PRÁCTICA: CRIPTOGRAFÍA ANDRES NAVARRETE

Objetivo:

Contestar un conjunto de preguntas / ejercicios relacionados con la materia aprendida en el curso demostrando la adquisición de conocimientos relacionados con la criptografía.

Detalles:

En esta práctica el alumno aplicará las técnicas y utilizará las diferentes herramientas vistas durante el módulo.

Cualquier password que sea necesaria tendrá un valor 123456.

Evaluación

Es obligatorio la entrega de un informe para considerar como APTA la práctica. Este informe ha de contener:

- Los enunciados seguido de las respuestas justificadas y evidenciadas.
- En el caso de que se hayan usado comandos / herramientas también se deben nombrar y explicar los pasos realizados.

El código escrito para la resolución de los problemas se entrega en archivos separados junto al informe.

Se va a valorar el proceso de razonamiento aunque no se llegue a resolver completamente los problemas. Si el código no funciona, pero se explica detalladamente la intención se valorará positivamente.

El objetivo principal de este módulo es adquirir conocimientos de criptografía y por ello se considera fundamental usar cualquier herramienta que pueda ayudar a su resolución, demostrando que no sólo se obtiene el dato sino que se tiene un conocimiento profundo del mismo. Si durante la misma no se indica claramente la necesidad de resolverlo usando programación, el alumno será libre de usar cualquier herramienta, siempre y cuando aporte las evidencias oportunas.

Ejercicios:

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?

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/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4US
   t3aB/i50nvvJbBiG+Ie1ZhpR84oI=
   ```

   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).

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.

   La etiqueta es cifrado-sim-chacha20-256

   ¿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.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSB
sb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ.gfhw0
dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE
```

¿Qué algoritmo de firma hemos realizado?
¿Cuál es el body del jwt?

Un hacker está enviando a nuestro sistema el siguiente jwt:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBk
ZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9
.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODlHRI
```

¿Qué está intentando realizar?
¿Qué ocurre si intentamos validarlo con pyjwt?

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

```
import hashlib

texto = "En KeepCoding aprendemos cómo protegernos con criptografía"
hash_calculado = hashlib.sha3_256(texto.encode()).hexdigest()

print("Hash calculado:", hash_calculado)

# Comparar con el hash proporcionado
hash_proporcionado = "bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe"

if hash_calculado == hash_proporcionado:
    print("El hash calculado coincide con el hash proporcionado.")
else:
    print("El hash calculado NO coincide con el hash proporcionado.")
```

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

```
4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f
6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833
```

¿Qué hash hemos realizado?

SHA-512 produce un hash de 512 bits, lo que equivale a 128 caracteres hexadecimales.

```
import hashlib

# Hash proporcionado
hash_provided =
"4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c
488823b8d858283f1d05877120e8c5351c833"

# Calcular el hash SHA-512 del texto vacío (solo como ejemplo)
hash_calculated = hashlib.sha512(b"").hexdigest()

print("Hash calculado (SHA-512):", hash_calculated)

# Comparar con el hash proporcionado
if hash_calculated == hash_provided:
   print("El hash calculado coincide con el hash proporcionado (SHA-512).")
else:
   print("El hash calculado NO coincide con el hash proporcionado.")
```

Genera ahora un SHA3 ~~Keccak~~ de 256 bits con el siguiente texto: “En KeepCoding aprendemos cómo protegernos con criptografía.”

```
import hashlib

texto = "En KeepCoding aprendemos cómo protegernos con criptografía"
hash_sha3_256 = hashlib.sha3_256(texto.encode()).hexdigest()

print("Hash SHA3-256:", hash_sha3_256)
```

¿Qué propiedad destacarías del hash, atendiendo a los resultados anteriores?

Resistencia a colisiones: Es extremadamente improbable que dos textos diferentes produzcan el mismo hash SHA3-256. La función hash SHA3-256 está diseñada para minimizar la posibilidad de colisiones, lo que significa que es muy difícil encontrar dos textos distintos que generen el mismo hash.

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.

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 uso de SHA-1 para almacenar contraseñas es considerado inseguro por varias razones principales:

- Seguridad Debilitada: SHA-1 es vulnerable a colisiones, lo que significa que dos contraseñas diferentes podrían tener el mismo hash, lo que facilita ataques de fuerza bruta y otros métodos de ataque.

- Velocidad de Cómputo: Debido a las mejoras en la capacidad de cálculo y técnicas de ataque, los hashes SHA-1 se pueden romper más fácilmente en comparación con métodos más seguros.

- Por lo tanto, proponer SHA-1 para almacenar contraseñas es una mala opción debido a su vulnerabilidad conocida.

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?

- La decisión de optar por SHA-256 representa un paso significativo hacia una mayor seguridad, superando las vulnerabilidades conocidas de SHA-1.

- Salting: Antes de aplicar el hash, cada contraseña debe ser acompañada por un valor único y aleatorio conocido como "salt". Este simple gesto asegura que incluso contraseñas idénticas generen hashes distintos, lo cual es esencial para proteger contra ataques de tabla arcoíris y fortalecer nuestra resistencia ante intentos de fuerza bruta.

- Iteraciones (Key stretching): Es recomendable aplicar el hash repetidamente utilizando técnicas como PBKDF2, bcrypt o Argon2. Este enfoque amoroso incrementa significativamente el tiempo y los recursos necesarios para generar el hash, dificultando considerablemente cualquier intento malintencionado, incluso si alguien accede a nuestra base de datos de hashes.

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

- Uso de Funciones de Hash Específicas para Contraseñas: Optar por funciones especialmente diseñadas para almacenar contraseñas, como bcrypt, scrypt o Argon2, añade una capa adicional de seguridad. Estas herramientas están meticulosamente diseñadas para proteger nuestros datos sensibles y gestionar la autenticación de manera segura.

8. Tenemos la siguiente API REST, muy simple.

Request:

Post /movimientos

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

Petición de ejemplo que se desea enviar:

{"idUsuario":1,"usuario":"José Manuel Barrio Barrio","tarjeta":4231212345676891}

Response:

Campo	Tipo	Requiere Confidencialidad	Observaciones
idUsuario	Number	N	Identificador
movTarjeta	Array	S	Formato del ejemplo

Saldo	Number	S	Tendra formato 12300 para indicar 123.00
Moneda	String	N	EUR, DOLLAR


```
{
        "idUsuario": 1,
        "movTarjeta": [{
                "id": 1,
                "comercio": "Comercio Juan",
                "importe": 5000
        }, {
                "id": 2,
                "comercio": "Rest Paquito",
                "importe": 6000
        }],
```

```
    "Moneda": "EUR",
    "Saldo": 23400

}
```

Como se puede ver en el API, tenemos ciertos parámetros que deben mantenerse confidenciales. Así mismo, nos gustaría que nadie nos modificase el mensaje sin que nos enterásemos. Se requiere una redefinición de dicha API para garantizar la integridad y la confidencialidad de los mensajes. Se debe asumir que el sistema end to end no usa TLS entre todos los puntos.

¿Qué algoritmos usarías?

· Confidencialidad de los Datos:
· Implementar cifrado de extremo a extremo utilizando algoritmos como AES.
Gestionar adecuadamente las claves criptográficas para asegurar la protección de los datos transmitidos.
· Integridad de los Mensajes:
· Utilizar firmas digitales (RSA, ECDSA) para verificar la autenticidad y la integridad de los mensajes.
Aplicar hashing de mensajes para detectar cambios no autorizados durante la transmisión.
· Autenticación y Autorización:
· Implementar un mecanismo robusto de autenticación, como tokens de acceso o certificados digitales, para validar la identidad de los clientes y servidores.
· Protección Adicional:
· Detectar y prevenir ataques de repetición que puedan comprometer la seguridad de la API.
Establecer un sistema de monitoreo continuo y auditorías periódicas para asegurar el cumplimiento de las políticas de seguridad.

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.

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.

Para abordar esta solicitud, vamos a seguir los siguientes pasos utilizando PGP:

Verificación de la Firma Digital de Pedro

1. Verificar la firma digital del mensaje de Pedro a RRHH:

- Tenemos los archivos:
  - `MensajeRespoDeRaulARRHH.txt.sig`: Archivo que contiene la firma digital del mensaje de Pedro.
  - `Pedro-priv.txt`: Clave privada de Pedro para desencriptar y firmar.
  - `RRHH-publ.txt`: Clave pública de RRHH para verificar la firma.

- Utilizaremos el comando `gpg` para verificar la firma digital:

bash

```
gpg --verify MensajeRespoDeRaulARRHH.txt.sig
```

Este comando intentará verificar la firma utilizando la clave pública de RRHH que está contenida en `RRHH-publ.txt`.

Firma Digital del Mensaje por RRHH

2. Firmar el mensaje de RRHH:

- El mensaje a firmar es: "viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario, saludos."

- Utilizaremos el archivo `RRHH-priv.txt`, que contiene la clave privada de RRHH, para firmar el mensaje.

- Comando para firmar el mensaje:
bash

```
gpg --local-user "RRHH" --output mensaje_firmado.asc --sign <<< "viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario, saludos."
```

Este comando generará un archivo `mensaje_firmado.asc` que contiene el mensaje firmado por RRHH.

Cifrado del Mensaje con Claves Públicas

3. Cifrar el mensaje con las claves públicas de RRHH y Pedro:
- El mensaje a cifrar es: "Estamos todos de acuerdo, el ascenso será el mes que viene, agosto si no hay sorpresas."

- Utilizaremos los archivos:
  - `RRHH-publ.txt`: Clave pública de RRHH para cifrar el mensaje.
  - `Pedro-publ.txt`: Clave pública de Pedro para cifrar el mensaje.

- Comandos para cifrar el mensaje con ambas claves públicas:

Bash

```
gpg --output mensaje_cifrado_RRHH.gpg --recipient "RRHH" --encrypt <<< "Estamos todos de acuerdo, el ascenso será el mes que viene, agosto si no hay sorpresas."

gpg --output mensaje_cifrado_Pedro.gpg --recipient "Pedro" --encrypt <<< "Estamos todos de acuerdo, el ascenso será el mes que viene, agosto si no hay sorpresas."
```

Estos comandos generarán dos archivos cifrados:
- `mensaje_cifrado_RRHH.gpg`: Mensaje cifrado con la clave pública de RRHH.
- `mensaje_cifrado_Pedro.gpg`: Mensaje cifrado con la clave pública de Pedro.

Explicacion final:

- Se ha verificado la firma digital del mensaje de Pedro a RRHH utilizando la clave pública de RRHH.
- Se ha firmado un mensaje en nombre de RRHH utilizando la clave privada correspondiente.
- Se ha cifrado un mensaje utilizando las claves públicas de RRHH y Pedro respectivamente.

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:

b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c96b4a8a20d5dbc639e9d
83a53681e6d96f76a0e4c279f0dffa76a329d04e3d3d4ad629793eb00cc76d10fc00475eb76bfbc127330388260
9957c4c0ae2c4f5ba670a4126f2f14a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8624071be7
8ccef573d896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f67d3f98b4cd907f27639f4b1df3c50e05d5bf
63768088226e2a9177485c54f72407fdf358fe64479677d8296ad38c6f77ea7cb74927651cf24b01dee27895d4f
05fb5c161957845cd1b5848ed64ed3b03722b21a526a6e447cb8ee

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

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

- Componente Aleatorio (Salt): Como se mencionó, RSA-OAEP utiliza un valor aleatorio diferente cada vez que cifras la misma clave. Este valor aleatorio, conocido como salt, se concatena con el mensaje antes de la operación de exponenciación modular. Por lo tanto, aunque la entrada (la clave simétrica) sea la misma, la presencia del componente aleatorio asegura que cada cifrado produzca un resultado diferente.

- Seguridad Criptográfica: La introducción de un componente aleatorio fortalece la seguridad del cifrado RSA-OAEP al prevenir ataques basados en la repetición de mensajes cifrados. Esto asegura que incluso si el mismo mensaje se cifra varias veces, los textos cifrados sean diferentes, lo que dificulta que un atacante pueda deducir información sobre la clave simétrica original.

- Llos textos cifrados son diferentes cada vez que cifras la misma clave simétrica utilizando RSA-OAEP debido al componente aleatorio (salt) que se incorpora durante el proceso de cifrado para mejorar la seguridad y la resistencia a ataques criptoanalíticos.

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?

- No parece ser un Nonce válido para AES-GCM. AES-GCM requiere que el Nonce tenga exactamente 12 bytes (96 bits) de longitud.

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.

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
import base64

# Datos proporcionados
key = b'E2CFF885901B3449E9C448BA5B948A8C'
nonce = b'9Yccn/f5nJJhAt2S'  # Este no es un nonce válido para AES-GCM

# Corregir el nonce para que tenga 12 bytes (96 bits)
# Generamos un nonce aleatorio de 12 bytes
nonce = get_random_bytes(12)

# Texto a cifrar
texto = "He descubierto el error y no volveré a hacerlo mal"

# Inicializar el cifrado AES-GCM
cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)

# Cifrar el texto
ciphertext, tag = cipher.encrypt_and_digest(texto.encode())

# Convertir el texto cifrado a hexadecimal y base64
ciphertext_hex = ciphertext.hex()
ciphertext_base64 = base64.b64encode(ciphertext).decode()

print("Texto Cifrado en Hexadecimal:", ciphertext_hex)
print("Texto Cifrado en Base64:", ciphertext_base64)
```

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:

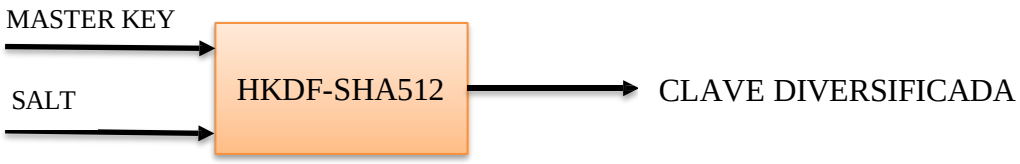
El equipo está preparado para seguir con el proceso, necesitaremos más recursos.

¿Cuál es el valor de la firma en hexadecimal?

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

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?

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?
¿Para qué algoritmo se ha definido la clave?

- La clave de transporte A1A10101010101010101010101010102 está definida específicamente para el algoritmo de desenvolvimiento (unwrap) del bloque TR31

¿Para qué modo de uso se ha generado?

- El modo de uso de la clave (A1A10101010101010101010101010102) se refiere a cómo se utiliza esta clave para desenvolver (unwrap) el bloque TR31 cifrado.

¿Es exportable?

- La exportabilidad de una clave generalmente depende de las políticas de seguridad y los estándares de cifrado aplicables en el contexto específico del sistema que la utiliza.

¿Para qué se puede usar la clave?

- La clave A1A10101010101010101010101010102 se utiliza específicamente para desenvolver (unwrap) el bloque TR31 que ha sido protegido con un algoritmo de cifrado.

¿Qué valor tiene la clave?

- La clave A1A10101010101010101010101010102 tiene un valor de 128 bits (16 bytes)