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 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?

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?

El valor introducido en properties en el primer caso es: **20553975C31055ED**

La clave con la que se trabaja en memoria es: **08653F75D31455C0**

Para obtenerlas en ambos casos aplicamos XOR a ambos términos tal y como se realiza en el archivo ***01 - disociación.py***. En Python iteramos byte a byte de cada una de las partes de la clave usando el método zip, y aplicamos XOR en cada caso.

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/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nvvJbBiG+le1ZhpR84oI=

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).

El dato en claro es: **Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.**

Para obtenerlo primero obtenemos la clave del KeyStore utilizando librería PyJKS, y creamos el IV también bytes del tamaño del bloque de AES (16 bytes). Decodificamos el dato de base64 a bytes también. Una vez hecho esto lo desciframos y

posteriormente eliminamos el padding tal y como se muestra en el archivo ***02 - cifrado-sim-aes-256.py***. Finalmente lo decodificamos en UTF-8.

Respecto al cambio de padding, si cambiamos el style de PKCS7 (por defecto) a x923 **en este caso no supone ningún cambio** ni da ningún error, ya que el último byte es la longitud del padding, igual que en PKCS7 y la implementación de PyCryptodome lo maneja bien. En otras implementaciones podría fallar ya que el padding esperado en x923 es de 0s hasta el último byte que indica la cantidad final de padding.

El padding que se ha añadido es de un byte. Esto lo sabemos accediendo a la última posición del dato descifrado antes de eliminar el padding (dato_descifrado[-1]).

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-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.

Para garantizar integridad además de confidencialidad, la mejora más sencilla es usar ChaCha20-Poly1305, que añade un código de autenticación (MAC) Poly1305. Además podemos añadir datos asociados para validar la integridad de la información encriptada.

El desarrollo se muestra en el archivo ***03 - cifrado-sim-chacha-256.py***.

En este caso el texto cifrado quedaría:
TslZIcqLdX4jNmBcfbq49NQLW00iDmaql490DT5ZsM1w4yFyQpkcwUC7Hho=

La MAC:
Q8XzUHgv1eu8onddwwJ+Ag==

Y los datos asociados son:
Usuario: Keepcoding

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?

El body del jwt es la parte central entre los dos puntos:
eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ

Que decodificado de base64 nos da lo siguiente:

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

"rol":"isNormal",

"iat":1667933533}

Decodificando el header nos da lo siguiente: *{"typ":"JWT","alg":"HS256"}*

Por lo que el algoritmo de firma es **HS256.**

Comparando el primer jwt con el segundo, comprobamos que en el primero la entrada rol es isNormal mientras que en el segundo es isAdmin, por lo que estaría intentando realizar una escalada de privilegios. Si lo intentamos validar con pyjwt nos da un ***InvalidSignatureError*** tal y como se muestra en el archivo ***04 - jwt.py***.

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:

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

¿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?

El primero tiene 64 caracteres hexadecimales, que corresponden a 32 bytes, es decir a 256 bits, así que sería un **SHA3-256**.

El SHA2 tiene 128 caracteres hexadecimales que corresponden a 64 bytes y por tanto a 512 bits, así que sería un **SHA2-512**.

El SHA3 Keccak del texto propuesto es[1]:
302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

Lo que observamos comparando este hash con el primero es que, con tan solo añadir un punto al final del texto, el hash ha cambiado prácticamente en su totalidad, lo que se conoce como la propiedad de difusión, por la que un pequeño cambio en el input genera un cambio drástico en el output.

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.

El HMAC-256 es:
857D5AB916789620F35BCFE6A1A5F4CE98200180CC8549E6EC83F408E8CA0550

El desarrollo se muestra en el archivo ***06 - hmac.py***, y consiste en obtener la clave con PyJKS, pasar el mensaje a bytes, y usar el método new del módulo HMAC que nos pide la clave, el mensaje y el digestmod que en este caso es SHA256.

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

[1] https://gchq.github.io/CyberChef/#recipe=SHA3('256')&input=RW4gS2VlcENvZGluZyBhcHJlbmRlbW9zIGPDs21vIHByb3RlZ2Vybm9zIGNvbiBjcmlwdG9ncmFmw61hLg&ienc=65001&oenc=65001

quedado conforme, tras mejorar la propuesta del SHA-256, no obstante, hay margen de mejora. ¿Qué propondrías?

SHA-1 no se considera seguro porque tiene **colisiones encontradas desde 2017**. Además, al calcularse tan rápido es vulnerable a ataques de fuerza bruta.

En el caso del SHA-256, aunque es más seguro que SHA-1, sigue siendo vulnerable a ataques de fuerza bruta. Pero existen algunos métodos para fortalecerlo: por un lado tenemos el **salt y el pepper**, que consiste en añadir antes y después de la contraseña unos valores en un caso fijo para toda la aplicación y en el otro variable según el usuario, de tal forma que se evitan las rainbow tables. Por el otro lado tenemos las **Key Derivation Functions** que aplican los hashes de manera recursiva un número concreto de veces, haciendo el sistema relativamente lento a propósito, para evitar los ataques de fuerza bruta.

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?

Para cifrar los datos sensibles usaríamos un algoritmo AES-GCM, ya que proporciona confidencialidad y además integridad. Como es un algoritmo de criptografía simétrica nos proporciona más velocidad que un algoritmo de criptografía asimétrica.

La request a la API quedaría de la siguiente manera:
{

“idUsuario”: 1,
“datosCifrados”: AES–GCM({“usuario”: “...”, “tarjeta”: “...”}),
“nonce”: “...”,,
“tag”: “...”

}

La respuesta de la API quedaría:
{

"idUsuario": 1,
"moneda": "EUR",
"datosCifrados": AES-GCM({“movTarjeta”: “...”, “saldo”: “...”}),
"nonce": "...",
“tag”: “...”

}

El nonce se debe generar de manera aleatoria para cada request. Los datos no cifrados nos sirven como AAD, y aunque no sean confidenciales podemos garantizar su integridad.

Para cifrar la clave AES, podemos usar un algoritmo RSA-OAEP. En una petición inicial a la API, el cliente genera una clave AES aleatoria, la cifra con la clave pública del servidor y el servidor la descifra con su clave privada.

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.

El KCV(SHA-256) es **DB7DF2**.

El KCV(AES) es **5244DB**.

El desarrollo se puede ver en el archivo ***09 - kcv.py***. En el primer caso calculamos el has SHA-256 con la librería hashlib, y utilizamos slicing para sacar los 3 primeros bytes (que corresponden a los 6 primeros caracteres hexadecimales).

En el segundo, definimos un bloque de texto plano de 16 bytes, todos a cero, ya que AES tiene un tamaño de bloque fijo de 128 bits, y un IV también compuesto por 16 bytes de ceros binarios, ciframos, y sacamos igualmente los 3 primeros bytes.

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.*

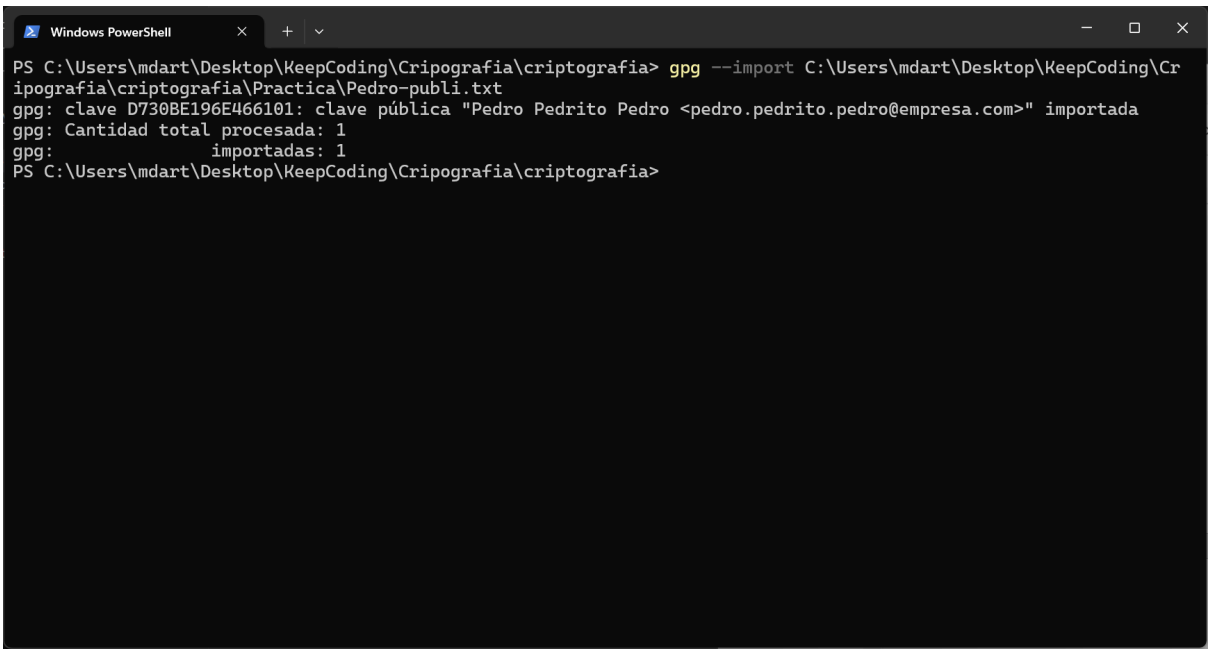
Lo acompaña del siguiente fichero de firma PGP (MensajeRespoDeRaulARRHH.txt.sig). Nosotros, que pertenecemos a RRHH vamos al directorio a recuperar la clave para verificarlo. Tendremos los ficheros Pedro priv.txt y Pedro-publ.txt, con las claves privada y pública. Las claves de los ficheros de RRHH son 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.*

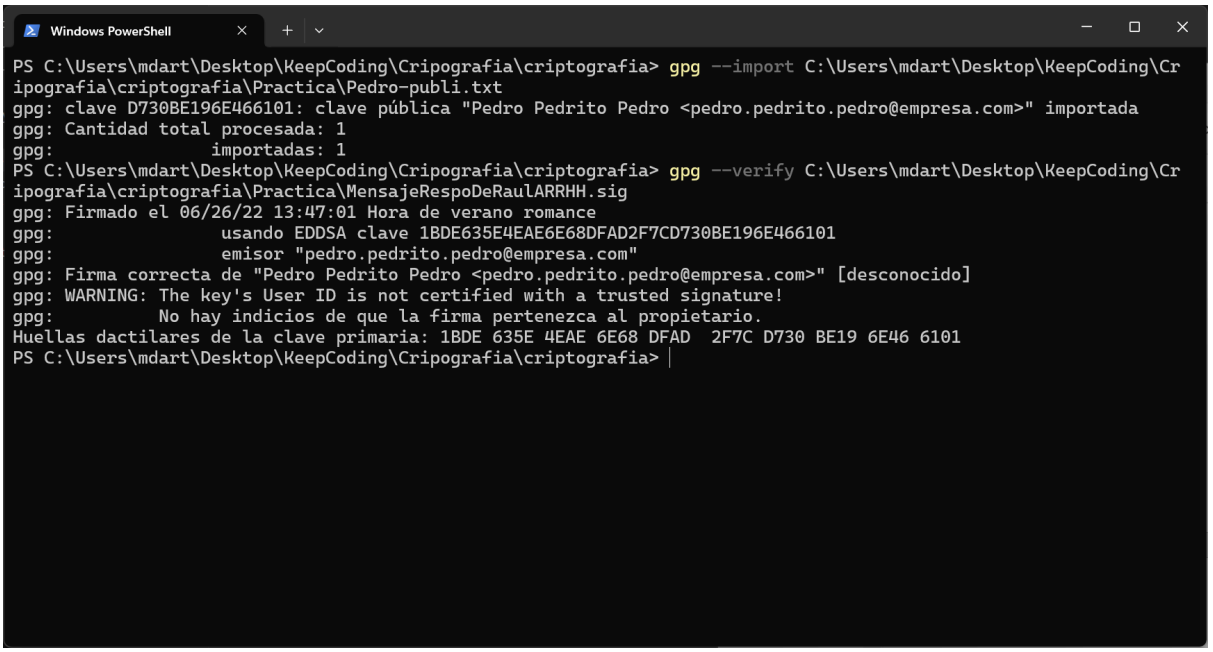
Por último, cifra el siguiente mensaje tanto con la clave pública de RRHH como la de Pedro y adjunta el fichero con la práctica.

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

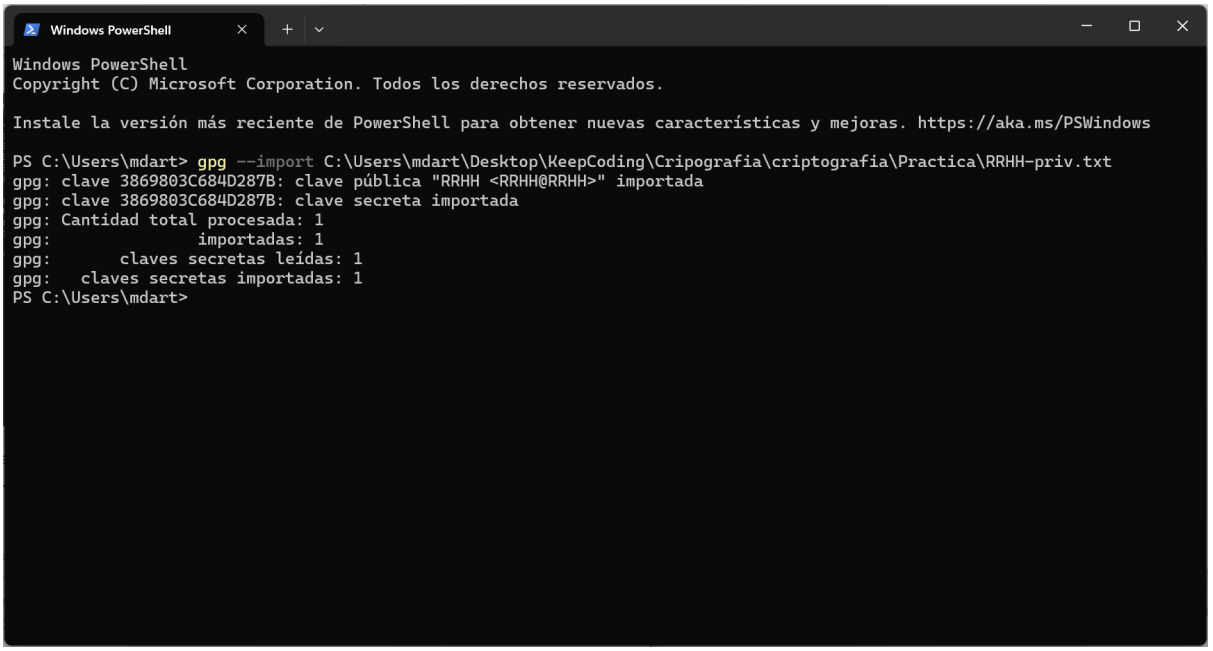
En primer lugar debemos instalar PGP si no lo tenemos y cargar la clave pública de Pedro:



Una vez hecho esto usamos gpg para verificar el fichero con la firma:

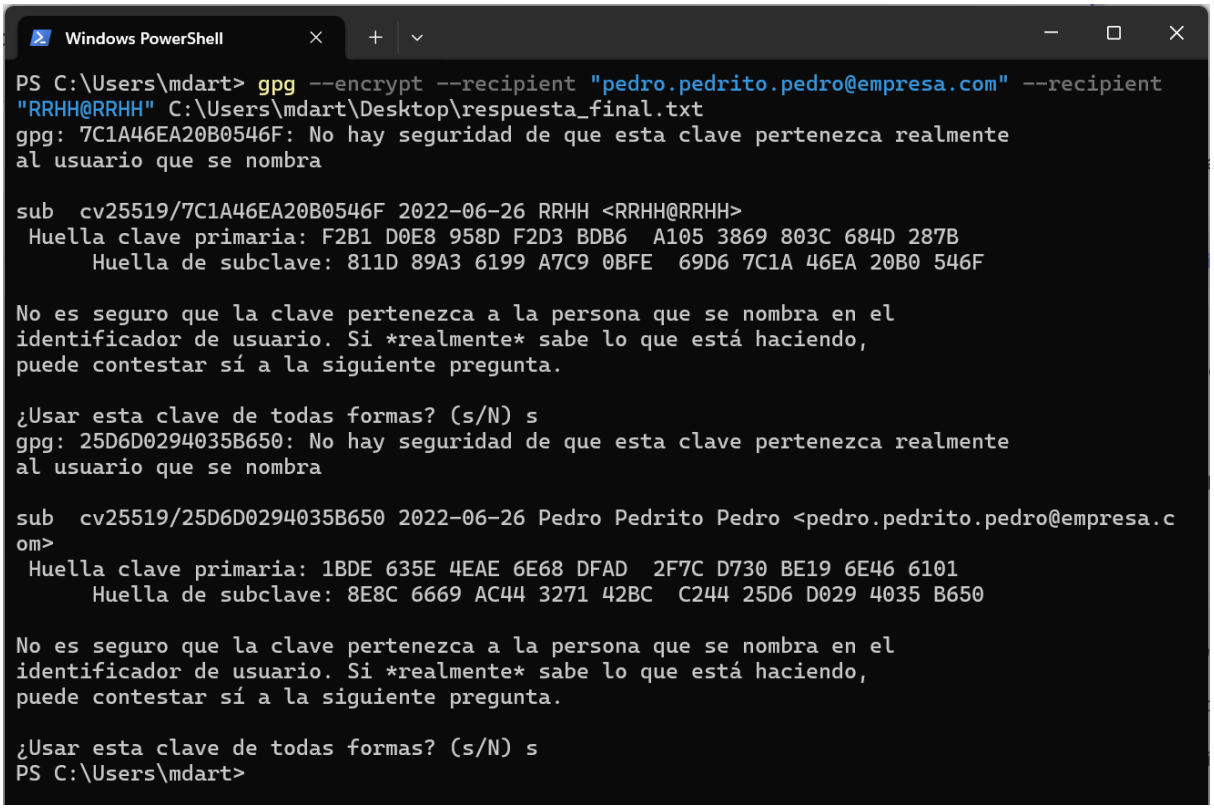


El mensaje que nos aparece nos confirma que la firma es correcta.
A continuación creamos el archivo TXT (respuesta_RRHH.txt) con el contenido indicado. Importamos la clave privada de RRHH.



A continuación firmamos el mensaje con el siguiente comando que creará un archivo respuesta_RRHH.txt.asc con el texto y la firma (incluído en el repositorio).

Para cifrar el último mensaje, repetimos el proceso de crear un archivo .txt (respuesta_final.txt). La clave pública de Pedro ya la teníamos, y al importar la privada de RRHH ya la tenemos también. De esta manera generamos un respuesta_final.txt.gpg



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:

b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d04e3d3d4ad629793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae2c4f5ba670a4126f2f14a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8624071be78ccef573d896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f67d3f98b4cd907f27639f4b1df3c50e05d5bf63768088226e2a9177485c54f72407fdf358fe64479677d8296ad38c6f177ea7cb74927651cf24b01dee27895d4f05fb5c161957845cd1b5848ed64ed3b0372 2b21a526a6e447cb8ee

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?

La clave descifrada es:
e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

Si la volvemos a cifrar (tal y como se muestra en el archivo ***11 - rsa-oaep.py***) efectivamente nos da un resultado completamente diferente. Esto es debido a que OAEP tiene un componente aleatorio del padding, y como hemos visto en otros ejercicios, al pasar por funciones de hashing, un byte diferente crea resultados completamente diferentes.

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.

Con un algoritmo como el AES/GCM **no se puede reutilizar el Nonce**, ya que este debe ser aleatorio y de un solo uso. Si se repite el mismo Nonce con la misma clave facilitamos que se rompa la confidencialidad.

Texto encriptado hexadecimal[2]:

5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abddb70dfae7a38a050f484e7bb4b11a2821f7

Tag: **79a814ad17003637eaa06a8e1057ed67**

Texto encriptado en base64[3]:

Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq923Dfrno4oFD0hOe7SxGigh9w==

Tag: **eagUrRcANjfqoGqOEFftZw==**

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 ed25519priv y ed25519-publ.

La firma con PKCS#1 v1.5 es:
A4606C518E0E2B443255E3626F3F23B77B9D5E1E4D6B3DCF90F7E118D6063950A23885C6DECE92AA3D6EFF2A72886B2552BE969E11A4B7441BDEADC596C1B94E67A8F941EA998EF08B2CB3A925C959BCAAE2CA9E6E60F95B989C709B9A0B90A0C69D9EACCD863BC924E70450EBBBB87369D721A9EC798FE66308E045417D0A56B86D84B305C555A0E766190D1AD0934A1BEFBBE031853277569F8383846D971D0DAF05D023545D274F1BDD4B00E8954BA39DACC4A0875208F

---

2 https://cyberchef.io/#recipe=AES_Encrypt(%7B'option':'Hex','string':'E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74'%7D,%7B'option':'Base64','string':'9Yccn/f5nJJhAt2S'%7D,'GCM','Raw','Hex',%7B'option':'Hex','string':''%7D)&input=SGUgZGVzY3ViaWVydG8gZWwgZXJyb3IgeSBubyB2b2x2ZXLpIGEgaGFjZXJsbyBtYWwgCg

3 https://cyberchef.io/#recipe=From_Hex('Auto')To_Base64('A-Za-z0-9%2B/%3D')&input=NWRjYmI2MjYxZDBmYmEyOWNlMzk0MzFlOWEwMTNiMzRjYmNhMmE0ZTA0YmIyZDkwMTQ5ZDYxZjRhZmQwNGQ2NWUyYWJkZGI3MGRmYWU3YTM4YTA1MGY0ODRlN2JiNGIxMWEyODIxZjc

36D3C9207AF096EA0F0D3BAA752B48545A5D79CCE0C2EBB6FF601D92978A33C1A8A707C1AE1470A09663ACB6B9519391B61891BF5E06699AA0A0DBAE21F0AAAA6F9B9D59F41928D

La firma con curva elíptica ed25519 es:
BF32592DC235A26E31E231063A1984BB75FFD9DC5550CF30105911CA4560DAB52ABB40E4F7E2D3AF828ABAC1467D95D668A80395E0A71C51798BD54469B7360D

El desarrollo se muestra en el archivo ***13 - firmas.py***. En el primer caso, importamos la clave privada usando el método importKey de RSA del módulo PublicKey de Cryptodome. Posteriormente hasheamos con SHA256 el mensaje usando el módulo Hash, y finalmente usamos el método sign con el hash de una instancia de PKCS115_SigScheme creada con con la clave. Con la clave pública verificamos la firma.

Para obtener la firma con curva elíptica ed25519 el proceso es similar, en este caso tenemos que usar la librería PyNaCl. Para crear la firma cogemos los 32 primeros bytes de la clave que serán la semilla (el resto es la clave pública) y la usamos para crear una instancia de SigningKey; con esta firmamos el mensaje en bytes. Posteriormente verificamos usando VerifyKey con la clave pública.

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?

La clave obtenida es:
F6FBC6204BD24B43C42FE1BE7D970EEECBCEE87481711A64433EA1B7EF655FFA

El desarrollo se muestra en el archivo ***14 - hkdf.py***. Respecto a los valores introducidos, escogemos una key_len de 32 ya que se trata de una clave AES-256 (32bytes). Para el salt introducimos 00 binarios del tamaño del hash (64 bytes para SHA-512).

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?

Desenvolvemos el bloque tal y como se muestra en el archivo ***15 - tr31.py*** y nos da el siguiente resultado:

- Clave importada (hex): C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1
- Versión: D
- Uso de clave: D0
- Algoritmo: A
- Modo de uso: B
- Exportabilidad: S
- Número de versión: 00

Podemos interpretarla según la documentación de psec[4]: la clave se ha protegido con **AES (AES Key Derivation Binding Method)**, la clave se ha definido para el algoritmo **AES**, se ha definido para **encriptar y desencriptar**, respecto a la exportabilidad se considera **sensible exportable bajo una clave no confiable**, se puede usar para **encriptación de clave simétrica** y tiene el valor hexadecimal de:
C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1C1

[4] https://github.com/knovichikhin/psec/blob/master/psec/tr31.py