5

Security

CERTIFICATION OBJECTIVES

- Security Mechanisms
- Deployment Descriptor Security Declarations
- Authentication Types

✓ Two-Minute Drill

Q&A Self Test

Up until now, we have considered the essential mechanics of servlet applications. In this chapter you will explore how you can attach a layer of security to the web applications that you have learned about so far.

This chapter will take you through the three security objectives for the SCWCD exam. The first section is devoted to the "simple" definition of terms. It's straightforward enough, but be warned that your grip on these terms needs to be firm enough to recognize when they apply to any given security requirement.

The second section returns to the deployment descriptor—with a vengeance. Most web application security can be defined without writing a line of code. Although we do touch on a few Java APIs, you are encouraged to do as much as possible "declaratively." And you'll find that there are any number of elements that concern resources, users, and roles, and yet more that cover security across a network.

The third section will look at how you can cater for "logging in" to a web application (the proper term for this as we'll shortly discover is "authentication"). You'll see how this is also achieved through deployment descriptor configuration, through yet more elements.

A rough count reveals that of the 18 or so top-level deployment descriptor elements you need to learn for the exam (that's the elements that are direct children of `<web-app>`), 3 of those are explicitly about security. So that's one-sixth. However, when you look at all the elements (including subelements), around 20 out of 60 are security related. That accounts for a third of your deployment descriptor knowledge for the SCWCD! All this and more are covered in this chapter.

CERTIFICATION OBJECTIVE

Security Mechanisms (Exam Objective 5.1)

Based on the servlet specification, compare and contrast the following security mechanisms: (a) authentication, (b) authorization, (c) data integrity, and (d) confidentiality.

We start off with definitions of four terms. These aren't here for background: The exam explicitly tests this knowledge. The definitions used all come from the servlet specification, so what you're learning is J2EE's take on security. That said, the explanations that the servlet specification provides for security terms are more or less standard in any security environment.

Security Mechanisms

There are four terms you need to know. "Authentication" is the first: This is the process of identifying some party to a web application. The term "party" is deliberately vague, for authentication can occur not only between human users and web applications, but also between other systems and web applications.

Once authentication has taken place, "authorization" comes into play. Authorization rules determine what an identified party is allowed to do within a web application—which resources can be accessed and what can be done with those resources. Authentication and authorization go hand in hand. For one thing, you can't have authorization without authentication happening first. For another, it's rare to find a system that goes to the trouble of authenticating someone without employing some kind of authorization rule as well—even if it's all or nothing (authenticated users can use anything in the web application; unauthenticated users can't). In fact, as we'll discover, the base mechanics of web application security invite you to authenticate only if you attempt to access a resource protected by authorization rules.

It's an obvious point to make, but security is really necessary only because of a network. Accepted, if you have a stand-alone PC, you might want to have password protection in place. But who has a stand-alone PC these days? And in the context of J2EE and web applications, we are always considering a network. And a network provides an open invitation to malcontents and evildoers: What is a network packet for if not to have its contents spilled open and perhaps repackaged in some twisted form? This is where the other two security concepts come in. There's "data integrity," which is the business of proving that what you sent to or from a web application has not been tampered with on the way. And in addition, you're likely to want "confidentiality" (or "data privacy")—mechanisms to encrypt your network traffic so that no code-cracking approach can reveal the plain contents.

Security Definitions in Detail

So now let's consider these four security mechanisms in a bit more detail, including the definitions as found in the servlet specification. These may seem a bit formal, but they're precise—and very often, this wording is directly quoted in exam questions.

Authentication As we've said, authentication is the process of proving you are who (or what) you claim to be. The servlet specification puts it this way: "The means by which communicating entities prove to one another that they are acting on behalf of specific identities that are authorized for access." In the web application

sense, "communicating entities" typically indicates a client web browser on one end of the telephone and a J2EE web container on the other.

What does the spec mean, though, when it talks about "acting on behalf of specific identities"? It's most obvious from the client-to-server perspective. The server (web application) wants to know that you (the browser user) are a paid-up and registered member of the exclusive club the web application serves. In technical terms, the server is simply interacting with a piece of software described as a Mozilla-compatible browser, but clearly, it wants some means of knowing that behind that browser is a "specific identity," perhaps a human being called David Bridgewater. Let's not neglect the other direction, though. If I'm using a web application to check on my personal bank accounts, I need the server to prove to me at every step of the way (with every request/response) that it is acting on behalf of my bank.

Various means exist for trading authentication details. At the simplest and most insecure end, authentication involves a user providing a user ID and a password, sent unencrypted over the network. This is absolutely fine for the server to establish trust in the client, as long as some provision is made to protect the network between them (maybe through a virtually private network, or through an internal network unconnected with the wider world). More secure approaches might go for encrypting the authentication information, or even the entire request and response. The most secure means of authentication is through digital certificates—which contain rather more information than a mere user ID and password, and which can be used to establish trust in either direction: for the server in the client, or the client in the server.

In the final section of this chapter, we'll see how web applications ask the web container for authentication support. This is through the `<login-config>` element in the deployment descriptor, and we'll learn that this element makes provision for everything from basic user/password authentication to full-blown digital certificates.

Authorization Our simple definition of authorization stated that it's the mechanism controlling what you are allowed to do in a system. Again, the servlet specification is a little more formal and precise—it uses the term "access control for resources" to spell out what authorization does, and it defines this term as "The means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints."

When talking about "interactions with resources," we are mostly concerned with HTTP requests and responses to specific URLs (the "R" in this stands for "resource"). The process of authentication tells us what user (or program—i.e., system) is attempting to interact with our protected resource. But what does the definition mean by "collections of users or programs"? What this acknowledges is that you have

within your J2EE web server some means or other of associating those users and systems with particular roles. The J2EE specification doesn't say how this is to be achieved—it's server specific. For once, it's not something that's defined in the web application's deployment descriptor. Using Tomcat as an example, we find within it a configuration file called tomcat-users.xml. This contains a list of roles, followed by a list of users together with a comma-separated list of the roles to which they belong. Here's the file from my very simple configuration:

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <role rolename="lowlife"/>
  <role rolename="manager"/>
  <user username="tomcat" password="tomcat" roles="tomcat"/>
  <user username="role1" password="tomcat" roles="role1"/>
  <user username="both" password="tomcat" roles="tomcat,role1"/>
  <user username="david" password="tomcat" roles="lowlife,manager"/>
</tomcat-users>
```

You can see that the last user listed, "david," has a password of "tomcat" and belongs to both the "lowlife" and "manager" roles.

Other servers offer more sophisticated user registries, but whatever one you use, the important thing—in J2EE terms—is the association of someone or something you can authenticate with a given role. When you come to protecting resources in your web application, you say nothing about individual users in the deployment descriptor: What you do is associate a URL with a particular role. Only users within that role can use that URL. There are a few more nuances that we'll visit when we look at the `<security-constraint>` deployment descriptor security declaration, but here we have the essence of how to enforce "availability constraints" as talked about in our servlet specification definition of access control to resources above.

The only aspect of the definition we haven't covered is the enforcing of integrity and confidentiality constraints—but that has less to do with authorization and everything to do with the next two security terms: data integrity and data privacy.

Data Integrity The servlet specification is straightforward: "The means used to prove that information has not been modified by a third party while in transit." The means themselves may be complex—and invariably involve some kind of encryption. If a client encrypts its request in a way that only the server will understand (and vice versa), that's a guarantee that no modification has occurred. If there was some kind of

tampering, the request could not be de-encrypted: One byte out of place will ruin the whole.

You could rightly point out that encryption is an over-the-top method to prove the integrity of data. A client could, for example, use a "checksum" algorithm to compute a unique number dependent on the request contents. The server could verify integrity by running the same checksum algorithm over the request on arrival. The problem is that anyone snooping on the request could easily work out the algorithm and, having tampered with the request body, recalculate an appropriate checksum to fool the server. So encryption is invariably used, blurring the line between data integrity and data privacy (our next security term).

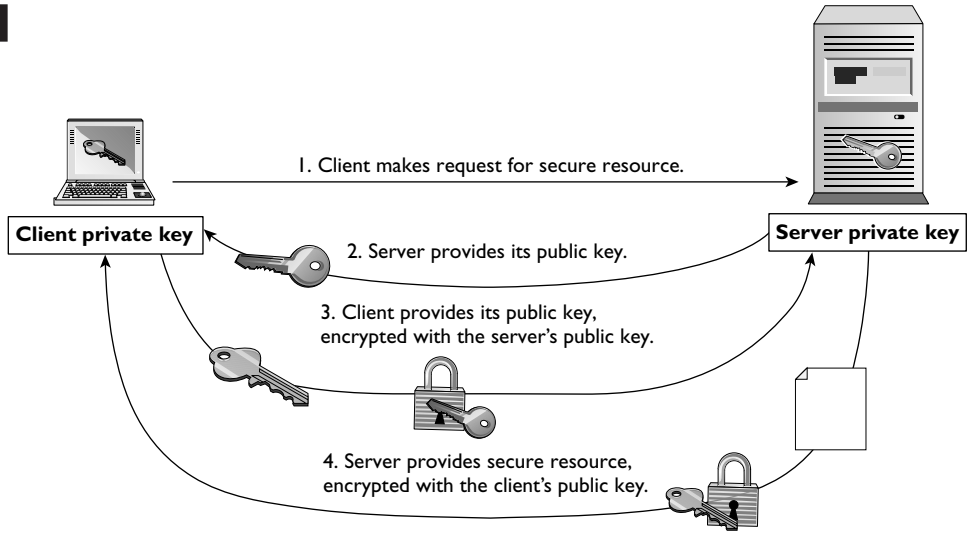
Before we move on, know that the deployment descriptor is again used when requesting data integrity for a particular resource: Look out for the line `<transport-guarantee>INTEGRAL</transport-guarantee>` in web.xml. We'll see how this fits into the wider scheme when we look at the deployment descriptor in more detail.

Confidentiality (Data Privacy) The servlet specification defines this last term as "The means used to ensure that information is made available only to users who are authorized to access it." The means is always encryption—translating your plain text into indecipherable code. This usually involves the use of a pair of matching keys, termed private and public. The public key (for a server or client) can be issued to any interested party. The private key is held absolutely privately: client's private key by the client and server's private key by the server. It's impossible to deduce what the private key is from the public key (or vice versa). You can encrypt plain text with (say) the public key, but then you'll need the matching private key to decrypt the enciphered message.

Let's consider the case of sending data that must remain confidential from the server to the client: Figure 5-1 shows this scenario. In (1) the client asks the server for a secure resource. So the server responds (2) by sending its public key. This can be used to encrypt messages in such a way that only the server will understand—only its private key can decode the messages. So the client takes advantage of this in (3) to send its own public key to the server, but encrypted in such a way that only the server can make use of it. Because it's a public key, the client doesn't have to encrypt it, but it gives an added layer of security: Why give your public key to anyone except those who want to make legitimate use of it? Then the server can encrypt the secure resource using the *client's* public key (4) and transmit it to the client. Now the resource can pass through the insecure medium of the Internet in the comfortable knowledge that nobody can de-encrypt it—except the intended client, using its private key.

FIGURE 5-1

Encryption



This is a somewhat simplified picture of the full set of likely security interactions between the client and server. For one thing, private/public (asymmetric) key encryption takes a great deal of computing power. What usually happens is that asymmetric encryption is used as a secure means for exchanging symmetric keys (same one encodes and decrypts); then the symmetric keys are used for communication beyond this point. Why? Symmetric key encryption is that much faster—and still just as secure, provided you can be absolutely sure nobody other than the intended parties has hold of the symmetric key. And because new symmetric keys can be manufactured for each request/response pairing as necessary . . . well, you get the picture that security is an involved business. I am content to be a humble web component developer, not a security programmer. And for purposes of the SCWCD, we have gone as deep as we need to into encryption mechanisms!

Note that you might want to encode a message with your own private key instead of someone else's public key. That may not make sense at first: Surely, anyone with your public key can read the message. And the whole point of a public key is that it's—well—public? The point is that you're encrypting for a different purpose. As long as those who have your public key are sure it's your public key and no one else's, then they can be sure that a message you encoded with your private key comes from you. So that's how encryption solves the integrity issue.

And just as for data integrity, confidentiality in a web application is ensured by the deployment descriptor element `<transport-guarantee>`, this time with a

value of CONFIDENTIAL. The actual medium used by web containers is very often SSL—secure sockets layer (you know when you're using SSL because the URL your browser points to begins "https"—the secure version of HTTP). SSL is a private/public key technology for communicating privately over the Internet that was originally developed by the Netscape Communications Corporation and is now incorporated in practically every web device under the sun.

EXERCISE 5-1

Security Mechanisms You Have Encountered

Because we haven't looked at any specific web application technology yet, this exercise is one of recollection. Note down all the application security mechanisms you have met in the past (at least, those that you can remember). For each one, identify which parts had to do with each of the four "big ideas": authentication, authorization, data integrity, and confidentiality.

For the authorization part, sketch out the structure of the steps involved: How exactly were authorized users, groups, or principals tied to specific resources? From this, imagine how web applications might solve the authorization problem. The combined exercise of memory (or imagination!) will prepare you well for the specifics that we encounter in the next two sections.

You can look at my take on the above exercise by deploying the web application ex0501.war, found in the CD in /sourcecode/ch05. Point to the following URL:

```
http://localhost:8080/ex0501/security.html
```

CERTIFICATION OBJECTIVE

Deployment Descriptor Security Declarations (Exam Objective 5.2)

In the deployment descriptor, declare a security constraint, a Web resource, the transport guarantee, the login configuration, and a security role.

Most applications that I worked with in the past have their own security structure described in program code. J2EE web applications make a valiant attempt to separate

out the security layer. The idea is that application developers can hand over their work to deployers, who can then construct a security mechanism without touching the program code. In business terms, the deployer has to know what each resource in the web application does but can remain thankfully oblivious of program code details.

The mechanism is "declarative": In other words, you declare the security you want in the deployment descriptor instead of enshrining it in code. Declarative mechanisms bring everything to the surface and keep management simpler. That said, it's not altogether simple—there are three top-level deployment descriptor elements that we explore in this section. The first is `<security-constraint>`, the most complex of the three—it defines what resource we're securing, what roles can access the resource, and how the resource is to be transmitted across the network. The second is `<login-config>`, which defines what authentication mechanism is to be used. The third (and easiest) is `<security-role>`, which simply catalogues any security roles in use by the web application. We'll look at these in turn in the headings that follow.

exam Watch

You should not be asked about the order of the three top-level security elements, for order at this level doesn't matter—as we have previously noted. However, the traditional order (that used to be enforced) is <security-constraint>, <login-config>, then <security-role>. I would advise you to stick to this for compatibility with servers enforcing the old servlet level 2.3 DTD. However, do note that order of the subelements within the three top-level elements does still matter in servlet level 2.4 and that you could well be quizzed on this.

The `<security-constraint>` Element

The `<security-constraint>` element has three parts to it, each represented by a subelement:

- Web resource collection (`<web-resource-collection>`) defines the resource to be protected and also the HTTP methods by which it can be accessed (GET, POST, PUT, DELETE, etc.).
- Authorization constraint (`<auth-constraint>`) determines which roles are allowed access to the resource.

- User data constraint (`<user-data-constraint>`) decides what protection (if any) is required when transporting the resource over the network.

We're going to look at each of the three parts in detail. Figure 5-2 maps out where these fall and what subelements each contains.

Web Resource Collections

The one thing a security constraint must contain is one (or more) web resource collections. Figure 5-2 shows you that a `<web-resource-collection>` consists of a name, some optional description lines, one or more URL patterns, and an optional list of HTTP methods. Let's look at each of these in turn.

`<web-resource-name>` This is just a logical name for the web resource collection. You have to include it, but it has no technical significance—it's just a memory aid to help you understand why you have grouped what are potentially many URL patterns together.

`<description>` Any number of description lines contained by `<description>` tags can follow—including none at all. This is no different from the many other appearances we have seen for the `<description>` element. A pop quiz question (don't worry, I don't see this one appearing in an exam, but it will encourage you

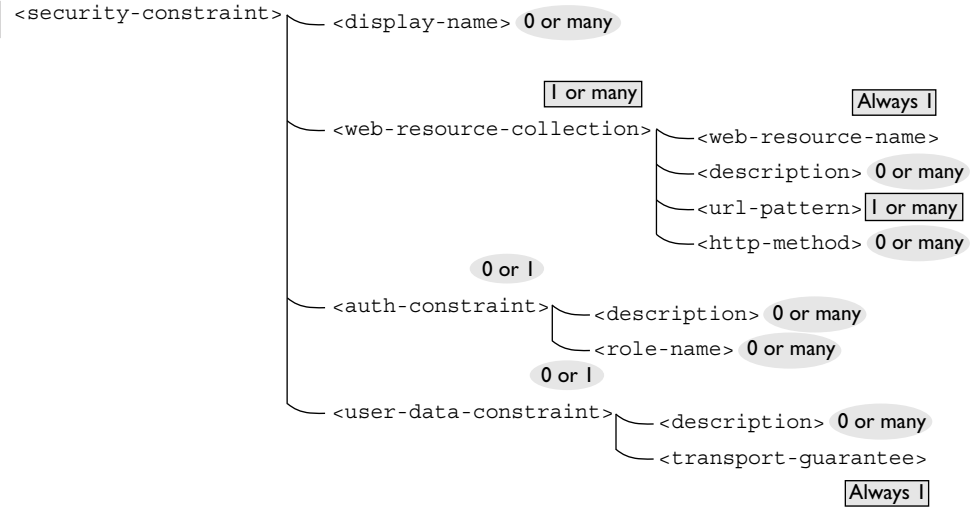


FIGURE 5-2 The `<security-constraint>` Element

to look at the deployment descriptor). Where else have you encountered the description element?

<url-pattern> This is a URL pattern just like the ones you have seen in <servlet-mapping> and <filter-mapping>: The same rules apply—see the section on URL pattern strategies for a refresher on these (see Chapter 2). You can define a URL pattern for any resource in your web application. You're not restricted to matching URL patterns with those for servlets (though you may well want to do that)—you can also reference static HTML or JSP pages in the directly available web content for your application.

There must be at least one <url-pattern> included for the web-resource-collection, and you're quite likely to include a whole list in a full-sized production application. These will then all be governed according to the same security rules defined elsewhere in the security constraint.

<http-method> Finally, you can include a list of HTTP methods. More often, you're likely to miss out this element, which means that any HTTP method (that's the "big seven" we discussed in Chapter 1 plus any other more obscure ones) executed against the resources defined by your URL patterns will be subject to the same rules. No matter if you POST or GET or DELETE or PUT, the same roles will be needed for access, and the same transport guarantees will apply.

However, you can specify individual methods as needed. For example, you might want to impose a blanket ban on the use of dangerous HTTP methods on resources in your web application. You could achieve that with the following <security-constraint> configuration:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>All Resources</web-resource-name>
    <url-pattern>/</url-pattern>
    <http-method>DELETE</http-method>
    <http-method>PUT</http-method>
  </web-resource-collection>
  <auth-constraint />
</security-constraint>
```

The URL pattern here (/) encompasses all resources within the context for the current web application. The HTTP methods listed are DELETE (to remove a resource at the requested URL) and PUT (to place a resource at the requested URL,

overwriting what's there). We'll meet `<auth-restraint />` next—suffice to say for now that this is the configuration setting you use to deny access to any request for these HTTP methods to this web resource collection.

If you use a browser that can generate DELETE methods (such as the one you used in the exercises in Chapter 1), and target a resource protected as above, you'll get an HTTP 403 error (access forbidden).

on the Job

It's a really good idea to be as restrictive as possible. Limit any given resource to only the HTTP methods that are reasonable to execute on that resource. This usually means GET and HEAD, with POST thrown in only when required.

Authority Constraint

So second up in subelements of `<security-constraint>` is `<auth-constraint>`. If you refer back to Figure 5-2, you'll see that it's an optional element with the security constraint and contains two elements of its own—the ubiquitous `<description>` element and the crucial `<role-name>`. The purpose of an authority constraint is to list permitted roles: Any users within the roles can operate on the resources defined in the web resource collection.

The normal use of this element might look like this:

```
<auth-constraint>
  <role-name>employee</role-name>
  <role-name>supervisor</role-name>
</auth-constraint>
```

Anyone falling within the employee or supervisor roles—or both—will have access to the resource. Even more simply, the following definition:

```
<auth-constraint>
  <role-name>supervisor</role-name>
</auth-constraint>
```

restricts resource access to only those in the supervisor role.

The roles names that you choose must (according to the servlet specification) be listed in the `<security-role>` element, which we have yet to meet.

You'll recall that the mapping of specific users to these roles is not a job for the deployment descriptor. The server must provide some way of achieving the mapping, but the exact mechanism is server specific. We met the tomcat-users.xml file earlier on, which showed you one way in which the Tomcat server resolves this need.

There are three special cases to consider with regard to `<auth-constraint>`:

Absent from the Security Constraint What if there is no `<auth-constraint>` for your security constraint? That's fine—it simply means that the web resource collection is open to all, regardless of role or authentication. You may wonder what the point of this is—why go to the trouble of setting up a web resource collection if you're not going to ascribe it to any role? We'll discover one possible reason when we look at `<user-data-constraint>`.

Present with No Role Names The `<auth-constraint>` element might be present in the security constraint, but with no role names listed. It might manifest itself like this:

```
<auth-constraint></auth-constraint>
```

Or this:

```
<auth-constraint />
```

which is the XML shorthand for an opening and closing tag with no value, or this:

```
<auth-constraint>
  <description>Trust No-one!</description>
</auth-constraint>
```

The effect in every case is to deny access to the resource for any role whatsoever. We saw an example of this when we looked at the `<http-method>` element a little earlier and discovered a technique to deflect any DELETE or PUT methods executed against any resource in our web application.

When more than one security constraint is set up for the same web resource collection—which can happen—the effect of the no-value authority constraint is overriding. No matter if you set up a web resource collection open to the world—if the same URL patterns (for the same HTTP methods) are protected elsewhere with the no-value authority constraint, access will be denied.

Present with the Special Role Name of "*" You might want to use an `<auth-constraint>` element that uses the special role name of "*":

```
<auth-constraint>
  <role-name>*</role-name>
</auth-constraint>
```

This role name is a shorthand way to include all the roles defined within the web application. These are all the roles that appear in all the `<security-role>` elements (which we will be discussing very shortly).

exam Watch

What HTTP response codes result from the authentication and authorization process? When a resource is requested to which access is always denied (because of the "no-value" authorization constraint), the web server rejects the request with a 403 (SC_FORBIDDEN) response code. Authentication (identification of the user) may not even have happened at this point: There's no need, for any user would be forbidden. If there are potential roles, then authentication must happen. If it hasn't happened already, the web server sends a 401 (SC_UNAUTHORIZED) response code, which causes the browser to supply authentication information in some form or other. If subsequent checking shows that the authenticated user is not a role entitled to the resource, the web server rejects the request with a 403 (SC_FORBIDDEN) response code.

User Data Constraints

In addition to — or as well as — authority constraints, you can impose user data constraints. These apply to the requests and responses that pass to and from the web container. Figure 5-2 shows you that when you include a `<user-data-constraint>`, you must have a `<transport-guarantee>` element within it. This has three valid values, as shown next.

NONE No constraints are applied to the traffic in and out of the container. The requests and responses can pass in plain text over the network. Setting the transport guarantee to this value is the same as leaving out `<user-data-constraint>` altogether.

INTEGRAL The web container must impose data integrity on requests and responses: That's the term we defined earlier as meaning that messages are not tampered with in transit. How it does this is up to the web container, but typically it will use SSL (secure sockets layer) as the communication medium.

CONFIDENTIAL The web container must ensure that communicated data remains private—no one must be able to understand the secret messages passed between the client and the web container. In theory, the CONFIDENTIAL guarantee is stronger than INTEGRAL: If you have confidentiality, integrity is implied. However, web containers again mostly achieve the guarantee by using SSL—and so may look no different from a transport guarantee of INTEGRAL.

Let's return to that question we posed earlier—what's the point of a web resource collection without an `<auth-constraint>`—such that any user (even an unathenticated user) can access the resource? Well, you might still protect your resource collection with a `<user-data-constraint>` but allow open access. Consider a web page for a user to register personal details. Anyone can access this page, but it's best to ensure that their details remain private when transmitted back to the web server. By specifying a CONFIDENTIAL transport guarantee without any authority constraint, you achieve this end.

INSIDE THE EXAM

Addition with Security Constraints

The exam might well test your knowledge on adding together security constraints for identically defined web resource collections. Suppose you have two security constraints declared as follows:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Employee Page</web-resource-name>
    <url-pattern>/EmployeeServlet</url-pattern>
    <http-method>PUT</http-method>
    <http-method>DELETE</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description>Administrator Permissions</description>
   <role-name>administrator</role-name>
 </auth-constraint>
</security-constraint>
```

INSIDE THE EXAM (continued)

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Employee Page</web-resource-name>
    <url-pattern>/EmployeeServlet</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description>Any Authenticated Permissions</description>
    <role-name>*</role-name>
  </auth-constraint>
</security-constraint>
<!—Other details like login configuration omitted for brevity . . . →
<security-role>
  <role-name>administrator</role-name>
</security-role>
<security-role>
  <role-name>employee</role-name>
</security-role>
```

The same URL pattern is protected in both cases (/EmployeeServlet). There are two valid roles for users to be in when logging on to the application—administrator and employee. It's clear from the first security constraint that users in the administrator role can execute the HTTP PUT, DELETE, or POST methods on this URL. But from the second security role, we can see that a user in any valid role (administrator or employee) can GET or POST to the URL—both roles are part of the "*" (all roles defined) role name. So the second security constraint extends the range of things that an administrator can do: GET as well as PUT, DELETE, and POST from the first security constraint. An employee is covered only by the second security constraint and so can only GET or POST.

And don't forget that if `<auth-constraint />` appears against a URL pattern/HTTP method combination, this "addition of permissions" rule is irrelevant: The resource is blocked.

Enforcing security by HTTP method is fine, but it can be obscure to the hapless deployer charged with imposing security. At least when a deployer provides security for EJBs (Enterprise JavaBeans), he or she is likely to be confronted with method names that reflect a business process: `transferFunds()` ***might be an example. HTTP methods are—well—HTTP methods; they say nothing by themselves about a business process. Two design practices may help your deployer: (1) make it obvious what a resource does, and (2) keep the scope of any one resource narrow. For (1), this usually comes down to naming your resources well—if a deployer is faced with the name "TransferFundsServlet," it's pretty clear what the resource does—all that's left is to protect all HTTP methods (or at least GET and POST). Achieving (2) is trickier. After all, the TransferFundsServlet may be capable of transferring funds in one direction, from customer to branch accounts, and also in the other direction, from branch to customer. However, different security roles may well apply to the two different directions of fund transfer. Separate resources—a BranchToCustomer Servlet and a CustomerToBranchServlet—may make security easier to impose. (And if you're thinking this might lead to code duplication, you're right—but remember that the same underlying servlet class can be declared twice in the deployment descriptor, with different initialization parameters and with independent servlet mappings that can be protected separately.)***

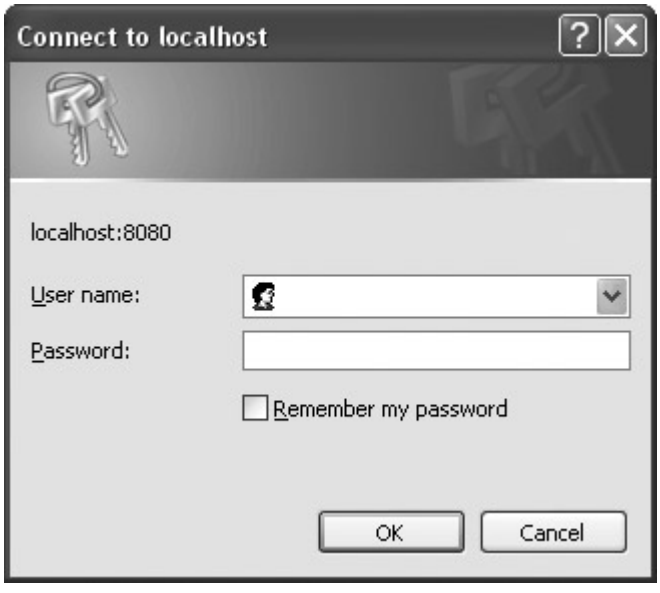
The <login-config> Element

Now we move on to the next top-level deployment descriptor element governing security, which is `<login-config>`. This governs authentication: how you "log on" to the web application. We're going to see one way we can use `<login-config>` to control authentication here, just to get us going, but revisit it more fully in the next section, for authentication methods are an exam objective in their own right.

The simplest way to ensure that some kind of authentication occurs is to have `<login-config>` set up as follows:

```
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
```

This works as follows: When a browser (or other client) requests a secure resource from a web application for the first time, the web application doesn't return the resource straightaway (naturally enough!). Instead, the server requests authentication from the browser. Nearly all browsers are built to understand such a request, and they respond by popping up a dialog box that requests a user ID and password—the following illustration shows how this looks in Internet Explorer in Windows XP.



The user fills in the details and presses OK, and the server checks the user ID and password against whatever database or other registry it is set up to use. Assuming that it finds a match (and other valid criteria are met—such as the user being in the correct role), the server returns the resource to the user.

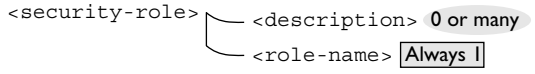
exam Watch

Whenever a Request Dispatcher is used to forward or include a resource—be it dynamic (servlet) or static (plain HTML)—the security model does not apply. It is applicable only to web resources requested by a client.

The <security-role> Element

As we've noted already, <security-role> is the simplest of the "big three" security-related deployment descriptor elements. We use it to catalogue all the security roles in use in an application. You can see in the following illustration

that it has only one functional subelement, called `<role-name>`. (I'm not counting `<description>`, which is just for documentation purposes.)



exam Watch

The current specification says this about valid role names: They have to obey XML "name token" (NMTOKEN) rules. (Actually, this is a throwback to the old style of DTD validation for the deployment descriptor, but if the current specification still says it, who am I to argue?) This means that you should not have embedded spaces or punctuation in the name. Characters and numbers are fine (and you can even begin the role name with a number).

This might also be a good time to point out that the web container matches role names case sensitively when determining access to secured resources, so "rolename" is not the same as "RoleName."

You only include one role name per security role, but `<security-role>` can appear as many times as needs be in the deployment descriptor.

The idea is that any role names used anywhere in the deployment descriptor must appear here. This means any role names you use for authority constraints within a security constraint, and any for role links in the security role reference in the servlet element.

The servlet specification insists that role names "must" appear as a role name here. I find Tomcat's default behavior is to treat the absence of a security role in this list as a warning-type message on startup. However, for exam question purposes, you should go with the servlet specification version of the truth.

on the Job

Programmatic security—that is, using APIs in your code to enforce authentication and authorization rules—is not officially covered by the exam. Why not? Perhaps to encourage J2EE application builders to place their trust in declarative security. However, programmatic security is more flexible, and there are often good reasons to use it. Http Servlet Request has three methods: `getRemoteUser()`, `isUserInRole()`, ***and*** `getUserPrincipal()`. `getRemoteUser()` ***returns a String containing***

the name of the authenticated user. `getUserPrincipal()` *is essentially a replacement for* `getRemoteUser()`*. Instead of returning a plain String, it returns a java.security.Principal object, partly because Java prefers to call authenticated parties "principals" rather than "users." Users tend to be human beings; principals might be human beings or other computer systems. However, ultimately the only useful thing you can do with a Principal object is to call* `getName()` *to return a String with the user's (sorry—principal's) name. The third method—*`isUserInRole(String roleName)`*—returns a* `boolean` *indicating if the user is in the role name passed as a parameter to the method. Note that you don't first have to discover the user's name to make use of this method.* `isUserInRole()` *will take account of the roles you have set up in security constraints in your deployment descriptor. However, it also takes account of some subelements in the* `<servlet>` *element. You might find declarations such as the following:*

```
<servlet>
  <servlet-name>EmployeeDetails</servlet-name>
  <servlet-class>com.osborne.EmployeeDetails</servlet-class>
  <security-role-ref>
    <role-name>MGR</role-name>
    <role-link>manager</role-link>
  </security-role-ref>
</servlet>
```

Suppose that anyone can access the EmployeeDetails servlet but that certain sensitive details are viewable only by managers. So there's no need to associate the servlet with a security constraint, but we do have a reason to use programmatic security to limit some of the output. For this to work, in addition to the servlet's deployment declarations above, the role name "manager" should be defined as an allowed security role for the application—i.e., as a `<role-name>` *in the* `<security-role>` *element. The servlet code has the option of using isUserInRole("manager") or isUserInRole("MGR") as a check on whether the authenticated user is a manager of not. The* `<role-name>` *of MGR in the* `<security-role-ref>` *is mapped on to the* `<role-link>` *of manager, which is a real security role defined in the deployment descriptor. What this facility allows for is taking servlets coded against one set of role names, then deploying them in an environment where a different set of role names is defined—without having to revisit the code.*

EXERCISE 5-2

Securing a Servlet

In this exercise, we'll take a servlet and make it a secured resource. Just for good measure, we'll include some programmatic security inside the servlet so that some of the web page will display itself only if the user belongs to a specific role. Because we don't yet officially know how to turn on authentication, we'll see what happens when you try to access this servlet in an unauthenticated manner. This will be a frustrating experience, for the web page will not display properly until we introduce authentication in the next exercise.

The context directory for this exercise is ex0502, so set up your web application structure under a directory of this name.

Set Up the Deployment Descriptor

1. Define a servlet called CheckedServlet, with a suitable servlet mapping. I doubt you'll need to refer back to Chapter 2 now for this—you've done it many times!
2. However, by way of a departure, include a `<security-role-ref>` subelement within `<servlet>`, with a `<role-name>` of MGR and a `<role-link>` of manager.
3. Define a `<security-constraint>` element.
4. Define a `<web-resource-collection>` for the `<security-constraint>`. The `<web-resource-name>` must be included, but it's immaterial what text you choose. However, the `<url-pattern>` should match the `<url-pattern>` of the `<servlet-mapping>` for CheckedServlet—that's the resource we are trying to protect.
5. Define an `<auth-constraint>` for the `<security-constraint>`. Set the value of `<role-name>` to *.

Write CheckedServlet

6. Create a Java source file CheckedServlet.java in /WEB-INF/classes or in an appropriate package directory. Write the class declaration in the source file, extending HttpServlet.
7. Override the `doGet()` method in CheckedServlet.

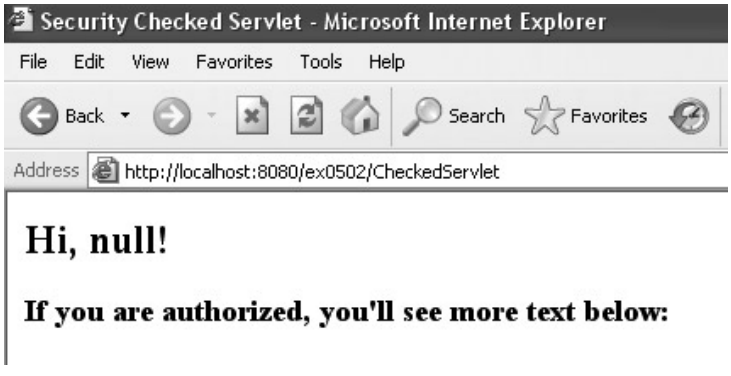
8. Obtain the authenticated user name—you can use the `HttpServlet Request.getRemoteUser()` method or the more tortuous (but preferred) approach to `HttpServletRequest.getUserPrincipal()` to return a Principal object, on which you can execute the `getName()` method. Protect yourself from null values—when we run the servlet in this exercise, there won't be an authenticated user (that comes in the next exercise).
9. Write out some text to the response's writer, including the user name (even if it will be null at present).
10. Use the `HttpServletRequest.isUserInRole()` as the condition for writing some additional text to the web page. The role to check is MGR.

Run the Application

11. Once you're satisfied that the servlet is compiled, deploy the application, and run it with an appropriate URL, such as

```
http://localhost:8080/ex0502/CheckedServlet
```

12. You should get output much like that shown in the following illustration. Because there's been no attempt to authenticate a user, no user name can be displayed. Also, the extra text that appears by virtue of being an authenticated user in the MGR role fails to appear.



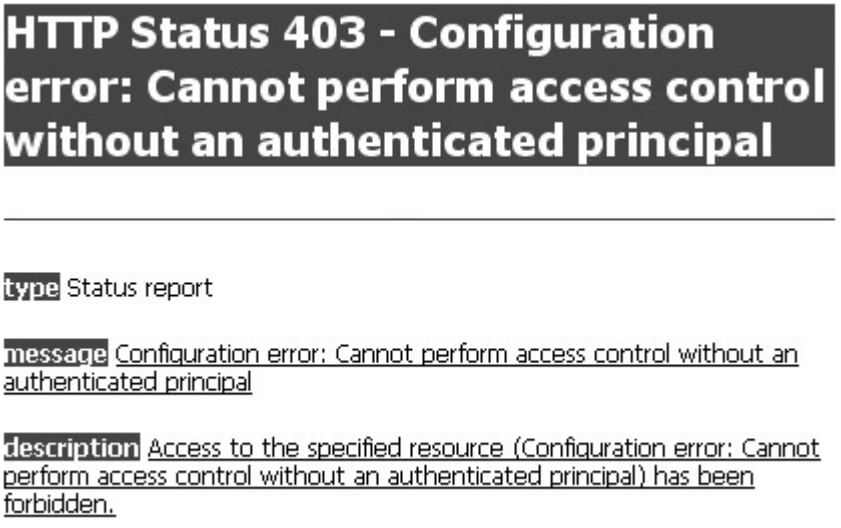
Revise the Application

13. Now revisit the deployment descriptor web.xml. Change the `<role-name>` of `<auth-constraint>` to say lowlife. Remake the WAR, and redeploy the application.

14. Note any startup messages your web container produces. Do you get an informational/warning message complaining about the absence of a `<security-role>` element for the lowlife role in your deployment descriptor? I do with Tomcat, though it doesn't stop the servlet from being deployed.
15. Try running the servlet with an appropriate URL, such as

```
http://localhost:8080/ex0502/CheckedServlet
```

16. Note the error that you get. Tomcat gives me an HTTP 403 error (shown in the following illustration), and tells me that the resource is not available to an unauthenticated user. We'll fix this in the next exercise.



CERTIFICATION OBJECTIVE

Authentication Types (Exam Objective 5.3)

Compare and contrast the authentication types (BASIC, DIGEST, FORM, and CLIENT-CERT); describe how the type works; and, given a scenario, select an appropriate type.

As I intimated earlier, there is more to be said about the `<login-config>` element, which we saw in the last section. We met the simplest authentication type it supports—BASIC—which we're going to find out is also the least secure. In fact, this element allows for four authentication types in all. We'll discuss all of these in this section—how to set up their configuration, how they work, and what their benefits and drawbacks are.

Authentication Types

The four authentication types are BASIC, FORM, DIGEST, and CLIENT-CERT. Let's do a quick survey of the types before exploring in detail. BASIC, as we've seen, forces the appearance of a dialog box in browsers inviting user and password details. Behind the scenes, we'll see that although these details aren't quite sent over the wire in plain text, they're not very secure either. FORM is more or less a cosmetically improved version of BASIC—you supply your own design of web page to solicit user and password details, instead of being stuck with the browser's dialog box. The behind-the-scenes transmission details are just the same. DIGEST imposes encryption rules on the password, improving this situation. CLIENT-CERT goes a step further—all security details are kept in an electronic document called a certificate. We'll see how this is the most secure arrangement but also the most work to set up.

Let's first of all have a look at the full layout of the `<login-config>` element, which controls authentication. You see in Figure 5-3 that this element has several subelements but that the only one common to all authentication types is `<auth-method>`, which describes which of the four authentication types is in force for the web application.

exam Watch

In common with all the other top-level deployment descriptor elements (those directly under the root element, `<web-app>`), `<login-config>` is described in the schema as being able to appear "0 or many" times. The truth is that `<login-config>` should appear 0 or 1 time only: If present, it applies to the whole web application, so it only makes sense for it to appear once. So although it's legal (in XML terms) for `<login-config>` to appear more than once, the container is supposed to do additional validation to trap and reject such a faux pas. Indeed, this is true for all the top-level elements that end in -config, including `<session-config>` and `<jsp-config>`.

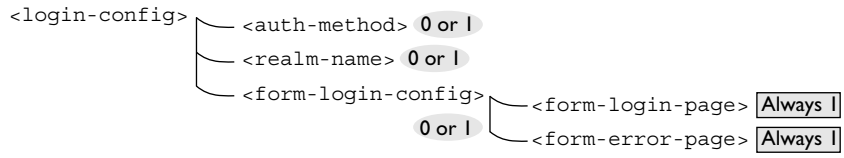


FIGURE 5-3

The <login-config> Element

BASIC Authentication

We saw BASIC authentication at work in the last section. There isn't that much left to say, except for a subelement we bypassed called <realm-name>. Here's the deployment descriptor for BASIC authentication, this time with the realm name included:

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>MyUserRegistry</realm-name>
</login-config>
```

The realm is simply the registry used to store user account information. It could be that your server has more than one realm at its disposal and that a deployer will need to specify which one is meant here in web.xml. However, it's not mandatory—mostly, servers are concerned only with validating against one realm, so there is no need to specify. Another quirk of realms is that they are applicable only to BASIC and DIGEST forms of authentication.

When you press the OK button on the browser dialog and transmit your user ID and password over the Internet, there is a token attempt made to fool prying eyes. The password is not sent as plain text but is passed through a process called Base64 encoding. This turns the password into something that is no longer human-readable. But it's not the same as encryption: A hacker with any level of sophistication will have a Base64 decoder to turn the password back into plain text. (Base64 encoding and decoding tools are freely available on the Web—there's even one in the J2SDK, though it's pretty well hidden and lacks Javadoc.)

This doesn't make BASIC authentication useless, however. It's perfectly secure when you provide a transport guarantee to ensure encryption. That way, all parts of the request—including password details—are concealed from theft across the network.

DIGEST Authentication

DIGEST authentication improves a little on BASIC by using a secure algorithm to encrypt the password and other security details. When the browser tries to access a secure resource, the server generates a random value called a "nonce" and passes this

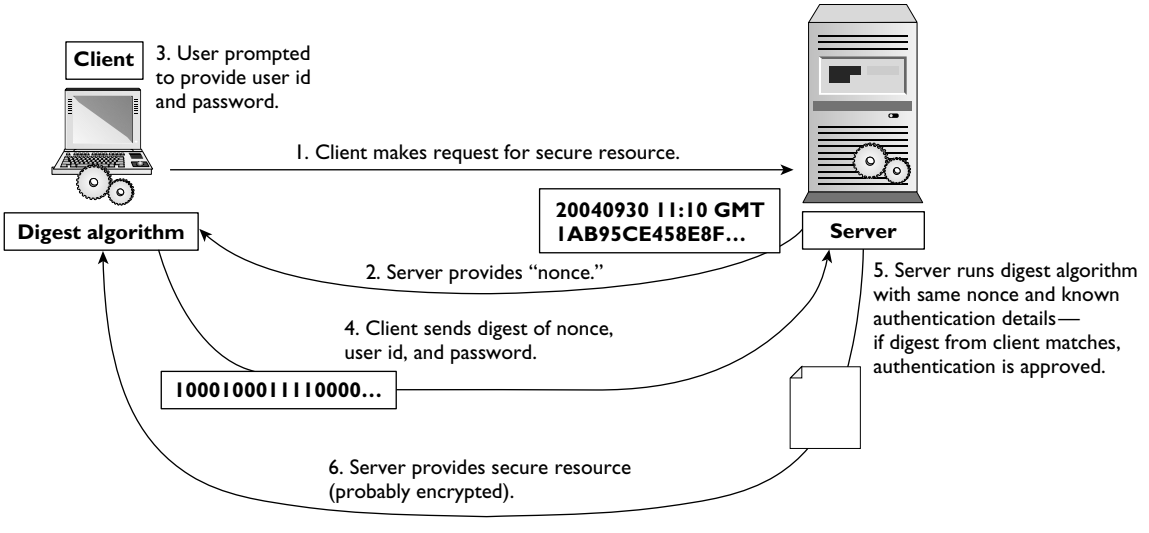
to the browser. The nonce can be based on anything—often, a unique identifier for the server (such as an IP address) and a timestamp.

The browser uses this value and, together with other pieces of information—always the user ID and password, sometimes URI and HTTP method as well—applies the digest encryption algorithm (usually the very secure MD5 algorithm). This is a one-way process: The idea is to turn the seed information into junk, which can never be translated back into human-readable text. This junk is called the digest, and the client sends it to the server.

The server can't use the digest to decode security details. What it can do, though, is to generate its own digest—using the nonce value it provided and known user and password details—and compare this with the digest passed from the browser. If they match, the client is considered authenticated. The process is shown in Figure 5-4.

Because nonces are generated on the fly from transient information, each session (and sometimes each request) uses unique digests—the possibility of intercepting and using an existing digest to fool a server is practically nonexistent. So it's very secure. What it does rely on, though, is that the server and browser have identical expectations about the digest: which algorithm to use and which pieces of information to apply the algorithm to. And there's the rub—different browser vendors do different things in support of DIGEST authentication. You need to know (and test) the clients you expect your web application to support, and that may not be easy (or even possible) to predict. Hence, DIGEST lags behind other

FIGURE 5-4 Digest Authentication



authentication types in terms of adoption, though it is much more widely supported than it used to be.

FORM Authentication

FORM authentication is primarily provided for aesthetic purposes. Why be at the mercy of an ugly browser dialog when you can provide your own nicely designed logging-in page? There are only a few rules you have to abide by when constructing such a page:

- The HTML form must use the POST method (GET is not acceptable).
- The form must have "j_security_check" as its action.
- The form must include an input-capable field for user called "j_username."
- The form must also include an input-capable field for password called "j_password."

Here's an example form that puts all the rules together (though in aesthetic terms, it's more minimalist than the authentication dialog box my browser provides):

```
<html>
<head><title>Login Form</title></head>
<body>
<form action="j_security_check" method="POST">
<br />Name: <input type="text" name="j_username" />
<br />Password: <input type="password" name="j_password" />
<br /><input type="submit" value="Log In" />
</form>
</body>
</html>
```

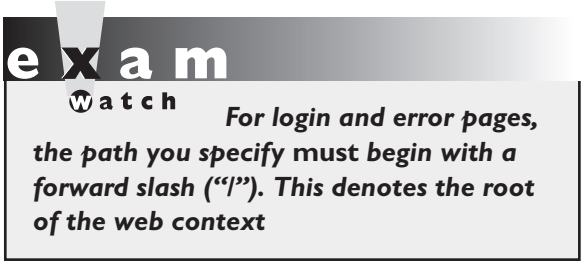
The above is a static HTML page, but dynamic JSP pages are just as valid.

Form-based authentication also demands that you provide an error page. There are no rules for the content of such a page. Once the pages are designed, you need to look at how to plug them into the deployment descriptor. This is a variant of `<login-config>` that fits the criteria:

```
<login-config>
 <auth-method>FORM</auth-method>
 <form-login-config>
  <form-login-page>/login.html</form-login-page>
  <form-error-page>/error.html</form-error-page>
 </form-login-config>
</login-config>
```

The <auth-method> is FORM, as you might expect. <realm-name> has gone: It's not used for form authentication. However, form authentication has an element that is unique to its type: <form-login-config>. This in turn has two subelements—<form-login-page> and <form-error-page>—whose values point to the location and name of these pages within the web context.

The mechanism of logging in works like this: The first secure resource you attempt to access in a web application will not be sent to you directly. Instead, the server caches the URL you are trying to reach and redirects you to the form login page. You supply a user ID and password; assuming that the server is happy with these credentials, you are then passed on to the URL you requested in the first place. However, if your login fails for any reason, the server redirects you to the error page you specified.



Assuming successful login, access to subsequent secure resources will not require re-authentication. It's the web server's business to achieve this in any way it can—the servlet specification doesn't mandate an approach. Most of the time, this comes down to attaching some additional information to the JSESSIONID cookie. This implies that invalidating the session will log you out of the system, but this implication can't be guaranteed.

One of the frustrations of form-based authentication is that you can't go directly to the login form. You have to attempt to access an otherwise secured resource and let the server redirect you to the login page. Try putting the address for your login page directly in your browser address line—experience the error that you get (usually something along the lines of "cannot perform j_security_check directly"). So if you want to design an unsecured home page with a login field for registered users in the top right-hand corner, then form-based authentication is not for you.

Also in common with BASIC authentication, there is no intrinsic protection for security information. You don't even get the Base64 encoding of the password. But again, as for BASIC authentication, you can get around that by using a virtually private network, or a secure protocol such as SSL for your transport guarantee.

CLIENT-CERT Authentication

The fourth and final method to discuss is CLIENT-CERT, which uses digital certificates to achieve authentication. This is the most secure form of authentication, but it also requires the most understanding and the most setup.

When we talked about data privacy earlier in the chapter, we introduced the idea of public and private keys for encryption. Digital certificates build on this idea by providing a home and an identity for a public key. Anyone can create a certificate, using specialized (but publicly available) software such as "keytool," which is shipped with the J2SDK. By passing the right parameters, keytool (or the equivalent) generates a private key and a matching public key, usually stored in some fully encrypted form on the creating computer's hard drive.

From this "keystore" you can extract a certificate file that is fully technically valid. But if anybody can mint one of these things, what trust can be placed in it? The usual procedure is to pass on your "self-signed" certificate details to a properly established certificate authority. The VeriSign Corporation (http://www.verisign.com) is well known, as is Thawte Consulting (http://www.thawte.com). These companies verify your identity (with different grades of background checking possible, reflecting different levels of cost and trustworthiness) and "rubber-stamp" your certificate — or, more correctly, produce a new certificate based on the details you supplied, vouched for by them. The most important action they take is to use their own private keys to digitally sign your certificate. Practically all browsers and web servers are in possession of these company's public keys. This gives them the means to check a digital signature (from VeriSign, Thawte, or whomever) on your certificate, verifying that your certificate is at least vouched for by a trusted third party. Here's what you can expect to find in an X.509 certificate (X.509 is the most popular standard):

- Version of the X.509 standard (v1, v2, or v3).
- A serial number unique to the certificate authority (VeriSign, Thawte) issuing the certificate. (The certificate authorities can use these numbers to maintain a "blacklist" of revoked certificates.)
- The signature algorithm used to digitally sign the certificate.
- Validity period: when the certificate will start and expire.
- The subject: in other words, you, the requester of the certificate. This is held as a "distinguished name"—nothing to do with your social standing, but more about uniqueness. A distinguished name has several components, including a common name (an individual—"David Bridgewater"), organizational unit (department), organization name (company), locality name (often, city), state name (or province), and country (two-character ISO code).
- Issuer name: the name of the certificate authority, again held as a distinguished name.

- A digital signature, encoded with the certificate authority's private key. This will be a digest of information within the certificate—which also means that tampering with the certificate (not just the digest itself) will render it immediately invalid.
- Last but very much not least: the subject's (your) public key.

Once you have your certificate, you can install it into your browser. Every browser is different, but most have a relatively simple mechanism for installation. Then, when you access a web application that demands client certification, your browser can supply a client certificate. If this is on the approved list of the server's allowed certificates, the transaction can continue.

From general use of the Internet, you're probably fairly used to this process working in reverse—where the server provides a certificate to your browser. Depending on your browser's security settings, you generally see a dialog box asking whether or not you want to trust the certificate the server is offering you (we hope signed by Thawte, VeriSign, or whomever). If you accept, the transaction can continue, and the server's public key can be used to encrypt communications between you. So certificates provide the foundation for secure transport as well as dealing with the issue of identify.

on the job

There's nothing stopping a web server vendor from supporting its own style of authentication and permitting new values (other than BASIC, FORM, DIGEST, and CLIENT-CERT) for the `<auth-method>` element. Of course, a web application subscribing to a vendor-specific authentication mechanism will almost certainly not transfer cleanly to a different vendor's web container.

EXERCISE 5-3

Setting Up FORM Authentication

In this exercise you'll set up authentication for the servlet you built in the previous exercise. You'll use a custom web page for the login, so the `<auth-method>` will be FORM.

The context directory for this exercise is ex0503, so set up your web application structure under a directory of this name.

Set Up the Deployment Descriptor

1. Copy the deployment descriptor web.xml from the previous exercise (ex0502) into the WEB-INF directory for this exercise (under ex0503). You'll still be using CheckedServlet, declared in the deployment descriptor. When you copy it forward to this web application (in a few steps' time), you may choose to change the package structure (as the solution code does). In that case, don't forget to change the package name in the `<servlet-class>` element here in web.xml.
2. Define a `<login-config>` element.
3. Within `<login-config>`, define an `<auth-method>` element with the value FORM.
4. Still with `<login-config>`, define a `<form-login-config>` element, using the appropriate subelements to set up a login page called "login.html" and an error page called "error.html." Predictability may be boring, but your support staff will love you for it!

Define Error and Login Pages

5. Create a web page called "error.html." Put any text you like into this page, just so that you will recognize it as the error page.
6. Create a web page called "login.html." Include a form with a method of POST and an action of j_security_check. Within the form, include a text field (named j_username) for the login name, and a password field (named j_password) for the password. See Chapter 1 if you need a refresher on constructing form fields. Don't forget a submit button in the form.

Copy CheckedServlet

7. Copy CheckedServlet from the previous exercise (it should be somewhere under ex0502/WEB-INF/classes), and paste it into the current web application's directory structure. If you follow the pattern of the solution code, you'll copy the source and change the package structure to reflect this exercise. If you do that, don't forget to change the package statement in the source code, or you won't be able to compile the servlet!

Run the Application

8. Deploy the application—it should start up without error or warning messages.
9. Try running the servlet with an appropriate URL, such as

```
http://localhost:8080/ex0503/CheckedServlet
```

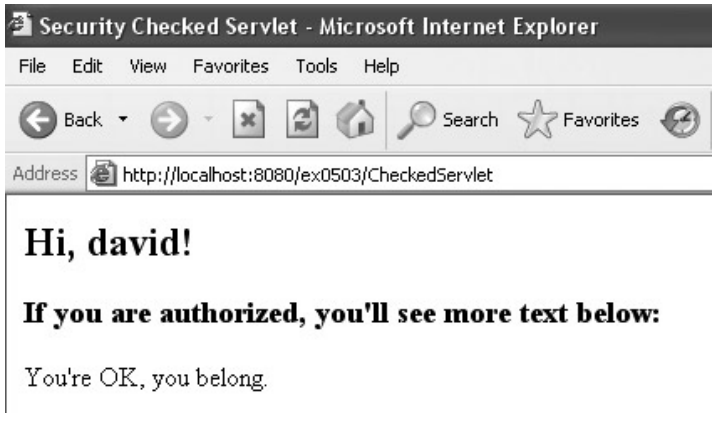
10. You should be redirected to the login page. Log in with an invalid user ID and/or password. Make sure that you get the error page. If you don't get the expected result (at any point), close down all browser windows and restart the browser—in case the browser is caching users and passwords for the duration of the session.
11. Now add a user into your server's user directory, for the lowlife role. For Tomcat, this means editing the tomcat-users.xml file in the <TOMCAT-INSTALL-LOCATION>/conf directory. You can see my additions (for user "david") to this file in bold:

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
 <role rolename="tomcat"/>
 <role rolename="role1"/>
 <role rolename="lowlife"/>
 <user username="tomcat" password="tomcat" roles="tomcat"/>
 <user username="both" password="tomcat" roles="tomcat,role1"/>
 <user username="role1" password="tomcat" roles="role1"/>
 <user username="david" password="tomcat" roles="lowlife"/>
</tomcat-users>
```

12. Restart the server. Re-access the servlet, and log on using your user ID and password. At this point, the servlet should display properly—everything except the text protected by the `isUserInRole()` method. Unlike our run of the servlet in the previous exercise, you should also see your authenticated user name displayed in the text. Refresh your memory of the source code if all this seems strange and unfamiliar.
13. To make the protected text visible, give your user the role of manager as well as lowlife. My tomcat-users.xml user entry now looks like this:

```
<user username="david" password="tomcat" roles="lowlife,manager" />
```

14. Restart the server (with Tomcat, this will in fact automatically add in the role of manager into the roles list in tomcat-users.xml).
15. Access the servlet again. Now all text should display. The solution page is shown in the following illustration.



CERTIFICATION SUMMARY

In this chapter you explored the world of web application security. First of all, you met some fundamental security terms as defined by the servlet specification:

- Authentication: proving you are who (or what) you claim to be
- Authorization: ensuring that an authenticated party gains access only to the resources he or she is entitled to
- Data Integrity: ensuring that any messages passed through a network have not been tampered with in transit
- Confidentiality (data privacy): ensuring that the information in a message is available only to users authorized to see that information

You learned that there were several methods for supplying authentication, ranging from the simple (user IDs and passwords) to the sophisticated (digital certificates). You saw that authorization—in web application terms—first of all involves identifying resources as identified by their URLs and the HTTP methods used to access them. Second, you saw that these can be associated with logical roles.

Third, you learned that each web server has its own method for associating those logical roles with specific users in a registry of users. You learned that encryption is a key component for both data integrity and confidentiality, and that at the root of encryption lie matching pairs of public and private encryption keys (often known as asymmetric keys). You learned that one of the matching pair can encrypt a message, but then only the other half can do the de-encryption. You saw that private keys are held very private and secure by their owners but that public keys can be shared with any interested parties. Armed with private and public key knowledge, you learned that data integrity can be ensured by encrypting a message with a private key. Anyone with the public key can de-encrypt such a message—provided no tampering has taken place, for that would render the message useless. You further learned that confidentiality can be ensured by using keys in the other direction. Using someone's public key to encrypt the message, you can rest assured that only that someone can read the message—as only that person will have a private key.

We then moved on to look at web applications in more detail—and in particular, the role played by deployment descriptor definitions. You learned that the preferred way of providing web application security is through "declarative" means—in other words, putting information inside the deployment descriptor instead of in code. You had a glimpse of programmatic security, using such methods as `HttpServletRequest.getRemoteUser()` and `HttpServletRequest.isUserInRole(String roleName)`—but quickly dismissed those because they are not core SCWCD syllabus items!

Back in the deployment descriptor, you met three top-level elements controlling security: `<security-constraint>`, `<login-config>`, and `<security-role>`. You saw that a `<security-constraint>` consists of the subelements `<web-resource-collection>`, `<auth-constraint>`, and `<user-data-constraint>`. You learned that the web resource collection defines the actual resources to protect, that the authority constraint associates the resources with logical roles (the authorization element), and that the user data constraint can provide guarantees of data integrity and probably, in addition, confidentiality.

You went on to learn that the `<web-resource-collection>` element has several subelements: a mandatory `<web-resource-name>`, some optional `<description>` lines, at least one `<url-pattern>`, and some optional `<http-method>`s. You heard that the web resource name has no technical significance and is simply there to help administrators. You learned that the `<url-pattern>` element behaves in just the same way as it does within servlet and filter mappings, with the same four possibilities for values: exact path (/exactmatch), path prefix (longest match first) (/partial/*), extension matching (*.jsp), and default (/). You

saw that you can protect this URL pattern for specific HTTP methods, but leaving out the <http-method> element implies that every HTTP method used on this URL will be subject to the same authority and user data constraints.

From there, you looked at <auth-constraint>, which simply consists of an optional number of <role-name>s. You learned that leaving out any role names and supplying a "no value" authority constraint is equivalent to saying that no user in any role can access the resource. You saw that this is an overriding setting: Even if other security constraints give permissions to the same resource, this "no-value" authority setting takes precedence and blocks access. You also met the special <role-name> with a value of asterisk (*), and learned that this is a shorthand way of saying that all role names defined in the <security-role> element are allowed access to the resource. You learned that when you have separate web resource collections but with the same URL pattern/HTTP methods, protected by separate sets of authority constraints, that the authority constraints should be considered "added together."

After that, you examined <user-data-constraint> and found that this has one subelement—<transport-guarantee>—which dictates how communication between client and server should be handled. You learned that <transport-guarantee> has three possible values—NONE (which is equivalent to using no user data constraint at all; there are no guarantees made about client/server communication), INTEGRAL (which promises data integrity between client and server), and CONFIDENTIAL (which promises confidentiality, i.e., data privacy, in addition to data integrity). You also saw that web servers generally use SSL (secure sockets layer) as the network transport layer to deal with encrypted messages passed in INTEGRAL and CONFIDENTIAL communication.

Before looking at the next top-level deployment descriptor element, <login-config>, in any detail, you were introduced to <security-role>. You learned that there can be as many <security-role> elements in the deployment descriptor as required, each containing a single <role-name> element. You saw that the function of this is simply to list all the valid logical authorization roles known to the web application. You also learned that the web container is supposed to validate the use of role names elsewhere in the deployment descriptor (in authority constraints, and as the <role-link> in <servlet> elements) by reference to this list of security roles. You incidentally learned that a valid role name value can contain characters and numbers—even begin with a number—but should not contain embedded white space or punctuation.

In the third and final section of the chapter, you learned in more detail about authentication types and saw how these are controlled by the <login-config> element. You learned that the first and most crucial subelement of <login-config>

is called <auth-method> (authentication method) and that this has four valid values: BASIC, DIGEST, FORM, and CLIENT-CERT—and vendor-specific values are possible.

You learned about BASIC authentication and saw that this works by the server issuing a standard response to a browser requesting a secure resource. You saw that this will prompt any standard browser to launch a dialog box, requesting a user ID and password. You learned that this dialog box can display a piece of text called a realm name and that this text can be set using the <realm-name> subelement of <login-config>. Although no server-side validation is performed on the realm name text, you saw that it should in some way describe the server-side registry that is used to validate user authentication credentials. You learned that by itself, BASIC authentication provides practically no security for authentication details—that this is limited to encoding the password with an easily-reversed Base64 algorithm.

Then you met DIGEST authentication. You saw how this is a more secure method as far as the transmission of authentication details go. You saw that this security is provided by a one-way encryption process (the digest algorithm) and that the resulting "digest" will reveal nothing. You learned that the principle of authentication using digests goes like this: Both the client and server use the same digest algorithm on the same input data, which includes user and password details. You saw that the server can compare both digests, and if they match, the client is deemed to be authenticated. To prevent anyone sending an old digest to the server (pretending to be the real client), you saw that the digests are never repeated, because part of the input data is a semirandom "nonce" value generated by the server. You learned in passing that a realm name (<realm-name>) can be associated with DIGEST authentication, which then appears in the browser's dialog box requesting user name and password—and that BASIC and DIGEST are the only forms of authentication for which realm names are used.

You then moved on to perusing FORM authentication and were introduced to the idea that this represents little more than a cosmetic improvement over browser dialog boxes. You learned that this method of authentication demands its own subelement within <login-config>, called <form-login-config>. You saw that this comprises two subelements, <form-login-page> and <form-error-page>. You learned that the value of <form-login-page> points to the location of a custom login page for the input of user ID and password, and that the <form-error-page> defines the location of a custom error page. You were warned that these values hold a path to a filename and must begin with a forward slash ("/") to denote the root of the web context. You learned the mechanism of FORM authentication: that when a user accesses a secure resource in a web application for

the first time, he or she is redirected to the custom login page. If the user provides a bona fide user ID and password, the server redirects the user to the secure resource, and subsequent access to secure resources in the same web application should not require re-authentication. You also learned that on failure to authenticate through the custom login page, the server redirects the user to the custom error page. You learned that the custom error page doesn't obey any particular rules, but that there are some formalities for the custom login page: It must contain an HTML <FORM> whose method is POST; the action of the form must be called j_security_check (exactly that, all lowercase and complete with underscores); and that there must be two input fields on the form, one named j_username and the other j_password (no prizes for guessing their purpose). You learned that FORM authentication is no more secure than BASIC, for authentication details are passed in plain text in the request body.

You had seen by now that the three authentication methods—BASIC, DIGEST, and FORM—are generally supplemented with SSL encryption when used anywhere but over a very secure network. You finally met CLIENT-CERT authentication and saw that certificates are always transmitted over a secure network using SSL (so a transport guarantee of CONFIDENTIAL is firmly implied, even though the servlet specification doesn't say you have to have one). You learned that a client certificate contains a public key and attaches identification details to the public key. You saw that the identification details can be underwritten by a third-party certification authority, who can digitally sign your certificate—meaning that anyone using your certificate is assured that you are who you say you are, at least in the eyes of the certification authority. You learned that you supply your client certificate in place of a user ID and password to a server that secures resources—and that the server checks this certificate against a known list of certificates.

TWO-MINUTE DRILL

Security Mechanisms

- ❏ There are four security mechanisms detailed by the servlet specification: authentication, authorization (access to controlled resources), data integrity, and confidentiality (data privacy).
- ❏ Authentication is the process of proving you are who (or what) you claim to be.
- ❏ The servlet specification has much to say on client-to-server authentication: clients (human beings or other systems) proving their identity to our web applications.
- ❏ The servlet specification has little to say about server-to-client authentication (do I trust this web application?); however, it is still an aspect you should be aware of.
- ❏ Authentication can be achieved through basic means (user IDs and passwords) versus complex means (digital certificates). The trade-off is usually simplicity versus security.
- ❏ Authorization is the process of ensuring that an authenticated party gains access only to the resources it is entitled to.
- ❏ In servlet spec terms, this process means identifying resources by their URLs (and the HTTP methods used to access them), and associating these with logical roles.
- ❏ It's the web server's job to supply some means or other for relating these logical roles to specific users or groups of users. There's no standard way to achieve this—it is server specific. Indeed, a web server may have more than one way of approaching this, for the registries (databases) of users with which it has to interact may be quite diverse.
- ❏ Data integrity is the process of ensuring that any messages passed through a network have not been tampered with in transit.
- ❏ Data integrity very often involves encrypting the contents of a message. Integrity comes about because any tampering with an encrypted message will render the message impossible to decrypt.
- ❏ Confidentiality (data privacy) goes one step further than data integrity, by promising that the information in a message is available only to users authorized to see that information.

- Again, encryption is the key to confidentiality. The encryption used by any particular web server may be stronger than the encryption used to ensure integrity, though very often the same algorithms are used for integrity and confidentiality.
- The encryption process usually involves public and private keys (sometimes called asymmetric—they're a matched pair, but not identical).
- Private keys are kept strictly private, whether client-side or server-side.
- Public keys are broadcast to interested parties.
- If a server uses a client's public key to encrypt a message, only the client's private key can decode it.

Deployment Descriptor Security Declarations

- Inside the root element `<web-app>`, there are three top-level elements devoted to security: `<security-constraint>`, `<login-config>`, and `<security-role>`.
- `<security-constraint>` is the biggest and most complex of these three.
- Its purpose is to associate resources (and HTTP methods executed on those resources) with logical roles for authorization, and also with guarantees on resource security in transit over a network.
- `<security-constraint>` has three main subelements—`<web-resource-collection>`, `<auth-constraint>`, and `<user-data-constraint>`.
- The first of these main subelements, `<web-resource-collection>`, defines the resources to be secured.
  - The first element inside `<web-resource-collection>` must be `<web-resource-name>`, whose value is a logical name to describe the group of resources protected.
  - Next, `<web-resource-collection>` defines the URL patterns to protect using one or more `<url-pattern>` subelements.
  - The value of a URL pattern is a path to a resource (or resources) within the web application.
  - Valid values for URL patterns are the same as for servlet mappings and filter mappings: exact path (/exactmatch), path prefix (longest match first) (/partial/*), extension matching (*.jsp), and default servlet (/). As for servlet mappings, web resource collection URL patterns types are processed in that order.

- Any resource can be protected—static or dynamic.
- `<web-resource-collection>` optionally contains `<http-method>` elements (zero to many).
- Use `<http-method>` to associate given HTTP methods with the resource protected. You may want to associate specific protection on certain resources just for the HTTP POST method, for example.

- The second subelement of `<security-constraint>` is `<auth-costraint>`.
- `<auth-constraint>` lists the named roles authorized to the resources defined in the web resource collection.
  - Each named role is placed in a subelement called `<role-name>`.
- An authority constraint with no value (e.g., `<auth-constraint />`) denotes that there should be no permitted access whatsoever to the resource.
- Identical URL patterns are protected, for the identical HTTP methods may appear in separate web resource collections—effectively protecting the same resource. In this case, all the role names specified in all the authority constraints are "added together"—a user in any one of those roles can access the resource with the given HTTP method.
- There's an exception to this authority constraint addition rule: If the no-value authority constraint (e.g., `<auth-constraint />`) is one of several authority constraints protecting the same resource, it *overrides everything else*—no access is allowed.
- The third and final subelement of `<security-constraint>` is `<user-data-constraint>`.
- `<user-data-constraint>` serves to define guarantees on the network used to transmit resources to clients.
  - `<user-data-constraint>` contains the element `<transport-guarantee>` for this purpose.
  - `<transport-guarantee>` has three valid values: NONE, INTEGRAL, and CONFIDENTIAL.
  - A value of NONE means that no guarantee is offered on the network traffic—and is equivalent to omitting `<user-data-constraint>` altogether (the default).
  - A value of INTEGRAL means that the web server must be able to detect any tampering with HTTP requests and responses for protected resources.

- ❏ A value of CONFIDENTIAL means that the web server must ensure the content of HTTP requests and responses so that protected resources remain secret to all but authorized parties.
- ❏ It is common practice for a web server to employ SSL (secure sockets layer) as the network transport layer to fulfill INTEGRAL and CONFIDENTIAL guarantees.

❏ Neither `<auth-constraint>` nor `<user-data-constraint>` is a mandatory element of `<security-constraint>`. Either or both may be used to protect resources. It makes little sense to go to the trouble of defining a `<web-resource-collection>`, then to omit both these elements, but it is legal to do so.

❏ After `<security-constraint>`, the next security-related deployment descriptor element is `<login-config>`. This determines how users (or other systems) authenticate themselves to a web application.

❏ The last security-related deployment descriptor element is `<security-role>`.

❏ Each `<security-role>` element (there can be as many as you like) must contain one `<role-name>` element.

❏ The value of `<role-name>` is a logical role name against which resources are authorized.

❏ Role names listed here may be used in the `<role-name>` element in `<auth-constraint>` and in `<security-constraint>`, and in the `<role-link>` element in `<security-role-ref>` in `<servlet>`.

❏ A logical role name must not contain embedded spaces or punctuation.

Authentication Types

❏ Authentication types are set up in the `<login-config>` deployment descriptor element.

❏ The subelement `<auth-method>` names the authentication scheme. There are four standard values: BASIC, DIGEST, FORM, and CLIENT-CERT.

❏ It's possible to name a vendor-specific authentication scheme not covered by the four standard values above (but then your web application will be tied to that vendor's application server).

❏ The simplest `<auth-method>` of BASIC will trigger a browser to show a standard dialog when accessing a secure resource for the first time in your

web application. The dialog allows entry of a user name and password, and displays a realm name.

- The realm name (appearing in the browser dialog) can be set with the `<realm-name>` subelement of the deployment descriptor and should name a registry of user credentials accessible from your web server.
- The password details are Base64-encoded when passed from the browser to your web application. This provides a small measure of protection, but very little—Base64 decoders are freely available to hackers.
- BASIC is not useless, however, if SSL is used to encrypt all network traffic from browser to web server.
- DIGEST authentication (`<auth-method>DIGEST</auth-method>`) imposes better security by encrypting authentication information.
- The encryption process uses as input transient server information (often called a "nonce") and authentication information (including the user ID and password).
- The output from the encryption process is called a digest.
- The digest can't be de-encrypted.
- The digest is sent to the server. The server has access to all the same input values to make its own digest. If the digests match, the user is authenticated.
- Not all browsers support digests or make the digests in the same way—this makes adoption of the DIGEST method more difficult.
- FORM authentication associates a custom web page with the login process, as an alternative to a browser dialog.
- The custom web page for logging in must contain an HTML <FORM> whose action is j_security_check and whose method is POST.
- The form must include input fields named j_username and j_password.
- A custom error page must also be provided—there are no rules about the HTML for this.
- The login and error pages are then specified in the deployment descriptor in the `<form-login-config>` element, which is a subelement of `<login-config>`.
- The login page goes in the `<form-login-page>` subelement of `<form-login-config>`.

- ❑ The error page is specified in the `<form-error-page>` subelement of `<form-login-config>`.
- ❑ The values for these elements *must* begin with a forward slash, which denotes the root of the web context.
- ❑ `<form-login-config>` is — obviously — only relevant when the `<auth-method>` is FORM.
- ❑ The first secured resource you request in a web application will cause you to be redirected to the login page.
- ❑ If there is an error logging in, you are redirected to the error page.
- ❑ Log-in information (user, password) is not protected in any way across the network with FORM authentication. As for BASIC authentication, you need to also use a secure protocol such as SSL if greater protection is needed.
- ❑ CLIENT-CERT is the fourth and final authentication method. It is the most secure but also the trickiest to set up.
- ❑ This method relies on asymmetric keys — that is, a pair of keys, one public (available to anyone) and one private (kept secure on the key owner's hardware). Anything encrypted with the public key can be decrypted with the private key — and vice versa.
- ❑ The client (and would-be digital certificate owner) first generates a private and public key. The client sends the public key — and other information — to a third-party certificate authority. The certificate authority binds this information and the client public key into a certificate.
- ❑ The certificate authority adds a digital signature encrypted with its private key, which makes a digest of information already in the certificate.
- ❑ This process serves two purposes: (1) to prove the certificate is vouched for by the certificate authority (only its public keys can be used to read the signature) and (2) to prevent tampering with any aspect of the certificate.
- ❑ The certificate is returned to the client, who installs it in his or her browser (or other client device).
- ❑ When the server requests authentication from the client browser, the browser supplies the certificate. If the certificate is on the server's approved list of certificates, authentication takes place.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there might be more than one correct answer. The number of correct choices to make is stated in the question, as in the real SCWCD exam.

Security Mechanisms

1. Which security mechanism proves that data has not been tampered with during its transit through the network? (Choose one.)
   - A. Data validation
   - B. Data integrity
   - C. Authentication
   - D. Packet sniffing
   - E. Data privacy
   - F. Authorization
2. Which security mechanism limits access to the availability of resources to permitted groups of users or programs? (Choose one.)
   - A. Authentication
   - B. Authorization
   - C. Data integrity
   - D. Confidentiality
   - E. Checksum validation
   - F. MD5 encryption
3. Which of the following deployment descriptor elements play some part in the authentication process? (Choose three.)
   - A. `<login-config>`
   - B. `<transport-guarantee>`
   - C. `<role-name>`
   - D. `<auth-method>`
   - E. `<form-error-page>`
   - F. `<security-role-ref>`

4. In a custom security environment, for which security mechanisms would a filter be incapable of playing any useful part? (Choose one.)
   A. Authentication
   B. Authorization
   C. Data integrity
   D. Confidentiality
   E. All of the above
   F. None of the above

5. Review the following scenario; then identify which security mechanisms would be important to fulfill the requirement. (Choose two.)

   An online magazine company wishes to protect part of its web site content, to make that part available only to users who pay a monthly subscription. The company wants to keep client, network, and server processing overheads down: Theft of content is unlikely to be an issue, as is abuse of user IDs and passwords through network snooping.

   A. Authorization
   B. Authentication
   C. Indication
   D. Client certification
   E. Data integrity
   F. Confidentiality

Deployment Descriptor Security Declarations

6. Identify which choices in the list below show immediate subelements for `<security-constraint>` in the correct order. (Choose two.)
   A. `<security-role-ref>,<auth-method>,<transport-guarantee>`
   B. `<web-resource-name>,<auth-constraint>`
   C. `<web-resource-collection>,<auth-constraint>,<user-data-constraint>`
   D. `<auth-method>,<web-resource-name>`
   E. `<web-resource-collection>,<auth-constraint>`
   F. `<auth-constraint>,<web-resource-name>`
   G. `<web-resource-collection>,<transport-guarantee>,<auth-method>`

7. Identify valid configurations for the `<transport-guarantee>` element in the deployment descriptor. (Choose four.)
   A. `<transport-guarantee>CONFIDENTIAL</transport-guarantee>`
   B. `<transport-guarantee>ENCRPYTED</transport-guarantee>`
   C. `<transport-guarantee>FAILSAFE</transport-guarantee>`
   D. `<transport-guarantee>ENCIPHERED</transport-guarantee>`
   E. Absent altogether from the deployment descriptor
   F. `<transport-guarantee />`
   G. `<transport-guarantee>NONE</transport-guarantee>`
   H. `<transport-guarantee>INTEGRAL</transport-guarantee>`

8. Given the following incomplete extract from a deployment descriptor, what are possible ways of accessing the protected resource named TheCheckedServlet? (Choose three.)

```
<security-constraint>
    <web-resource-collection>
      <web-resource-name>TheCheckedServlet</web-resource-name>
      <url-pattern>/CheckedServlet</url-pattern>
    </web-resource-collection>
    <auth-constraint />
</security-constraint>
<security-constraint>
    <web-resource-collection>
      <web-resource-name>TheCheckedServlet</web-resource-name>
      <url-pattern>/CheckedServlet</url-pattern>
      <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>bigwig</role-name>
    </auth-constraint>
</security-constraint>
```

   A. Via another URL pattern (if one is set up elsewhere within the deployment descriptor).
   B. Any authenticated user can access the resource.
   C. Any user (authenticated or not) can access the resource.
   D. Via `RequestDispatcher.include()`.
   E. Via `RequestDispatcher.forward()`.
   F. Via the URL pattern /CheckedServlet, provided the user is authenticated and has bigwig as a valid role.

9. Which of the following might a web server consider important in ensuring a transport guarantee of CONFIDENTIAL? (Choose four.)

A. Base64 encoding
B. Server-side digital certificates
C. Symmetric keys
D. Asymmetric (public/private keys)
E. SSL
F. Client-side digital certificates

10. (drag-and-drop question) The following illustration shows the declaration of a security constraint in a deployment descriptor. Match the lettered blanks in the declaration with numbered choices from the list on the right.

```
<security-constraint>
  <  A  >
    <  B  >
      TheCheckedServlet
    </  C  >
    <  D  >/MyServlet</  E  >
    <http-method>  F  </http-method>
    <http-method>  G  </http-method>
  </  H  >
  <auth-  I  >
    <role-name>J</role-name>
  </auth-  K  >
  <user-data-constraint>
    <  L  >
      CONFIDENTIAL
    </  M  >
  </user-data-constraint>
</security-constraint>
```

1	collection
2	check
3	url-collection
4	transport-guarantee
5	resource-collection
6	?
7	method
8	web-resource-name
9	POST
10	transport-constraint
11	web-resource-collection
12	url-pattern
13	constraint
14	*
15	GET

Authentication Types

11. The following web page is defined as the custom form login page for authentication. Assuming that you have attempted to access a protected resource and been redirected to this web page, what is the result of filling in the user name and password fields and pressing SUBMIT? (Choose one.)

```
<html>
<head><title>Login Form</title></head>
<body>
<form action="jsecuritycheck" method="POST">
<br />Name: <input type="text" name="jusername" />
<br />Password: <input type="password" name="jpassword" />
<br /><input type="submit" value="Log In" />
</form>
</body>
</html>
```

A. You will not be redirected to this page in the first place.
B. HTTP 401 or 403 error (forbidden/not authorized).
C. HTTP 404 error (page not found).
D. HTTP 500 error (server error).
E. The page is redisplayed.

12. (drag-and-drop question) The following illustration shows the declaration of a login configuration in a deployment descriptor. Match the lettered blanks in the declaration with numbered choices from the list on the right.

```
<login-config>
  <[ A ]>BASIC</[ B ]>
  <[ C ]>osborne<[ D ]>
</login-config>

<login-config>
  <[ E ]>[ F ]</[ G ]>
</login-config>

<login-config>
  <[ H ]>FORM</[ I ]>
  <[ J ]>
    <[ K ]>[ L ]</[ M ]>
    <[ N ]>[ O ]</[ P ]>
  </[ Q ]>
</login-config>
```

1	error.html
2	error-page
3	realm
4	domain
5	realm-name
6	/login.jsp
7	auth-method
8	auth-constraint
9	form-login
10	CLIENT-CERT
11	CONFIDENTIAL
12	form-login-config
13	form-error-page
14	login-page
15	form-login-page
16	login.html
17	/error.jsp

13. What is the result of the following login configuration? (Choose one.)

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>login.html</form-login-page>
    <form-error-page>error.html</form-error-page>
  </form-login-config>
</login-config>
```

A. Application fails to start.
B. Application starts with warning errors.
C. Application runs, but access to a protected resource results in an HTTP 404 error.
D. Application runs and presents login form on a user's first access to a protected resource.

14. Which of the following subelements might you expect to find in the `<login-config>` for BASIC authorization? (Choose two.)

A. `<auth-constraint>`
B. `<form-login-config>`
C. `<role-name>`
D. `<form-login-page>`
E. `<realm>`
F. `<auth-method>`
G. `<realm-name>`

15. Which of the following subelements would you not expect to find in the `<login-config>` for CLIENT-CERT authorization? (Choose four.)

A. `<auth-constraint>`
B. `<role-name>`
C. `<form-login-page>`
D. `<auth-method>`
E. `<realm-name>`

LAB QUESTION

In this lab, you're going to attempt to set up secure transport over SSL using a server-side certificate. Exactly how you do this is somewhat dependent on your environment. Because I'm working with the Tomcat server running under Windows XP, my instructions are biased toward that environment. However, even if you're using another server and operating system, it shouldn't be too hard to discover your local equivalents for what I describe below.

First, we need a certificate, and we'll use Java's J2SDK facilities for creating one. Here are the (Windows XP) blow-by-blow instructions for this:

- Get to a command prompt. You'll need access to the keytool command (in <your J2SDK installation directory>\bin).
- Enter the following command:

```
keytool -genkey -alias webcert -keyalg RSA
```

- Follow the on-screen prompts—the questions are straightforward! Take careful note of the password you create—you'll need it later.
- By default (on a Windows machine), the key information you have just generated is stored in your home directory—for me, that's C:\Documents and Settings\David. Look for a file there called .keystore.
- In the absence of involving (and paying for) the services of a VeriSign or Thawte, you are going to self-certify your key information. Do this by entering the command

```
keytool -selfcert -alias webcert
```

- You'll be prompted for the password you created earlier.

That's your certificate, created and safely stored in a Java-style keystore.

Next, you need your server to recognize the certificate you have just set up. For Tomcat, that means pointing a configuration item called a "connector" toward your keystore file. Find Tomcat's main configuration file—server.xml—in <Tomcat Installation Directory>/conf. Because you are going to make changes to this file—and restore the original configuration later—*make a copy of server.xml.*

I'm running the 5.5 version of Tomcat, which meant I had to find the lines referring to the secure connector shown below. By default, this connector configuration is commented out (with <!-- --> tags), so I uncommented the element and added the lines in bold:

```
<!—Define a SSL Coyote HTTP/1.1 Connector on port 8443—>
<Connector port="8443"
  maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
```

```
enableLookups="false" disableUploadTimeout="true"
acceptCount="100" debug="0" scheme="https" secure="true"
clientAuth="false" sslProtocol="TLS"
keystoreFile="C:\Documents and Settings\David\.keystore"
keypass="passwordyouchose">
```

Note that the value for the keypass attribute is the password you supplied when making the keystore, so substitute the right value in the XML above. I'm hoping the steps for you are very similar—they're slightly different for earlier versions of Tomcat (but then why are you using an earlier version?)—and clearly, on a different server you'll need to find your own salvation!

All that remains is to implement a web application with a resource that's protected by a CONFIDENTIAL transport guarantee. Install this on your server, point your browser to the protected resource, and see what happens.

SELF TEST ANSWERS

Security Mechanisms

1. ☑ **B** is the correct answer. The process of ensuring that data has not been tampered with in transit through a network is described as proving the integrity of the data.
   ☒ **A** is incorrect: Data validation is not a term generally applied to security (it's more usually applied to checking the input data on a form, for example). **C** is incorrect: Authentication is the process of proving you are who you claim to be. **D** is incorrect: Packet sniffing is a technique for monitoring TCP/IP network traffic, used for good or evil—so is not in itself any kind of security mechanism. **E** is incorrect—just because data has not been tampered with in transit does not mean it's remained private. Finally, **F** is incorrect: Authorization is the process of determining what resources an authenticated user can use.

2. ☑ **B** is the correct answer. Authorization is the security mechanism that limits the availability of resources to permitted groups.
   ☒ **A** is incorrect, even though authentication (identifying who you are) is a prerequisite for authorization. **C** and **D** are incorrect: The integrity and confidentiality of network traffic have no bearing on resources to which you may or may not be authorized. **E** and **F** are incorrect—checksum validation is a (weak) technique for proving integrity, whereas MD5 encryption is a (strong) technique to assist in encryption (often of passwords), but neither has to do with authorization.

3. ☑ **A**, **D**, and **E** are the correct answers. The deployment descriptor element that has to do with authentication is `<login-config>`, and `<auth-method>` and `<form-error-page>` are subelements (or sub-sub-elements) of `<login-config>`.
   ☒ **B** is incorrect, for `<transport-guarantee>` has to do with specifying data integrity and confidentiality. **C** is incorrect: `<role-name>` turns up in several places in the deployment descriptor. But regardless of position, its role (no pun intended) is always in authorization. **F** is incorrect: `<security-role-ref>` can be found as a subelement of `<servlet>`, and has to do with programmatic authorization (not authentication).

4. ☑ **F** is the correct answer. Filters can potentially play a part in all four aspects of security; hence, they are not excluded from any aspect listed, and "none of the above" is the correct answer. Filters can intercept user and password data on requests, and perform look-ups on appropriate user directories, and hence perform authentication. Because filters are tied to resources by URL pattern, and can in any case look at the requested URL, they can further make a determination about whether a user is authorized to a resource. Furthermore, filters can perform any check you want on incoming data to prove its integrity and perform any amount of encryption and de-encryption in a chain of confidentiality. Of course, to do any or

all of this yourself in filter code might involve ignoring what your web container provides for free, but the point remains that a filter can be used for almost any security purpose when a web container mechanism is insufficient. And to go further than that—it could well be that your web container (under the covers) is already making use of filters to provide the standard J2EE security requirements described in this chapter.

☒ **A**, **B**, **C**, **D**, and **E** are incorrect according to the reasoning in the correct answer.

5. ☑ **A** and **B**. **A** (authentication) is necessary to identify subscribed users. **B** (authorization) is necessary to tie in the protected content with subscribing users.

☒ **C** is incorrect and is simply a red herring word: "indication" is not a security mechanism. **D** is incorrect—although client certification is a form of authentication (which will be required in some form), it's the least likely to be used in this circumstance—certificates imply heavy use of encryption, which will add to processing overheads on client and server (and this is contrary to the company's requirements). **E** and **F** are incorrect, for while they are bona fide security mechanisms, the requirements make it clear that absolute data integrity and privacy are not crucial.

Deployment Descriptor Security Declarations

6. ☑ **C** and **E** are the correct answers. **C** gives the full set of the three top-level elements; **E** has only the first two (you don't have to have a `<user-data-constraint>`—the default of NONE for `<transport-guarantee>` is implied when this element is missing).

☒ **A** is incorrect, for it mixes in security elements from wholly different parts of the deployment descriptor and has `<transport-guarantee>` as an immediate child of `<security-constraint>` when it is a grandchild. **B** is close, but still incorrect—`<web-resource-name>` is a subelement of `<web-resource-collection>`, which would be correct. **D** is incorrect: again, because of `<web-resource-name>`, and `<auth-method>`, which has strayed over from `<login-config>`! **F** is incorrect for similar reasons; **G** is incorrect: Only the first element is correct, and the other two are repeats of errors in earlier choices.

7. ☑ **A**, **E**, **G**, and **H** are correct. The valid values for `<transport-guarantee>` are CONFIDENTIAL, INTEGRAL, and NONE. In addition, having the element absent altogether is just fine (provided the parent element, `<user-data-constraint>`, is absent also)—and will be interpreted as equivalent to the element being present with a value of NONE.

☒ **B**, **C**, and **D** are incorrect—all have bogus values. **F** is incorrect: To have the element present with no value at all is illegal, and—even if not caught on application startup—may cause unpredictable results.

8. ☑ **A**, **D**, and **E** are the correct answers. **D** and **E** are correct because authority constraints are applicable only for direct client requests. Internal web application servlet code that forwards or includes a resource (even with the same URL pattern) entirely bypasses authority checking. Of course, you can protect the method calls to forward and include with your own programmatic authority checking if you wish. **A** is correct because it's not the actual resource you are protecting—rather, it's a URL pattern to a resource. Authority checking on another URL pattern is completely independent.
☒ **B** and **C** are both incorrect and show a misunderstanding of a "blank" `<auth-constraint />` element. The significance is that the URL pattern is denied to any user in any role—and because as the `<http-method>` element is missing, this applies to any HTTP method used to access the resource. **F** is incorrect: Although additional `<security-constraint>` elements for the same URL pattern are generally compounded together, the effect of the "blank" `<auth-constraint />` element is to override any other specification.

9. ☑ **B**, **C**, **D**, and **E** are the correct answers. **B** is correct—to establish SSL between client and server, the server has to supply the client with its public key—and the best way of doing that is typically through supplying a digital certificate. **D** is correct because asymmetric (public/private) keys are part and parcel of encryption. **C** is also correct—symmetric keys (usually of 128 bits) are usually used for secure communication once asymmetric keys have been used to pass the symmetric keys confidentially! Symmetric key encryption is faster than asymmetric encryption, hence the attraction of this approach. **E** is correct because SSL (secure sockets layer) is the usual network transport layer used for encrypted traffic.
☒ **A** is incorrect—the one thing in the list that won't do anything toward confidentiality is Base64 encoding, which is a highly insecure (but better than nothing) approach to obfuscating the password when BASIC authentication is used. **F** is also incorrect—client-side digital certificates may well contribute to confidentiality, but their use has more to do with highly secure authentication of clients. They are not essential for ensuring transport guarantees.

10. ☑ The correct pairings are **A**, **11**; **B**, **8**; **C**, **8**; **D**, **12**; **E**, **12**; **F**, **9**; **G**, **15** (or **F**, **15**; **G**, **9**); **H**, **11**; **I**, **7**; **J**, **14**; **K**, **7**; **L**, **4**; and **M**, **4**. Either you know your `<security-constraint>` elements or you don't!
☒ Beware of almost correct but plausible choices such as `<transport-constraint>` instead of `<transport-guarantee>`.

Authentication Types

11. ☑ **E** is the correct answer. The key to the question is noticing that the form HTML has something close to the right values for the form action, user name, and password fields—but

not close enough. The proper attribute values have underscores: j_security_check, j_username, j_password. So the form submits to the server. Instead of (as you might expect) an HTTP 404 error (because the resource jsecuritycheck doesn't exist), the server sees that no authorization data has been provided, so it simply redirects to the log-in page again.

☒ **A** is incorrect — only a deployment descriptor error would prevent forwarding to the page, not the HTML of the page itself. **B** is incorrect because you haven't had a chance to access anything with incorrect authentication information — you're still in the process of gathering that information. **C** is incorrect for reasons explained in the correct answer. And finally, **D** is incorrect: The server doesn't get to run a resource that is likely to terminate in an error.

12. ☑ The correct pairings are **A**, **7**; **B**, **7**; **C**, **5**; **D**, **5**; **E**, **7**; **F**, **10**; **G**, **7**; **H**, **7**; **I**, **7**; **J**, **12**; **K**, **15**; **L**, **6**; **M**, **15**; **N**, **13**; **O**, **16**; **P**, **13**; and **Q**, **12**. Again, there's no real wiggle room here — you just have to know what permutations of `<login-config>` make sense.

☒ No other combinations for `<login-config>` make sense from the choices that are available.

13. ☑ **A** is the correct answer — the application fails to start, for the application loading process ends (or should end) in a deployment descriptor parsing error. The values for `<form-login-page>` and `<form-error-page>` *must* begin with a forward slash ("/").

☒ **B**, **C**, and **D** are incorrect because of the reasoning in the correct answer.

14. ☑ **F** and **G** are the correct answers. **F** is correct because you must have an `<auth-method>` element set to a value of basic. **G** is correct because you might (optionally) expect to find a `<realm-name>` specified for BASIC (or DIGEST) authentication.

☒ **A** is incorrect because an `<auth-constraint>` — while being a valid element name — is part of `<security-constraint>`, not `<login-config>`. **B** and **D** are incorrect because they are valid `<login-config>` elements, but are only appropriate for FORM authentication. **C** is incorrect because the role-name element — while appearing in several places in the deployment descriptor — has no "role" in login configuration. Finally, **E** is incorrect — the element is `<realm-name>`, not `<realm>`.

15. ☑ **A**, **B**, **C**, and **E** are the correct answers. **A** and **B** belong to other elements entirely. **C** and **E** do belong to `<login-config>` but are appropriate to other forms of authentication, not CLIENT-CERT.

☒ **D** is incorrect, for `<auth-method>` is the one and only subelement of `<login-config>` that should appear when client certification is used for authentication (with the appropriate value of CLIENT-CERT, of course).

LAB ANSWER

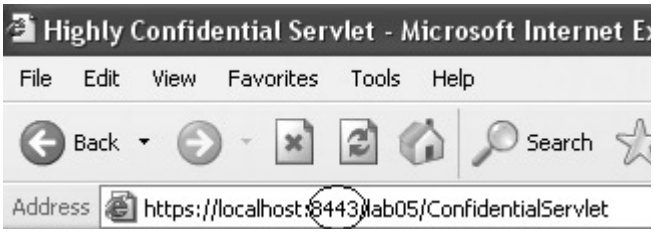
There is a WAR file from the CD called lab05.war, in the /sourcecode/chapter05 directory. This contains a sample web application, with a servlet called ConfidentialServlet appropriately protected in the deployment descriptor. Call the servlet using a URL such as

```
http://localhost:8080/lab05/ConfidentialServlet
```

You should find that you're prompted to accept a certificate in your browser (suitably dubious, given that you signed it yourself—but at least you know where it came from!). Then you gain access to the servlet, but you should notice the address line in your browser will subtly change to something like

```
https://localhost:8443/lab05/ConfidentialServlet
```

In other words, you have been redirected to the secure port, and https gives you the clue that SSL is being used for the transport layer. The solution page is shown in the following illustration:



This all assumes that you managed to create your certificate and configure your server correctly—and there, I'm afraid, I can't give you more help than I did in the lab instructions.

Don't forget to restore Tomcat to its original state without security. Stop Tomcat. Take server.xml in the conf directory and rename this to server.xml.secure. Take the copy you made of server.xml, and replace this in the conf directory (if you renamed the copy, make sure you rename it back to server.xml). Restart Tomcat.