BOOTCAMP CIBERSEGURIDAD

PRÁCTICA CRIPTOGRAFÍA

Alejandro Delgado Martínez

53344428E

INDICE

EJERCICIOS:

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.

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?

Para este ejercicio abordaremos un problema de ciberseguridad relacionado con la gestión de forma segura de las claves. El objetivo es proteger esta clave para que ninguna persona tenga acceso directo a ella. Por ello, hemos desasociado la clave.

La disociación de la clave significa dividir la clave en dos partes:

- Clave fija, que se codifica en el software y que solo el desarrollador tiene acceso.
- La otra parte se almacena en un archivo de propiedades la cual rellenaba el Key Manager.

La clave final se ha generado en tiempo de ejecución a través de una operación XOR utilizando la clave del archivo de propiedades y la que está codificada en el software.

Se nos proporcionaron dos piezas de información: la **clave fija** codificada en el software (B1EF2ACFE2BAEEFF) y la **clave final** que se generaba en tiempo de ejecución (91BA13BA21AABB12).

Para realizar este ejercicio debemos determinar el valor que ha puesto el Key Manager en el archivo de propiedades para poder producir la clave final aportada.

XOR es una operación binaria que toma dos bits y devuelve 1 si los dos bits son diferentes; de lo contrario, devuelve 0.

Primero, convertimos las claves de hexadecimal a binario. Luego, realizamos la operación XOR en los valores binarios. Finalmente, convertimos el resultado de vuelta a hexadecimal para obtener la clave que se puso en el archivo de propiedades.

Para facilitar este proceso, desarrollamos un script en Python:

- El script toma dos números hexadecimales como entrada.
- Los convierte a enteros.
- Realiza la operación XOR.
- Convierte el resultado de vuelta a hexadecimal.

Además, aseguramos que el resultado tenga 16 bytes (32 caracteres hexadecimales) rellenando con ceros a la izquierda si es necesario.

Al realizar la operación XOR en el software, se añade una capa adicional de seguridad, ya que un atacante necesitaría tanto la clave del archivo de propiedades como el conocimiento de la operación XOR para obtener la clave final.

RESULTADO FINAL: **00000000000000020553975c31055ed**
En la carpeta adjunta, el archivo está denominado como: **ejercicio1a.py**

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?

En esta ocasión, se nos proporcionó la clave fija (B1EF2ACFE2BAEEFF) y la clave dinámica que se modifica en los archivos de propiedades (B98A15BA31AEBB3F). Nuestra tarea era determinar cuál sería la clave con la que se trabaja en memoria.

Utilizamos un script en Python para realizar la operación XOR entre la clave fija y la clave dinámica:

- El script convierte los números hexadecimales a enteros.
- Realiza la operación XOR.
- Convierte el resultado de vuelta a hexadecimal.

Además, como hicimos en el ejercicio anterior, aseguramos de que el resultado tenga 16 bytes (32 caracteres hexadecimales) rellenando con ceros a la izquierda si es necesario.

RESULTADO FINAL: **00000000000000008653f75d31455c0**

En la carpeta adjunta, el archivo está denominado como: **ejercicio1b.py**

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:

TQ9SOMKc6aFS9SIxhfK9wT18UXpPCd505Xf5J/5nLI7Of/o0QKIWXg3nu1RRz4QWElezdrLAD5LO4USt3aB/i50nvvJbBiG+le1ZhpR84oI=

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

En este ejercicio trabajaremos con el descifrado de datos. Se nos ha proporcionado un texto cifrado, una clave de cifrado, un vector de inicialización (IV) y, como se nos comenta en el ejercicio, se ha utilizado el algoritmo AES/CBC/PKCS7 para el cifrado.

Para comprender el algoritmo vamos a desgranarlo en diferentes partes:

- **AES:** algoritmo de cifrado simétrico que utiliza la misma clave para cifrar que para descifrar.
- **CBC:** modo de operación para los algoritmos de cifrado en bloque que proporciona confidencialidad.
- **PKCS7:** define el formato de los datos cifrados y especifica como añadir el padding a los datos antes del cifrado.

Para descifrar el texto cifrado utilizamos un script en Python que utiliza la biblioteca pycryptodome. Este script toma la clave de cifrado, el IV y el texto cifrado como entrada, crea un objeto AES cipher con la clave de cifrado y el IV, y luego descifra el texto cifrado y lo desempaqueta utilizando la función unpad.

Durante este proceso, nos encontramos con la necesidad de manejar los errores que pueden ocurrir durante el descifrado. Por ejemplo, si la clave de cifrado o el texto cifrado están mal formados, o si la clave de cifrado es incorrecta, el descifrado fallará. Para manejar estos errores, añadimos un bloque try/except al script para capturar y manejar estos errores.

Finalmente, obtuvimos estos resultados:
TEXTO DESCIFRADO:
Esto es un cifrado en bloque típico. Recuerda, vas por el buen camino. Ánimo.
Clave en el KeyStore:
A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72

En la carpeta adjunta, el archivo está denominado como: **ejercicio2.py**

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

Si decidimos cambiar el padding a x923 en el descifrado hay que recordar que el padding es utilizado en los algoritmos de cifrado de bloque para asegurar que los datos a cifrar se ajustan a un número específico de bloques. Si el padding que utilicemos cuando desciframos no coincide con el padding que se utilizó en el cifrado, el descifrado fallará porque los datos descifrados no serán válidos.

¿Cuánto padding se ha añadido en el cifrado?

La cantidad de padding depende del tamaño de los datos de entrada y del tamaño del bloque del algoritmo de cifrado. En el caso de AES, el tamaño del bloque es de 128 bits o 16 bytes. Si los datos de entrada no son un múltiplo de 16 bytes, se añadirá padding para hacer que los datos se ajusten a un múltiplo de 16 bytes. Como podemos ver en el texto cifrado, se ha añadido 1 "=", lo que se conoce como padding cuando utilizamos este tipo de cifrado. Por ello, podemos determinar que la cantidad de padding añadido es de 1.

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-chacha20-256". El nonce "9Yccn/f5nJJhAt2S". El algoritmo que se debe usar es un Chacha20.

En este ejercicio trabajaremos con el cifrado de los datos. Para ello, se nos ha proporcionado un texto en claro, una clave de cifrado y un nonce. Además, se nos pide que cifremos con el algoritmo Chacha20. ChaCha20 es un algoritmo de cifrado simétrico que utiliza la misma clave para el cifrado y el descifrado.

Para cifrar el texto en claro utilizaremos un script de Python que utiliza la biblioteca pycryptodome. Este script toma la clave de cifrado, el nonce y el texto en claro como entrada, crea un objeto ChaCha20 cipher con la clave de cifrado y el nonce, y luego cifra el texto en claro.

TEXTO CIFRADO:
aaxO58TFUlN6AKGbyvfwqu18nI92mVagm85vre9sNTXyIRyUZwZ89cSoQqs=

Clave en el KeyStore:
AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120

En la carpeta adjunta, el archivo está denominado como: **ejercicio3a.py**

¿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.

Podríamos mejorar este sistema para garantizar no sólo la confidencialidad de los datos, sino también su integridad. Para lograr esto, utilizaremos un algoritmo de cifrado autenticado como ChaCha20-Poly1305.

ChaCha20-Poly1305 combina el cifrado ChaCha20 con el código de autenticación de mensajes Poly1305 para proporcionar tanto confidencialidad (a través del cifrado) como integridad (a través de la autenticación). Esto significa que, además de cifrar los datos, ChaCha20-Poly1305 también genera un tag de autenticación que se puede utilizar para verificar la integridad de los datos.

Para demostrar cómo funciona ChaCha20-Poly1305, modificamos nuestro script de cifrado para utilizar ChaCha20-Poly1305 en lugar de ChaCha20. Este nuevo script realiza las mismas operaciones que el script original, pero también genera un tag de autenticación cuando cifra el texto en claro.

Finalmente, obtuvimos estos resultados:

TEXTO CIFRADO:
TslZIcqLdX4jNmBcfbq49NQLW00iDmaqI490DT5ZsM1w4yFyQpkcwUC7Hho=

TAG DE AUTENTICACIÓN:
cQzUcj2m1e838jvuZiheVw==

Clave en el KeyStore:
AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120

En la carpeta adjunta, el archivo está denominado como: **ejercicio3b.py**

EJERCICIO 4:

Tenemos el siguiente jwt, cuya clave es “Con KeepCoding aprendemos”.
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE
¿Qué algoritmo de firma hemos realizado? ¿Cuál es el body del jwt?

El JWT proporcionado está compuesto por tres partes, separadas por puntos: la cabecera, el cuerpo y la firma. Cada parte está codificada en Base64.

La cabecera del JWT especifica el algoritmo de firma que se ha utilizado. En este caso, la cabecera es eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9, que decodificado desde Base64 se traduce a: **{"typ":"JWT","alg":"HS256"}.** Determinamos que es la cabecera ya que, si observamos el JWT, está separado por un punto.

Esto indica que se ha utilizado el algoritmo de firma **HMAC con SHA-256 (HS256).**

El cuerpo del JWT contiene los reclamos o afirmaciones sobre una entidad y datos adicionales. En este caso, el cuerpo es:
eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ

Decodificado desde Base64 se traduce a **{"usuario":"Don Pepito de los palotes","rol":"isNormal","iat":1667933533}**.

Por lo tanto, el algoritmo de firma utilizado es HS256 y el cuerpo del JWT es **{"usuario":"Don Pepito de los palotes","rol":"isNormal","iat":1667933533}**.

En la carpeta adjunta, el archivo está denominado como: **ejercicio4a.py**

Un hacker está enviando a nuestro sistema el siguiente jwt:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODlHRI
¿Qué está intentando realizar? ¿Qué ocurre si intentamos validarlo con pyjwt?

El hacker está intentando enviar un JWT que tiene un reclamo de "rol" establecido en "isAdmin". Esto sugiere que el hacker está intentando obtener privilegios de administrador en el sistema.

Si intentas validar este JWT con pyjwt utilizando la clave secreta correcta, la validación fallará si la firma del JWT no coincide con la firma que se calcularía con la clave secreta correcta. Esto se debe a que la firma del JWT es un resumen de la cabecera y el cuerpo del JWT, calculado utilizando la clave secreta y el algoritmo de firma especificado en la cabecera del JWT.

Si la firma del JWT no coincide con la firma calculada, esto indica que el JWT ha sido manipulado después de ser firmado, lo que significa que no se puede confiar en su contenido.

En la carpeta adjunta, el archivo está denominado como: **ejercicio4b.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?

El hash proporcionado tiene una longitud de 64 caracteres hexadecimales. Dado que cada carácter hexadecimal representa 4 bits, el hash tiene un total de 64 * 4 = 256 bits. Por lo tanto, el hash se ha generado utilizando el algoritmo SHA3-256. El número después de SHA3 indica el tamaño del hash en bits. Por lo tanto, SHA3-256 genera un hash de 256 bits.

Y si hacemos un SHA2, y obtenemos el siguiente resultado:
4cec5a9f85dcc5c4c6ccb603d124cf1cdc6dfe836459551a1044f4f2908aa5d63739506f6468833d77c07cfd69c488823b8d858283f1d05877120e8c5351c833
¿Qué hash hemos realizado?

El hash proporcionado tiene una longitud de 128 caracteres hexadecimales. Dado que cada carácter hexadecimal representa 4 bits, el hash tiene un total de 128 * 4 = 512 bits. Por lo tanto, el hash se ha generado utilizando el algoritmo SHA-512.

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

EL HASH SHA3-256 GENERADO ES:
302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf

En cuanto a la propiedad qué destacaría del hash, es la propiedad de resistencia a colisiones. Esto significa que es extremadamente difícil encontrar dos entradas diferentes que produzcan el mismo hash. Esta propiedad es fundamental para la seguridad de las funciones hash y es lo que las hace útiles para verificar la integridad de los datos.

En la carpeta adjunta, el archivo está denominado como: **ejercicio5.py**

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.

Para calcular el HMAC-256 hemos utilizado un script de Python que realiza la generación y validación de un HMAC utilizando el algoritmo SHA256. El código define dos funciones: getHMAC y validateHMAC.

La **función getHMAC** toma como argumentos una clave en bytes y los datos en bytes. Utiliza la biblioteca Crypto.Hash para crear un nuevo objeto HMAC con la clave y los datos proporcionados, utilizando SHA256 como el algoritmo de hash. Luego, devuelve el HMAC en formato hexadecimal.

La **función validateHMAC** toma como argumentos una clave en bytes, los datos en bytes y un HMAC. Al igual que getHMAC, crea un nuevo objeto HMAC con la clave y los datos proporcionados. Luego, intenta verificar el HMAC proporcionado con el método hexverify. Si la verificación es exitosa, la función devuelve "OK". Si la verificación falla (es decir, si el HMAC proporcionado no coincide con el HMAC calculado), la función captura la excepción ValueError y devuelve "KO".

Finalmente, el código principal del script convierte una clave hexadecimal en bytes, codifica una cadena de texto en bytes, genera un HMAC de los datos con la clave utilizando getHMAC, e imprime el HMAC. Luego, valida el HMAC utilizando validateHMAC e imprime el resultado de la validación.

EL HMAC-256 CALCULADO ES:
857d5ab916789620f35bcfe6a1a5f4ce98200180cc8549e6ec83f408e8ca0550
CLAVE EN LA KEYSTORE:
A212A51C997E14B4DF08D55967641B0677CA31E049E672A4B06861AA4D5826EB

En la carpeta adjunta, el archivo está denominado como: **ejercicio6.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?

SHA-1 fue utilizado ampliamente en el pasado pero ya no es considerado seguro contra los atacantes. En 2005, los investigadores de seguridad descubrieron debilidades teóricas en SHA-1, y en 2017, un equipo de Google y la Universidad de Tecnología de Eindhoven produjo la primera colisión práctica.

Una colisión ocurre cuando dos entradas diferentes producen el mismo hash. Esto es un problema porque las funciones hash se utilizan para verificar la integridad de los datos. Si un atacante puede producir una colisión, puede engañar a un sistema para que acepte datos maliciosos en lugar de los datos legítimos.

Además, SHA-1 produce un hash de 160 bits, que es más corto que los hashes producidos por algoritmos más modernos como SHA-256 y SHA-3. Esto significa que es más fácil para un atacante adivinar un hash SHA-1 mediante la fuerza bruta.

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?

El uso de SHA-256 para almacenar contraseñas se considera más seguro que el SHA-1, pero aun así tiene algunas debilidades que debemos considerar. La más importante es que el SHA-256 es vulnerable a los ataques de fuerza bruta y de diccionario, de forma significativa si estas contraseñas no son fuertes.

La forma común de fortalecer el almacenamiento de contraseñas es utilizar el “Salting”. Un “Salt” es un valor aleatorio que es generado para cada usuario y se añade a la contraseña antes de calcular el hash. Este salt se almacena junto al hash de la contraseña y se utilizará para calcular el hash cada vez que el usuario se identifique.

Este salt tiene varias ventajas:

- Disminuye la fuerza de los ataques de fuerza bruta y de diccionario, ya que el atacante debería calcular un hash separado para cada posible hash.
- Asegura que, si dos usuarios tienen la misma contraseña, el hash sería diferente, ya que el salt es aleatorio y, por lo tanto, diferente.

Otra técnica que podemos utilizar para fortalecer el almacenamiento de contraseñas es el "Pepper". La idea detrás del uso de un pepper es añadir una capa adicional de seguridad a los hashes de contraseñas. Incluso si un atacante logra obtener la base de datos y descifrar las contraseñas (por ejemplo, mediante un ataque de fuerza bruta o un ataque de diccionario), no podrá obtener las contraseñas originales sin el pepper.

El pepper se utiliza en combinación con el salt y el hash de la contraseña. Primero, se añade el salt a la contraseña y se calcula el hash de la combinación. Luego, se añade el pepper a este hash y se calcula el hash de la combinación resultante. Este último hash es el que se almacena en la base de datos.

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

Una opción sería utilizar un algoritmo de hash diseñado específicamente para el almacenamiento de contraseñas, como bcrypt, scrypt o Argon2. Estos algoritmos están diseñados para ser resistentes a una variedad de ataques, incluyendo los ataques de fuerza bruta y de diccionario.

Además, incorporan automáticamente técnicas como el salting, lo que hace que sean más seguros y más fáciles de usar correctamente que SHA-256.

Argon2 es el más reciente de los tres y es resistente a una amplia gama de ataques y puede ser configurado para utilizar una cantidad arbitraria de memoria y CPU, lo que lo hace muy flexible.

EJERCICIO 8:

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

Para garantizar la integridad y la confidencialidad de los mensajes en una API REST sin depender de TLS, podríamos considerar el uso de los siguientes algoritmos y técnicas:

- **Cifrado simétrico para la confidencialidad:** Podríamos usar un algoritmo de cifrado simétrico como AES (Advanced Encryption Standard) para cifrar los datos sensibles en el mensaje. AES es un estándar de cifrado ampliamente utilizado y seguro que proporciona un buen equilibrio entre seguridad y rendimiento. Concretamente, utilizaremos el AES-GCM, que es el que mayor protección puede ofrecernos antes vulnerabilidades.
- **HMAC para la integridad:** Podríamos usar HMAC (Hash-based Message Authentication Code) para garantizar la integridad de los mensajes. HMAC combina una función hash criptográfica (como SHA-256) con una clave secreta para generar un código de autenticación del mensaje. El receptor del mensaje puede calcular su propio HMAC con la misma clave y compararlo con el HMAC recibido para verificar que el mensaje no ha sido alterado.
- **Salting y Hashing para las contraseñas:** Para almacenar las contraseñas de los usuarios de forma segura, podríamos usar una técnica de salting y hashing. Esto implica añadir un valor aleatorio (el "salt") a la contraseña del usuario antes de calcular su hash. Esto hace que los ataques de fuerza bruta y de diccionario sean mucho más difíciles.
- **JWT para la autenticación:** Podríamos usar JWT (JSON Web Tokens) para la autenticación de los usuarios. Un JWT es un token que puede ser firmado y/o cifrado y que contiene una serie de reclamaciones sobre el usuario. El servidor puede verificar la firma del JWT para autenticar al usuario.

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.

El KCV se utiliza para verificar la integridad de una clave sin revelar la clave en sí. La clave AES que hemos usado es la siguiente:
A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72.

Para calcular el KCV, hemos utilizado dos métodos diferentes: SHA-256 y AES.

El primer método, SHA-256, es una función hash criptográfica que produce un hash de 256 bits. Para calcular el KCV usando SHA-256, hemos creado un nuevo objeto SHA-256 y lo hemos actualizado con la clave AES. Luego, hemos tomado los primeros 3 bytes del hash resultante. Este es el KCV de la clave utilizando SHA-256.

El segundo método, AES, es un algoritmo de cifrado simétrico. Para calcular el KCV usando AES, hemos creado un nuevo objeto AES en modo ECB (Electronic Codebook) con la clave AES. Luego, hemos cifrado un bloque de 16 bytes de ceros y hemos tomado los primeros 3 bytes del texto cifrado. Este es el KCV de la clave utilizando AES.

Hay que destacar que estos dos métodos producen KCVs diferentes, ya que utilizan algoritmos diferentes. Pero, aun así, ambos son válidos y podrían utilizarse para verificar la integridad de la clave AES.

RESULTADOS OBTENIDOS:

- KCV(SHA-256): **db7df2**
- KCV(AES): **5244db**

En la carpeta adjunta, el archivo está denominado como: **ejercicio9.py**

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.

Así mismo, se requiere firmar el siguiente mensaje con la clave correspondiente de las anteriores, simulando que eres personal de RRHH.

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

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

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

En este ejercicio, hemos trabajado con conceptos de criptografía y seguridad de la información para verificar y firmar mensajes, así como para cifrar datos. Utilizaremos GnuPG, una herramienta de cifrado de código abierto que implementa el estándar OpenPGP, para realizar estas tareas.

Primero, hemos importado las claves necesarias para verificar, firmar y cifrar los mensajes. Estas claves se han almacenado en archivos de texto, y las hemos importado utilizando el comando gpg --import. Este comando lee las claves de los archivos de texto y las añade al llavero de GnuPG, que es donde GnuPG guarda todas las claves que conoce.

```
gpg --import Pedro-publ.txt
gpg --import RRHH-priv.txt
```

Después, hemos verificado la firma de un mensaje enviado por Pedro, el responsable de Raúl. Pedro ha firmado su mensaje con su clave privada, y hemos verificado esta firma con la clave pública de Pedro. Para hacer esto, hemos utilizado el comando gpg --verify. Este comando comprueba si la firma del mensaje coincide con la clave pública, lo que nos permite confirmar que el mensaje realmente proviene de Pedro y que no ha sido modificado después de que Pedro lo firmara.

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

A continuación, hemos firmado un mensaje como si fuéramos personal de RRHH. Para hacer esto, hemos creado un archivo con el mensaje y luego hemos utilizado el comando gpg --sign para firmar este archivo con la clave privada de RRHH. Esta firma permite a cualquier persona que tenga la clave pública de RRHH verificar que el mensaje proviene de RRHH y que no ha sido modificado después de que lo firmáramos.

```
echo "Viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario. Saludos." > MensajeRRHH.txt
```

```
gpg --sign --local-user RRHH --output MensajeRRHH.txt.sig --armor MensajeRRHH.txt
```

Este comando creará un archivo firmado llamado MensajeRRHH.txt.sig.

Por último, hemos cifrado un mensaje para RRHH y Pedro. Para hacer esto, hemos creado un archivo con el mensaje y luego hemos utilizado el comando gpg --encrypt para cifrar este archivo con las claves públicas de RRHH y Pedro. Este cifrado asegura que sólo RRHH y Pedro, que tienen las claves privadas correspondientes, pueden descifrar y leer el mensaje.

```
echo "Estamos todos de acuerdo, el ascenso será el mes que viene, agosto, si no hay sorpresas." > MensajeFinal.txt
```

```
gpg --encrypt --recipient RRHH --recipient Pedro --output MensajeFinal.txt.gpg MensajeFinal.txt
```

Este comando creará un archivo cifrado llamado MensajeFinal.txt.gpg.

En la carpeta adjunta, podrás encontrar los archivos generados en la subcarpeta:
Ejercicio 10

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:
b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d04e3d3d4ad629793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae2c4f5ba670a4126f2f14a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8624071be78ccef573d896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f67d3f98b4cd907f27639f4b1df3c50e05d5bf63768088226e2a9177485c54f72407fdf358fe64479677d8296ad38c6f177ea7cb74927651cf24b01dee27895d4f05fb5c161957845cd1b5848ed64ed3b03722b21a526a6e447cb8ee

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

El RSA-OAEP es un esquema de cifrado de clave pública que utiliza el algoritmo RSA y un esquema de relleno para hacer que el cifrado sea más seguro. El RSA-OAEP es resistente a los ataques de texto cifrado elegido, lo que significa que incluso si un atacante puede hacer que se descifren ciertos textos cifrados, no puede usar la información que obtiene para descifrar otros textos cifrados.

Durante el ejercicio he considerado la idea de que fuera un texto legible (UTF8, por ejemplo) pero daba constantemente error en el descifrado, por lo que opté por descifrar directamente en bytes.

Clave recuperada:
b'\xe2\xcf\xf8\x85\x90\x1aTI\xe9\xc4H\xba[\x94\x8a\x8cN\xe3w\x15+?\x1a\xcf\xa0\x14\x8f\xb3\xa4&\xdbr'

En la carpeta adjunta, podrás encontrar la solución en el archivo: **ejercicio11a.py**

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

Una de las características del RSA-OAEP es que utiliza un valor aleatorio en su esquema de relleno. Esto significa que incluso si cifras el mismo mensaje dos veces con la misma clave, obtendrás dos textos cifrados diferentes. Esto se debe a que el valor aleatorio utilizado en el

relleno será diferente en cada cifrado. Gracias a esta aleatoriedad, el RSA-OAEP se considera seguro contra los ataques de texto cifrado elegido.

Por lo tanto, si has recuperado la clave y vuelves a cifrarla con el mismo algoritmo, obtendrás un texto cifrado diferente debido a este valor aleatorio en el esquema de relleno.

Nuevo cifrado:
4d95d2212c4dcefc9b3f154c4119be8458bb2106c34df9d81094a6ae846bda4b95e933d6b4ada
4e6119b1acebae7cc5fed30562c0014ec5b673f47cd70c63e0cc3ff1d8f921140da6c9ff140b414a5
2e65bde2971a324b6b99afe2c0e4e8344d138d2c9fa0271e5a3a1772f587b2184e0dee61b3ef3c5
07f1a3d93bb4c21990868ec03494852414cd295d73beb4e30f9bb452006a619042f61a999b9107
58d81c5ae6a2d59d96012bf4c98b3141c31e19805eec65e7cfc3dc06c36bfa60dce191423fadf2d2
eb01a853aa393616da358740f55ffac3d2945026ccf9e45ec4bf24f98db1c6150dada3479bf29f2
8b412e4214365829b2c5618a9bebff9ce2223a

En la carpeta adjunta, podrás encontrar la solución en el archivo: **ejercicio11b.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?

El problema aquí es que estás utilizando el mismo nonce (número que se usa una sola vez) para cada comunicación. En el modo de operación GCM (Galois/Counter Mode) de AES, el nonce debe ser único para cada mensaje cifrado con la misma clave.

Si se reutiliza el mismo nonce, un atacante podría potencialmente descifrar los mensajes o incluso falsificar mensajes. El nonce no necesita ser secreto y puede transmitirse en texto plano junto con el mensaje cifrado, pero debe cambiar para 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 preséntalo en hexadecimal y en base64.

Para este ejercicio utilizaremos el algoritmo de cifrado AES/GCM, muy utilizado ya que proporciona tanto confidencialidad como autenticación de los datos, lo que lo hace seguro y eficiente.

Primero, hemos tomado una clave y un nonce dados. La clave es un valor secreto que se utiliza para cifrar y descifrar los datos, mientras que el nonce es un valor que se debe cambiar con cada mensaje cifrado para garantizar la seguridad del algoritmo.

En este caso, la clave y el nonce nos fueron proporcionados en formato hexadecimal y base64 respectivamente, por lo que tuvimos que convertirlos a bytes antes de poder utilizarlos.

A continuación, hemos creado un objeto de cifrado AES/GCM utilizando la clave y el nonce. Este objeto de cifrado se utiliza para cifrar los datos que queremos proteger. En este caso, los datos son una cadena de texto que dice "He descubierto el error y no volveré a hacerlo mal".

Después de cifrar los datos, obtenemos el texto cifrado en formato de bytes. Sin embargo, este formato no es muy legible ni fácil de manejar, por lo que lo hemos convertido a dos formatos más manejables: hexadecimal y base64.

RESULTADOS OBTENIDOS:

- **Texto cifrado en Hexadecimal:**
  5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd9d84bba6eb8307095f5078fbfc16256d

- **Texto Cifrado en Base64:**
  Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq92dhLum64MHCV9QePv8FiVt

En la carpeta adjunta, el archivo está denominado como: **ejercicio12.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.pem 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?

En este último ejercicio, hemos trabajado con el algoritmo de firma PKCS#1 v1.5 y claves RSA para firmar un mensaje. El objetivo era calcular el valor de la firma en formato hexadecimal.

El primer paso fue importar las bibliotecas necesarias para el trabajo. Usamos Crypto.PublicKey.RSA para trabajar con las claves RSA, Crypto.Hash.SHA256 para crear un hash del mensaje, Crypto.Signature.pkcs1_15 para implementar el esquema de firma PKCS#1 v1.5 y binascii para convertir datos binarios en una representación hexadecimal.

Luego, definimos la ruta del archivo que contiene la clave privada RSA. Esta clave se importó utilizando el método importKey de la clase RSA. El mensaje que queríamos firmar se convirtió en bytes y se almacenó en la variable msg.

Después, creamos un objeto hash del mensaje utilizando SHA256. Este hash se utilizó para generar la firma digital. Para ello, creamos un objeto signer utilizando el esquema de firma PKCS#1 v1.5 y la clave privada RSA. Luego, firmamos el hash del mensaje con este objeto signer y almacenamos la firma en la variable signature.

Finalmente, imprimimos la firma en formato hexadecimal utilizando el método hex().

RESULTADOS OBTENIDOS:

- **FIRMA EN HEXADECIMAL:**

a4606c518e0e2b443255e3626f3f23b77b9d5e1e4d6b3dcf90f7e118d6063950a23885c6dece92
aa3d6eff2a72886b2552be969e11a4b7441bdeadc596c1b94e67a8f941ea998ef08b2cb3a925c
959bcaae2ca9e6e60f95b989c709b9a0b90a0c69d9eaccd863bc924e70450ebbbb87369d7
21a9ec798fe66308e045417d0a56b86d84b305c555a0e766190d1ad0934a1befbbe0318532775
69f8383846d971d0daf05d023545d274f1bdd4b00e8954ba39dacc4a0875208f36d3c9207af0
96ea0f0d3baa752b48545a5d79cce0c2ebb6ff601d92978a33c1a8a707c1ae1470a09663acb6b
9519391b61891bf5e06699aa0a0dbae21f0aaaa6f9b9d59f41928d

En la carpeta adjunta, el archivo está denominado como: **ejercicio13a.py**

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

En este ejercicio, hemos trabajado con la firma de mensajes utilizando la criptografía de curva elíptica, específicamente la curva ed25519. La criptografía de curva elíptica es una forma de criptografía de clave pública que utiliza las matemáticas de las curvas elípticas para proporcionar un alto nivel de seguridad con claves relativamente pequeñas.

Primero, hemos definido la ruta de los archivos que contienen las claves públicas y privadas. Estos archivos se encuentran en la carpeta 'Practica' y tienen las extensiones '.pem'. Hemos abierto estos archivos en modo de lectura binaria y hemos leído las claves.

A continuación, hemos creado una clave de firma a partir de la clave privada utilizando la biblioteca 'ed25519'. Esta clave de firma se utiliza para firmar un mensaje específico, que en este caso es 'El equipo está preparado para seguir con el proceso, necesitaremos más recursos.'.

Después de firmar el mensaje, hemos impreso la firma generada en formato hexadecimal. Esta firma es una cadena de 64 bytes que se puede utilizar para verificar la autenticidad del mensaje.

Finalmente, hemos verificado la firma utilizando la clave pública. Si la firma es válida, el programa imprime 'La firma es válida'. Si la firma no es válida, el programa imprime 'Firma inválida!'.

RESULTADOS OBTENIDOS:

- **FIRMA EN CURVA ELÍPTICA ed25519:**

b'bf32592dc235a26e31e231063a1984bb75ffd9dc5550cf30105911ca4560dab52abb40e4f7e2d3af828abac1467d95d668a80395e0a71c51798bd54469b7360d'

En la carpeta adjunta, el archivo está denominado como: **ejercicio13b.py**

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?

En este último ejercicio, hemos trabajado con la generación de una clave derivada a partir de una clave maestra utilizando la función HKDF (HMAC-based Extract-and-Expand Key Derivation Function). Esta función es una forma común de generar claves seguras a partir de una clave maestra, y es especialmente útil cuando se necesita generar múltiples claves a partir de una única clave maestra.

El código en Python que hemos utilizado hace uso de la biblioteca Crypto, que proporciona una serie de funciones criptográficas para Python. En particular, hemos utilizado la función HKDF de esta biblioteca para generar nuestra clave derivada.

La función HKDF toma varios argumentos: la clave maestra, la longitud de la clave derivada que se desea generar, un salt y el algoritmo de hash que se utilizará. En nuestro caso, hemos utilizado SHA-512 como nuestro algoritmo de hash.

La clave maestra y el salt se proporcionan en formato hexadecimal, y se convierten a bytes antes de ser utilizados en la función HKDF. Esto se hace utilizando la función bytes.fromhex de Python, que convierte una cadena de caracteres hexadecimales en una secuencia de bytes.

Una vez que tenemos nuestra clave maestra y nuestro salt en formato de bytes, podemos pasarlos a la función HKDF junto con la longitud deseada de nuestra clave derivada y nuestro algoritmo de hash. La función HKDF entonces genera nuestra clave derivada, que se imprime en formato hexadecimal utilizando el método hex de Python.

RESULTADOS OBTENIDOS:

- **CLAVE MAESTRA:**
  A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72

- **CLAVE DERIVADA:**
  e716754c67614c53bd9bab176022c952a08e56f07744d6c9edb8c934f52e448a

En la carpeta adjunta, el archivo está denominado como: **ejercicio14.py**

EJERCICIO 15:

Nos envían un bloque TR31:
D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37018E111B

Donde la clave de transporte para desenvolver (unwrap) el bloque es:
A1A1010101010101010101010101010102

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

El algoritmo que se utilizó para proteger el bloque de clave es el indicado por el valor 'A'.
'A' corresponde a un algoritmo de cifrado Triple-DES (TDEA) de doble longitud.

¿Para qué algoritmo se ha definido la clave?

La clave se ha definido para el algoritmo indicado por el valor 'A', que corresponde a Triple-DES (TDEA) de doble longitud.

¿Para qué modo de uso se ha generado?

El modo de uso para el que se ha generado la clave es el indicado por el valor 'B'.
'B' corresponde a un modo de uso de cifrado de clave de datos.

¿Es exportable?

La exportabilidad de la clave se indica con el valor 'S'.
'S' significa que la clave es exportable bajo ciertas condiciones de seguridad.

¿Para qué se puede usar la clave?

El uso de la clave se indica con el valor 'D0'.
'D0' corresponde a una clave que se puede utilizar para cualquier propósito.

¿Qué valor tiene la clave?

El valor de la clave es 'c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1'.

RESULTADOS OBTENIDOS:

- c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1
- Key Version ID: D
- Algoritmo: A
- Modo de uso: B
- Uso de la clave: D0
- Exportabilidad: S

En la carpeta adjunta, el archivo está denominado como: **ejercicio15.py**

CONCLUSIONES:

En los ejercicios que hemos estado resolviendo, hemos explorado una variedad de temas relacionados con la criptografía y la seguridad de la información. Hemos trabajado con algoritmos de cifrado simétrico y asimétrico, firmas digitales, funciones de derivación de claves, y hemos aprendido sobre la importancia de la gestión segura de las claves.

En particular, hemos trabajado con el algoritmo de cifrado AES en modo GCM, que es un algoritmo de cifrado simétrico muy utilizado en la actualidad debido a su eficiencia y seguridad. Hemos aprendido sobre la importancia de gestionar correctamente los nonces en este algoritmo para evitar la reutilización de nonces, lo cual puede llevar a vulnerabilidades.

También hemos trabajado con firmas digitales utilizando RSA y el esquema de firma PKCS#1 v1.5. Las firmas digitales son una herramienta esencial para garantizar la autenticidad e integridad de los mensajes en las comunicaciones digitales.

Además, hemos explorado la función de derivación de claves HKDF y su uso para generar claves seguras a partir de una clave maestra. Hemos aprendido sobre la importancia de utilizar un salt en la derivación de claves para garantizar la unicidad de las claves derivadas.

Por último, hemos trabajado con el formato de bloque de clave TR31, que es un estándar para el intercambio seguro de claves en la industria de los pagos. Hemos aprendido sobre la estructura de estos bloques de clave y cómo se pueden proteger con una clave de transporte.

Finalmente, decir que estos ejercicios nos han proporcionado una visión práctica de varios aspectos de la criptografía y la seguridad de la información, nos ha ayudado a visualizar la importancia de la gestión segura de las claves y la correcta implementación de los algoritmos de cifrado y firma.

Considero que estos ejercicios nos han aportado conocimientos esenciales para que podamos trabajar en un futuro en el campo de la ciberseguridad, considerando la criptografía como una herramienta fundamental para que podamos garantizar la confidencialidad, la integridad y la autenticidad de la información en la comunicación digital.