Ideally, properly implemented security mechanisms will also provide the following functionality:

- Easy to administer
- Transparent to system users
- Interoperable across application and enterprise boundaries

Characteristics of Application Security

Java EE applications consist of components that can contain both protected and unprotected resources. Often, you need to protect resources to ensure that only authorized users have access. *Authorization* provides controlled access to protected resources. Authorization is based on identification and authentication. *Identification* is a process that enables recognition of an entity by a system, and *authentication* is a process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system.

Authorization and authentication are not required for an entity to access unprotected resources. Accessing a resource without authentication is referred to as unauthenticated or anonymous access.

These and several other well-defined characteristics of application security that, when properly addressed, help to minimize the security threats faced by an enterprise, include the following:

- **Authentication**: The means by which communicating entities (for example, client and server) prove to one another that they are acting on behalf of specific identities that are authorized for access. This ensures that users are who they say they are.
- **Authorization**, or **Access Control**: 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. This ensures that users have permission to perform operations or access data.
- **Data integrity**: The means used to prove that information has not been modified by a third party (some entity other than the source of the information). For example, a recipient of data sent over an open network must be able to detect and discard messages that were modified after they were sent. This ensures that only authorized users can modify data.
- **Confidentiality** or **Data Privacy**: The means used to ensure that information is made available only to users who are authorized to access it. This ensures that only authorized users can view sensitive data.
- **Non-repudiation**: The means used to prove that a user performed some action such that the user cannot reasonably deny having done so. This ensures that transactions can be proven to have happened.
- **Quality of Service (QoS)**: The means used to provide better service to selected network traffic over various technologies.

- **Auditing**: The means used to capture a tamper-resistant record of securityrelated events for the purpose of being able to evaluate the effectiveness of security policies and mechanisms. To enable this, the system maintains a record of transactions and security information.

Security Implementation Mechanisms

The characteristics of an application should be considered when deciding the layer and type of security to be provided for applications. The following sections discuss the characteristics of the common mechanisms that can be used to secure Java EE applications. Each of these mechanisms can be used individually or with others to provide protection layers based on the specific needs of your implementation.

Java SE Security Implementation Mechanisms

Java SE provides support for a variety of security features and mechanisms, including:

- **Java Authentication and Authorization Service (JAAS)**: JAAS is a set of APIs that enable services to authenticate and enforce access controls upon users. JAAS provides a pluggable and extensible framework for programmatic user authentication and authorization. JAAS is a core Java SE API and is an underlying technology for Java EE security mechanisms.
- **Java Generic Security Services (Java GSS-API)**: Java GSS-API is a token-based API used to securely exchange messages between communicating applications. The GSS-API offers application programmers uniform access to security services atop a variety of underlying security mechanisms, including Kerberos.
- **Java Cryptography Extension (JCE)**: JCE provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. Block ciphers operate on groups of bytes while stream ciphers operate on one byte at a time. The software also supports secure streams and sealed objects.
- **Java Secure Sockets Extension (JSSE)**: JSSE provides a framework and an implementation for a Java version of the SSL and TLS protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication to enable secure Internet communications.
- **Simple Authentication and Security Layer (SASL)**: SASL is an Internet standard (RFC 2222) that specifies a protocol for authentication and optional establishment of a security layer between client and server applications. SASL defines how authentication data is to be exchanged but does not itself specify the contents of that data. It is a framework into which specific authentication mechanisms that specify the contents and semantics of the authentication data can fit.

Java SE also provides a set of tools for managing keystores, certificates, and policy files; generating and verifying JAR signatures; and obtaining, listing, and managing Kerberos tickets.

For more information on Java SE security, visit its web page at `http://java.sun.com/javase/6/docs/technotes/guides/security/`.

Java EE Security Implementation Mechanisms

Java EE security services are provided by the component container and can be implemented using declarative or programmatic techniques (container security is discussed more in "Securing Containers" on page 774). Java EE security services provide a robust and easily configured security mechanism for authenticating users and authorizing access to application functions and associated data at many different layers. Java EE security services are separate from the security mechanisms of the operating system.

Application-Layer Security

In Java EE, component containers are responsible for providing application-layer security. Application-layer security provides security services for a specific application type tailored to the needs of the application. At the application layer, application firewalls can be employed to enhance application protection by protecting the communication stream and all associated application resources from attacks.

Java EE security is easy to implement and configure, and can offer fine-grained access control to application functions and data. However, as is inherent to security applied at the application layer, security properties are not transferable to applications running in other environments and only protect data while it is residing in the application environment. In the context of a traditional application, this is not necessarily a problem, but when applied to a web services application, where data often travels across several intermediaries, you would need to use the Java EE security mechanisms along with transport-layer security and message-layer security for a complete security solution.

The advantages of using application-layer security include the following:

- Security is uniquely suited to the needs of the application.
- Security is fine-grained, with application-specific settings.

The disadvantages of using application-layer security include the following:

- The application is dependent on security attributes that are not transferable between application types.
- Support for multiple protocols makes this type of security vulnerable.
- Data is close to or contained within the point of vulnerability.

For more information on providing security at the application layer, read "Securing Containers" on page 774.

Transport-Layer Security

Transport-layer security is provided by the transport mechanisms used to transmit information over the wire between clients and providers, thus transport-layer security relies on secure HTTP transport (HTTPS) using Secure Sockets Layer (SSL). Transport security is a point-to-point security mechanism that can be used for authentication, message integrity, and confidentiality. When running over an SSL-protected session, the server and client can authenticate one another and negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. Security is "live" from the time it leaves the consumer until it arrives at the provider, or vice versa, even across intermediaries. The problem is that it is not protected once it gets to its destination. One solution is to encrypt the message before sending.

Transport-layer security is performed in a series of phases, which are listed here:

- The client and server agree on an appropriate algorithm.
- A key is exchanged using public-key encryption and certificate-based authentication.
- A symmetric cipher is used during the information exchange.

Digital certificates are necessary when running secure HTTP transport (HTTPS) using Secure Sockets Layer (SSL). The HTTPS service of most web servers will not run unless a digital certificate has been installed. Digital certificates have already been created for the Application Server. If you are using a different server, use the procedure outlined in "Working with Digital Certificates" on page 788 to set up a digital certificate that can be used by your web or application server to enable SSL.

The advantages of using transport-layer security include the following:

- Relatively simple, well understood, standard technology.
- Applies to message body and attachments.

The disadvantages of using transport-layer security include the following:

- Tightly-coupled with transport-layer protocol.
- All or nothing approach to security. This implies that the security mechanism is unaware of message contents, and as such, you cannot selectively apply security to portions of the message as you can with message-layer security.
- Protection is transient. The message is only protected while in transit. Protection is removed automatically by the endpoint when it receives the message.
- Not an end-to-end solution, simply point-to-point.

For more information on transport-layer security, read "Establishing a Secure Connection Using SSL" on page 785.

This section discusses setting up users so that they can be correctly identified and either given access to protected resources, or denied access if the user is not authorized to access the protected resources. To authenticate a user, you need to follow these basic steps:

1. The Application Developer writes code to prompt the user for their user name and password. The different methods of authentication are discussed in "Specifying an Authentication Mechanism" on page 858.
2. The Application Developer communicates how to set up security for the deployed application by use of a deployment descriptor. This step is discussed in "Setting Up Security Roles" on page 782.
3. The Server Administrator sets up authorized users and groups on the Application Server. This is discussed in "Managing Users and Groups on the Application Server" on page 781.
4. The Application Deployer maps the application's security roles to users, groups, and principals defined on the Application Server. This topic is discussed in "Mapping Roles to Users and Groups" on page 784.

What Are Realms, Users, Groups, and Roles?

A realm is defined on a web or application server. It contains a collection of users, which may or may not be assigned to a group, that are controlled by the same authentication policy. Managing users on the Application Server is discussed in "Managing Users and Groups on the Application Server" on page 781.

An application will often prompt a user for their user name and password before allowing access to a protected resource. After the user has entered their user name and password, that information is passed to the server, which either authenticates the user and sends the protected resource, or does not authenticate the user, in which case access to the protected resource is denied. This type of user authentication is discussed in "Specifying an Authentication Mechanism" on page 858.

In some applications, authorized users are assigned to roles. In this situation, the role assigned to the user in the application must be mapped to a group defined on the application server. Figure 28–6 shows this. More information on mapping roles to users and groups can be found in "Setting Up Security Roles" on page 782.

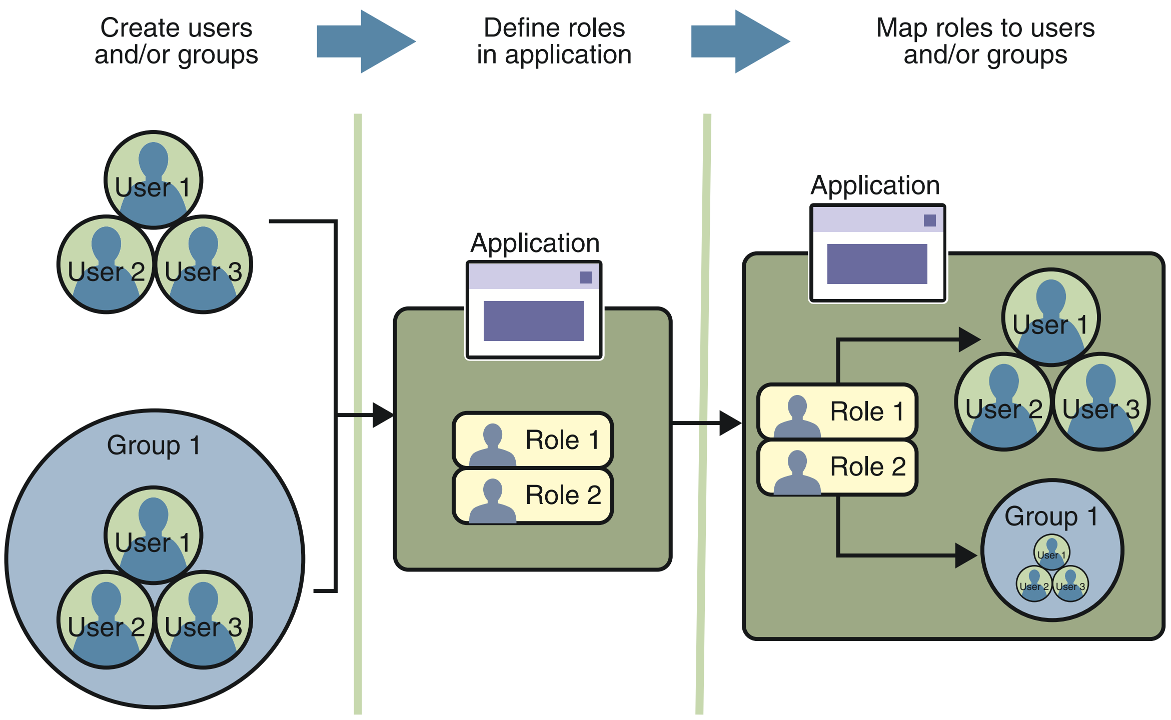

FIGURE 28–6 Mapping Roles to Users and Groups

The following sections provide more information on realms, users, groups, and roles.

What Is a Realm?

For a web application, a *realm* is a complete database of *users* and *groups* that identify valid users of a web application (or a set of web applications) and are controlled by the same authentication policy.

The Java EE server authentication service can govern users in multiple realms. In this release of the Application Server, the `file`, `admin-realm`, and `certificate` realms come preconfigured for the Application Server.

In the `file` realm, the server stores user credentials locally in a file named `keyfile`. You can use the Admin Console to manage users in the `file` realm.

When using the `file` realm, the server authentication service verifies user identity by checking the `file` realm. This realm is used for the authentication of all clients except for web browser clients that use the HTTPS protocol and certificates.

In the `certificate` realm, the server stores user credentials in a certificate database. When using the `certificate` realm, the server uses certificates with the HTTPS protocol to authenticate web clients. To verify the identity of a user in the `certificate` realm, the authentication service verifies an X.509 certificate. For step-by-step instructions for creating this type of certificate, see "Working with Digital Certificates" on page 788. The common name field of the X.509 certificate is used as the principal name.

The `admin-realm` is also a `FileRealm` and stores administrator user credentials locally in a file named `admin-keyfile`. You can use the Admin Console to manage users in this realm in the same way you manage users in the `file` realm. For more information, see "Managing Users and Groups on the Application Server" on page 781.

What Is a User?

A *user* is an individual (or application program) identity that has been defined in the Application Server. In a web application, a user can have a set of *roles* associated with that identity, which entitles them to access all resources protected by those roles. Users can be associated with a group.

A Java EE user is similar to an operating system user. Typically, both types of users represent people. However, these two types of users are not the same. The Java EE server authentication service has no knowledge of the user name and password you provide when you log on to the operating system. The Java EE server authentication service is not connected to the security mechanism of the operating system. The two security services manage users that belong to different realms.

What Is a Group?

A *group* is a set of authenticated *users*, classified by common traits, defined in the Application Server.

A Java EE user of the `file` realm can belong to an Application Server group. (A user in the `certificate` realm cannot.) An Application Server *group* is a category of users classified by common traits, such as job title or customer profile. For example, most customers of an e-commerce application might belong to the `CUSTOMER` group, but the big spenders would belong to the `PREFERRED` group. Categorizing users into groups makes it easier to control the access of large numbers of users.

An Application Server *group* has a different scope from a *role*. An Application Server group is designated for the entire Application Server, whereas a role is associated only with a specific application in the Application Server.

What Is a Role?

A *role* is an abstract name for the permission to access a particular set of resources in an application. A *role* can be compared to a key that can open a lock. Many people might have a copy of the key. The lock doesn't care who you are, only that you have the right key.

Overview of Web Application Security

In the Java EE platform, *web components* provide the dynamic extension capabilities for a web server. Web components are either Java servlets, JSP pages, JSF pages, or web service endpoints. The interaction between a web client and a web application is illustrated in Figure 30–1.

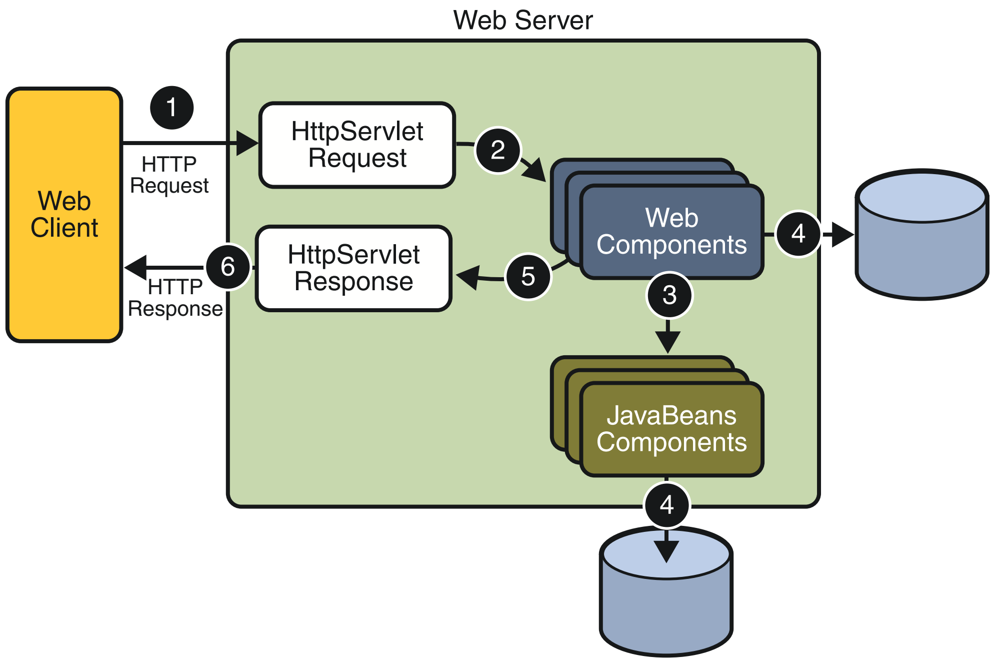

FIGURE 30–1 Java Web Application Request Handling

Web components are supported by the services of a runtime platform called a *web container*. A web container provides services such as request dispatching, security, concurrency, and life-cycle management.

Certain aspects of web application security can be configured when the application is installed, or *deployed*, to the web container. Annotations and/or deployment descriptors are used to relay information to the deployer about security and other aspects of the application. Specifying this information in annotations or in the deployment descriptor helps the deployer set up the appropriate security policy for the web application. Any values explicitly specified in the deployment descriptor override any values specified in annotations. This chapter provides more information on configuring security for web applications.

For secure transport, most web applications use the HTTPS protocol. For more information on using the HTTPS protocol, read "Establishing a Secure Connection Using SSL" on page 785.

Working with Security Roles

If you read "Working with Realms, Users, Groups, and Roles" on page 777, you will remember the following definitions:

- In applications, roles are defined using annotations or in application deployment descriptors such as `web.xml`, `ejb-jar.xml`, and `application.xml`.

  A *role* is an abstract name for the permission to access a particular set of resources in an application. For more information, read "What Is a Role?" on page 780.

  For more information on defining roles, see "Declaring Security Roles" on page 841.
- On the Application Server, the following options are configured using the Admin Console:
  - A *realm* is a complete database of *users* and *groups* that identify valid users of a web application (or a set of web applications) and are controlled by the same authentication policy. For more information, read "What Is a Realm?" on page 779.
  - A *user* is an individual (or application program) identity that has been defined in the Application Server. On the Application Server, a user generally has a user name, a password, and, optionally, a list of *groups* to which this user has been assigned. For more information, read "What Is a User?" on page 780.
  - A *group* is a set of authenticated *users*, classified by common traits, defined in the Application Server. For more information, read "What Is a Group?" on page 780.
  - A *principal* is an entity that can be authenticated by an authentication protocol in a security service that is deployed in an enterprise.

    For more information on configuring users on the Application Server, read "Managing Users and Groups on the Application Server" on page 781.
- During deployment, the deployer takes the information provided in the application deployment descriptor and maps the roles specified for the application to users and groups defined on the server using the Application Server deployment descriptors `sun-web.xml`, `sun-ejb-jar.xml`, or `sun-application.xml`.

  For more information, read "Mapping Security Roles to Application Server Groups" on page 844.

Declaring Security Roles

You can declare security role names used in web applications using either the `@DeclareRoles` annotation (preferred) or the `security-role-ref` elements of the deployment descriptor. Declaring security role names in this way enables you to link the security role names used in the code to the security roles defined for an assembled application. In the absence of this linking step, any security role name used in the code will be assumed to correspond to a security role of the same name in the assembled application.

```
        myCart.getTotal();
        //....
    }
}
//....
}
```

The @RunAs annotation is equivalent to the run-as element in the deployment descriptor.

Declaring Security Requirements in a Deployment Descriptor

Web applications are created by application developers who give, sell, or otherwise transfer the application to an application deployer for installation into a runtime environment. Application developers communicate how the security is to be set up for the deployed application *declaratively* by use of the *deployment descriptor* mechanism. A deployment descriptor enables an application's security structure, including roles, access control, and authentication requirements, to be expressed in a form external to the application.

A web application is defined using a standard Java EE web.xml deployment descriptor. A deployment descriptor is an XML schema document that conveys elements and configuration information for web applications. The deployment descriptor must indicate which version of the web application schema (2.4 or 2.5) it is using, and the elements specified within the deployment descriptor must comply with the rules for processing that version of the deployment descriptor. Version 2.5 of the Java Servlet Specification, which can be downloaded at *SRV.13, Deployment Descriptor* (http://jcp.org/en/jsr/detail?id=154), contains more information regarding the structure of deployment descriptors.

The following code is an example of the elements in a deployment descriptor that apply specifically to declaring security for web applications or for resources within web applications. This example comes from section SRV.13.5.2, *An Example of Security*, from the Java Servlet Specification 2.5.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
        version="2.5">
    <display-name>A Secure Application</display-name>

    <!-- SERVLET -->
    <servlet>
        <servlet-name>catalog</servlet-name>
        <servlet-class>com.mycorp.CatalogServlet</servlet-class>
        <init-param>
            <param-name>catalog</param-name>
            <param-value>Spring</param-value>
        </init-param>
        <security-role-ref>
            <role-name>MGR</role-name>
            <!-- role name used in code -->
            <role-link>manager</role-link>
        </security-role-ref>
    </servlet>

    <!-- SECURITY ROLE -->
    <security-role>
        <role-name>manager</role-name>
    </security-role>

    <servlet-mapping>
        <servlet-name>catalog</servlet-name>
        <url-pattern>/catalog/*</url-pattern>
    </servlet-mapping>

    <!-- SECURITY CONSTRAINT -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>CartInfo</web-resource-name>
            <url-pattern>/catalog/cart/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>manager</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>

    <!-- LOGIN CONFIGURATION-->
    <login-config>
        <auth-method>BASIC</auth-method>
    </login-config>
</web-app>
```

As shown in the preceding example, the `<web-app>` element is the root element for web applications. The `<web-app>` element contains the following elements that are used for specifying security for a web application:

- `<security-role-ref>`

  The *security role reference* element contains the declaration of a security role reference in the web application's code. The declaration consists of an optional description, the security role name used in the code, and an optional link to a security role.

  The security *role name* specified here is the security role name used in the code. The value of the `role-name` element must be the `String` used as the parameter to the `HttpServletRequest.isUserInRole(String role)` method. The container uses the mapping of `security-role-ref` to `security-role` when determining the return value of the call.

  The security *role link* specified here contains the value of the name of the security role that the user may be mapped into. The `role-link` element is used to link a security role reference to a defined security role. The `role-link` element must contain the name of one of the security roles defined in the `security-role` elements.

  For more information about security roles, read "Working with Security Roles" on page 841.

- `<security-role>`

  A *security role* is an abstract name for the permission to access a particular set of resources in an application. A security role can be compared to a key that can open a lock. Many people might have a copy of the key. The lock doesn't care who you are, only that you have the right key.

  The `security-role` element is used with the `security-role-ref` element to map roles defined in code to roles defined for the web application. For more information about security roles, read "Working with Security Roles" on page 841.

- `<security-constraint>`

  A *security constraint* is used to define the access privileges to a collection of resources using their URL mapping. Read "Specifying Security Constraints" on page 854 for more detail on this element. The following elements can be part of a security constraint:

  - `<web-resource-collection>` element: *Web resource collections* describe a URL pattern and HTTP method pair that identify resources that need to be protected.
  - `<auth-constraint>` element: *Authorization constraints* indicate which users in specified roles are permitted access to this resource collection. The role name specified here must either correspond to the role name of one of the `<security-role>` elements defined for this web application, or be the specially reserved role name `*`, which is a compact syntax for indicating all roles in the web application. Role names are case sensitive. The roles defined for the application must be mapped to users and groups defined on the server. For more information about security roles, read "Working with Security Roles" on page 841.

  - `<user-data-constraint>` element: *User data constraints* specify network security requirements, in particular, this constraint specifies how data communicated between the client and the container should be protected. If a user transport guarantee of INTEGRAL or CONFIDENTIAL is declared, all user name and password information will be sent over a secure connection using HTTP over SSL (HTTPS). Network security requirements are discussed in "Specifying a Secure Connection" on page 857.
- `<login-config>`

  The *login configuration* element is used to specify the user authentication method to be used for access to web content, the realm in which the user will be authenticated, and, in the case of form-based login, additional attributes. When specified, the user must be authenticated before access to any resource that is constrained by a security constraint will be granted. The types of user authentication methods that are supported include basic, form-based, digest, and client certificate. Read "Specifying an Authentication Mechanism" on page 858 for more detail on this element.

Some of the elements of web application security must be addressed in server configuration files rather than in the deployment descriptor for the web application. Configuring security on the Application Server is discussed in the following sections and books:

- "Securing the Application Server" on page 777
- "Managing Users and Groups on the Application Server" on page 781
- "Installing and Configuring SSL Support" on page 785
- "Deploying Secure Enterprise Beans" on page 819
- *Sun Java System Application Server 9.1 Administration Guide*
- *Sun Java System Application Server 9.1 Developer's Guide*

The following sections provide more information on deployment descriptor security elements:

- "Specifying Security Constraints" on page 854
- "Working with Security Roles" on page 841
- "Specifying a Secure Connection" on page 857
- "Specifying an Authentication Mechanism" on page 858

Specifying Security Constraints

Security constraints are a declarative way to define the protection of web content. A security constraint is used to define access privileges to a collection of resources using their URL mapping. Security constraints are defined in a deployment descriptor. The following example shows a typical security constraint, including all of the elements of which it consists:

```
<security-constraint>
    <display-name>ExampleSecurityConstraint</display-name>
    <web-resource-collection>
        <web-resource-name>
            ExampleWRCollection
        </web-resource-name>
```

```
            <url-pattern>/example</url-pattern>
            <http-method>POST</http-method>
            <http-method>GET</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>exampleRole</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
```

As shown in the example, a security constraint (`<security-constraint>` in deployment descriptor) consists of the following elements:

- *Web resource collection* (`web-resource-collection`)

  A web resource collection is a list of URL patterns (the part of a URL *after* the host name and port which you want to constrain) and HTTP operations (the methods within the files that match the URL pattern which you want to constrain (for example, `POST`, `GET`)) that describe a set of resources to be protected.
- *Authorization constraint* (`auth-constraint`)

  An authorization constraint establishes a requirement for authentication and names the roles authorized to access the URL patterns and HTTP methods declared by this security constraint. If there is no authorization constraint, the container must accept the request without requiring user authentication. If there is an authorization constraint, but no roles are specified within it, the container will not allow access to constrained requests under any circumstances. The wildcard character * can be used to specify all role names defined in the deployment descriptor. Security roles are discussed in "Working with Security Roles" on page 841.
- *User data constraint* (`user-data-constraint`)

  A user data constraint establishes a requirement that the constrained requests be received over a protected transport layer connection. This guarantees how the data will be transported between client and server. The choices for type of transport guarantee include `NONE`, `INTEGRAL`, and `CONFIDENTIAL`. If no user data constraint applies to a request, the container must accept the request when received over any connection, including an unprotected one. These options are discussed in "Specifying a Secure Connection" on page 857.

Security constraints work only on the original request URI and not on calls made throug a `RequestDispatcher` (which include `<jsp:include>` and `<jsp:forward>`). Inside the application, it is assumed that the application itself has complete access to all resources and would not forward a user request unless it had decided that the requesting user also had access.

Many applications feature unprotected web content, which any caller can access without authentication. In the web tier, you provide unrestricted access simply by not configuring a security constraint for that particular request URI. It is common to have some unprotected resources and some protected resources. In this case, you will define security constraints and a login method, but they will not be used to control access to the unprotected resources. Users won't be asked to log in until the first time they enter a protected request URI.

The Java Servlet specification defines the request URI as the part of a URL *after* the host name and port. For example, let's say you have an e-commerce site with a browsable catalog that you would want anyone to be able to access, and a shopping cart area for customers only. You could set up the paths for your web application so that the pattern `/cart/*` is protected but nothing else is protected. Assuming that the application is installed at context path `/myapp`, the following are true:

- `http://localhost:8080/myapp/index.jsp` is *not* protected.
- `http://localhost:8080/myapp/cart/index.jsp` *is* protected.

A user will not be prompted to log in until the first time that user accesses a resource in the `cart/` subdirectory.

Specifying Separate Security Constraints for Different Resources

You can create a separate security constraint for different resources within your application. For example, you could allow users with the role of `PARTNER` access to the `POST` method of all resources with the URL pattern `/acme/wholesale/*`, and allow users with the role of `CLIENT` access to the `POST` method of all resources with the URL pattern `/acme/retail/*`. An example of a deployment descriptor that would demonstrate this functionality is the following:

```
// SECURITY CONSTRAINT #1
<security-constraint>
    <web-resource-collection>
        <web-resource-name>wholesale</web-resource-name>
        <url-pattern>/acme/wholesale/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>PARTNER</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>

// SECURITY CONSTRAINT #2
<security-constraint>
    <web-resource-collection>
        <web-resource-name>retail</web-resource-name>
        <url-pattern>/acme/retail/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>CLIENT</role-name>
    </auth-constraint>
</security-constraint>
```

When the same `url-pattern` and `http-method` occur in multiple security constraints, the constraints on the pattern and method are defined by combining the individual constraints, which could result in unintentional denial of access. Section 12.7.2 of the *Java Servlet 2.5 Specification* (downloadable from `http://jcp.org/en/jsr/detail?id=154`) gives an example that illustrates the combination of constraints and how the declarations will be interpreted.

Specifying a Secure Connection

A user data constraint (`<user-data-constraint>` in the deployment descriptor) requires that all constrained URL patterns and HTTP methods specified in the security constraint are received over a protected transport layer connection such as HTTPS (HTTP over SSL). A user data constraint specifies a transport guarantee (`<transport-guarantee>` in the deployment descriptor). The choices for transport guarantee include `CONFIDENTIAL`, `INTEGRAL`, or `NONE`. If you specify `CONFIDENTIAL` or `INTEGRAL` as a security constraint, that type of security constraint applies to all requests that match the URL patterns in the web resource collection and not just to the login dialog box. The following security constraint includes a transport guarantee:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>wholesale</web-resource-name>
        <url-pattern>/acme/wholesale/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>PARTNER</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

The strength of the required protection is defined by the value of the transport guarantee. Specify `CONFIDENTIAL` when the application requires that data be transmitted so as to prevent other entities from observing the contents of the transmission. Specify `INTEGRAL` when the application requires that the data be sent between client and server in such a way that it cannot be changed in transit. Specify `NONE` to indicate that the container must accept the constrained requests on any connection, including an unprotected one.

The user data constraint is handy to use in conjunction with basic and form-based user authentication. When the login authentication method is set to `BASIC` or `FORM`, passwords are not protected, meaning that passwords sent between a client and a server on an unprotected session can be viewed and intercepted by third parties. Using a user data constraint with the user authentication mechanism can alleviate this concern. Configuring a user authentication mechanism is described in "Specifying an Authentication Mechanism" on page 858.

To guarantee that data is transported over a secure connection, ensure that SSL support is configured for your server. If your server is the Sun Java System Application Server, SSL support is already configured. If you are using another server, consult the documentation for that server for information on setting up SSL support. More information on configuring SSL support on the Application Server can be found in "Establishing a Secure Connection Using SSL" on page 785 and in the *Sun Java System Application Server 9.1 Administration Guide*.

Note – Good Security Practice: If you are using sessions, after you switch to SSL you should never accept any further requests for that session that are non-SSL. For example, a shopping site might not use SSL until the checkout page, and then it might switch to using SSL to accept your card number. After switching to SSL, you should stop listening to non-SSL requests for this session. The reason for this practice is that the session ID itself was not encrypted on the earlier communications. This is not so bad when you're only doing your shopping, but after the credit card information is stored in the session, you don't want a bad guy trying to fake the purchase transaction against your credit card. This practice could be easily implemented using a filter.

Specifying an Authentication Mechanism

To specify an authentication mechanism for your web application, declare a `login-config` element in the application deployment descriptor. The `login-config` element is used to configure the authentication method and realm name that should be used for this application, and the attributes that are needed by the form login mechanism when form-based login is selected. The sub-element `auth-method` configures the authentication mechanism for the web application. The element content must be either BASIC, DIGEST, FORM, CLIENT-CERT, or a vendor-specific authentication scheme. The `realm-name` element indicates the realm name to use for the authentication scheme chosen for the web application. The `form-login-config` element specifies the login and error pages that should be used when FORM based login is specified.

The authentication mechanism you choose specifies how the user is prompted to login. If the `<login-config>` element is present, and the `<auth-method>` element contains a value other than `NONE`, the user must be authenticated before it can access any resource that is constrained by the use of a `security-constraint` element in the same deployment descriptor (read "Specifying Security Constraints" on page 854 for more information on security constraints). If you do not specify an authentication mechanism, the user will not be authenticated.

When you try to access a web resource that is constrained by a `security-constraint` element, the web container activates the authentication mechanism that has been configured for that resource. To specify an authentication method, place the `<auth-method>` element between `<login-config>` elements in the deployment descriptor, like this:

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

An example of a deployment descriptor that constrains all web resources for this application (in *italics* below) and requires HTTP basic authentication when you try to access that resource (in **bold** below) is shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
             http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <display-name>basicauth</display-name>
    <servlet>
        <display-name>index</display-name>
        <servlet-name>index</servlet-name>
        <jsp-file>/index.jsp</jsp-file>
    </servlet>
    <security-role>
        <role-name>loginUser</role-name>
    </security-role>
    <security-constraint>
        <display-name>SecurityConstraint1</display-name>
        <web-resource-collection>
            <web-resource-name>WRCollection</web-resource-name>
            <url-pattern>/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>loginUser</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
    </login-config>
</web-app>
```

Before you can authenticate a user, you must have a database of user names, passwords, and roles configured on your web or application server. For information on setting up the user database, refer to "Managing Users and Groups on the Application Server" on page 781 and the *Sun Java System Application Server 9.1 Administration Guide*.

The authentication mechanisms are discussed further in the following sections:

- "HTTP Basic Authentication" on page 860
- "Form-Based Authentication" on page 861
- "HTTPS Client Authentication" on page 863
- "Digest Authentication" on page 866

HTTP Basic Authentication

HTTP Basic Authentication requires that the server request a user name and password from the web client and verify that the user name and password are valid by comparing them against a database of authorized users. When basic authentication is declared, the following actions occur:

1. A client requests access to a protected resource.
2. The web server returns a dialog box that requests the user name and password.
3. The client submits the user name and password to the server.
4. The server authenticates the user in the specified realm and, if successful, returns the requested resource.

Figure 30–2 shows what happens when you specify HTTP basic authentication.

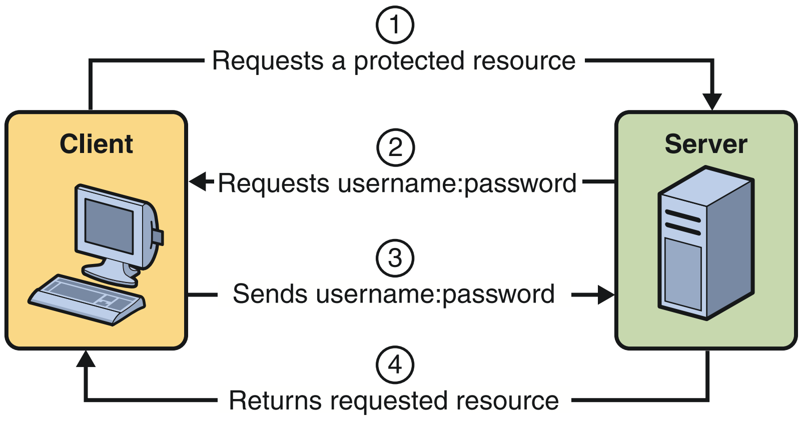
1
Requests a protected resource
Client
Server
2
Requests username:password
3
Sends username:password
4
Returns requested resource

FIGURE 30–2 HTTP Basic Authentication

The following example shows how to specify basic authentication in your deployment descriptor:

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

HTTP basic authentication is not a secure authentication mechanism. Basic authentication sends user names and passwords over the Internet as text that is Base64 encoded, and the target server is not authenticated. This form of authentication can expose user names and passwords. If someone can intercept the transmission, the user name and password information can easily be decoded. However, when a secure transport mechanism, such as SSL, or security at the network level, such as the IPSEC protocol or VPN strategies, is used in conjunction with basic authentication, some of these concerns can be alleviated.

"Example: Basic Authentication with JAX-WS" on page 885 is an example application that uses HTTP basic authentication in a JAX-WS service. "Example: Using Form-Based Authentication with a JSP Page" on page 868 can be easily modified to demonstrate basic authentication. To do so, replace the text between the `<login-config>` elements with those shown in this section.

Form-Based Authentication

Form-based authentication allows the developer to control the look and feel of the login authentication screens by customizing the login screen and error pages that an HTTP browser presents to the end user. When form-based authentication is declared, the following actions occur:

1. A client requests access to a protected resource.
2. If the client is unauthenticated, the server redirects the client to a login page.
3. The client submits the login form to the server.
4. The server attempts to authenticate the user.
    a. If authentication succeeds, the authenticated user's principal is checked to ensure it is in a role that is authorized to access the resource. If the user is authorized, the server redirects the client to the resource using the stored URL path.
    b. If authentication fails, the client is forwarded or redirected to an error page.

Figure 30–3 shows what happens when you specify form-based authentication.

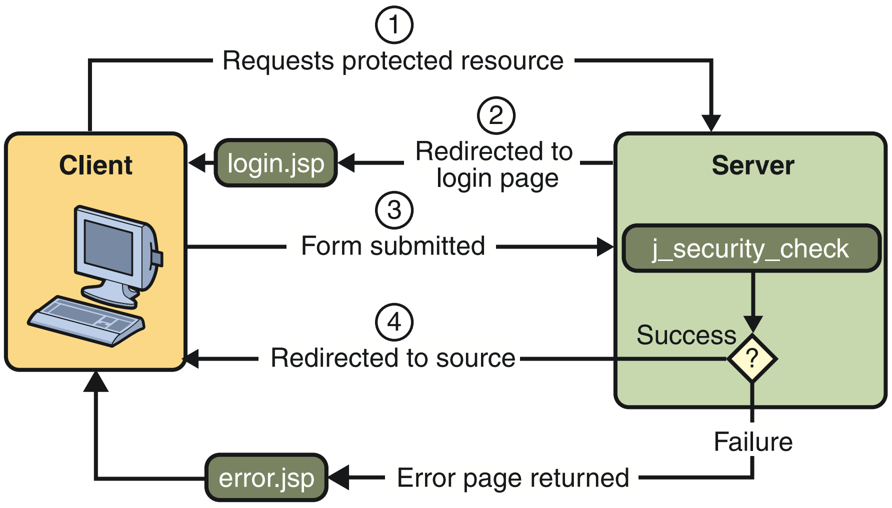

FIGURE 30–3 Form-Based Authentication

The following example shows how to declare form-based authentication in your deployment descriptor:

```
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>file</realm-name>
    <form-login-config>
        <form-login-page>/logon.jsp</form-login-page>
        <form-error-page>/logonError.jsp</form-error-page>
    </form-login-config>
</login-config>
```

The login and error page locations are specified relative to the location of the deployment descriptor. Examples of login and error pages are shown in "Creating the Login Form and the Error Page" on page 869.

Form-based authentication is not particularly secure. In form-based authentication, the content of the user dialog box is sent as plain text, and the target server is not authenticated. This form of authentication can expose your user names and passwords unless all connections are over SSL. If someone can intercept the transmission, the user name and password information can easily be decoded. However, when a secure transport mechanism, such as SSL, or security at the network level, such as the IPSEC protocol or VPN strategies, is used in conjunction with form-based authentication, some of these concerns can be alleviated.

The section "Example: Using Form-Based Authentication with a JSP Page" on page 868 is an example application that uses form-based authentication.

Using Login Forms

When creating a form-based login, be sure to maintain sessions using cookies or SSL session information.

As shown in "Form-Based Authentication" on page 861, for authentication to proceed appropriately, the action of the login form must always be `j_security_check`. This restriction is made so that the login form will work no matter which resource it is for, and to avoid requiring the server to specify the action field of the outbound form. The following code snippet shows how the form should be coded into the HTML page:

```
<form method="POST" action="j_security_check">
<input type="text" name="j_username">
<input type="password" name="j_password">
</form>
```

HTTPS Client Authentication

HTTPS Client Authentication requires the client to possess a Public Key Certificate (PKC). If you specify *client authentication*, the web server will authenticate the client using the client's public key certificate.

HTTPS Client Authentication is a more secure method of authentication than either basic or form-based authentication. It uses HTTP over SSL (HTTPS), in which the server authenticates the client using the client's Public Key Certificate (PKC). Secure Sockets Layer (SSL) technology provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection. You can think of a public key certificate as the digital equivalent of a passport. It is issued by a trusted organization, which is called a certificate authority (CA), and provides identification for the bearer.

Before using HTTP Client Authentication, you must make sure that the following actions have been completed:

- Make sure that SSL support is configured for your server. If your server is the Sun Java System Application Server 9.1, SSL support is already configured. If you are using another server, consult the documentation for that server for information on setting up SSL support. More information on configuring SSL support on the application server can be found in "Establishing a Secure Connection Using SSL" on page 785 and the *Sun Java System Application Server 9.1 Administration Guide*.
- Make sure the client has a valid Public Key Certificate. For more information on creating and using public key certificates, read "Working with Digital Certificates" on page 788.

The following example shows how to declare HTTPS client authentication in your deployment descriptor:

```
<login-config>
    <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

Mutual Authentication

With *mutual authentication*, the server and the client authenticate one another. There are two types of mutual authentication:

- Certificate-based mutual authentication (see Figure 30–4)
- User name- and password-based mutual authentication (see Figure 30–5)

When using certificate-based mutual authentication, the following actions occur:

1. A client requests access to a protected resource.
2. The web server presents its certificate to the client.
3. The client verifies the server's certificate.
4. If successful, the client sends its certificate to the server.
5. The server verifies the client's credentials.
6. If successful, the server grants access to the protected resource requested by the client.

Figure 30–4 shows what occurs during certificate-based mutual authentication.

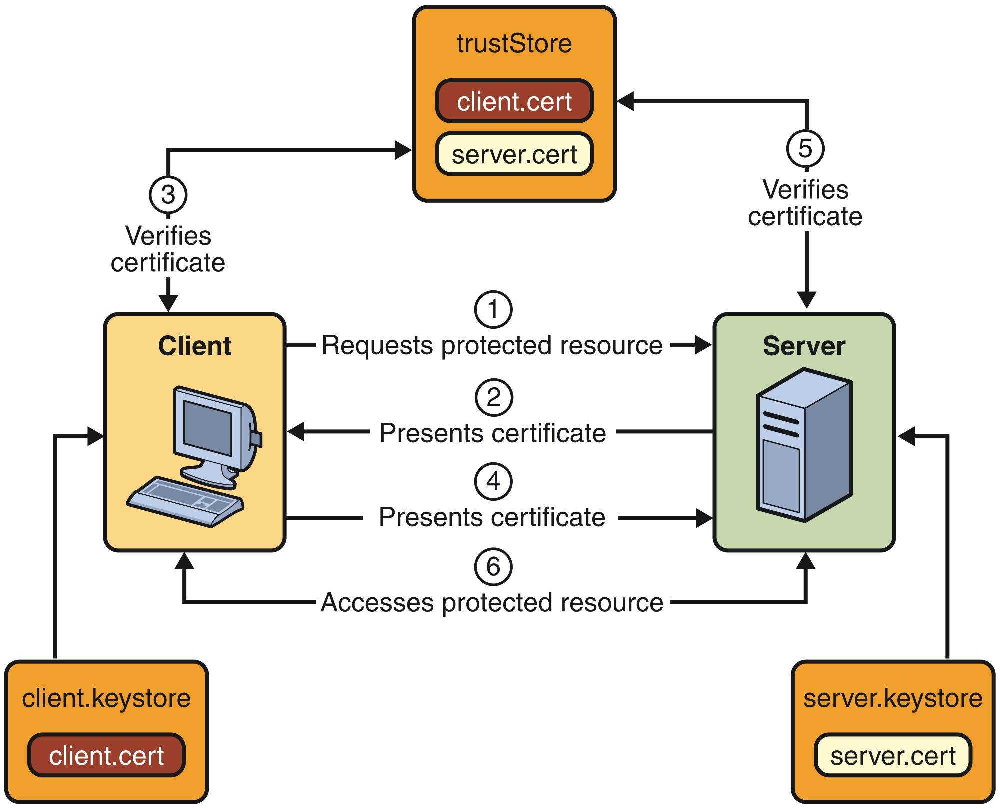

FIGURE 30–4 Certificate-Based Mutual Authentication

In user name- and password-based mutual authentication, the following actions occur:

1. A client requests access to a protected resource.
2. The web server presents its certificate to the client.
3. The client verifies the server's certificate.
4. If successful, the client sends its user name and password to the server, which verifies the client's credentials.
5. If the verification is successful, the server grants access to the protected resource requested by the client.

Figure 30–5 shows what occurs during user name- and password-based mutual authentication.

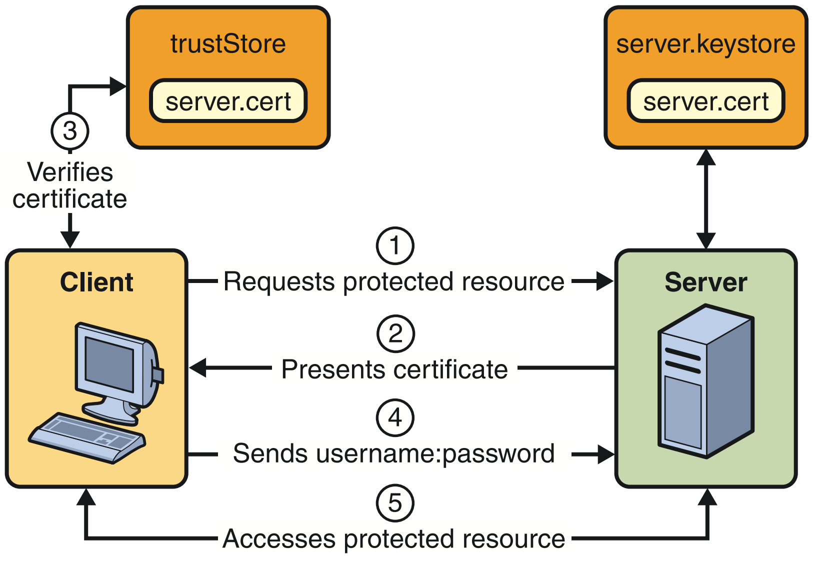

FIGURE 30–5 User Name- and Password-Based Mutual Authentication

Digest Authentication

Like HTTP basic authentication, *HTTP Digest Authentication* authenticates a user based on a user name and a password. However, the authentication is performed by transmitting the password in an encrypted form which is much more secure than the simple Base64 encoding used by basic authentication. Digest authentication is not currently in widespread use, and is not implemented in the Application Server, therefore, there is no further discussion of it in this document.