Contenido

[5.1. Criptografía y Sistemas Criptográficos. 2](#_Toc503725279)

[5.2. Criptografía simétrica. 4](#_Toc503725280)

[5.3. Criptografía asimétrica o de clave pública. 6](#_Toc503725281)

[5.4. Comparación entre criptografía simétrica y asimétrica. 8](#_Toc503725282)

[5.5. JCA: Java Cryptographics Architecture. 9](#_Toc503725283)

[5.5.1.- Resúmenes de mensajes. 10](#_Toc503725284)

[5.5.2.- Generando y verificando firmas digitales. 10](#_Toc503725285)

[5.5.3.- Cifrado simétrico. 13](#_Toc503725286)

[5.5.4.- Cifrado asimétrico. 15](#_Toc503725287)

[5.6. Sockets seguros vía SSL. 17](#_Toc503725288)

[Bibliografía Web 21](#_Toc503725289)

# 5.1. Criptografía y Sistemas Criptográficos.

La **criptografía** es la práctica y el estudio de la ocultación de información. En otras palabras, es el arte o ciencia de cifrar y descifrar cierta información mediante técnicas especiales y se emplea frecuentemente para permitir un intercambio de mensajes que sólo puedan ser leídos por personas a las que van dirigidos y que poseen los medios para descifrarlos.

La criptografía que nosotros conocemos y utilizamos, data de los años 70 aunque ya se utilizaban técnicas criptográficas en la segunda guerra mundial e incluso pueblos de la antigüedad como los griegos y romanos utilizaban mecanismos para enviar mensajes cifrados en sus campañas militares.

Existen muchos ámbitos en los que hoy en día la criptografía aporta seguridad en la comunicación por canales no seguros.

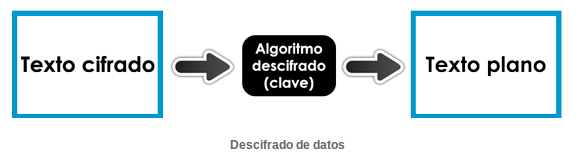
Si por ejemplo hablamos de **teléfonos móviles**, se usan cada vez más sistemas que permiten la firma y verificación local de documentos a través de los propios dispositivos y de acuerdo a estándares de clave pública. La consolidación de estándares de firma digital y en particular los relativos a infraestructura **PKI** (Public Key Infrastructure) ha permitido que complicadas soluciones de firma digital utilizadas casi exclusivamente en entornos de Administración Pública se hayan simplificado y generalizado permitiendo a las empresas acceder a ellas e incorporarlas en sus flujos de relación con clientes, empleados, suministradores, proveedores, etc.

Infinidad de usos y finalidades se aplican cada día en el mundo de la tecnología, ya sea en comunicaciones militares, protocolo https, envío de correos electrónicos cifrados o las telecomunicaciones en general.

La información original que debe protegerse se denomina **texto plano** o **texto en claro**. El cifrado es el proceso de convertir el texto plano en un texto imposible de leer llamado texto cifrado. Para obtener un **texto cifrado**, se aplica un algoritmo de cifrado, utilizando una clave, al texto plano.



De la misma manera, si aplicamos un algoritmo de descifrado, que también utiliza una clave, al texto cifrado, obtendremos, de nuevo, el texto plano.



La **clave debe ser secreta** y sólo deben conocerla las dos partes implicadas en ambos procesos. Si alguien más supiera la clave, se vería comprometida la seguridad del cifrado de los datos.

Dependiendo del número de claves utilizadas por los algoritmos de cifrado/descifrado, existen dos tipos de métodos criptográficos: criptografía de clave simétrica, que utiliza una única clave, y criptografía de clave asimétrica que utiliza dos claves.

# 5.2. Criptografía simétrica.

**En la criptografía simétrica se usa una única clave para cifrar y descifrar mensajes.** Las dos partes que se comunican han de ponerse de acuerdo de antemano sobre la clave a usar. Una vez compartida la clave, el remitente cifra un mensaje con ella, lo envía al destinatario y éste lo descifra con la misma.

En las siguientes imágenes se puede observar el funcionamiento del intercambio de claves y de cifrado simétrico:

El emisor envía al receptor la clave con la que cifrar y descifrar los mensajes a través de un canal seguro.



El emisor envía el mensaje cifrado al receptor. Éste último descifra el mensaje para ver su contenido.



**Los algoritmos más utilizados para realizar el cifrado y descifrado de los datos son:** DES, Triple-DES (3DES), AES, Blowfish, etc.

Actualmente, los ordenadores pueden obtener claves desconocidas con extrema rapidez, y ésta es la razón por la cual el tamaño de la clave es importante en los sistemas criptográficos modernos. El algoritmo de cifrado DES usa una clave de 56 bits, lo que significa que hay 2 elevado a 56 claves posibles. Aunque esto representa un número muy alto de claves, un ordenador genérico puede comprobar el conjunto posible de claves en cuestión de días. Una máquina especializada puede hacerlo en horas. Algoritmos de cifrado de diseño más reciente como Triple-DES o Blowfish pueden usar claves de tamaños comprendidos entre los 128 bits y los 256 bits, lo que significa que existen 2 elevado a 128/256 claves posibles.

El **principal problema** con los sistemas de cifrado simétrico **está ligado al intercambio de claves.** Ningún canal de comunicación es lo bastante seguro como para garantizar que nadie interceptará el mensaje en el que se envía la clave.

**Otro problema es el número de claves que se necesitan.** Para que un número n de personas se comuniquen entre sí, son necesarias n-1 claves por cada persona, lo que implica que son necesarias n \* ((n - 1) / 2) claves en total. Por ejemplo, para un grupo de 4 personas serían necesarias 6 claves, pero para un grupo de 100 personas serían necesarias 4950 claves. Por lo tanto, esto puede funcionar con un grupo reducido de personas, pero sería inviable llevarlo a cabo con grupos más grandes.

# 5.3. Criptografía asimétrica o de clave pública.

**La criptografía asimétrica usa dos claves para el cifrado/descifrado de mensajes: una clave pública y una clave privada.**

**Cada usuario tiene una clave pública y una privada asociadas a él.** El usuario debe mantener en secreto la privada y distribuir la pública a todos los receptores con los que desea comunicarse.

Los métodos criptográficos garantizan que esa pareja de claves sólo se pueda generar una vez, de modo que se puede asumir que no es posible que dos personas hayan obtenido casualmente la misma pareja de claves.

En este sistema, lo que cifra la clave pública sólo puede ser descifrado con la privada y lo que cifra la clave privada sólo lo descifra la pública.

El procedimiento consiste en que el emisor cifra los datos con la clave pública del receptor, de esta forma se garantiza la confidencialidad del mensaje ya que sólo el receptor puede descifrarlo con su clave privada.

En las siguientes imágenes se puede observar el funcionamiento cifrado y descifrado en la criptografía asimétrica:

El emisor envía al receptor el mensaje cifrado con la clave pública del receptor. El receptor descifra el mensaje con su clave privada para así poder ver el mensaje en claro.



De igual manera, el receptor cifra el mensaje con la clave pública del receptor. El emisor descifra el mensaje con su clave privada.



La criptografía asimétrica es la base para realizar operaciones de autenticación y firma electrónica.

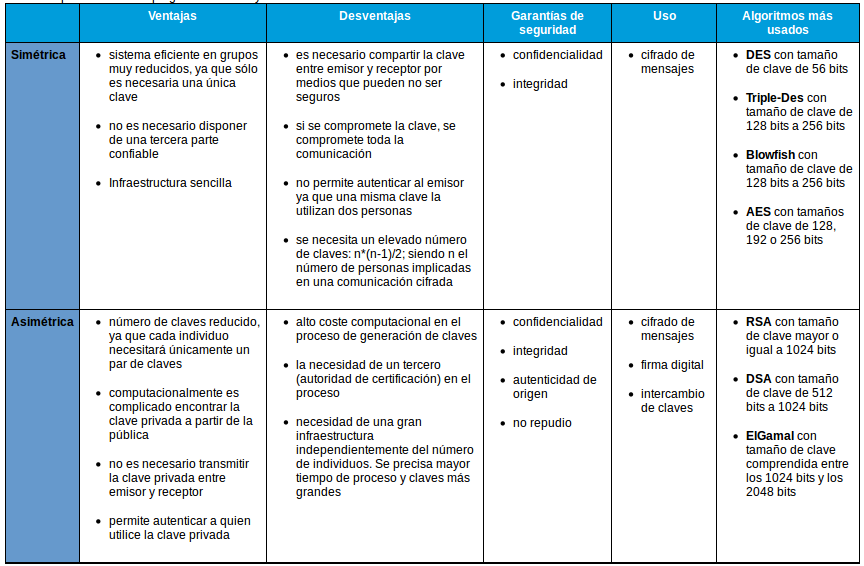
**Algunos de los algoritmos más utilizados son:** RSA, DSA, Diffie-Hellman y ElGamal.

**En este tipo de criptografía el tamaño de las claves es muy importante ya que toda la seguridad recae en la complejidad de las claves y en la imposibilidad de obtener la clave privada a partir de la pública. Por este motivo el tamaño de las claves es importante, cuanto más largas, más complejas, mayor dificultad. El tamaño de las claves a calcular puede variar entre los 512 bits y los 4096 bits.** **En la actualidad se considera como seguras las claves con un tamaño mínimo de 1024 bits.**

Como todos los sistemas, la criptografía asimétrica también tiene **desventajas**, como por ejemplo el tamaño de las claves y el tiempo de proceso. Para una misma longitud de clave y mensaje, se necesita mayor tiempo de proceso que para la criptografía simétrica porque las claves tienen un tamaño mayor que las utilizadas en un sistema de clave simétrica y por la complejidad de los algoritmos.

# 5.4. Comparación entre criptografía simétrica y asimétrica.

En la siguiente tabla, se muestra una comparativa entre los dos tipos de criptografía:



# 5.5. JCA: Java Cryptographics Architecture.

JCA es un Framework para criptografía que forma parte de la distribución estándar de la JVM (máquina virtual de Java). Reemplaza (y amplia) al API JCE (*Java Cryptographic Extensions*)

Ofrece un API (*application programming interface*) que permite:

* **generación de claves** (claves secretas y pares de claves pública y privada)
* **cifrado simétrico** (*DES, 3DES, IDEA*, etc)
* **cifrado asimétrico** (*RSA, DSA, Diffie-Hellman, ElGamal*...)
* **funciones de resumen** (*MD5* y *SHA1* ) y **algoritmos MAC** (*Message Authentication Code*)
* **generación y validación de firmas**
* **acuerdo de claves**

**Estructura general.**

Las implementaciones de los distintos algoritmos de cifrado, generación de claves, etc son ofertadas por paquetes externos denominados ***providers***.

* La distribución básica de Java incluye por defecto el *provider* ''SUN'' con implementaciones de los algoritmos más representativos.
* Otros fabricantes ofrecen *providers* adicionales que incluyen nuevos algoritmos o implementaciones alternativas de los ya existentes en el *provider* ''SUN''.

Así es posible dotar a JCA de nuevas funcionalidades sin necesidad de cambiar el API básica y permitir la distribución de algoritmos criptográficos con limitaciones de exportación.

Para usar las clases y métodos del API JCA las aplicaciones tienen que importar, como mínimo, los siguientes paquetes:

import java.security.\*;

import java.security.interfaces.\*;

import java.security.spec.\*;

import javax.crypto.\*;

import javax.crypto.interfaces.\*;

import javax.crypto.spec.\*;

## 5.5.1.- Resúmenes de mensajes.

A las funciones hash también se les llama funciones picadillo, funciones resumen o funciones de *digest*. Los hash o funciones de resumen son algoritmos que consiguen crear a partir de una entrada (ya sea un texto, una contraseña o un archivo, por ejemplo) una salida alfanumérica de longitud normalmente fija que representa un resumen de toda la información que se le ha dado (es decir, a partir de los datos de la entrada crea una cadena que solo puede volverse a crear con esos mismos datos).

Las funciones hash son útiles para almacenar *passwords* en bases de datos, verificar la integridad de ficheros y también se aplican en procesos de firma digital.

Para generar estas cadenas de resumen utilizaremos la clase ***MessageDigest* de Java que permite a las aplicaciones implementar algoritmos de resumen de mensajes, como MD5, SHA-1, o SHA-256. Algunos de sus métodos son:**

* Implementación de un algoritmo de resumen especificado.

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

* Introducir el texto a resumir.

byte datos[] = texto.getBytes(); //Texto en bytes

md.update(datos);

* Realizar el resumen.

byte resumenCalculado[] = md.digest();

* Comprobar dos resúmenes.

if (MessageDigest.isEqual(resumenOriginal, resumenCalculado)) {….

## 5.5.2.- Generando y verificando firmas digitales.

Los algoritmos de resumen no proporcionan un alto nivel de seguridad. Se puede cambiar el texto original y enviar un resumen acorde y no podremos estar seguros de que el texto sea el que debería ser.

Las firmas digitales pueden autenticar mensajes y asegurar que no ha sido alterado y procede del emisor correcto. Para crear una firma digital se precisa de una clave privada y la clave pública correspondiente con el fin de verificar la autenticidad de la firma.

En algunos casos, el par de claves (clave pública y 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** nos permite generar esas claves.

La clase **KeyPair** es una clase soporte para generar las claves pública y privada.

Los pasos para firmar/verificar datos es:

1.- Generar el par de claves.

//La clase KeyPairGenerator nos permite gernerar el par de claves.

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

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

keyGen.initialize(1024, numero);

//Creamos el par de claves (privada y pública).

KeyPair par = keyGen.generateKeyPair();

PrivateKey clavepriv = par.getPrivate();

PublicKey clavepubl = par.getPublic();

2.- Almacenar las claves y/o transmitirlas a sus respectivos destinatarios. Este punto lo veremos más adelante.

3.- Firmar los datos. Usaremos la clase ***Signature***. **Se firma con la clave privada y se verifica con la pública.** La firma es devuelta como un *array* de *bytes*.

//Firmamos con la clave privada el mensaje.

//Al especificar el nombre del algoritmo de firma se debe especificar, también,

//el nombre del algoritmo resumen utilizado por el algoritmo de firma.

//Tendremos dos:

// - SHAwithDSA --> firma con DSA resumen con SHA.

// - MD5withRSA --> firma con RSA resumen con MD5.

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

dsa.initSign(clavepriv);

String mensaje = “Texto a firmar”;

dsa.update(mensaje.getBytes());

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

3.- Verificar los datos. Usaremos la clase **Signature**. Se verificará con la pública.

//El receptor del mensaje, verifica con clave pública

//el mensaje firmado.

Signature verifica\_dsa = Signature.getInstance("SHA1withDSA");

verifica\_dsa.initVerify(clavepubl);

verifica\_dsa.update(mensaje.getBytes());

boolean check = verifica\_dsa.verify(firma);

**Para almacenar y recuperar las claves públicas y privadas en fichero.**

Guardar la clave privada: es necesario codificarla en **PKCS8** usando la clase **PKCS8EncodedKeySpec**.

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

//Escribir a fichero binario la clave privada.

FileOutputStream outpriv = new FileOutputStream(nom\_fichero);

outpriv.write(pk8Spec.getEncoded());

outpriv.close();

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

X509EncodedKeySpec pkX509 = new X509EncodedKeySpec(clavepubl.getEncoded());

//Escribir a fichero binario la clave pública.

FileOutputStream out = new FileOutputStream(nom\_fichero);

out.write(pkX509.getEncoded());

out.close();

Para recuperar las claves de los ficheros necesitamos la clase **KeyFactory** que proporciona métodos para convertir claves de formato criptográfico (PKCS8, X.509) a especificaciones de claves y viceversa.

Recuperar la clave privada:

//Leemos el fichero.

in = new FileInputStream(nom\_fichero);

BufferPriv = new byte[in.available()];//Definimos el buffer dle tamaño exacto

in.read(BufferPriv); //Leemos los bytes

in.close();

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

//Recuperamos la clave privada desde datos codificados en PKCS8

PKCS8EncodedKeySpec clavePrivadaSpec = new PKCS8EncodedKeySpec(BufferPriv);

cp = keyDSA.generatePrivate(clavePrivadaSpec);

Recuperar la clave pública:

//Leemos el fichero.

inpub = new FileInputStream(nom\_fichero);

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

inpub.read(bufferPub);

inpub.close();

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

//Recuperamos la clave pública desde datos codificados en X509.

X509EncodedKeySpec clavePublicaSpec = new X509EncodedKeySpec(bufferPub);

pk = keyDSA.generatePublic(clavePublicaSpec);

## 5.5.3.- Cifrado simétrico.

Hasta ahora hemos utilizado la API de seguridad de Java para realizar autenticación de mensajes, documentos y ficheros mediante firmas digitales. Pero la información ha sido visible en todo momento; cuando interesa que dicha información no pueda ser visible y se deba codificar es cuando entra en juego el **cifrado** o **encriptación**.

El uso de cifradores simétricos (y también asimétricos) se hace mediante instancias que heredan de la clase ***Cypher***.

**Creación de instancias.**

Mediante un método *factoría* (no con new), ya que será el *provider* quien cree el objeto concreto con su implementación del correspondiente algoritmo.

Se debe indicar el nombre (alias) del algoritmo de cifrado y, si son necesarios (en caso de cifradores de bloque), una especificación del modo de funcionamiento (ECB, CBC, ...) y del algoritmo de relleno. (Opcionalmente puede ser necesario indicar nombre del *provider*).

static Cipher getInstance(String transformation)

static Cipher getInstance(String transformation, String provider)

**Creación de la clave.**

Las claves se gestionan mediante objetos que implementan el interfaz SecretKey (que a su vez hereda del interfaz Key).

Las claves se crean empleando un objeto KeyGenerator.

* Creación del KeyGenerator empleando un método factoría (se debe indicar el alias del algoritmo de cifrado y, opcionalmente, el nombre del provider):

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

* Configuración del KeyGenerator (normalmente especificación del tamaño de clave):

generadorDES.init(56); // clave de 56 bits

* Creación de la clave:

SecretKey clave = generadorDES.generateKey();

**Cifrado y descifrado.**

* Se debe establecer el modo de funcionamiento del cifrador (método init(...)).
  + DECRYPT\_MODE : modo descifrado
  + ENCRYPT\_MODE : modo cifrado
  + PRIVATE\_KEY: clave para indicar el ''desempaquetado'' de una clave privada.
  + PUBLIC\_KEY: clave para indicar el ''desempaquetado'' de una clave pública.
  + SECRET\_KEY: clave para indicar el ''desempaquetado'' de una clave secreta (simétrica).
  + UNWRAP\_MODE: modo ''desempaquetado'' de claves (descifrado de claves).
  + WRAP\_MODE: modo ''empaquetado'' de claves (cifrado de claves).

cifrador.init(Cipher.ENCRYPT\_MODE, clave);

cifrador.init(Cipher.DECRYPT\_MODE, clave);

* Se debe ''alimentar'' al algoritmo con los datos a cifrar/descrifrar (mediante los métodos update()).

byte[] update(byte[] input)

byte[] update(byte[] input, int offset, int len)

Devuelve un array de byte con el resultado ''parcial'' del cifrado/descrifrado

* Se debe finalizar el cifrado/descrifrado invocando al método doFinal() que finaliza las operaciones (Importante: es necesario por si hay que manejar relleno).

byte[] doFinal()

byte[] doFinal(byte[] input)

byte[] doFinal(byte[] input, int inputOffset, int inputLen)

Devuelve un array de byte con el resultado del cifrado/descrifrado del último bloque.

**Ejemplos con clave simétrica:**

* Cifrar un mensaje String (ver código de ejemplo de clase).
* Cifrar el contenido de un fichero (ver código de ejemplo de clase).

## 5.5.4.- Cifrado asimétrico.

El funcionamiento de los cifradores asimétricos es idéntico al de los simétricos, mediante el uso de objetos que hereden de la clase ***Cypher***.

La única diferencia es el modo en que se **generan los pares de claves públicas y privadas** y el uso que se hace de ellas (en cifrado y/o descrifrado).

Necesitaremos descargar librerías criptográficas que proporcionen las funcionalidades necesarias para la generación de pares. Podemos utilizar las librerías proporcionadas por el proyecto Bouncy Castle.

**Provider Bouncy Castle.**

Bouncy Castle es un proyecto de software libre que pretende desarrollar una serie de librerías criptográficas libres y, entre otros, ofrece un provider para el JCA de java.

En la página web del proyecto (*http://www.bouncycastle.org/java.html*) es posible descargar la versión actual del *provider* ***Bouncy Castle*** para distintas versiones de la máquina virtual de Java. También incluye información resumida y el *javadoc* completo de la distribución.

**Instalación y uso de Provider Bouncy Castle.**

Para usar las clases de JCA/JCE ofrecidas por *BounceCastle*, basta con incluir en el código que lo utilice la siguiente orden de importación:

import org.bouncycastle.jce.provider.BouncyCastleProvider;

Dentro del código será necesario indicar la carga del *provider BouncyCastle* del siguiente modo:

Security.addProvider(new BouncyCastleProvider());

Este código se deberá situar antes de cualquier uso que se realice de este *provider*. La abreviatura que identifica a este *provider* es ''BC'' y deberá indicarse cuando se solicite alguna de las implementaciones de algoritmos que ofrece el *provider BouncyCastle*.

Cipher cifrador = Cipher.getInstance("DES/ECB/PKCS1Padding", "BC");

KeyGenerator keyGen = KeyGenerator.getInstance("AES", "BC");

**Creación de claves asimétricas.**

Las claves se gestionan mediante objetos que implementan el interfaz *PublicKey* y *PrivateKey* (que a su vez hereda del interfaz Key)

Las claves se crean empleando un objeto *KeyPairGenerator* específico para cada algoritmo asimétrico.

* Creación del ***KeyPairGenerator*** empleando un método factoría

Se debe indicar el alias del algoritmo de cifrado y, OBLIGATORIAMENTE, el nombre del *provider* [inicialmente el *provider* por defecto ''SUN'' no incluía RSA por limitaciones a la exportación de algoritmos de cifrado]).

Security.addProvider(new BouncyCastleProvider()); // Cargar el provider BC

...

KeyPairGenerator keyGenRSA = KeyPairGenerator.getInstance("RSA", "BC");

// Usa BouncyCastle

* Configuración del ***KeyPairGenerator*** (normalmente especificación del tamaño de clave).

gkeyGenRSA.initialize(512); // clave RSA de 512 bits

* Creación del par de claves (y recuperación de las claves pública y privada).

KeyPair clavesRSA = keyGenRSA.generateKeyPair();

PrivateKey clavePrivada = clavesRSA.getPrivate();

PublicKey clavePublica = clavesRSA.getPublic();

**Ejemplos con clave simétrica:**

* Cifrar un mensaje String (ver código de ejemplo de clase).
* Cifrar el contenido de un fichero (ver código de ejemplo de clase).

# 5.6. Sockets seguros vía SSL.

**SSL (*Secure Sockets Layer*).**

SSL proporciona autenticación y privacidad de la información entre extremos sobre Internet mediante el uso de criptografía. Habitualmente, sólo el servidor es autenticado (es decir, se garantiza su identidad) mientras que el cliente se mantiene sin autenticar.

SSL implica una serie de fases básicas:

* Negociar entre las partes el algoritmo que se usará en la comunicación
* Intercambio de claves públicas y autenticación basada en certificados digitales
* Cifrado del tráfico basado en cifrado simétrico

Durante la primera fase, el cliente y el servidor negocian qué algoritmos criptográficos se van a usar. Las implementaciones actuales proporcionan las siguientes opciones:

* Para criptografía de clave pública: RSA, Diffie-Hellman, DSA (Digital Signature Algorithm) o Fortezza.
* Para cifrado simétrico: RC2, RC4, IDEA (International Data Encryption Algorithm), DES (Data Encryption Standard), Triple DES y AES (Advanced Encryption Standard).
* Con funciones hash: MD5 o de la familia SHA.

El protocolo SSL está soportado a través del paquete javax.net.ssl, que es bastante configurable y, por lo tanto, complejo en su utilización. Sin embargo, los usos más normales de SSL son muy fáciles de implementar porque se basan en las clases ***SSLSocket*** y ***SSLServerSocket*** que funcionan de forma muy similar a las ya familiares ***Socket*** y ***ServerSocket***.

**Programa Servidor Seguro.**

Un servidor típico sería:

import java.io.\*;

import javax.net.ssl.\*;

public class Servidor {

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

// Obtenemos el objeto de tipo Factory para crear sockets SSL

SSLServerSocketFactory fact = (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();

// Utilizamos el objeto para crear un socket servidor seguro

SSLServerSocket socketServidorSsl = (SSLServerSocket)fact.createServerSocket( 9999 );

SSLSocket socketSsl = (SSLSocket)socketServidorSsl.accept();

// Creamos un canal de entrada sobre el socket seguro que hemos abierto

BufferedReader entrada = new BufferedReader(new InputStreamReader(socketSsl.getInputStream()));

String linea = null;

System.out.println( "Esperando..." );

// Presentamos todas las líneas que vayan llegan entrando en

// el canal a través del socket

while( (linea = entrada.readLine()) != null ) {

System.out.println( linea );

System.out.flush();

}

}

}

Para ejecutar el servidor es necesario indicar el certificado que se utilizará. Si el certificado se llamara: ***claveSSL.crt***, el comando a utilizar para la ejecución del servidor será:

java -Djavax.net.ssl.keyStore=StoreSSL -Djavax.net.ssl.keyStorePassword=cualquiera Servidor

La apliación cliente sería:

import java.io.\*;

import javax.net.ssl.\*;

public class Cliente {

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

// Obtenemos el objeto de tipo Factory para crear sockets SSL

SSLSocketFactory fact = (SSLSocketFactory)SSLSocketFactory.getDefault();

// Utilizamos el objeto para crear un socket seguro

SSLSocket socketSsl = (SSLSocket)fact.createSocket( "localhost",9999 );

// Consola desde la que leemos la entrada del usuario

BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));

// Canal de comunicación con el servidor de eco

BufferedWriter salida = new BufferedWriter(new OutputStreamWriter(socketSsl.getOutputStream()));

String linea = null;

System.out.println( "Listo..." );

// Vamos enviando las líneas al servidor

while( (linea = entrada.readLine()) != null ) {

salida.write( linea+'\n' );

salida.flush();

}

}

}

El comando para invocar al cliente es el siguiente:

java -Djavax.net.ssl.trustStore=TrustSSL -Djavax.net.ssl.trustStorePassword=cualquiera Cliente

Para crear un certificado diferente es necesario utilizar la herramienta ***keytool***, proporcionada por la plataforma Java 2. A continuación, se reproducen los comandos utilizados para generar el archivo de certificado *claveSSL.crt*.

% keytool -genkey -alias claveSsl -keyalg RSA -keystore StoreSSL

El siguiente comando exporta el certificado a un archivo:

% keytool -export -alias claveSsl -keystore StoreSSL -rfc -file claveSSL.crt

Finalmente, es necesario incorporar el certificado al nuevo almacenamiento para permitir realizar la validación; lo cual se hace con el comando siguiente:

% keytool -import -alias claveSsl -file claveSSL.crt -keystore TrustSSL

Observando el código de las aplicaciones servidor y cliente, se observa que la arquitectura de seguridad en Java utiliza el patrón de diseño *Factory* de modo que lo primero que hay que hacer siempre es obtener un objeto de este tipo. En este caso se obtiene a través de los métodos estáticos que proporcionan los objetos por defecto ***ServerSocketFactory***, en el caso del servidor, y ***SocketFactory***, en el caso del cliente.

Dos métodos importantes del API de seguridad de sockets en los que se incluyen estos objetos son ***getDefaultCipherSuites***() y ***getSupportedCipherSuites***(), que proporcionan una lista de combinación de algoritmos criptográficos definidos para un determinado nivel de seguridad en una conexión segura basada en SSL. Es decir, define si la conexión es encriptada, si la integridad es verificada o como se realiza la autenticación.

Otro método importante en el caso del servidor es ***getNeedClientAuth***(), que será el que determine si el socket requerirá autenticación al cliente, porque la especificación SSL indica que la autenticación del cliente es opcional.

El API de la extensión de seguridad para sockets define más métodos para el control de la conexión, todos ellos concentrados en las clases ***SSLSocketFactory***, ***SSLServerSocketFactory***, ***SSLSocket*** y ***SSLServerSocket***, que permite crear y utilizar sockets seguros.

La conexión segura utilizando el protocolo HTTPS, que es el protocolo HTTP sobre SSL, se maneja de forma muy sencilla. Es suficiente con indicar el acceso a direcciones que comiencen con https a través de la siguiente propiedad:

System.setProperty( "java.protocol.handler.pkgs","com.sun.nel.ssl.internal.www.protocol" );

Estableciendo esta propiedad es posible acceder a una URL segura de la forma siguiente:

URL url = new URL ( "https://digitalid.verisign.com" );

InputStream is = (InputStream)url.getContent();

# Bibliografía Web

<http://issuu.com/carlosegomez/docs/cripto>

<http://ccia.ei.uvigo.es/docencia/SSI-grado/1213/practicas/tutorial-jca/index.html>

<http://zonatic.usatudni.es/aprendizaje/aprende-sobre-el-dnie/57-aspectos-tecnicos/196-criptografia-y-esquemas-de-clave-publica.html>

<http://www.kobu.com/java/ssl/index-en.htm>

<https://www.digicert.com/es/easy-csr/keytool.htm>

<http://fismat.umich.mx/computacion/notas/parte20/cap20-19.html>