Seguridad y autentificación con Tomcat

Índice

1. Seguridad del servidor

1.1. Políticas de seguridad

Si los usuarios del servidor pueden colocar en él sus propias aplicaciones web es necesario asegurar que dichas aplicaciones no van a comprometer la seguridad de las demás, del servidor o del propio sistema. Aunque confiemos en la buena fe del que desarrolla las aplicaciones, alguien podría aprovechar un "bug" en una de ellas para efectuar operaciones no permitidas.

Tomcat utiliza los mecanismos estándar de seguridad de Java para asegurar que las clases ejecutadas en el servidor cumplen una serie de restricciones. Dichas restricciones se definen en el fichero de políticas de seguridad `conf/catalina.policy`. No obstante, por defecto estas restricciones no están activadas salvo que se arranque Tomcat con la opción `-security`.

Políticas de seguridad por defecto

Si se observa el contenido del fichero `conf/catalina.policy` se verá que por defecto, a las clases que componen Tomcat y a las pertenecientes al JDK se les asignan todos los permisos posibles. A continuación se muestra un extracto de dicho fichero:

```
...
// Permisos para las clases del JDK
// (están dentro de JAVA_HOME)
grant codeBase "file:${java.home}/lib/-" {
        permission java.security.AllPermission;
};
...
// Permisos para las clases de Tomcat
// (están dentro de CATALINA_HOME)
grant codeBase "file:${catalina.home}/bin/bootstrap.jar" {
        permission java.security.AllPermission;
};
...
// Permisos para aplicaciones web
grant {
        // Required for JNDI lookup of named JDBC DataSource's and
        // javamail named MimePart DataSource used to send mail
        permission java.util.PropertyPermission "java.home",
"read";
        ...
```

Como puede observarse si se examina detenidamente el fichero, a las clases que componen una aplicación web cualquiera se les dan permisos adecuados para poder acceder a JNDI, leer ciertas propiedades del sistema, etc.

Cambiar las políticas de seguridad

Cambiando el fichero de políticas podemos asignar permisos especiales a ciertas

aplicaciones web que lo requieran, manteniendo las restricciones en las demás. Por ejemplo, para dar ciertos permisos a la aplicación j2ee habrá que introducir un código similar al siguiente en el fichero de políticas

```
grant codeBase "file:${catalina.home}/webapps/j2ee/-" {
        permission ...
}
```

1.2. Autentificación en Tomcat: realms

Los mecanismos de autentificación en Tomcat se basan en el concepto de `realm`. Un `realm` es un conjunto de usuarios, cada uno con un *password* y uno o más *roles*. Los roles determinan qué permisos tiene el usuario en una aplicación web (esto es configurable en cada aplicación a través del descriptor de despliegue, `web.xml`).

Tomcat proporciona distintas implementaciones para los realms, que básicamente se diferencian en dónde están almacenados los datos de *logins*, *passwords* y *roles*. En la configuración de Tomcat por defecto, dichos datos están en el fichero `conf/tomcat-users.xml`, pero pueden almacenarse en una base de datos con JDBC, tomarse de un directorio LDAP u obtenerse mediante JAAS, el API estándar de Java para autentificación.

Los *realms* se definen en el fichero de configuración `server.xml` introduciendo el elemento `Realm` con un atributo `className` que indique la implementación que deseamos utilizar. El resto de atributos dependen de la implementación en concreto. Podemos definir *realms* en distintos puntos del fichero, de manera que afecten a todo el servidor, a un *host* o solo a una aplicación. En cada caso, siempre se utilizará el *realm* aplicable más bajo en la "jerarquía de configuración", de modo que si se configura para una aplicación en concreto, el general deja de utilizarse.

UserDatabaseRealm

Es la implementación que utiliza por defecto la distribución de Tomcat, y como se ha comentado, toma los datos de un fichero. Este fichero se lee cuando arranca el servidor, de modo que cualquier cambio en el mismo requiere el rearranque de Tomcat para tener efecto. La definición del *realm* en el fichero de configuración se realiza de la siguiente manera:

```
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"
       debug="0" resourceName="UserDatabase"/>
```

Donde el atributo `resourceName` determina el recurso del que se obtiene la información. Este está definido dentro de GlobalNamingResources para que apunte al fichero `conf/tomcat-users.xml`.

Dicho fichero tiene un formato similar al siguiente:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

```
<tomcat-users>
        <user name="pepe" password="pepepw" roles="usuario"/>
        <user name="manuel" password="manolo" roles="admin"/>
        <user name="toni" password="toni" roles="usuario, admin"/>
</tomcat-users>
```

Así, por ejemplo, para un recurso (URL) al que sólo puedan acceder roles de tipo *admin*, podrían acceder los usuarios *manuel* y *toni*. Notar también que los passwords están visibles en un fichero de texto fácilmente accesible por casi cualquiera, con lo que no es una buena forma de gestionar los passwords para una aplicación profesional.

JDBCRealm

Esta implementación almacena los datos de los usuarios en una base de datos con JDBC, lo cual es mucho más flexible y escalable que el mecanismo anterior y además no requiere el rearranque de Tomcat cada vez que se cambia algún dato. Para configurar Tomcat con esta implementación de *realm* se puede descomentar uno que ya viene definido en el `server.xml`, a nivel global dentro del elemento `Engine`. Los atributos de este *realm* indican cómo efectuar la conexión JDBC y cuáles son los campos de la base de datos que contienen *logins*, *passwords* y roles.

Atributo	Significado
`className`	clase Java que implementa este *realm*. Debe ser `org.apache.catalina.realm.JDBCRealm`
`connectionName`	nombre de usuario para la conexión JDBC
`connectionPassword`	password para la conexión JDBC
`connectionURL`	URL de la base de datos
`debug`	nivel de depuración. Por defecto 0 (ninguno). Valores más altos indican más detalle.
`digest`	Algoritmo de "digest" (puede ser SHA, MD2 o MD5). Por defecto es `cleartext`
`driverName`	clase Java que implementa el *driver* de la B.D.
`roleNameCol`	nombre del campo que almacena los roles
`userNameCol`	nombre del campo que almacena los *logins*
`userCredCol`	nombre del campo que almacena los *passwords*
`userRoleTable`	nombre de la tabla que almacena la relación entre *login* y roles
`userTable`	nombre de la tabla que almacena la relación entre *login* y *password*

2. Seguridad en aplicaciones web

2.1. Tipologías de seguridad y autentificación

Podemos tener básicamente dos motivos para proteger una aplicación web:

- Evitar que usuarios no autorizados accedan a determinados recursos.
- Prevenir que se acceda a los datos que se intercambian en una transferencia a lo largo de la red.

Para cubrir estos agujeros, un sistema de seguridad se apoya en tres aspectos importantes:

- **Autentificación**: medios para identificar a los elementos que intervienen en el acceso a recursos.
- **Confidencialidad**: asegurar que sólo los elementos que intervienen entienden el proceso de comunicación establecido.
- **Integridad**: verificar que el contenido de la comunicación no se modifica durante la transmisión.

Control de la seguridad

Desde el punto de vista de quién controla la seguridad en una aplicación web, existen dos formas de implantación:

- **Seguridad declarativa**: Aquella estructura de seguridad sobre una aplicación que es externa a dicha aplicación. Con ella, no tendremos que preocuparnos de gestionar la seguridad en ningún servlet, página JSP, etc, de nuestra aplicación, sino que el propio servidor Web se encarga de todo. Así, ante cada petición, comprueba si el usuario se ha autentificado ya, y si no le pide login y password para ver si puede acceder al recurso solicitado. Todo esto se realiza de forma transparente al usuario. Mediante el descriptor de la aplicación principalmente (fichero *web.xml* en Tomcat), comprueba la configuración de seguridad que queremos dar.
- **Seguridad programada**: Mediante la seguridad programada, son los servlets y páginas JSP quienes, al menos parcialmente, controlan la seguridad de la aplicación.

En este tema veremos la seguridad declarativa, que es la que puede configurar el administrador del servidor web, dejando para módulos posteriores la seguridad programada

Autentificación

Tenemos distintos tipos de autentificación que podemos emplear en una aplicación web:

- **Autentificación *basic***: Con HTTP se proporciona un mecanismo de autentificación básico, basado en cabeceras de autentificación para solicitar datos del usuario (el servidor) y para enviar los datos del usuario (el cliente). Esta autentificación no proporciona confidencialidad ni integridad, sólo se emplea una codificación Base64.

- **Autentificación *digest***: Existe una variante de lo anterior, la autentificación **digest**, donde, en lugar de transmitir el password por la red, se emplea un password codificado utilizando el método de encriptado MD5. Sin embargo, algunos servidores no soportan este tipo de autentificación.
- **Autentificación basada en formularios**: Con este tipo de autentificación, el usuario introduce su login y password mediante un formulario HTML (y no con un cuadro de diálogo, como las anteriores). El fichero descriptor contiene para ello entradas que indican la página con el formulario de autentificación y una página de error. Tiene el mismo inconveniente que la autentificación *basic*: el password se codifica con un mecanismo muy pobre.
- **Certificados digitales y SSL**: Con HTTP también se permite el uso de SSL y los certificados digitales, apoyados en los sistemas de criptografía de clave pública. Así, la capa SSL, trabajando entre TCP/IP y HTTP, asegura, mediante criptografía de clave pública, la integridad, confidencialidad y autentificación.

2.2. Autentificación basada en formularios

Veremos ahora con más profundidad la autentificación basada en formularios comentada anteriormente. Esta es la forma más comúnmente usada para imponer seguridad en una aplicación, puesto que se emplean **formularios HTML**.

El programador emplea el descriptor de despliegue para identificar los recursos a proteger, e indicar la página con el formulario a mostrar, y la página con el error a mostrar en caso de autentificación incorrecta. Así, un usuario que intente acceder a la parte restringida es redirigido automáticamente a la página del formulario, si no ha sido autentificado previamente. Si se autentifica correctamente accede al recurso, y si no se le muestra la página de error. Todo este proceso lo controla el servidor automáticamente.

Este tipo de autentificación no se garantiza que funcione cuando se emplea reescritura de URLs en el seguimiento de sesiones. También podemos incorporar SSL a este proceso, de forma que no se vea modificado el funcionamiento aparente del mismo.

Para utilizar la autentificación basada en formularios, se siguen los pasos que veremos a continuación. Sólo el primero es dependiente del servidor que se utilice.

1. Establecer los logins, passwords y roles

En este paso definiríamos un *realm* del modo que se ha explicado en apartados anteriores.

2. Indicar al servlet que se empleará autentificación basada en formularios, e indicar las páginas de formulario y error.

Se coloca para ello una etiqueta `<login-config>` en el descriptor de despliegue. Dentro, se emplean las subetiquetas:

- `<auth-method>` que en general puede valer:
  - `FORM`: para autentificación basada en formularios (como es el caso)

  - `BASIC`: para autentificación BASIC
  - `DIGEST`: para autentificación DIGEST
  - `CLIENT-CERT`: para SSL
- `<form-login-config>` que indica las dos páginas HTML (la del formulario y la de error) con las etiquetas:
  - `<form-login-page>` (para la de autentificación)
  - `<form-error-page>` (para la página de error).

Por ejemplo, podemos tener las siguientes líneas en el descriptor de despliegue:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
        ...
        <login-config>
                <auth-method>FORM</auth-method>
                <form-login-config>
                        <form-login-page>
                                /login.jsp
                        </form-login-page>
                        <form-error-page>
                                /error.html
                        </form-error-page>
                </form-login-config>
        </login-config>
        ...
</web-app>
```

3. Crear la página de login

El formulario de esta página debe contener campos para introducir el login y el password, que deben llamarse *j_username* y *j_password*. La acción del formulario debe ser *j_security_check*, y el METHOD = POST (para no mostrar los datos de identificación en la barra del explorador). Por ejemplo, podríamos tener la página:

```
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<body>
        <form action="j_security_check" METHOD="POST">
        <table>
        <tr>
                <td>
                        Login:<input type="text"
name="j_username"/>
                </td>
        </tr>
        <tr>
                <td>
                        Password:<input type="text"
name="j_password"/>
                </td>
```

```
        </tr>
        <tr>
                <td>
                        <input type="submit" value="Enviar"/>
                </td>
        </tr>
        </table>
        </form>
</body>
</html>
```

4. Crear la página de error

La página puede tener el mensaje de error que se quiera. Ante fallos de autentificación, se redirigirá a esta página con un código 401. Un ejemplo de página sería:

```
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<body>
        <h1>ERROR AL AUTENTIFICAR USUARIO</h1>
</body>
</html>
```

5. Indicar qué direcciones deben protegerse con autentificación

Para ello utilizamos etiquetas <security-constraint> en el descriptor de despliegue. Dichos elementos debe ir inmediatamente antes de *<login-config>*, y utilizan las subetiquetas:

- `<display-name>` para dar un nombre identificativo a emplear (opcional)
- `<web-resource-collection>` para especificar los patrones de URL que se protegen (requerido). Se permiten varias entradas de este tipo para especificar recursos de varios lugares. Cada uno contiene:
  - Una etiqueta `<web-resource-name>` que da un nombre identificativo arbitrario al recurso o recursos
  - Una etiqueta `<url-pattern>` que indica las URLs que deben protegerse
  - Una etiqueta `<http-method>` que indica el método o métodos HTTP a los que se aplicará la restricción (opcional)
  - Una etiqueta `<description>` con documentación sobre el conjunto de recursos a proteger (opcional)

  NOTA: este modo de restricción se aplica sólo cuando se accede al recurso directamente, no a través de arquitecturas MVC (Modelo-Vista-Controlador), con un *RequestDispatcher*. Es decir, si por ejemplo un servlet accede a una página JSP protegida, este mecanismo no tiene efecto, pero sí cuando se intenta a acceder a la página JSP directamente.
- `<auth-constraint>` indica los roles de usuario que pueden acceder a los recursos indicados (opcional) Contiene:
  - Uno o varios subelementos `<role-name>` indicando cada rol que tiene permiso de acceso. Si queremos dar permiso a todos los roles, utilizamos una etiqueta

<role-name></role-name>*.

- Una etiqueta <description> indicando la descripción de los mismos.

En teoría esta etiqueta es opcional, pero omitiéndola indicamos que ningún rol tiene permiso de acceso. Aunque esto puede parecer absurdo, recordar que este sistema sólo se aplica al acceso directo a las URLs (no a través de un modelo MVC), con lo que puede tener su utilidad.

Añadimos alguna dirección protegida al fichero que vamos construyendo:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
        <security-constraint>
                <web-resource-collection>
                        <web-resource-name>
                                Prueba
                        </web-resource-name>
                        <url-pattern>
                                /prueba/*
                        </url-pattern>
                </web-resource-collection>
                <auth-constraint>
                        <role-name>admin</role-name>
                        <role-name>subadmin</role-name>
                </auth-constraint>
        </security-constraint>

        <login-config>
                ...
</web-app>
```

En este caso protegemos todas las URLs de la forma *http//host/ruta_aplicacion/prueba/**, de forma que sólo los usuarios que tengan roles de *admin* o de *subadmin* podrán acceder a ellas.

Además,por cada role-name que hayamos usado hay que incluir una definición aparte. Esta es obligatoria pero algunos servidores la ignoran ,ya que parece un poco superflua:

```
<security-role>
  <description>administrador de la web</description>
  <role-name>admin</role-name>
</security-role>
<security-role>
  <description>el segundo de a bordo</description>
  <role-name>subadmin</role-name>
</security-role>
```

2.3. Autentificación basic

El método de autentificación basada en formularios tiene algunos inconvenientes: si el navegador no soporta cookies, el proceso tiene que hacerse mediante reescritura de URLs, con lo que no se garantiza el funcionamiento.

Por ello, una alternativa es utilizar el modelo de autentificación *basic* de HTTP, donde se emplea un cuadro de diálogo para que el usuario introduzca su login y password, y se emplea la cabecera *Authorization* de petición para recordar qué usuarios han sido autorizados y cuáles no. Una diferencia con respecto al método anterior es que es difícil entrar como un usuario distinto una vez que hemos entrado como un determinado usuario (habría que cerrar el navegador y volverlo a abrir).

Al igual que en el caso anterior, podemos utilizar SSL sin ver modificado el resto del esquema del proceso.

El método de autentificación *basic* consta de los siguientes pasos:

1. Establecer los logins, passwords y roles

Este paso es exactamente igual que el visto para la autentificación basada en formularios.

2. Indicar al servlet que se empleará autentificación BASIC, y designar los dominios

Se utiliza la misma etiqueta `<login-config>` vista antes, pero ahora una etiqueta `<auth-method>` con valor BASIC. Se emplea una subetiqueta `<realm-name>` para indicar qué dominio se empleará en la autorización. Por ejemplo:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
        ...
        <login-config>
                <auth-method>BASIC</auth-method>
                <realm-name>dominio</realm-name>
        </login-config>
        ...
</web-app>
```

3. Indicar qué direcciones deben protegerse con autentificación

Este paso también es idéntico al visto en la autentificación basada en formularios.

3. Autentificación y confidencialidad mediante SSL

3.1. Configurar SSL en Tomcat

SSL (Secure Socket Layer) es una tecnología que permite que servidores y clientes web se comuniquen a través de un canal seguro. Los datos que se intercambian cliente y

servidor están encriptados, asegurando la confidencialidad. Además, cuando se inicia la conexión, el servidor le presenta al cliente un certificado que garantiza su identidad, con lo que se garantiza también la autentificación del servidor, suponiendo que dicho certificado está firmado por una autoridad certificadora conocida (como Verisign). El servidor puede solicitar también al cliente un certificado para autentificar su identidad, aunque esto es menos común.

La instalación por defecto de Tomcat no soporta SSL, y tenemos que dar algunos pasos extra para añadir esta funcionalidad. Suponiendo que disponemos como mínimo de la versión 1.4 del Java SDK, estos pasos se detallan a continuación.

Crear un certificado de clave pública. Los servidores basados en SSL utilizan certificados X509 para validar que los usuarios son quienes dicen ser. En el mundo real, el certificado necesita ser firmado por una autoridad de confianza, como por ejemplo Verisign. Pero para realizar pruebas, un certificado auto-firmado es suficiente. Para hacer esto ejecutamos:

```
keytool -genkey -alias tomcat -keyalg RSA
```

El sistema nos pedirá cierta información: primero un password. Este mismo password lo deberemos indicar más adelante en el fichero server.xml. Después alguna información adicional como nombre y apellidos (para un certificado del servidor, esto debe ser el nombre del servidor, no el nuestro), organización, localización, y finalmente otro password que debe ser el mismo que el primero que pusimos.

Con todo esto, el sistema crea un fichero llamado .keystore en nuestro directorio por defecto (/home/nuestro-usuario en Unix, o C:\Documents and Settings\nuestro-usuario en Windows XP, por ejemplo). Podemos utilizar el parámetro -keystore al generar el certificado para indicar dónde guardar el fichero y con qué nombre.

```
keytool -genkey -alias tomcat -keyalg RSA -keystore /dir/keystore
```

Opcionalmente, podemos copiar el certificado en el directorio inicial de Tomcat, que es el directorio donde se espera encontrar el certificado por defecto, aunque este aspecto es configurable a través del fichero server.xml.

Descomentar la conexión SSL en el fichero de configuración. Tenemos finalmente que editar el fichero conf/server.xml de Tomcat y buscar un elemento Connector que originalmente está comentado y que viene indicado en el fichero como que representa un "SSL HTTP/1.1 Connector" o similar. Una vez localizado, descomentamos ese código. Luego, podemos cambiar algunos atributos de configuración

- Normalmente se cambiará el puerto al valor por defecto para SSL, que es 443 (hay que tener en cuenta que el uso de puertos por debajo de 1024 en muchos sistemas requiere permisos de administrador). Si lo cambiamos aquí, luego lo tendremos que cambiar también en el atributo redirectPort del Connector que maneja la conexión no SSL
- En el elemento Factory que hay dentro del Connector, puede que necesitemos

configurar los siguientes atributos:

- className: la clase que implementa el SocketFactory. Dejar el valor por defecto
- clientAuth: dejar en true si queremos que todos los clientes presenten un certificado para utilizar el socket. En nuestro caso lo dejaríamos en false, ya que el que se autentifica es el servidor, no los clientes.
- keystoreFile: indicando el nombre del fichero "keystore" generado anteriormente (en caso de que no se encuentre en el directorio por defecto del usuario). Podemos indicar rutas absolutas, o relativas al directorio raíz de Tomcat.
- keystorePass: indicando el password que hayamos especificado al crear el fichero "keystore".
- protocol: protocolo de encriptado/desencriptado. Dejar el valor por defecto.

Cambiar la entrada del conector principal en el fichero de configuración para que utilice el puerto para redirecciones SSL

Como hemos dicho anteriormente, si hemos cambiado el puerto SSL, deberemos reflejar ese cambio en el conector principal en el atributo redirectPort.

Reiniciar y probar el servidor. Para probar la conexión SSL accedemos a la página raíz de Tomcat utilizando HTTPS (HTTP + SSL):

```
https://localhost:8443/
```

Tras esto, en nuestro navegador veremos algunas pantallas iniciales de advertencia indicando que el sitio utiliza certificados y que el firmante del certificado no es una autoridad reconocida (lógico, ya que somos nosotros mismos), y después ya veremos la página de arranque de Tomcat.

3.2. Configurar SSL en la aplicación web

Podemos incorporar SSL a los métodos de autentificación basados en seguridad declarativa vistos antes, sin más que incluyendo, dentro de la etiqueta `<security-constraint>`, una subetiqueta `<user-data-constraint>`. No necesitamos modificar el resto de la estructura para utilizar SSL, lo único que se necesita es adaptar el servidor Web para que soporte SSL, y añadir esta etiqueta, que contiene:

- Una subetiqueta <transport-guarantee>, que tendrá como posibles valores NONE, INTEGRAL o CONFIDENTIAL.
  - Con NONE no se imponen restricciones en el protocolo de comunicación
  - Con INTEGRAL se impone una variedad de comunicación que previene el hecho de modificar los datos mientras se envían, sin que se detecte dicho cambio.
  - Con CONFIDENTIAL se indica que los datos se envían para evitar que cualquiera que los intercepte pueda leerlos.
- Un elemento <description> opcional

La API de servlets proporciona, además, una forma de requerir a los clientes que se

identifiquen con un certificado. Se puede proporcionar un valor de CLIENT-CERT a la etiqueta <auth-method> de <login-config>. Sin embargo, sólo los servidores que soporten J2EE por completo soportarán esta posibilidad. Incluso los servidores que soportan las versiones 2.3 de servlets y 1.2 de JSP no tienen por qué soportar SSL, con lo que esta característica puede no ser portable.