Seguridad para Backend: Guía teórica simple de cifrado, hashing y autenticación

Por *github.com/NehuenLian*.

Esta guía está enfocada a programadores backend que recién comienzan en el área o que si bien ya comenzaron, pasaron por alto algunos conceptos de seguridad.
La idea es que sea un punto intermedio entre lo técnico y lo sencillo, es decir, que no sea un documento extremadamente técnico pero que tampoco se convierta en información poco útil debido a su simpleza.

Esta guía da por hecho que el lector conoce los siguientes temas:

- Protocolo HTTP.
- Arquitectura Cliente-Servidor.
- Qué es una API.
  Nociones de arquitectura de sistemas/software (monolitos/microservicios).

Temas a cubrir

-Diferencia entre cifrado y hashing.
-Cifrado simétrico vs asimétrico.

-Distintas capas de seguridad

- Cifrado TLS.
- Creación de Tokens, firmas digitales y validación de las mismas.
- Hashing de contraseñas.
- Cifrado en persistencia.

-Flujo al crear una cuenta en una página web o servicio.
-Flujo al iniciar sesión en un sitio.

Cifrado y hashing: diferencias entre cada uno

Tanto el cifrado como el hashing tienen el fin de proteger datos, pero con diferente enfoque según la naturaleza del dato a proteger y el contexto:

Cifrado

La característica que más diferencia al cifrado del hashing es que es reversible (puede recuperarse el dato una vez ya no se necesite ocultarlo o se lo quiera leer) esto se hace descifrándolo.

Consiste en transformar datos que estén en texto plano en un formato codificado, esto se hace usando una clave.

Para obtener el dato original, se necesita descifrarlo y para descifrarlo se necesita también una clave definida previamente por el desarrollador.

Se utiliza cuando la información necesita ser leída o procesada nuevamente más adelante, como mensajes, números de tarjeta, datos clínicos, etc.

Ejemplo:

Supongamos que usamos la clave *"clave1"* para cifrar un texto.
Si queremos cifrar el mensaje *"Hola mundo"*, aplicamos el algoritmo con esa clave y obtenemos un texto cifrado, por ejemplo:
"3f1e9a7bc2a3e8c5f67a69e70f3c4c2a"

Para leer el mensaje original, aplicamos el algoritmo inverso (descifrado) usando la misma clave. Así recuperamos*:*
"Hola mundo"

Ahora el dato puede leerse normalmente.

Hashing

El hashing es un método unidireccional de cubrir el contenido real de un dato para protegerlo. Lo que significa que, a diferencia del cifrado, no es reversible: No se puede recuperar el dato original de ninguna forma una vez fue hasheado, no hay clave para volver a hacerlo, es un “viaje de ida”.

Transforma el dato en una cadena de longitud fija, se compone de caracteres sin significado alguno pero determinados por el contenido original.

El hashing es utilizado en casos donde justamente no se necesite recuperar el dato a proteger, como contraseñas, o para verificar la integridad o la legitimidad de un dato.

Ejemplo:

Si aplicamos un algoritmo de hashing sobre el texto *"Hola mundo"*, obtenemos un valor como:

"d9e3f04a363698763cfc3c3cc50a757e7ce2e6a95b2a31eb3e4f639a2e2e3f28"

Este valor es único para ese texto, pero no es posible volver al texto original desde él.
El hashing es un proceso de “viaje de ida”: una vez aplicado, no puede revertirse.

Cifrado simétrico vs asimétrico

Existen varios tipos de cifrado, pero el simétrico y el asimétrico son los más utilizados en el contexto de la web. Recordando que en el cifrado se utiliza una clave para cifrar y descifrar, tenemos:

Cifrado simétrico

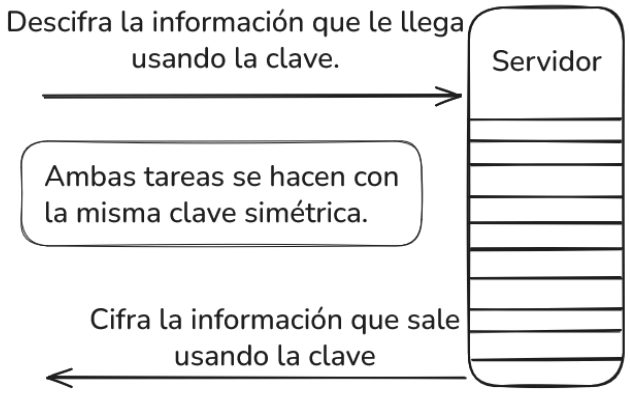
El cifrado simétrico consiste en utilizar una única clave para cifrar y descifrar la información.

Todos los mensajes cifrados con esta clave necesitan la misma clave para descifrar.

En arquitecturas como microservicios se llama “Clave simétrica compartida”, ya que todos los microservicios van a necesitar la clave para comunicarse, cifrar y descifrar mensajes. En cambio, en arquitecturas monolíticas o con un solo servidor, el servidor se encarga de cifrar y descifrar sin necesidad de compartir la clave, porque no hay otros actores que la necesiten.

Es común que se confunda este concepto, ya que el cifrado simétrico suele presentarse directamente en ejemplos con microservicios.

<u>Ejemplo de flujo en una arquitectura monolítica que utiliza cifrado simétrico:</u>



Cifrado asimétrico

El cifrado asimétrico involucra dos claves en lugar de una:
una clave pública y una clave privada.

¿Cómo se generan estas claves?

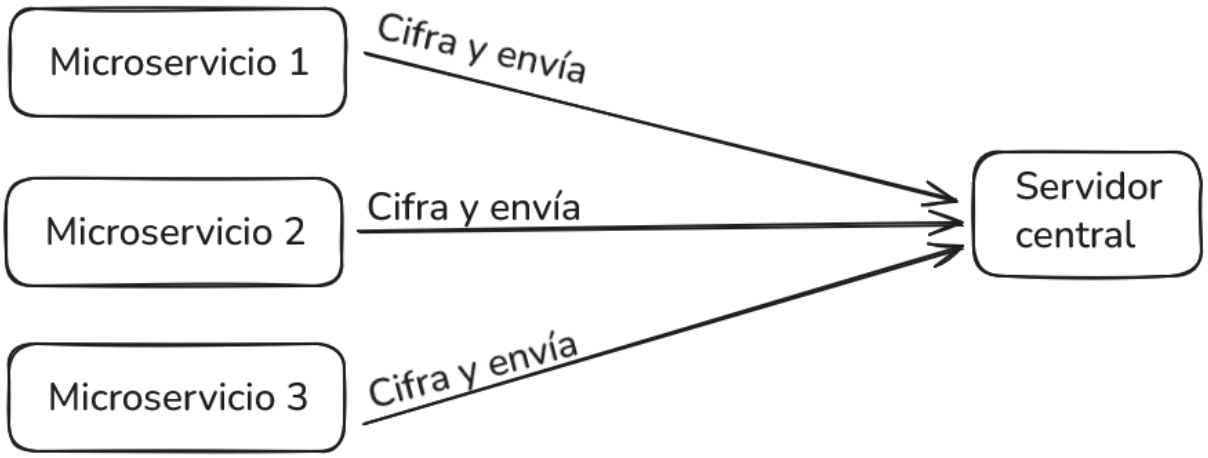
Se generan a partir de ciertos parámetros matemáticos, dependiendo del algoritmo de cifrado elegido.

Ambas claves se generan en conjunto una vez aplicado el algoritmo. Están vinculadas matemáticamente y siguen esta dinámica:

- La clave privada se mantiene en secreto.
- La clave pública puede compartirse con cualquier nodo/microservicio.

La clave del cifrado asimétrico es que con la clave pública solo se puede cifrar información, pero no tiene la capacidad de descifrar. Quien descifra la información solo es quien tiene la clave privada.

Ejemplo de cifrado asimétrico en una arquitectura de microservicios:



Como se ve en la imágen, los microservicios son quienes tienen la clave pública, por lo tanto solo pueden cifrar información, y el servidor central es quien la descifra, ya que es quien tiene la clave privada.

Capas de seguridad

A continuación, voy a explicar las distintas capas de seguridad que hay desde abajo hacia arriba, para mostrar cómo se va construyendo la seguridad en distintos niveles.

1. Cifrado TLS:

TLS es un protocolo criptográfico que funciona debajo de la capa de aplicación (ver modelo OSI). Su función es cifrar las comunicaciones entre el cliente y el servidor, convirtiendo las peticiones HTTP en HTTPS para que el contenido no pueda ser leído ni alterado fácilmente.

TLS también funciona con claves asimétricas: una pública que se envía al cliente, y una privada que se mantiene en el servidor. La clave pública se encarga de cifrar los paquetes que van a ser enviados al servidor, y el servidor utiliza su clave privada para descifrar los mensajes que le llegan.

TLS se establece al validar una conexión segura entre el cliente y el servidor, este proceso se denomina Handshake, que no se va a profundizar demasiado en esta guía.

Dentro del handshake, se encuentran varios pasos al establecerse el protocolo TLS, estos pasos involucran un intercambio de claves bastante engorroso, así que voy a intentar de explicarlo a continuación de la manera más amigable posible.

Una vez configurado qué cifrado se va a utilizar para los paquetes (esto lo hace el servidor automáticamente al negociar los parámetros del Handshake con el cliente).

Sabiendo que el servidor posee las dos claves (pública y privada):

1. El cliente inicia la conexión al servidor (por ejemplo, ingresando a un sitio con *https://*).
2. El servidor envía su clave pública al cliente
3. El cliente genera una nueva clave, esta vez es una clave simétrica AES temporal.
4. Luego, el cliente cifra esa clave simétrica con la clave pública del servidor y se la envía
5. El servidor usa su clave privada para descifrar esa clave que le envió el cliente.

Luego de este proceso, la conexión ya se encuentra cifrada para poder enviar peticiones y recibir información de forma segura.
TLS puede parecer complejo por dentro, pero la realidad es que esto ocurre automáticamente, el desarrollador casi nunca tiene que preocuparse, se maneja sin intervención manual.

<u>Tips relevantes:</u>

- Si el descifrado falla significa que la clave no viene de un cliente legítimo.
- Si el cifrado es correcto se acepta y a partir de ahora todos los paquetes viajarán cifrados.
- Todo este proceso ocurre dentro del handshake.
- La seguridad viene del hecho de que sólo el servidor tiene la clave privada para poder descifrar.

2. Certificado X.509

Un certificado X.509 es una credencial que se utiliza para identificar a una entidad (como un servidor). Además de asegurar que el servidor es legítimo también sirve para cifrar la comunicación y firmar datos para garantizar su autenticidad.

El certificado es emitido y firmado por una entidad llamada Autoridad Certificadora (CA), y hay varias alrededor del mundo.

- Físicamente, una CA es también un servidor o conjunto de servidores, que operan bajo políticas de seguridad muy estrictas para emitir certificados.

Cuando un cliente (como un navegador) quiere conectarse a un servidor, durante el proceso de Handshake, el servidor envía su certificado al cliente y este lo verifica. Pero cómo hace para verificar el certificado? Bueno, la clave de esto es que todos los navegadores tienen almacenado una lista de CAs donde busca y averigua la legitimidad del certificado que recibió.

Cómo funciona el proceso de verificación dentro del proceso de Handshake (simplificado):

1. El cliente inicia con un ClientHello
2. El servidor responde con un ServerHello y envía su certificado
3. El cliente verifica que ese certificado sea válido, consultando a las CAs que tiene almacenadas.
4. Si todo está bien, continúa el proceso del Handshake para establecer la sesión y cifrarla.

Cómo funciona el certificado X.509 a nivel criptográfico

Es un mecanismo de cifrado asimétrico, lo cual como ya sabemos, funciona con una clave pública y una clave privada.
La clave pública está alojada en el certificado X.509, y la clave privada está en el servidor, este es el par asimétrico.

La CA que emite el certificado tiene su propia clave privada, y la usa para firmarlo. Esta firma es el resultado de aplicar un hash a los datos del certificado + la clave privada de la CA.

El cliente calcula el hash del contenido del certificado recibido y usa la clave pública de la CA para verificar la firma digital que lo acompaña. Si la verificación es exitosa, significa que el certificado es válido y fue firmado por una CA confiable. Si el resultado coincide, es porque el certificado es válido y el servidor es confiable.

Tip:

- A veces el certificado del servidor no está firmado directamente por una CA raíz, sino por una CA intermedia. En ese caso, el servidor también envía la cadena de certificados intermedios para que el cliente pueda construir la “cadena de confianza” hasta una CA raíz conocida.

3. Tokens y firmas digitales

- ¿Qué es un token?

Un token es un objeto de datos que contiene información y actúa como identificador.
Son emitidos por el servidor, y se incluyen dentro de las solicitudes (por ejemplo, en el header) para que el servidor pueda verificar si ese token fue emitido por él mismo y, por ende, si la solicitud es legítima.
Uno de los tokens más comunes es el JWT (JSON Web Token).

Un JWT tiene tres partes, codificadas en formato Base64:

1. Header
2. Payload
3. Signature

- Lo importante: ¿Cómo se firma un JWT?

Cuando un servidor emite un JWT, el flujo es el siguiente:

1. El servidor recibe los datos

2. Crea el token. El token emitido va a estar vinculado a ese usuario y va a usarse para identificarlo.
3. El servidor firma el token.
4. El servidor le envía ese token al cliente.
5. El cliente almacena ese token (por ejemplo, en localStorage) y lo incluye dentro de las siguientes solicitudes que envíe.
6. Cada vez que el servidor recibe una solicitud con el token, verifica la firma y la validez.

- ¿Y, cómo se firma un token?

Para firmar tokens se utilizan algoritmos de firma, los más comunes son HMAC (simétrico) y RSA (asimétrico), ambos dependiendo del caso.

4. HMAC:

HMAC es un algoritmo de firma, de las siglas Hash-based message authentication, que utiliza una clave secreta o compartida (dependiendo de si la arquitectura en cuestión es monolítica o de microservicios).

Consiste en “extraer” las partes del token y realizar una serie de pasos y cálculos para agregar una firma a ese token.
Una vez el servidor tiene el token los pasos son los siguientes:

Se toma el header y el payload, los concatena para obtener un string resultante.
Luego, a ese string resultante se le agrega la clave secreta.
Una vez obtenido el resultado del string + la clave secreta, se le aplica una función hash, ese hash es ahora la firma del token.
Luego este hash se agrega al token como valor del atributo “signature”.

Una vez el token está firmado, el servidor va a poder verificar si ese token es legítimo cuando le llegue.

- ¿Y cómo hace para verificar?

Cuando el servidor recibe un JWT firmado previamente, antes de aprobar la solicitud tiene que verificarlo.

El servidor vuelve a recalcular la firma de ese token haciendo el mismo procedimiento que realizó antes para firmarlo por primera vez.
Si la firma resultante es idéntica a la firma del JWT, es porque ese token fue firmado por ese servidor, lo que significa que es legítimo.

Aclaración sobre la “clave compartida”:

En HMAC se le llama “clave compartida” pero esto no siempre es así. Lo que sucede es que, en una arquitectura de microservicios todos los microservicios tienen que compartir esa clave. Pero en monolitos la clave no es compartida por nadie, sólo la tiene el único servidor disponible, el cual sólo la utiliza para validar los tokens que recibe desde fuera.

En microservicios, todos los servicios que validan JWTs deben compartir la misma clave secreta, ya que todos necesitan recalcular firmas. Es el mismo principio del cifrado simétrico.

5. RSA:

RSA es un algoritmo criptográfico asimétrico. Aunque puede usarse para cifrar datos, en este contexto nos interesa su aplicación a firmas digitales, como alternativa a HMAC para firmar tokens.

Su objetivo y uso es el mismo que HMAC (firma, verificación) pero su implementación es distinta.

RSA resuelve ciertas limitaciones de HMAC:

- HMAC requiere que todos los servicios compartan la misma clave. (En caso de microservicios). Lo que provoca que en un sistema distribuido o con muchos equipos esto no escale de forma óptima; si se filtra la clave, cualquiera puede firmar tokens válidos.

Como vimos en el cifrado asimétrico, RSA genera una clave pública y una privada, vinculadas matemáticamente.

El proceso de firma es muy similar:

1. Toma el payload del JWT y se genera su hash.
2. Después lo cifra utilizando la clave privada.
3. Luego ese resultado se agrega también al token como “signature”.

Ahora, cualquiera que tenga la clave pública puede verificar los tokens sin exponer la clave privada y sin necesidad de conocerla. Pero sólo pueden firmar tokens los servicios que tengan esa clave privada.

- Lo que está firmado con la clave privada sólo se puede verificar su origen usando la clave pública.
- La responsabilidad está dividida entre nodos firmantes y nodos verificadores. Los verificadores no pueden firmar y los firmantes no pueden verificar, siguiendo el *principio de responsabilidad única*.
- Si el JWT trae configurado en su payload un algoritmo de firma distinto al definido en el servidor, se rechaza automáticamente y ni siquiera intentará firmarlo, siguiendo el *principio de fail-fast*. Esto sirve para prevenir ataques como el algorithm confusion.
- Si bien es más seguro, consume más recursos y poder de cómputo.
- No tiene mucho sentido utilizarlo en monolitos.

6. Hashing de contraseñas

Cuando nos registramos en un sitio e ingresamos nuestra contraseña, esta debe guardarse en la base de datos de forma segura para protegerla ante posibles robos o filtraciones.

- ¿Cómo se protege la contraseña?

Al crear una cuenta, tus datos se insertan en la base de datos, luego tu contraseña se hashea y pasa de ser la contraseña introducida, a ser un conjunto de caracteres que representa la contraseña, pero sin ser el texto original.

No necesitamos recuperar la contraseña original. Pero esto no es un problema, es intencional.

Si la contraseña en la base de datos está hasheada y nosotros para iniciar sesión ingresamos la contraseña en texto plano, ¿cómo se hace para saber que la contraseña es la misma?

Cuando iniciamos sesión con nuestra contraseña, el servidor vuelve a aplicarle el hash cuando la recibe, y si coincide con el hash guardado en la base de datos es porque la contraseña es la correcta, ya que siempre es el mismo resultado para la misma entrada.

Para guardar las contraseñas de forma segura se usan distintos algoritmos de hash.

Aunque, algunos algoritmos que se utilizan en otros escenarios (como SHA256, o MD5) no son recomendables en estos casos.
El problema de estos algoritmos es que son demasiado rápidos para realizarse, lo que causa que un atacante pueda probar millones de combinaciones por segundo para intentar adivinar las contraseñas.

Para hashear contraseñas se recomiendan otros algoritmos creados específicamente para esta tarea.
Como pueden ser por ejemplo bcrypt o argon2.

- ¿Por qué se recomiendan estos algoritmos?

Porque estos consumen más recursos, son más lentos e incluso se puede ajustar su consumo de CPU y memoria. Dificultando que alguien pueda realizar ataques de fuerza bruta probando millones de combinaciones.

Además, a estos algoritmos se les puede agregar *SALT*, como capa de seguridad extra al aplicar el hash a las contraseñas.

- ¿Qué es SALT exactamente?

Un SALT es un valor aleatorio que se agrega a la contraseña antes de hacer el hash. Sirve para asegurar que dos contraseñas iguales

produzcan hashes diferentes. Y gracias a esto se previenen algunos ataques como las famosas rainbow tables.

Ejemplo

Si tomamos una contraseña “contraseña123”, al aplicarle SALT, el resultado se vería algo así:

“c0ntr@s3ñ4!23”

Una vez tenemos la contraseña con SALT, le aplicamos el hash (usando por ejemplo un algoritmo como bcrypt), y el resultado es algo como esto:

“$2b$12$N9qo8uLOZMRZo5i.u8rJu5PfF6ylCx7jRy8T/HhXqJvW“.

7 .AES

AES (Advanced Encryption Standard) es un algoritmo de cifrado que se usa para proteger datos. Es un estándar común para cifrar comunicación HTTP de forma asimétrica. como vimos antes en TLS.

Pero otro de sus usos principales es cifrar datos en la persistencia, aunque esta tarea la realiza de forma simétrica, y se llama cifrado en reposo. Se usa para proteger datos los cuales sí se necesita recuperar su contenido original, como nombres, números de tarjetas, etc. Lo que significa que no se puede simplemente hashearlos como las contraseñas.

- ¿Cómo se aplica AES para cifrar en reposo?

1. El servidor toma los datos sensibles antes de almacenarlos.
2. Utiliza una clave AES para cifrarlos.
3. Una vez los datos estén cifrados, los guarda en la base de datos.
4. Cuando sea necesario acceder a ese dato, se descifra usando la misma clave.

Consideraciones:

- La clave AES debe protegerse, si se filtra, se puede descifrar todo.
- Hoy en día, para mayor seguridad, la clave se suele almacenar en servicios externos (como AWS KMS o Azure Key Vault) para aislar aún más.

Conclusión sobre las capas de seguridad

Podemos ver que hay al menos 4 capas de seguridad en la web:

1. El certificado X.509
2. El cifrado TLS (HTTPS)
3. La firma de tokens (HMAC, RSA)
4. El hashing de contraseñas (con capas extra como SALT) y cifrado en reposo

La metodología para comprobar la legitimidad de los datos o solicitudes suele repetirse:

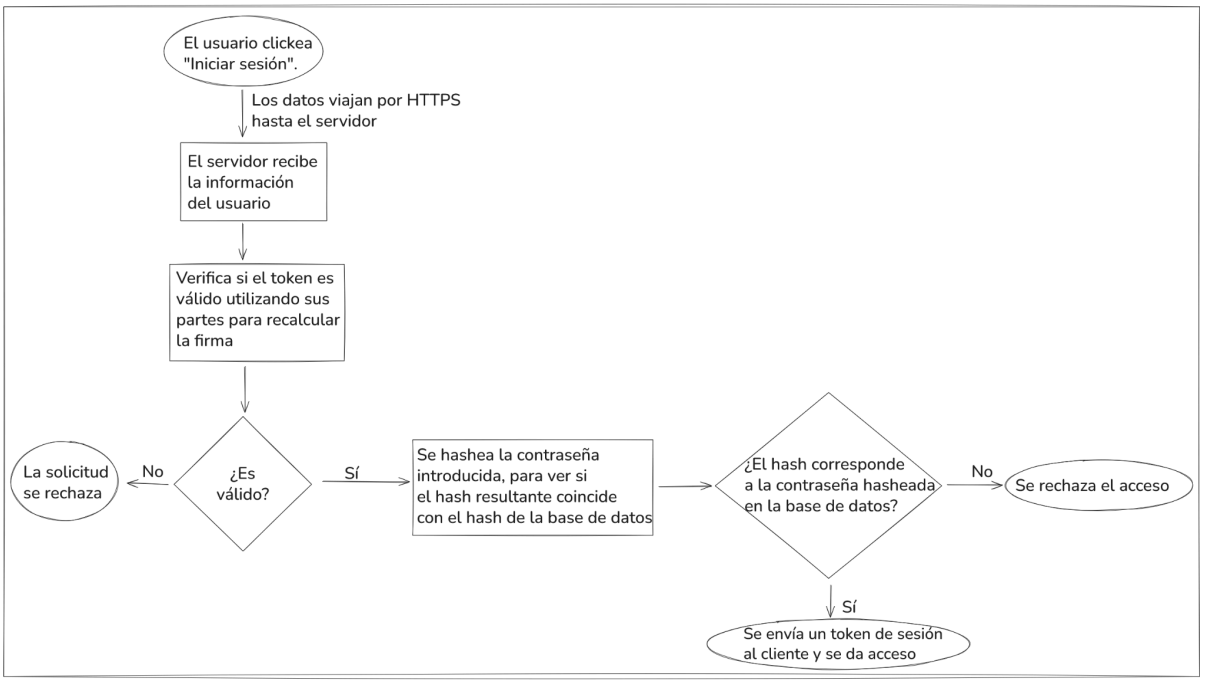
- Se recalculan strings usando la misma clave en caso de cifrado simétrico.
- Se recalculan strings usando una clave pública en casos de cifrado asimétrico, mientras la clave privada no es expuesta.

Diagramas de flujo y ejemplos

Flujo al registrarse en un sitio web

NOTA: Luego de firmarse el token, este puede enviarse de nuevo al cliente o no dependiendo de la configuración del sitio.

Pero, ¿qué ocurre al iniciar sesión en un sitio donde ya estamos registrados?



Este es un flujo simplificado de lo que ocurre al iniciar sesión.