
UD5. TÉCNICAS DE PROGRAMACIÓN SEGURA

JOAQUÍN FRANCO ROS
Programación de Servicios y Procesos

Contenido

1. Introducción	2
1.1 Prácticas de programación segura	2
1.2 Validación de entradas	4
2. Objetivos de la seguridad	6
3. Criptografía.....	7
3.1 Encriptación de la información	7
3.2 Resumen de mensajes	7
3.3 Criptografía de clave privada o simétrica	9
3.4 Criptografía de clave pública o asimétrica	9
3.5 Firma digital y certificados digitales.....	11
4. Protocolos seguros de comunicaciones	15
4.1 Protocolo criptográfico SSL/TLS	15
4.2 Otros protocolos seguros.....	16
5. Criptografía en Java.....	17
5.1 Arquitectura criptográfica de Java	17
5.2 Proveedores y motores criptográficos.....	18
5.3 Gestión de claves en Java	18
5.4 Resúmenes de mensajes con la clase MessageDigest	19
5.5 Firma digital con la clase Signature.....	20
5.6 Encriptación con la clase Cipher	21
6. Programación de aplicaciones con comunicaciones seguras	22
6.1 Creación de los certificados	22
6.2 Sockets SSL en servidor.....	25
6.3 Sockets SSL en el cliente	25
7. Bibliografía	27

1. Introducción

Hoy en día, las aplicaciones informáticas se utilizan para procesar o, al menos, como vía de acceso, a una gran cantidad de información en formato digital, lo que hace que ésta sea cada vez más y más importante. Piense que bajo el concepto información en formato digital se puede incluir desde la lista de clientes y contactos de una importante multinacional hasta el PIN, el número y la fecha de caducidad de su tarjeta de crédito.

Desafortunadamente, esto también significa que ésta se ha convertido en un objetivo muy atractivo para **entidades maliciosas** que quieren sacar provecho a costa de los demás. Por este motivo, hoy en día es muy importante **garantizar que las aplicaciones gestionan los datos de forma segura**. Afortunadamente, poco a poco, los desarrolladores de lenguajes de programación han cobrado conciencia de este hecho y han ido aportando una serie de herramientas que pueden facilitar alcanzar este objetivo.

Desde siempre ha sido necesario intercambiar información de manera que se garantizara que ninguna otra persona implicada en el proceso pudiera interceptarla y descubrirla. Ya sea un papiro egipcio con una propuesta de alianza, una carta manuscrita entre dos amantes en la edad media, las comunicaciones de radio entre tropas militares en plena guerra mundial o un mensaje de correo electrónico con información sensible por una multinacional en pleno siglo XXI. Desde que los humanos intercambiamos mensajes que esta problemática existe. En algunos casos, hay que garantizar la privacidad de los datos.

La **privacidad** es el servicio de seguridad que garantiza que unos datos no puedan ser accedidos libremente. Sólo tienen que poder ser accedidos por aquellos individuos que estén autorizados.

De entrada, el caso más fácil de ver dónde habría que aplicar este servicio es durante la **transmisión de un mensaje**. En el momento que el mensaje deja las manos del emisor y antes de que lleguen al receptor, éste es susceptible de ser interceptado. En las comunicaciones por Internet, el mensaje se va desplazando de manera que cualquier parte implicada en su recorrido puede hacer una copia.

La privacidad también afecta el **almacenamiento de los datos**. Por ejemplo, considere una aplicación de almacenamiento de archivos en la nube. Los datos físicos, los archivos en sí, están almacenadas en el disco duro de un servidor en algún lugar del mundo. Por lo tanto, cualquier persona que pueda pedir el acceso a este servidor puede acceder también libremente a sus datos. Todo lo que guarde en la nube no es sólo suyo, sino que también automáticamente de la empresa en la que se almacena. Por ejemplo, esto pasaría a Dropbox.

1.1 Prácticas de programación segura

Es muy importante tener siempre en cuenta los factores que pueden hacer tu aplicación vulnerable.

- **Fallas en el registro y monitoreo:** nos permitirá registrar qué está pasando en el sistema, quién hizo sesión, cuándo hizo inicio de sesión, cuánto tiempo, cuánto tiempo realizó cierta tarea, cuando se fue, etc.
- **Fallas en el software e integridad:** hay que revisar qué herramienta o librería instalamos en nuestro sistema pueda tener un objetivo malicioso. Lo mejor, es que la herramienta que utilicemos tenga una gran comunidad detrás y sea una herramienta común. Hay

que revisar las actualizaciones de la herramienta y cerciorarnos de que no nos descargamos una herramienta maliciosa y de una fuente confiable.

- **Fallas en la autenticación:** el sistema debe comprobar que el usuario que se está conectando a nuestra aplicación es quién dice ser. Hay que tener un límite de intentos y obligar a que el usuario sólo pueda usar contraseñas seguras. También hay que fijarse en el método de recuperación de contraseña. Otro tipo de vulnerabilidad es cuando guardamos las contraseñas en la base de datos sin cifrar. Mostrar el id de usuario en la url es otra vulnerabilidad. Hoy en día en las grandes empresas se lleva a cabo la autenticación multifactor.
- **Componentes vulnerables y desactualizados:** hay que actualizar el sistema operativo, los sistemas gestores de base de datos, el *framework*, etc. Si hay algún componente que no se utiliza es recomendable eliminarlo. Si vas a instalar algún software siempre hay que comprobar que sea de una procedencia lícita.
- **Configuración de seguridad incorrecta:** se deben cambiar las contraseñas y usuarios por defecto en las instalaciones de nuestro software, cerrar puertos que no se necesitan, limitar el acceso a ips específicas, crear usuarios con distintos privilegios, obligar a que los usuarios pongan una nueva contraseña cada cierto tiempo, etc.
- **Diseño inseguro:** cuando se hace un mal análisis de requisitos o un mal diseño del software. Se recomienda conocer buenas prácticas de codificación. El equipo de trabajo debe tener una serie de políticas y convenciones de código. Una solución para paliar este problema es hacer pruebas unitarias de seguridad en cada cambio que se haga al código.
- **Inyección:** la más conocida es la inyección SQL. Se produce cuando el sistema permite meter cosas que no esperas dentro de tus consultas. Esta vulnerabilidad también se puede dar en cualquier aplicación que envía a un servidor comandos. Es recomendable filtrar y validar la entrada del usuario para restringirla lo máximo posible. Se deben comprobar los límites, validar los ficheros de configuración, comprobar los parámetros de la línea de comandos, etc. Para las consultas SQL en Java, por ejemplo, es mejor utilizar los ***prepareStatement*** donde se puede comprobar el tipo que estás recibiendo.
- **Fallas criptográficas:** hay información sensible (tarjetas de crédito, información de salud, contraseñas, etc) que se debe guardar encriptada y con técnicas actualizadas y fuertes. Por ejemplo, hoy en día no es recomendable encriptar con MD5 o SHA-1. También es recomendable utilizar una semilla que hace que estos métodos generen resultados más difíciles de descryptar. Normalmente si se puede evitar guardar en tu base de datos información sensible mejor. Por ejemplo, regularmente cuando tienes que hacer pagos en línea, tendrás que basarte en un proveedor auditado que te va a dar un conjunto de bibliotecas que te van a ayudar a hacer este proceso seguro.
- **Pérdida de control de acceso:** esta vulnerabilidad radica en la jerarquización, roles y permisos de lo que un usuario puede hacer. Es importante en los sistemas, validar la propiedad de la información. Un usuario sólo debe poder acceder allí donde tenga permiso. Debe haber algún middleware que actúe siempre que un usuario autenticado intente acceder a algún módulo del sistema.

1.2 Validación de entradas

Una vía importante de errores de seguridad y de inconsistencia de datos dentro de una aplicación se produce a través de los datos que introducen los usuarios. Un fallo de seguridad muy frecuente, consiste en los errores basados en *buffer overflow*, se produce cuando se desborda el tamaño de una determinada variable, vector, matriz, etc y se consigue acceder a zonas de memoria reservadas.

La validación de datos permite:

- Mantener la consistencia de los datos. Por ejemplo, si a un usuario le indicamos que debe introducir su DNI éste debe tener el mismo formato siempre.
- Evitar desbordamientos de memoria *buffer overflow*. Al comprobar el formato y la longitud del campo evitamos que se produzcan los desbordamientos de memoria.

Java incorpora una potente y útil librería (**import java.util.regex**) para utilizar la clase **Pattern** que nos permite definir expresiones regulares. Las expresiones regulares permiten definir exactamente el formato de entrada de datos.

Algunos ejemplos de uso de expresiones regulares pueden ser:

- Para comprobar que la fecha leída cumple el patrón dd/mm/aaaa
- Para comprobar que un NIF está formado por 8 cifras, un guion y una letra
- Para comprobar que una dirección de correo electrónico es una dirección válida.
- Para comprobar que una contraseña cumple unas determinadas condiciones.
- Para comprobar que una URL es válida.
- Para comprobar cuantas veces se repite dentro de la cadena una secuencia de caracteres determinada.
- Etc.

Para utilizar las expresiones regulares debemos realizar los siguientes pasos:

1. Importamos la librería:

```
3 import java.util.regex.*;
```

2. Definimos *Pattern* y *Matcher*:

```
Pattern pat=null;  
Matcher mat=null;
```

3. Compilamos el patrón a utilizar

```
pat=Pattern.compile(patron);
```

Donde el patrón a comprobar es la parte más importante ya que es donde tenemos que indicar el formato que va a tener la entrada.

Algunos ejemplos de patrones son:

```
//Sólo números
Pattern patron3 = Pattern.compile(regex: "\\d+");

//Sólo letras
Pattern patron4 = Pattern.compile(regex: "[a-zA-Z]+");
```

4. Le pasamos al evaluador de expresiones el texto a comprobar.

```
mat=pat.matcher(texto_a_comprobar); mat=pat.matcher(texto_a_comprobar);
```

5. Comprobamos si hay alguna coincidencia:

```
//Devuelve true si la cadena cumple con el patrón
if(mat.matches()) {
    System.out.println("Coincide con el patrón");
} else {
    System.out.println("No coincide con el patrón");
}

//Devuelve true si se encuentra el patrón en una subcadena de la cadena
if (mat.find()) {
    System.out.println("Es un DNI válido");
} else {
    System.out.println("No es un DNI válido");
}
```

1. **Coincidencia de patrones:** las expresiones regulares le permiten definir patrones que pueden coincidir con secuencias específicas de caracteres dentro de una cadena.
2. **Comodines:** caracteres como puntos (.) o asteriscos (*) pueden actuar como comodines y representan cualquier carácter o cualquier número de caracteres, respectivamente.
3. **Clases de caracteres:** los corchetes [] se pueden usar para definir un conjunto de caracteres y el patrón coincidirá con cualquiera de esos caracteres.
4. **Cuantificadores:** símbolos como +, * y ? indican la cantidad o grupo de caracteres. Por ejemplo, * significa cero o más apariciones, + significa una o más apariciones y ? significa cero o una ocurrencia.
5. **Anclas:** el cursor (^) y el signo de dólar (\$) se utilizan como anclas para especificar que un patrón debe coincidir al principio (^) o al final (\$) de una cadena.
6. **Caracteres de escape:** algunos caracteres tienen significados especiales en expresiones regulares, como el punto (.) o el asterisco (*). Si desea hacer coincidir estos caracteres literalmente, debe evitarlos con una barra invertida (\).

7. **Agrupación y captura:** los paréntesis () se utilizan para agrupar partes de un patrón y también permiten capturar grupos para extraer partes específicas del texto coincidente.

Las expresiones regulares son soportadas en múltiples lenguajes:

Curso online gratuito: <https://regexlearn.com/learn>

Calculadoras online de expresiones:

- Regex101: <https://regex101.com>
- Regexr: <https://regexr.com>

API oficial de la clase *Pattern* en Java:

<https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>

2. Objetivos de la seguridad

Cuando se desarrollan aplicaciones con comunicaciones, se debe proporcionar seguridad tanto a la aplicación como a los datos transmitidos, ya que, las operaciones que se realizan por la red podrían ser interceptadas, y por tanto manipuladas por personas no autorizadas.

El estudio habitual de la seguridad se realiza considerando sus objetivos, que podemos clasificar del siguiente modo:

- **Confidencialidad.** Calidad del mensaje, comunicación o datos, para que únicamente accesibles para las personas autorizadas. Un ejemplo sería el cifrado de archivos.
- **Integridad.** Comprobar que no ha sido alterada cierta información o comunicación. Los datos son íntegros cuando permanecen invariables desde su origen y no han sido modificados, alterados o destruidos de forma accidental ni con mala intención. Un ejemplo sería la firma digital.
- **Disponibilidad.** Capacidad de un servicio, datos o sistema a ser accesible. La información ha de estar disponible para los usuarios autorizados cuando la necesiten. Se habla también de alta disponibilidad cuando los datos y aplicaciones se encuentren disponibles en todo momento sin interrupciones.

Otros objetivos secundarios:

- **Autenticidad:** Capacidad de asegurar que el emisor de un mensaje es quien diceser y no un tercero que esté intentando suplantarle.
- **No repudio.** Permite probar la participación de las partes en una comunicación.
- **Trazabilidad:** es un conjunto de procedimientos que permiten registrar e identificar las acciones realizadas sobre los datos o aplicaciones de un sistema informático.
- **Control de acceso:** capacidad del sistema de restringir los accesos a los recursos en función de los permisos asignados a cada usuario.

3. Criptografía

La criptografía es la técnica de alterar las representaciones lingüísticas de un mensaje de modo que se puedan enviar mensajes confidenciales que únicamente puedan ser comprendidos por personas autorizadas. Esta operación se puede conseguir mediante dos métodos:

- a) Utilizando un mecanismo de cifrado que solo conozcan emisor y receptor. Por ejemplo, haciendo desplazamientos y sustituciones de letras.
- b) Haciendo que el mecanismo sea conocido por todos, pero se utiliza una clave o llave que regula el comportamiento del algoritmo.



3.1 Encriptación de la información

La principal técnica para garantizar la confidencialidad es el cifrado de información. Un algoritmo de cifrado especifica dos transformaciones:

- El **cifrado** es la conversión del texto claro en texto cifrado o criptograma mediante el empleo de una función parametrizada mediante una clave de codificación.
- El **descifrado** es el proceso inverso, para el cual se emplea otra función que necesita como parámetro una clave de descifrado.

Los métodos clásicos de cifrado hacían énfasis en el secreto del algoritmo de cifrado, luego en muchos de ellos no existía el concepto de clave. En los sistemas criptográficos modernos, la seguridad de un sistema cifrado radica casi por completo en la **privacidad de las claves usadas**, dado que los algoritmos se suponen públicos. Por tanto, los ataques a realizar por un criptoanalista están destinados al descubrimiento de las claves.

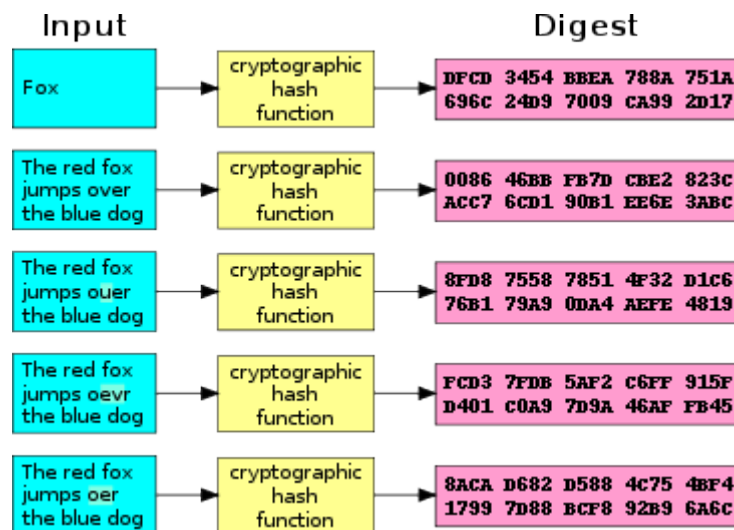
Los algoritmos de cifrado pueden ser divididos en Simétricos (de clave privada) y Asimétricos (de clave pública).

3.2 Resumen de mensajes

Una manera de facilitar la gestión de claves criptográficas es generarlas a partir de una contraseña legible. Que elija una contraseña fácil o difícil de adivinar ya es otro tema.

Para ello puede aprovechar la particularidad de que cualquier secuencia de bytes de tamaño correcto puede servir como una clave. Basta, partiendo de la contraseña, generar tantos bytes como sea necesario. Por lo que, dada una contraseña, sólo ésta genere una secuencia de bytes dada. Y que nunca sea posible, o al menos extremadamente improbable, que dos contraseñas diferentes lleguen a generar la misma clave.

Una función de **resumen o hash** es un método matemático para generar claves o llaves que representen de forma casi unívoca a un conjunto de datos. La operación se realiza sobre el conjunto de datos de cualquier longitud y su salida será una huella digital de tamaño fijo e independiente del documento original, de forma que no sea legible.



Requisitos que debe cumplir una función hash:

- Que sea imposible obtener el texto original a partir de la huella digital.
- Que se imposible encontrar un conjunto de datos diferentes que tengan la misma huella (aunque en algunos casos particulares este requisito puede no cumplirse debido al tamaño de 160 bits SHA-1, ocasiona que haya la misma huella para distintos documentos).
- Que pueda transformar un texto de longitud variable en una huella de tamaño fijo (SHA-1 es de 160 bits).
- Que sea fácil de usar e implementar.

Ejemplos de funciones hash:

- MD5 (128 bits).
- SHA-1 (160 bits).
- SHA-256 (256 bits)

Software, por ejemplo:

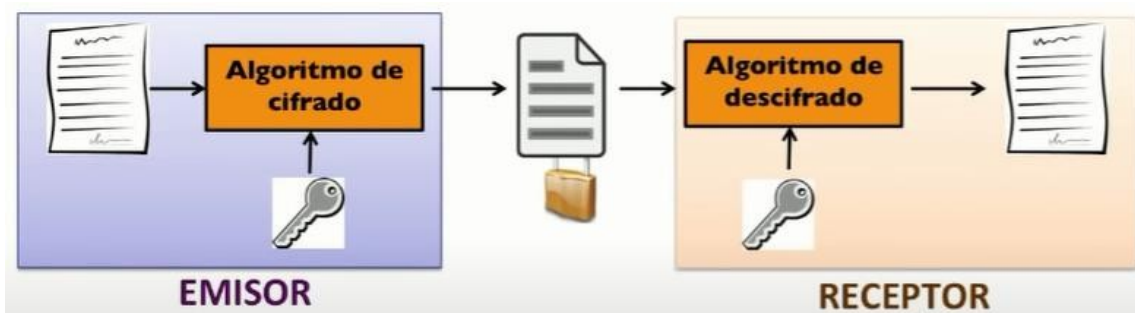
- GNU/Linux: md5sum y sha1 sum (por defecto instalados).
- Windows: Snap MD5 (MD5 y SHA-1), herramienta externa.

3.3 Criptografía de clave privada o simétrica

La criptografía con clave secreta usa la **misma clave** para cifrar y descifrar un mensaje, luego su seguridad se basa en el **secreto** de dicha clave. Generalmente se usan **dos funciones**: una para realizar la codificación y otra para realizar la decodificación.

Su principal **desventaja** es que hace falta que el emisor y el receptor compartan la clave, luego es preciso que la **clave viaje de un modo seguro** del origen al destino.

Uno de los métodos de clave secreta más conocido, usado en aplicaciones de tipo comercial donde no se requieren grandes niveles de seguridad es **DES** (*Data Encryption Standard*), que se basa en una serie de permutaciones, sustituciones y XOR que tienen la propiedad de ser **reversibles** aplicando la misma clave de cifrado. Debido al actual desarrollo de la tecnología, el algoritmo ha visto degradado su grado de seguridad. Por ello se han ideado otros (Triple-DES, AES...).



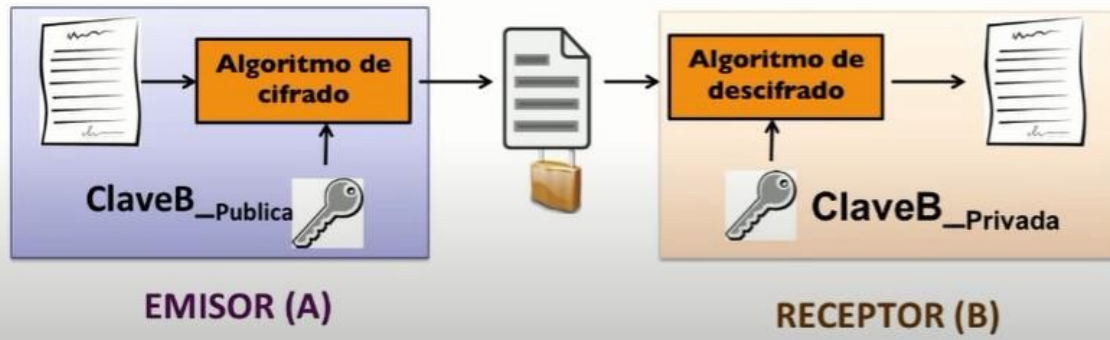
3.4 Criptografía de clave pública o asimétrica

Evita el problema de distribución de las claves de manera segura. La criptografía con clave pública usa **dos claves**, una para cifrar y otra para descifrar, **relacionadas matemáticamente** de forma que los datos codificados por una de las dos sólo pueden ser decodificados por la otra.

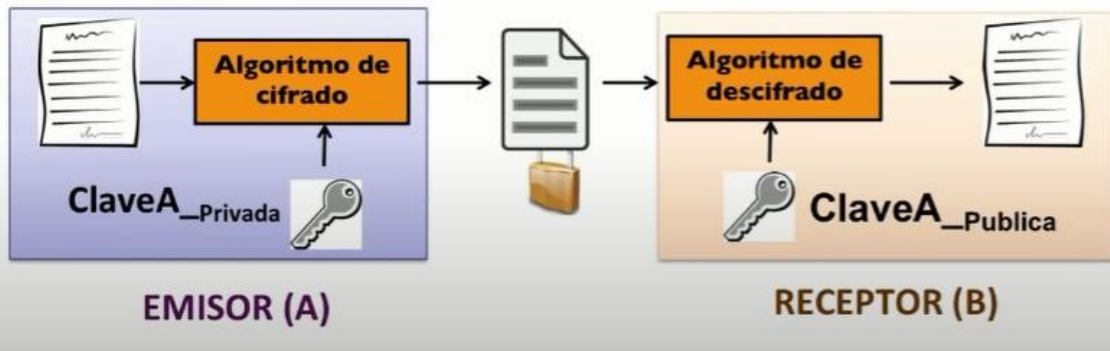
Cada usuario poseerá dos claves, haciendo una de ellas **pública** (la distribuye) y la otra **privada**. Los algoritmos pueden ser implementados de dos formas, dependiendo de si la clave pública se emplea como clave de cifrado o de descifrado.

- En el primer caso, cuando un usuario A quiere enviar información a un usuario B, usa la clave pública de B para encriptar los datos. El usuario B usará su clave privada para desencriptar los datos. Este modo se emplea para proporcionar servicios de **confidencialidad**, pues sólo el usuario B puede descifrar los datos que se le envían. El problema es que, si se quiere enviar un mensaje a varios usuarios, es necesario manejar una clave pública por cada mensaje.
- En el otro modo de operación que funciona en algoritmos como **RSA**, el propietario de las claves es quien cifra la información usando su clave privada, de modo que cualquiera que conozca la clave pública puede descifrarla. El sistema no proporciona confidencialidad, pero si se puede emplear para garantizar el servicio de **autenticación**, dado que la obtención del texto correcto es una garantía de que el emisor del mensaje es el propietario de la clave pública usada. Se trata de la base de la técnica conocida como **firma digital**.

CASO DE USO 1



CASO DE USO 2



3.5 Firma digital y certificados digitales

Firma digital

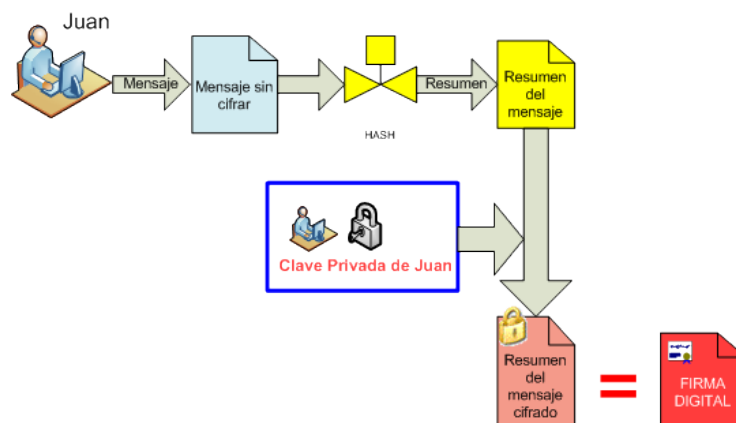
Consiste en adjuntar con el mensaje un hash del mismo. El conjunto se envía a un destinatario quien vuelve a calcular el hash y si su resultado coincide con el hash que adjuntó el emisor, entonces se puede asegurar que el mensaje original no fue alterado, por tanto, proporciona **integridad**.

Cabe la posibilidad de que un tercer agente intercepte el mensaje y el hash originales y los sustituya por otros de producción propia y los envíe al destinatario. Para solucionar este problema se utiliza la técnica de criptografía asimétrica para cifrar el hash. El emisor cifra con su clave privada el hash calculado a partir del mensaje que enviará al receptor. El resultado de este cifrado es lo que se denomina la **firma digital**. Esta firma solo puede ser descifrada en el destino por el receptor con clave pública del emisor. El receptor comprueba si el hash que ha descifrado y el hash calculado por sí mismo coinciden. Si es así el mensaje es original puesto que solo puede haber descifrado correctamente la firma digital si fue cifrada con la clave privada del emisor en origen, lo que es absolutamente cierto puesto que esta clave privada solo está en posesión del emisor.

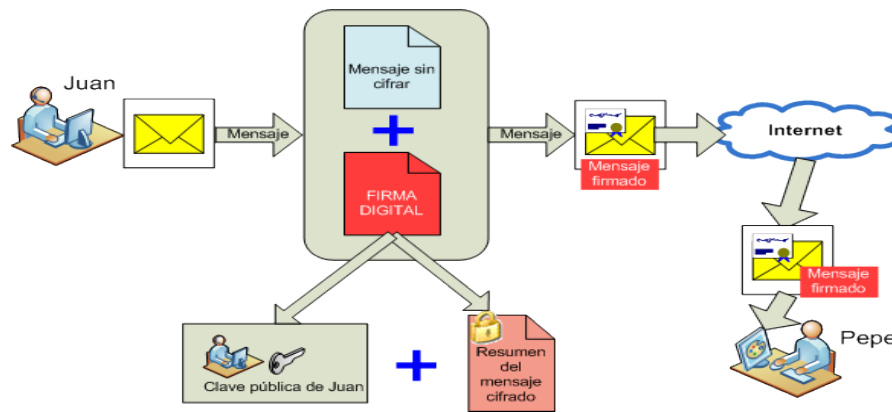
La **firma digital** es el mecanismo criptográfico que traslada todas las propiedades de la firma manuscrita del mundo físico a la información en formato digital.

En resumen, la firma digital proporciona:

- **Integridad:** Le dota de veracidad, garantizando que el mensaje no ha sido modificado y que se respeta su integridad.
- **Autenticación:** poder garantizar cuál es la identidad del autor del documento, evitando que sea suplantado.
- **No repudio:** El usuario que lo ha firmado no puede repudiarlo.
- Ejemplo de firma digital de generación de una firma a partir de un mensaje:



- Ejemplo de envío de un mensaje con firma y recepción y verificación



El proceso de firma tiene los siguientes pasos:

- Se calcula el resumen hash del documento.
- El resumen se cifra con la clave privada del usuario. Así se asegura que el único que ha firmado el documento es el usuario, porque es el único que conoce la clave privada.
- El resultado se conoce como firma digital del documento.

El proceso de verificación tiene los siguientes pasos:

- La firma se descifra usando la clave pública del usuario (cualquiera la puede tener, por lo tanto, cualquiera puede verificarla).
- Se obtienen el valor resumen del documento firmado.
- Se comparan los dos resúmenes y si coinciden la firma es válida.

Certificado digital

Un certificado digital es una **cadena de caracteres que proporciona identidad a un usuario**, un servidor web, una dirección de correo electrónico, etc. La criptografía asimétrica tiene el problema de que no se sabe con seguridad que la clave pública que hemos recibido proviene de donde tiene que provenir. El certificado digital surge a partir de la necesidad de que **una tercera entidad llamada autoridad de certificación (CA)** intervenga en la administración de claves públicas y asegure que estas son de confianza.

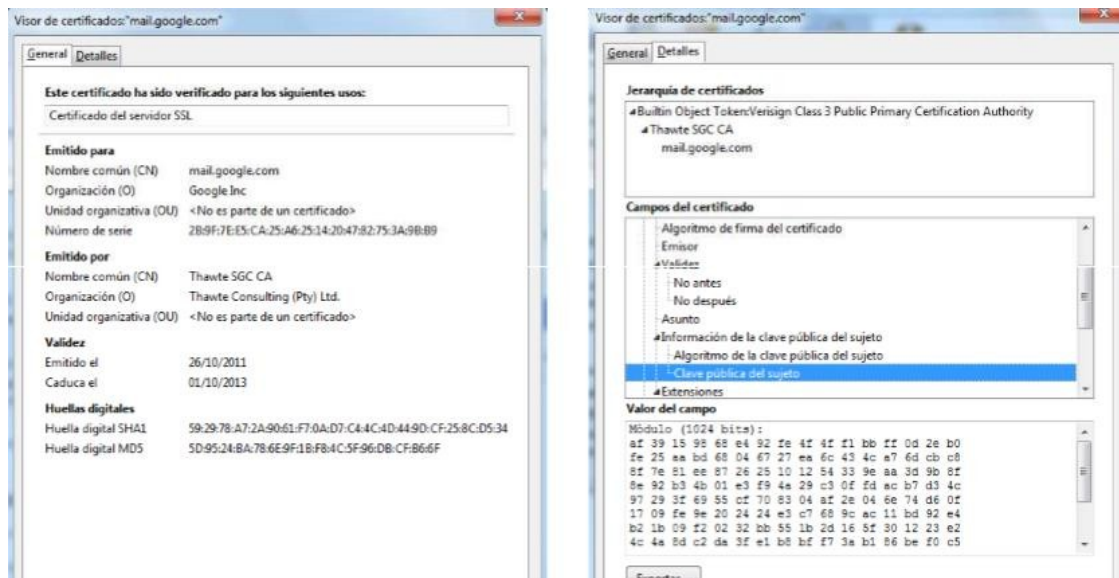
Un certificado digital se compone de muchos elementos, pero entre ellos no deben faltar los siguientes:

- Una clave pública del que será el propietario del certificado.
- La identidad del propietario. Nombre, DNI, CIF, correo, etc.
- La identidad del emisor del certificado.
- La firma digital de la autoridad de certificación (CA) que tanto emisor y receptor conocen y confían. La autoridad certificadora firma el certificado con su clave privada. Esta firma digital garantiza que la clave pública que contiene el certificado se corresponde al propietario del mismo.

Las identidades que contiene un certificado digital, tanto del propietario como del emisor, se codifican mediante un nombre distinguido (o *Distinguished Name*, en inglés, también abreviado como DN).

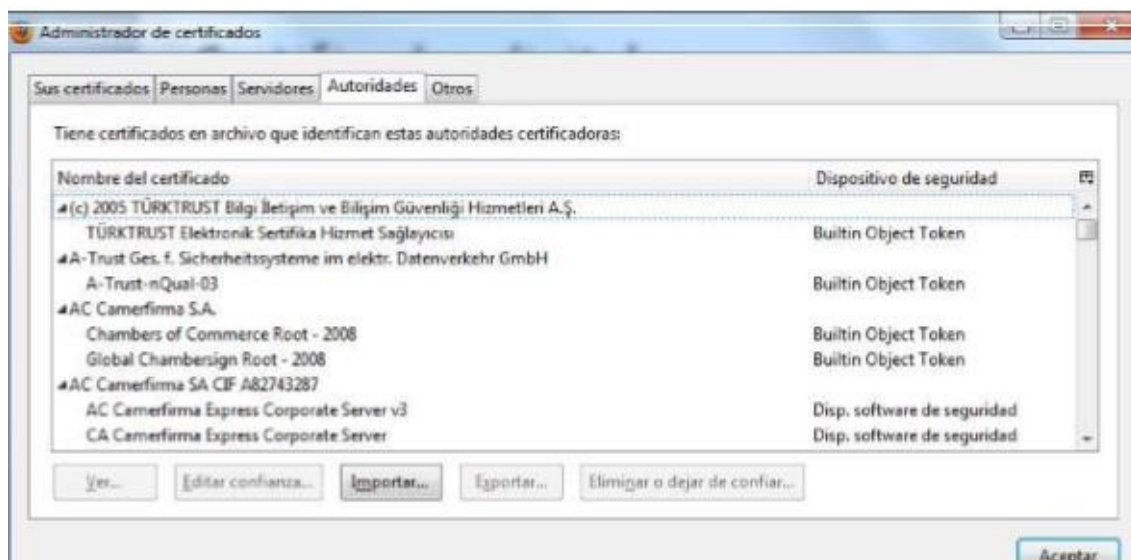
- **Formato X.509.**

- Basado en criptografía asimétrica y firma digital.



- **Certificados raíz.**

- Emitidos por autoridades de certificación para sí mismas con su clave pública.
- Son necesarios para verificar la autenticidad de los certificados que emiten.



- **Certificados auto-firmados:**

- Es el que se realiza sin la intervención de una CA.
- No existe ningún mecanismo automático que garantice su autenticidad.

Los certificados digitales son útiles para:

- Verificar que nos conectamos con el servidor que deseamos y no con un impostor.
- El servidor sepa que cuando nos conectamos con él somos nosotros y no otra identidad.

Los certificados **auto-firmados** resultan muy útiles como certificados de prueba para aplicaciones en desarrollo, ya que permiten la máxima flexibilidad en los datos que contienen.

En la mayoría de los casos, un certificado no será auto firmado, ya que precisamente en Internet a menudo las partes implicadas no se conocen y siempre existe el peligro de que un atacante use sistemas bien sofisticados para intentar hacerse pasar por alguien de manera expresamente malintencionada.

Una de las responsabilidades de una CA es disponer de toda una infraestructura para poder llevar a cabo este servicio, así como poner a disposición del público su clave pública de forma segura y poder dar de baja certificados emitidos.

Se denomina una **infraestructura de clave pública (o PKI, Public Key Infrastructure)** al marco para desplegar un mecanismo seguro de intercambio de claves públicas.

Algunos de los servicios ofrecidos por una ICP (PKI) son los siguientes:

- Registro de claves: emisión de un nuevo certificado para una clave pública.
- Revocación de certificados: cancelación de un certificado previamente emitido.
- Selección de claves: publicación de la clave pública de los usuarios.
- Evaluación de la confianza: determinación sobre si un certificado es válido y qué operaciones están permitidas para dicho certificado.
- Recuperación de claves: posibilidad de recuperar las claves de un usuario.

Las Infraestructuras de Clave Pública ICP (PKI) están compuestas por:

- **Autoridad de Certificación (AC):** realiza la firma de los certificados con su clave privada y gestiona la lista de certificados revocados.
- **Autoridad de Registro (AR):** es la interfaz hacia el mundo exterior. Recibe las solicitudes de los certificados y revocaciones, comprueba los datos de los sujetos que hacen las peticiones y traslada los certificados y revocaciones a la AC para que los firme.

Un ejemplo es la Fábrica Nacional de Moneda y Timbre (FNMT), que generan certificados válidos para hacer la declaración de la renta por Internet, entre otras cosas.

4. Protocolos seguros de comunicaciones

Si combinamos la criptografía con tecnologías de comunicación en red, entonces hablaremos de protocolos seguros de comunicaciones o protocolos criptográficos.

Algunos de estos protocolos son los siguientes:

- **SSL**: Proporciona comunicación segura en una conexión cliente/servidor, frente a posibles ataques en la red, como por ejemplo el problema que ya te comentamos al hablar de la criptografía asimétrica, conocido como man in the middle u "hombre en el medio".
- **TLS**: Es una evolución de SSL, que amplía los algoritmos criptográficos que puede utilizar.

En el siguiente esquema puedes ver el nivel, según la pila TCP/IP, en el que se utilizan estos protocolos:



Tanto SSL como TLS son protocolos criptográficos que se ejecutan en una capa intermedia entre un protocolo de aplicación, y un protocolo de transporte como TCP o UDP, por lo que pueden ser utilizados para el cifrado de protocolos de aplicación como Telnet, FTP, SMTP, IMAP o el propio HTTP.

Cuando un protocolo de aplicación, como HTTP o Telnet se ejecuta sobre un protocolo criptográfico como SSL o TLS, se habla de la versión segura de ese protocolo, por ejemplo:

- **SSH**: Protocolo usado exclusivamente en reemplazo de Telnet, para comunicaciones seguras.
- **HTTPS**: Protocolo usado exclusivamente para comunicaciones seguras de páginas web.

Por tanto, podemos decir que el protocolo HTTPS no es ni más ni menos que el protocolo HTTP, pero ejecutándose sobre el protocolo criptográfico SSL, y por ello se dice que HTTPS es un protocolo seguro.

4.1 Protocolo criptográfico SSL/TLS

Es un protocolo que trabaja sobre el nivel de transporte y es independiente del protocolo del nivel superior. Su objetivo es proporcionar una comunicación segura entre cliente y servidor, ofreciendo los siguientes tres servicios: **confidencialidad**, **autenticación**, e **integridad** de mensajes.

El protocolo TLS es una evolución del protocolo SSL. Es una versión avanzada que proporciona más algoritmos criptográficos y mayor seguridad, pero opera de igual forma que SSL.

SSL permite que un servidor SSL y un cliente SSL se autenticuen, permitiendo a ambas máquinas establecer una conexión encriptada. Para ello utiliza criptografía asimétrica y criptografía simétrica (**criptografía híbrida**). Para cifrar, el cliente y el servidor eligen una clave con la que utilizar un algoritmo simétrico. Para proteger dicha clave utilizan criptografía asimétrica.

De manera simplificada las **fases que se siguen con SSL**.

1. Fase inicial. Negociar los algoritmos criptográficos a utilizar.
2. Fase de autenticación y clave de sesión. Intercambio de claves y autenticación mediante certificados, utilizando criptografía asimétrica. Se crea la clave para cifrar los datos transmitidos con criptografía simétrica, para agilizar las transmisiones.
3. Fase fin. Verificación del canal seguro.
4. A partir de aquí, comunicación segura.

Este protocolo también es utilizado para crear redes privadas virtuales **VPNs**.

4.2 Otros protocolos seguros

HTTPS

La **idea principal de HTTPS es la de crear un canal seguro sobre una red insegura, utilizando cifrado basado en SSL/TLS, el puerto 443** y proporcionando protección frente ataques como el del "hombre en el medio" o man-in-the-middle, siempre que se utilicen métodos de cifrado adecuados y que el certificado del servidor sea verificado y resulte de confianza.

La confianza inherente de HTTPS está basada en una Autoridad de Certificación que viene preinstalada en el software del navegador.

Para saber si la página web que estamos visitando es segura y por tanto utiliza el protocolo HTTPS, debemos observar que en la barra de navegación a la izquierda aparece HTTPS y no HTTP, pues a veces, aunque el propio navegador indique las páginas seguras mediante la imagen de un candado amarillo a la derecha de la barra de direcciones, éste puede haber sido colocado por un atacante.

SSH

SSH (o Secure SHell) es el nombre de un protocolo y del programa que lo implementa cuya principal función es el **acceso remoto a un servidor** por medio de un canal seguro en el que toda la información está cifrada. Además de la conexión a otros dispositivos, SSH permite copiar datos de forma segura (tanto archivos sueltos como simular sesiones FTP cifradas), gestionar claves RSA para no escribir contraseñas al conectar a los dispositivos y pasar los datos de cualquier otra aplicación por un canal seguro tunelizado mediante SSH y también puede redirigir el tráfico del (Sistema de Ventanas X) para poder ejecutar programas gráficos remotamente. El puerto TCP asignado es el 22.

5. Criptografía en Java

Las principales APIs de Java para seguridad, criptografía, infraestructura de clave pública, autenticación, comunicación segura y control de acceso, permiten integrar la seguridad en nuestras aplicaciones. Algunas son:

- **JCA (Java Cryptography Architecture)** → permite acceder y desarrollar funciones criptográficas en la plataforma Java. Proporciona la infraestructura para la ejecución de los servicios de cifrado, firmas digitales, hash, validación de certificados, encriptación, generación y gestión de claves y de números aleatorios.
- **JSSE (Java Secure Socket Extension)** → Conjunto de paquetes Java provistos para la comunicación segura en Internet. Implementa SSL y TLS e incluye funciones de cifrado de datos, autenticación del servidor, integridad de los mensajes y autenticación del cliente.
- **JAAS (Java Authentication and Authorization Service)** → Interfaz que permite a las aplicaciones Java acceder a servicios de control de autenticación y acceso. Para:
 - Autenticar usuarios que permita saber quién está ejecutando el código Java.
 - Autorizar a usuarios para garantizar que quién está ejecutando dispone de los permisos apropiados.

5.1 Arquitectura criptográfica de Java

La arquitectura JCA está diseñada en torno a los siguientes principios:

- Independencia de la aplicación: los algoritmos criptográficos están implementados por los proveedores criptográficos. Así que, el usuario no tiene que implementarlos.
- Interoperabilidad: los diferentes proveedores son compatibles con todas las aplicaciones.
- Extensibilidad: se pueden instalar proveedores personalizados que implementan nuevos servicios.

La extensión criptográfica de Java JCE (Java Cryptography Extension) está dentro del API JCA, incluida en el JDK. Tiene dos componentes:

- El entorno para definir y apoyar servicios criptográficos para las implementaciones. Incluye los paquetes:
 - `java.security`: clases abstractas e interfaces que proporcionan manejo de certificados, claves, resúmenes y firmas digitales.
 - `javax.crypto`: clases e interfaces para realizar operaciones criptográficas como encriptación/desencryptación, generación de claves, etc.
 - `javax.crypto.spec`: incluye varias clases de especificación de claves y de parámetros de algoritmos.
 - `javax.crypto.interfaces`: presenta las interfaces de las claves empleadas en algoritmos de tipo Diffie-Hellman (algoritmo de establecimiento de claves entre pares).

5.2 Proveedores y motores criptográficos

Los proveedores con implementaciones criptográficas reales, Sun, SunRsaSign, SunJCE que se encargan de dar la implementación de los algoritmos para el programador. En el fichero `jdk_14.0.2/conf/java.security` están los proveedores definidos:

```
# List of providers and their preference orders (see above):
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=sun.security.ec.SunEC
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
security.provider.7=com.sun.security.sasl.Provider
...
```

El concepto de proveedor se hace bajo la clase `Provider` de `java.security`. Tiene los métodos para acceder al nombre del proveedor, número de versión e información sobre la implementación de los algoritmos, generación de claves, firmas y resúmenes.

5.3 Gestión de claves en Java

En la generación de claves se utilizan números aleatorios seguros, que son números aleatorios que se generan en base a una semilla. Esto permite crear algoritmos seguros, pues será muy difícil determinar los valores generados sin conocer la semilla.

El paquete `java.security` proporciona las siguientes clases para la gestión de claves:

- El interface **key**, permite la representación de claves, su almacenamiento y envío de forma serializada dentro de un sistema.
 - `getAlgorithm()`. Devuelve el nombre del algoritmo con el que se ha generado la clave (RSA, DES, etc).
 - `getEncoded()`. Devuelve la clave como un array de bytes.
 - `getFormat()`. Devuelve el formato en que está codificada la clave.
- La clase **KeyPairGenerator**, permite la generación de claves públicas y privadas (asimétricas). Genera objetos de tipo **KeyPair**, que a su vez contienen un objeto del tipo **PublicKey** y otro **PrivateKey**.
 - El método `getInstance` de la clase `KeyPairGenerator` sirve para obtener un objeto `KeyPairGenerator` pasándole el algoritmo.
 - El método `initialize()` permite establecer el tamaño de la clave y el número aleatorio a partir de cual será generada.
 - El método `generatePair()` sirve para generar un objeto del tipo `KeyPair`.
- La clase **KeyGenerator** permite la generación de claves simétricas. Generar objetos del tipo **SecretKey**.
 - El método `init()` permite establecer el tamaño de la clave y el número aleatorio a partir del cual será generada.

- La clase **SecureRandom** permite generar números aleatorios seguros.
 - El método `setSeed()` permite establecer el valor de la semilla.
 - El constructor `SecureRandom()` utiliza la semilla del proveedor SUN.
 - El método `next()` y `nextBytes()` obtienen el valor de los números generados.
- Si queremos guardar las claves simétricas en un fichero, haremos uso del método **`java.security.Key.getEncoded()`** que la codificará la clave en forma de bytes. Con **`javax.crypto.SecretKeySpec`** pasándole los bytes de la clave y el algoritmo obtenemos la clave en forma de objeto.

```
byte[] bytesClave = clave.getEncoded();
SecretKeySpec clave = new SecretKeySpec(bytesClave,
"Blowfish");
```

En claves asimétricas existe una mayor complejidad y se haría de la siguiente manera:

Para la clave pública hay que codificarla usando X.509 y pasarla a un `KeyFactory`.

```
X509EncodedKeySpec spec = new X509EncodedKeySpec(bytesClave);
KeyFactory factoria = KeyFactory.getInstance("RSA");
PublicKey clavePublica = factoria.generatePublic(spec);
```

Para la clave privada hay que codificarla en formato PKCS#8 y pasarla también a un `KeyFactory`.

```
PKCS8EncodedKeySpec spec = new
PKCS8EncodedKeySpec(bytesClave);
KeyFactory factoria = KeyFactory.getInstance("RSA");
PrivateKey clavePrivada = factoria.generatePrivate(spec);
```

5.4 Resúmenes de mensajes con la clase `MessageDigest`

Un algoritmo de *resumen de mensajes* o *función de dispersión criptográfica* es el que toma como entrada un mensaje de longitud variable y produce un resumen de longitud fija. Se conocen con el nombre de *message digest*, *digest* o *hash* y el algoritmo *message digest algorithm* o *one way hash algorithm*.

Estos algoritmos deben tener tres propiedades para ser criptográficamente seguros:

1. No debe ser posible averiguar el mensaje de entrada basándose sólo en su resumen, es decir, el algoritmo es una función irreversible de una sola dirección.
2. Dado un resumen debe ser imposible encontrar un mensaje que lo genere.
3. Debe ser computacionalmente imposible encontrar dos mensajes que generen el mismo resumen.

Los algoritmos de este tipo se emplean en la generación de *códigos de autenticación de mensajes* y en las *firmas digitales*.

Entre los algoritmos diseñados para procesar estos mensajes están el **SHA1** y **MD5**.

La clase **MessageDigest** nos deja implementar los algoritmos de resumen de mensajes como MD5, SHA-1, SHA-256. La forma de acceder al constructor es con el método:

getInstance (String algoritmo)

Método	Acción
Public static MessageDigest getInstance (string algoritmo)	Devuelve un mensaje digest que implementa el algoritmo de resumen especificado. Los proveedores de seguridad se buscan en el orden establecido en java.security. Lanza NoSuchAlgorithmException si no encuentra proveedor.
Public static MessageDigest getInstance (string algoritmo, string proveedor)	Busca el proveedor dado. Lanza NoSuchAlgorithmException si no encuentra proveedor el nombre del proveedor.
Void update (byte input) Void update (byte[] input)	Hace el resumen del byte especificado o del array de bytes indicado
Byte[] digest()	Completa el cálculo del valor hash y devuelve el resumen que se obtiene
Byte[] digest(byte[] input)	Actualiza el resumen usando el array de bytes y completa el cálculo del resumen
Void reset()	Reinicializa el objeto resumen para volver a usarlo
Int getDigestLength()	Da la longitud en bytes del resumen o 0 si no la soporta el servidor
String getAlgorithm()	String que identifica al algoritmo
Provider getProvider()	Da el proveedor del objeto
Static boolean isEqual (byte[] digest, byte[] digesttb)	Comprueba si los resúmenes son iguales (true)

5.5 Firma digital con la clase Signature

La clase **Signature** del paquete **java.security** permite realizar una firma digital, así como hacer su verificación.

Los pasos que debes seguir para realizar la firma de un mensaje y después verificarla son los siguientes:

- Generar las claves públicas y privadas mediante la clase KeyPairGenerator:
 - La PrivateKey la utilizaremos para firmar.
 - La PublicKey la utilizaremos para verificar la firma.
- Realizar la firma digital mediante la clase Signature y un algoritmo asimétrico, por ejemplo DSA.
 - Crearemos un objeto Signature.
 - Al método initSign() le pasamos la clave privada.
 - El método update() creará el resumen de mensaje.
 - El método sign() devolverá la firma digital.
- Verificar la firma creada mediante la clave pública generada.
 - Al método initVerify() le pasaremos la clave pública.
 - Con update() se actualiza el resumen de mensaje para comprobar si coincide con el enviado.

- El método `verify()` realizará la verificación de la firma.

EL algoritmo de firma digital DSA viene implementado en el JDK de SUN, es parte del estándar de firmas digitales DSS y con él se pueden utilizar los algoritmos de resumen MD5 y SHA-1.

5.6 Encriptación con la clase Cipher

La clase **Cipher** permite realizar encriptación y desencriptación, tanto con clave pública como privada. Para ello:

- Mediante `getInstance()` se indica el algoritmo y proveedor que utilizará el objeto cifrador Cipher.
- Mediante el método `init()` se indicará el modo de operación del objeto Cipher, por ejemplo encriptar o desencriptar.
- Mediante los métodos `update()` y `doFinal()` se insertarán datos en el objeto cifrador. Update y doFinal hacen lo mismo sólo que cuando el bloque a cifrar es muy grande hay que hacer varios update y el último tiene que ser un doFinal para que añada un padding si es necesitado por el algoritmo.

Podemos utilizar modos de operación, entre ellos:

- `ENCRYPT_MODE`. Es el modo encriptación.
- `DECRYPT_MODE`. Es el modo desencriptación.
- `WRAP_MODE`. Convierte la clave en una secuencia de bytes para transmitirla de forma segura.
- `UNWRAP_MODE`. Permite obtener las claves generadas con `WRAP_MODE`.

La encriptación mediante un objeto Cipher puede ser:

- De bloque o Block Cipher. El texto a cifrar se divide en bloques de un tamaño fijo de bits, normalmente 64 bits. Cada uno de estos bloques se cifra de manera independiente, y posteriormente se construye todo el texto cifrado. Si el texto a cifrar no es múltiplo de 64 se completa con un relleno o padding.
Por ejemplo: `Cipher.getInstance("Rijndael/ECB/PKCS5Padding")` indica que:
 - Se utiliza el algoritmo Rijndael.
 - El modo es ECB (Electronic Code Book).
 - El relleno es PKCS5 Padding.
- De flujo o Stream Cipher. El texto se cifra bit a bit, byte a byte o carácter a carácter, en lugar de bloques completos de bits. Resulta muy útil cuando hay que transmitir información cifrada según se va creando, es decir, se cifra sobre la marcha.

6. Programación de aplicaciones con comunicaciones seguras

Vamos a ver cómo programar aplicaciones cliente/servidor seguras utilizando sockets seguros o SSLSockets.

Cuando dos socket SSL intentan establecer conexión (un socket cliente y un socket servidor), lo primero que hacen es "presentarse" el uno al otro y cada uno de ellos comprueba que el otro es de "confianza". Si todo va bien, la conexión se establece. Si uno no confía en el otro, la conexión no se establece. Para ello, hay que crear primero unos certificados. En este caso, vamos a crear dos uno para el servidor y otro para el cliente. Estos certificados irán almacenados en cada ordenador en un fichero almacén de certificados que son de confianza para el usuario.

6.1 Creación de los certificados

Para la generación de certificados utilizaremos la herramienta keytool vista en el tema anterior.

```
jebaSSL>keytool -genkey -keyalg RSA -alias serverKey -keystore serverKey.jks -storepass servpass
```

- Keytool está en el directorio bin donde tengamos instalado Java.
- -genkey le decimos que genere un certificado.
- -keyalg le decimos el algoritmo de encriptado.
- -alias le decimos el alias que va a tener en certificado en el almacén.
- -keystore es el fichero que será el almacén de certificados.
- -storepass es la clave del acceso al almacén

A continuación, nos preguntará unos datos asociados al servidor.

```
¿Cuáles son su nombre y su apellido?
[Unknown]: Server
¿Cuál es el nombre de su unidad de organización?
[Unknown]: Unidad
¿Cuál es el nombre de su organización?
[Unknown]: Organizacion
¿Cuál es el nombre de su ciudad o localidad?
[Unknown]: Ciudad
¿Cuál es el nombre de su estado o provincia?
[Unknown]: Estado
¿Cuál es el código de país de dos letras de la unidad?
[Unknown]: ES
¿Es correcto CN=Server, OU=Unidad, O=Organizacion, L=Ciudad, ST=Estado, C=ES?
[no]: si

Introduzca la contraseña de clave para <serverKey>
(INTRO si es la misma contraseña que la del almacén de claves):
```

Una vez creado el almacén vamos a exportar el certificado para enviárselo al cliente en un fichero.

```
p\PruebaSSL>keytool -export -keystore serverKey.jks -alias serverKey -file ServerPublicKey.cer
```


Este comando generará un archivo “ServerPublicKey.cer” que habrá que mandar al cliente.

Ahora, tenemos que meterlo en el almacén de certificados del cliente con el comando Keytool.

```
C:\Users\joaqf\Desktop\PruebaSSL>keytool -import -alias serverKey -file ServerPublicKey.cer -keystore clientTrustedCerts.jks -keypass clientpass -storepass clientpass
```

Ahora te preguntará la aplicación si deseas confiar en el certificado o no:

```
1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
000: 08 1D B8 CE FB 95 BD 7C 59 5A D5 9A 9D 1D E1 D7 .....YZ.....
010: AC 26 AA CE .&..

Confiar en este certificado? [no]:
```

Le decimos que sí:

```
¿Confiar en este certificado? [no]: si
Se ha agregado el certificado al almacén de claves
```

Ahora deberemos tener estos ficheros generados:

```
10/04/2021 19:29 949 clientTrustedCerts.jks
10/04/2021 19:22 2.238 serverKey.jks
10/04/2021 19:25 883 ServerPublicKey.cer
```

Por último, deberemos repetir todo el proceso para añadir en el almacén de certificados de confianza del servidor del certificado del cliente.

****Para no tener que repetir el proceso vamos a usar el mismo almacén en el cliente y en el servidor, aunque esto en una implementación sería con almacenes separados.**

6.2 Sockets SSL en servidor

Primero hay que establecer donde está el almacén de certificados y el almacén de certificados de confianza con `System.setProperty`.

```
System.setProperty("javax.net.ssl.keyStore", "serverKey.jks");
System.setProperty("javax.net.ssl.keyStorePassword", "servpass");

System.setProperty("javax.net.ssl.trustStore", "clientTrustedCerts.jks");
System.setProperty("javax.net.ssl.trustStorePassword", "clientpass");
```

A continuación, con una factoría de Socket creamos el socket del servidor:

```
SSLServerSocketFactory sslFactory = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
SSLServerSocket srvSocket = (SSLServerSocket) sslFactory.createServerSocket(4043);
SSLSocket socketSsl=(SSLSocket) srvSocket.accept();

DataInputStream dis= new DataInputStream(socketSsl.getInputStream());

String mensaje= dis.readUTF();

System.out.println(mensaje);
```

6.3 Sockets SSL en el cliente

En el cliente, también habrá que establecer las propiedades indicando los almacenes de certificados, a continuación, hay que crear el socket del cliente.

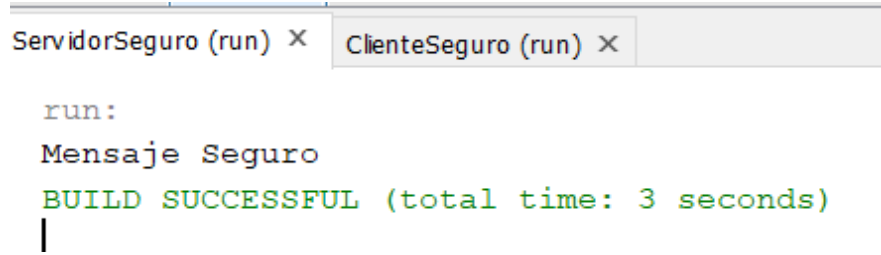
Una vez creados los sockets SSL se manejan de la misma manera que los sockets normales.

```
try {
    SSLSocketFactory sslFactory = (SSLSocketFactory) SSLSocketFactory.getDefault();
    SSLSocket cliSocket =(SSLSocket) sslFactory.createSocket("localhost", 4043);

    DataOutputStream dos= new DataOutputStream (cliSocket.getOutputStream());

    dos.writeUTF("Mensaje Seguro");
```

Vemos cómo llega el mensaje del cliente al servidor de manera segura.



The image shows a terminal window with two tabs: 'ServidorSeguro (run)' and 'ClienteSeguro (run)'. The 'ServidorSeguro (run)' tab is active, displaying the following output:

```
run:
Mensaje Seguro
BUILD SUCCESSFUL (total time: 3 seconds)
|
```

7. Bibliografía

- **Programación de Servicios y Procesos.**
Autor: M^a Jesús Ramos Martín.
Editorial Garceta, 2018.
- Programación de servicios y procesos.
Autor: Alberto Sánchez Campos, Jesús Montes Sánchez.
Editorial Ra-Ma, 2013.
- Programación concurrente.
Autor: José Tomás Palma Méndez, M^a Carmen Garrido y otros.
Editorial: Paraninfo.
- Piensa en Java.
Autor: Bruce Eckel.
Editorial: Prentice Hall.

Enlaces:

[http://chuwiki.chuidiang.org/index.php?title=Encriptacion con Java](http://chuwiki.chuidiang.org/index.php?title=Encriptacion_con_Java)

https://www.tutorialspoint.com/java_cryptography/

<http://www.jtech.ua.es/j2ee/2003-2004/modulos/sj/sesion02-apuntes.htm>