Práctica 6.- Seguridad en HTTP.

Introducción

Esta práctica nos introduce en los dos puntos importantes sobre seguridad en HTTP: la autentificación 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 autentificación básica consiste en un sencillo sistema de nombres de usuarios y contraseñas.
- Opcionalmente veremos un sistema de autentificación usando certificados personales. Como sistema seguro en el transporte de datos veremos SSL.

Objetivos

- Identificar las partes implicadas en un sistema seguro.
- Distinguir entre autentificación y transporte seguro de datos.
- Saber decidir en que casos aplicar las técnicas de autentificación y transporte seguro.
- Aprender a utilizar técnicas de seguridad.
- Diseñar y programar sistema 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 (autentificar) 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 publica para comprobar que el certificado y la identidad de un servidor estan 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) [opcional y poco frecuente] Que el servidor compruebe la identidad del usuario cliente. Con las mismas técnicas de la autentificación 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 empiezan 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 numero de versión del SSL, ciertas propiedades del cifrado y un dato generado aleatoriamente.
- El servidor envía al cliente el numero de versión del SSL, ciertas propiedades del cifrado, un dato generado aleatoriamente y su propio certificado. Si se usa autentificación 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 publica del servidor, forma parte del certificado, y lo envía al servidor. Si se usa autentificación 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.

Instalación de Java Secure Socket Extension

Puesto que las clases Java Secure Socket Extention, JSSE, no están en el JDK 1.2, tenemos que descargarlas e integrarlas en nuestra instalación actual del JDK. En JSSE encontramos los ficheros `jsse.jar, jcert.jar y jnet.jar`. Comprobemos la instalación del JSSE ejecutando el programa de *JSSETest*.

`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 autentificación 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`

Certificados, *Keystores*, y *Truststores*

Puesto que el SSL utiliza los certificados para la autentificación, 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 llaves y de los certificados que se utilizan para la autentificación 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 inicia 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*. Utilicamos el comando siguiente de crear un certificado de RSA, con el nombre de roc , y salvarlo en un keystore llamado certs.

```
keytool -genkey -keystore certs -keyalg rsa -alias roc -storepass
serverkspw -keypass serverpw
```

Ahora el keytool nos pide la información que ha de aparecer en el certificado: nombre, organización,....

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 autentificación del cliente con certificados (parte opcional de la práctica) De momento no utilizaramos autentificación 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/` 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 esta firmado por una *Certificate Authority* válida, el navegador nos avisará de ello. Después de aceptar y validar el certificado, el navegador visualiza el contendido HTML del documento.

Autentificación básica

HTTP/1.0, incluye la especificación para un esquema básico de la autentificación de acceso. Este esquema no se considera un método seguro de autentificación 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 palabra de paso "open sesame", utilizaría la siguiente cabecera:

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

Un navegador debe asumir que todos los URL que sean una extensión de la URL protegida también estará 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 QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Accept: */*
```

Tareas

- Compilar, ejecutar y probar *HTTPServer*
- Crear el certificado para el servidor.
- Compilar, ejecutar y probar *SecureServer*
- Modificar *HTTPServer* para utilizar autentificación básica, por extensión también en el *SecureServer*

Tarea Opcional (ver Anexo 1)

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

Bibliografia

Sun, *Java Secure Socket Extension*. [en línea] v1.0.2 <http://www.javasoft.com/products/jsse/> [Consulta: 16 mayo 2001]

W3C. *HTTP - Hypertext Transfer Protocol* [en línea] <http://www.w3.org/Protocols/> [Consulta: 16 mayo 2001]

Netscape Communications Corporation, Introduction to SSL [en línea] <http://developer.netscape.com/docs/manuals/security/sslin/index.htm> [Consulta: 16 mayo 2001]

N. Borenstein, Bellcore, N. Freed, Innosoft Base64 - MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies [en línea] sección 5.2 del RFC 1341 http://www.fourmilab.ch/webtools/base64/rfc1341.html

Anexo 1.-

Cliente SSL en Java

Nuestras aplicaciones de Java pueden también requerir a clientes utilizar SSL. El listado 6 es un *Browser* textual básico en Java. El listado 6 presenta *SecureBrowser* , que amplía *Browser* para proporcionar SSL. Podemos ejecutar *SecureBrowser* contra un sitio con SSL, por ejemplo, `https://www.sun.com/`

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*.

Uso keytool para exportar el certificado del servidor del keystore de los certs.

```
keytool -export –keystore certs  -alias roc -file server.cer
Enter keystore passaword: serverkspw
Certificado salvado en fichero
```

Utilizamos el keytool para crear un nuevo keystore llamado jssecacerts (que será utilizado como truststore por SecureBrowser). Importe server.cer en jssecacerts.

```
keytool -import -keystore -jssecacerts -alias roc -file server.cer
```

Finalmente, copia jssecacerts a lib/security del directorio java.home. (en la máquina cliente.)

Ahora *SecureBrowser* utilizará jssecacerts como truststore para validar *SecureServer*.

Configuración de la autentificación Mutua

En algunas aplicaciones, la autentificación del cliente es esencial. Hasta ahora, tenemos *SecureBrowser* y *SecureServer* el ejecutarse cara a cara. *SecureServer* proporciona a su certificado a *SecureBrowser* y *SecureBrowse*r 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 autentificación 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.

```
keytool -genkey -keyalg rsa -alias cliente
Enter keystore password: 12345678
...
```

Lo siguiente que necesitamos hacer es exportar el certificado. Una vez más utilizaremos el keytool.

```
keytool -export -alias cliente -file jj.cer
Enter keystore password: 12345678
...
```

Ahora, importamos el certificado (jj.cer) en un keystore en el servidor. Este keystore funcionará como el truststore del servidor.

```
keytool -import -alias cliente -file jj.cer
Enter keystore password: 12345678
...
```

El keystore que se crea del comando se llama keystore . Lo renombramos a jssecacerts y lo ponemos en subdirectorio lib/security del directorio java.home. Esto hará que nuestro servidor conozca el certificado del cliente.

Modificaciones en SecureServer y SecureBrowser

Deseamos que *SecureBrowser* presente su certificado a *SecureServer*, de modo que *SecureServer* pueda autentificar a *SecureBrowser* . Cambiamos simplemente la línea siguiente dentro del constructor de *SecureServer*

```
 this("SecureServer", "1,0", 443, false);
```

a

```
 this("SecureServer", "1,0", 443, true);
```

Esto activa autentificación mutua dentro de *SecureServer*.

En el lado del cliente, debemos decir qué keystore utilizar y qué palabra de paso a utilizar para tener acceso al keystore. Añadimos simplemente las dos líneas siguientes al final del constructor de *SecureBrowser*.

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