usuario': 'Don Miguel Feroz', 'rol': 'isNormal', 'iat': 1667933533}
**********
{'usuario': 'Don Miguel Feroz', 'rol': 'isAdmin', 'iat': 1667933533}
**********
```

Pero lo lógico sería que desconociera completamente la clave con la que hemos generado el JWT, por lo que probaría cualquier cosa (en el código he puesto la clásica clave 123456 para hacer la prueba) y la verificación daría un error; de aquí la importancia de realizar la operación de verificación en todo caso para comprobar este tipo de ataques.

```
(Criptologia) D:\Ciberseguridad\Criptografia\criptografia-main>C:/Users/Gerard/anaconda3/envs/Criptologia/python.exe "d:/Ciberseguridad/Criptografia/criptografia-main/Código Fuente/Hashing y Authentication/Ejercicio04.py"
b64pwd: 3476695663786337535O4a776b6a2b67384963355673683362 6a3376734c476568612b5178596235377453d
2432612431322448 4d776237354952694633 6a5a7a442e5a734f69454f52657551346b6c4578787976 7b7244705672463354635361 2f546c6f716465
Password correcta
Codigo normal: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIE1pZ3VlbCBGZXJveiIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ.snCUE066pVyiDstmWwQAkvDizxB_XXhxcWaVZmUHZbU
Codigo modificado: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIE1pZ3VlbCBGZXJveiIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.MX_AeJhPo3xNK_Be-i4Ddp8FRpewOVQwkZ6oTlnJtec
Traceback (most recent call last):
  File "d:\Ciberseguridad\Criptografia\criptografia-main\Código Fuente\Hashing y Authentication\Ejercicio04.py", line 13, in <module>
    decode_jwt_hackeado_sin_contrasenya = jwt.decode(encoded_jwt_hackeado,"123456", algorithms="HS256")
  File "C:\Users\Gerard\anaconda3\envs\Criptologia\lib\site-packages\jwt\api_jwt.py", line 129, in decode
    decoded = self.decode_complete(jwt, key, algorithms, options, **kwargs)
  File "C:\Users\Gerard\anaconda3\envs\Criptologia\lib\site-packages\jwt\api_jwt.py", line 100, in decode_complete
    decoded = api_jws.decode_complete(
  File "C:\Users\Gerard\anaconda3\envs\Criptologia\lib\site-packages\jwt\api_jws.py", line 182, in decode_complete
    self._verify_signature(signing_input, header, signature, key, algorithms)
  File "C:\Users\Gerard\anaconda3\envs\Criptologia\lib\site-packages\jwt\api_jws.py", line 269, in _verify_signature
    raise InvalidSignatureError("Signature verification failed")
jwt.exceptions.InvalidSignatureError: Signature verification failed
```

5) **El siguiente hash se corresponde con un SHA3 Keccak del texto "*En KeepCoding aprendemos cómo protegernos con criptografía*".**

Bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

(Observar código del archivo Ejercicio05.py)

¿Qué tipo de SHA3 hemos generado?
Hemos generado un **sha3_256**.
Podemos saberlo utilizando la siguiente línea de código que nos informa del algoritmo utilizado:

```
print(s.name)
```

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

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

¿Qué hash hemos realizado?
En este caso hemos utilizado un **sha512.**

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?

En este caso estamos añadiendo un punto final al texto en claro. Gracias a esta pequeña modificación, comprobamos claramente la propiedad de difusión, con el efecto avalancha, por el cual nos cambia completamente el hash generado con sólo modificar un bit del texto en claro.

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.

(Observar el código del archivo Ejercicio06.py)

La clave, obtenida del Keystore es:
7212A51C997E14B4DF08D55967641B0677CA31E049E672A4B06861AA4D5826EB

Utilizando el código correspondiente y los datos que nos ofrece el propio enunciado, sacamos fácilmente el hmac-256 solicitado. Supongo que en este caso no hacen falta más explicaciones.
El resultado obtenido por consola es el siguiente:

```
(Criptologia) D:\Ciberseguridad\Criptografia\criptografia-main>C:/Users/Gerard/anaconda3/envs/Criptologia/python.exe "d:/Ciberseguridad/Criptografia/criptografia-main/Código Fuente/Hashing y Authentication/Ejercicio06.py"
915bf9fbe64f837a0dffbacba6b85a49758c829f2e2970d8471f600dfd250bdc
hmac en verificacion: 915bf9fbe64f837a0dffbacba6b85a49758c829f2e2970d8471f600dfd250bdc
result: OK
OK
```

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?

Un hash SHA-1 será una cadena de 20 bytes, longitud que no presta la mejor seguridad posible a día de hoy. Además, desde el año 2005 no se considera seguro y, en 2017, se encontraron ataques de colisión perpetrados. Teniendo varias opciones más seguras por escoger, no se puede recomendar el uso de un simple hash SHA-1.

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?

Podríamos añadir 2 valores aleatorios concatenados al password. Estos 2 valores serían:
Salt (que es un aleatorio no secreto)
Pepper (otro aleatorio, pero esta vez sí debería ser secreto)
La estructura en la que deberían utilizarse sería: **password + Salt + Pepper** (concatenación) y hashearíamos esta combinación.

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

Un tercer nivel de seguridad podría ser utilizando KDF (Key Derivation functions), que son funciones específicamente diseñadas para el uso con passwords, que implementan ralentizaciones intencionadas en la ejecución para dificultar ataques de fuerza bruta y no son vulnerables a diccionarios, hash tables o rainbow tables.
La más recomendada a día de hoy sería **Argon2id**, puesto que es muy sencilla de implementar por código, ya que no nos pide casi ningún parámetro de entrada.

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 cualquier lenguaje de programación de cómo resolverlo?

Felipe, este ejercicio tampoco me ha dado tiempo a desarrollarlo. Teniendo en cuenta que no tengo muy claro el concepto de API REST, lo que sí haría es implementar un cifrado con AES-GCM (que siempre has dicho que es de lo mejor que podemos utilizar, en términos de seguridad) y nos dará integridad y autenticación.

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 si no hay sorpresas.

...
No me ha dado tiempo a realizarlo, Felipe. Pero acabaré haciéndolo por mi cuenta.
...

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:

    7edee3ec0b808c440078d63ee65b17e85f0c1adbc0da1b7fa842f24fb06b33
    2c156038062d9daa8ccfe83bace1dca475cfb7757f1f6446840044fe698a63
    1fe882e1a6fc00a2de30025e9dcc76e74f9d9d721e9664a6319eaa59dc9011
    bfc624d2a63eb0e449ed4471ff06c9a303465d0a50ae0a8e5418a1d12e9392
    faaaf9d4046aa16e424ae1e26844bcf4abc4f8413961396f2ef9ffcd432928
    d428c2a23fb85b497d89190e3cfa496b6016cd32e816336cad7784989af89f
    f853a3acd796813eade65ca3a10bbf58c6215fdf26ce061d19b39670481d03
    b51bb0eecc926c9d6e9cb05ba56082a899f9aa72f94c158e56335c5594fcc7
    f8f301ac1e15a938

    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 cifrarla con el mismo algoritmo. ¿Por qué son diferentes los textos cifrados?

    (Observar archivo Ejercicio10.py)
    Para descifrar y recuperar la clave, utilizaremos el bloque de código correspondiente a desencrypt de **RSA-OAEP**, utilizando la clave privada **clave-rsaoaep-priv.pem**

    De este modo obtenemos la clave:
    e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

    Si volvemos a cifrar esta clave con el bloque de código correspondiente a encrypt RSA-OAEP obtendremos cada vez un código diferente por el funcionamiento interno del propio algoritmo RSA-OAEP y las operaciones que realiza intrínsecamente, utilizando, además, claves diferentes que va generando.

    Ejemplos de cifrado resultantes:

1) *83872bb77b60d467d3940acfd559f74975007f43c8a2593a2d6092a2423edab1c8ab0b9f2db6*
   44a63174b000f4a7b6dce4e0cab2cec2632eb7e014304695e78cd8622c4ba331596dcc0e34ec
   7ed40629bb32a1aa1138b082888753f5a08b48bfc3fadc41506d87b90243019c097bd83abd6a
   8c662b7cef193221849c48b7d1cadfd8c050df4747912410dda2b8ba8b5b68c1277e8c3915b6
   e1b6b5e7c6fa26786fd8ee67b72181856c56cbcca44191a1d4ee7f3d4233b0a0c39c29e2a47de
   21bddf72745ef893b2cea880d504cce4aed0d8b0815ce17a24b6c62a77259c2a9340be4d532c
   d2f2d09980859a87093daadb56665b2b11ba2abc631b76dfed188c1
2) *bb437f741374ad645eb92a0837156f9bd6e9e3a5659c62c5b443ccebfcedb389244fae949f51f1*
   8ce25d2a417476ec72fd2739fcf29b38d90c2a513760f3f8b92882038be1269d2909abdf4a76f6
   27cc5dc868b874330b581202d7512d419cf5028b5fd528a29b625c0490b004e6fa51b3c0fc24
   be825e98d0e5919a4a4b9124a1d814a5a98afd4886644b0f5c734748fee0aa62fc0d695025b6
   011e8ab67746121747080322b108d288bbc2db3802340e00bd6bd88c08f890802fa8010186d
   4762d79bb5d2f53ba0b56c05e9540bd1f0ecaedd52f85be208bac89a4a381624c420329dafc9

6a93931ed855a0da5edaea4935d7c0b525559b4cfa4c486d70353

3) etc.

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 definición, el nonce debería ser siempre y en todo caso un valor aleatorio, para asegurar la autenticación, confidencialidad e integridad en el cifrado del mensaje. En este caso, vemos que se nos está dando un Nonce fijo o predefinido, siendo esto todo lo contrario a lo que la buena praxis recomienda.

Para obtener un nonce aleatorio, podríamos usar la siguiente línea de código en Python, importando la librería correspondiente primeramente:

```
from Crypto.Random import get_random_bytes

nonce = get_random_bytes(8)
```

o también:

```
import secrets

nonce = secrets.token_bytes(8)
```

12. 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

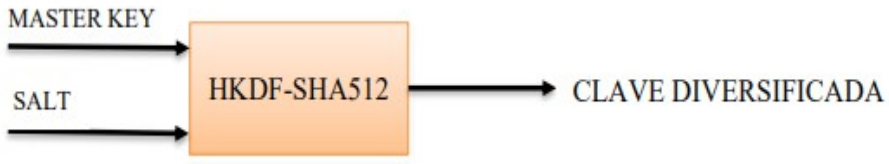
¿Cuál es el valor de la firma en hexadecimal?

(Revisar archivo Ejercicio12.py)
En este caso tan sólo ha sido necesario utilizar el código de "**RSASignature**", concretamente el bloque de código que sirve para firmar un dato determinado.

El valor de la firma en hexadecimal obtenido es:
4fc878ac4a9b45c73eebe23bfb9aea1c01ce368a516e9e783d6bae21c55d20c1baac0868f7331cc53e8dbace34840b2189535d470b984c105c4f422d4cf7fa793550f783810dbfc8f3f69eba0d3892b7bb00388a8056d94738a569a16a5a02cc67c18cb1ec408110ee230ea8b5383c6a54095dec841f0d9bca94a241bd03fbd546c6ab9dd906793921c7c72d2567bcd654d3d87fea33e0c7be6dc44e95f3c6f944d9c20bebcbc7c46a6d172feb6fa37ec9b6080722375cbb640f182872c6862457fd26cd0430f9f605c52227fd1ef84e4b36fd842bd76b38a8ce93e8c5137b4b8f19073c5d6900eade3761d34ad60a4752eb78cc38591ada0469c2da6e10a3d2

13. **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?

    Clave del Keystore:
    E2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72

    En este caso también se puede revisar el código fuente del archivo Ejercicio13.py para ver cómo lo he implementado. Los datos de entrada los pasamos a binario y posteriormente aplicamos la siguiente línea de código:

```
key1 = HKDF(master_secret, 32, salt, SHA512, 1)
```

    Obtenemos así la clave siguiente:
    1dc9ec8a63df7374ee3e83f2a7cb0b7d68909f8dea7e59d79c194254799bc7f3

14. **Nos envían un bloque TR31:**

    D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACD BE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495 E03CD857FD37018E111B

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

    A1A1010101010101010101010101010102

    Obtenemos la siguiente información aplicando el código fuente de "CabeceraTR31":

```
(Criptologia) D:\Ciberseguridad\Criptografia\criptografia-main>C:
14.py"
c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1
Key Version ID: D
Algoritmo: A
Modo de uso: B
Uso de la clave: D0
Exportabilidad: S
```

Podemos utilizar el documento de la siguiente web, para observar la información obtenida: *https://github.com/knovichikhin/psec/blob/master/psec/tr31.py*

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

Key Version ID: D ---> Se ha protegido con AES

Value	Description	Algorithm
'A'	Key block protected using the Key Variant Binding Method. This version is deprecated.	DES
'B'	Key block protected using the TDES Key Derivation Binding Method. Recommended over versions 'A' and 'C' for new implementations.	DES
'C'	Key block protected using the TDES Key Variant Binding Method. Same as 'A' with some header value clarifications.	DES
'D'	Key block protected using the AES Key Derivation Binding Method.	AES

¿Para qué algoritmo se ha definido la clave?

Algoritmo: A ---> Para el algoritmo AES también.

Value	Definition
'A'	AES

¿Para qué modo de uso se ha generado?

Modo de uso: B ---> Para modos de cifrado y descifrado.

Value	Definition
'B'	Both Encrypt & Decrypt / Wrap & Unwrap

¿Es exportable?

Exportabilidad: S ---> Sí, es exportable

Value	Definition
'E'	Exportable under trusted key
'N'	Not exportable
'S'	Sensitive, exportable under untrusted key

¿Para qué se puede usar la clave?

Uso de la clave: D0 ---> Clave simétrica para encriptar

Value	Description	Mode of Use
'B0'	BDK Base Derivation Key	'X'
'B1'	Initial DUKPT Key	'X'
'B2'	Base Key Variant Key	'X'
'C0'	CVK Card Verification Key	'C', 'G', 'V'
'D0'	Symmetric Key for Data Encryption	'B', 'D', 'E'

¿Qué valor tiene la clave?
c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1