



Programación de Servicios y Procesos

6.5 Comunicaciones seguras en Java



I.E.S.
Doctor Balmis

Apuntes de PSP (https://psp2dam.github.io/psp_sources/es/) creados por Vicente Martínez bajo licencia
CC BY-NC-SA 4.0  (<http://creativecommons.org/licenses/by-nc-sa/4.0/deed.es>)

6.5 Comunicaciones seguras en Java

- 6.5.1. Protocolo SSL (Secure Sockets Layer)
- 6.5.2. JSSE
 - SSLSocket y SSLServerSocket
 - keytool: certificados, confianza y almacenes de claves
- 6.5.3. Servidor y Cliente SSL

6.5.1. Protocolo SSL (Secure Sockets Layer)

SSL es el protocolo habitualmente usado para encriptar la comunicación cliente-servidor. Casi todo el tráfico de la red puede encriptarse con SSL: POP, IMAP, telnet, FTP, etc, pero es especialmente interesante para dotar de seguridad al protocolo HTTP, es decir como base del HTTPS.

La implementación de SSL es una extensión de los sockets que permite establecer un canal (stream) de comunicación. Dicha comunicación se inicia con un **handshake** durante el cual, el cliente y el servidor construyen una session-key (clave simétrica encriptada con par de claves asimétricas) compartida para verificar su identidad mutua.

6.5.2. 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 Java de los protocolos SSL y TLS e incluye funcionalidad de

- encriptación de datos
- autenticación de servidores
- integridad de mensajes
- autenticación de clientes

Con JSSE, los programadores 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.

SSLSocket y SSLServerSocket

Las clases SSLSocket y SSLServerSocket representan sockets seguros y son derivadas de las ya conocidas 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 estático getDefault().

Para obtener un socket servidor seguro o *SSLServerSocket*:

```
SSLServerSocketFactory sfact = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
SSLServerSocket servidorSSL = (SSLServerSocket) sfact.createServerSocket(puerto);
```

java

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);
```

java

El método createSocket (String host, int puerto) crea un socket y lo conecta con el host y el puerto especificados.

keytool: certificados, confianza y almacenes de claves

Cuando dos socket SSL, uno cliente y otro servidor, intentan establecer conexión, tienen que "presentarse" el uno al otro y comprobar que el otro es de confianza. Si todo va bien y uno confía en el otro, la conexión se establece, en caso contrario, no se establece.

Para establecer esa confianza se debe crear un certificado en el servidor y añadirlo a los certificados de confianza del cliente.

- El servidor debe tener su propio certificado. Si no lo tenemos, se puede generar primero una pareja de claves con la herramienta **keytool**, que viene incluida en el JDK de Java. La herramienta guardará la pareja de claves en un almacén (el cual tiene su propia clave).
- Después generaremos un certificado a partir de esa pareja.
- El código del servidor necesitará indicar el fichero donde se almacenan las claves y la clave para acceder a ese almacén.
- El cliente necesitará indicar que confía en el certificado del servidor. Dicho certificado del servidor puede estar guardado (por ejemplo) en el almacén de claves del cliente.

Seguridad mutua

Aunque no suele hacerse también podría hacerse a la inversa y obligar al cliente a tener un certificado que el servidor pudiera importar, lo que aumentaría la seguridad.

Vamos a ver cómo realizar estas operaciones previas con la herramienta **keytool**

Primero las acciones a realizar en el servidor

```
# El servidor genera una pareja de claves que se almacena en un
# fichero llamado "clavesservidor". Dentro del fichero se indica
# un alias para poder referirnos a esa clave fácilmente
# keytool -genkey -keyalg RSA -alias servidor -keystore ClavesServidor -storepass 12345678
"C:/Program Files/Java/jdk-11.0.11/bin/keytool.exe" -genkey -keyalg RSA -alias servidor -keystore ClavesServidor -storepass 1
What is your first and last name?
[Unknown]: Vicente Martinez
What is the name of your organizational unit?
[Unknown]: Dpto. Informatica
What is the name of your organization?
[Unknown]: IES Doctor Balmis
What is the name of your City or Locality?
[Unknown]: Alicante
What is the name of your State or Province?
[Unknown]: Alicante
What is the two-letter country code for this unit?
[Unknown]: ES
Is CN=Vicente Martinez, OU=Dpto. Informatica, O=IES Doctor Balmis, L=Alicante, ST=Alicante, C=ES correct?
[no]: yes

# El servidor genera su "certificado", es decir un fichero que
# de alguna forma indica quien es él. El certificado se almacena
# en un fichero llamado clavesservidor y a partir de él queremos
# generar el certificado de un alias creado previamente con nombre servidor
```

sh

```
# keytool -exportcert -alias servidor -file servidor.cer -keystore ClavesServidor
"C:/Program Files/Java/jdk-11.0.11/bin/keytool.exe" -exportcert -alias servidor -file servidor.cer -keystore ClavesServidor
Enter keystore password:
Certificate stored in file <servidor.cer>
```

En la carpeta donde hemos ejecutado el comando keytool, se ha creado el almacén de claves en un fichero llamado **ClavesServidor** y el certificado exportado en el archivo **servidor.cer**

y a continuación las que habría que realizar en el cliente para generar el almacén de confianza que, en nuestro caso contenga el mismo certificado que hemos exportado del servidor.

```
# Se importa el certificado del servidor indicando que pertenece a
# La lista de certificados confiables.
# keytool -importcert -trustcacerts -alias servidor -file servidor.cer -keystore clavescliente -storepass 87654321
"C:/Program Files/Java/jdk-11.0.11/bin/keytool.exe" -importcert -trustcacerts -alias servidor -file servidor.cer -keystore Ce
Owner: CN=Vicente Martinez, OU=Dpto. Informatica, O=IES Doctor Balmis, L=Alicante, ST=Alicante, C=ES
Issuer: CN=Vicente Martinez, OU=Dpto. Informatica, O=IES Doctor Balmis, L=Alicante, ST=Alicante, C=ES
Serial number: 4eb6f2a5
Valid from: Mon Jan 24 00:20:24 CET 2022 until: Sun Apr 24 01:20:24 CEST 2022
Certificate fingerprints:
    SHA1: DD:A2:75:4C:1C:BC:39:60:BE:B1:20:67:E1:5C:45:8C:48:B5:1F:54
    SHA256: 82:C8:56:C2:DB:DE:8C:73:A9:21:C6:7D:DE:1F:39:4F:79:CC:5F:D5:10:BC:61:DA:E3:EE:E1:1D:21:EA:D2:33
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 16 AE 35 7C 58 97 B1 95   91 9B FA 6C 2A 80 D2 90   ..5.X.....1*...
0010: 1C 50 7F C0                .P..
]
]

Trust this certificate? [no]: yes
Certificate was added to keystore
```

En esta ocasión también se crea, en la carpeta donde se ha ejecutado el comando keytool, un fichero **CertificadosConfianzaCliente** con el/los certificados de confianza para el cliente.

6.5.3. Servidor y Cliente SSL

En le siguiente ejemplo podemos observar como la inicialización del `ServerSocket` es diferente, pero a partir de que se llama al método `accept` y, en este caso se obtiene una instancia de tipo `SSLSocket`, el resto del código es igual que con la clase `Socket`.

```
1 public class U6S5_SSLServer {
2
3     public static void main(String[] arg) throws IOException {
4
5         SSLSocket clienteConectado = null;
6         DataInputStream flujoEntrada = null; //FLUJO DE ENTRADA DE CLIENTE
7         DataOutputStream flujoSalida = null; //FLUJO DE SALIDA AL CLIENTE
8
```

```

9      // Las propiedades se pueden especificar mediante código, o bien mediante
10     // argumentos de La JVM en La Llamada a La aplicación
11     // System.setProperty("javax.net.ssl.keyStore", System.getProperty("user.dir") + "\\ClavesServidor");
12     // System.setProperty("javax.net.ssl.keyStorePassword", "12345678");
13
14     // Inicialización del ServerSocket SSL
15     int puerto = 6000;
16     SSLServerSocketFactory sfact = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
17     SSLServerSocket servidorSSL = (SSLServerSocket) sfact.createServerSocket(puerto);
18
19     for (int i = 1; i < 5; i++) {
20
21         System.out.println("Esperando al cliente " + i);
22
23         // Se espera La conexión de un cliente con accept
24         clienteConectado = (SSLSocket) servidorSSL.accept();
25         // Trabajamos do DataInputStream y DataOutputStream para simplificar
26         // el código del ejemplo
27         flujoEntrada = new DataInputStream(clienteConectado.getInputStream());
28         flujoSalida = new DataOutputStream(clienteConectado.getOutputStream());
29
30         // El cliente envía un mensaje
31         System.out.println("Recibiendo del CLIENTE: " + i + " \n\t" + flujoEntrada.readUTF());
32
33         // El Servidor responde con un saludo
34         flujoSalida.writeUTF("Saludos al cliente del servidor");
35     }
36
37     // CERRAR STREAMS Y SOCKETS
38     flujoEntrada.close();
39     flujoSalida.close();
40     clienteConectado.close();
41     servidorSSL.close();
42 }
43 }

```

En el cliente el proceso es el mismo, tras la llamada al método createSocket, obtenemos una instancia de SSLSocket que utilizamos igual que si fuese un Socket.

```

1      public class U6S5_SSLClient {
2
3          public static void main(String[] arg) throws IOException {
4
5              SSLSocket clienteSSL = null;
6              DataInputStream flujoEntrada = null; //FLUJO DE ENTRADA DE CLIENTE
7              DataOutputStream flujoSalida = null; //FLUJO DE SALIDA AL CLIENTE
8
9              // Las propiedades se pueden especificar mediante código, o bien mediante
10             // argumentos de La JVM en La Llamada a La aplicación
11             // System.setProperty("javax.net.ssl.trustStore", System.getProperty("user.dir") + "/CertificadosConfianzaClien
12             System.setProperty("javax.net.ssl.trustStore", "CertificadosConfianzaCliente");
13             System.setProperty("javax.net.ssl.trustStorePassword", "87654321");
14             // Inicialización del ServerSocket SSL
15             int puerto = 6000;
16             String host = "localhost";
17
18             System.out.println("Programa Cliente iniciado...");
19             SSLSocketFactory sfact = (SSLSocketFactory) SSLSocketFactory.getDefault();
20             clienteSSL = (SSLSocket) sfact.createSocket(host, puerto);

```

```

21
22 // Trabajamos do DataInputStream y DataOutputStream para simplificar
23 // el código del ejemplo
24 flujoSalida = new DataOutputStream(clienteSSL.getOutputStream());
25 flujoEntrada = new DataInputStream(clienteSSL.getInputStream());
26
27 // Envío un saludo al servidor
28 flujoSalida.writeUTF("Saludos al SERVIDOR DESDE EL CLIENTE");
29
30 // EL Servidor responde con un mensaje
31 System.out.println("Recibiendo del SERVIDOR: \n\t" + flujoEntrada.readUTF());
32
33 // CERRAR STREAMS Y SOCKETS
34 flujoEntrada.close();
35 flujoSalida.close();
36 clienteSSL.close();
37 }
38
39 void mostrarInformacionSesionSSL(SSLSocket cliente) throws SSLPeerUnverifiedException {
40
41 //-----
42 //Ejemplo de la múltiple información sobre la sesión SSL
43 // que se puede obtener a partir
44 SSLSession session = ((SSLSocket) cliente).getSession();
45 System.out.println("Host: " + session.getPeerHost());
46 System.out.println("Cifrado: " + session.getCipherSuite());
47 System.out.println("Protocolo: " + session.getProtocol());
48
49 System.out.println("IDentificador:" + new BigInteger(session.getId()));
50
51 System.out.println("Creación de la sesión: " + session.getCreationTime());
52
53 X509Certificate certificate = (X509Certificate) session.getPeerCertificates()[0];
54 System.out.println("Propietario: " + certificate.getSubjectDN());
55 System.out.println("Algoritmo: " + certificate.getSigAlgName());
56 System.out.println("Tipo: " + certificate.getType());
57 System.out.println("Emisor: " + certificate.getIssuerDN());
58 System.out.println("Número Serie: " + certificate.getSerialNumber());
59 //-----
60 }
61 }

```

Si ejecutamos el cliente y el servidor directamente, obtendremos el siguiente error

```

Programa Cliente iniciado...
Exception in thread "main" javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure

```

sh

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

Lo podemos indicar a la hora de ejecutar el programa, añadiendo a la línea de comandos

```
java -Djavax.net.ssl.keyStore=ClavesServidor -Djavax.net.ssl.keyStorePassword=12345678
```

sh

y en el programa cliente es necesario indicar la ubicación de los certificados de confianza

```
java -Djavax.net.ssl.trustStore=CertificadosConfianzaCliente -Djavax.net.ssl.trustStorePassword=87654321
```

sh

o bien, en las opciones de ejecución del IDE



Parámetros de JVM o código

En el código anterior hay unas líneas comentadas que especifican cómo podemos configurar el valor de las propiedades desde código mediante el uso del método `System.setProperty(String propiedad, String valor)`.

En el programa servidor incluiríamos las siguientes líneas:

```
System.setProperty("javax.net.ssl.keyStore", "ClavesServidor"); System.setProperty("javax.net.ssl.keyStorePassword", "12345678");
```

Y en el programa cliente serían estas:

```
System.setProperty("javax.net.ssl.trustStore", "CertificadosConfianzaCliente");
System.setProperty("javax.net.ssl.trustStorePassword", "87654321");
```

Además del nombre, podemos indicar la ruta donde se encuentran los almacenes, recordando que NO ES RECOMENDABLE el uso de barras invertidas para indicar rutas.

Una vez hechos los cambios, bien en las opciones de la JVM o bien en el código de las aplicaciones, la salida que obtendremos será esta

```
Esperando al cliente 1
Recibiendo del CLIENTE: 1
    Saludos al SERVIDOR DESDE EL CLIENTE
Esperando al cliente 2
```

sh

```
Recibiendo del CLIENTE: 2
    Saludos al SERVIDOR DESDE EL CLIENTE
Esperando al cliente 3
```

```
Programa Cliente iniciado...
Recibiendo del SERVIDOR:
    Saludos al cliente del servidor
```

sh



Default trusted certificates

Whenever Java attempts to connect to another application over SSL (e.g.: HTTPS, IMAPS, LDAPS), it will only be able to connect to that application if it can trust it. The way trust is handled in the Java world is that you have a keystore (typically `$JAVA_HOME/lib/security/cacerts`), also known as the truststore. This contains a list of all known Certificate Authority (CA) certificates, and Java will only trust certificates that are signed by one of those CAs or public certificates that exist within that keystore.

If `-Djavax.net.ssl.trustStore` has been configured, it will override the location of the default truststore, which will need to be checked.