

# Ejercicio 1

Se han disociado dos claves, se nos da una clave fija y otra clave que se ha obtenido de la memoria. Se nos pide que obtengamos la clave original establecida por el clavelero.

## Datos

memoria: 0x91ba13ba21aabb12 fija: 0xb1ef2acfe2baeeff

## Procedimiento

Para obtener la clave original, se debe realizar una operación XOR entre la clave fija y la clave obtenida de la memoria. Gracias a la propiedad conmutativa de la operación XOR, se puede obtener la clave original:

0101 XOR 1100 = 1001

0101 XOR 1001 = 1100

1100 XOR 1001 = 0101

keymanager = clave\_memoria XOR clave\_fija: 0x20553975c31055ed

```
• ~/Escritorio/practica_criptografia master python 1_ejercicio.py
Ejercicio 1
-----
Datos conocidos:
memoria: 0x91ba13ba21aabb12
fija: 0xb1ef2acfe2baeeff
keymanager = clave_memoria @ clave_fija: 0x20553975c31055ed

Mientras conozcamos dos de las tres claves, podemos obtener la tercera gracias a la propiedad conmutativa de la operación XOR.

Datos conocidos (caso contrario):
keymanager: 0x20553975c31055ed
fija: 0xb1ef2acfe2baeeff
memoria = clave_keymanager @ clave_fija: 0x91ba13ba21aabb12
-----
Datos conocidos:
memoria: 0xb98a15ba31aebb3f
fija: 0xb1ef2acfe2baeeff
keymanager = clave_memoria @ clave_fija: 0x8653f75d31455c0
```

Figure 1: Ejercicio 1

# Ejercicio 2

## Enunciado

Se nos pide descifrar el mensaje cifrado con el algoritmo AES-256 en modo CBC y relleno PKCS.

## Datos

tag en el keystore: cifrado-sim-aes-256 iv: 00000000000000000000000000000000 (16 bytes) clave: A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72    cifrado: TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4US t3aB/i50nvvJbBiG+le1ZhpR84oI= tipo: AES/CBC/PKCS

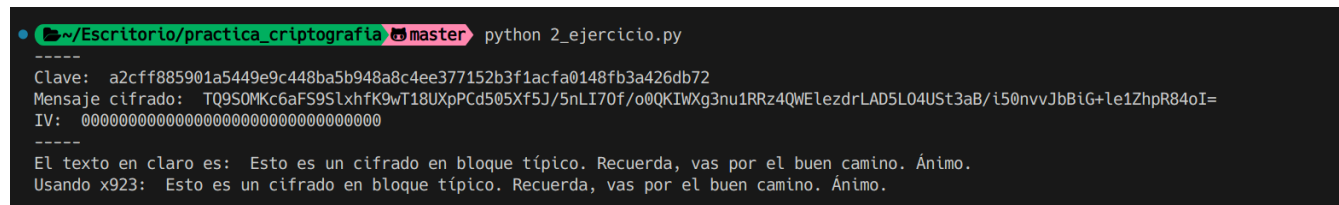
## Procedimiento

En cyberchef, podemos usar la receta “AES Decrypt” con los datos, deberemos convertir el mensaje de base64 a hex o raw, podemos usar la siguiente receta:

```
From_Base64('A-Za-z0-9+/',true,false)
```

```
To_Hex('Space',0)
```

```
AES_Decrypt({'option':'Hex','string':'A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4US t3aB/i50nvvJbBiG+le1ZhpR84oI='})
```



```
~/Escritorio/practica_criptografia master python 2_ejercicio.py
-----
Clave: a2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72
Mensaje cifrado: TQ9SOMKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4US t3aB/i50nvvJbBiG+le1ZhpR84oI=
IV: 00000000000000000000000000000000
-----
El texto en claro es: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.
Usando x923: Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.
```

Figure 1: Ejercicio 2

## Ejercicio 3

Se nos pide cifrar un mensaje en ChaCha20. Después se nos pide aumentar la seguridad del cifrado autenticando el mensaje con Poly1305, esto nos permitirá detectar si el mensaje ha sido modificado.

Después de cifrar el mensaje, se nos pide mejorar la seguridad del cifrado autenticando el mensaje con Poly1305, esto nos permitirá detectar si el mensaje ha sido modificado.

### Datos

mensaje: KeepCoding te enseña a codificar y a cifrar  
clave: AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120  
nonce: 9Yccn/f5nJJhAt2S

### Procedimiento

En cyberchef, podemos usar la receta “ChaCha20 Encrypt” con los datos, deberemos convertir el mensaje a hex, podemos usar la siguiente receta:

En python, deberemos asegurarnos de tener los datos necesarios, el mensaje, clave y nonce, convertimos a bytes el mensaje y la clave, y ciframos el mensaje con la clave y el nonce.

Para autenticar el mensaje, deberemos usar Poly1305, deberemos generar un Nonce de 12 bytes, unos datos adicionales que servirán para autenticar el mensaje, y la clave, con estos datos, generamos un tag que servirá para autenticar el mensaje.

```
• ~/Escritorio/practica_criptografia master python 3_ejercicio.py; python 3_ejercicio_poly.py
Mensaje: KeepCoding te enseña a codificar y a cifrar
Clave: AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120
Nonce: 9Yccn/f5nJJhAt2S
Mensaje cifrado: 69ac4ee7c4c552537a00a19bcaf7f0aaed7c9c8f769956a09bce6fadedf6c3535f2211c9467067cf5c4a842ab
-----
Clave: AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120
Nonce: 9Yccn/f5nJJhAt2S
Mensaje en claro = KeepCoding te enseña a codificar y a cifrar
Mensaje: KeepCoding te enseña a codificar y a cifrar
Clave: AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120
Nonce: b'G\x14DV\xc1X\xe5W\x1c\xdf\xa3U'
Datos asociados: Datos no cifrados sólo autenticados
-----
Mensaje cifrado: a4e772c56b1e53adeb826bfb62d3d1d4a49503a4b5d426dbee8ae9ca0ee1070718f39a06cfbe98c47e9501a5
Clave: AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120
Nonce: b'G\x14DV\xc1X\xe5W\x1c\xdf\xa3U'
Datos asociados: Datos no cifrados sólo autenticados
-----
```

Figure 1: Ejercicio 3

## Ejercicio 4

Se nos da un jwt y se nos pide con que algoritmo fue firmado. Luego se nos proporciona otro jwt, debemos decodificarlo y analizar que está intentando hacer el usuario e intentar validarlo con pyjwt.

### Datos

jwt: "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiriRG9uIFBlcGl0byBkZSBsb3MgeGFsb3Rle" (truncated)

### Procedimiento

Podemos separar las partes del jwt usando como separador el punto. Luego decodificamos el contenido y obtenemos el algoritmo en el campo "alg" del header.

El algoritmo con el que fue firmado el jwt es HS256.

El cuerpo del jwt es:

```
{"usuario": "Don Pepito de los palotes", "rol": "isNormal", "iat": 1667933533}
```

Si analizamos el segundo token, podemos ver que el usuario intenta cambiar su rol a "isAdmin" para obtener más permisos.

```
• ~/Escritorio/practica_criptografia master python 4_ejercicio.py
Header: {"typ": "JWT", "alg": "HS256"}
Payload: {"usuario": "Don Pepito de los palotes", "rol": "isNormal", "iat": 1667933533}
Signature: b'\x81\xf8p\xd1\xd0\xf1\xa7\xaa\xc4\xc2\xd7]\x13\xfd\xed\x13r:\xef\xd3.\xc1\xe5\x88\xc3\xd10"\xc6\xb1'
----
Header: {"typ": "JWT", "alg": "HS256"}
Payload: {"usuario": "Don Pepito de los palotes", "rol": "isAdmin", "iat": 1667933533} <-- esta intentando hacerse pasar por un administrador
Signature: b'\x92\xb8\x01\x930\x81C\x95\x99\xf0\x99\xd9\x1e\xe4o\x9ap\x19v\x0e\x191\xe4M\xbf`\x88\x00\xe0\xe5\xd1'
```

Figure 1: Ejercicio 4

## Ejercicio 5

Se nos da un mensaje y su salida en keccak SHA3 y SHA2 y se nos pide identificar el tipo de SHA que se utilizó. También se nos pide convertir a SHA3 Keccak de 256 bits y analizar un mensaje con un cambio menor en el string.

### Datos

mensaje original: En KeepCoding aprendemos cómo protegernos con criptografía  
SHA3: bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe  
SHA2: 4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c4888

mensaje a convertir: En KeepCoding aprendemos cómo protegernos con criptografía.

### Procedimiento

Podemos saber el tipo de SHA3 por el largo del hash, en este caso, el hash tiene 32 bytes, por lo que se utilizó SHA3-256.

Podemos comprobarlo con la salida al convertir el mensaje:

SHA3 del ejercicio: bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

Mensaje: En KeepCoding aprendemos cómo protegernos con criptografía  
sha3\_224 28 5f12ac6c044097c694bf740504679ef78e38a4a8fca86eb4ef9e05ae  
sha3\_256 32 bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe  
sha3\_384 48 e5bf162669b89ec5e6e7a406bf148719906ed3755baab32c891b1e0e59ec75e40564e2a3d9d4432  
sha3\_512 64 cc4d56beacf9a488f92b32b612147c088e87d2c9563c6e38bca6e834d7c742dff906dcd68b8bb8e

Si utilizamos SHA2: SHA2 del ejercicio: 4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c4888

Mensaje: En KeepCoding aprendemos cómo protegernos con criptografía  
sha224 28 a4544beb16e1dfb9b578d518bf19e2a8109ffe27cab9172911e55543  
sha256 32 13067f558aed141a490bf95775e0c6fc583a09178ae7a0fefe93a8336be81237  
sha384 48 dca9a06f36b492b374216e60dc7668bea8119ec35ca259aa797ec8125654f4dc088144b00f16d515f  
sha512 64 4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c4888

Podemos ver que el algoritmo utilizado fue sha512.

Si convertimos el mensaje dado y comparamos con el anterior:

Mensaje: En KeepCoding aprendemos cómo protegernos con criptografía.  
sha3\_256 32 302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

Mensaje: En KeepCoding aprendemos cómo protegernos con criptografía  
sha3\_256 32 bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

Podemos ver que el hash cambia completamente, por lo que los hashes son lo suficientemente distintos con un cambio menor.

```

• ~/Escritorio/practica_criptografia master python 5_ejercicio.py
Mensaje: En KeepCoding aprendemos cómo protegernos con criptografía
sha3_224 28 5f12ac6c044097c694bf740504679ef78e38a4a8fca86eb4ef9e05ae
sha3_256 32 bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe
sha3_384 48 e5bf162669b89ec5e6e7a406bf148719906ed3755baab32c891b1e0e59ec75e40564e2a3d9d4432defb28904eec7e827
sha3_512 64 cc4d56beacf9a488f92b32b612147c088e87d2c9563c6e38bca6e834d7c742dff906dcd68b8bb8ed98f045e02c2e59c6608216225179348ae4db66c65e6e927
-----
Mensaje: En KeepCoding aprendemos cómo protegernos con criptografía
sha224 28 a4544beb16e1dfb9b578d518bf19e2a8109ffe27cab9172911e55543
sha256 32 13067f558aed141a490bf95775e0c6fc583a09178ae7a0fefe93a8336be81237
sha384 48 dca9a06f36b492b374216e60dc7668bea8119ec35ca259aa797ec8125654f4dc088144b00f16d5155bcb3c1e295784f4
sha512 64 4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833
-----
Mensaje: En KeepCoding aprendemos cómo protegernos con criptografía.
sha3_256 32 302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

Mensaje: En KeepCoding aprendemos cómo protegernos con criptografía
sha3_256 32 bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe

```

Figure 1: Ejercicio 5

## Ejercicio 6

Se nos pide calcular la HMAC del mensaje “Siempre existe más de una forma de hacerlo, y más de una solución válida” usando la clave almacenada en el keystore.

## Datos

Mensaje: Siempre existe más de una forma de hacerlo, y más de una solución válida Clave: A212A51C997E14B4DF08D55967641B0677CA31E049E672A4B06861AA4D5826EB

## Procedimiento

Extraemos la clave del keystore

Utilizamos la función para crear el HMAC y obtenemos y validamos el resultado:

```
• ~/Escritorio/practica_criptografia master python 6_ejercicio.py
Clave: a212a51c997e14b4df08d55967641b0677ca31e049e672a4b06861aa4d5826eb
Datos: Siempre existe más de una forma de hacerlo, y más de una solución válida.
result: OK
HMAC 857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550
Validación: OK
```

Figure 1: Ejercicio 6

## Ejercicio 7

Se nos hacen las siguientes preguntas:

- ¿Por qué SHA-1 no es seguro?
- ¿Cómo fortalecer SHA-256 y mejorar su seguridad?

### Procedimiento

SHA-1 es vulnerable a colisiones entre hashes, con diferentes mensajes se puede generar el mismo hash, esto puede ser un problema para la integridad de los datos.

En cuanto a SHA-256, es un algoritmo más seguro que SHA-1, pero se puede mejorar su seguridad añadiendo un salt a los datos antes de cifrarlos, esto hará más difícil la generación de colisiones entre hashes.

Adicionalmente, podemos añadir un pepper a los datos antes de cifrarlos, esto añadirá una capa de seguridad adicional a los datos cifrados.

```
• ~/Escritorio/practica_criptografia master python 7_ejercicio.py
Mensaje: Hola mundo
Salt: b'\xe1k|\x02\xf6L\xaa\x02\xd8\xe3\xfd\x87\xa2\xa8\xfag'
Pepper: b'\xb7&\xb3\xec^\x1d:\xe9\xa0Y\x90\xbf\x8e\F\x98'
Hash con salt y pepper: f6d9f2980af762f92a74e8efcde71e9ff03a17d73e4f43cf6c8526dcb1c97bf1
```

Figure 1: Ejercicio 7



## Ejercicio 8

Se nos muestra una API REST que recibe datos sensibles, como tarjetas de crédito, nombres, ids, etc.

Se nos pide buscar información sobre qué algoritmo usar para cifrar estos datos, asegurando la integridad y confidencialidad de los mismos.

Se asume que el sistema no usa TLS (https) para cifrar los datos.

### Procedimiento

Podemos usar o AES/GCM o Chacha20-poly1305 para el cifrado, los dos aseguran integridad y confidencialidad.

Si buscamos rapidez, podemos elegir Chacha20-poly1305, que es más rápido que AES/GCM.



```
● ~/Escritorio/practica_criptografia master python 8_ejercicio.py
Clave: 5bc409b58122c92d56e067d5bb923416
IV Request: 6d0ddf4e1dfdc1ff95d1ece2c08059e9
IV Response: e858fb65ea7d5f0b24d00569403ebb84
Request original: {'idUsuario': 1, 'usuario': 'Jose Manuel Barrio Barrio', 'tarjeta': 123456789}
Request cifrado: d9f3f0cbb5978d6f14746841341111729de562953c1f935bef0ab0fb46336bb11983aeb55492af8dbc0efc7b556d0fba61721452d1fe17919f53f8ccbab0d454623e1c7bfdc38a50a750ba025701e48
Response original: {'idUsuario': 1, 'movTarjeta': [{'id': 1, 'comercio': 'Comercio Juan', 'importe': 5000}, {'id': 2, 'comercio': 'Rest Paquito', 'importe': 6000}], 'Moneda': 'EUR', 'Saldo': 23400}
Response cifrado: 0948ea15a7f0c3d10123b2d2f8510a70065548e0cdac68f2c2b1ced66d21618f99a1bf7481e80a7fb2ec30b2b07005263157b08508b63a868a6ed756ad95422ce9c2f16951dfd7f7af8dcd9ca3641880f16fc04de5425f57d54186d3145c5c2c221bbdcccfd6947f0c12bc9ebe465f826bde67eb1ee052671042fb74786ecea46982abf38c6da3852ca4ef24bd5c3b97edf8cbf35a7e1331c86566eb7d cc2986da50a172f5fe753118d65ae02b50468306bbd7cf83f351d767626ed89589f890
```

Figure 1: Ejercicio 8

# Ejercicio 9

## Enunciado

Se nos pide calcular el KCV de una clave AES, necesitaremos el KCV(SHA-256) y el KCV(AES).

KCV(SHA-256) => primeros 3 bytes de SHA-256 de la clave

KCV(AES) => primeros 3 bytes de AES de la clave o toda??

## Datos

Clave AES: A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72

mensaje: 00000000000000000000000000000000 iv: 00000000000000000000000000000000

## Procedimiento

Para obtener el KCV de la clave AES ciframos el mensaje (en este caso 16 bytes de 0) con la clave AES y el IV proporcionados. Luego obtenemos los primeros 3 bytes de la salida y los convertimos a hexadecimal.

Para obtener el KCV de la clave SHA-256, simplemente obtenemos los primeros 3 bytes de la salida de la función hash SHA-256 de la clave y los convertimos a hexadecimal.

```
• ~/Escritorio/practica_criptografia master python 9_ejercicio.py
Clave: A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72
Mensaje: 00000000000000000000000000000000
IV: 00000000000000000000000000000000
-----
KCV AES: 5244db
-----
KCV SHA256: db7df2
```

Figure 1: Ejercicio 9

## Ejercicio 10

Se nos pide verificar la firma del fichero “MensajeRespoDeRaulARRHH.txt.sig” y evidenciar que el mensaje fue firmado por Pedro.

Además, se nos pide firmar el siguiente mensaje:

### Datos

mensaje: Se debe ascender inmediatamente a Raúl. Es necesario mejorarle sus condiciones económicas un 20% para que se quede con nosotros.

firma pgp: MensajeRespoDRaulARRHH.txt.sig claves: Pedro-priv.txt y Pedro-publ.txt

mensaje a firmar: Viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario. Saludos.

mensaje a cifrar con publica de RRHH y con la publica de Pedro: Estamos todos de acuerdo, el ascenso será el mes que viene, agosto, si no hay sorpresas.

## Ejercicio 11

Se nos da una clave rsa privada y una clave rsa pública y un mensaje cifrado en RSA OAEP mediante SHA-256. Se nos pide descifrar el mensaje, una vez hecho, lo volvemos a descifrar y comprobamos si el cifrado es el mismo que el original.

## Datos

SHA256: b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c96b4a8a20d5dbc639e9

Hacemos `cat *.pem` para ver el contenido de los archivos y vemos que son claves RSA.

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

```
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQC/absrLf79T7cz
tzjt/hHGJ+2LTBrZ90mJqVTwCtLU5xCd9helf0iVQ+ZFZH5a1ewI3Q5hPA16R/Ad
g63clqWY4iRp4JZt84GGw2XeLURQ60VNxlufQt1aC9oU0Qi1YksI1+LqNa6y5K0w
HqZFkoq+25EGkduNh9zAPevy1kVne/lfUJsxvtgjuNFN+WieCtq3M5fszjeBM2ew
5HfHPLINKr5YpYTRkU80TmrN0R0iewSmlupaAk/hSL2ADUdmzraVqLzqvJ763R79
c0+SmugoEEDEK1xK+xCE42vu9W00d9m2ZSEJgiVeV5yDCoOKzFyJnmhL6dKYFMuD
UU1K40xtAgMBAAECggEAPqkQGoy0IsKLyKQ8QLyhe0rtOmKJj70CF8yU/5ereQLD
T9KV3xjK0sJNiX3iVz4cbLJg2Lfd+Z+/HQpUShg00c0GGBr/Y7MJPeKNYHQVHyBF
qbY7nCE5cRbcJ2Bep3Ir+hMiN2Wnc0ykIS2HZNMaFGyWRyRMaUKGo3Ah43b9dWhx
RYhLee8CD1c9I1lkrZ2UycmeJdgWe+CmU0iFWH87r0FBcqVI+6zlKMk2IRh/HfVp
v646k0wKBF3XPT4YFjX+t2JSeLSaQbRQ+aVq3Twyz354GvPvaVsON91FsToQbj+1
f0JRzleWz/CU1X1bhnTvd8TCMv8+hPzf3wey0eTcgQKBgQDdm6wdu54/B/pfQ1UX
T3cyYHFKxDj5S/s96H1LcuUR89Sn7LQwwkkZYwki/osm9B5e54/rbSop/1+beCo0
OxKpSozc8/xms3ulbwpDxR3+xTMj9vAGjjp2hw5vylTH6o7Kyq8kYpjmGnqNpCf
ANv7mfD18Jvw/kIAEpWxEo6+HQKBgQDdHm4jLWek8PZbBUmZajzzf1MddXMKX50u
dzhKOF2W57WutJpYRbg4/sz3Ty6qtulDexuWnw+feEkroq8cMfBB7FQaQPtq3nac
coWkTSEUG7Tz1RQ318kslVVJ3Y93iSaoMtrThcaa18+FmVG3SwBef10uEX8SpAVg
1iP0+pfWkQKBgQCTDDwuUpOT4ZhaY2qRGDLQ47vpT8E6cxeYoczyqp+jxPcEIoyC
oLjetp+Wb+8n/u60LNWL85j52zG2uQq2/K3KVeSYrPF7uHdAdCkMhRz9NB9WKwJk
ZzYV91I3DbwqF9N+bvW+oGZtHHKTbneSeoB+OEzoVzsys5RZ9fsMT3MWZQKBgAye
W/Kt+Kg1CBoRpy2WHnxW28tmlHYXF5U8EH5L0St3darOq7A16112UQQcBLHBVnZ/
ZAeodB/JoYNX+V5Gi0t3zSTiaHak02gCMRY7QJQBMMZpdonpSpW8v+1DM5jCvu4C
WPKRQ9A6WKFrKnqnURITbAXhAbtymMv57HtigZ/BAoGAdpmMRDQNKqai7aGbmbmF
Wy1GbLITkxWAOFScQUYrFs8cu0Gu79aB7PHwze0IHk/5ESj/gz7hoKJt0gi4ikx
zG21Yqqe11/Gg6wHendR1qR8VrbLBkpqylFTGusmLBUq7y4E/z9y2b4rMciU30Y2
X230g/Q6y6kMprauaCuxNSk=
```

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

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

```
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAv2m7Ky3+/U+3M7c47f4R
xifti0wa2fdJialU8ArS10cQnfYXpXzolUPmRWR+WtXsCN00YTwNekfWHY0t3Ja1
mOIkaeCWbfOBhsN13i1EU0tFTcZbn0LdWgvaFNEItWJLCNfi6jWusuSjsB6mRZKK
vtuRBpHbjYfcd3r8tZFZ3v5X1CbMb7YI7jRTflongratzOX7M43gTNnsOR3xzyy
DSq+WKEOZFPNE5qzdEdInsEppbqWgJP4Ui9gA1HZs62lai86rye+t0e/XDvkpro
KBBAxCpcSvsQhONr7vVtNHfZtmUhCYI1X1ecgwqDisxciZ5oS+nSmBTLg1FNSuDs
```

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

## Procedimiento

Usaremos la clave RSA privada para descifrar y volver a cifrar el mensaje.

El padding de PKCS1\_OAEP que usamos introduce aleatoriedad en el cifrado, por lo que al descifrar y volver a cifrar el mensaje, obtendremos un mensaje cifrado diferente.

```
~/Escritorio/practica_criptografia master python 11_ejercicio.py [11:17]
Mensaje cifrado: b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d04e3d3d4ad629793eb00c
c76d10fc00475eb76bfbcb1273303882609957c4c0ae2c4f5ba670a4126f2f14a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8624071be78cccf573d896b8eac86f5d43ca7b10b59be4acf8f8
e0498a455da04f67d3f98b4cd907f27639f4b1df3c50e05d5bf63768088226e2a9177485c54f72407fdf358fe64479677d8296ad38c6f177ea7cb74927651cf24b01dee27895d4f05fb5c161957845cd1b5848
ed64ed3b03722b21a526a6e447cb8ee 512
Clave privada: Private RSA key at 0x7F84B3545870
Clave pública: Public RSA key at 0x7F84B3F18FD0
Mensaje descifrado: e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72
-----
Mensaje cifrado de nuevo: 58fb8604479956df1f3edb215c4a9499a491c9516e263e1d5519075041804f7a7d7d1bd7c62e3dab714b73ebb6366163389c2bdb164738ded9b0a136e552541ae34eb92be1f4
928011180576a2e7e4f3ebee5c1a8b342af2267760bde32178fc94de425fffc155f9d992a69672341276947f3528fff95f437ed68dcd9127578f7937127b7112dc6d879cbc81e9c3598bf1bfb2d576f7aaba66
317eeb3bb71c41673bdc80eb839eacd1f7d732de1e19a14d600346f2cf4dc65a25a77e7e500cf04391b93c544a5ce20c1313f470812ac5620a7a1038520abb91ac27fbb8be48b36a6dec487e58be721789d625
f84e280f90cc12ceb19a3da371487e2174de8cc9 512
```

Figure 1: Ejercicio 11

## Ejercicio 12

Debemos detectar un fallo en el uso del siguiente algoritmo AES/GCM.

### Datos

Clave: E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74 Nonce:  
9Yccn/f5nJJhAt2S

### Procedimiento

Usando los datos conseguimos cifrar y descifrar un mensaje. Sin embargo, en el enunciado se dice que en cada comunicación el nonce es compartido. Lo ideal es que el nonce sea único para cada mensaje cifrado.

## Ejercicio 13

Se nos pide calcular una firma en hexadecimal con PKCS#1 v1.5 del mensaje dado. Para ello, se nos proporciona la clave privada y publica en formato pem.

Además, debemos calcular el valor de la firma hexadecimal con la curva elíptica ed25519 del mensaje dado con otras claves.

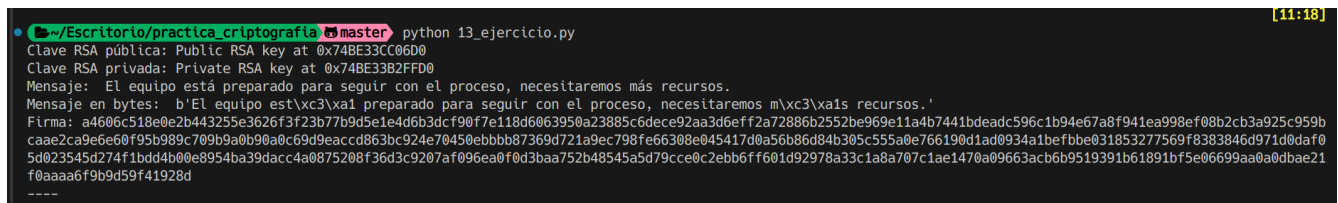
### Datos

mensaje: El equipo está preparado para seguir con el proceso, necesitaremos más recursos clave  
privada: clave-rsa-oaep-priv.pem clave publica: clave-rsa-oaep-publ.pem

clave privada ed: ed25519-priv clave publica ed: ed25519-publ

### Procedimiento

Importamos nuestras claves publica y privada en formato pem y el mensaje a firmar. Calculamos el hash del mensaje con sha256 y firmamos el hash con la clave privada. Finalmente, mostramos la firma en hexadecimal.



```
~/Escritorio/practica_criptografia master python 13_ejercicio.py
Clave RSA pública: Public RSA key at 0x748E33C06D0
Clave RSA privada: Private RSA key at 0x748E33B2FFD0
Mensaje: El equipo está preparado para seguir con el proceso, necesitaremos más recursos.
Mensaje en bytes: b'El equipo est\x03\x01 preparado para seguir con el proceso, necesitaremos m\x03\x01s recursos.'
Firma: a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959b
caae2ca9e6e60f95b989c709b9a0b90a0c69d9eacc863bc924e70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf0
5d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21
f0aaaa6f9b9d59f41928d
----
```

Figure 1: Ejercicio 13

### Dudas/Errores

No consigo firmar con la curva elíptica ed25519. En el archivo 13\_ejercicio.py, la siguiente importacion falla, aunque en el resto de ejercicios la libreria funciona correctamente:

```
from Crypto.PublicKey import Ed25519
```

## Ejercicio 14

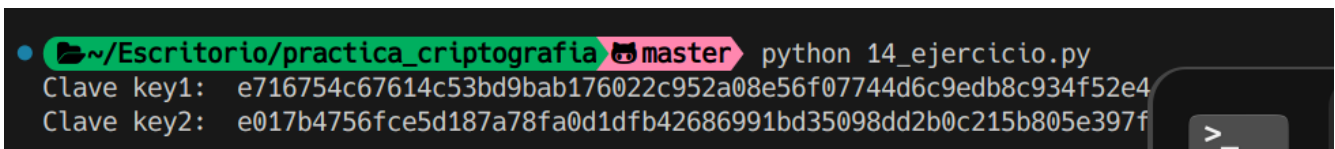
Se nos pide calcular una clave AES, usando una HKDF con SHA-512, se nos da una clave maestra en el keystore y un identificador de dispositivo.

### Datos

Master key: A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72  
identificador: e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3

### Procedimiento

Importamos la masterkey del keystore y el identificador del dispositivo. Calculamos la clave AES con HKDF y SHA-512. Finalmente, mostramos la clave en hexadecimal.



```
• ~/Escritorio/practica_criptografia master python 14_ejercicio.py
Clave key1: e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e4
Clave key2: e017b4756fce5d187a78fa0d1dfb42686991bd35098dd2b0c215b805e397f
```

Figure 1: Ejercicio 14



## Ejercicio 13

Se nos da un bloque TR31 y su clave de transporte.

Debemos obtener los siguientes datos a partir de los mismos:

- Algoritmo para proteger el bloque de clave.
- Para que algoritmo se ha definido la clave
- Para que modo de uso se ha generado
- Si es exportable o no
- Para que podemos usar la clave
- Valor de la clave

## Datos

Bloque TR31:

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C0

Clave de transporte:

A1A101010101010101010101010101010102

## Procedimiento

Podemos separar las partes del bloque TR31 para obtener los datos que necesitamos.

Utilizando python y la librería pytr31 podemos obtener los datos que necesitamos:

c2c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c2

Key Version ID: D

Algoritmo: A

Modo de uso: B

Uso de la clave: D0

Exportabilidad: S

Interpretando estos valores podemos saber que:

Key Version: Key block protected using the AES Key Derivation Binding Method. Modo de uso: Both Encrypt & Decrypt / Wrap & Unwrap Algoritmos posibles: AES Key usage: Symmetric Key for Data Encryption Exportable: Sensitive, exportable under untrusted key

```

• C:\Escritorio\practica_criptografia\master python 13_ejercicio.py
Clave RSA pública: Public RSA key at 0x74BE33C060D0
Clave RSA privada: Private RSA key at 0x74BE33B2FFD0
Mensaje: El equipo está preparado para seguir con el proceso, necesitaremos más recursos.
Mensaje en bytes: b'El equipo está preparado para seguir con el proceso, necesitaremos m\xc3\xa1s recursos.'
Firma: a4606e318e0e2b44325e3e26f3f23b77b9d5e1e4d6b3dcf90f7e118d663950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b9467a8f941ea998ef08b2cb3a925c959bcaae2ca9e660f9b5989c709b9a0b90a0c69d9eacd86b3c924e70450e6bbb87369d721a9ec798fe66308e45417d0a56b86d4b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0da0f5d023545d27471b0db4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d9278a33c1a8a707c1ae1470a09663ac6b9519391b61891bf5e06699aa0a0dbae21f0aaa6f9b9d59f41928d
-----

```

Figure 1: Ejercicio 13