Contenido

[1. PRÁCTICAS DE PROGRAMACIÓN SEGURA 2](#_Toc156859324)

[1.1. Lista de buenas prácticas al escribir código en las aplicaciones 2](#_Toc156859325)

[1.2. Lista de malas prácticas al escribir código en las aplicaciones 4](#_Toc156859326)

[2. TÉCNICAS DE SEGURIDAD. VISIÓN GENERAL 5](#_Toc156859327)

[2.1. Criptografía 5](#_Toc156859328)

[2.2. Certificados digitales 6](#_Toc156859329)

[2.3. Control de acceso 7](#_Toc156859330)

[3. CRIPTOGRAFÍA CON JAVA 8](#_Toc156859331)

[3.1. Resúmenes de mensajes 9](#_Toc156859332)

[3.2. Firmas digitales 12](#_Toc156859333)

[3.2.1. Almacenar en un fichero las claves públicas y privadas 15](#_Toc156859334)

[3.2.2. Recuperar desde un fichero las claves pública y privada 16](#_Toc156859335)

[3.2.3. Firmar un documento con clave privada 18](#_Toc156859336)

[3.2.4. Verificar un documento con clave pública 19](#_Toc156859337)

[3.3. Herramientas Java para firmar ficheros 19](#_Toc156859338)

[3.4. Encriptar con Java 23](#_Toc156859339)

[3.4.1. Encriptar y desencriptar con clave privada 24](#_Toc156859340)

[3.4.2. Almacenar la clave secreta en un fichero 25](#_Toc156859341)

[3.4.3. Encriptar y desencriptar con clave pública 25](#_Toc156859342)

[3.4.4. Encriptar y desencriptar con flujos de datos 28](#_Toc156859343)

[4. COMUNICACIONES SEGURAS CON JAVA. JSSE 31](#_Toc156859344)

[5. CONTROL DE ACCESO CON JAVA. JAAS 36](#_Toc156859345)

[5.1. Autenticación 37](#_Toc156859346)

[5.2. Autorización 42](#_Toc156859347)

# PRÁCTICAS DE PROGRAMACIÓN SEGURA

Aunque se ha avanzado mucho en los mecanismos para escribir código libre de fallos, todavía se siguen creando programas incorrectos. Se estima que por cada 1000 líneas de código que se escriben, aparecen entre 5 y 10 errores. Sólo una pequeña parte de esos errores de programación se pueden considerar vulnerabilidades de seguridad.

Pero, ¿tan difícil es escribir código seguro? A continuación se incluye una lista de buenas prácticas y otra de malas prácticas a la hora de escribir código en las aplicaciones.

# Lista de buenas prácticas al escribir código en las aplicaciones

Una breve lista de cuestiones a tener siempre presentes cuando se desarrolla software es la siguiente:

**Informarse:**

* Una forma de evitar fallos es estudiar y comprender los errores que otros hayan cometido a la hora de desarrollar software
* Internet es el hogar de una gran variedad de foros públicos donde se debaten a veces problemas de vulnerabilidad de software
* Leer libros y artículos sobre prácticas de codificación segura, así como análisis de los defectos del software
* Explorar el software de código abierto, ya que hay muchos ejemplos de cómo llevar a cabo diversas acciones de programación

**Precaución en el manejo de datos.** La mayoría de los programas aceptan la entrada de algún tipo de dato. Los datos pueden ser adquiridos desde múltiples fuentes de entrada. El programador debería verificar cada pieza de entrada de datos al programa. Algunas de las prácticas a seguir en este apartado son las siguientes:

* Limpiar los datos: es el proceso de examen de los datos de entrada. Los atacantes a menudo intentan introducir contenido que está más allá de lo que el programador prevé para la entrada de datos del programa. Por ejemplo: la alteración del juego de caracteres, uso de caracteres no permitidos y desbordamiento del búfer de datos.
* Realizar la comprobación de límites: un problema típico es el desbordamiento del búfer. Es preciso asegurarse de verificar que los datos proporcionados al programa puedan caber en el espacio que se asigna para ello. En los arrays hay que revisar los índices para garantizar que permanecen dentro de sus límites.
* Revisar los ficheros de configuración: es necesario validarlos, como si se tratase de una entrada de datos por teclado, ya que pueden ser manipulados por un atacante.
* Comprobar los parámetros de línea de comandos.
* No fiarse de las URLs web: muchos diseñadores de aplicaciones web utilizan URLs para insertar variables y sus valores. El usuario puede alterar la URL directamente dentro de su navegador por variables de ajuste y/o de sus valores a cualquier configuración que elija.
* Cuidado con los contenidos web: algunas aplicaciones web insertan variables en campos HTML ocultos. Tales campos también pueden ser modificados por el usuario en una sesión de navegación.
* Comprobar las cookies web. Los valores pueden ser modificados por el usuario final y no se debe confiar en ellas.
* Comprobar las variables de entorno: un uso común de las variables de entorno es pasar configuración de preferencias a los programas. Los atacantes pueden proporcionar variables de entorno no previstas por el programador.
* Establecer valores iniciales válidos para los datos: es un buen hábito inicializar correctamente las variables.
* Comprender las referencias de nombre de fichero (rutas de acceso de ficheros y directorios) y utilizarlas correctamente dentro de los programas.
* Proteger la confidencialidad e integridad de información considerada como confidencial: contraseñas, números de cuenta de tarjetas de crédito, etc.

**Reutilizar código bueno siempre que sea posible.** Se refiere a la reutilización de software que ha sido completamente revisado y probado, y ha resistido las pruebas del tiempo y de los usuarios.

**Revisar los programas.** Siempre es aconsejable seguir una práctica de revisión de los fallos de seguridad en el código fuente. Si un programa es confiado a varias personas, todas deben participar en su revisión. Algunas prácticas comúnmente utilizadas son:

* Realizar una revisión por pares (dos o más revisores). Para entornos de desarrollo relativamente informales, un proceso de revisión de código por partes puede ser suficiente.
* Realizar una validación y verificación independiente. Algunos proyectos de programación necesitan una revisión más formal que implica revisar el código fuente de un programa, línea por línea, para garantizar que se ajusta a su diseño, así como a otros criterios (los de seguridad, por ejemplo).
* Identificar y utilizar las herramientas de seguridad disponibles. Hay herramientas de software disponibles que ayudan en la revisión de fallos en el código fuente. Son útiles para la captura de errores comunes, aunque no son tan útiles para detectar cualquier otro error. Algunas de estas herramientas son:
  + **FindBugs:** detección de errores de código Java (incluyendo algunos fallos de seguridad). Tiene un plugin para el IDE Eclipse. Ver <http://findbugs.sourceforge.net>
  + **PMD:** herramienta de auditoría y verificación de código estático que permite detectar errores potenciales en las aplicaciones, en base a un conjunto de reglas parametrizables. Tiene un plugin para el IDE Eclipse. Ver <https://pmd.github.io/>
  + **Visual Code Grepper:** herramienta de revisión automática de la seguridad del código disponible para los lenguajes de programaciónC++, C#, VB, PHP, Java y PL/SQL. Pensada para acelerar el proceso de revision del código fuente.Ver <https://sourceforge.net/projects/visualcodegrepp/>

**Utilizar listas de control de seguridad.** Estas listas pueden ser muy útiles para asegurarse de que se han cubierto todas las fases durante su ejecución. Un posible ejemplo sería el siguiente:

* La aplicación requiere una contraseña para que los usuarios puedan acceder.
* Todos los inicios de sesión de usuario son únicos.
* La aplicación utiliza el sistema de control de acceso basado en roles.
* Las contraseñas no se transmiten a través de la red en texto plano.
* El cifrado se utiliza para proteger los datos que se transfieren entre servidores y clientes.

**Mantener el código en buen estado**. El mantenimiento del código puede ser de vital importancia para la seguridad del software en el transcurso de su vida útil. Seguiremos las siguientes prácticas de mantenimiento del código:

* Utilizar normas. Se pueden tener normas con respecto a cosas como la documentación en línea del código fuente, la selección de los nombres de las variables, etc. Esto permite hacer más fácil la vida a los desarrolladores de software que luego mantendrán el código. El código modular, que está bien documentado y que es fácil de seguir, se mantiene mejor.
* Retirar código obsoleto.
* Analizar todos los cambios en el código.

# Lista de malas prácticas al escribir código en las aplicaciones

A continuación se incluye una lista no exhaustiva de malas prácticas a la hora de escribir código en las aplicaciones:

* Escribir código que utilice nombres de fichero relativos: la referencia a un nombre de fichero debe ser completa.
* Referirse dos veces en el mismo programa a un fichero por su nombre. Se recomienda abrir el fichero una vez por su nombre y utilizar un identificador a partir de entonces.
* Invocar programas no confiables dentro de los programas.
* Asumir que los usuarios no son maliciosos.
* Dar por sentado el éxito. Cada vez que se realiza una llamada al sistema, comprobar el valor de retorno por si la llamada fallase.
* Invocar dentro del programa a un shell o una línea de comandos.
* Utilizar áreas de almacenamiento con permisos de escritura. Si es absolutamente necesario, hay que suponer que la información pueda ser manipulada, alterada o destruida por cualquier persona o proceso que así lo desee.
* Guardar datos confidenciales en una base de datos sin protección de contraseña.
* Hacer eco de las contraseñas o mostrarlas en la pantalla del usuario.
* Enviar contraseñas vía e-mail.
* Distribuir mediante programación, información confidencial a través de correo electrónico.
* Guardar las contraseñas sin cifrar.
* Transmitir entre los sistemas contraseñas sin encriptar.
* Tomar decisiones de acceso basadas en variables de entorno o parámetros de línea de comandos que se pasan en tiempo de ejecución.
* Evitar en la medida de lo posible, el uso de software o los servicios de terceros para operaciones críticas.

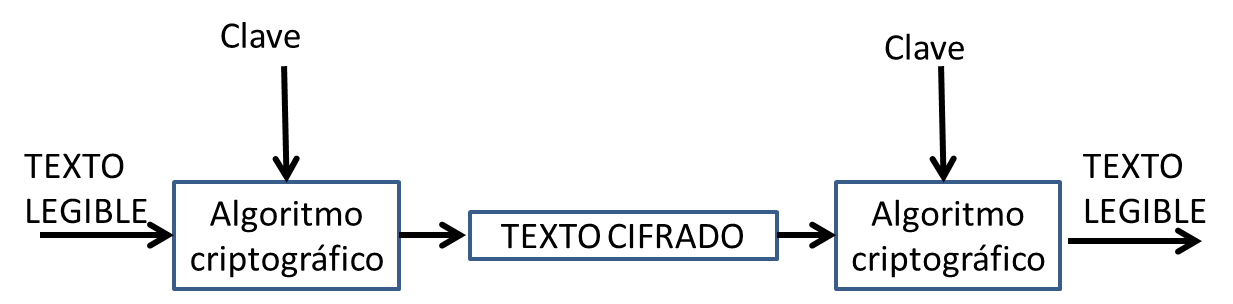
# TÉCNICAS DE SEGURIDAD. VISIÓN GENERAL

A continuación se presentan algunas de las técnicas y mecanismos más importantes para asegurar sistemas y aplicaciones: criptografía, certificados digitales y control de acceso.

# Criptografía

El término criptografía es un derivado de la palabra griega *kryptos*, que significa “oculto”. El objetivo de la criptografía es ocultar el significado de un mensaje mediante el cifrado o codificación del mensaje.

El proceso general de cifrado y descifrado de mensajes se muestra en la siguiente figura:



*Figura 2.1 – Proceso general de cifrado y descifrado de mensajes*

* Si a un texto legible se le aplica un algoritmo de cifrado, que en general depende de una clave, esto arroja como resultado un texto cifrado que es el que se envía o guarda. A este proceso se le llama **cifrado o encriptación**.
* Si a ese texto cifrado se le aplica el mismo algoritmo, dependiente de la mima clave o de otra clave (esto depende del algoritmo), se obtiene el texto legible original. A este segundo proceso se le llama **descifrado o desencriptación**.

Existen tres **clases de algoritmos criptográficos**:

* **Funciones de una sola vía (o funciones hash):** convierten un mensaje de cualquier tamaño en un mensaje de longitud constante. Permiten mantener la integridad de los datos, tanto en el almacenamiento como en el tráfico de redes. También se usan como parte de los mecanismos de firma digital. Reciben su nombre debido a su naturaleza matemática. Dado un mensaje *x*, es muy fácil calcular el resultado de *f(x)*. A este *f(x)* se le denomina el hash, resumen de x o message digest de x. La clave está en que es prácticamente imposible calcular x a partir de las f(x). Este tipo de funciones tienen un amplio abanico de usos en la seguridad informática. Casi cualquier protocolo usa este tipo de funciones así como la autenticación por firmas digitales. Los dos algoritmos de una sola vía más utilizados son el MD5 o SHA-1.
* **Algoritmos de clave secreta o de criptografía simétrica.** El emisor y el receptor comparten el conocimiento de una clave que no debe ser revelada a nadie más. La clave se usa tanto para cifrar como para descifrar el mensaje. La comunicación de las claves entre el emisor y el receptor (diciéndola en alto, mandándola por correo electrónico o postal, haciendo una llamada telefónica, …) constituye el punto débil de este tipo de criptografía. El algoritmo de cifrado simétrico más popular es el **DES**, que utiliza claves de 56 bits y un cifrado de bloques de 64 bits. Una variante de éste es el Triple DES o 3DES, cuya clave es de 128 bits (112 de clave y 16 de paridad). Otro algoritmo es el **AES** con un tamaño de clave variable siendo el estándar el de 256 bits.
* **Algoritmos de clave pública o de criptografía asimétrica.** El emisor de un mensaje emplea una clave pública (difundida previamente por el receptor) para encriptar el mensaje. El receptor emplea la correspondiente clave privada (no debe ser revelada nunca) para desencriptar el mensaje y sólo él puede desencriptar el mensaje gracias a su clave privada.

El algoritmo asimétrico más popular es el **RSA,** cuyo usoes prácticamente universal como método de autenticación y firma digital. Además, es componente de protocolos y sistemas como IPSec, SSL, PGP, etc.

Una **firma digital** está compuesta por una serie de datos asociados a un mensaje, estos datos permiten asegurar la identidad del firmante (emisor del mensaje) y la integridad del mensaje. El método de firma digital más extendido es el RSA. Su funcionamiento es el siguiente:

* El emisor:
* Genera un hash del mensaje a enviar mediante una función acordada.
* Este hash es cifrado con su clave privada. El resultado es lo que se conoce como firma digital que se envía adjunta al mensaje.
* El emisor envía el mensaje y su firma digital, es decir, el mensaje firmado.
* El receptor:
* Separa el mensaje de la firma digital.
* Genera el resumen del mensaje recibido usando la misma función que el emisor.
* Descifra la firma digital con la clave pública del emisor, obteniendo el hash original.
* Compara los dos hash y, si son iguales, el mensaje recibido es correcto y con garantías de que nos lo ha enviado el emisor.

**En resumen**, la criptografía juega 3 papeles principales en la implementación de sistemas seguros:

* Se usa para mantener el secreto y la integridad de la información, donde quiera que pueda estar expuesta a ataques.
* Se utiliza como base para los mecanismos de autentación de la comunicación entre pares de principales (un principal puede ser un usuario o un proceso).
* Se usa para implementar el mecanismo de la firma digital.

# Certificados digitales

Un **certificado digital** o certificado electrónico es un fichero informático generado por una entidad de servicios de certificación que asocia unos datos de identidad a una persona física, organismo o empresa confirmando de esta manera su identidad digital en Internet. El certificado digital es válido principalmente para autenticar a un usuario o sitio web en internet, por lo que es necesaria la colaboración de un tercero que sea de confianza para cualquiera de las partes que participe en la comunicación.

Estos terceros que son de confianza se llaman **Autoridades de Certificación (AC).** Son entidades que se encargan de emitir y gestionar tales certificados y que tienen una propiedad muy importante: que se puede confiar en ellas. La forma en la que la AC hace válido el certificado es firmándolo digitalmente. Al aplicar el algoritmo de firma digital al documento se obtiene un texto, una secuencia de datos que permiten asegurar que el titular de ese certificado ha firmado electrónicamente el texto y que éste no ha sido modificado.

Un certificado digital tiene un formato estándar universalmente aceptado que se llama **X.509**, cuya estructura principal está formada por los siguientes campos:

* Versión: indica la versión del formato del certificado, normalmente X.509v3.
* Número de serie: identificador numérico único dentro del dominio de la AC.
* Algoritmo de firma y parámetros, que identifican el algoritmo asimétrico y la función de una sola vía que se usa para firmar el certificado.
* Emisor del certificado: el nombre X.500 de la AC.
* Fechas de inicio y final de validez, que determinan el periodo de validez del certificado.
* Nombre del propietario de la clave pública que se está firmando.
* Identificador del algoritmo que se está utilizando, la clave pública del usuario y otros parámetros si son necesarios.
* La firma digital de la AC, es decir, el resultado de cifrar mediante el algoritmo asimétrico y la clave privada de la AC, el hash obtenido del documento X.509.

Para ver el almacén de certificados del navegador **Mozilla Firefox** pulsamos en la opción de menú Herramientas > Opciones > Avanzado, pestaña de cifrado y pulsamos en el botón Ver Certificados. Desde el navegador **Internet Explorer** pulsamos en la opción de menú Herramientas > Opciones de internet > pestaña Contenido y pulsamos en el botón Certificados.

Podemos ver el certificado de la página que estamos visitando pinchando sobre el candado.

Las principales **aplicaciones de los certificados digitales** son:

* Autenticar la identidad del usuario de forma electrónica ante terceros.
* Trámites electrónicos ante organismos públicos: la Agencia Tributaria, la Seguridad Social, las Cámaras, …
* Trabajar con facturas electrónicas.
* Firmar digitalmente correos electrónicos y todo tipo de documentos.
* Cifrar datos para que solo el destinatario del documento pueda acceder a su contenido.

Algunas AC españolas son: FNMT (Fábrica Nacional de Moneda y Timbre), Izenpe y la Agencia Catalana de Certificació (CATCert).

# Control de acceso

En general, cualquier empresa u organización tiene diversos tipos de recursos de carácter privado y secreto que necesita asegurar, de manera que solo las personas indicadas puedan acceder a ellos para realizar las tareas requeridas. Esos recursos pueden ser físicos (por ejemplo un equipo informático muy caro), informativos (datos confidenciales) o de personal (los empleados).

Se hace necesario realizar un control de acceso a los recursos, pero este control de acceso es algo más que simplemente requerir nombres de usuario y contraseñas. Hay tres componentes importantes de control de acceso:

* **Identificación:** proceso mediante el cual el sujeto suministra información diciendo quién es.
* **Autenticación:** es cualquier proceso por el cual se verifica que alguien es quien dice ser. Esto implica generalmente un nombre de usuario y una contraseña, pero puede incluir cualquier otro método para demostrar la identidad, como una tarjeta inteligente, exploración de la retina, reconocimiento de voz o huellas dactilares.
* **Autorización**: es el proceso de determinar si el sujeto, una vez autenticado, tiene acceso al recurso. La autorización es equivalente a la comprobación de la lista de invitados de una fiesta.

El sistema de control de acceso debe permitir el acceso al usuario correctamente autenticado y debe impedir el acceso a los demás. Debería también guardar un buen registro de auditoría de todas las entradas e intentos fallidos.

Las medidas de identificación y autenticación se centran en una de estas tres formas:

* Algo que se sabe, algo que se conoce, típicamente las **contraseñas**, son la más extendida.
* Algo que se es, medidas que utilizan la **biometría** (identificación por medio de la voz, la retina, la huella dactilar, geometría de mano, etc.)
* Algo que se tiene, las **tarjetas**.

# CRIPTOGRAFÍA CON JAVA

El API **JCA** (Java Cryptography Architecture) incluye la extensión criptográfica de Java **JCE** (Java Cryptography Extension) y está incluído dentro del paquete JDK. Tiene los siguientes componentes de software:

* El marco que define y apoya los servicios criptográficos para que los prestadores faciliten implementaciones. Este marco incluye paquetes como **java.security, javax.crypto, javax.crypto.spec y javax.crypto.interfaces.**
* Los proveedores reales (tales como Sun, SunRsaSign y SunJCE): contienen las implementaciones criptográficas reales. El proveedor es el encargado de proporcionar la implementación de uno o varios algoritmos al programador. Los proveedores de seguridad se definen en el fichero *java.security* localizado en la carpeta *java.home\lib\security*. Forman una lista de entradas con un número que indica el orden de búsqueda cuando en los programas no se especifica un proveedor:

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

security.provider.8=org.jcp.xml.dsig.internal.dom.XMLDSigRI

security.provider.9=sun.security.smartcardio.SunPCSC

security.provider.10=sun.security.mscapi.SunMSCAPI

**JCA** define el concepto de proveedor mediante la clase **Provider** del paquete **java.security**. Se trata de una clase abstracta que debe ser redefinida por clases específicas de proveedor. Tiene métodos para acceder al nombre del proveedor, al número de versión y a otras informaciones sobre las implementaciones de los algoritmos, para la generación, conversión y gestión de claves y la generación de firmas y resúmenes.

# Resúmenes de mensajes

Un *message digest* o resumen de mensajes (también conocido como función hash) es una marca digital de un bloque de datos. Existe un gran número de algoritmos diseñados para procesar estos resúmenes, siendo los dos más conocidos SHA-1 y MD5. Los resúmenes de mensaje son funciones hash seguras de una sola vía (obtienen una salida segura de longitud fija, partiendo de una entrada de longitud variable).

La clase **MessageDigest** permite a las aplicaciones implementar algoritmos de resumen de mensaje, como MD5, SHA-1 o SHA-256.

**Ejemplo 1 - Generar el resumen de un texto plano. Con el método *MessageDigest.getInstance(“SHA”)* se obtiene una instancia del algoritmo SHA con el que se hará el resumen. El texto plano se pasa a un array de bytes y el array se pasa como argumento al método *update()*; finalmente, aplicando el método *digest()* se obtiene el resumen del mensaje. Después se muestra en pantalla el número de bytes generados en el mensaje, el algoritmo utilizado, el resumen generado y convertido a hexadecimal mediante el método *Hexadecimal()* y por último la información del proveedor**

**import** java.security.MessageDigest;

**import** java.security.NoSuchAlgorithmException;

**import** java.security.Provider;

**public** **class** Ejemplo1 {

**public** **static** **void** main(String [] args) {

MessageDigest md;

**try** {

md= MessageDigest.*getInstance*("SHA");

String texto = "Esto es un texto plano";

**byte** dataBytes[] = texto.getBytes(); // Texto a bytes

md.update(dataBytes); // Se introduce texto en bytes a resumir

**byte** resumen[]= md.digest(); // Se calcula el resumen

System.***out***.println("Mensaje original:" + texto);

System.***out***.println("Número de bytes:" + md.getDigestLength());

System.***out***.println("Algoritmo: " + md.getAlgorithm());

System.***out***.println("Mensaje Resumen: " + **new** String(resumen));

System.***out***.println("Mensaje en Hexadecimal: " + *Hexadecimal*(resumen));

Provider proveedor = md.getProvider();

System.***out***.println("Proveedor: "+ proveedor.toString());

} **catch** (NoSuchAlgorithmException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

} // main

// Convierte un array de bytes a hexadecimal

**static** String Hexadecimal (**byte**[] resumen) {

String hex="";

**for** (**int** i =0; i< resumen.length; i++) {

String h = Integer.*toHexString*(resumen[i] & 0xFF);

**if** (h.length() == 1) hex +="0";

hex += h;

}

**return** hex.toLowerCase();

}

}

**Resultado esperado:**

Mensaje original:Esto es un texto plano

Número de bytes:20

Algoritmo: SHA

Mensaje Resumen: o^š[¼…×T·›§Œ?­upªµ\_Ö

Mensaje en Hexadecimal: 6f5e9a5bbc85d754b79ba78c3fad7570aab517d6

Proveedor: SUN version 1.8

**Ejemplo 2 - Guardar un objeto String en un fichero de nombre DATOS.DAT, pero queriendo estar seguros de que a la hora de leer el String el fichero no esté dañado o no haya sido manipulado y los datos sean los correctos. Además de guardar el String en el fichero, se guardará el resumen del mismo. Usar la clase *ObjectOutputStream* para escribir en el fichero, tanto el String como su resumen**

**import** java.io.\*;

**import** java.security.\*;

**public** **class** Ejemplo2 {

**public** **static** **void** main (String args[]) {

FileOutputStream fileout;

**try** {

fileout = **new** FileOutputStream ("DATOS.DAT");

ObjectOutputStream dataOS = **new** ObjectOutputStream(fileout);

MessageDigest md = MessageDigest.*getInstance*("SHA");

String datos = "En un lugar de la Marcha, "

+ "de cuyo nombre no quiero acordarme, no ha mucho tiempo ...";

**byte** dataBytes[] = datos.getBytes();

md.update(dataBytes); // Texto a resumir

**byte** resumen[] = md.digest(); // Se calcula el resumen

dataOS.writeObject(datos); // Se escriben los datos

dataOS.writeObject(resumen); // Se escribe el resumen

dataOS.close();

fileout.close();

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (NoSuchAlgorithmException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**Ejemplo 3 - Recuperar los datos del fichero DATOS.DAT, leyendo primero el String y luego el resumen. Calcular de nuevo el resumen del String leído, comparándolo con el resumen leído del fichero. Usar el método *isEgual() p*ara comparar los dos resúmenes. Si los resúmenes son idénticos significa que el contenido del fichero no ha sido modificado**

**import** java.io.FileInputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**import** java.security.MessageDigest;

**import** java.security.NoSuchAlgorithmException;

**public** **class** Ejemplo3 {

**public** **static** **void** main (String args[]) {

**try** {

FileInputStream fileout = **new** FileInputStream("DATOS.DAT");

ObjectInputStream dataOS = **new** ObjectInputStream(fileout);

Object o = dataOS.readObject();

// Primera lectura para obtener el String

String datos = (String) o;

System.***out***.println("Datos: "+datos);

// Segunda lectura para obtener el resumen

o = dataOS.readObject();

**byte** resumenOriginal [] = (**byte**[])o;

MessageDigest md = MessageDigest.*getInstance*("SHA");

// Calcular el resumen del String leido del fichero

md.update(datos.getBytes()); // Texto a resumir

**byte** resumenActual[] = md.digest(); // Calcular el resumen

**if** (MessageDigest.*isEqual*(resumenActual, resumenOriginal))

System.***out***.println("DATOS VÁLIDOS");

**else**

System.***out***.println("DATOS NO VÁLIDOS");

dataOS.close();

fileout.close();

} **catch** (IOException | ClassNotFoundException | NoSuchAlgorithmException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**Resultado esperado tras ejecutar los Ejemplos 2 y 3:**

Datos: En un lugar de la Marcha, de cuyo nombre no quiero acordarme, no ha mucho tiempo ...

DATOS VÁLIDOS

# Firmas digitales

El resumen de un mensaje no proporciona un alto nivel de seguridad. En el ejemplo anterior se puede afirmar que el fichero no es correcto si el texto que se lee no produce la misma salida que el resumen guardado. Pero alguien puede cambiar el texto y el resumen, y no se puede estar seguro de que el texto sea el que debería ser.

En este apartado se verá cómo las firmas digitales pueden autenticar un mensaje y asegurar que el mensaje no ha sido alterado y procede del emisor correcto. Para crear una firma digital se necesita una **clave privada** y la correspondiente **clave pública**, con el fin de verificar la autenticidad de la firma.

En algunos casos, el par de claves (clave pública y la correspondiente clave privada) están disponibles en ficheros. En ese caso, el programa puede importar y utilizar la clave privada para firmar. En otros casos, el programa necesita generar el par de claves. La clase **KeyPairGenerator** permite generar el par de claves.

La clase **KeyPair** es una clase soporte para generar las claves pública y privada. Dispone de los siguientes métodos:

|  |  |
| --- | --- |
| **Método** | **Función** |
| PrivateKey getPrivate() | Devuelve una referencia a la clave privada del par de claves. |
| PublicKey getPublic() | Devuelve una referencia a la clave pública del par de claves. |

El **primer paso** para generar el par de claves es obtener un objeto generador. Por ejemplo, para obtener un generador de claves DSA se escribe lo siguiente:

KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");

El **segundo paso** es inicializar el generador del par de claves. Se necesita el método *initialize()* con dos argumentos:

* El tamaño de clave: para un generador de claves DSA puede ser un valor entre 512 y 1024 y debe ser múltiplo de 64. Si se pasa un valor incorrecto, se produce la excepción *InvalidParameterException*.
* El generador de números aleatorios: se puede utilizar una instancia de la clase **SecureRandom**.

El siguiente ejemplo pide un número aleatorio que usa el algoritmo de generación de números pseudoaleatorios SHA1PRNG. Luego pasa el número de SecureRandom al método de inicialización del generador de claves:

SecureRandom numero = SecureRandom.getInstance("SHA1PRNG");

keyGen.initialize(1024, numero);

El **tercer y último paso** es generar el par de claves pública y privada y almacenarlas en los objetos PrivateKey y PublicKey:

KeyPair par = keyGen.generateKeyPair();

PrivateKey clavepriv = par.getPrivate();

PublicKey clavepub = par.getPublic();

Una vez creadas las claves se pueden firmar los datos. Para firmar los datos se usa una instancia de la clase **Signature**. Un objeto de esta clase se usa para generar y verificar firmas digitales. Dispone de un constructor protegido y se accede a él mediante el método *getInstance(String algoritmo)*.

Al especificar el nombre del algoritmo de firma, también hay que incluir el nombre del algoritmo de resumen de mensajes utilizado por el algoritmo de firma. Por ejemplo:

* **SHA1withDSA:** es una forma de especificar el algoritmo de firma DSA, usando el algoritmo de resumen SHA-1.
* **MD5widthRSA:** significa que utiliza el algoritmo de resumen MD5 con algoritmo de firma RSA.

Existen tres **fases en el uso de los objetos Signature**, ya sea para firmar o para verificar datos:

* **Inicialización:** initVerify() con clave pública, o initSign() con clave privada.
* **Actualización:** update().
* **Firma** (sign()) o **verificación** (verify()).

**Ejemplo 4 – Firmar un mensaje (“Este mensaje va a ser firmado”) con una clave privada DSA y verificar el mensaje firmado con la correspondiente clave pública**

**import** java.security.InvalidKeyException;

**import** java.security.KeyPair;

**import** java.security.KeyPairGenerator;

**import** java.security.NoSuchAlgorithmException;

**import** java.security.PrivateKey;

**import** java.security.PublicKey;

**import** java.security.SecureRandom;

**import** java.security.Signature;

**import** java.security.SignatureException;

**public** **class** Ejemplo4 {

**public** **static** **void** main (String[] args) {

**try** {

KeyPairGenerator keyGen = KeyPairGenerator.*getInstance*("DSA");

// SE INICIALIZA EL GENERADOR DE CLAVES

SecureRandom numero = SecureRandom.*getInstance*("SHA1PRNG");

keyGen.initialize(1024, numero);

// SE CREA EL PAR DE CLAVES PRIVADA Y PÚBLICA

KeyPair par = keyGen.generateKeyPair();

PrivateKey clavepriv = par.getPrivate();

PublicKey clavepub = par.getPublic();

/// SE FIRMA EL MENSAJE CON LA CLAVE PRIVADA

/// AL OBJETO Signature SE LE SUMINISTRAN LOS DATOS A FIRMAR

Signature dsa = Signature.*getInstance*("SHA1withDSA");

dsa.initSign(clavepriv);

String mensaje = "Este mensaje va a ser firmado";

dsa.update(mensaje.getBytes());

**byte**[] firma = dsa.sign(); // Mensaje firmado

// EL RECEPTOR DEL MENSAJE

// VERIFICA CON LA CLÁVE PÚBLICA EL MENSAJE FIRMADO

// AL OBJETO Signature SE LE SUMINISTRAN LOS DATOS A VERIFICAR

Signature verificadsa = Signature.*getInstance*("SHA1withDSA");

verificadsa.initVerify(clavepub);

verificadsa.update(mensaje.getBytes());

**boolean** check = verificadsa.verify(firma);

**if** (check)

System.***out***.println("FIRMA VERIFICADA CON CLAVE PÚBLICA");

**else**

System.***out***.println("FIRMA NO VERIFICADA");

} **catch** (NoSuchAlgorithmException | InvalidKeyException | SignatureException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**Resultado:**

FIRMA VERIFICADA CON CLAVE PÚBLICA

# Almacenar en un fichero las claves públicas y privadas

Para almacenar la **clave privada** en discoes necesario codificarla en formato PKCS8 usando la clase **PKCS8EncodedKeySpec**. En la siguiente tabla se muestran el constructor y sus métodos:

|  |  |
| --- | --- |
| **Método** | **Función** |
| PKCS8EncodedKeySpec (byte[] encodedKey) | Crea un nuevo objeto PKCS8EncodedKeySpec con la clave codificada |
| byte[] getEncoded() | Devuelve los bytes codificados de la clave de acuerdo con el estádar PKCS #8 |
| String getFormat() | Devuelve el nombre del formato de codificación asociado con esta especificación de clave. |

Para almacenar la **clave pública** en disco es necesario codificarla en formato X.509 usando la clase **X509EncodedKeySpec**. En la siguiente tabla se muestra el constructor y sus métodos:

|  |  |
| --- | --- |
| **MÉTODOS** | **FUNCIÓN** |
| X509EncodedKeySpec (byte[] encodedKey) | Crea un nuevo objeto X509EncodedKeySpec con la clave codificada |
| byte[] getEncoded() | Devuelve los bytes codificados de la clave de acuerdo con el estádar X.509 |
| String getFormat() | Devuelve el nombre del formato de codificación asociado con esta especificación de clave |

**Ejemplo 5 – Guardar la clave privada en un fichero en disco llamado *Clave.privada* y la clave pública en un fichero en disco llamado *Clave.publica***

**import** java.io.\*;

**import** java.security.\*;

**import** java.security.spec.\*;

**public** **class** Ejemplo5 {

**public** **static** **void** main (String[] args) {

**try** {

KeyPairGenerator keyGen = KeyPairGenerator.*getInstance*("DSA");

// SE INICIALIZA EL GENERADOR DE CLAVES

SecureRandom numero = SecureRandom.*getInstance*("SHA1PRNG");

keyGen.initialize(1024, numero);

// SE CREA EL PAR DE CLAVES PRIVADA Y PÚBLICA

KeyPair par = keyGen.generateKeyPair();

PrivateKey clavepriv = par.getPrivate();

PublicKey clavepub = par.getPublic();

// SE ALMACENA EN DISCO LA CLAVE PRIVADA

PKCS8EncodedKeySpec pk8Spec = **new** PKCS8EncodedKeySpec(clavepriv.getEncoded());

FileOutputStream outpriv= **new** FileOutputStream ("Clave.privada");

outpriv.write (pk8Spec.getEncoded());

outpriv.close();

// SE ALMACENA EN DISCO LA CLAVE PÚBLICA

X509EncodedKeySpec pkX509 = **new** X509EncodedKeySpec(clavepub.getEncoded());

FileOutputStream outpub= **new** FileOutputStream ("Clave.publica");

outpub.write (pkX509.getEncoded());

outpub.close();

} **catch** (NoSuchAlgorithmException | IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

# Recuperar desde un fichero las claves pública y privada

Para recuperar las claves pública y privada desde ficheros se usa la clase **KeyFactory**, la cualproporciona métodos para convertir claves de formato criptográfico (PKCS8, X.509) a especificaciones de claves y viceversa. En la siguiente tabla se muestra el constructor y sus métodos:

|  |  |
| --- | --- |
| **Método** | **Función** |
| static KeyFactory getInstance(String algoritmo)  static KeyFactory getInstance(String algoritmo, String provider) | Devuelve un objeto KeyFactory capaz de importar y exportar las claves generadas con el algoritmo dado. Puede lanzar la excepción NoSuchAlgorithmException.  El segundo constructor con un segundo parámetro proveedor puede lanzar un NoSuchProviderException. |
| PrivateKey generatePrivate (KeySpec KeySpec) | Genera un objeto de clave privada a partir de la expecificación de la clave suministrada. |
| PrivateKey generatePublic (KeySpec KeySpec) | Genera un objeto de clave pública a partir de la expecificación de la clave suministrada. |

**Ejemplo 6 – Generar las claves privada y pública, guardarlas respectivamente en los ficheros de nombre *Clave.privada* y *Clave.publica,* recuperar las claves desde los ficheros y mostrarlas por pantalla**

**import** java.io.\*;

**import** java.security.\*;

**import** java.security.spec.\*;

**public** **class** Ejemplo6 {

**public** **static** **void** main (String[] args) {

**try** {

KeyPairGenerator keyGen = KeyPairGenerator.*getInstance*("DSA");

// SE INICIALIZA EL GENERADOR DE CLAVES

SecureRandom numero = SecureRandom.*getInstance*("SHA1PRNG");

keyGen.initialize(1024, numero);

// SE CREA EL PAR DE CLAVES PRIVADA Y PÚBLICA

KeyPair par = keyGen.generateKeyPair();

PrivateKey clavepriv = par.getPrivate();

PublicKey clavepub = par.getPublic();

// SE ALMACENA EN DISCO LA CLAVE PRIVADA

PKCS8EncodedKeySpec pk8Spec = **new** PKCS8EncodedKeySpec(clavepriv.getEncoded());

FileOutputStream outpriv= **new** FileOutputStream ("Clave.privada");

outpriv.write (pk8Spec.getEncoded());

outpriv.close();

// SE ALMACENA EN DISCO LA CLAVE PÚBLICA

X509EncodedKeySpec pkX509 = **new** X509EncodedKeySpec(clavepub.getEncoded());

FileOutputStream outpub= **new** FileOutputStream ("Clave.publica");

outpub.write (pkX509.getEncoded());

outpub.close();

// SE LEE DEL DISCO LA CLAVE PRIVADA

FileInputStream in = **new** FileInputStream("Clave.privada");

**byte**[] bufferPriv = **new** **byte**[in.available()];

in.read(bufferPriv); // Lectura de bytes

in.close();

KeyFactory keyDSA= KeyFactory.*getInstance*("DSA");

// SE VISUALIZAN LOS DATOS RECUPERADOS DE LA CLAVE PRIVADA

PKCS8EncodedKeySpec clavePrivadaSpec = **new** PKCS8EncodedKeySpec(bufferPriv);

PrivateKey clavePrivada=keyDSA.generatePrivate(clavePrivadaSpec);

System.***out***.println("Clave privada:" + clavePrivada.toString());

// SE LEE DEL DISCO LA CLAVE PÚBLICA

FileInputStream inpub = **new** FileInputStream("Clave.publica");

**byte**[] bufferPub = **new** **byte**[inpub.available()];

inpub.read(bufferPub); // Lectura de bytes

inpub.close();

// SE VISUALIZAN LOS DATOS RECUPERADOS DE LA CLAVE PUBLICA

X509EncodedKeySpec clavePublicaSpec = **new** X509EncodedKeySpec(bufferPub);

PublicKey clavePublica=keyDSA.generatePublic(clavePublicaSpec);

System.***out***.println("Clave publica:" + clavePublica.toString());

} **catch** (NoSuchAlgorithmException | IOException | InvalidKeySpecException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**Resultado:**

Clave privada:sun.security.provider.DSAPrivateKey@fffe7ecb

Clave publica:Sun DSA Public Key

Parameters:DSA

p: fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400 c31e3f80 b6512669

455d4022 51fb593d 8d58fabf c5f5ba30 f6cb9b55 6cd7813b 801d346f f26660b7

6b9950a5 a49f9fe8 047b1022 c24fbba9 d7feb7c6 1bf83b57 e7c6a8a6 150f04fb

83f6d3c5 1ec30235 54135a16 9132f675 f3ae2b61 d72aeff2 2203199d d14801c7

q: 9760508f 15230bcc b292b982 a2eb840b f0581cf5

g: f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82 f9574c0b 3d078267

5159578e bad4594f e6710710 8180b449 167123e8 4c281613 b7cf0932 8cc8a6e1

3c167a8b 547c8d28 e0a3ae1e 2bb3a675 916ea37f 0bfa2135 62f1fb62 7a01243b

cca4f1be a8519089 a883dfe1 5ae59f06 928b665e 807b5525 64014c3b fecf492a

y:

3c6e8068 54bd6626 e149ae82 c831d28d 3ed07462 c3ab4e2f 41f21850 3e563036

92e72d9a 5290fe78 ca970d82 04d21578 a8858fb9 9b94473e 2c2fa127 d098a90a

112f20db 7b021cdb 2419e41f ab1b6a8b 7e6ee98a b15a4dc1 d7719a47 d081d908

fc54f629 ac212e09 bdcb2750 f4430adb edf4252f 559fa22d 73e8fec9 5b9f1978

# Firmar un documento con clave privada

**Ejemplo 7 – Firmar el fichero FICHERO.DAT a partir de la clave privada almacenada en el fichero *Clave.privada.* El resultado de la firma se almacenará en el fichero FICHERO.FIRMA**

**import** java.io.BufferedInputStream;

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.security.InvalidKeyException;

**import** java.security.KeyFactory;

**import** java.security.NoSuchAlgorithmException;

**import** java.security.PrivateKey;

**import** java.security.Signature;

**import** java.security.SignatureException;

**import** java.security.spec.InvalidKeySpecException;

**import** java.security.spec.PKCS8EncodedKeySpec;

**public** **class** Ejemplo7 {

**public** **static** **void** main (String[] args) {

**try** {

// LECTURA DEL FICHERO DE CLAVE PRIVADA

FileInputStream inpriv= **new** FileInputStream ("Clave.privada");

**byte**[] bufferPriv = **new** **byte**[inpriv.available()];

inpriv.read(bufferPriv); // lectura de bytes

inpriv.close();

// RECUPERACIÓN CLAVE PRIVADA DESDE DATOS CODIFICADOS EN FORMATO PKCS8

PKCS8EncodedKeySpec clavePrivadaSpec = **new** PKCS8EncodedKeySpec(bufferPriv);

KeyFactory keyDSA= KeyFactory.*getInstance*("DSA");

PrivateKey clavePrivada=keyDSA.generatePrivate(clavePrivadaSpec);

// INICIALIZACIÓN DE LA FIRMA CON CLAVE PRIVADA

Signature dsa = Signature.*getInstance*("SHA1withDSA");

dsa.initSign(clavePrivada);

// LECTURA DEL FICHERO A FIRMAR

// Se suministra al objeto signatura los datos a firmar

FileInputStream fichero = **new** FileInputStream("FICHERO.DAT");

BufferedInputStream bis = **new** BufferedInputStream(fichero);

**byte**[] buffer = **new** **byte**[bis.available()];

**int** len;

**while** ((len = bis.read(buffer)) >= 0)

dsa.update(buffer, 0, len);

bis.close();

// GENERACIÓN DE LA FIRMA DE LOS DATOS DEL FICHERO

**byte**[] firma = dsa.sign();

// GUARDADO DE LA FIRMA EN OTRO FICHERO

FileOutputStream fos = **new** FileOutputStream ("FICHERO.FIRMA");

fos.write(firma);

fos.close();

} **catch** (NoSuchAlgorithmException | IOException | InvalidKeySpecException | InvalidKeyException | SignatureException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

# Verificar un documento con clave pública

Para verificar la validez de un determinado documento previamente firmado de forma digital, es preciso disponer de la clave pública asociada a la clave privada con la que fue firmado el documento.

# Herramientas Java para firmar ficheros

A veces se necesita enviar electrónicamente un documento importante a un destinatario (por ejemplo, un contrato de trabajo), y que el destinatario del documento pueda comprobar que el documento proviene del emisor esperado y no ha sido alterado en el camino. Java dispone de la herramienta de línea de comandos llamada **keytool** para generar y manipular certificados. Mediante un **certificado** se garantiza que el documento no se ha modificado y que el emisor es quien dice ser.

A continuación se muestra cómo un emisor puede firmar un documento (un fichero) y enviárselo a un destinatario y cómo el destinatario puede verificar que ese documento realmente lo ha firmado el emisor y no un impostor. Para firmar un documento es necesario seguir los siguientes pasos:

1.- Crear un fichero JAR que contiene el documento a firmar utilizando la herramienta **jar**.

2.- Generar las claves publica y privada, utilizando el comando **keytool -genkey**.

3.- Firmar el fichero JAR, usando la herramienta **jarsigner** y la clave privada generada en el paso anterior.

4.- Exportar el certificado de clave pública para que el receptor del documento verifique la firma del emisor, utilizando el comando **keytool -export**.

5.- Suministrar al receptor tanto el fichero JAR firmado como el certificado.

**Paso 1**

Supongamos que se dispone de un fichero llamado *Contrato.pdf.* Usar la herramienta **jar** desde la línea de comandos para crear el fichero *Contrato.jar*:

**$ jar -cvf Contrato.jar Contrato.pdf**

manifest agregado

agregando: Contrato.pdf(entrada = 7804) (salida = 7096)(desinflado 9%)

**Paso 2**

Para generar el par de claves pública/privada, se usa la herramienta **keytool**. Se crea un almacén de claves (keystore) con el nombre *AlmacenClaves* y el alias con el que se hará referencia al par de claves creado será *FirmaContrato*:

**$ keytool -genkey -alias FirmaContrato -keystore AlmacenClaves**

Introduzca la contraseña del almacén de claves:

La contraseña del almacén de claves es demasiado corta, debe tener al menos 6 caracteres

Introduzca la contraseña del almacén de claves:

Volver a escribir la contraseña nueva:

¿Cuáles son su nombre y su apellido?

[Unknown]: Aitor

¿Cuál es el nombre de su unidad de organización?

[Unknown]: CiudadJardin

¿Cuál es el nombre de su organización?

[Unknown]: IES Ciudad Jardin

¿Cuál es el nombre de su ciudad o localidad?

[Unknown]: Vitoria-Gasteiz

¿Cuál es el nombre de su estado o provincia?

[Unknown]: Alava

¿Cuál es el código de país de dos letras de la unidad?

[Unknown]: ES

¿Es correcto CN=Aitor, OU=CiudadJardin, O=IES Ciudad Jardin, L=Vitoria-Gasteiz, ST=Alava, C=ES?

[no]: si

Introduzca la contraseña de clave para <FirmaContrato>

(INTRO si es la misma contraseña que la del almacén de claves):

Volver a escribir la contraseña nueva:

Para ver el contenido del almacen de claves, teclear el siguiente comando:

**$ keytool -list -keystore AlmacenClaves**

Introduzca la contraseña del almacén de claves:

Tipo de Almacén de Claves: JKS

Proveedor de Almacén de Claves: SUN

Su almacén de claves contiene 1 entrada

firmacontrato, 08-feb-2016, PrivateKeyEntry,

Huella Digital de Certificado (SHA1): C4:F9:E2:4E:89:AE:DB:42:7E:54:4A:96:08:04:81:15:BA:41:20:41

**Paso 3**

Para firmar el fichero JAR utilizando la clave privada creada anteriormente, usar la herramienta **jarsigner**. Para firmar el fichero *Contrato.jar* usando la clave privada del almacen de claves con el alias *FirmaContrato* y dejando el resultado de la firma en un archivo llamado *DocumentoFirmado.jar*, ejecutar el siguiente comando:

**$ jarsigner -keystore AlmacenClaves -signedjar DocumentoFirmado.jar Contrato.jar FirmaContrato**

Enter Passphrase for keystore:

jar signed.

Warning:

The signer certificate will expire within six months.

No -tsa or -tsacert is provided and this jar is not timestamped. Without a timestamp, users may not be able to validate this jar after the signer certificate's expiration date (2016-05-08) or after any future revocation date.

**Paso 4**

Los que vayan a recibir el fichero firmado *DocumentoFirmado.jar* necesitarán autenticar la firma del emisor. Para ello se necesita la clave pública correspondiente a la clave privada usada para generar su firmar. Para generar el certificado con clave pública (*Aitor.cer*) se usa el siguiente comando de la herramienta **keytool**:

**$ keytool -export -keystore AlmacenClaves -alias FirmaContrato -file Aitor.cer**

Introduzca la contraseña del almacén de claves:

Certificado almacenado en el archivo <Aitor.cer>

**Paso 5**

En este último paso el receptor recibe el fichero JAR firmado (*DocumentoFirmado.jar*) y el certificado que contiene la clave pública (*Aitor.cer*). El receptor necesita importar el certificado como un certificado de confianza (herramienta **keytool**) y verificar la firma del fichero JAR (herramienta **jarsigner**).

* Para importar el certificado como un certificado de confianza:

**$ keytool -import -alias Aitor -file Aitor.cer -keystore AlmacenReceptor**

Introduzca la contraseña del almacén de claves:

Volver a escribir la contraseña nueva:

Propietario: CN=Aitor, OU=CiudadJardin, O=IES Ciudad Jardin, L=Vitoria-Gasteiz, ST=Alava, C=ES

Emisor: CN=Aitor, OU=CiudadJardin, O=IES Ciudad Jardin, L=Vitoria-Gasteiz, ST=Alava, C=ES

Número de serie: 696daf1b

Válido desde: Mon Feb 08 21:25:24 CET 2016 hasta: Sun May 08 22:25:24 CEST 2016

Huellas digitales del Certificado:

MD5: 9D:07:6D:F1:E8:8F:1C:54:E3:4F:52:2A:70:09:6B:48

SHA1: C4:F9:E2:4E:89:AE:DB:42:7E:54:4A:96:08:04:81:15:BA:41:20:41

SHA256: CD:DC:6F:80:9B:36:39:22:55:4B:51:F5:4E:F0:69:FD:F3:68:6A:91:51:8C:C0:A1:A9:57:38:4E:E8:3E:5A:21

Nombre del Algoritmo de Firma: SHA1withDSA

Versión: 3

Extensiones:

#1: ObjectId: 2.5.29.14 Criticality=false

SubjectKeyIdentifier [

KeyIdentifier [

0000: 31 CE B4 55 53 7A 49 90 EB FD 88 9B 80 3E DC 24 1..USzI......>.$

0010: CB CF 7D 08 ....

]

]

¿Confiar en este certificado? [no]: si

Se ha agregado el certificado al almacén de claves

* Para verificar la autenticidad de la firma del fichero JAR:

**$ jarsigner -verify -verbose -certs -keystore AlmacenReceptor DocumentoFirmado.jar**

s k 152 Mon Feb 08 21:37:08 CET 2016 META-INF/MANIFEST.MF

X.509, CN=Aitor, OU=CiudadJardin, O=IES Ciudad Jardin, L=Vitoria-Gasteiz, ST=Alava, C=ES (aitor)

[certificate will expire on 8/05/16 22:25]

314 Mon Feb 08 21:37:08 CET 2016 META-INF/FIRMACON.SF

1122 Mon Feb 08 21:37:08 CET 2016 META-INF/FIRMACON.DSA

0 Mon Feb 08 21:23:28 CET 2016 META-INF/

smk 7804 Mon Feb 08 21:23:02 CET 2016 Contrato.pdf

X.509, CN=Aitor, OU=CiudadJardin, O=IES Ciudad Jardin, L=Vitoria-Gasteiz, ST=Alava, C=ES (aitor)

[certificate will expire on 8/05/16 22:25]

s = signature was verified

m = entry is listed in manifest

k = at least one certificate was found in keystore

i = at least one certificate was found in identity scope

jar verified.

Warning:

This jar contains entries whose signer certificate will expire within six months.

This jar contains signatures that does not include a timestamp. Without a timestamp, users may not be able to validate this jar after the signer certificate's expiration date (2016-05-08) or after any future revocation date.

# Encriptar con Java

Hasta ahora se ha usado la API de seguridad de Java para hacer autenticación de mensajes, documentos y ficheros mediante firmas digitales. La firma digital verifica que la información no ha cambiado. Cuando la información es autenticada su contenido queda visible y puede ser manipulado por algún intruso. Sin embargo, cuando la información está encriptada su contenido no es visible. Sólo puede desencriptarse con una clave coincidente.

La autenticación es suficiente para firmar la información, no hay necesidad de ocultarla**. La encriptación es necesaria** cuando las aplicaciones transfieren información confidencial como números de tarjetas de crédito, contraseñas u otros datos personales.

Java proporciona la clase **Cipher** (definida en el paquete javax.crypto.Cipher) para encriptar (cifrar) o desencriptar (o descifrar) información. Esta clase forma el núcleo de la extensión criptográfica de Java (JCE). El cifrado de un texto legible consite en transformarlo con ayuda de una clave en un texto ilegible; el descifrado es el proceso inverso, se toman los datos ilegibles y la clave y se produce texto legible.

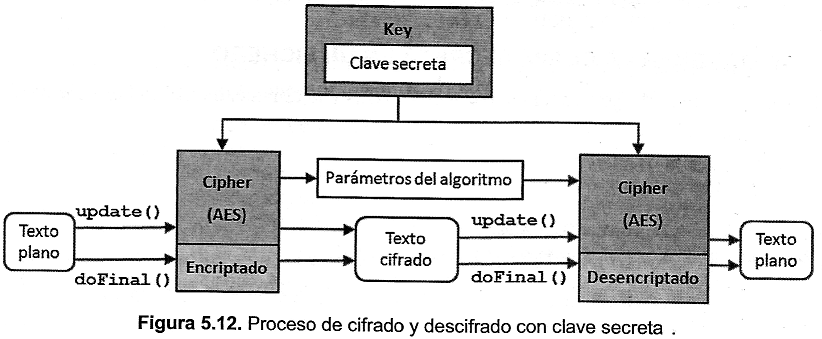
Para crear un objeto Cipher se llama al método *getInstance()* pasando como argumento el algoritmo y opcionalmente, se puede especificar el nombre de un proveedor.

Para proporcionar una clave al método *init()* del objeto Cipher se puede usar la clase **KeyGenerator** (definida en el paquete javax.crypto) que proporciona funcionalidad para generar claves secretas para usarse en **algoritmos simétricos**. Algunos de sus métodos son:

|  |  |
| --- | --- |
| **Método** | **Función** |
| static KeyGenerator getInstance(String algoritmo) | Devuelve un objeto KeyGenerator que genera claves secretas para el algoritmo especificado, por ejemplo DES. Puede lanzar la excepción NoSuchAlgorithmException. |
| void init(int keysize, SecureRandom random)  void init(int keysize)  Void init( SecureRandom random) | Inicializa el generador del claves para un determinado tamaño de clave y un generador de números aleatorios. En el segundo método, solo el tamaño de clave y en el tercero el generador. |
| SecretKey generateKey() | Genera una clave secreta. |

# Encriptar y desencriptar con clave privada

La siguiente figura muestra el esquema básico para encriptar y desencriptar con clave privada (o secreta). La misma clave se usará para encriptar y desencriptar la información.



*Figura 5.1 – Proceso de cifrado y descifrado con clave privada*

En primer lugar se crea la clave secreta usando el algoritmo AES (o también el algoritmo DES) y se define un tamaño para la clave de 128 bits.

Luego se crea un objeto Cipher con el algoritmo AES/ECB/PKCSSPadding y se inicializa en modo encriptación con la clave creada anteriormente.

A continuación se cifra la información con el método *doFinal()* de la clase Cipher.

Más adelante se configura el objeto Cipher en modo desencriptación con la clave anterior. Para desencriptar el texto se usa el método *doFinal()* de nuevo.

# Almacenar la clave secreta en un fichero

**Ejemplo 8 – Generar una clave secreta AES y almacenarla en el fichero *Clave.secreta***

**import** java.io.\*;

**import** java.security.NoSuchAlgorithmException;

**import** javax.crypto.KeyGenerator;

**import** javax.crypto.SecretKey;

**public** **class** Ejemplo8 {

**public** **static** **void** main(String[] args) {

**try** {

//Crear una clave secreta de 128 bits con el algoritmo AES

KeyGenerator kg = KeyGenerator.*getInstance*("AES");

kg.init(128);

SecretKey clave = kg.generateKey();

//Guardar la clave secreta en el fichero Clave.secreta

ObjectOutputStream out = **new** ObjectOutputStream( **new** FileOutputStream("Clave.secreta"));

out.writeObject(clave);

out.close();

} **catch** (NoSuchAlgorithmException | IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

# Encriptar y desencriptar con clave pública

En el **cifrado simétrico** o de clave privada se encripta y desencripta la información con la misma clave. El problema está en la distribución de la clave. Si un emisor encripta la información con una clave, el receptor de la información debe desencriptarla con la misma clave. Si el emisor cambia la clave, debe proporcionársela de nuevo al receptor, con lo que pone en peligro el cifrado, ya que alguien puede interceptar la clave.

En el **cifrado asimétrico** o de clave pública se resuelve el problema, ya que la clave para encriptar se puede compartir sin problemas entre el emisor y el receptor, y la clave para desencriptar sólo la tiene el receptor del mensaje. Supongamos que dos participantes A y B deben mantener una conversación secreta. Harían lo siguiente:

* El participante A crea un par de claves, una privada y una pública de cifrado, y manda la clave pública al participante B sin tener que tomar precauciones.
* El participante B crea su par de claves privada y pública, y manda al participante A la clave pública.
* El participante A crea un mensaje, lo cifra con la clave pública de B y le manda el mensaje. El participante B recibe el mensaje y lo descifra con su clave privada.
* El participante B crea un mensaje, lo cifra con la clave pública de A y le manda el mensaje. El participante A recibe el mensaje y lo descifra con su clave privada. Y así sucesivamente.

El esquema básico de **cifrado y descifrado con clave pública** es el siguiente:

* En primer lugar se crea el par de claves pública y privada (se puede almacenar en ficheros).
* Después se crea un objeto Cipher con el algoritmo RSA, RSA/ECB/PKS1Padding, y se inicializa en modo encriptación con la clave pública.
* Luego se realiza el cifrado de la información con el método *doFinal().*
* Más adelante se configura el objeto Cipher en modo desencriptación con la clave privada para desencriptar el texto, usando el método *doFinal()*.

El cifrado mediante clave pública (asimétrico) es más lento que el cifrado mediante clave privada (simétrico). No es práctico utilizar este cifrado para encriptar grandes candidades de información. Este problema se puede solucionar combinando cifrado asimétrico con cifrado simétrico. Este esquema combinado (**cifrado híbrido**) funciona de la siguiente forma:

* El participante A crea una clave de encriptación simétrica. La utiliza para encriptar sus mensajes de texto.
* El participante A encripta la clave simétrica con la clave pública del participante B.
* El participante A envía al participante B, tanto la clave simétrica encriptada como el mensaje de texto encriptado.
* El participante B utiliza su clave privada para desencriptar la clave simétrica de A.
* El participante B utiliza la clave simétrica desencriptada para desencriptar el mensaje.

Nadie excepto el participante B puede desencriptar la clave simétrica, ya que solo él tiene la clave privada para desencriptarla. La encriptación de clave pública sólo se aplica a una pequeña porción de datos.

**Ejemplo 9 – Generar un par de claves pública y privada con el algoritmo RSA. Crear una clave secreta con el algoritmo AES. Esta clave se usará para encriptar el texto. La clave secreta encriptada mediante la clave pública, utilizando el método *wrap()***

**import** java.security.InvalidKeyException;

**import** java.security.Key;

**import** java.security.KeyPair;

**import** java.security.KeyPairGenerator;

**import** java.security.NoSuchAlgorithmException;

**import** java.security.PrivateKey;

**import** java.security.PublicKey;

**import** javax.crypto.BadPaddingException;

**import** javax.crypto.Cipher;

**import** javax.crypto.IllegalBlockSizeException;

**import** javax.crypto.KeyGenerator;

**import** javax.crypto.NoSuchPaddingException;

**import** javax.crypto.SecretKey;

**public** **class** Ejemplo9 {

**public** **static** **void** main (String args[]) {

**try** {

// Crear las claves pública y privada con el algoritmo RSA

KeyPairGenerator keyGen = KeyPairGenerator.*getInstance*("RSA");

keyGen.initialize(1024);

KeyPair par = keyGen.generateKeyPair();

PrivateKey clavepriv = par.getPrivate();

PublicKey clavepub= par.getPublic();

// Crear la clave secreta con el algoritmo AES

KeyGenerator kg = KeyGenerator.*getInstance*("AES");

kg.init(128);

SecretKey clavesecreta=kg.generateKey();

// Encriptar la clave secreta con la clave pública RSA

Cipher c = Cipher.*getInstance*("RSA/ECB/PKCS1Padding");

c.init(Cipher.***WRAP\_MODE***, clavepub);

**byte** claveenvuelta[] = c.wrap(clavesecreta);

// Cifrar el texto con la clave secreta

c = Cipher.*getInstance*("AES/ECB/PKCS5Padding");

c.init(Cipher.***ENCRYPT\_MODE***, clavesecreta);

**byte** textoPlano[] ="Esto es un texto plano".getBytes();

**byte** textoCifrado [] = c.doFinal(textoPlano);

System.***out***.println("Encriptado:"+**new** String (textoCifrado));

// Para desencriptar el texto, primero hay que desencriptar la clave secreta con la clave privada

// y a continuación desencriptar el texto con esa clave. Se usará el método unwrap();

// Desencriptar la clave secreta con la clave privada RSA

Cipher c2 = Cipher.*getInstance*("RSA/ECB/PKCS1Padding");

c2.init(Cipher.***UNWRAP\_MODE***, clavepriv);

Key clavedesenvuelta = c2.unwrap(claveenvuelta, "AES", Cipher.***SECRET\_KEY***);

// Descifrar el texto con la clave clave secreta desencriptada

c2 = Cipher.*getInstance*("AES/ECB/PKCS5Padding");

c2.init(Cipher.***DECRYPT\_MODE***, clavedesenvuelta);

**byte** desencriptado[] = c2.doFinal(textoCifrado);

System.***out***.println("Desencriptado:" + **new** String(desencriptado));

} **catch** (NoSuchAlgorithmException | NoSuchPaddingException | InvalidKeyException | IllegalBlockSizeException | BadPaddingException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**Resultado:**

Encriptado:����)L��QHt���6<��0\*$=�K�����p

Desencriptado: Esto es un texto plano

# Encriptar y desencriptar con flujos de datos

La biblioteca JCE proporciona un conjunto de clases que encriptan o desencriptan automáticamente flujos de datos. Estas son **CipherOutputStream** para encriptar datos hacia un fichero y **CipherInputStream** para leer y desencriptar datos de un fichero. Estas clases manipulan de forma transparente las llamadas a los métodos *update()* y *doFinal()*.

Algunos métodos de estas clases son:

|  |  |
| --- | --- |
| **Método** | **Función** |
| CipherOutputStream(OutpuStream salida, Cipher codificador) | Construye un flujo de salida que escribe datos en salida y lo encripta o desencripta utilizando el objeto Cipher indicado. |
| void write(int b) | Escribe el byte especificado en el stream de salida |
| void write (byte[] b, int off, int len) | Escribe len bytes en el array de bytes especificado comenzando en off. |
| void flush() | Limpia el buffer del objeto Cipher y efectúa el relleno si es necesario |

|  |  |
| --- | --- |
| **Método** | **Función** |
| CipherInputStream(InpuStream salida, Cipher codificador) | Construye un flujo de entrada que escribe datos procedentes de la entrada y lo encripta o desencripta utilizando el objeto Cipher indicado. |
| void rea(int b) | Lee el byte especificado en el stream de entrada |
| void read (byte[] b, int off, int len) | Lee hasa len bytes de datos de este flujo de entra da en un array de bytes. |

**Ejemplo 10 – Usar la clave secreta almacenada en un fichero para cifrar un documento PDF de nombre FICHERO.pdf**

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**import** java.security.InvalidKeyException;

**import** java.security.Key;

**import** java.security.NoSuchAlgorithmException;

**import** javax.crypto.Cipher;

**import** javax.crypto.CipherOutputStream;

**import** javax.crypto.NoSuchPaddingException;

**public** **class** Ejemplo10 {

**public** **static** **void** main (String[] args) {

**try** {

// Recuperar la clave secreta del fichero

ObjectInputStream oin = **new** ObjectInputStream (**new** FileInputStream("Clave.secreta"));

Key clavesecreta = (Key) oin.readObject();

oin.close();

// Definir el objeto Cipher para encriptar

Cipher c = Cipher.*getInstance*("AES/ECB/PKCS5Padding");

c.init(Cipher.***ENCRYPT\_MODE***, clavesecreta);

// Fichero a cifrar

FileInputStream filein = **new** FileInputStream ("FICHERO.pdf");

// OBJETO CipherOutputStream donde se almacena el fichero cifrado

CipherOutputStream out = **new** CipherOutputStream(**new** FileOutputStream("FicheroPDF.Cifrado"),c);

**int** tambloque = c.getBlockSize(); // tamaño de bloque objeto Cipher

**byte**[] bytes= **new** **byte**[tambloque]; // bloque de bytes

// Leer bloques de bytes del fichero, escribiéndolo en CipherOutputStream

**int** i = filein.read(bytes);

**while** (i !=-1) {

out.write(bytes, 0,i);

i = filein.read(bytes);

}

out.flush();

out.close();

filein.close();

System.***out***.println("Fichero cifrado con la clave secreta");

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (NoSuchAlgorithmException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (NoSuchPaddingException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (ClassNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (InvalidKeyException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**Ejemplo 11 – Usar la clave secreta almacenada en un fichero para descifrar un documento *FICHEROPDF.Cifrado* cifrado anteriormente**

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**import** java.security.InvalidKeyException;

**import** java.security.Key;

**import** java.security.NoSuchAlgorithmException;

**import** javax.crypto.Cipher;

**import** javax.crypto.CipherInputStream;

**import** javax.crypto.NoSuchPaddingException;

**public** **class** Ejemplo11 {

**public** **static** **void** main (String[] args) {

**try** {

// Recuperar la clave secreta del fichero

ObjectInputStream oin = **new** ObjectInputStream (**new** FileInputStream("Clave.secreta"));

Key clavesecreta = (Key) oin.readObject();

oin.close();

// Definir el objeto Cipher para desencriptar

Cipher c = Cipher.*getInstance*("AES/ECB/PKCS5Padding");

c.init(Cipher.***DECRYPT\_MODE***, clavesecreta);

// OBJETO CipherIntputStream CUYO CONTENIDO SE VA A DESCIFRAR

CipherInputStream in = **new** CipherInputStream(**new** FileInputStream("FicheroPDF.Cifrado"),c);

**int** tambloque = c.getBlockSize(); // tamaño de bloque objeto Cipher

**byte**[] bytes= **new** **byte**[tambloque]; // bloque de bytes

/// FICHERO CON EL CONTENIDO DESCIFRADO QUE SE CREARÁ

FileOutputStream fileout = **new** FileOutputStream ("FICHEROdescifrado.pdf");

// LEEMOS BLOQUES DE BYTES DEL FICHERO cifrado

// Y LO VAMOS ESCRIBIENDO AL desencriptado

**int** i = in.read(bytes);

**while** (i !=-1) {

fileout.write(bytes, 0,i);

i = in.read(bytes);

}

fileout.close();

in.close();

System.***out***.println("Fichero descifrado con la clave secreta");

} **catch** (IOException | ClassNotFoundException | NoSuchAlgorithmException | NoSuchPaddingException | InvalidKeyException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

# COMUNICACIONES SEGURAS CON JAVA. JSSE

Los datos que viajan a través de la red pueden ser accedidos por personas que no son destinatarias de los mismos. Cuando incluyen información privada, como contraseñas o números de tarjetas de crédito, se deben tomar medidas para que sean incomprensibles para las partes no autorizadas. También es importante asegurarse de que los datos no se modifiquen durante el transporte, ya sea intencionadamente o no. Los protocolos **SSL** (Secure Sockets Layer) y **TLS** (Transport Layer Security) se han diseñado para ayudar a proteger la privacidad y la integridad de los datos mientras se transportan a través de una red.

**JSSE** (Java Secure Socket Extension) es un conjunto de paquetes que permiten el desarrollo de aplicaciones seguras en Internet. Proporciona un marco y una implementación para la versión Java de los protocolos **SSL y TLS** e incluye funcionalidad de encriptación de los datos, autenticación de servidores, integridad de mensajes y autenticación de clientes. Con JSSE, los desarrolladores pueden ofrecer intercambio seguro de datos entre un cliente y un servidor que ejecuta un protocolo de aplicación (HTTP, Telnet, FTP, …) a través de TCP/IP. Las clases de JSSE se encuentran en los paquetes **javax.net y javax.net.ssl**.

Java incorpora las clases **SSLSocket y SSLServerSocket** que representan sockets seguros y son derivadas de las clases Socket y ServerSocket, respectivamente. Aparte de estas clases, Java dispone de otras dos clases (**SSLSocketFactory** y **SSLServerSocketFactory**), que sirven para la creación de sockets seguros.

Para obtener un socket servidor seguro o **SSLServerSocket**:

SSLServerSocketFactory sfact = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();

SSLServerSocket servidorSSL= (SSLServerSocket) sfact.createServerSocket(puerto);

El método *createServerSocket(int puerto)* devuelve un socket de servidor enlazado al puerto especificado.

Para obtener un socket cliente seguro o **SSLSocket**:

SSLSocketFactory sfact = (SSLSocketFactory) SSLSocketFactory.getDefault();

SSLSocket Cliente= (SSLSocket) sfact.createSocket(Host,puerto);

Para el correcto funcionamiento de los programas que usan estos sockets seguros es preciso configurar adecuadamente las variables de entorno PATH y CLASSPATH, tanto las correspondientes al usuario como a las del sistema.

Asimismo, si los programas Java están incluidos en algún paquete y quieren ejecutarse desde la línea de comandos (con permisos de administrador), es preciso indicar el nombre del paquete más un punto, antes del nombre del archivo que contiene los bytecodes (\*.class) del programa a ejecutar.

**Ejemplo 12 – El programa servidor (ServidorSSL) crea una conexión sobre un socket servidor seguro y atiende hasta 4 conexiones de clientes que se identificarán con un certificado válido. El servidor espera las conexiones de los clientes. De cada cliente que se conecta, el servidor recibe un mensaje y a continuación le envía un saludo al cliente**

**import** java.io.\*;

**import** javax.net.ssl.\*;

**public** **class** ServidorSSL {

**public** **static** **void** main (String[] arg) **throws** IOException {

**int** puerto=6000;

SSLServerSocketFactory sfact = (SSLServerSocketFactory) SSLServerSocketFactory.*getDefault*();

SSLServerSocket servidorSSL= (SSLServerSocket) sfact.createServerSocket(puerto);

SSLSocket clienteConectado = **null**;

DataInputStream flujoEntrada = **null**;

DataOutputStream flujoSalida = **null**;

**for** (**int** i =1; i < 5 ; i++) {

System.***out***.println("Esperando al cliente " + i);

clienteConectado = (SSLSocket) servidorSSL.accept();

flujoEntrada = **new** DataInputStream(clienteConectado.getInputStream());

// El servidor recibe un mensaje del cliente

System.***out***.println("Recibiendo del cliente: " + i + "\n\t" + flujoEntrada.readUTF());

flujoSalida = **new** DataOutputStream (clienteConectado.getOutputStream());

// Saludo del servidor al cliente

flujoSalida.writeUTF("Saludos del servidor al cliente");

}

// Cierre de streams y sockets

flujoEntrada.close();

flujoSalida.close();

clienteConectado.close();

servidorSSL.close();

}

}

El **programa cliente (ClienteSSL)** envía un mensaje al servidor y visualiza el que el servidor le devuelve:

**import** java.io.\*;

**import** javax.net.ssl.\*;

**public** **class** ClienteSSL {

**public** **static** **void** main (String[] arg) **throws** IOException {

String Host="localhost";

**int** puerto=6000;

System.***out***.println("PROGRAMA CLIENTE INICIADO....");

SSLSocketFactory sfact = (SSLSocketFactory) SSLSocketFactory.*getDefault*();

SSLSocket Cliente= (SSLSocket) sfact.createSocket(Host,puerto);

// Creación del flujo de salida al servidor

DataOutputStream flujoSalida = **new** DataOutputStream (Cliente.getOutputStream());

// Envío de un saludo al servidor

flujoSalida.writeUTF("Saludos al servidor desde el cliente");

// Creación del flujo de entrada al servidor

DataInputStream flujoEntrada = **new** DataInputStream(Cliente.getInputStream());

// El servidor me envia un mensaje

System.***out***.println("Recibiendo del SERVIDOR: \n\t" + flujoEntrada.readUTF());

// Cierre de streams y sockets

flujoEntrada.close();

flujoSalida.close();

Cliente.close();

}

}

El **servidor necesita disponer de un certificado** que mostrar a los clientes para que se conecten con él. Se usará la herramienta *keytool* para crearlo, el nombre del almacén de claves que se creará será *AlmacenSSL*, y el valor de la clave (contraseña) será *1234567*:

**$ keytool -genkey -alias claveSSL -keyalg RSA -keystore AlmacenSSL -storepass 1234567**

¿Cuáles son su nombre y su apellido?

[Unknown]: Manolita Desconocida

¿Cuál es el nombre de su unidad de organización?

[Unknown]: Ciudad Jardin

¿Cuál es el nombre de su organización?

[Unknown]: CIFP Ciudad Jardin LHII

¿Cuál es el nombre de su ciudad o localidad?

[Unknown]: Vitoria-Gasteiz

¿Cuál es el nombre de su estado o provincia?

[Unknown]: Araba-Alava

¿Cuál es el código de país de dos letras de la unidad?

[Unknown]: ES

¿Es correcto CN=Aitor, OU=CiudadJardin, O=CIFP Ciudad Jardin LHII, L=Vitoria-Gasteiz, ST=Araba-Alava, C=ES?

[no]: si

Introduzca la contraseña de clave para <claveSSL>

(INTRO si es la misma contraseña que la del almacén de claves):

La contraseña de clave es demasiado corta; debe tener al menos 6 caracteres

Introduzca la contraseña de clave para <claveSSL>

(INTRO si es la misma contraseña que la del almacén de claves):

Para **ejecutar el programa servidor** es necesario indicar el certificado que se utilizará. Para ello es necesaria la siguiente orden desde la línea de comandos:

**$ java -Djavax.net.ssl.keyStore=AlmacenSSL -Djavax.net.ssl.keyStorePassword=1234567 ServidorSSL**

Esperando al cliente 1

Antes de **ejecutar el programa cliente** es necesario colocar el certificado en el keystore del usuario. Para ello se exporta a un fichero llamado, por ejemplo, *Certificado.cer:*

**$ keytool -export -alias claveSSL -keystore AlmacenSSL -storepass 1234567 -file Certificado.cer**

Certificado almacenado en el archivo <Certificado.cer>

Una vez que se dispone del fichero exportado, es necesario incorporarle al nuevo almacenamiento para permitir realizar la validación. A continuación se crea un keystore de nombre *UsuarioAlmacenSSL* con la clave *890123* y se incorpora el fichero de certificado *Certificado.cer*. Esto se hace con la siguiente orden en la línea de comandos:

**$ keytool -import -alias claveSSL -file Certificado.cer -keystore UsuarioAlmacenSSL**

**-storepass 890123**

Propietario: CN=Manolita Desconocida, OU=Ciudad Jardin, O=CIFP Ciudad Jardin LHII, L=Vitoria-Gasteiz, ST=Araba-Alava, C=ES

Emisor: CN=Manolita Desconocida, OU=Ciudad Jardin, O=CIFP Ciudad Jardin LHII, L=Vitoria-Gasteiz, ST=Araba-Alava, C=ES

Número de serie: 26b5b227

Válido desde: Tue Feb 09 12:39:34 CET 2016 hasta: Mon May 09 13:39:34 CEST 2016

Huellas digitales del Certificado:

MD5: 40:71:43:0A:7C:55:CC:BE:ED:E3:6A:B6:C5:EA:70:34

SHA1: B5:A6:33:62:A9:AE:E7:65:51:F6:24:6A:70:5D:03:52:1B:89:D5:CA

SHA256: 6B:2E:51:5B:CE:6E:FB:2B:05:B3:9F:09:83:CF:DC:96:83:5D:7F:B5:98:B6:ED:86:D2:8F:98:36:3C:9E:AA:50

Nombre del Algoritmo de Firma: SHA256withRSA

Versión: 3

Extensiones:

#1: ObjectId: 2.5.29.14 Criticality=false

SubjectKeyIdentifier [

KeyIdentifier [

0000: 65 7A D9 3E 16 F1 30 5D DD 23 2C 32 F9 38 12 BA ez.>..0].#,2.8..

0010: 71 2C D1 6F q,.o

]

]

¿Confiar en este certificado? [no]: si

Se ha agregado el certificado al almacén de claves

Para ejecutar el programa cliente se escribe lo siguiente**:**

**$ java -Djavax.net.ssl.trustStore=UsuarioAlmacenSSL -Djavax.net.ssl.trustStorePassword=890123 ClienteSSL**

PROGRAMA CLIENTE INICIADO….

Recibiendo del SERVIDOR:

Saludos del cliente al servidor

En este ejemplo, para ejecutar los programas servidor y cliente se han establecido propiedades JSSE desde la línea de comandos y usando la sintaxis *-Dpropiedad=Valor*. El valor de estas propiedades también puede establecerse usando el método *System.setProperty(String propiedad, String valor)* de la clase *System*. En la siguiente tabla se muestran las propiedades JSSE:

|  |  |
| --- | --- |
| **PROPIEDAD** | **SIGNIFICADO** |
| Javax.net.ssl.keyStore | Ubicación del fichero de almacén de claves de Java. En Windows, la ruta de acceso especificade debe utilizar las barras inclinadas / en lugar de las barras invertidas, \. |
| Javax.net.ssl.keyStorePassword | Contraseña para acceder a la clave privada en el fichero de almacén de claves especificado por javax.net.ssl.keyStore. |
| Javax.net.ssl.trustStore | Ubicación del fichero de almacén de claves que contiene la colección de certificados de confianza. En Windows, la ruta de acceso especificada debe utilizar barras inclinadas, /, en lugar de barras invertidas, \. |
| Javax.net.ssl.trustStorePassword | Contraseña para abrir el fichero de almacén de claves especificado por javax.net.ssl.trustStore. |

En el programa **ServidorSSL** se incluirían las siguientes líneas después de inicializar la variable puerto:

System.*setProperty*("javax.net.ssl.keyStore","AlmacenSSL");

System.*setProperty*("javax.net.ssl.keyStorePassword","1234567");

En el programa **ClienteSSL** se incluirían las siguientes líneas después de inicializar la variable puerto:

System.*setProperty*("javax.net.ssl.trustStore","UsuarioAlmacenSSL");

System.*setProperty*("javax.net.ssl.trustStorePassword","890123");

# CONTROL DE ACCESO CON JAVA. JAAS

**JAAS** (Java Authentication and Authorization Service, servicio de autorización y autenticación de Java) es una interfaz que permite a las aplicaciones Java acceder a servicios de control de autenticación y acceso. Se puede utilizar para dos propósitos:

* Para **autenticación de usuarios**: para determinar de forma fiable y segura quién está ejecutando nuestro código Java, y
* Para **autorización de los usuarios**: para asegurarse de que quien lo ejecuta tiene los permisos necesarios para realzar acciones.

En el proceso de autenticación y autorización mediante JAAS están involucradas las siguientes clases e interfaces Java:

* **LoginContext**, contexto de inicio de sesión: inicia y gestiona el proceso de autenticación mediante la creación de un Subject. La autenticación se hace llamando al método *login().*
* **LoginModule**, módulo de conexión: es la interfaz que se debe implementar para definir los mecanismos de autenticación en la aplicación. Se deben implementar los siguientes métodos: *initialize(),* *login(),* *commit(), abort()* y *logout()*. Se encarga de validar los datos en un proceso de autenticación.
* **Subject**: clase que representa a un ente autenticable dentro de la aplicación (entidad, usuario, sistema).
* **Principal**: clase que representa los atributos cada Subject recuperado una vez que se efectúa el ingreso a la aplicación. Un Subject puede contener varios principales.
* **CallbackHandler**: interfaz que se debe implementar cuando se necesita recibir del usuario la información para la autenticación, se encarga de la interacción con el usuario para obtener los datos de autenticación. Al implementarla se debe desarrollar el método *handle().*

Los paquetes en los que están disponibles las clases e interfaces principales de JAAS son:

* **javax.security.auth.\*:** contiene las clases de base e interfaces para los mecanismos de autenticación y autorización.
* **javax.security.auth.callback.\*:** contiene las clases e interfaces para definir las crecenciales de autenticación de la aplicación.
* **javax.seucirty.auth.login.\*:** contiene las clases para entrar y salir de un dominio de aplicación.
* **javax.security.auth.spi.\*:** contiene interfaces para un proveedor de JAAS para implementar módulos JAAS.

En los siguientes apartados se verán ejemplos básicos para llevar a cabo la autenticación y autorización con JAAS.

# Autenticación

El proceso básico de autenticación con JAAS consta de los siguientes pasos:

1. Creación de una instancia de **LoginContext**, uno o más **LoginModule** son cargados basándose en el archivo de configuración JAAS.
2. La instalación de cada **LoginModule** es opcionalmente provista con un **CallbackHandler** que gestionará el proceso de comunicación con el usuario para obtener los datos con los que éste tratará de autenticarse.
3. Invocación del método *login()* del **LoginContext**, el cual invocará el método *login()* del **LoginModule**.
4. Los datos del usuario se obtienen por medio del **CallbackHandler**.
5. El **LoginModule** compureba los datos introducidos por el usuario y valida. Si la validación tiene éxito el usuario queda autenticado.

**Ejemplo12 - Para probar la autenticación básica con JAAS se van a crear las siguientes clases Java y ficheros:**

* Fichero *EjemploJaasAuthenticacion.java*, es la aplicación que será autenticada y autorizada mediante JAAS.
* Fichero *EjemploLoginModule.java*, implementa el módulo **LoginModule** que autentica a los usuarios mediante su nombre y contraseña.
* Fichero *MyCallbackHandler.java* que implementa **CallbackHandler**. Transfiere la información requerida al módulo **LoginModule**.
* Fichero de autenticación JAAS, **jaas.config,** donde se configura el módulo de login. En el ejemplo se vincula el nombre *EjemploLogin* al módulo **LoginModule** de nombre *EjemploLoginModule*, la palabra *required* al lado indica que el módulo de conexión asociado debe ser capaz de autenticar al sujeto en todo el proceso de autenticación para poder tener éxito. El contenido es:

EjemploLogin {

EjemploLoginModule required;

};

* Fichero de autorización JAAS, **policy.config**. En el se van a incluir los permisos de lectura sobre las propiedades *usuario* y *clave* que se introducirán desde la línea de comandos y el permiso para crear un contexto de inicio de sesión, *createLoginContext*, al nombre *EjemploLogin* vinculado con la clase **EjemploLoginModule**. El contenido del fichero es el siguiente:

grant {

permission java.util.PropertyPermission "usuario", "read";

permission java.util.PropertyPermission "clave", "read";

permission javax.security.auth.AuthPermission "createLoginContext.EjemploLogin";

};

A este fichero de configuración *policy.config* se le puede añadir la siguiente entrada:

/Library/Java/JavaVirtualMachines/jdk1.8.0\_101.jdk/Contents/Home/jre/lib/security/java.policy

El código de los ficheros Java se muestra a continuación:

**Ejemplo13 - Código del fichero *EjemploJaasAutenticacion.java***

**import** javax.security.auth.callback.CallbackHandler;

**import** javax.security.auth.login.\*;

**public** **class** EjemploJaasAutenticacion {

**public** **static** **void** main(String[] args) {

// datos proporcionados desde la linea de comandos

String user = System.*getProperty*("usuario");

String pass = System.*getProperty*("clave");

// Paso al CallbackHandler el nombre de usuario y la clave para que

// el LoginModule acceda a ellos

CallbackHandler handler = **new** MyCallbackHandler (user,pass);

**try** {

LoginContext loginContex = **new** LoginContext("EjemploLogin", handler);

// llamada al método login para realizar la autenticación

loginContex.login();

System.***out***.println("Usuario autenticado....");

} **catch** (LoginException e) {

// **TODO** Auto-generated catch block

System.***err***.println("ERROR->No puede autenticar el usuario");

}

}

}

**Ejemplo13 - Código del fichero *MyCallbackHandler.java***

**import** javax.security.auth.callback.Callback;

**import** javax.security.auth.callback.CallbackHandler;

**import** javax.security.auth.callback.NameCallback;

**import** javax.security.auth.callback.PasswordCallback;

**public** **class** MyCallbackHandler **implements** CallbackHandler {

**private** String usuario;

**private** String clave;

//// Constructor recibe parámetros usuario y clave

**public** MyCallbackHandler (String usu, String clave) {

**this**.usuario = usu;

**this**.clave = clave;

}

// metodo handle sera invocado por el LoginModule

**public** **void** handle (Callback[] callbacks) {

**for** (**int** i = 0; i < callbacks.length; i++) {

Callback callback = callbacks[i];

**if** (callback **instanceof** NameCallback) {

NameCallback nameCB= (NameCallback) callback;

// se asigna al NameCallback el nombre de usuario

nameCB.setName(usuario);

} **else** **if** (callback **instanceof** PasswordCallback) {

PasswordCallback passwordCB = (PasswordCallback) callback;

// se asigna al PasswordCallback la clave

passwordCB.setPassword(clave.toCharArray());

}

}

}

}

**Ejemplo13 - Código del fichero *EjemploLoginModule .java***

**import** java.io.IOException;

**import** java.util.Map;

**import** javax.security.auth.Subject;

**import** javax.security.auth.callback.Callback;

**import** javax.security.auth.callback.CallbackHandler;

**import** javax.security.auth.callback.NameCallback;

**import** javax.security.auth.callback.PasswordCallback;

**import** javax.security.auth.callback.UnsupportedCallbackException;

**import** javax.security.auth.login.LoginException;

**public** **class** EjemploLoginModule {

**private** Subject subject;

**private** CallbackHandler callbackHandler;

**public** **boolean** commit () **throws** LoginException {**return** **true**;}

**public** **boolean** logout () **throws** LoginException {**return** **true**;}

**public** **boolean** abort () **throws** LoginException {**return** **true**;}

**public** **void** initialize (Subject subject, CallbackHandler handler, Map state, Map options) {

**this**.subject = subject;

**this**.callbackHandler = handler;

}

// método login - se realiza la autenticación

**public** **boolean** login() **throws** LoginException {

**boolean** autenticado = **true**;

**if** (callbackHandler == **null**) {

**throw** **new** LoginException ("Se necesita CallbackHandler");

}

// Se crea el Array de Callbacks

Callback[] callbacks = **new** Callback[2];

// Constructor de NameCallback y PasswordCallback con promt

callbacks[0] = **new** NameCallback ("Nombre de usuario: ");

callbacks[1] = **new** PasswordCallback ("Clave:", **false**);

**try** {

// se invoca al método handle del CallbackHandler para solicitar el usuario y la contraseña

//

callbackHandler.handle(callbacks);

String usuario = ((NameCallback) callbacks[0]).getName();

**char** [] passwd = ((PasswordCallback) callbacks[1]).getPassword();

String clave = **new** String (passwd);

// La autenticación se realiza aqui

// el nombre de usuario: aitor y su clave 1234

autenticado = ("aitor".equalsIgnoreCase(usuario) & "1234".equals(clave));

} **catch** (IOException | UnsupportedCallbackException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("Autenticado"+autenticado);

**return** autenticado; // devuelve true o false

}

}

**Ejemplo 13 – Ejecución:**

Para ejecutar este ejemplo todos los ficheros de configuración y los ficheros compilados de Java (\*.class) tienen que estar en la misma carpeta.

* Primero se ejecuta el programa *EjemploJaasAutenticacion* con el gestor de seguridad habilitado, indicando el fichero de políticas y el de configuración JAAS y proporcionando una clave correcta (1234):

**$ java -Dusuario=aitor -Dclave=1234 -Djava.security.manager -Djava.security.policy=policy.config -Djava.security.auth.login.config=jaas.config EjemploJaasAutenticacion**

Usuario autenticado....

* Luego se ejecuta el programa *EjemploJaasAutenticacion* con el gestor de seguridad habilitado, indicando el fichero de políticas y el de configuración JAAS y proporcionando una clave incorrecta (12347):

**$ java -Dusuario=aitor -Dclave=12347 -Djava.security.manager -Djava.security.policy=policy.config -Djava.security.auth.login.config=jaas.config EjemploJaasAutenticacion**

ERROR->No puede autenticar el usuario

# Autorización

La autorización de JAAS extiende la arquitectura de seguridad de Java centrada en el código y se basa en el uso de políticas de seguridad para especificar cuáles son los permisos de control de acceso que se concederán para ejecutar el código. Los permisos se otorgarán no sólo en función de **qué** código se está ejecutando, sino también en **quién** lo está ejecutando.

Cuando una aplicación utiliza la autenticación de JAAS para autenticar al usuario (u otra entidad, como un servicio), se crea un **Subject** como resultado, que representará al usuario autenticado. Un **Subject** se compone de un conjunto de principales (clase **Principal**), donde cada principal representa un atributo para ese usuario. Por ejemplo, un **Subject** puede tener dos principales: uno es el nombre y el otro su DNI.

Para que la autorización JAAS tenga lugar, es necesario lo siguiente:

* El usuario debe autenticarse (esto se ha visto en el apartado de Autenticación).
* En el fichero de políticas se deben configurar entradas para los principales.
* Se debe asociar al Subject el contexto de control de acceso actual, usando los métodos *doAs()* o *doAsPrivileged()* de la clase Subject.