Bootcamp Ciberseguridad
Práctica Criptografía

Carlos Gutiérrez Torrejón

Documentación adjunta:

Los archivos con el código utilizado para resolver las prácticas, claves, keystore y demás archivos necesarios se encuentran en el mismo repositorio que este PDF. Específicamente en este enlace: https://github.com/AtajoDeLocos/Main/tree/main

Ejercicio 1

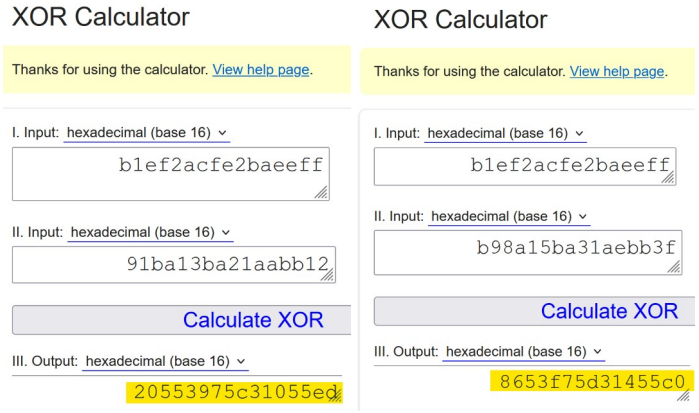
Tenemos un sistema que usa claves de 16 bytes. Por razones de seguridad vamos a proteger la clave de tal forma que ninguna persona tenga acceso directamente a la clave. Por ello, vamos a realizar un proceso de disociación de la misma, en el cuál tendremos, una clave fija en código, la cual, sólo el desarrollador tendrá acceso, y otra parte en un fichero de propiedades que rellenará el Key Manager. La clave final se generará por código, realizando un **XOR** entre la que se encuentra en el properties y en el código.

1. 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?

   Clave FIJA (código): B1EF2ACFE2BAEEFF
   Clave Key Manager (desarrollo): 20553975c31055ed → RESPUESTA
   Clave FINAL (desarrollo): 91BA13BA21AABB12

2. 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?

   Clave FIJA (código): B1EF2ACFE2BAEEFF
   Clave Key Manager (producción): B98A15BA31AEBB3F
   Clave FINAL (producción): 8653f75d31455c0 → RESPUESTA

- El código utilizado se puede encontrar en el archivo "**E1_Claves_XOR.py**"
- También se ha utilizado la siguiente herramienta online https://xor.pw/
  - 

Ejercicio 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=
```

1. 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.“

2. ¿Qué ocurre si decidimos cambiar el padding a x923 en el descifrado?
   - No pasa nada, lo descifra correctamente.
   - Al llevar sólo un byte de relleno (padding), este coincide en ambos tipos (01).

3. ¿Cuánto padding se ha añadido en el cifrado?
   - Se ha añadido 1 byte (01)

```
>>>>> Relleno:
4573746f2065657320756e206369667261646f20656e20626c6f7175652074c3ad7069636f2e20526563756572646612c207661732070f722065
6c206275656e2063616d696e6f2e20c3816e696d6f2e01
>>>>> Relleno:  b'Esto es un cifrado en bloque t\xc3\xadpico. Recuerda, vas por el buen camino. \xc3\x81nimo.\x01'
```

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

- El código utilizado esta en los archivos:
  - **E2_Descifrado_AES_PKCS7.py**
  - **E2_Descifrado_AES_X923.py**

Ejercicio 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.

- Texto cifrado en hex: `d86c1e07244508bf023b700d49f10343fc12f284537904dc3cebd675786b13a62549aca8de413a17462ebd8f`
- Código en archivo **E3_Cifrado_Chacha20.py**

¿Cómo podríamos mejorar de forma sencilla el sistema, de tal forma, que no sólo garanticemos la confidencialidad sino, además, la integridad del mismo?

- Podemos añadir un MAC (TAG?) para poder verificar la procedencia e integridad de los datos
  - **Chacha20-Poly1305**

Se requiere obtener el dato cifrado, demuestra tu propuesta por código, así como añadir los datos necesarios para evaluar tu propuesta de mejora.

- Criptograma: **4ec95921ca8b757e2336605c7dbab8f4d40b5b4d220e66aa978f740d3e59b0cd70e3217242991cc140bb1e1a**
- MAC: **de5434f2375b6f93fdcbcf02e54c0478**

- La propuesta se comprueba en el archivo **E3_Cifrado_Chacha20-Poly1305.py**
  - Como datos asociados se usa el usuario: *carlos.gutierrez*

Ejercicio 4

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

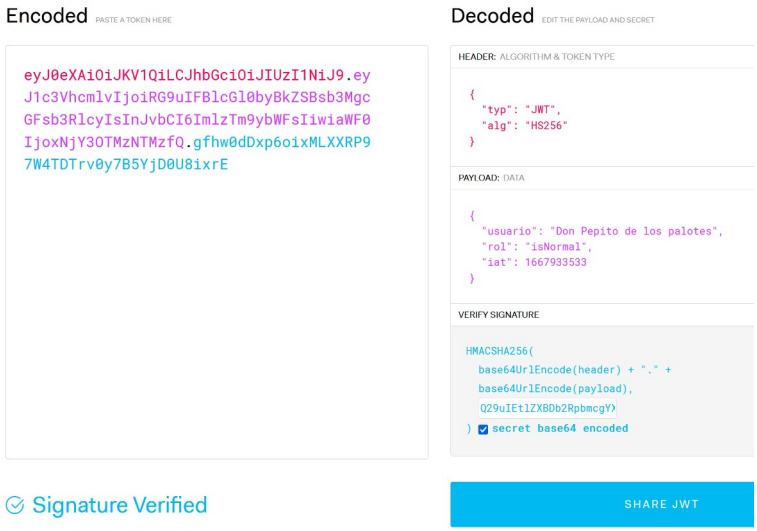
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE

¿Qué algoritmo de firma hemos realizado?

- HMAC-SHA256

¿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.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2ClAODlHRI

¿Qué está intentando realizar?

Intenta colarse con un perfil admin

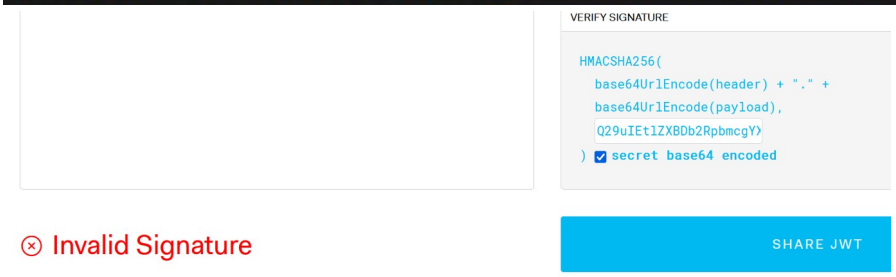
```
{
  "usuario": "Don Pepito de los palotes",
  "rol": "isAdmin",
  "iat": 1667933533
}
```

```
--------------------
{'usuario': 'Don Pepito de los palotes', 'rol': 'isNormal', 'iat': 1667933533}
{'usuario': 'Don Pepito de los palotes', 'rol': 'isAdmin', 'iat': 1667933533}
--------------------
```

¿Qué ocurre si intentamos validarlo con pyjwt?

No reconoce la firma y devuelve un error

```
    raise InvalidSignatureError("Signature verification failed")
jwt.exceptions.InvalidSignatureError: Signature verification failed
```



- La verificación de la firma y el resto se puede ver en el código **E4_jwt.py**

Ejercicio 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?
SHA3-256

- La clave es de 32bytes
  - 32 * 8 = 256

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

4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833

¿Qué hash hemos realizado?
SHA512

Genera ahora un SHA3 Keccak de 256 bits (sólo 256) con el siguiente texto: “En KeepCoding aprendemos cómo protegernos con criptografía.”
¿Qué propiedad destacarías del hash, atendiendo a los resultados anteriores?

- Hash generado: **302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf**
- Se que es más resistente a colisiones, pero no se si van por ahí los tiros.

Ejercicio 6

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

Siempre existe más de una forma de hacerlo, y más de una solución válida.

Se debe evidenciar la respuesta. Cuidado si se usan herramientas fuera de los lenguajes de programación, por las codificaciones es mejor trabajar en hexadecimal.

RESPUESTA → 857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550

Se evidencia en el archivo **E6_HMAC-256.py**

Ejercicio 7

Trabajamos en una empresa de desarrollo que tiene una aplicación web, la cual requiere un login y trabajar con passwords. Nos preguntan qué mecanismo de almacenamiento de las mismas proponemos.

Tras realizar un análisis, el analista de seguridad propone un hash SHA-1. Su responsable, le indica que es una mala opción. ¿Por qué crees que es una mala opción?

- El SHA1 es un HASH ya vulnerado.
  - Hereda de MD5
- El NIST recomienda usar SHA2 o 3 en su lugar.

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?

- **Salt**
  - Es un valor aleatorio que se agrega al mensaje antes de calcular el hash

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

- **Argon2 es la mejor opción**
  - Resiste mejor a los ataques de fuerza bruta y tablas.

Ejercicio 8

Tenemos la siguiente API REST, muy simple.
Request:
Post /movimientos

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

Petición de ejemplo que se desea enviar:
{"idUsuario":1,"usuario":"José Manuel Barrio Barrio","tarjeta":4231212345676891}
Response:

Campo	Tipo	Confidencialidad	Observaciones
idUsuario	Number	N	Identificador
movTarjeta	Array	S	Formato del Ejemplo
Saldo	Number	S	Tendrá formato 12300 para identificar 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?

1. **AES**: Advanced Encryption Standard (AES). Se considera seguro y ayuda a proteger la confidencialidad de los datos
2. **RSA**: Protege la confidencialidad y también la integridad de los datos.
3. **HMAC**: Fortalece la integridad de los datos usando una firma digital.

Ejercicio 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.

- KCV AES: **5244db**

- texto_cifrado con padding:
  - **5244dbd02d57d56ae08e064c56c7ca74a35eccad6db31f05841bde3d4e3ada4a**

- KCV SHA256: **db7df2**

Para este ejercicio he ejecutado el código **E9_KCV.py** (aunque no he terminado de eentenderlo muy bien, espero comentarios o tutorias).

Ejercicio 10

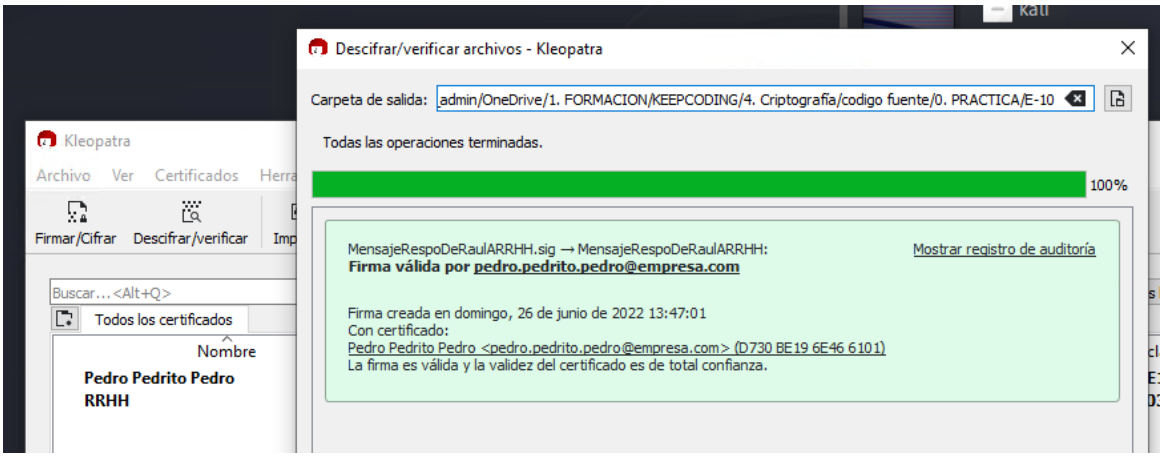
El responsable de Raúl, Pedro, ha enviado este mensaje a RRHH:

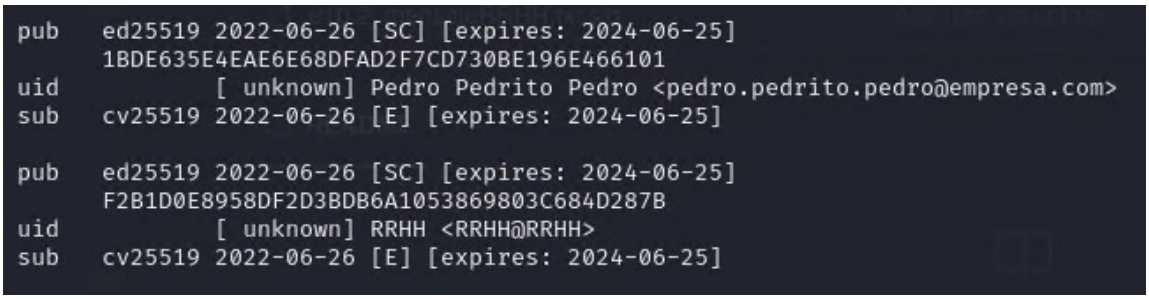
Se debe ascender inmediatamente a Raúl. Es necesario mejorarle sus condiciones económicas un 20% para que se quede con nosotros.

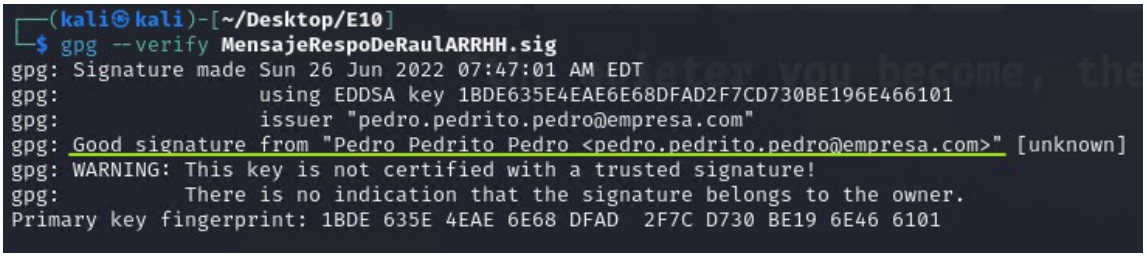
Lo acompaña del siguiente fichero de firma PGP (MensajeRespoDeRaulARRHH.txt.sig). Nosotros, que pertenecemos a RRHH vamos al directorio a recuperar la clave para verificarlo. Tendremos los ficheros Pedro-priv.txt y Pedro-publ.txt, con las claves privada y pública.

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.

- La firma se ha verificado
  - 

  - 

  - 

- **Se ha utilizado Kali Linux y el software Kleopatra para verificar la firma**

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.

- Se ha firmado el mensaje simulando pertenecer al departamento de RRHH

```
┌──(kali㉿kali)-[~/Desktop/E10]
└─$ cat e10.2_mensajeRRHH.txt.sig
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

Viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario. Sal
udos.
-----BEGIN PGP SIGNATURE-----

iIAEARYIACgWIQTysdDolY3y0722oQU4aYA8aE0oewUCZY4WggoccnJoaEBycmho
AAoJEDhpgDxoTSh7O88BAL744HHhtwkW2ldetvcYbqsLVTfhbzaXcxH8BCpc/A0m
AQDNU+htV5JWFH6RXEO2HgFzTWYkTvZLyHFBPCrvb7j+Dw==
=Gos3
-----END PGP SIGNATURE-----
```

- En los archivos E10.2_mensajeRRHH.txt.sig y E10.2_mensajeRRHH.txt.sig se evidencias el archivo firmado y cifrado,respectivamente (por duda en el enunciado).

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 los archivos **E10.3_mensajeCifradoRRHHPedro.txt** y **E10.3_mensajeCifradoRRHHPedro.txt.gpg** están los archivos original y cifrado.

Ejercicio 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:

b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c96b4a8a20d5dbc639e
9d83a53681e6d96f76a0e4c279f0dffa76a329d04e3d3d4ad629793eb00cc76d10fc00475eb76bfbc12733038
82609957c4c0ae2c4f5ba670a4126f2f14a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de86240
71be78ccef573d896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f67d3f98b4cd907f27639f4b1df3c50
e05d5bf63768088226e2a9177485c54f72407fdf358fe64479677d8296ad38c6f177ea7cb74927651cf24b01d
ee27895d4f05fb5c161957845cd1b5848ed64ed3b03722b21a526a6e447cb8ee

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

- Clave obtenida:
  - **e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72**
- Para esta parte se ha utilizado el código **E11_Descifrado_RSA-OAEP.py**

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

- Al volver a cifrar obtenemos otro resultado y cada vez que ciframos obtenemos un resultado distinto.
  - **537a435095d17d83d43cffd87b968da0af4165b5919da26d8af6366db8fecd8d91c5e8db15ab5a23ecd914da3ea28f3438013658bc17074c6296710c441af82232a0e3d3b5eb723918a48f3effa0c74a3f7ed965c12aa7d7ec456779b0397713c1b9fc6a141a1f958a9810ef44cd07cd82e10a040e880e392697f254eefaa5dc317f051d764c2c3feedd6732fa3fc20c66df1af3502308ea9ca2246d91d0e56bafa7926084014ce71d8f0024f6ac2800d4de675f3a843be9243aed133d589df18a7fbcc4108207483e771643b2170a79e8ebfda6d90f6f8a5e4d9ab4c59a7a528270932b94050a62fa92dde6414c8a5b0a5571dea93ff92ce4968ccd3e88df58**
- RSA-OAEP utiliza un esquema de relleno aleatorio que agrega una capa adicional de seguridad al cifrado. Por lo tanto, si se cifra el mismo mensaje dos veces con la misma clave pública, los resultados cifrados serán diferentes.
- Para cifrar se ha usado el siguiente código **E11_Recifrado_RSA-OAEP.py**

Ejercicio 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?

- Faltan los datos asociados (**TAG**)
- El nonce ya se ha utilizado antes en esta práctica, debería ser único por 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.

- Como datos asociados se usa el usuarios: *carlos.gutierrez*
- **HEX**: 5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d
- **BASE64**: Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq92dhLum64MHCV9QePv8FiVt
- El código utilizado está en el archivo **E12_Cifrado_AES-GCM.py**

Ejercicio 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?

- Código en archivo **E13_Firma_PKCS1v15.py**
- Firma generada
  - a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d721a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe031853277569f8383846d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af096ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b9519391b61891bf5e06699aa0a0dbae21f0aaaa6f9b9d59f41928d

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

- Código en archivo **E13_Firma_ED25519.py**
- Firma generada:
  - 6266333235393264633233356132366533316532333130363361313938346262373566666439646335353530636633303130353931316361343536306461623532616262343065346637653264336166383238616261633134363764393564363638613830333935653061373163353137393862643534343639623733363064

Ejercicio 14

Necesitamos generar una nueva clave AES, usando para ello una HKDF (HMAC-based Extract-and-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 obtenida
  - e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a
- Código en archivo **E14_Generar_clave_HKDF_S512.py**

Ejercicio 15

Nos envían un bloque TR31:

D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9E
E1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37018E111B

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

A1A10101010101010101010101010102

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

- (D) Método de vinculación derivada de clave 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
  - SI

¿Para qué se puede usar la clave?

- (D0) Symmetric Key for Data Encryption
  - B: Both Encrypt & Decrypt / Wrap & Unwrap
  - D: ?
  - E: Encrypt / Wrap Only

¿Qué valor tiene la clave?

- c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1

- Para este ejercicio se han utilizado los siguientes archivos de codigo **E15_tr31.py** y **E15_tr31_import.py**
- Y a la siguiente documentación: https://github.com/knovichikhin/psec/blame/master/psec/tr31.py#L200