

MP_0490.

**Programación de servicios y procesos
UF4. Técnicas de programación segura**

4.4. Sockets seguros

Índice

☰	Objetivos	3
☰	SSL/TLS y HTTPS	4
☰	SSH	6
☰	SFTP vs FTPS	8
☰	Introducción	9
☰	Programación con sockets seguros	10
☰	Resumen	16

Objetivos

En esta lección perseguimos los siguientes objetivos:

1

Conocer algunos protocolos de red seguros, como SSL, TLS, SSH y SFTP.

2

Comprender cómo funciona el protocolo HTTPS.

3

Crear programas Java que usen *sockets* seguros utilizando las clases *SSLSocket*, *SSLServerSocket*, *SSLSocketFactory* y *SSLServerSocketFactory*.

¡Ánimo y adelante!

SSL/TLS y HTTPS

SSL (Secure Socket Layer) y TLS (Transport Layer Security) son dos servicios de red criptográficos que funcionan de manera similar, proporcionando seguridad a las comunicaciones a través de la red.

Estos servicios se sitúan en la capa de transporte dentro de la pila de protocolos IP.

(i) Importante: TLS es el sucesor de SSL.

Los sitios web que utilizan el protocolo HTTPS (*Hypertext Transport Protocol Secure*) hacen uso de estos servicios, de manera que todas las comunicaciones que se establecen (petición y respuesta) van cifradas.

Cómo funcionan los servicios SSL y TLS

Vamos a verlo **paso a paso**:

1

Un usuario realiza una petición a un sitio web seguro.



2

El servidor enviará un certificado específico para ese sitio web, que incluye una firma electrónica (identificación del sitio) y su clave pública.

Si el servidor no tiene dicho certificado, se produce un **error**.



3

El navegador realiza las verificaciones oportunas para comprobar que el certificado enviado por el servidor es confiable. De lo contrario, preguntará al usuario si desea continuar la navegación bajo su responsabilidad.

4

Si el certificado ha sido verificado con éxito, el navegador aprovechará la clave pública que ha enviado el servidor para iniciar una comunicación cifrada. A partir de este momento, cliente y servidor pueden comunicarse bidireccionalmente de manera segura.



SSH

SSH (*Secure Shell*) es un protocolo que permite a los administradores de servidores controlar estos de manera remota, es decir, desde otro equipo conectado a la misma red.

Pero, ¿eso no lo hacía el protocolo Telnet?

Podríamos decir que **SSH es la versión segura de Telnet**, ya que utiliza cifrado en las comunicaciones establecidas durante el control remoto.



Linux dispone de un sencillo comando para conexión remota a otro equipo. Es tan simple como utilizar la siguiente sintaxis:

```
ssh usuario@[IP]:[puerto]
```

Ejemplo:

```
ssh federico@192.168.3.1:25
```

El usuario *Federico* da comienzo a una conexión remota segura al *host 192.168.3.1* a través del puerto *25*.

SFTP vs FTPS

SFTP y FTPS son protocolos seguros para la transferencia de ficheros. Pero, ¿son iguales?

La respuesta es NO.

1

FTPS (File Transfer Protocol SSL): utiliza el servicio SSL para el cifrado de datos. Soporta el algoritmo simétrico AES y los algoritmos asimétricos DSA y RSA. Actualmente es el más utilizado, ya que es una mejora añadida al protocolo FTP que emplea el mismo tipo de comandos, solo que utilizando cifrado para la transferencia de los archivos.

2

SFTP (SSH File Transfer Protocol): funciona de manera muy diferente al protocolo FTPS, ya que no se trata de una mejora de FTP, sino de un nuevo protocolo construido desde cero para añadir la transferencia de archivos en el protocolo SSH estudiado anteriormente.

SFTP es más moderno y avanzado que FTPS, pero no es compatible todavía con todos los sistemas y dispositivos. Sin embargo, **FTPS es mucho más compatible**, ya que realmente es una extensión de FTP.

Introducción

Las librerías estándar de Java cuentan con un conjunto de clases que implementan la funcionalidad de sockets seguros mediante servicios SSL.

Estas clases están localizadas en el paquete **javax.net.ssl**

Estas clases funcionan de manera muy similar a las vistas anteriormente, pero incorporan seguridad mediante los servicios de cifrado SSL. Estas son las **clases que utilizaremos**:

- **SSLSocketFactory**
- **SSLSocket**
- **SSLServerSocketFactory**
- **SSLServerSocket**

Para la parte práctica, **adaptaremos el primer ejemplo que desarrollamos sobre Socket**, el más sencillo. Recuerda que se trataba de un modelo en el que el cliente enviaba sucesivos mensajes al servidor, finalizando al enviar la cadena "FIN". El servidor respondía indicando el número de caracteres del mensaje.

¿Comenzamos una versión segura de aquella aplicación?

Programación con sockets seguros

La aplicación cliente/servidor que vamos a desarrollar se corresponde con la que aparece en el siguiente apartado del curso:

2.2. Desarrollo de aplicaciones cliente/servidor con sockets
SOCKETS EN JAVA
Utilizando *sockets stream*

Nos limitaremos a adaptar la aplicación para:

- Implementar el servidor, utilizando un objeto de tipo *SSLServerSocket*.
- Implementar el cliente, utilizando un objeto de tipo *SSLSocket*.

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.Socket;

import javax.net.ssl.SSLServerSocket;
import javax.net.ssl.SSLServerSocketFactory;

public class Servidor {

    public static void main(String[] args) {
        System.out.println("      APPLICACIÓN DE SERVIDOR      ");
    }
}
```

```
System.out.println("-----");
try {
    SSLServerSocketFactory fabrica = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
    SSLServerSocket servidor = (SSLServerSocket) fabrica.createServerSocket();
    InetSocketAddress direccion = new InetSocketAddress("192.168.56.1", 2000);
    servidor.bind(direccion);

    System.out.println("Servidor creado y escuchando .... ");
    System.out.println("Dirección IP: " + direccion.getAddress());

    Socket enchufeAlCliente = servidor.accept();
    System.out.println("Comunicación entrante");

    InputStream entrada = enchufeAlCliente.getInputStream();
    OutputStream salida = enchufeAlCliente.getOutputStream();

    String texto = "";
    while (!texto.trim().equals("FIN")) {
        byte[] mensaje = new byte[100];
        entrada.read(mensaje);
        texto = new String(mensaje);
        if (texto.trim().equals("FIN")) {
            salida.write("Hasta pronto, gracias por establecer conexión".getBytes());
        } else {
            System.out.println("Cliente dice: " + texto);
            salida.write(("Tu mensaje tiene " + texto.trim().length() + " caracteres").getBytes());
        }
    }
    entrada.close();
    salida.close();
    enchufeAlCliente.close();
    servidor.close();

    System.out.println("Comunicación cerrada");
} catch (IOException e) {
    System.out.println(e.getMessage());
}
}
```

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.UnknownHostException;
import java.util.Scanner;

import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;

public class SocketCliente {

    public static void main(String[] args) {
        System.out.println("      APLICACIÓN CLIENTE");
        System.out.println("-----");

        Scanner lector = new Scanner(System.in);
        try {
```

```
SSLocketFactory fabrica = (SSLocketFactory) SSLocketFactory.getDe-
fault();
SSLocket cliente = (SSLocket) fabrica.createSocket();
InetSocketAddress direccionServidor = new InetSocketAd-
dress("192.168.56.1",2000);
System.out.println("Esperando a que el servidor acepte la conexión");
cliente.connect(direccionServidor);
System.out.println("Comunicación establecida con el servidor");

InputStream entrada = cliente.getInputStream();
OutputStream salida = cliente.getOutputStream();

String texto = "";
while (!texto.equals("FIN")) {
    System.out.println("Escribe mensaje (FIN para terminar): ");
    texto = lector.nextLine();
    salida.write(texto.getBytes());
    byte[] mensaje = new byte[100];
    entrada.read(mensaje);
    System.out.println("Servidor responde: " + new String(mensaje));
}

entrada.close();
salida.close();
cliente.close();
lector.close();
System.out.println("Comunicación cerrada");

} catch (UnknownHostException e) {
    System.out.println("No se puede establecer comunicación con el servi-
dor");
    System.out.println(e.getMessage());
} catch (IOException e) {
    System.out.println("Error de E/S");
    System.out.println(e.getMessage());
}
}
```

Como ves, tenemos implementados un programa cliente y un programa servidor que se comunican de manera segura. Pero, para que esta comunicación sea posible, **es necesario que el servidor cuente con un certificado digital en el que el cliente pueda confiar.**

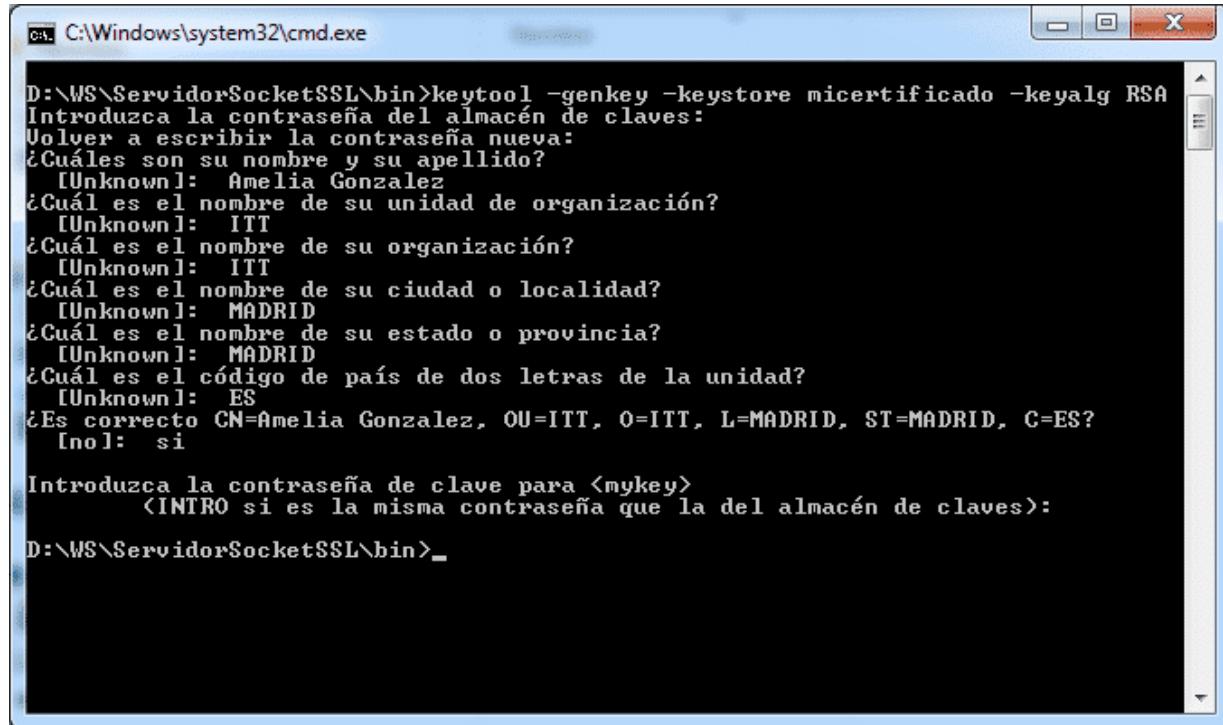
Cómo gestionar certificados

Java dispone de una herramienta, denominada **keytool**, para la gestión de certificados. A continuación, **vamos a mostrarte cómo funciona**.

Para crear el certificado de tu servidor, abre una ventana de símbolo del sistema, sitúate en la ruta donde se encuentra el fichero *Servidor.class*, y escribe el siguiente comando:

```
keytool -genkey -keystore micertificado -keyalg RSA
```

El comando *keytool* te solicitará una contraseña que deberás introducir dos veces; después te realizará una serie de preguntas, como puedes comprobar en la siguiente imagen:



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The command entered is 'keytool -genkey -keystore micertificado -keyalg RSA'. The window displays the following interaction:

```
D:\WS\ServidorSocketSSL\bin>keytool -genkey -keystore micertificado -keyalg RSA
Introduzca la contraseña del almacén de claves:
Volver a escribir la contraseña nueva:
¿Cuáles son su nombre y su apellido?
[Unknown]: Amelia Gonzalez
¿Cuál es el nombre de su unidad de organización?
[Unknown]: ITT
¿Cuál es el nombre de su organización?
[Unknown]: ITT
¿Cuál es el nombre de su ciudad o localidad?
[Unknown]: MADRID
¿Cuál es el nombre de su estado o provincia?
[Unknown]: MADRID
¿Cuál es el código de país de dos letras de la unidad?
[Unknown]: ES
¿Es correcto CN=Amelia Gonzalez, OU=ITT, O=ITT, L=MADRID, ST=MADRID, C=ES?
[no]: si

Introduzca la contraseña de clave para <mykey>
<INTRO si es la misma contraseña que la del almacén de claves>:

D:\WS\ServidorSocketSSL\bin>
```

Con la ejecución de este comando has creado un certificado y lo has guardado en un fichero de almacén de claves o *keystore*. El algoritmo que has utilizado es RSA.



Para ejecutar el cliente y el servidor, tendrás que especificar el **nombre del certificado** y la **contraseña** que indicaste en el momento de lanzar ambos programas. Debes lanzar ambos desde el símbolo del sistema.

Cómo lanzar el servidor

Para lanzar el servidor deberás introducir el siguiente comando desde el símbolo del sistema:

```
Java -Djavax.net.ssl.keyStore=micertificado -Djavax.net.ssl.trustStore=mi-  
certificado -Djavax.net.ssl.keyStorePassword=cocodrilo Servidor
```

Puedes guardar este comando en un archivo de procesamiento por lotes, denominado *lanzarservidor.bat*, para no tener que escribirlo cada vez que ejecutes el servidor.

Cómo lanzar el cliente

Para lanzar el cliente debes introducir el siguiente comando desde una ventana de símbolo del sistema:

```
Java -Djavax.net.ssl.keyStore=micertificado -Djavax.net.ssl.trustStore=mi-  
certificado -Djavax.net.ssl.keyStorePassword=cocodrilo SocketCliente
```

Igual que en el caso del servidor, podemos guardar este comando en un archivo llamado *lanzarcliente.bat* para evitar escribirlo completo cada vez que deseemos ejecutar.

Observa los dos programas (servidor y cliente) en funcionamiento:

```
ca C:\Windows\system32\cmd.exe
D:\WS\ServidorSocketSSL\bin>lanzarservidor
D:\WS\ServidorSocketSSL\bin>Java -Djavax.net.ssl.keyStore=micertificado -Djavax.net.ssl.trustStore=micertificado -Djavax.net.ssl.keyStorePassword=cocodrilo Servidor
    APPLICACIÓN DE SERVIDOR
Servidor creado y escuchando ....
Dirección IP: /192.168.1.43
Comunicación entrante
Cliente dice: HOLA CARACOLA
Cliente dice: ESTO FUNCIONA BIEN
Cliente dice: ESTE ES EL ULTIMO MENSAJE
Comunicación cerrada
D:\WS\ServidorSocketSSL\bin>
```

```
ca C:\Windows\system32\cmd.exe
D:\WS\ServidorSocketSSL\bin>lanzarcliente
D:\WS\ServidorSocketSSL\bin>Java -Djavax.net.ssl.keyStore=micertificado -Djavax.net.ssl.trustStore=micertificado -Djavax.net.ssl.keyStorePassword=cocodrilo SocketCliente
    APPLICACIÓN CLIENTE
Esperando a que el servidor acepte la conexión
Comunicación establecida con el servidor
Escribe mensaje <FIN para terminar>:
HOLA CARACOLA
Servidor responde: Tu mensaje tiene 13 caracteres
Escribe mensaje <FIN para terminar>:
ESTO FUNCIONA BIEN
Servidor responde: Tu mensaje tiene 18 caracteres
Escribe mensaje <FIN para terminar>:
ESTE ES EL ULTIMO MENSAJE
Servidor responde: Tu mensaje tiene 25 caracteres
Escribe mensaje <FIN para terminar>:
FIN
Servidor responde: Hasta pronto, gracias por establecer conexión
Comunicación cerrada
D:\WS\ServidorSocketSSL\bin>
```

Resumen

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- SSL (*Secure Socket Layer*) y TLS (*Transport Layer Security*) son dos servicios de red criptográficos que funcionan de manera similar, proporcionando seguridad a las comunicaciones a través de la red. Estos servicios se sitúan en la capa de transporte dentro de la pila de protocolos IP. Los sitios web que utilizan el protocolo HTTPS (*Hypertext Transport Protocol Secure*) hacen uso de estos servicios de manera que todas las comunicaciones que se establecen (petición y respuesta) van cifradas.
- SSH (*Secure Shell*) es un protocolo que permite a los administradores de servidores controlar estos de manera remota, es decir, desde otro equipo conectado a la misma red.
- SFTP y FTPS son protocolos seguros para la transferencia de ficheros, el primero basado en SSH y el segundo en SSL.
- Las librerías estándar de Java cuentan con un conjunto de **clases que implementan la funcionalidad de sockets seguros mediante servicios SSL**. Todas estas clases están en el paquete *javax.net.ssl*.



PROEDUCA