## [**Programación segura III. Seguridad en entornos Java**](http://www.programandoapasitos.com/2016/03/procesos-y-servicios-programacion_24.html)

Antes de que la JVM comience el proceso de interpretación y ejecución de los *bytecodes* (código objeto) debe realizar una serie de tareas para preparar el entorno en el que el programa se ejecutará. Este es el punto en el que se implementa la seguridad interna de Java.

Hay tres componentes en el proceso:

* El cargador de clases
* El verificador de ficheros de clases
* El gestor de seguridad

**El cargador de clases**

Es el responsable de encontrar y cargar los bytecodes que definen las clases. Cada programa Java tiene como mínimo tres cargadores:

* El cargador de clases bootstrap que carga las clases del sistema (normalmente desde el fichero JAR rt.jar)
* El cargador de clases de extensión que carga una extensión estándar desde el directorio jre/lib/ext
* El cargador de clases de la aplicación que localiza las clases y los ficheros **JAR/ZIP** de la ruta de acceso a las clases (según está establecido por la variable de entorno **CLASSPATH** o por la opción –classpath de la línea de comandos)

**El verificador de ficheros de clases**

Se encarga de validar los bytecodes. Algunas de las comprobaciones que lleva a cabo son:

* Que las variables estén inicializadas antes de ser utilizadas
* Que las llamadas a un método coinciden con los tipos de referencias a objetos
* Que no se han infrigido las reglas para el acceso a los métodos y clases privados, etc.

**El gestor de seguridad**

Es una clase que controla si está permitida una determinada operación.

Alguna de las operaciones que comprueban son las siguientes:

* Si el hilo actual puede cargar un subproceso
* Si puede acceder a un paquete específico
* Si puede acceder o modificar las propiedades del sistema
* Si puede leer desde o escribir en un fichero específico
* SI puede eliminar un fichero específico
* Si puede aceptar una conexión socket desde un host o número de puerto específico, etc.

Por defecto no se instala de forma automática ningún gestor de seguridad cuando se ejecuta una aplicación Java. En el siguiente ejemplo veremos la salida que produce el programa ejecutándolo sin gestor de seguridad y con gestor de seguridad. El programa muestra los valores de ciertas propiedades de sistema (usamos el método System.getProperty(propiedad) para mostrar los valores), la siguiente tabla describe alguna de las más importantes:

**Archivo EJEMPLO1.JAVA**

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

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

//propiedades de sistema en un array

String t[] = { "java.class.path", "java.home", "java.vendor",

"java.version", "os.name", "os.version","user.dir",

"user.home", "user.name"};

System.*setProperty* ("java.security.policy", System.*getProperty*("user.dir") + "\\src\\\_01SinConGestor\\Politica1.policy");

System.*setSecurityManager*(**new** SecurityManager());

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

System.***out***.print("Propiedad:" + t[i]);

**try** {

String s = System.*getProperty*(t[i]); //valor de la propiedad

System.***out***.println("\t==> " + s);

} **catch** (Exception e) { System.***err***.println("\n\tEXcepción " + e.toString()); }

}//Fin de for

}//Fin de main

}//Fin de Ejemplo1

**Archivo POLITICA1.POLICY (para política de permisos)**

grant {

permission java.util.PropertyPermission "java.class.path", "read";

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

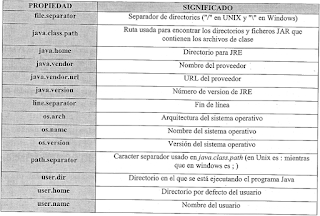
permission java.util.PropertyPermission "user.home", "read";

permission java.util.PropertyPermission "user.name", "read";

permission java.util.PropertyPermission "user.dir", "read";

permission java.util.PropertyPermission "on.version", "read";

};



**APIs Java para seguridad**

**JCA (*Java Cryptography Architecture*, Arquitectura Criptográfica de Java).** Es un marco de trabaj para acceder y desarrollar funciones criptográficas en la plataforma Java. Proporciona la infraestructura para la ejecución de los principales servicios de cifrado, incluyendo:

* Firmas digitales
* Resúmenes de mensajes (hash)
* Certificados y validación de certificados
* Encriptación (cifrado de bloques, cifrado simétrico y asimétrico)
* Generación y gestión de claves y generación segura de números aleatorios

**JSSE (*Java Secure Extension*, Extensión de Sockets Seguros Java).** Es un conjunto de paquetes Java provistos para la comunicación segura en Internet. Implementa una versión Java de los protocolos SSL y TLS. Incluye funcionalidades como:

* Cifrado de datos
* Autenticación del servidor
* Integridad de mensajes
* Autenticación del cliente

**JAAS (*Java Authentication and Authorization Service*, Servicio de Autenticación y Autorización de Java).** Es un interfaz que permite a las aplicaciones Java acceder a servicios de control de autenticación y acceso. Puede usarse con dos fines:

* La autenticación de usuarios para conocer quién está ejecutando código Java.
* La autorización de usuarios para garantizar que quién lo ejecuta tiene los permisos necesarios para hacerlo.

**Ficheros de política en Java**

**Localización**

El fichero de políticas predeterminado **java.policy**, está localizado en los siguientes directorios.

* Sistemas operativos Windows: java.home\lib\security\java.policy
* Sistemas UNIX: java.home/lib/security/java.policy

Donde java.home representa el valor de la propiedad **java.home**, que especifica el directorio en el que se instaló el JRE.

El valor por defecto es tener un solo fichero de políticas en todo el sistema y un fchero de políticas en el directorio home (dado por la variable **user.home**) del usuario (este tiene un punto delante).

En el fichero **java.security** localizado en la carpeta java.home\lib\security\ se encuentran las localizaciones de estos ficheros:

policy.url.1=file:${java.home}/lib/security/java.policy

policy.url.2=file:${user.home}/.java.policy

**Entrada grant**

Un fichero de políticas especifica qué permisos están disponibles para el código de varias fuentes.

Se utiliza para conceder permisos del sistema.

Contiene una secuencia de entradas **grant**, cada una de ellas tiene una o más entradas, el formato básico es el siguiente:

grant codeBase "URL"{

permission Nombre\_clase "Nombre\_destino", "Acción";

permission Nombre\_clase "Nombre\_destino", "Acción";

}

A la derecha de **codeBase** se indica la ubicación del código base sobre el que se van a definir los permisos. Su valor es una URL y siempre se debe utiliza la barra diagonal (/) como separador de directorio incluso para las URL de tipo file en Windows. Por ejemplo: grant codeBase “file:/C:/somepath/api/”.

Si se omite **codeBase** entonces los permisos se aplican a todas las fuentes.

**Nombre\_clase** contiene el nombre de la clase de permisos, por ejemplo:

* java.io.FilePermission (representa acceso a fichero o directorio)
* java.net.SocketPermission (acceso a la red vía socket)
* java.util.PropertyPermission (permiso sobre propiedades del sistema), etc.

El parámetro **Nombre\_destino** especifica el destino del permiso y depende de la clase de permiso.

Por ejemplo si la clase de permiso es java.io.FilePermission en **Nombre\_destino** se puede poner un fichero o un directorio; si la clase es **java.net.SocketPermission** se pondría un servidor y un número de puerto; si es **java.util.PropertyPermission** se pondría una propiedad del sistema.

En el parámetro **Acción** se indica una lista de acciones separadas por comas, por ejemplo **read**, **write**, **delete** o **execute** para una clase de permiso java.io.FilePermission; **accept**, **listen**, **connect**, **resolve** para una clase de permiso java.netSocketPermission; **read**, **write** para una clase de permiso java.util.PropertyPermission.

Desde la URL:

http://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html

Se puede consultar los permisos, sus destinos y acciones.

**Ejemplo de ficheros de políticas**

Creamos un fichero en una carpeta determinada (en Windows C:\Ficheros) e insertamos una línea, después lee el contenido del fichero y lo muestra en pantalla. El comportamiento del programa será diferente si usamos o no el gestor de seguridad y ficheros de políticas.

Se añade el gestor de seguridad en el método main(), la línea System.setSecurityManager(new SecurityManager()); antes de la cláusula try, la salida muestra un error de acceso denegado al escribir datos en el fichero, tampoco se podrá realizar la lectura del mismo.

**Archivo EJEMPLO2.JAVA**

**import** java.io.BufferedReader;

**import** java.io.BufferedWriter;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**import** java.io.IOException;

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

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

//El directorio C:/Ficheros debe estar creado previamente.

//System.setProperty ("java.security.policy", System.getProperty("user.dir") + "\\src\\\_02FicherosPoliticas\\Politica2.policy");

//System.setSecurityManager(new SecurityManager());

**try** {

//escritura en fichero

BufferedWriter fichero = **new** BufferedWriter (**new** FileWriter("C://Ficheros//Fichero.txt"));

fichero.write("Escritura de una linea en fichero.");

fichero.newLine(); // escribe un salto de línea

fichero.close();

System.***out***.println("Final proceso de escritura...");

//lectura del fichero

BufferedReader fichero2 = **new** BufferedReader (**new** FileReader("C://Ficheros//Fichero.txt"));

String linea = fichero2.readLine();

System.***out***.println("Contenido del fichero: ");

System.***out***.println("\t" + linea);

fichero2.close();

System.***out***.println("Final proceso de lectura...");

} **catch** (FileNotFoundException fn) {

System.***out***.println("No se encuentra el fichero");

} **catch** (IOException io) {

System.***out***.println("Error de E/S ");

} **catch** (Exception e) {

System.***out***.println("ERROR => " + e.toString());

}//Fin de try

}// Fin de main

}// Fin de Ejemplo2

Ahora se crea el siguiente fichero de políticas de nombre Politica2.policy con el siguiente contenido:

**Archivo POLITICA2.POLICY**

grant codeBase "file:${user.dir}/bin/"{

permission java.io.FilePermission "C:\\Ficheros\\\*", "read, write";

};

que permite a los programas localizados en la carpeta indicada (incluido permiso para crear) ficheros en la ruta C\Ficheros.

**Formato grant**

El parámetro **Nombre\_destino** para la clase de permiso **java.io.FilePermission** puede terminar en una serie de caracteres:

* “/” (donde “/” es el carácter separador de ficheros) indica un directorio y todos los ficheros contenidos en ese directorio.
* “/-“ indica un directorio y (recursivamente) todos los ficheros y subdirectorios contenidos en ese directorio.
* “<<ALL FILES>>” indica cualquier fichero
* También es posible usar propiedades de sistema como **user.dir** o **user.home** para dar privilegios sobre el directorio en el que se está ejecutando el programa o sobre el directorio por defecto del usuario. Se debe utilizar la propiedad encerrada entre llaves y con el caracer $ delante. **${propiedad}**.

El parámetro **Nombre\_destino** para la clase de permiso **java.net.SocketPermission** tiene el siguiente formato:

Host = (nombre del host | Direccion IP) [:número de puerto] donde número de puerto = N | -N | N-[M]

Las acciones son **accept**, **listen**, **connect** y **resolve**.

El host se especifica como un nombre de host o dirección IP.

Especificación de puerto:

* “N”, puerto N
* “N-“, todos los puertos desde N en adelante
* “-N”, todos los puertos desde N hacia atrás
* “M-N”, rango de puertos

**Tercer ejemplo de ficheros de políticas de Sockets**

**Archivo EJEMPLO3SERVIDOR.JAVA**

**import** java.net.ServerSocket;

**import** java.net.Socket;

**public** **class** Ejemplo3Servidor {

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

**int** numeroPuerto = 6000;// Puerto

ServerSocket servidor = **null**;

System.*setProperty* ("java.security.policy", System.*getProperty*("user.dir") + "\\src\\\_03FicherosPoliticasSockets\\Politica3.policy");

System.*setSecurityManager*(**new** SecurityManager());

**try** {

servidor = **new** ServerSocket(numeroPuerto);

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

Socket clienteConectado = servidor.accept();

System.***out***.println("Cliente conectado.");

clienteConectado.close();

System.***out***.println("Cliente desconectado.");

servidor.close();

} **catch** (Exception e) { System.***err***.println("ERROR=> " + e.toString()); }

}// Fin de main

}//Fin de Ejemplo3Servidor

**Archivo EJEMPLO3.JAVA**

**import** java.net.Socket;

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

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

String Host = "localhost";

**int** Puerto = 6000;

System.*setProperty* ("java.security.policy", System.*getProperty*("user.dir") + "\\src\\\_03FicherosPoliticasSockets\\Politica4.policy");

System.*setSecurityManager*(**new** SecurityManager());

**try** {

Socket Cliente = **new** Socket(Host, Puerto);

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

Cliente.close();

System.***out***.println("CLIENTE FINALIZADO.");

} **catch** (Exception e) { System.***err***.println("ERROR=> " + e.toString()); }

}// Fin de main

}//Fin de Ejemplo3

**Archivo POLITICA3.POLICY**

/\*Politica para el SERVIDOR\*/

grant codeBase "file:${user.dir}/bin/" {

permission java.net.SocketPermission "localhost", "listen, accept";

};

**Archivo POLITICA4.POLICY**

/\*Politica para el CLIENTE\*/

grant codeBase "file:${user.dir}/bin/" {

permission java.net.SocketPermission "localhost:6000", "connect";

};

El programa servidor escucha y acepta peticiones de clientes en el puerto 6000, cuando el cliente se conecta se visualiza un mensaje y al cerrarse también.

El programa cliente se conecta al servidor en el puerto 6000, visualiza unos mensajes y luego se desconecta.

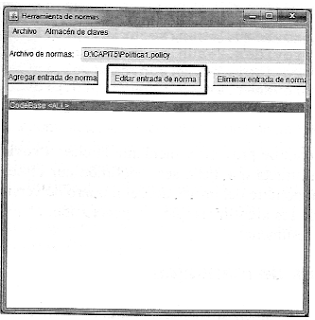
El fichero de políticas para el **servidor**, *Politica3.policy*, representa un permiso que permite a los programas localizados en la carpeta **${user.dir}/bin/** escuchar y aceptar conexiones en localhost.

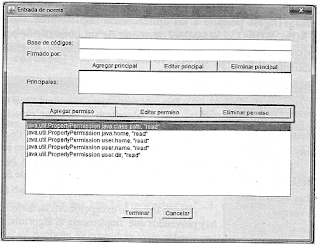
El fichero de políticas para el cliente, *Politica4.policy*, representa un permiso que permite a los programas localizados en la carpeta **${user.dir}/bin/** conectarse al puerto 6000 en localhost.

**Herramienta PolicyTool**

Se recomienda usar la herramienta para editar cualquier fichero de políticas y verificar la sintaxis de su contenido.

Para lanzarla, desde la terminal escribiremos el comando **policytool**.





## [**Criptografía con Java**](http://www.programandoapasitos.com/2016/03/procesos-y-servicios-programacion_25.html)

**Proveedores de servicios criptográficos**

El API **JCA** (*Java Cryptography Architecture*, incluye la extensión criptográfica de **Java JCE** – *Java Cryptography Extension*) incluída dentro del paquete JDK incluye dos componentes de software:

* El marco que define y apoya los servicios criptográficos para que los proveedores faciliten implementaciones. Este marco incluye paquetes como
  + java.security
  + javax.crypto
  + javax.crypto.spec
  + javax.crypto.interfaces
* Los proveedores reales, tales como Sun, SunRsaSign, SunJCE, que contienen las implementaciones criptográficas reales. El proveedore 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** localizo en la carpeta **java.home\lib\security**. Forman una lista de entradas con un número que indican 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=com.sun.net.ssl.internal.ssl.Provider
  + security.provider.4=com.sun.crypyo.provider.SunJCE
  + security.provider.5=sun.security.jgss.SunProvider

**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 proveedor específicas.

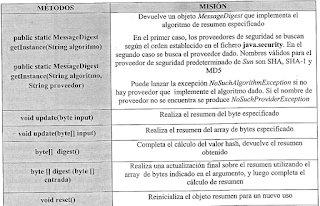
Tiene métodos para acceder a informaciones sobre las implementaciones de los algoritmos para la generación, conversión y gestión de claves y la generación de firmas y resúmenes, como el nombre del proveedor, el número de versión, etc.

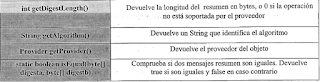
**Resúmenes de mensajes**

Un *message digest* o resúmen de mensajes (también se le conoce como **función hash**) es una marca digital de un bloque de datos.

La clase **MessageDigest** permite a las aplicaciones implementar algoritmos de resumen de mensajes, como **MD5**, **SHA-1** o **SHA-256**. Dispone de un constructor protegido, por lo que se accede a él mediante el método ***getInstance(String algoritmo)***.

Algunos métodos de la clase **MessageDigest** son:





**Ejemplo**

**Archivo EJEMPLO4.JAVA**

**import** java.security.MessageDigest;

**import** java.security.NoSuchAlgorithmException;

**import** java.security.Provider;

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

**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 INTRQDUCE TEXTO EN BYTES A RESUMIR

**byte** resumen[] = md.digest();//SE CALCULA EL RESUMEN

//PARA CREAR UN RESUMEN CIFRADO CON CLAVE

//String clave="clave de cifrado";

//byte resumen[] = md.digest(clave.getBytes()); // SE CALCULA EL RESUMEN CIFRADO

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) { e.printStackTrace(); }

}//Fin de 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;

}//Fin de for

**return** hex.toUpperCase();

}// Fin de Hexadecimal

}//Fin de Ejemplo4

Genera el resúmen de un texto plano. Con el método MessageDigest.getInstance(“SHA”) se obtiene una instancia del algoritmo SHA. El texto plano lo pasamos a un array de bytes y el array se pasa como argumento al método upate(), finalmente con 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 y por último información del proveedor.

Se puede crear un resúmen cifrado con clave usando el segundo método digest(bytes[]), donde se proporciona la clave en un array de bytes.

String clave="clave de cifrado";

**byte** dataBytes[] = texto.getBytes(); //TEXTO A BYTES

md.update(dataBytes) ; // SE INTRODUCE TEXTO EN BYTES A RESUMIR

**byte** resumen[] = md.digest(clave.getBytes()); // SE CALCULA EL RESUMEN CIFRADO

**Segundo ejemplo**

**Archivo EJEMPLO5.JAVA**

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectOutputStream;

**import** java.security.MessageDigest;

**import** java.security.NoSuchAlgorithmException;

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

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

**try** {

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

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

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

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

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

+ "que vivía un hidalgo de los de lanza en astillero, "

+ "adarga antigua, rocín flaco y galgo corredor.";

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

md.update(dataBytes) ;// TEXTQ 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** (IOException e) { e.printStackTrace();

} **catch** (NoSuchAlgorithmException e) { e.printStackTrace(); }

}//Fin de main

}//Fin de Ejemplo5

Guardaremos un mensaje en un fichero.

También guardaremos en el fichero el resumen del mensaje, para asegurarnos de que a la hora de leer el mensaje el fichero **no esté dañado o no haya sido manipuado y los datos sean los correctos**.

**Tercer ejemplo**

**Archivo EJEMPLO6.JAVA**

**import** java.io.FileInputStream;

**import** java.io.ObjectInputStream;

**import** java.security.MessageDigest;

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

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

**try** {

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

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

Object o = dataOS.readObject();

// Primera lectura, se obtiene el String

String datos = (String) o;

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

// Segunda lectura, se obtiene el resumen

o = dataOS.readObject();

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

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

//Se calcula el resumen del String leído del fichero

md.update(datos.getBytes());// TEXTO A RESUMIR

**byte** resumenActual[] = md.digest(); // SE CALCULA EL RESUMEN

//Se comprueban lo dos resúmenes

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

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

**else**

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

dataOS.close();

fileout.close();

}**catch** (Exception e) { e.printStackTrace(); }

}//Fin de main

}//Fin de Ejemplo6

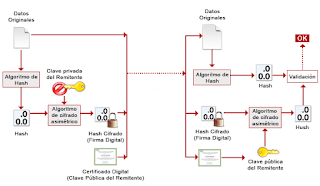
Al recuperar los datos del fichero primero necesitamos leer el String y luego el resumen, a continuación hemos de calcular de nuevo el resumen con el String leído y comparar este resúmen con el leído del fichero.

**Generando y verificando firmas digitales**

El resumen de un mensaje no nos da un alto nivel de seguridad.

Se puede decir que el fichero no es correcto si el texto que se lee no produce la misma salida que el resúmen guardado.

Pero alguien puede cambiar el texto y el resumen, y no podemos estar seguros de que el texto sea el que debería ser.

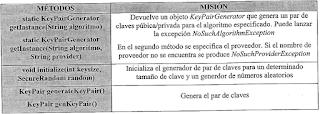


**Clase KeyPairGenerator**

En algunos casos, el par de claves (**clave pública** y **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 KeyParGenerator nos permite generar el par de claves. Dispone de un constructor protegido, por lo que se accede a él mediante el método **getInstance(String algoritmo)**.



**Clase KeyPair**

Es una clase soporte para generar las claves pública y privada.

Dispone de dos métodos:



**PrivateKey** y **PublicKey** son interfaces que agrupan todas las interfaces de clave privada y pública respectivamente.

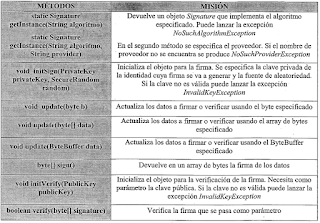
**Clase Signature**

Se usa para firmar los datos.

Un objeto de esta clase se puede utilizar para generar y verificar firmas digitales.

Dispone de un constructor protegido y se acceder a él mediante el método **getInstance(String algoritmo)**.

Algunos de sus métodos:



Al especifiar el nombre del algoritmo de firma, también se debe incluir el nombre del algoritmo de resumen de mensajes utilizado por el algoritmo de firma. **SHA1withDSA** es una forma de especificar el algoritmo de firma DSA, usando el algoritmo de resumen SHA-1. **MD5withRSA** significa algoritmo de resumen MD5 con algortmo de firma RSA.

Existen tres fases en el uso de un objeto **Signature** ya sea para firmar o verificar los datos: inicialización (ya sea con clave pública **initVerify()** o clave privada **initSign()**), actualización (**update()**) y firma (**sign()**)o verificación (**verify()**).

**Ejemplo**

**Archivo EJEMPLO7.JAVA**

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

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

**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();

//FIRMA CON CLAVE PRIVADA EL MENSAJE

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

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

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 CLAVE PÚBLICA EL MENSAJE FIRMADO

//AL OBJETO signature SE LE SUMINIST. LOS DATOS A VERIFICAR

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

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 e1) { e1.printStackTrace();

} **catch** (InvalidKeyException e) { e.printStackTrace();

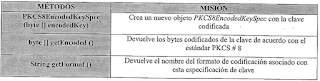
} **catch** (SignatureException e) { e.printStackTrace(); }

}//Fin de main

}//Fin de Ejemplo7

**Almacenar las claves pública y privada en ficheros**

Para almacenar la clave privada en disco es necesario codificarla en formato PKCS8 usando la clase **PKCS8EncodedKeySpec**.



PKCS8EncodedKeySpec pk8Spec =

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

//Escribir a fichero binario la clave privada

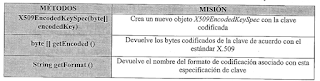
FileOutputStream outpriv =

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

outpriv.write(pk8Spec.getEncoded());

outpriv.close();

Para almacenar la clave pública en disco es necesario codificarla en formato X.509 usando la clase **X509EncodedKeySpec.**

****

X509EncodedKeySpec pkX509 =

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

//Escribir a fichero binario la clave pública

FileOutputStream outpub =

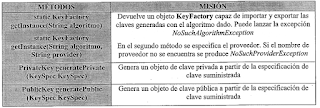
**new** FileOutputStream("Clave.publica");

outpub.write(pkX509.getEncoded());

outpub.close();

**Recuperar las claves pública y privada de ficheros**

Clase **KeyFactory**. Para recuperar las claves de los ficheros que proporciona métodos para convertir claves de formato criptográfico (PKCS8, X.509) a especificaciones de claves y viceversa. Su constructor y alguno de sus métodos:



**Firmar los datos de un fichero con la clave privada**

**Archivo EJEMPLO8.JAVA**

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

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

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

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

**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();

//RECUPERA CLAVE PRIVADA DESDE DATOS CODIFICADOS EN FORMATO PKCS8

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

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

PrivateKey clavePrivada = keyDSA.generatePrivate(clavePrivadaSpec);

//INICIALIZA FIRMA CON CLAVE PRIVADA

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

dsa.initSign (clavePrivada);

//LECTURA DEL FICHERO A FIRMAR

//Se suministra al objeto Signature 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();

//GENERA LA FIRMA DE LOS DATOS DEL FICHERO

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

// GUARDA LA FIRMA EN OTRO FICHERO

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

fos.write(firma);

fos.close();

} **catch** (Exception e1) { e1.printStackTrace(); }

}//Fin de main

}//Fin de Ejemplo8

Genera la firma del fichero *DATOS.DAT* a partir de la clave privada almacenada en el fichero *Clave.privada*. La firma se almacenará en el fichero *DATOS.FIRMA*.

**Verificar la firma de un fichero con la clave pública**

**Archivo EJEMPLO9.JAVA**

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

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

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

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

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

**try** {

//LECTURA DE LA CLAVE PUBLICA DEL FICHERO

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

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

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

inpub.close();

//RECUPERA CLAVE PUBLICA DESDE DATOS CODIFICADOS EN FORMATO X509

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

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

PublicKey clavePublica = keyDSA.generatePublic(clavePublicaSpec);

//LECTURA DEL FICHERO QUE CONTIENE LA FIRMA

FileInputStream firmafic = **new** FileInputStream("FICHERO.FIRMA");

**byte**[] firma = **new** **byte**[firmafic.available()];

firmafic.read(firma); firmafic.close();

//INICIALIZA EL OBJETO Signature CON CLAVE PÚBLICA PARA VERIFICAR

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

dsa.initVerify (clavePublica);

//LECTURA DEL FICHERO QUE CONTIENE LOS DATOS A VERIFICAR

//Se suministra al objeto Signature los datos a verificar

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();

//VERIFICAR LA FIRMA DE LOS DATOS LEIDOS

**boolean** verifica = dsa.verify(firma);

//COMPROBAR LA VERIFICACIÓN

**if** (verifica)

System.***out***.println("LOS DATOS SE CORRESPONDEN CON SU FIRMA.");

**else**

System.***out***.println("LOS DATOS NO SE CORRESPONDEN CON SU FIRMA");

} **catch** (Exception el) { el.printStackTrace(); }

}//Fin de main

}//Fin de Ejemplo9

Necesitamos la clave púlica almacenada en el fichero *Clave.publica*, la firma del fichero almacenada en *DATOS.FIRMA* y el fichero de datos *DATOS.DAT*. En primer lugar obtendremos la clave pública del fichero *Clave.publica*, a continuación obtenemos la firma digital almacenada en el fichero *DATOS.FIRMA*. A continuación se leen los datos del fichero de datos *DATOS.DAT* y se suministran al objeto **Signature**. Por último se verifica la firma con la clave pública.

**Herramientas para firmar ficheros**

Java dispone de la herramienta de línea de comandos **keytool** para generar y manipular certificados.

Para firmar un documento seguiremos los siguientes pasos:

**1. Crear un fichero JAR que contiene el documento a firmar.**

jar cvf Contrato.jar Contrato.pdf

**2. Generar las claves púlica y privada (si no existen), con keytool-genkey.**

keytool –genkey –alias FirmaContrato –keystore AlmacenClaves

Creamos un almacén de claves (**keystore**) con el nombre *AlmacenClaves*. *FirmaContrato* es el nombre con el que haremos referencia al par de claves creado.

Nos pedirá contraseña para el almacén de claves y para la clave privada del par de claves generado.

El certificado generado tiene una validez de 90 días a no ser que se especifique la opción **-validity** en **keytool**.

Los certificados autofirmados son útiles para desarrollar y probar una aplicación.

La aplicación está firmada con un certificado que no es de confianza, por tanto, al ejecutarla nos preguntará antes si queremos ejecutarla.

Se recomienda no importar en un almacén de claves un certificado en el que no se confíe plenamente.

**3. Firmar el fichero JAR, usando jarsigner y la clave privada.**

jarsigner –keystore AlmacenClaves –signedjar DocumentoFirmado.jar Contrato.jar FirmaContrato

**4. Con keytool –export exportar el certificado de clave pública para que el receptor autentique la firma del emisor.**

keytool –export –keystore AlmacenClaves –alias FirmaContrato –file MariaJesus.cer

**5. Por último suministrar el fichero JAR firmado y el certificado al receptor.**

El receptor necesita importar el certificado como un certificado de confianza

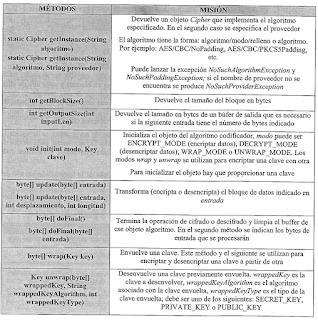
keytool –import –alias MJesus –file MariaJesus.cer –keystore AlmacenReceptor

y verificar la firma del fichero JAR.

jarsigner –verify –verbose –certs –keystore AlmacenReceptor DocumentoFirmado.jar

**Clase Cipher**

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



Como algoritmo en el método getInstance() se pueden poner los siguientes (**algoritmo/modo/relleno**), entre paréntesis se especifica el tamañi de la clave en bits:

AES/CBC/NoPadding (128)

AES/CBC/PKCS5Padding (128)

AES/ECB/NoPadding (128)

AES/ECB/PKCS5Padding (128)

DES/CBC/NoPadding (56)

DES/CBC/PKCS5Padding (56)

DES/ECB/NoPadding (56)

DES/ECB/PKCS5Padding (56)

DESede/CBC/NoPadding (168)

DESede/CBC/PKCS5Padding (168)

DESede/ECB/NoPadding (168)

DESede/ECB/PKCS5Padding (168)

RSA/ECB/PKCS1Padding (1024, 2048)

RSA/ECB/OAEPWithSHA-1AndMGF1Padding (1024, 2048)

RSA/ECB/OAEPWithSHA-256AndMGF1Padding (1024, 2048)

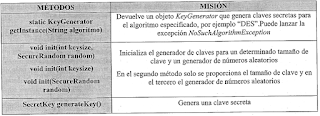
Los modos son la forma de trabajar del algoritmo

* **ECB** (Electronic Cookbook Mode). Los mensajes se dividen en bloques y cada uno de ellos es cifrado por separado por separado utilizando la misma clave K. A bloques de texto plano o claro idénticos les corresponden bloques idénticos de texto cifrado, de anera que se pueden reconocer estos patrones. De ahí que no sea recomendable.
* **CBC** (Cipher Block Chaining), a cada bloque de texto plano se le aplica la operación XOR con el bloque cifrado anterior antes de ser cifrado. De esta forma, cada bloque de texto cifrado depende de todo el texto en claro procesado hasta este punto. Para hacer cada men saje único se utilza asimismo un vector de inicialización.

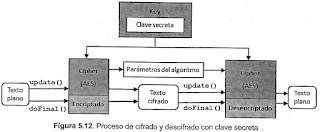
El relleno se utiliza cuando el mensaje a cifrar no es múltiplo de la longitud de cifrado del algoritmo, entonces es necesario indicar la forma de rellenar los últimos bloques.

**Clase KeyGenerator**

Proporciona funcionalidades para generar claves secretas para usarse en algoritmos simétricos. Algunos métodos son:



**Pasos para encriptar y desencriptar con clave secreta**



è Creamos la clave secreta usando **AES** o **DES.**

è Creamos un objeto **Cipher**con el **algoritmo/modo/relleno** que creamos oportuno, lo inicializamos en **modo encriptación** con la clave creada anteriormente.

è Realizamos el cifrado de la información con el método doFinal().

è Configuramos el objeto **Cipher** en modo desencriptación con la clave anterior para desencriptar el texto, usamos el método doFinal().

**Archivo EJEMPLO10.JAVA**

**import** java.security.InvalidAlgorithmParameterException;

**import** java.security.InvalidKeyException;

**import** java.security.Key;

**import** java.security.NoSuchAlgorithmException;

**import** javax.crypto.BadPaddingException;

**import** javax.crypto.Cipher;

**import** javax.crypto.IllegalBlockSizeException;

**import** javax.crypto.KeyGenerator;

**import** javax.crypto.NoSuchPaddingException;

**import** javax.crypto.SecretKey;

**import** javax.crypto.spec.IvParameterSpec;

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

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

**try** {

//Creamos la clave secreta usando el algoritmo AES y deﬁnimos un tamaño de clave de 128 bits

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

kg.init (128);

SecretKey clave = kg.generateKey();

//Creamos un objeto Cipher con el algoritmo AES/ECB/PKCS5Padding, lo inicializamos en modo encriptación con la clave creada anteriormente.

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

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

//Realizamos el cifrado de la información con el método doFinal()

**byte** textoPlano[] = "Esto es un Texto Plano".getBytes();

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

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

//Conﬁguramos el objeto Cipher en modo desencriptación con la clave anterior para desencriptar el texto, usamos el método doFinal()

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

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

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

/\*

//Muchos modos de algoritmo (por ejemplo CBC) requieren un vector de inicialización que se especifica cuando se inicializa

//el objeto Cipher en modo desencriptación. En estos casos, se debe pasar al método init() el vector de inicialización.

//La clase IvParameterSpec se usa para hacer esto en el cifrado DES.

KeyGenerator kg = KeyGenerator.getInstance("DES");

Cipher c = Cipher.getInstance("DES/CBC/PKCS5Padding");

Key clave = kg.generateKey();

//Devuelve el vector IV inicializado en un nuevo buffer

byte iv[]=c.getIV();

IvParameterSpec dps = new IvParameterSpec(iv);

c.init(Cipher.DECRYPT\_MODE, clave, dps);

\*/

} **catch** (NoSuchAlgorithmException e) {

} **catch** (NoSuchPaddingException e) {

} **catch** (InvalidKeyException e) {

} **catch** (IllegalBlockSizeException e) {

} **catch** (BadPaddingException e) {

// } catch (InvalidAlgorithmParameterException e) {

}

}// Fin de main

}// Fin de Ejemplo10

**Almacenar la clave secreta en un fichero**

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

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

**import** javax.crypto.\*;

**public** **class** AlmacenaClaveSecreta {

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

**try** {

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

kg.init(128);

//genera clave secreta

SecretKey clave = kg.generateKey();

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

out.writeObject(clave);

out.close();

/\*

//Para recuperar la clave secreta del fichero

ObjectInputStream in = new ObjectInputStream(new FileInputStream("Clave.secreta"));

Key secreta = (Key) in.readObject();

in.close();

\*/

} **catch** (NoSuchAlgorithmException e) {e.printStackTrace();

} **catch** (FileNotFoundException e) {e.printStackTrace();

//} catch (ClassNotFoundException e) { e.printStackTrace();//Para recuperar la clave secreta

} **catch** (IOException e) {e.printStackTrace();}

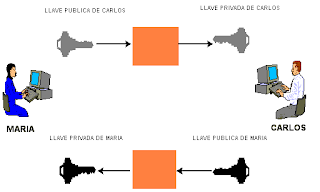
}//Fin de main

}//Fin de AlmacenaClaveSecreta

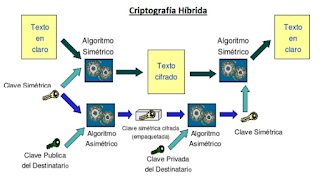
Genera una clave secreta **AES** y la almacena en el fichero **Clave.secreta**.

**Encriptar y desencriptar con clave pública**

**Conversación encriptada**



**Criptografía híbrida**

****

**Archivo EJEMPLO12.JAVA**

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

**import** javax.crypto.\*;

**public** **class** Ejemplo12 {

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

**try** {

//SE CREA EL PAR DE CLAVES PÚBLICA Y PRIVADA

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

keyGen.initialize (1024);

KeyPair par = keyGen.generateKeyPair();

PrivateKey clavepriv = par.getPrivate();

PublicKey clavepub = par.getPublic();

//SE CREA LA CLAVE SECRETA AES

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

kg.init (128);

SecretKey clavesecreta = kg.generateKey();

//SE ENCRIPTA LA CLAVE SECRETA CON LA CLAVE RSA PÚBLICA

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

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

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

//CIFRAMOS 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 necesitamos desencriptar la clave Secreta con la clave privada y a continuación desencriptar el texto con esa clave; usaremos el método unwrap():\*/

//SE DESENCRIPTA LA CLAVE SECRETA CON LA CLAVE RSA PRIVADA

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

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

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

//DESCIFRAMOS EL TEXTO CON LA CLAVE DESENVUELTA

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** (Exception e) { e.printStackTrace(); }

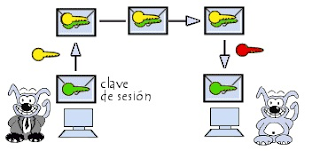
}//Fin de main

}//Fin de Ejemplo12

* Se genera un par de claves pública y privada con el algoritmo RSA.
* Se crea una clave secreta con el algoritmo AES.
* Esta clave se creará para encriptar el texto.
* La clave secreta es encriptada mediante la clave pública utilizando el método wrap()
* Para desencriptar el texto primero necesitamos desencriptar la clave Secreta con la clave priada y a continuación desencriptar el texto con esa clave; usaremos el método unwrap()

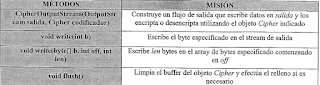
**Clave de sesión**

Es un término medio entre el cifrado simétrico y asimétrico que permite combinar las dos técnicas. Consiste en generar una clave de sesión K y cifrarla usando la clave pública del receptor. El receptor descifra la clave de sesión usando su clave privada. El emisor y el receptor comparten una clave que solo ellos conocen y pueden cifrar sus comunicaciones usando la misma clave de sesión.

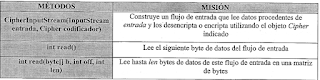


**Encriptar y desencriptar flujos de datos**

**Clase CipherOutputStream.** Encriptar datos hacia un fichero



**CipherInputStream.** Leer y desencriptar datos de un fichero



Ambas manipular de forma transparente las llamadas a update() y doFinal()

**Archivo EJEMPLO13CIFRA.JAVA**

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

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

**import** javax.crypto.\*;

**public** **class** Ejemplo13Cifra {

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

**try** {

//RECUPERAMOS CLAVE SECRETA DEL FICHERO

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

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

oin.close();

//SE DEFINE 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

//LEEMOS BLOQUES DE BYTES DEL FICHERO PDF

//Y int LO VAMOS ESCRIBIENDO AL 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 clave secreta.");

} **catch** (Exception e) {e.printStackTrace();}

}//Fin de main

}// Fin de Ejemplo13Cifra

Utiliza la clave secreta almacenada en un fichero llamado para cifrar un documento PDF de nombre *Fichero.pdf*.

**Archivo EJEMPLO13DESCIFRA.JAVA**

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

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

**import** javax.crypto.\*;

**public** **class** Ejemplo13Descifra {

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

**try** {

//RECUPERAMOS CLAVE SECRETA DEL FICHERO

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

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

oin.close();

//SE DEFINE EL OBJETO Cipher para desencriptar

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

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

//OBJETO CipherInputStream CUYO CONTENIDO SE VA A DESCIFRAR

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

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

**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 desencriptados al FileOutputStream

**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 clave secreta.");

} **catch** (Exception e) {e.printStackTrace();}

}//Fin de main

}//Fin de Ejemplo13Descifra

Utiliza la clase **CipherInputStream** para leer y desencriptar datos de un fichero cifrado.

## [**Comunicación segura con Java**](http://www.programandoapasitos.com/2016/03/procesos-y-servicios-programacion_26.html)

**JSSE**

**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 **TSL** e incluye funcionalidad de encriptación de 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, tales como HTTP, Telnet o FTP, a través de TCP/IP.

Las clases de JSSE se encuentran en los paquetes **javax.net** y **javax.net.ssl**.

**SSL**

Las clas **SSLSocket** y **SSLServerSocket** representan sockets seguros y son derivadas de las ya familiares **Socket**  y **ServerSocket** respectivamente.

**JSSE** tiene dos clases **SSLServerSocketFactory** y **SSLSocketFactory** para la creación de sockets seguros. No tienen constructor, se obtienen a través del método estatico getDefault().

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 crear un **SSLSocket:**

SSLSocketFactory sfact = (SSLSocketFactory)

SSLSocketFactory.*getDefault*();

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

**Archivo SERVIDORSSL.JAVA**

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

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

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

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

//System.setProperty("javax.net.ssl.keyStore", System.getProperty("user.dir") + "\\AlmacenSSL");

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

**int** puerto = 6000;

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

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

SSLSocket clienteConectado = **null**;

DataInputStream flujoEntrada = **null**; //FLUJO DE ENTRADA DE CLIENTE

DataOutputStream flujoSalida = **null**; //FLUJO DE SALIDA AL CLIENTE

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

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

clienteConectado = (SSLSocket) servidorSSL.accept();

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

// EL CLIENTE ME ENVIA UN MENSAJE

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

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

// ENVIO UN SALUDO AL CLIENTE

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

}// Fin de for

// CERRAR STREAMS Y SOCKETS

flujoEntrada.close();

flujoSalida.close();

clienteConectado.close();

servidorSSL.close();

}// Fin de main

}// Fin de ServidorSSL

Crea una conexión sobre un socket servidor seguro y que atenderá hasta cuatro conexiones de clientes que se identificarán con un certificado válido. El servidor espera las conexiones, de cada cliente que se conecta recibe un mensaje y a continuación le envía un saludo.

**Archivo CLIENTESSL.JAVA**

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

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

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

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

//System.setProperty("javax.net.ssl.trustStore", System.getProperty("user.dir") + "\\UsuarioAlmacenSSL");

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

String Host = "localhost";

**int** puerto = 6000;

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

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

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

// CREO FLUJO DE SALIDA AL SERVIDOR

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

// ENVIO UN SALUDO AL SERVIDOR

flujoSalida.writeUTF("Saludos al SERVIDOR DESDE EL CLIENTE");

// CREO 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());

/\*------------------------------------------------------------------------------

//Información sobre la sesión SSL

SSLSession session = ((SSLSocket) Cliente).getSession();

System.out.println("Host: "+session.getPeerHost());

System.out.println("Cifrado: " + session.getCipherSuite());

System.out.println("Protocolo: " + session.getProtocol());

System.out.println("IDentificador:" + new BigInteger(session.getId()));

System.out.println("Creación de la sesión: " + session.getCreationTime());

X509Certificate certificate = (X509Certificate)session.getPeerCertificates()[0];

System.out.println("Propietario: " + certificate.getSubjectDN());

System.out.println("Algoritmo: " + certificate.getSigAlgName());

System.out.println("Tipo: " + certificate.getType());

System.out.println("Emisor: " + certificate.getIssuerDN());

System.out.println("Número Serie: " + certificate.getSerialNumber());

------------------------------------------------------------------------------\*/

// CERRAR STREAMS Y SOCKETS

flujoEntrada.close();

flujoSalida.close();

Cliente.close();

}// Fin de main

}// Fin de ClienteSSL

Envía un mensaje al servidor y visualiza el que el servidor le devuelve.

El servidor necesita disponer de un certificado que mostrar a los clientes que se conecten a él. Usaremos la herramienta **keytool** para crearlo, en el ejemplo le damos el nombre de *AlmacenSSL* y el valor de la clave es *1234567*.

C:>keytool -genkey -alias claveSSL -keyalg RSA –keystore AlmacenSSL -storepass 1234567

Para ejecutar el programa servidor es necesario indicar el certificado que se utilizará

C:>java -Djavax.net.ssl.keyStore=AlmacenSSL -Djavax.net.ssl.keyStorePassword=1234567 ServidorSSL

Antes de ejecutar el programa cliente necesitamos colocar el certificado en el keystore del usuario, para ello lo exportamos a un fichero, le llamamos por ejemplo *CertificadoSSL.cer*

C:>keytool -export -alias claveSSL -keystore AlmacenSSL -storepass 1234567 -file Certificado.cer

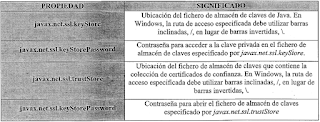
Una vez que tenemos el fichero exportado es necesario incorporarle al nuevo almacenamiento para permitir realizar la validación. A continuación se crea un keystore de nombre *UsuarioAlmacenSSL* con la clave *890123* y se incorpora el fichero de certificado *Certificado.cer*. Esto lo hacemos donde ejecutemos el cliente.

C:>keytool -import -alias claveSSL -file Certificado.cer –keystore UsuarioAlmacenSSL -storepass 890123

Para ejecutar el programa cliente escribimos lo siguientes

C:>java -Djavax.net.ssl.trustStore=UsuarioAlmacenSSL -Djavax.net.ssl.trustStorePassword=890123 ClienteSSL

En estos ejemplos para ejecutar el programa cliente y el servidor hemos establecido las **propiedades JSSE** desde la línea de comandos usando la sintaxis **–Dpropiedad=Valor**. También se pueden establecer desde el programa usando el método System.setProperty(String propiedad, String valor).



En el programa servidor incluimos las siguientes líneas después de definir la variable puerto:

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

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

Y en el programa cliente:

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

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

Opcionalmente podemos registrar un proveedor SSL en los programas añadiendo esta línea:

java.security.Security.addProvider(**new** com.sun.net.ssl.internal.ssl.Provider());

El método getSession() de la clase **SSLSocket** devuelve un objeto **SSLSession** con la sesión SSL utilizada por la conexión, a partir de ella podemos obtener información como el identificador de la sesión, el cifrado SSL, el protocolo, el certificado, etc.

SSLSession session = ((SSLSocket) Cliente).getSession();

System.out.println("Host: "+session.getPeerHost());

System.out.println("Cifrado: " + session.getCipherSuite());

System.out.println("Protocolo: " + session.getProtocol());

System.out.println("IDentificador:" + **new** BigInteger(session.getId()));

System.out.println("Creación de la sesión: " + session.getCreationTime());

X509Certificate certificate = (X509Certificate)session.getPeerCertificates()[0];

System.out.println("Propietario: " + certificate.getSubjectDN());

System.out.println("Algoritmo: " + certificate.getSigAlgName());

System.out.println("Tipo: " + certificate.getType());

System.out.println("Emisor: " + certificate.getIssuerDN());

System.out.println("Número Serie: " + certificate.getSerialNumber());

En el programa servidor para obtener la información del certificado tendríamos que utilizar el método getLocalCertificates() en lugar de getPeerCertificates().

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

En estos procesos están involucradas las clases e interfaces:

* **LoginContext**, contexto de inicio de sesión: clase para iniciar y gestionar el proceso de autenticación mediante la creación de un **Subject**. La autenticación se hace llamando al método login().
* **LoginModulo**, módulo de conexión: interfaz para definir los mecanismos de autenticación. Se deben implementar los siguientes métodos: iniatilize(), login(), commit(), abort() y logout(). Se encarga de validar los datos en un proceso de autenticación.
* **Subject**, clase para representar a un ente autenticable (entidad, usuario, sistema)
* **Principal,** clase que representa los atributos que posee cada **Subject** recuperado una vez que se efectúa el ingreso a la aplicación. Un **Subject** puede contener varios principales.
* **CallBackHandler,** encargado de la interacción con el usuario para obtener los datos necesarios para la autenticación. Se debe implementar el método handle().

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

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

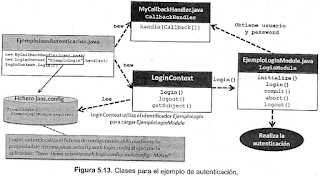
**Autenticación**

El proceso básico de autenticación consta de los siguientes pasos

* Creación de una instancia de **LoginContext**, uno o más **LoginModulo** son cargados basándose en el archivo de configuración de JAAS.
* La instanciación 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 este tratará de autenticarse.
* Invocación del método login() del **LoginContext** el cual invocará el método login() del **LoginModule**.
* Los datos del usuario se obtienen por medio del **CallbackHandler**.
* El **LoginModule** comprueba los datos introducidos por el usuario y los valida. Si la validación tiene éxito el usuario queda autenticado.

**Ejemplo**

Esquema básico de las clases y ficheros que se van a utilizar en el ejemplo para probar la autenticación básica con JAAS



Los ficheros son los siguientes:

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

EjemploLogin{

EjemploLoginModule required;

}

* Fichero de autenticación JAAS, **policy.config**. Se conceden 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";

};

**Archivo MYCALLBACKHANDLER.JAVA**

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

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

**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;

}

//método handle sera invocado por el LoginMbdule

**public** **void** handle(Callback[] callbacks) **throws** IOException, UnsupportedCallbackException {

**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());

}// fin del if

}// fin del for

}// Fin de handle

}// Fin de handle MyCallbackHandler

**CallbackHandler** tiene el método handle() que será invocado por el **LoginModule** al que le pasará un array de objetos **Callbacks**, que contiene los campos de datos que se necesitan. Cada **Callback** representa uno de los datos comunicados por el usuarios en el proceso de autenticación (nombre, password, etc), es necesario recorrer este array para recuperar los datos.



**Archivo EJEMPLOLOGINMODULE.JAVA**

**import** java.util.Map;

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

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

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

**import** javax.security.auth.spi.LoginModule;

**public** **class** EjemploLoginModule **implements** LoginModule {

**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 prompt

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

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

//La autenticación se realiza aquí

//el nombre de usuario: maria, su clave: 1234

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

} **catch** (Exception e) {e.printStackTrace();}

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

}//Fin de login

}//Fin de EjemploLoginModule

Implementa el **LoginModule** que autenticará a los usuarios en su método login(). Debe implementar los siguientes métodos:

* initialize(): El propósito de este método es inicializar el **LoginModule** con la información con la información relevante. El **Subject**pasasdo a este método se usa para almacenar los principales (**Principal**) y credenciales si la conexión tiene éxito. Recibe un **CallbackHandler** que puede ser usado para introducir información de la autenticación.
* login(): El propósito de este método es autenticar al **Subject**.
* commit(): Se llama a este método si tiene éxito la autenticación total del **LoginContext**.
* abort(): Informa al **LoginModule** de que algún proveedor o módulo ha fallado al autenticar al **Subject**. Se llama a este método si la autenticación global del **LoginContext** ha fallado.
* logout(): Desconecta al **Subject** borrando los principales y credenciales del **Subject**. Finalizará la sesión del usuario.

C:>java -Dusuario=maria -Dclave=1234 EjemploJAASAutenticacion

**Autorización**

Para que la autorización JAAS tenga lugar, se requiere lo siguiente:

* El usuario debe **autenticarse**. Ya visto
* 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**.

Para lo cual insertaríamos dos nuevas clases al ejemplo anterior.

**Archivo EJEMPLOACCION.JAVA**

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

**import** java.security.PrivilegedAction;

**public** **class** EjemploAccion **implements** PrivilegedAction {

**public** Object run() {

File f = **new** File("fichero.txt");

**if** (f.exists()) {

System.***out***.println("EL FICHERO EXISTE ... ");

//Si existe se muestra su contenido

FileReader fic;

**try** {

fic = **new** FileReader(f);

**int** i;

System.***out***.println("Su contenido es: ");

**while** ((i = fic.read()) != -1)

System.***out***.print((**char**) i);

fic.close();

} **catch** (Exception e) { e.printStackTrace();}

}**else** {

//Si no existe se crea y se insertan datos

System.***out***.println("EL FICHERO NO EXISTE, LO CREO ... ");

**try** {

FileWriter fic = **new** FileWriter(f);

String cadena = "Esto es una linea de texto";

fic.append(cadena); fic.close();// cerrar fichero

System.***out***.println("Fichero creado con datos...");

} **catch** (IOException e) {

System.***out***.println("ERROR ==> " + e.getMessage());

}

}// fin de if

**return** **null**;

}// Fin de run

}// Fin de EjemploAccion

Esta clase implementa **PrivilegedAction**, contiene el método run() que es el código que se ejecutará una vez que el usuario ha sido autenticado, teniendo en cuenta las autorizaciones de los principales definidas en el fichero de políticas.

**Archivo EJEMPLOPRINCIPAL.JAVA**

**import** java.security.Principal;

**public** **class** EjemploPrincipal **implements** Principal, java.io.Serializable {

**private** String name;//nombre del principal

//Crea un EjemploPrincipal con el nombre suministrado.

**public** EjemploPrincipal(String name) {

**if** (name == **null**)

**throw** **new** NullPointerException("Entrada nula");

**this**.name = name;

}

//Devuelve el nombre del Principal

**public** String getName() {**return** name;}

//Compara el objeto especificado con el Principal

//para ver si son iguales.

**public** **boolean** equals(Object o) {

**if** (o == **null**) **return** **false**;

**if** (**this** == o) **return** **true**;

**if** (!(o **instanceof** EjemploPrincipal)) **return** **false**;

EjemploPrincipal that = (EjemploPrincipal) o;

**if** (**this**.getName().equals(that.getName())) **return** **true**;

**return** **false**;

}//Fin de equals

**public** **int** hashCode() {**return** name.hashCode();}

**public** String toString() {**return** (name);}

}//Fin de EjemploPrincipal

Esta clase implementa la interface **Principal**, se usa en la clase **EjemploLoginModule** una vez que el usuario ha sido autenticado en el método commit().

**Privilegios de acceso en el fichero de políticas (policy.config)**

grant {

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

permission javax.security.auth.AuthPermission "modifyPrincipals";

permission javax.security.auth.AuthPermission "doAsPrivileged";

};

grant Principal EjemploPrincipal "maria" {

permission java.io.FilePermission "fichero.txt", "read";

} ;

grant Principal EjemploPrincipal "juan" {

permission java.io.FilePermission "fichero.txt", "write";

};

