Chapter 8: Programmatic Security

Topics in This Chapter

- Combining container-managed and programmatic security
- Using the `isUserInRole` method
- Using the `getRemoteUser` method
- Using the `getUserPrincipal` method
- Programmatically controlling all aspects of security
- Using SSL with programmatic security

Chapter 8

Chapter 7 introduced two fundamental aspects of Web application security:

1. **Preventing unauthorized users from accessing sensitive data.** This process involves *access restriction* (identifying which resources need protection and who should have access to them) and *authentication* (identifying the user to determine if they are one of the authorized ones). This aspect applies to virtually all secure applications; even intranets at locations with physical access controls usually require some sort of user authentication.
2. **Preventing attackers from stealing network data while it is in transit.** This process involves the use of Secure Sockets Layer (SSL) to encrypt the traffic between the browser and the server. This capability is generally reserved for particularly sensitive applications or particularly sensitive pages within a larger application.

There are two general strategies for implementing these security aspects: *declarative security* and *programmatic security*.

With declarative security, the topic of the previous chapter, none of the individual servlets or JSP pages need any security-aware code. Instead, both of the major security aspects are handled by the server. To prevent unauthorized access, you use the Web application deployment descriptor (*web.xml*) to declare that certain URLs need protection. You also designate the authentication method that the server should use to identify users. At request time, the server automatically prompts users for usernames and passwords when they try to access restricted resources, automat-

ically checks the results against a predefined set of usernames and passwords, and automatically keeps track of which users have previously been authenticated. This process is completely transparent to the servlets and JSP pages. To safeguard network data, you use the deployment descriptor to stipulate that certain URLs should only be accessible with SSL. If users try to use a regular HTTP connection to access one of these URLs, the server automatically redirects them to the HTTPS (SSL) equivalent.

Declarative security is all well and good. In fact, it is by far the most common approach to Web application security. But, what if you want your servlets to be completely independent of any server-specific settings such as password files? Or, what if you want to let users in various roles access a particular resource but customize the data depending on the role that they are in? Or, what if you want to authenticate users other than by requiring an exact match from a fixed set of usernames and passwords? That's where programmatic security comes in.

With programmatic security, the topic of this chapter, protected servlets and JSP pages at least partially manage their own security. To prevent unauthorized access, each servlet or JSP page must either authenticate the user or verify that the user has been authenticated previously. Even after the servlet or JSP page grants access to a user, it can still customize the results for different individual users or categories of users. To safeguard network data, each servlet or JSP page has to check the network protocol used to access it. If users try to use a regular HTTP connection to access one of these URLs, the servlet or JSP page must manually redirect them to the HTTPS (SSL) equivalent.

8.1 Combining Container-Managed and Programmatic Security

Declarative security is very convenient: you set up usernames, passwords, access mechanisms (HTML forms vs. BASIC authentication) and transport-layer requirements (SSL vs. normal HTTP), all without putting any security-related code in any of the individual servlets or JSP pages. However, declarative security provides only two levels of access for each resource: allowed and denied. Declarative security provides no options to permit resources to customize their output depending on the username or role of the client that accesses them.

It would be nice to provide this customization without giving up the convenience of container-managed security for the usernames, passwords, and roles as would be required if a servlet or JSP page completely managed its own security (as in Section 8.3). To support this type of hybrid security, the servlet specification provides three methods in `HttpServletRequest`:

- **`isUserInRole`.** This method determines if the currently authenticated user belongs to a specified role. For example, given the usernames, passwords, and roles of Listings 7.1 and 7.2 (Section 7.1), if the client has successfully logged in as user `valjean`, the following two expressions would return `true`.

  ```
  request.isUserInRole("lowStatus")
  request.isUserInRole("nobleSpirited")
  ```

  Tests for all other roles would return `false`. If no user is currently authenticated (e.g., if authorization failed or if `isUserInRole` is called from an unrestricted page and the user has not yet accessed a restricted page), `isUserInRole` returns `false`. In addition to the standard security roles given in the password file, you can use the `security-role-ref` element to define aliases for the standard roles. See the next subsection for details.
- **`getRemoteUser`.** This method returns the name of the current user. For example, if the client has successfully logged in as user `valjean`, `request.getRemoteUser()` would return `"valjean"`. If no user is currently authenticated (e.g., if authorization failed or if `isUserInRole` is called from an unrestricted page and the user has not yet accessed a restricted page), `getRemoteUser` returns `null`.
- **`getUserPrincipal`.** This method returns the current username wrapped inside a `java.security.Principal` object. The `Principal` object contains little information beyond the username (available with the `getName` method). So, the main reason for using `getUserPrincipal` in lieu of `getRemoteUser` is to be compatible with preexisting security code (the `Principal` class is not specific to the servlet and JSP API and has been part of the Java platform since version 1.1). If no user is currently authenticated, `getUserPrincipal` returns `null`.

It is important to note that this type of programmatic security does not negate the benefits of container-managed security. With this approach, you can still set up usernames, passwords, and roles by using your server's mechanisms. You still use the `login-config` element to tell the server whether you are using form-based or BASIC authentication. If you choose form-based authentication, you still use an HTML form with an `ACTION` of `j_security_check`, a textfield named `j_username`, and a password field named `j_password`. Unauthenticated users are still automatically sent to the page containing this form, and the server still automatically keeps track of which users have been authenticated. You still use the `security-constraint` element to designate the URLs to which the access restrictions apply. You still use the `user-data-constraint` element to specify that certain URLs require SSL. For details on all of these topics, see Section 7.1.

Security Role References

The `security-role-ref` subelement of `servlet` lets you define servlet-specific synonyms for existing role names. This element should contain three possible subelements: `description` (optional descriptive text), `role-name` (the new synonym), and `role-link` (the existing security role).

For instance, suppose that you are creating an online bookstore and your server's password file stipulates that user `marty` is in role `author`. However, you want to reuse a servlet of type `BookInfo` (in the `catalog` package) that was created elsewhere. The problem is that this servlet calls the role `writer`, not `author`. Rather than modifying the password file, you can use `security-role-ref` to provide `writer` as an alias for `author`.

Suppose further that you have a servlet of class `EmployeeData` (in the `hr` package) that provides one type of information to a `goodguy` and another type to a `meanie`. You want to use this servlet with the password file defined in Listings 7.1 and 7.2 (Section 7.1) that assign users to the `nobleSpirited` and `meanSpirited` roles. To accomplish this task, you can use `security-role-ref` to say that `isUserInRole("goodguy")` should return `true` for the same users that `isUserInRole("nobleSpirited")` already would. Similarly, you can use `security-role-ref` to say that `isUserInRole("meanie")` should return `true` for the same users that `isUserInRole("meanSpirited")` would.

Listing 8.1 shows a deployment descriptor that accomplishes both of these tasks.

Listing 8.1 *web.xml* (Excerpt illustrating security role aliases)

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

<web-app>
  <!-- ... -->
  <servlet>
    <servlet-name>BookInformation</servlet-name>
    <servlet-class>catalog.BookInfo</servlet-class>
    <security-role-ref>
      <role-name>writer</role-name> <!-- New alias. -->
      <role-link>author</role-link> <!-- Preexisting role. -->
    </security-role-ref>
  </servlet>
```

Listing 8.1 *web.xml* (Excerpt illustrating security role aliases) *(continued)*

```
  <servlet>
    <servlet-name>EmployeeInformation</servlet-name>
    <servlet-class>hr.EmployeeData</servlet-class>
    <security-role-ref>
      <role-name>goodguy</role-name>       <!-- New. -->
      <role-link>nobleSpirited</role-link> <!-- Preexisting. -->
    </security-role-ref>
    <security-role-ref>
      <role-name>meanie</role-name>       <!-- New. -->
      <role-link>meanSpirited</role-link> <!-- Preexisting. -->
    </security-role-ref>
  </servlet>
  <!-- ... -->
  <security-constraint>...</security-constraint>
  <login-config>...</login-config>
  <!-- ... -->
</web-app>
```

8.2 Example: Combining Container-Managed and Programmatic Security

Listing 8.2 presents a JSP page that augments the internal Web site for hot-dot-com.com that is introduced in Section 7.4. The page shows plans for employee pay. Because of entries in *web.xml* (Listing 8.3), the page can be accessed only by users in the `employee` or `executive` roles. Although both groups can access the page, they see substantially different results. In particular, the planned pay scales for executives is hidden from the normal employees.

Figure 8–1 shows the page when it is accessed by user `gates` or `ellison` (both in the `employee` role; see Listing 7.25). Figure 8–2 shows the page when it is accessed by user `mcnealy` (in the `executive` role). Remember that BASIC security provides no simple mechanism for changing your username once you are validated (see Section 7.3). So, for example, switching from user `gates` to user `mcnealy` requires you to quit and restart your browser.

Listing 8.2 *employee-pay.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Compensation Plans</TITLE>
<LINK REL=STYLESHEET
      HREF="company-styles.css"
      TYPE="text/css">
</HEAD>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">Compensation Plans</TABLE>
<P>
Due to temporary financial difficulties, we are scaling
back our very generous plans for salary increases. Don't
worry, though: your valuable stock options more than
compensate for any small drops in direct salary.

<H3>Regular Employees</H3>
Pay for median-level employee (Master's degree, eight year's
experience):
<UL>
  <LI><B>2002:</B> $50,000.
  <LI><B>2003:</B> $30,000.
  <LI><B>2004:</B> $25,000.
  <LI><B>2005:</B> $20,000.
</UL>

<% if (request.isUserInRole("executive")) { %>
<H3>Executives</H3>
Median pay for corporate executives:
<UL>
  <LI><B>2002:</B> $500,000.
  <LI><B>2003:</B> $600,000.
  <LI><B>2004:</B> $700,000.
  <LI><B>2005:</B> $800,000.
</UL>
<% } %>
</BODY>
</HTML>
```

Listing 8.3 *web.xml* (For augmented hotdotcom intranet)

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

<web-app>
  <!-- A servlet that redirects users to the home page. -->
  <servlet>
    <servlet-name>Redirector</servlet-name>
    <servlet-class>hotdotcom.RedirectorServlet</servlet-class>
  </servlet>

  <!-- Turn off invoker. Send requests to index.jsp. -->
  <servlet-mapping>
    <servlet-name>Redirector</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>

  <!-- If URL gives a directory but no filename, try index.jsp
       first and index.html second. If neither is found,
       the result is server specific (e.g., a directory
       listing). -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

  <!-- Protect financial plan. Employees or executives. -->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Financial Plan</web-resource-name>
      <url-pattern>/financial-plan.html</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>employee</role-name>
      <role-name>executive</role-name>
    </auth-constraint>
  </security-constraint>
```

Listing 8.3 *web.xml* (For augmented hotdotcom intranet) *(continued)*

```
  <!-- Protect business plan. Executives only. -->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Business Plan</web-resource-name>
      <url-pattern>/business-plan.html</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>executive</role-name>
    </auth-constraint>
  </security-constraint>

  <!-- Protect compensation plan. Employees or executives. -->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Compensation Plan</web-resource-name>
      <url-pattern>/employee-pay.jsp</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>employee</role-name>
      <role-name>executive</role-name>
    </auth-constraint>
  </security-constraint>

  <!-- Tell the server to use BASIC authentication. -->
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Intranet</realm-name>
  </login-config>
</web-app>
```

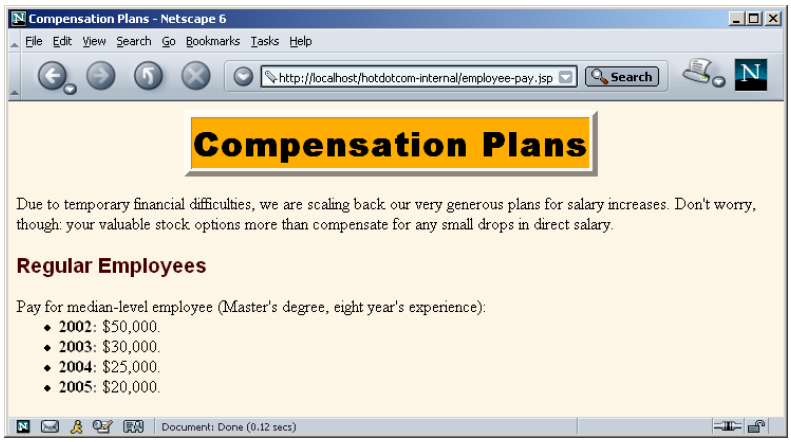

Figure 8–1 The *employee-pay.jsp* page when accessed by a user who is in the `employee` role.

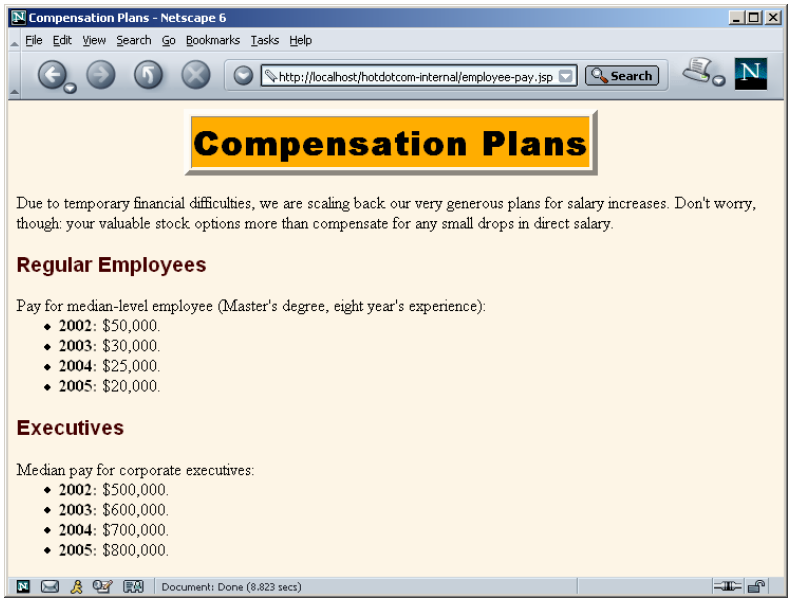

Figure 8–2 The *employee-pay.jsp* page when accessed by a user who is in the `executive` role.

8.3 Handling All Security Programmatically

Declarative security (Chapter 7) offers a number of advantages to the developer. Chief among them is the fact that individual servlets and JSP pages need no security-conscious code: the container (server) handles authentication in a manner that is completely transparent to the individual resources. For example, you can switch from form-based authentication to BASIC authentication or from regular HTTP connections to encrypted HTTPS connections, all without any changes to the individual servlets or JSP pages.

Even when you want a bit more control than just "access allowed" or "access denied," it is convenient to let the server maintain and process the usernames and passwords, as discussed in Section 8.1.

However, the convenience of container-managed security comes at a price: it requires a server-specific component. The method for setting up usernames, passwords, and user roles is not standardized and thus is not portable across different servers. In most situations, this disadvantage is outweighed by the faster and simpler servlet and JSP development process that results from leaving some or all of the authorization tasks to the server. In some cases, however, you might want a servlet or JSP page to be entirely self-contained with no dependencies on server-specific settings or even *web.xml* entries. Although this approach requires a lot more work, it means that the servlet or JSP page can be ported from server to server with much less effort than with container-managed security. Furthermore, it lets the servlet or JSP page use username and password schemes other than an exact match to a preconfigured list.

HTTP supports two varieties of authentication: BASIC and DIGEST. Few browsers support DIGEST, so I'll concentrate on BASIC here.

Here is a summary of the steps involved for BASIC authorization.

1. **Check whether there is an `Authorization` request header.** If there is no such header, go to Step 5.
2. **Get the encoded username/password string.** If there is an `Authorization` header, it should have the following form:

   ```
   Authorization: Basic encodedData
   ```

   Skip over the word `Basic`—the remaining part is the username and password represented in base64 encoding.
3. **Reverse the base64 encoding of the username/password string.** Use the `decodeBuffer` method of the `BASE64Decoder` class. This method call results in a string of the form `username:password`. The `BASE64Decoder` class is bundled with the JDK; in JDK 1.3 it can be found in the `sun.misc` package in *jdk_install_dir/jre/lib/rt.jar*.

4. **Check the username and password.** The most common approach is to use a database or a file to obtain the real usernames and passwords. For simple cases, it is also possible to place the password information directly in the servlet. In such a case, remember that access to the servlet source code or class file provides access to the passwords. If the incoming username and password match one of the reference username/password pairs, return the page. If not, go to Step 5. With this approach you can provide your own definition of "match." With container-managed security, you cannot.
5. **When authentication fails, send the appropriate response to the client.** Return a 401 (`Unauthorized`) response code and a header of the following form:

   ```
   WWW-Authenticate: BASIC realm="some-name"
   ```

   This response instructs the browser to pop up a dialog box telling the user to enter a name and password for `some-name`, then to reconnect with that username and password embedded in a single base64 string inside the `Authorization` header.

If you care about the details, base64 encoding is explained in RFC 1521. To retrieve RFCs, start at *http://www.rfc-editor.org/* to get a current list of the RFC archive sites. However, there are probably only two things you need to know about base64 encoding.

First, it is not intended to provide security, since the encoding can be easily reversed. So, base64 encoding does not obviate the need for SSL (see Section 7.5) to thwart attackers who might be able to snoop on your network connection (no easy task unless they are on your local subnet). SSL, or Secure Sockets Layer, is a variation of HTTP where the entire stream is encrypted. It is supported by many commercial servers and is generally invoked by use of *https* in the URL instead of *http*. Servlets can run on SSL servers just as easily as on standard servers, and the encryption and decryption are handled transparently before the servlets are invoked. See Sections 7.1 and 8.6 for examples.

The second point you should know about base64 encoding is that Sun provides the `sun.misc.BASE64Decoder` class, distributed with JDK 1.1 and later, to decode strings that were encoded with base64. In JDK 1.3 it can be found in the `sun.misc` package in *jdk_install_dir/jre/lib/rt.jar*. Just be aware that classes in the `sun` package hierarchy are not part of the official language specification and thus are not guaranteed to appear in all implementations. So, if you use this decoder class, make sure that you explicitly include the class file when you distribute your application. One possible approach is to make the class available to all Web applications on your server and then to explicitly record the fact that your applications depend on it. For details on this process, see Section 4.4 (Recording Dependencies on Server Libraries).

8.4 Example: Handling All Security Programmatically

Listing 8.4 shows a servlet that generates hot stock recommendations. If it were made freely available on the Web, it would put half the financial advisors out of business. So, it needs to be password protected, available only to people who have paid the very reasonable $2000 access fee.

Furthermore, the servlet needs to be as portable as possible because ISPs keep shutting it down (they claim fraud, but no doubt they are really being pressured by the financial services companies that the servlet outperforms). So, it uses complete programmatic security and is entirely self-contained: absolutely no changes or server-specific customizations are required to move the servlet from system to system.

Finally, requiring an exact match against a static list of usernames and passwords (as is required in container-managed security) is too limiting for this application. So, the servlet uses a custom algorithm (see the `areEqualReversed` method) for determining if an incoming username and password are legal.

Figure 8–3 shows what happens when the user first tries to access the servlet. Figure 8–4 shows the result of a failed authorization attempt; Figure 8–5 shows what happens if the user gives up at that point. Figure 8–6 shows the result of successful authorization.

Listing 8.4 *StockTip.java*

```
package stocks;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import sun.misc.BASE64Decoder;

/** Servlet that gives very hot stock tips. So hot that
 *  only authorized users (presumably ones who have paid
 *  the steep financial advisory fee) can access the servlet.
 */

public class StockTip extends HttpServlet {

  /** Denies access to all users except those who know
   *  the secret username/password combination.
   */
```

Listing 8.4 *StockTip.java (continued)*

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
  String authorization = request.getHeader("Authorization");
  if (authorization == null) {
    askForPassword(response);
  } else {
    // Authorization headers looks like "Basic blahblah",
    // where blahblah is the base64 encoded username and
    // password. We want the part after "Basic ".
    String userInfo = authorization.substring(6).trim();
    BASE64Decoder decoder = new BASE64Decoder();
    String nameAndPassword =
      new String(decoder.decodeBuffer(userInfo));
    // Decoded part looks like "username:password".
    int index = nameAndPassword.indexOf(":");
    String user = nameAndPassword.substring(0, index);
    String password = nameAndPassword.substring(index+1);
    // High security: username must be reverse of password.
    if (areEqualReversed(user, password)) {
      showStock(request, response);
    } else {
      askForPassword(response);
    }
  }
}

// Show a Web page giving the symbol of the next hot stock.

private void showStock(HttpServletRequest request,
                       HttpServletResponse response)
    throws ServletException, IOException {
      response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  String docType =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
    "Transitional//EN\">\n";
  out.println(docType +
              "<HTML>\n" +
              "<HEAD><TITLE>Hot Stock Tip!</TITLE></HEAD>\n" +
              "<BODY BGCOLOR=\"#FDF5E6\">\n" +
              "<H1>Today's Hot Stock:");
```

Listing 8.4 *StockTip.java (continued)*

```
    for(int i=0; i<3; i++) {
      out.print(randomLetter());
    }
    out.println("</H1>\n" +
                "</BODY></HTML>");
  }

  // If no Authorization header was supplied in the request.

  private void askForPassword(HttpServletResponse response) {
    response.setStatus(response.SC_UNAUTHORIZED); // I.e., 401
    response.setHeader("WWW-Authenticate",
                       "BASIC realm=\"Insider-Trading\"");
  }

  // Returns true if s1 is the reverse of s2.
  // Empty strings don't count.

  private boolean areEqualReversed(String s1, String s2) {
    s2 = (new StringBuffer(s2)).reverse().toString();
    return((s1.length() > 0) && s1.equals(s2));
  }

  private final String ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

  // Returns a random number from 0 to n-1 inclusive.

  private int randomInt(int n) {
    return((int)(Math.random() * n));
  }

  // A random letter from the alphabet.

  private char randomLetter() {
    return(ALPHABET.charAt(randomInt(ALPHABET.length())));
  }
}
```

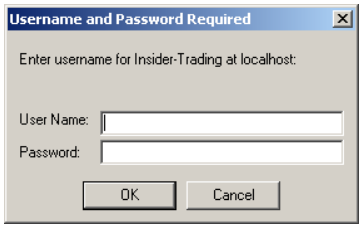

Figure 8–3 When the browser first receives the 401 (`Unauthorized`) status code, it opens a dialog box to collect the username and password.

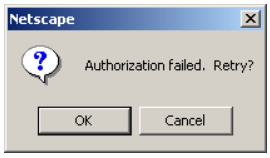

Figure 8–4 When the browser receives the 401 (`Unauthorized`) status code on later attempts, it indicates that authorization failed. Netscape 6 and Internet Explorer indicate authorization failure by showing the original dialog box with the previously entered username and an empty password field.

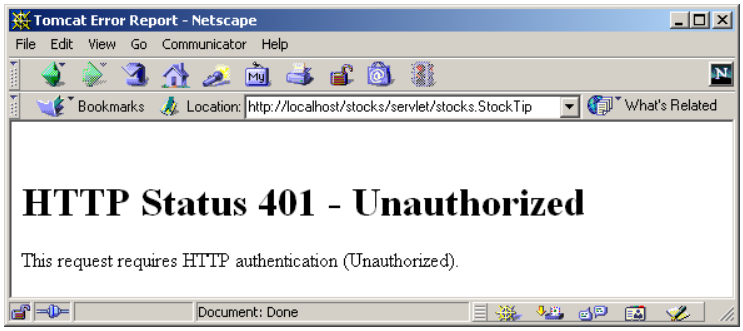

Figure 8–5 Result of cancelled authorization attempt with Tomcat—Tomcat returns an error page along with the 401 (`Unauthorized`) status code. JRun and ServletExec omit the error page in this case.

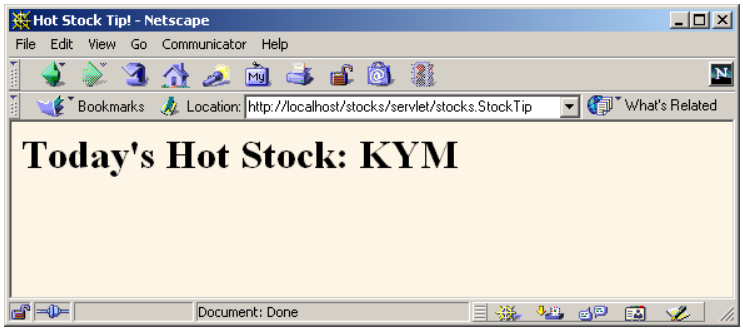

Figure 8–6 Result of successful authorization attempt. Invest now!

8.5 Using Programmatic Security with SSL

SSL can be used with security that is entirely servlet managed, just as it can be with container-managed security (see Section 7.1). As is typical with servlet-managed security, this approach is more portable but requires significantly more effort.

The use of SSL in programmatic security may require one or more of the following capabilities not needed in normal programmatic security.

- Determining if SSL is in use.
- Redirecting non-SSL requests.
- Discovering the number of bits in the key.
- Looking up the encryption algorithm.
- Accessing client X509 certificates.

Details on these capabilities follow.

Determining If SSL Is in Use

The `ServletRequest` interface provides two methods that let you find out if SSL is in use. The `getScheme` method returns `"http"` for regular requests and `"https"` for SSL requests. The `isSecure` method returns `false` for regular requests and `true` for SSL requests.

Redirecting Non-SSL Requests

With container-managed security, you can use the `transport-guarantee` subelement of `user-data-constraint` to ensure that the server redirects regular (*http*) requests to the SSL (*https*) equivalent. See Section 7.1 for details.

In programmatic security, you might want to explicitly do what the server automatically does with container-managed security. Once you have a URL, redirection is straightforward: use `response.sendRedirect` (Section 2.7).

The difficulty is in generating the URL in the first place. Unfortunately, there is no built-in method that says "give me the complete incoming URL with *http* changed to *https*." So, you have to call `request.getRequestURL` to get the main URL, change *http* to *https* manually, then tack on any form data by using `request.getQueryString`. You pass that result to `response.sendRedirect`.

Even this tedious manual approach runs some portability risks. For example: what if the server is running SSL on a port other than 443 (the default SSL port)? In such a case, the approach outlined here redirects to the wrong port. Unfortunately, there is no general solution to this problem; you simply have to know something about how the server is configured in order to redirect to a nonstandard SSL port. However, since you have to know that the server supports SSL in the first place, this additional burden is not too onerous.

Discovering the Number of Bits in the Key

Suppose that you have a servlet or JSP page that lets authorized users access your company's financial records. You might want to ensure that the most sensitive data is only sent to users that have the strongest (128-bit) level of encryption. Users whose browsers use comparatively weak 40-bit keys should be denied access. To accomplish this task, you need to be able to discover the level of encryption being used.

In version 2.3 of the servlet API, SSL requests automatically result in an attribute named `javax.servlet.request.key_size` being placed in the request object. You can access it by calling `request.getAttribute` with the specified name. The value is an `Integer` that tells you the length of the encryption key. However, since the return type of `getAttribute` is `Object`, you have to perform a typecast to `Integer`. In version 2.2 and earlier, there was no portable way to determine the key size. So, be sure to check if the result is `null` in order to handle non-SSL requests and SSL requests in servers compatible only with version 2.2 of the servlet API. Here is a simple example.

```
String keyAttribute = "javax.servlet.request.key_size";
Integer keySize =
  (Integer)request.getAttribute(keyAttribute);
if (keySize == null) { ... }
```

Looking Up the Encryption Algorithm

In version 2.3 of the servlet API, SSL requests also result in an attribute named `javax.servlet.request.cipher_suite` being placed in the request object. You can access it by calling `request.getAttribute` with the specified name. The value is a `String` that describes the encryption algorithm being used. However, since the return type of `getAttribute` is `Object`, you have to perform a typecast to `String`. Be sure to check if the result is `null` in order to handle non-SSL requests and SSL requests in servers compatible only with version 2.2 of the servlet API. Here is a simple example.

```
String cipherAttribute = "javax.servlet.request.cipher_suite";
String cipherSuite =
  (String)request.getAttribute(cipherAttribute);
if (cipherSuite == null) { ... }
```

Accessing Client X509 Certificates

Rather than using a simple username and password, some browsers permit users to authenticate themselves with X509 certificates. X509 certificates are discussed in RFC 1421. To retrieve RFCs, start at *http://www.rfc-editor.org/* to get a current list of the RFC archive sites.

If the client authenticates himself with an X509 certificate, that certificate is available by means of the `javax.servlet.request.X509Certificate` attribute of the request object. This attribute is available in both version 2.2 and 2.3 of the servlet API. The value is an object of type `java.security.cert.X509Certificate` that contains exhaustive information about the certificate. However, since the return type of `getAttribute` is `Object`, you have to perform a typecast to `X509Certificate`. Be sure to check if the result is `null` in order to handle non-SSL requests and SSL requests that include no certificate. A simple example follows.

```
String certAttribute = "javax.servlet.request.X509Certificate";
X509Certificate certificate =
  (X509Certificate)request.getAttribute(certAttribute);
if (certificate == null) { ... }
```

Once you have an X509 certificate, you can look up the issuer's distinguished name, the serial number, the raw signature value, the public key, and a number of other pieces of information. For details, see *http://java.sun.com/j2se/1.3/docs/api/java/security/cert/X509Certificate.html*.

8.6 Example: Programmatic Security and SSL

Listing 8.5 presents a servlet that redirects non-SSL requests to a URL that is identical to the URL of the original request except that *http* is changed to *https*. When an SSL request is received, the servlet presents a page that displays information on the URL, query data, key size, encryption algorithm, and client certificate. Figures 8–7 and 8–8 show the results.

In a real application, make sure that you redirect users when they access the servlet or JSP page that contains the form that *collects* the data. Once users submit sensitive data to an ordinary non-SSL URL, it is too late to redirect the request: attackers with access to the network traffic could have already obtained the data.

Listing 8.5 *SecurityInfo.java*

```
package moreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.security.cert.*; // For X509Certificate

/** Servlet that prints information on SSL requests. Non-SSL
 *  requests get redirected to SSL.
 */

public class SecurityInfo extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    // Redirect non-SSL requests to the SSL equivalent.
    if (request.getScheme().equalsIgnoreCase("http")) {
      String origURL = request.getRequestURL().toString();
      String newURL = httpsURL(origURL);
      String formData = request.getQueryString();
      if (formData != null) {
        newURL = newURL + "?" + formData;
      }
      response.sendRedirect(newURL);
    } else {
      String currentURL = request.getRequestURL().toString();
      String formData = request.getQueryString();
```

Listing 8.5 *SecurityInfo.java (continued)*

```
PrintWriter out = response.getWriter();
String docType =
  "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
  "Transitional//EN\">\n";
String title = "Security Info";
out.println
  (docType +
   "<HTML>\n" +
   "<HEAD><TITLE>" + title +
   "</TITLE></HEAD>\n" +
   "<BODY BGCOLOR=\"#FDF5E6\">\n" +
   "<H1>" + title + "</H1>\n" +
   "<UL>\n" +
   "  <LI>URL: " + currentURL + "\n" +
   "  <LI>Data: " + formData);
boolean isSecure = request.isSecure();
if (isSecure) {
  String keyAttribute =
    "javax.servlet.request.key_size";
  // Available only with servlets 2.3
  Integer keySize =
    (Integer)request.getAttribute(keyAttribute);
  String sizeString =
    replaceNull(keySize, "Unknown");
  String cipherAttribute =
    "javax.servlet.request.cipher_suite";
  // Available only with servlets 2.3
  String cipherSuite =
    (String)request.getAttribute(cipherAttribute);
  String cipherString =
    replaceNull(cipherSuite, "Unknown");
  String certAttribute =
    "javax.servlet.request.X509Certificate";
  // Available with servlets 2.2 and 2.3
  X509Certificate certificate =
    (X509Certificate)request.getAttribute(certAttribute);
  String certificateString =
    replaceNull(certificate, "None");
  out.println
    ("  <LI>SSL: true\n" +
     "  <UL>\n" +
     "    <LI>Key Size: " + sizeString + "\n" +
     "    <LI>Cipher Suite: " + cipherString + "\n" +
     "    <LI>Client Certificate: " +
     certificateString + "\n" +
     "  </UL>");
}
```

Listing 8.5 *SecurityInfo.java (continued)*

```
      out.println
        ("</UL>\n" +
         "</BODY></HTML>");
    }
  }

  // Given http://blah, return https://blah.

  private String httpsURL(String origURL) {
    int index = origURL.indexOf(":");
    StringBuffer newURL = new StringBuffer(origURL);
    newURL.insert(index, 's');
    return(newURL.toString());
  }

  // If the first argument is null, return the second argument.
  // Otherwise, convert first argument to a String and
  // return that String.

  private String replaceNull(Object obj, String fallback) {
    if (obj == null) {
      return(fallback);
    } else {
      return(obj.toString());
    }
  }
}
```

Figure 8–7 New-certificate page for Internet Explorer. View and import the certificate to suppress future warnings. For details on creating self-signed certificates for use with Tomcat, see Section 7.5. Again, self-signed certificates would not be trusted in real-world applications; they are for testing purposes only.

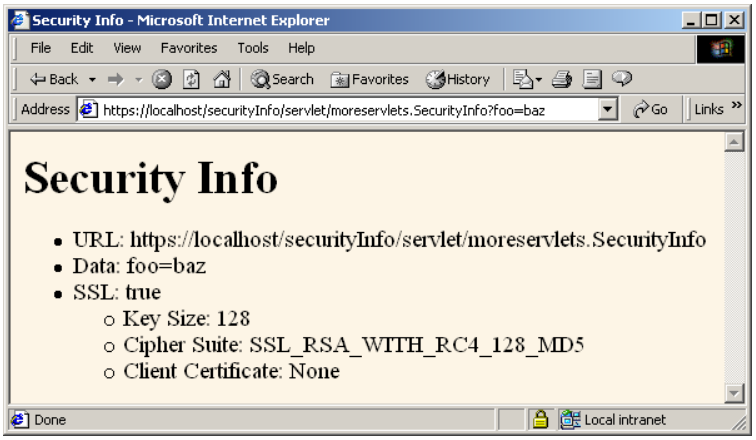

Figure 8–8 Result of the `SecurityInfo` servlet.