

## Seguridad en HTTP

### Introducción

Esta práctica nos introduce en los dos puntos importantes sobre seguridad en HTTP: la autenticación y el transporte seguro de datos.

Para el transporte seguro de datos se ha seleccionado SSL (Sockets seguros), que es la más usada (por ejemplo para conexiones http seguras).

Se describen dos técnicas para verificar la identidad de la persona que quiere acceder a un documento:

- La autenticación básica consiste en un sencillo sistema de nombres de usuarios y contraseñas.
- También veremos un sistema de autenticación usando certificados personales. Como sistema seguro en el transporte de datos veremos SSL.

### Objetivos

- Identificar las partes implicadas en un sistema seguro.
- Distinguir entre autenticación y transporte seguro de datos.
- Saber decidir en que casos aplicar las técnicas de autenticación y transporte seguro.
- Aprender a utilizar técnicas de seguridad.
- Diseñar y programar sistemas seguros.

### Sockets Seguros (SSL)

Secure Sockets Layer, SSL, es un protocolo desarrollado por Netscape para transmitir documentos privados por Internet. El SSL trabaja usando una llave privada para cifrar los datos que se transfieren por la conexión SSL. SSL permite:

1. Que el cliente compruebe (autenticar) la identidad de la máquina servidor (pero no del usuario final), si el servidor tiene habilitado SSL. Un cliente que soporte SSL puede utilizar técnicas de la criptografía de llave pública para comprobar que el certificado y la identidad de un servidor están validados por una Autoridad de Certificación (CA). El cliente confía en los certificados emitidos por unas CA concretas. Esta comprobación puede ser interesante si el cliente, por ejemplo, está enviando un número de la tarjeta de crédito y desea controlar la identidad del servidor.
2. [poco frecuente] Que el servidor compruebe la identidad del usuario cliente. Con las mismas técnicas de la autenticación del servidor, el servidor con SSL puede comprobar que el certificado y la identidad del usuario cliente están validados por una CA. Esta comprobación puede ser importante si el servidor, por ejemplo, es un sistema que envía la información financiera confidencial a un cliente y desea controlar la identidad del destinatario.
3. Permite que entre ambas máquinas se establezca una conexión cifrada o segura. Todos los datos enviados a través de una conexión cifrada SSL viajan no solamente cifrados sino protegidos con un mecanismo para detectar alteraciones, es decir, para detectar automáticamente si los datos se han modificado en tránsito.

El protocolo de transporte seguro SSL trabaja sobre el TCP/IP y da servicio a protocolos de alto nivel tales como HTTP, SMTP, IMAP. Por convenio, la URL de los sitios Web que requieren SSL empieza por https y utilizan un puerto distinto.

El proceso de inicio de una conexión SSL es:

- Fase de negociación:
  - El cliente envía al servidor el número de versión del SSL, ciertas propiedades del cifrado y un dato generado aleatoriamente.
  - El servidor envía al cliente el número de versión del SSL, ciertas propiedades del cifrado, un dato generado aleatoriamente y su propio certificado. Si se usa autenticación del cliente, el servidor pide el certificado del cliente.
- Fase de creación de llaves simétricas:
  - El cliente usa la información proporcionada por el servidor para comprobar su identidad. Si no se puede comprobar la identidad del servidor el usuario es avisado del problema.
  - Usando la información intercambiada el cliente crea un secreto de sesión temporal. Lo cifra con la llave pública del servidor, forma parte del certificado, y lo envía al servidor. Si se usa autenticación del cliente, el cliente también envía su certificado.
  - Si todo el proceso ha sido correcto, incluida la comprobación de la identidad del cliente, tanto el cliente como el servidor usa el secreto temporal para generar el secreto de la conexión, con el crearán las llaves simétricas de cifrado de la sesión.
- Fase de intercambio:
  - Tanto el cliente como el servidor envían un primer mensaje cifrado para indicar al otro que todo el proceso se ha realizado correctamente.
  - A partir de este momento todo el intercambio de datos entre el cliente y el servidor será cifrada.

## HTTPServer

*HTTPServer* es un sencillo servidor HTTP en java. Nos vamos a basar en él para la realización de la práctica. Queremos hacer las modificaciones pertinentes al *HTTPServer* para que conseguir un servidor HTTP con autenticación y/o transporte seguro, SSL.

Una vez compilado *HTTPServer*, podemos probar su funcionamiento accediendo con un navegador a `http://localhost/`. El servidor nos facilitará el fichero HTML `index.htm`.

### Tarea

- Compilar, ejecutar y probar HTTPServer

## Autenticación básica

HTTP/1.0, incluye la especificación para un esquema básico de la autenticación de acceso. Este esquema no se considera un método seguro de autenticación del usuario (a menos que se esté utilizado conjuntamente con un sistema de cifrado de datos como SSL). El nombre del usuario y la contraseña van por la red en texto (codificados en base64, que no ofrece ninguna protección, es reversible).

Cuando realizamos una petición de una URL dentro del espacio protegido, el servidor responde con un desafío como el siguiente:

```
WWW-Authenticate: Basic realm="WallyWorld"
```

Donde "WallyWorld" es el nombre asignado por el servidor para identificar el espacio protegido. Un Proxy puede responder con el mismo desafío usando el campo de la cabecera de Proxy-Authenticate.

Para recibir la autorización, el cliente envía el nombre de usuario y la contraseña, separados por un solo carácter de los dos puntos (":"), codificado en base64. El nombre de usuario puede ser sensible a Mayusculas/Minusculas. Si el navegador desea enviar el usuario "Aladdin" y contraseña "open sesame", utilizaría la siguiente cabecera:

```
Authorization: Basic QWxhZGRpbjpvGVuIHNlc2FtZQ==
```

Un navegador debe asumir que todas las URL que sean una extensión de la URL protegida también estarán dentro del espacio protegido especificado por `realm` del desafío actual. Un cliente podrá enviar la cabecera correspondiente de la autorización sin haber recibido otro desafío del servidor. Así mismo, cuando un cliente envía una petición a un proxy, puede reutilizar el nombre de usuario y la contraseña con `Proxy-Authorization` sin tener que recibir otro desafío del proxy.

Al pedir un documento en es espacio protegido el servidor nos responderá con algo como

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm= WallyWorld
```

Para poder obtener el documento, tendremos que contestar con algo como

```
GET /protegido/ HTTP/1.1
Authorization: Basic QWxhZGRpbjpvGVuIHNlc2FtZQ==
Accept: */*
```

## Tarea

- Modificar `HTTPServer` para que implemente autenticación básica.

## Certificados, Keystores, y Truststores

Puesto que el SSL utiliza los certificados para la autenticación, necesitamos crear los certificados para nuestro servidor. JSSE puede utilizar los certificados creados por el *keytool* de Java. De esta forma, la creación de los certificados se hace con la ejecución de *keytool* con los parámetros pertinentes.

Sin embargo, antes, hay unas características de JSSE que tenemos que conocer. JSSE distingue entre los *keystores* y los *truststores*. *Keystore*, almacén de certificados, (desde la perspectiva de JSSE) es una base de datos de los pares de llaves y de los certificados que se utilizan para la autenticación del SSL. En un *truststore* se almacenan certificados de las Autoridades de Certificación (CA). Se utilizan para verificar las identidades de otros clientes y servidores.

Cuando un cliente o un servidor inician una sesión del SSL, extrae sus certificados y claves de su almacén de certificados (*keystore*). Cuando verifica las identidades de otros clientes o servidores, extraerá los certificados de la Autoridad de la Certificación en sus *truststores*. Si se define la variable del sistema `javax.net.ssl.trustStore`, entonces el valor de ésta se utiliza como localización del truststore.

Podemos encontrar el valor de `javax.net.ssl.trustStore` usando el programa *ShowTrustStore*.

## Generación de un certificado del servidor

Los certificados del servidor se pueden generar con un solo comando del *keytool*. Utilizamos el comando siguiente de crear un certificado de RSA, con el nombre de servidor, y salvarlo en un keystore llamado certs. *Keytool* nos pedirá la información que ha de aparecer en el certificado: nombre, organización... Es imprescindible que cuando *keytool* nos pida el nombre y apellido del certificado pongamos el nombre del host donde correrá el servidor, en nuestro caso, y puesto que lo utilizamos en un ámbito de prueba escribiremos “localhost”:

```
¿Cuáles son su nombre y su apellido?  
[Unknown]: localhost
```

### Tarea

- Crear un certificado para el servidor *SecureServer* mediante la herramienta *keytool*

## SecureServer

Podemos ver que *SecureServer* es una extensión del *HTTPServer*. Lo que se hace es cambiar los sockets normales por sockets que utilizan SSL. El método `getServerSocket()` es donde se introduce el SSL. Se sobrescribe el método `getServerSocket()` del *HTTPServer* para sustituir un sencillo `ServerSocket` por `SSLServerSocket`.

La variable `requireClientAuthentication` hay que definirla `true` si queremos autenticación del cliente con certificados. De momento no utilizaremos autenticación del cliente por certificado.

```
serverSocket.setNeedClientAuth(requireClientAuthentication);
```

Una vez compilados *HTTPServer* y *SecureServer* y creado un fichero HTML llamado `index.htm` podemos ejecutar el servidor seguro. ***Tenemos que esperar uno o varios minutos para comenzar a realizar peticiones. La generación de los números primos para el cifrado retarda el arranque.*** Podemos probar su funcionamiento accediendo con un navegador `https://localhost:4430/` El navegador inicia la conexión con el servidor *SecureServer* e intenta configurar una conexión SSL. *SecureServer* envía su certificado al navegador. Como el certificado no está firmado por una *Certificate Authority* válida, el navegador nos avisará de ello. Después de aceptar y validar el certificado, el navegador visualiza el contenido HTML del documento.

### Nota Importante

- Recordad que en la tarea anterior habéis modificado *HTTPServer* añadiéndole autenticación básica y que *SecureServer* es una extensión de *HTTPServer*, por lo tanto *SecureServer* también implementa dicha autenticación.

### Tarea

- Compilar, ejecutar y probar *SecureServer* con el certificado creado anteriormente

## Certificado Cliente

En ésta última parte de la práctica veremos cómo autenticar también el cliente en una conexión segura. En esta parte deberéis realizar las siguientes tareas:

- Configurar SecureServer con autenticación por certificado personal
- Crear el certificado personal
- Compilar, ejecutar y probar SecureServer y SecureBrowser

### Nota Importante

- Para realizar ésta parte de la práctica utilizad el fichero `HTTPServer.java` no modificado. Recordad que en la tarea anterior habéis modificado *HTTPServer* añadiéndole autenticación básica y que *SecureServer* es una extensión de *HTTPServer*, por lo tanto *SecureServer* también implementa dicha autenticación. Sin embargo *SecureBrowser* no implementa la autenticación básica.

## Cliente SSL en Java

Nuestras aplicaciones de Java pueden también requerir a clientes utilizar SSL. *Browser.java* es un *Browser* textual básico en Java. *SecureBrowser* amplía *Browser* para proporcionar SSL. Podemos preguntarnos qué sucede cuando ejecutamos *SecureBrowser* contra *SecureServer*: no funciona. Eso es porque *SecureBrowser* no puede validar el certificado de *SecureServer*. Sin embargo, podemos modificar *SecureBrowser* para validar el certificado de *SecureServer*. Usa *keytool* para exportar el certificado del servidor del keystore de los certs.

```
keytool -export -alias servidor -storepass serverkspw -file
server.cer -keystore certs
```

Utilizamos el *keytool* para crear un nuevo truststore llamado *cacerts.jks* que será utilizado por *SecureBrowser*. Importe *server.cer* en *cacerts.jks*

```
keytool -import -v -trustcacerts -alias servidor -file server.cer
-keystore cacerts.jks -keypass serverkspw -storepass serverkspw
```

Ahora podemos ejecutar *SecureBrowser* puede utilizar *cacerts.jks* como truststore para validar *SecureServer*. Para indicar al browser qué truststore debe utilizar debéis ejecutarlo definiendo la variable `javax.net.ssl.trustore` para que apunte al truststore a usar (*cacerts.jks* en nuestro caso). Asimismo se debe dar el password (*serverkspw*) para que pueda acceder, esto se hace mediante la variable `javax.net.ssl.trustStorePassword`.

### Tarea

- Crear mediante la herramienta *keytool* el truststore para *SecureBrowser*
- Compilar, ejecutar y probar *SecureBrowser* con el trustore

## Configuración de la autenticación Mutua

En algunas aplicaciones, la autenticación del cliente es esencial. Hasta ahora, tenemos *SecureBrowser* y *SecureServer* el ejecutarse cara a cara. *SecureServer* proporciona a su certificado a *SecureBrowser* y *SecureBrowser* valida *SecureServer*. Entonces conseguimos un canal de comunicaciones seguro. ¿Cómo lo hace *SecureServer* para saber que esta hablando con *SecureBrowser*? Aquí es donde interviene la autenticación mutua.

### Creación de un certificado del cliente

Lo primero que necesitamos hacer es crear un certificado para nuestro cliente. Se logra esto usando el keytool tal y como se ha hecho anteriormente para crear un certificado para el cliente:

- 1.- Crear un certificado para el cliente.
- 2.- Exportar el certificado a un fichero.
- 3.- Importar el certificado (el fichero creado anteriormente) en el servidor.

#### Tarea

- Crear un certificado para el cliente (keystore) y un truststore para que el servidor pueda autenticarlo.

### Modificaciones en SecureServer y SecureBrowser

Deseamos que *SecureBrowser* presente su certificado a *SecureServer*, de modo que *SecureServer* pueda autenticar a *SecureBrowser*. Debemos realizar las siguientes tareas:

- 1.- Modificar *SecureServer* para que pida el certificado al cliente (*SecureBrowser*).
- 2.- Indicar a *SecureBrowser* qué keystore utilizar, asimismo se le debe indicar cuál es la contraseña para acceder al keystore.

Ahora ya podéis probar la autenticación mutua entre *SecureBrowser* y *SecureServer*. No olvidéis que debéis indicar tanto al browser como al servidor dónde encontrar los respectivos truststores así como los passwords para acceder a ellos. Esto lo podéis hacer mediante las variables `javax.net.ssl.trustore` y `javax.net.ssl.trustStorePassword`.

#### Tarea

- Modificar *SecureBrowser* y *SecureServer* para que implementen la autenticación mutua
- Probar *SecureBrowser* contra *SecureServer* usando los certificados creados

**Entregar**

- Un informe que documente con detalle todas las tareas que habéis realizado en ésta práctica
- El fichero *HTTPServer* modificado por vosotros con la autenticación básica
- El certificado creado para *SecureServer* mediante la herramienta *keytool*
- Todos los ficheros que hayáis modificado
- Los certificados creados así como los comandos usados
- El resultado de ejecutar *SecureBrowser* contra *SecureServer* con autenticación mutua (copiadlo en el fichero informe.txt).