

PRÁCTICA CRIPTOGRAFÍA

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?

Primero haremos:

VALOR = clave fija XOR clave final

XOR Calculator

Thanks for using the calculator. [View help page.](#)

I. Input: [hexadecimal \(base 16\)](#) ▾

b1ef2acfe2baeff

II. Input: [hexadecimal \(base 16\)](#) ▾

91ba13ba21aabb12

[Calculate XOR](#)

III. Output: [hexadecimal \(base 16\)](#) ▾

20553975c31055ed

Y segundo y último haremos la operación a la inversa para ver si esta bien:
clave fija = VALOR XOR clave final

XOR Calculator

Thanks for using the calculator. [View help page.](#)

I. Input: [hexadecimal \(base 16\) ▾](#)

b1ef2acfe2baeff

II. Input: [hexadecimal \(base 16\) ▾](#)

20553975c31055ed

[Calculate XOR](#)

III. Output: [hexadecimal \(base 16\) ▾](#)

91ba13ba21aabb12

[Home](#) [Help](#) [Privacy](#)

Como podemos comprobar, está todo correcto. El valor puesto por el Key Manager es : 20553975c31055ed

La clave fija, recordemos es B1EF2ACFE2BAEFF, 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?

Igual que antes haremos:

clave final = clave fija XOR clave dinámica

XOR Calculator

Thanks for using the calculator. [View help page.](#)

I. Input: hexadecimal (base 16) ▾

b1ef2acf2baeef

II. Input: hexadecimal (base 16) ▾

b98a15ba31aebb3f

Calculate XOR

III. Output: hexadecimal (base 16) ▾

8653f75d31455c0

[Home](#) [Help](#) [Privacy](#)

Y ahora para comprobar que está bien:

Clave dinámica= clave fija XOR clave final

XOR Calculator

Thanks for using the calculator. [View help page.](#)

I. Input: hexadecimal (base 16) ▾

b1ef2acfe2baeeff

II. Input: hexadecimal (base 16) ▾

8653f75d31455c0

Calculate XOR

III. Output: hexadecimal (base 16) ▾

b98a15ba31aebb3f

[Home](#) [Help](#) [Privacy](#)

Como podemos comprobar, esta todo correcto, la clave final es: 8653f75d31455c0

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/o0QKIWXg3nu1RRz4Q
WElezdrLAD5LO4UST3aB/i50nnvJbBiG+le1ZhpR84ol=

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

Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.

```

keystore.py > ...
1 import jks
2 import os
3
4 # Obteniendo el path
5 path = os.path.dirname(__file__)
6
7 keystore = path + '/KeystorePracticas'
8
9
10 ks = jks.KeyStore.load(keystore, "123456")
11
12 for alias, sk in ks.secret_keys.items():
13     if sk.alias == "cifrado-sim-aes-256":
14         key = sk.key
15
16     print("La clave es:", key.hex())
17
18 #

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

zsh: corrupt history file /home/kali/.zsh_history
source /home/kali/.zshrc
source /home/kali/criptografia/venv/bin/activate
└─[kali㉿kali]~/criptografia

- \$ source /home/kali/criptografia/venv/bin/activate
- [venv]-(kali㉿kali)~/criptografia
- \$ /home/kali/criptografia/venv/bin/python /home/kali/criptografia/keystore.py
- La clave es: a2cff885901a5449e5c448ba3d948a8c4ee377152b3f1acfa0148fb3a426db72
- [venv]-(kali㉿kali)~/criptografia

In 7, Col 19 Spaces:4 UTF-8 LF { } Python 3.13.9 (venv) Q

Last build: 2 years ago - Version 10 is here! Read about the new features here Options About / Support ?

Recipe Input

From Base64

Alphabet A-Za-zA-Z+= Remove non-alphabet chars

Strict mode

To Hex

Delimiter None Bytes per line 0

AES Decrypt

Key a2cff885901a544 ... HEX IV 0000000000000000 ... HEX

Mode CBC Input Hex Output Raw

STEP BAKE! Auto Bake

Input: TQ950MKc6aFS9SlxhfK9wT18UXpPCd505Xf5J/5nL170f/08QK1wKg3nu1RRz4QwElezdrLAD5L04UST3aB/i50nvvJbB1G+le1ZhpR84oI=

Output: Esto es un cifrado en bloque tipico. Recuerda, vas por el buen camino. Animales.

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

No ocurre nada ya que al tener solo 1 byte de padding ambos coinciden por eso funciona igualmente.

¿Cuánto padding se ha añadido en el cifrado?

1 byte.

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.

```

keystore.py ✘ Ejemplo KeyStore.py ● HKDF Clase.py ✘ prueba.py ● KeyStorePracticas .../Gestión de Claves ✘ Export731.py ✘ KeyStorePracticas / RSA-OAEP.py ● RSA_OAPV_ D> v
↳ keystore.py > ...
1 import jks
2 import os
3
4 # Obteniendo el path
5 path = os.path.dirname(__file__)
6
7 keystore = path + '/KeyStorePracticas'
8
9
10 ks = jks.KeyStore.load(keystore, "123456")
11
12 for alias, sk in ks.secret_keys.items():
13     if sk.alias == "cifrado-sim-chacha20-256":
14         key = sk.key
15
16     print("La clave es:", key.hex())
17
18 #

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

(vENV)-(kali㉿kali)-[~/criptografia/criptografia-main/codigo fuente]
$ cd ./Gestión de Claves/
(vENV)-(kali㉿kali)-[~/criptografia/criptografia-main/codigo fuente/Gestión de Claves]
$ /home/kali/criptografia/venv/bin/python "/home/kali/criptografia/criptografia-main/codigo fuente/Gestión de Claves/Ejemplo KeyStore.py"
Traceback (most recent call last):
  File "/home/kali/criptografia/criptografia-main/codigo fuente/Gestión de Claves/Ejemplo KeyStore.py", line 16, in <module>
    print("La clave es:", key.hex())
NameError: name 'key' is not defined
(vENV)-(kali㉿kali)-[~/criptografia/criptografia-main/codigo fuente/Gestión de Claves]
$ La clave es: af9df30474898787a45605ccb9b936d33b780d03cab81719d52383480dc3120
(vENV)-(kali㉿kali)-[~/criptografia/criptografia-main/codigo fuente/Gestión de Claves]
$ 

```

Ln 13, Col 47 Spaces: 4 UTF-8 LF () Python 3.13.9

Clave keystore:

af9df30474898787a45605ccb9b936d33b780d03cab81719d52383480dc3120

Last build: 4 months ago - Version 10 is here! Read about the new features [here](#)

Options About / Support

Recipe

ChaCha

Key 19d52383480dc3120	HEX	Nonce ff7f99c926102dd92	HEX
Counter 0	Rounds 20	Input Raw	Output Hex

Input
KeepCoding te enseña a codificar y a cifrar

Output
69 ac 4e e7 c4 c5 52 53 7a 00 a1 9b ca f7 f0 aa ed 7c ae 5f 37 d8 17 e3 97 c5 62 a2 e0 66 37 26 a0 78 45 d5
26 45 76 fa d0 bb 51

STEP Auto Bake

Mensaje cifrado:

69ac4ee7c4c552537a00a19bcacf7f0aaed7cae5f37d817e397c562a2e0663726a07845d
5264576fad0bb51

¿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.

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

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGI0byBkZSB

sb3MgcGFsb3RlcylsInJvbCI6ImlzTm9ybWFsliwiaWF0IjoxNjY3OTMzNTMzfQ.gfhw0

dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

¿Qué algoritmo de firma hemos realizado?

HS256

¿Cuál es el body del jwt?

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

Un hacker está enviando a nuestro sistema el siguiente jwt:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGI0byBkZSBsb3MgcGFsb3RlcylsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHURvmnAZdg4ZMeRNv2CIAODIHRI

¿Qué está intentando realizar?

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

Está intentando pasar el rol del usuario de normal a administrador.

¿Qué ocurre si intentamos validarla con pyjwt?

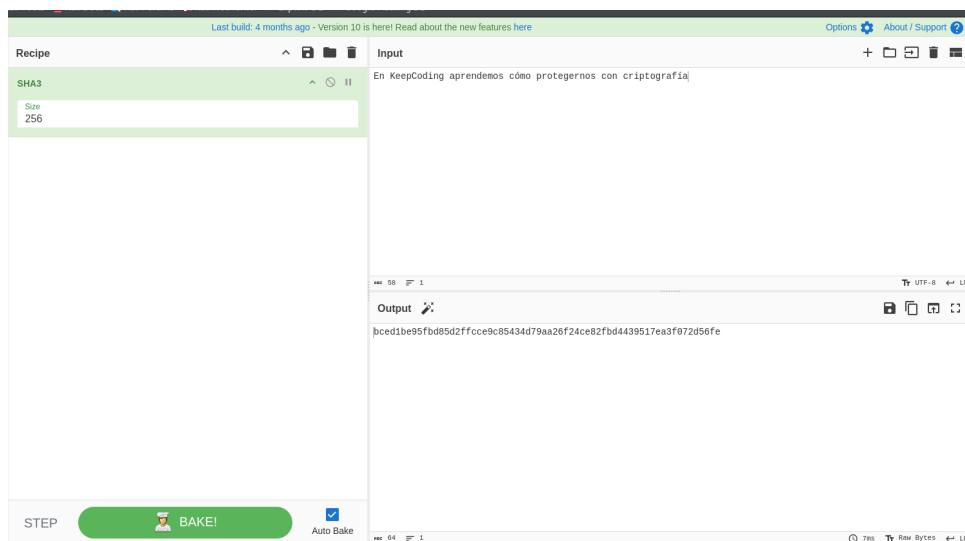
Da error ya que no conoce la clave.

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

bced1be95fb85d2ffcce9c85434d79aa26f24ce82fb4439517ea3f072d56fe

¿Qué tipo de SHA3 hemos generado?

Es un SHA3 de 256 bits.

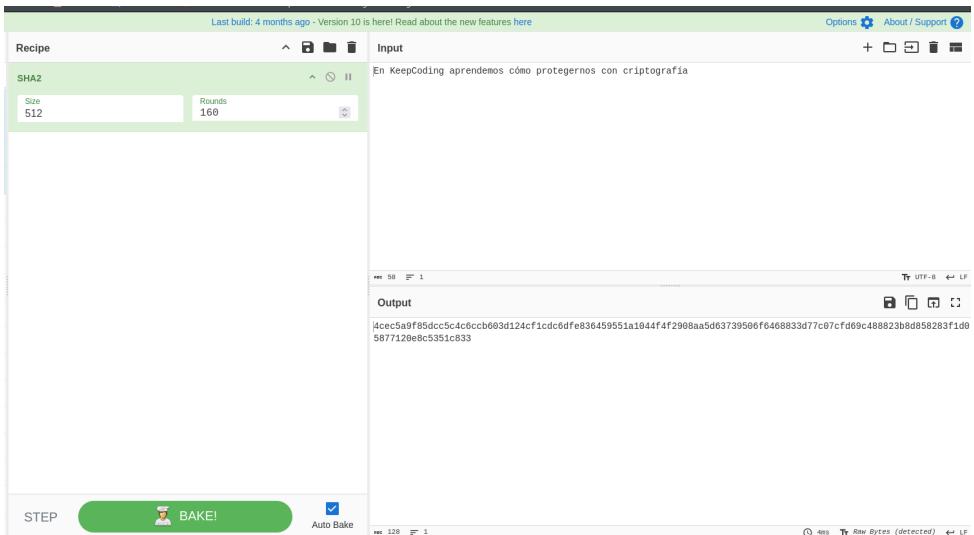


Y si hacemos un SHA2, y obtenemos el siguiente resultado:

**4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d6373950
6f6468833d77c07cf69c488823b8d858283f1d05877120e8c5351c833**

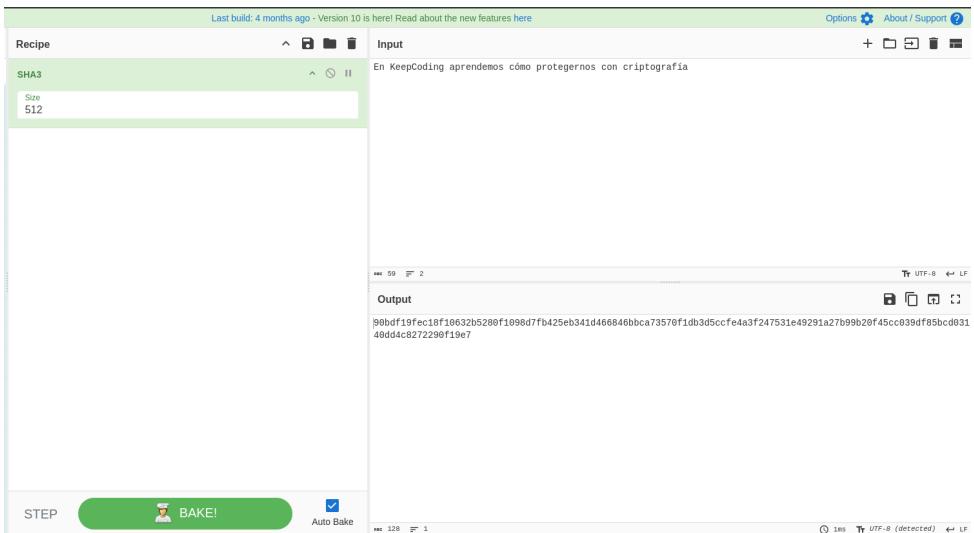
¿Qué hash hemos realizado?

Un SHA2 de 512 bits.



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?

Pues este texto es igual al anterior solo que tiene un punto al final, ese mínimo cambio hace que cambie por completo el hash, esta propiedad se llama “avalancha” que básicamente significa que cualquier cambio mínimo en el input hace que el output sea completamente diferente.

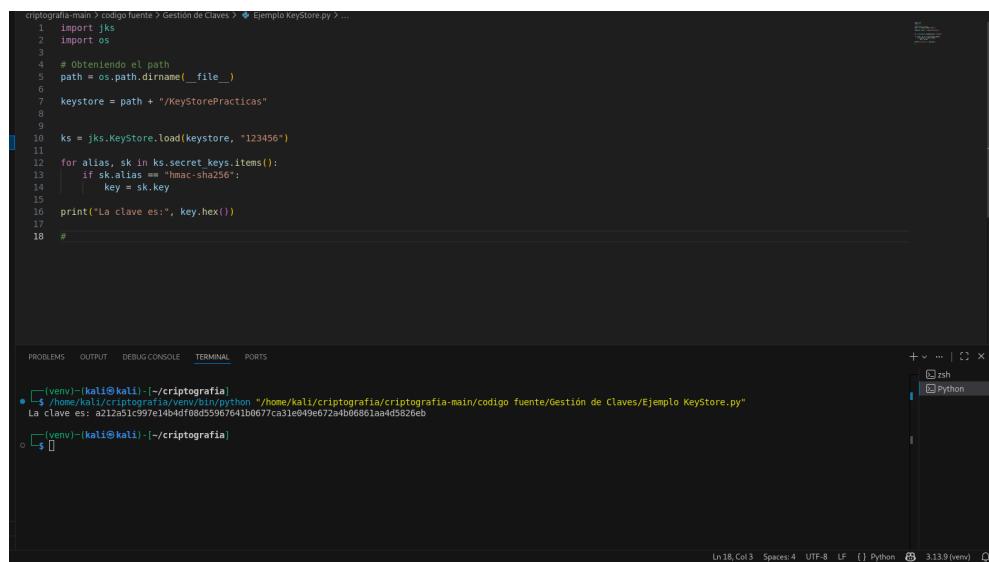


6. Calcula el hmac-256 (usando la clave contenida en el Keystore) del siguiente texto:

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.

HMAC: 857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550



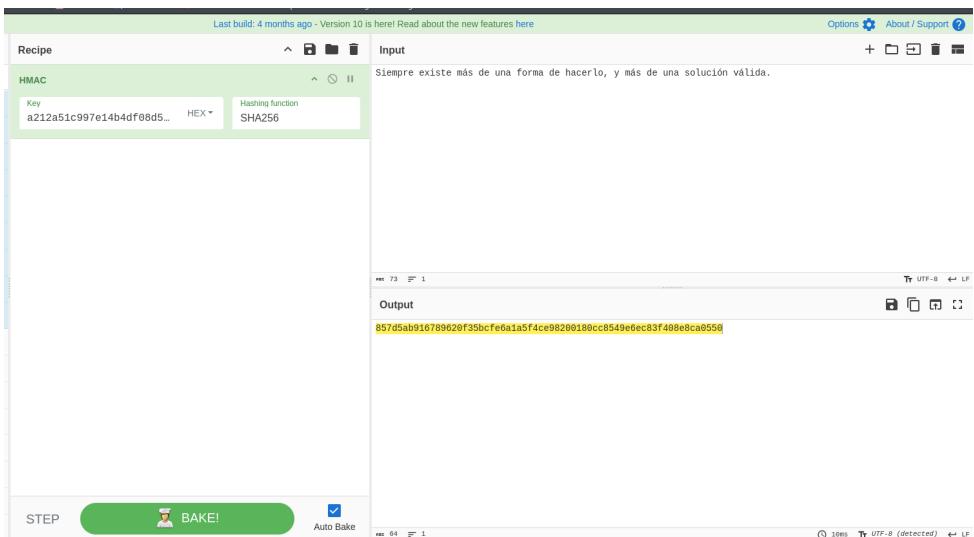
The screenshot shows a terminal window with the following content:

```
criptografia-main > código fuente > Gestión de Claves > Ejemplo KeyStore.py > ...
1 import jks
2 import os
3
4 # obteniendo el path
5 path = os.path.dirname(__file__)
6
7 keystore = path + "/KeystorePracticas"
8
9
10 ks = jks.KeyStore.load(keystore, "123456")
11
12 for alias, sk in ks.secret.keys.items():
13     if sk.alias == "hmac-sha256":
14         key = sk.key
15
16 print("La clave es:", key.hex())
17
18 #
```

TERMINAL

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
• [venv]-(kali㉿kali)-[~/criptografia]
$ /home/kali/criptografia/venv/bin/python "/home/kali/criptografia/criptografia-main/código fuente/Gestión de Claves/Ejemplo KeyStore.py"
La clave es: a212a51c997e14b4df08d55567641b0677ca31e649e672a4b06861aa4d5826eb
o [venv]-(kali㉿kali)-[~/criptografia]
o $
```

Ln 18, Col 3 Spaces: 4 UFT-8 LF [] Python 3.13.9 (venv) □



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?

El SHA-1 produce solo hashes de 160 bits, no se considera seguro y se encontraron ataques de colisión en el pasado.

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?

Usar Salt y Pepper evitando rainbow tables y creando un sistema complejo difícil de romper.

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

Usar Key Derivation Functions como PBKDF2, bcrypt, Argon2... que están diseñados para ser lentos, evitan ser vulnerados con fuerza bruta y son específicos para passwords.

8. Tenemos la siguiente API REST, muy simple. (Ver tabla en PDF original)

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?

Atendiendo a la petición del enunciado, necesitamos un algoritmo que nos asegure confidencialidad, y asimismo que mantenga la información de manera íntegra, sin cambios, para ello necesitaremos un MAC. El algoritmo que usaremos atendiendo a los trabajados será un AES/GCM que nos da el cifrado autenticado lo que nos aporta confidencialidad e integridad y es considerado el mejor mecanismo de bloque simétrico.

9. Se requiere calcular el KCV de las siguientes claves AES:

A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB

72

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.

KCV(AES): 5244db

KCV(SHA-256): 2ecdः

The screenshot shows the Cryptopp web interface with the following configuration:

- Recipe:** AES Encrypt
- Key:** F1ACFA0148FB3A426DB72 (HEX)
- IV:** 00000000000000000000000000000000 (HEX)
- Mode:** CBC
- Input:** Hex (empty)
- Output:** Hex (empty)

The **Input** field contains the value 00000000000000000000000000000000. The **Output** field displays the result: 5244dbd02d57d56ae08e64c56c7ca74a35eccad6db31f95841bde3d4e3ada4a.

At the bottom, there is a green button labeled **BAKE!** with a chef icon, and a checked checkbox for **Auto Bake**.

The screenshot shows the Cryptopp web interface with the following configuration:

- Recipe:** SHA2
- Size:** 256
- Rounds:** 64

The **Input** field contains the value A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72. The **Output** field displays the result: 2ecdःbc4a7156a631f169f1dcc2c4174d545d76a826d015882b0b4647837e4d.

At the bottom, there is a green button labeled **BAKE!** with a chef icon, and a checked checkbox for **Auto Bake**.

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.

```
(kali㉿kali)-[~/Desktop/Practica]
└─$ file MensajeRespoDeRaulARRHH.sig
cat MensajeRespoDeRaulARRHH.sig
MensajeRespoDeRaulARRHH.sig: PGP signed message
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA512

Se debe ascender inmediatamente a Raúl. Es necesario mejorarle sus condiciones económicas un 20% para que se quede con nosotros.
-----BEGIN PGP SIGNATURE-----

iJYEARYKAD4WIQQb3mNeTq5uaN+tL3zXML4ZbkZhAQUCYrhHNSAccGVkcm8ucGVk
cm10by5wZWRyb0BlbxByZXNhLmNvbQAKCRDXML4ZbkZhAcUcAP9G6wAIZDFlR1Q9
TCIJuLEHj6LTTRTNkATS4smub/+88gD+JlAZHDMhx3wThhfJq07TBmD+EBJNgv76
AEBhvVwVKgw=
=e8AU
-----END PGP SIGNATURE-----

(kali㉿kali)-[~/Desktop/Practica]
└─$ gpg --verify MensajeRespoDeRaulARRHH.sig
gpg: Signature made Sun 26 Jun 2022 07:47:01 AM EDT
gpg:                               using EDDSA key 1BDE635E4EAE6E68DFAD2F7CD730BE196E466101
gpg:                               issuer "pedro.pedrito.pedro@empresa.com"
gpg: Good signature from "Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.co
m>" [expired]
gpg: Note: This key has expired!
Primary key fingerprint: 1BDE 635E 4EAE 6E68 DFAD 2F7C D730 BE19 6E46 6101

(kali㉿kali)-[~/Desktop/Practica]
└─$ gpg --import RRHH-priv.txt
gpg: key 3869803C684D287B: public key "RRHH <RRHH@RRHH>" imported
gpg: key 3869803C684D287B: secret key imported
gpg: Total number processed: 1
gpg:                               imported: 1
gpg:                               secret keys read: 1
gpg:                               secret keys imported: 1

(kali㉿kali)-[~/Desktop/Practica]
└─$
```

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.

```
(kali㉿kali)-[~/Desktop/Practica]
└─$ gpg --clearsign --local-user F2B1D0E8958DF2D3BDB6A1053869803C684D287B menaje_RRHH.txt

(kali㉿kali)-[~/Desktop/Practica]
└─$ ls -la mensaje_RRHH.txt.asc
-rw-rw-r-- 1 kali kali 375 Dec  9 11:14 mensaje_RRHH.txt.asc

(kali㉿kali)-[~/Desktop/Practica]
└─$ cat mensaje_RRHH.txt.asc
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA512

Viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25%
su salario.
Saludos.
-----BEGIN PGP SIGNATURE-----
iHUEARYKAB0WIQTysdDolY3y0722oQU4aYA8aE0oewUCAThK9AAKCRA4aYA8aE0o
e3N7AQDo24rEPpHsu7ndH0lrlCkD6B4jjAbFTMc5oR8dMVOAFwEAnrtGYYnS04YO
TT2qeTwNMpMvw7uZP2nOvaDRjvCmWA0=
=5d3H
-----END PGP SIGNATURE-----
```

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.

```
(kali㉿kali)-[~/Desktop/Practica]
└─$ ls -la mensaje_cifrado.asc
-rw-rw-r-- 1 kali kali 537 Dec  9 11:27 mensaje_cifrado.asc

(kali㉿kali)-[~/Desktop/Practica]
└─$ cat mensaje_cifrado.asc
-----BEGIN PGP MESSAGE-----
hF4DfBpG6iCwVG8SAQdAlxpgbqGID18jQHH7wWNxbbKLZRRz3n3E47LJkqqGwhUw
kT+5RoH7bgTr7PoxyXuKyjGCGcehTlE6q/xyMAC6W3tQhXs+pY3e+GRUsdy19Mk1
hF4DJdbQKUA1tlASAQdAQkai0bKclHHi2e6ABV5czyY/2gohW7nAgN1uMiEo/V4w
2numL6/soShT9Wn0X7c8zytzZtGTj9lcNW01UzHtGzRzoIfVCvoUq47z2WWIY/zP
0p0BQkhinRGKKuosdkutAo2pkEVxR4SKU6kSa7v/BCzscHlt7RhupCg4tEQBzGPK
ahrq5SlauOvf+7QsFq0C8GCKqHzhh58eh3moJqyu+U705BmguPynIDA7dc+AzgxQ
CeZq47RhnLD32uiqrS0L+fiQH2HYb8BgsrhyWxRzp4Bu4fZkQN+jvIPhBJI6OU1S
TFBY31hH10pj0uTfBbGx
=lxII
-----END PGP MESSAGE-----
```

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:

b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30
c96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dfffa76a329d04e3d3d4ad6
29793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae2c4f5ba670a4126f2
f14a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8624071be78cce5
73d896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f67d3f98b4cd907f27639
f4b1df3c50e05d5bf63768088226e2a9177485c54f72407fdf358fe64479677d8296ad3
8c6f177ea7cb74927651cf24b01dee27895d4f05fb5c161957845cd1b5848ed64ed3b
03722b21a526a6e447cb8ee

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

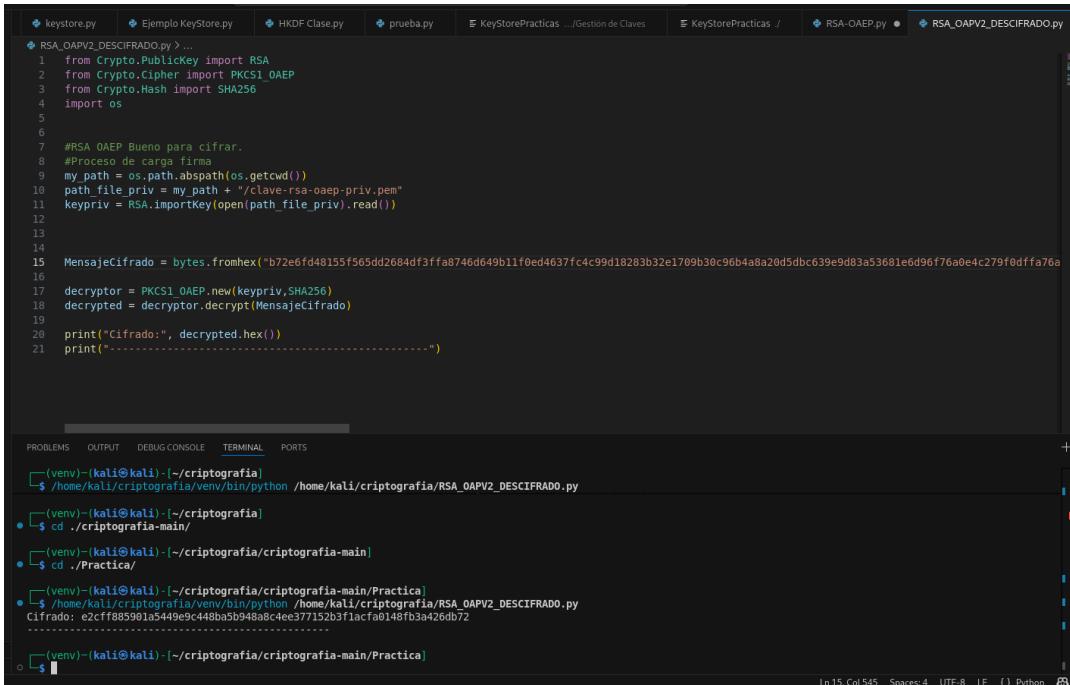
Si has recuperado la clave, vuelve a cifrarla con el mismo algoritmo. ¿Por qué son diferentes los textos cifrados?

Primero buscamos la clave privada:

```

zsh: corrupt history file /home/kali/.zsh_history
└─(kali㉿kali)-[~]
  └─$ cd Desktop
    └─(kali㉿kali)-[~/Desktop]
      └─$ cat clave-rsa-oaep-priv.pem
      BEGIN PRIVATE KEY
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggsJjAgEAAoIBAQC/absrLf79T7cz
tzjt/hHGJ+2LTBrZ90mJqTwctLU5x Cd9heffoiVQ+ZFZH5a1ewI3O5hPA16R/Ad
g63clqWY4iRp4JZt84Gw2XeLUR060VxlufQt1aC9oU0Qi1YksI1+LqNa6y5KOw
HqZFKoq+25EGkduNh9zAPevy1kVne/lFuJsvxtgjuNFN+WieCtq3M5fszieBM2ew
5HFHPLINKr5YpYTRkU80TmrN0R0iewSmrupaAk/hSL2ADUdmzraVqlLzqvJ763R79
c0+SmugoEEDEKlxK+xCE42vu9W00d9m2ZSEJgiVeV5yDCoOKzFyJnmhL6dKYFMuD
UU1K4oxTAgMBAEcgEAPkQGoyOisKLyK08QLyheortOmKj70CF8yU/SereQLD
T9KV3xjkOsJniX3iVz4cbLJg2Lfd+Z+/HQpUShg00cOGBBR/Y7M3PeKNYHQVHyBF
qbV7nCE5RbcJ2Bep3Ir+hM1N2WncOyIS2HZNMafGywRyRMaUKGo3Ah43b9dWhx
RYhLee8CD1c91llkR22UycmeJdgWe+Cmu0iFWH87+0F8cqVI+6z1KMk2IRh/HfVp
v646kOwKBF3XPT4YFjx+t2JSeLsAQBQR+aVq3Twyz354GvPvaVsON91FsToQbj+1
f0JRzleWb+8n/u60LNWL85j5zG2uQq2/K3KVeSYrPF7uHdAdCkMhRz9NB9WKwJk
T3cyYHFkxDj5S/s96HLcuUR89Sn7LQwkkZYwki/osm985e54/rbSop/1+beCo0
OxKpSozc8/xms3ulbwpxDxR3+xTMj9vAGjjp2hw5ylTH6o7Kyq8kYyPjmGnqNpCf
AnV7mfDL8Jvw/kIAEpwxEo6+HQKBgQDh4jLWek8PZDBUmzaJzzfIMddXMKX50u
dzhKOF2WS7WutJpYRbg4/sz3Ty6qtulDexuWnw+feEkroq8CMFB7FQaOPtq3nac
coWkTSEUG7Tz1R0318kslVVJ3Y93iSaoMtrThcaa18+FmVG3SwBefl0uEx8SpAvg
11P0+pfWkQKBgQCTDDwuUpoT4ZhaY2qrRGDLQ47vpT8E6cxeYoczypp+jxPcEl0YC
oIjetp+Wb+8n/u60LNWL85j5zG2uQq2/K3KVeSYrPF7uHdAdCkMhRz9NB9WKwJk
ZzV9LI3dbwqF9N+byW+ogZtHHKTbneSeoB+OEzoVzsys5R29fsMT3MWZQK8gAye
W/Kt+Kg1CboRpy2WHnxW28tmLHYXfsU8EH5L0St3dar0q7A16ll2UQQcBLHBvnZ/
ZAeodB/JoYNN+V5Gi0t3zSTiaHak02gCMRY7QJQBMMPdopnSpwv+1dM5jCvu4C
WPKR09AG6WKFrKknqURITbAxHabtMy57HtigZ/BaoGAdpmMRDQNkqai7aGombmF
Wy1GbLITkxWAOfScQQUYrFs8cu0Gu79aB7PHwzeOIHk/5Esj/gz7hoKJtogi4ikx
zG2lYqqe11/Gg6WhendR1qR8VrbLBkpqylFTGusmlBug7y4E/z9y2b4rMciU30Y
X230g/Q6y6kMprauaCuxNSk=
      END PRIVATE KEY
└─(kali㉿kali)-[~/Desktop/Practica]
  └─$
```

En segundo lugar hacemos la desencriptación con la clave privada obtenida y el mensaje proporcionado:



```

keystore.py EjemploKeyStore.py HKDF_Clase.py prueba.py KeyStorePracticas .../Gestión de Claves RSA-OAEP.py RSA_OAPV2_DESCIFRADO.py
RSA_OAPV2_DESCIFRADO.py > ...
1 from Crypto.Publickey import RSA
2 from Crypto.Cipher import PKCS1_OAEP
3 from Crypto.Hash import SHA256
4 import os
5
6
7 #RSA OAEP Bueno para cifrar.
8 #Proceso de carga firma
9 my_path = os.path.abspath(os.getcwd())
10 path_file_priv = my_path + "/clave-rsa-oaep-priv.pem"
11 keypriv = RSA.importKey(open(path_file_priv).read())
12
13
14
15 MensajeCifrado = bytes.fromhex("b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c964a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dfffa76a")
16
17 decryptor = PKCS1_OAEP.new(keypriv, SHA256)
18 decrypted = decryptor.decrypt(MensajeCifrado)
19
20 print("Cifrado:", decrypted.hex())
21 print("-----")
```

Clave simetrica:
e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

Ahora, obtenemos la clave pública:

```
(kali㉿kali)-[~/Desktop/Practica]
$ cat clave-rsa-oaep-publ.pem
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEAAQCAQ8AMIIIBCgKCAQEAv2m7Ky3+/U+3M7c47f4R
xifti0wa2fdJialU8ArS10cQnfYXpXzoLUPmRWR+WTx5CN0OYTwnNekfwHYt3Jal
m0IkaeCwbfOBhsNl3i1EU0tFTcZbn0LdWgvaFNEItWJLCNfi6jWusuSjsB6mRZKK
vtuRBpHbjYfcwD3r8tZFZ3v5X1CbMb7YI7jRTflongratz0X7M43gTNns0R3xzzy
DSq+WKWE0ZFPNE5qzdEdInsEppbqWgJP4Ui9gA1HZs62lai86rye+t0e/XDvkpro
KBBAxCpcSvsQhONr7vVtNHfZtmUhCYIlXlecgwqDisxciz5oS+nSmBTlg1FNSuDs
bQIDAQAB
-----END PUBLIC KEY-----
(kali㉿kali)-[~/Desktop/Practica]
```

Y volvemos a cifrarla otra vez:

```
keystore.py Ejemplo KeyStore.py HKDF Clase.py prueba.py KeyStorePracticas /Gestion de Claves RSA-OAEP.py RSA_OAPV2_DESCIFRADO.py
RSA_OAPV2_DESCIFRADO.py ...
4 import os
5
6
7 #RSA OAEP Bueno para cifrar.
8 #Proceso de carga firma
9 my_path = os.path.abspath(os.getcwd())
10 path_file_priv = my_path + "/clave-rsa-oaep-publ.pem"
11 keypub = RSA.importKey(open(path_file_priv).read())
12
13
14 clave_simetrica = bytes.fromhex("e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72")
15
16 #decryptor = PKCS1_OAEP.new(keypriv,SHA256)
17 #decrypted = decryptor.decrypt(MensajeCifrado)
18
19 encryptor = PKCS1_OAEP.new(keypub, SHA256)
20 cifrado_nuevo = encryptor.encrypt(clave_simetrica)
21
22
23
24 print("Cifrado:", cifrado_nuevo.hex())
25 print("-----")
```

```
[venv] -> /home/kali/criptografia/venv/bin/python /home/kali/criptografia/prueba.py
[venv] -> /home/kali/criptografia/venv/bin/python /home/kali/criptografia/RSA_OAPV2_DESCIFRADO.py
Traceback (most recent call last):
  File "/home/kali/criptografia/RSA_OAPV2_DESCIFRADO.py", line 24, in <module>
    print("Cifrado:", encryptor.hex())
AttributeError: 'PKCS1OAEP_Cipher' object has no attribute 'hex'
[venv] -> /home/kali/criptografia/criptografia-main/Practica
[venv] -> $ /home/kali/criptografia/venv/bin/python /home/kali/criptografia/RSA_OAPV2_DESCIFRADO.py
Cifrado: 598b5fd9e5a8c2863f9893608d7d2ed1f6f314a1b0920e414d5f951e54f266a0e1f943c56e421a6ff58a56eb254a639b3e641485be932d59ecc60f0832a5f6be340fc4db39bba3f0df302b824d5918a66
318f9215d520a7cb95d64dd48f5fe41fb162e2fd777c1l72d5d70a4489ec571fd1a0c1f8029aa753e172a54eba157db8906987b3b299988a361e7869523c96fd59f0098a549c4bca8f9440849a857e636f
cebe3140e0f97bb076b67a8b4e5ede2161538188041babe93695f8741a5fda050b62315e8744e819622973e4e9f2d947fe9d794c2566fc4eede852a2b5dadff247c1db7a60ba2f19b62b14a028a1ebdea3207f34d1
679edbba01c
-----
```

Cifrado obtenido:
598b5fd9e5a8c2863f9893608d7d2ed1f6f314a1b0920e414d5f951e54f266a0e1f943c5
6e421a6ff58a56eb254a639b3e641485be932d59ecc60f0832a5f6be340fc4db39bba3f0
df302b824d5918a66318f9215d520ac7b95db64dd48f5fed41fb9162e2ffd777c1172d55

d70a489ec571fd1a0c1f8020aa753e172a54aeba157dbb8906907b3b209980a361e706
9523c96fdc59f0098a549e4bca8f9440849a0857e636fcebe3140e0f97bb076b67a8b4e
5ede216153818804b1abe93695f8741a5fda050b62315e8744e819622973e4e9f2d947f
e9d794c2566fc4eede852a2b5dadff247c1db7a60ba2f19b62b14a028a1ebdea3207f34
d1679edbaa91c

Los cifrados son distintos ya que RSA-OAEP usa padding aleatorio y aun cifrando con las mismas claves el resultado que se obtiene es distinto.

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:E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A42
6DB74**

Nonce:9Yccn/f5nJJhAt2S

¿Qué estamos haciendo mal?

Fundamentalmente, lo que se está haciendo mal es usar la misma clave para todas las comunicaciones ya que si la clave se ve comprometida toda la comunicación con el tercero se puede ver expuesta. Lo mismo pasa con el nonce, debería ser aleatorio en cada mensaje.

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.

Hexadecimal:

78b686aaf4ee8acef90dad6209d4e8ce784970517c14976fc4385305d672b81009bcf18
907193d0a56a30a0dfa9937fea0e3e

Base

64:

eLaGqvTuis75Da1iCdTozhJcFF8FJdvxDhTBdZyuBAJvPGJBxk9ClajCg3+qZN/6g4+

The screenshot shows a terminal window with two tabs: '3des-cbc.py' and 'aes-gcm-encrypt.py'. The 'aes-gcm-encrypt.py' tab is active, displaying the following Python code:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

# AES-GCM --> (Datos Asociados + Datos a cifrar) + key + nonce

texto_gcm_bytes = bytes("He descubierto el error y no volveré a hacerlo mal", "utf-8")
key_bytes = bytes.fromhex('E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74')
nonce_bytes = get_random_bytes(8)
print("Nonce hex=", nonce_bytes.hex())
datos_asociados_bytes = bytes("Ejercicio 12", "utf-8")
cifrador = AES.new(key_bytes, AES.MODE_GCM, nonce=nonce_bytes)
cifrador.update(datos_asociados_bytes)
texto_cifrado_bytes, mac_bytes = cifrador.encrypt_and_digest(texto_gcm_bytes)
print("Texto cifrado:", texto_cifrado_bytes.hex())
print("MAC:", mac_bytes.hex())
```

Below the code, the terminal shows the command being run and its output:

```
source /home/bruno/criptografia/.venv/bin/activate
● bruno@bpernambucano:~/criptografia $ source /home/bruno/criptografia/.venv/bin/activate
● (.venv) bruno@bpernambucano:~/criptografia $ python aes-gcm-encrypt.py
Nonce hex= 39561c9c7bf97b9c9121d2d59a9121d2
Texto cifrado: 78b686aaaf4ee8acef90dad6209d4e8ce784970517c14976fc4385305d672b81009bcf18907193d0a56a30a0dfea9937fea0e3e
MAC: e41194911688850109f0738cdd9b7e24
○ (.venv) bruno@bpernambucano:~/criptografia $
```

The screenshot shows the Cryptool online tool interface. On the left, under 'Recipe', the 'AES Decrypt' recipe is selected. It has fields for 'Key' (HEX: E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74), 'IV' (HEX: 39561c9c7bf97b9c9121d2d59a9121d2), 'Mode' (GCM), 'Input' (Hex), 'Output' (Raw), 'GCM Tag' (HEX: e41194911688850109f0738cdd9b7e24), and 'Additional Authenticated Data' (UTF8: Ejercicio 12). On the right, under 'Input', the encrypted text '78b686aaaf4ee8acef90dad6209d4e8ce784970517c14976fc4385305d672b81009bcf18907193d0a56a30a0dfea9937fea0e3e' is pasted. Under 'Output', the decrypted text 'He descubierto el error y no volveré a hacerlo mal' is shown.

The screenshot shows the Cryptool interface with the following details:

- Recipe:** From Base64
- Alphabet:** A-Za-zA-Z0-9+=
- Remove non-alphabet chars:** Checked
- Strict mode:** Unchecked
- To Hex:** Delimiter: Space, Bytes per line: 0
- AES Decrypt:**
 - Key:** E2CFF885901B344...
 - IV:** 39561c9c7bf97b9...
 - Mode:** GCM
 - Input:** Hex
 - Output:** Raw
 - GCM Tag:** e41194911688850...
 - Additional Authenticated Data:** Ejercicio 12
 - UTF8:** HEX
- Output:** The decrypted message: "He descubierto el error y no volveré a hacerlo mal!"

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?

Primero obtenemos la firma:

```

... implo KeyStore.py   HKDF Clase.py   prueba.py   KeyStorePracticas .../Gestión de Claves   KeyStorePracticas / RSA-OAEP.py   RSA_OAPV2_DESCIFRADO.py   RSA-Firma-pkcs1v5.py ...
criptografia-main > codigo fuente > criptografia asimetrica e hibrida > RSA-Firma-pkcs1v5.py > ...
1  from Crypto.PublicKey import RSA
2  from Crypto.Signature.pkcs1_19 import PKCS115_SigScheme
3  from Crypto.Hash import SHA256
4  import binascii
5  import os
6
7  #Cargamos la clave PRIVADA porque generaremos una firma
8  my_path = os.path.abspath(os.getcwd())
9  path_file_priv = my_path + '/clave-rsa-oaep-priv.pem'
10 keypriv = RSA.importKey(open(path_file_priv).read())
11
12 mensaje_bytes = bytes("El equipo está preparado para seguir con el proceso, necesitaremos más recursos.", "utf-8")
13 hash = SHA256.new(mensaje_bytes)
14
15 firmador=PKCS115_SigScheme(keypriv) ## Generamos un Signer
16 firma = firmador.sign(hash)
17 print("Firma: ", firma.hex())
18

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

(venv)-[kali㉿kali:~/criptografia]
$ /home/kali/criptografia/venv/bin/python "/home/kali/criptografia/criptografia-main/codigo fuente/criptografia asimetrica e hibrida/RSA-Firma-pkcs1v5.py"
• $ cd ./criptografia
• (venv)-[kali㉿kali:~/criptografia/criptografia-main]
• $ cd ./Practica
• (venv)-[kali㉿kali:~/criptografia/criptografia-main/Practica]
• $ /home/kali/criptografia/venv/bin/python "/home/kali/criptografia/criptografia-main/codigo fuente/criptografia asimetrica e hibrida/RSA-Firma-pkcs1v5.py"
Firma: a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c
6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998
ef08b2cb3a925c959bc当地2ca9e6e60f95b989c709b9a0b98a0c69d9eacc0d863bc924e
70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d
1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8954
ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d
92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21f0
aaaa6f9b9d59f41928d

```

Ln 18, Col 1 Spaces: 4 UTF-8 LF (Python 3.13.9 (venv))

Firma:

a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c
6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998
ef08b2cb3a925c959bc当地2ca9e6e60f95b989c709b9a0b98a0c69d9eacc0d863bc924e
70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d
1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8954
ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d
92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21f0
aaaa6f9b9d59f41928d

Calcula la firma (en hexadecimal) con la curva elíptica ed25519, usando las claves ed25519priv y ed25519-publ.

Firma:

bf32592dc235a26e31e231063a1984bb75ffd9dc5550cf30105911ca4560dab52abb40e
4f7e2d3af828abac1467d95d668a80395e0a71c51798bd54469b7360d

```

# 4. Mensaje
msg = "El equipo está preparado para seguir con el proceso, necesitaremos más recursos."
msg_bytes = msg.encode('utf-8')

# 5. Firmar (FORMA CORRECTA en cryptography)
signature = private_key.sign(msg_bytes)

# 6. Mostrar firma
print(f"\n+FIRMA GENERADA (64 bytes en hexadecimal):{signature.hex()}")
print(f"Longitud: {len(signature)} bytes")

# 7. Verificar (FORMA CORRECTA en cryptography)
try:
    public_key.verify(signature, msg_bytes)
    print("La firma es válida (verificación interna)")
except Exception as e:
    print(f"Error en verificación: {e}")

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

(venv)-(kali㉿kali)-[~/criptografia/criptografia-main/Practica]
$ python ed25519_adaptado.py
Tomando primeros 32 bytes de 64 totales
Clave privada (32 bytes): c4225628eb2fcf184e90bade9b7fd8596f372473b1b9fb62cc975aacf34cbc
Clave pública: 7cb0d471d56e8aeff768b99d025e19df3b17f127e6d440f91c1fe643ac1ebe961

**FIRMA GENERADA (64 bytes en hexadecimal):**
bf3292dc235a2631e231063a1984bb7ffff9dc5550cf30105911ca4560dab52abb40e4f7e2d3af828abac1467d95d668a80395e0a71c51798bd54469b7360d
Longitud: 64 bytes
La firma es válida (verificación interna)

(venv)-(kali㉿kali)-[~/criptografia/criptografia-main/Practica]
$ 

```

Ln 40, Col.41 Spaces:4 UTF-8 () Python

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:

e43bb4067cbc fab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3

¿Qué clave se ha obtenido?

Clave	de	cifrado:
7c7ee3c565ea0188e1087ded09af3318ae0af0dbb781cc4020cff5b446b2fe71		

Clave MAC:

c5fa2f860641f26a2abe1bb22798b7b12c4211884ccaed2a5ec77e6052b65fc7

```

criptografia-main > codigo fuente > Gestión de Claves > Ejemplo KeyStore.py > ...
1 import jks
2 import os
3
4 # Obteniendo el path
5 path = os.path.dirname(__file__)
6
7 keystore = path + "/KeyStorePractices"
8
9
10 ks = jks.KeyStore.Load(keystore, "123456")
11
12 for alias, sk in ks.secret.keys.items():
13     if sk.alias == "clifaa05-1m-aes-256":
14         key = sk.key
15
16 print("La clave es:", key.hex())
17
18 #

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

zsh: corrupt history file /home/kali/.zsh_history
source /home/kali/criptografia/venv/bin/activate
[kali㉿kali]:~/criptografia]\$ source /home/kali/criptografia/venv/bin/activate
[venv]-(kali㉿kali):~/criptografia]\$ /home/kali/criptografia/venv/bin/python "/home/kali/criptografia-main/codigo fuente/Gestión de Claves/Ejemplo KeyStore.py"
La clave es: a2cff885901a5449e9c448b5b948a8c4ee377152b3f1acfa0148fb3a426db72
[venv]-(kali㉿kali):~/criptografia]\$

```

criptografia-main > codigo fuente > Hashing y Authentication > HKDF Clase.py > ...
1 from Crypto.Protocol.KDF import HKDF
2 from Crypto.Hash import SHA256
3 import secrets
4
5 salt = bytes.fromhex('e43bb4067cbcfa3becf4437b84be4623e345682d89de9948fbba0afedc461a3') # Mi identificador
6 master_secret = bytes.fromhex('a2cff885901a5449e9c448b5b948a8c4ee377152b3f1acfa0148fb3a426db72') # Mi clave maestra
7 key1, key2 = HKDF(master_secret, 32, salt, SHA256, 2)
8
9 print("Clave key1: ", key1.hex()) # Clave de cifrado
10 print("Clave key2: ", key2.hex()) # Clave de MAC

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

[kali㉿kali]:~/criptografia]\$ source /home/kali/criptografia/venv/bin/activate
[venv]-(kali㉿kali):~/criptografia]\$ /home/kali/criptografia/venv/bin/python "/home/kali/criptografia-main/codigo fuente/Gestión de Claves/Ejemplo KeyStore.py"
La clave es: a2cff885901a5449e9c448b5b948a8c4ee377152b3f1acfa0148fb3a426db72
[venv]-(kali㉿kali):~/criptografia]\$
[venv]-(kali㉿kali):~/criptografia]\$ /home/kali/criptografia/venv/bin/python "/home/kali/criptografia-main/codigo fuente/Hashing y Authentication/HKDF Clase.py"
Clave key1: 7ce03c05ea0188e1087ed09af318ea0f8db781cc4020cff5446c2fe71
Clave key2: c5fa7f66641f26a2a0el3b2279887b12c4211884ccae2a3ec77e6852b65fc7
[venv]-(kali㉿kali):~/criptografia]\$

**15. Nos envían un bloque TR31:
D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDB
E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E0
3CD857FD37018E111B**

**Donde la clave de transporte para desenvolver (unwrap) el bloque es:
A1A10101010101010101010101010102**

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

'D' | Key block protected using the AES Key Derivation Binding Method. | AES

¿Para qué algoritmo se ha definido la clave?

A: AES.

¿Para qué modo de uso se ha generado?

'B' | Both Encrypt & Decrypt / Wrap & Unwrap

¿Es exportable?

S: Sensitive, exportable under untrusted key

¿Para qué se puede usar la clave?

'D0' | Symmetric Key for Data Encryption | 'B', 'D', 'E'

¿Qué valor tiene la clave?

'0' | '0' | Key versioning is not used for this key.