CHAPTER 3

Enterprise Java Security Fundamentals

THE J2EE platform has achieved remarkable success in meeting enterprise needs, resulting in its widespread adoption. The security infrastructure plays a key role in the e-business strategy of a company. J2EE provides a standard approach to allow enterprise applications to be developed without hard-coded security policies. Instead, declarative policies are bundled with an assembled set of application components. Security policies specified using this security model are enforced in any operational environments and deployed in any of the application servers that host them.

The J2EE security model addresses authentication, authorization, delegation, and data integrity for the components that make up a J2EE environment. This environment includes J2EE applications—Web components, such as servlets and JSP files, EJB components, Java 2 connectors, and JavaMail—and secure interoperability requirements. The J2EE security model also considers the organizational roles that define and enforce these security policies:

- Application Component Provider
- Application Assembler
- Deployer
- System Administrator
- J2EE Product Provider

This chapter provides an overview of J2EE, exploring the J2EE security model. The chapter explains how various J2EE components are tied into enterprise security, describes how the J2EE security model addresses the security of J2EE components, and identifies the responsibility of each of the organizational roles in enforcing security. Declarative security policies and programmatic

security APIs are explained, in addition to the security requirements on JavaMail, Java connectors, client applications, and containers. This chapter also outlines the secure interoperability requirements that exist between various application servers.

Since its inception, one of the top requirements of the J2EE security model has been to support secure application deployments that do not rely on private networks or other application runtime isolation techniques. This allows application portability between containers. Another requirement has been to reduce the application developer's burden by delegating the security responsibilities to the J2EE roles. Finally, the policy-driven security model enables much of security enforcement to be handled without custom code.

3.1 Enterprise Systems

An *enterprise Java environment,* or WAS environment, is nominally viewed as a three-tier architecture (see Section 2.1.2 on page 25). Clients access the information made available through middle-tier systems, which connect to the back-end enterprise systems, as shown in Figure 3.1.

In an enterprise Java environment, the clients can be both Java based and non-Java based. Clients access the servers over a variety of protocols, including HTTP,

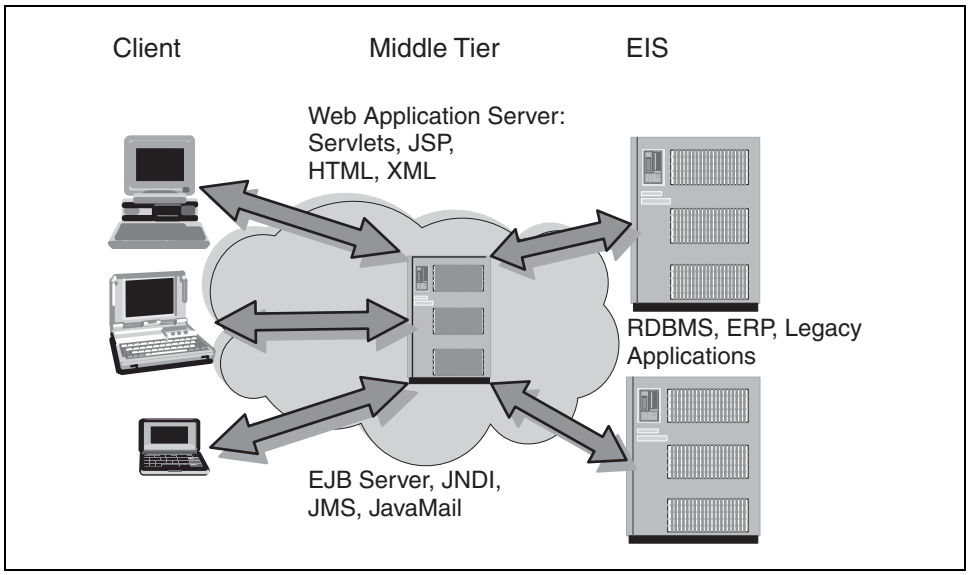

Figure 3.1. WAS Environment

IIOP, SSL, and other messaging protocols accessible through JMS. These clients connect to and access a J2EE-based server environment providing a hosting system for the enterprise components. These components constitute a presentation layer in the form of servlets, JSP files, HTML files, or XML documents. Alternatively, the components can abstract out the business logic in the form of enterprise beans. Clients may also submit their requests by using e-mail protocols through the JavaMail framework or connect to naming and directory services by using the Java Naming and Directory Interface (JNDI). In an enterprise environment, middle-tier applications are likely to connect to back-end enterprise information systems (EISs). Examples of back-end EISs include relational database management systems (RDBMSs) and ERP applications.

Before delving into the security implications of this architecturally rich environment, it is important to understand the technologies that comprise a J2EE environment.

3.2 J2EE Applications

A *J2EE application*, an enterprise application that conforms to the J2EE specification, is structured as shown in Figure 3.2 and consists of the following:

- Zero or more EJB modules
- Zero or more Web modules
- Zero or more application client modules

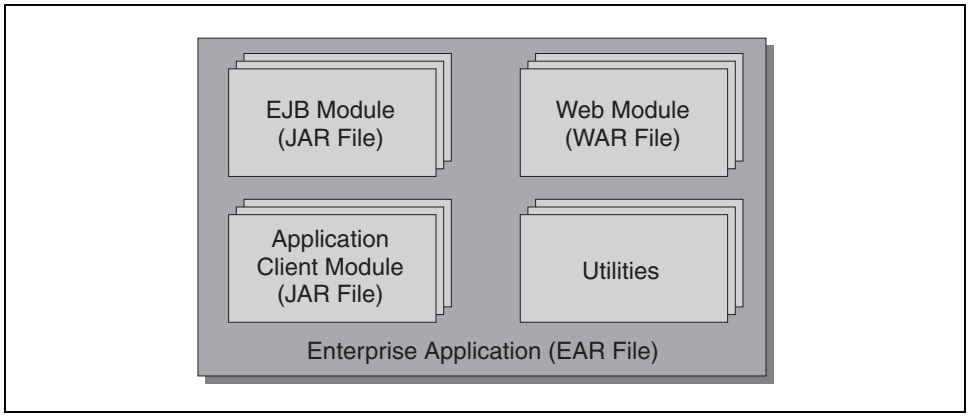

Figure 3.2. Contents of a J2EE Application

- Optionally, JAR files containing dependent classes or components required by the application
- Any combination of the preceding, as long as it contains at least one module

A J2EE application is represented by, and packaged in, an Enterprise Archive (EAR) file. The modules that comprise the EAR file are themselves packaged in archive files specific to their types. For example, a Web module is packaged in a Web Archive (WAR) file, and an EJB module, containing one or more enterprise beans, is packaged in a JAR file. WAR files can exist as independent deployment units from EAR files.

EAR files also contain a *deployment descriptor* file—an XML document describing the contents of the application and containing instructions for the deployment of the application. In particular, the deployment descriptor specifies the security settings to be enforced by the runtime environment. Each WAR file packaging a Web module, JAR file packaging enterprise beans, or JAR file packaging an application client module contains its own deployment descriptor as well.

3.2.1 EJB Modules

An *enterprise bean* is a Java component that can be combined with other resources to create distributed client/server applications. Instantiated enterprise beans reside in *enterprise bean containers*, or *EJB containers*. An EJB container provides an interface between the enterprise beans and the application server on which the enterprise beans reside. An enterprise bean is typically accessed using Java RMI-IIOP. An ORB manages the interaction between clients and enterprise beans, using IIOP. ORBs enable clients to make requests and receive responses from servers in distributed computing environments. Alternatively, enterprise beans are accessible through JMS. It is also possible to invoke an enterprise bean as a Web service via SOAP, as explained in Chapter 14 on page 497.

There are three types of enterprise beans: entity beans, session beans, and message-driven beans. *Entity beans* store persistent data and typically use database connections. Entity beans are of two types: CMP entity beans and BMP entity beans.

- Entity beans with *container-managed persistence* (CMP) let the EJB container transparently and implicitly manage the persistent state. The enterprise bean developer does not need to code any database access functions within the enterprise bean class methods.
- Entity beans with *bean-managed persistence* (BMP) manage persistent data in a manner defined by the application developer in the bean code. This usually includes writing to databases.

Session beans do not require database access, although they can obtain it indirectly, as needed, by accessing entity beans. Session beans can also obtain direct access to databases and other resources through the use of *resource references*, which include the use of JDBC. Session beans can be either stateless or stateful.

- A session bean is said to be *stateless* if it provides a stateless service to the client. A business method on a stateless session bean is similar to a procedural application or static method; there is no instance state. Therefore, all the data needed to execute a stateless session bean's method is provided by the method arguments.
- A session bean is said to be *stateful* if it acts as a server-side extension of the client that uses it. A stateful session bean is created by a client and will work for only that client until the client connection is dropped or the bean is explicitly removed. Unlike a stateless session bean, a stateful session bean has state or instance fields that can be initialized and changed by the client with each method invocation.

Message-driven beans are enterprise beans accessible asynchronously via JMS rather than synchronously through such protocols as RMI-IIOP. The EJB V2.1 specification expands the scope of message-driven beans beyond JMS to support any messaging system.

An *EJB module* is one or more enterprise beans assembled into a single deployable unit. As we have observed, an EJB module is stored in a standard JAR file, commonly referred to as *ejb-jar*. This file contains

- One or more deployable enterprise beans
- A deployment descriptor, stored in an XML file

Specifically, an EJB module's deployment descriptor file declares the contents of the module, specifies the structure and external dependencies of the enterprise beans in the module, explains how the enterprise beans are to be used at runtime, and defines the security policies applicable to the enterprise beans within the module. The format of the security policy is defined by the EJB specification (see Chapter 5 on page 157).

3.2.2 Web Modules

A *Web module* represents a *Web application*—an application that can be accessed over the Web using HTTP. A Web module is used to assemble servlets and JSP files, as well as static content, such as HTML pages, into a single deployable unit.

As we said earlier, Web modules are stored in WAR files, which are enhanced JAR files with a `.war` file extension, and contain

- One or more servlets, JSP files, and other supporting files
- A deployment descriptor, stored in an XML file

The deployment descriptor file, `web.xml`, declares the contents of the Web module. This file contains information about the structure and external dependencies of the components in the Web module and describes the components' runtime use. In addition, the deployment description file is used to declare the security policies applicable to the universal resource identifiers (URIs) that are mapped to the resources within the Web module. These security policies include both the authorization policy and the login configuration information. The format of the security policy is defined by the Java Servlet specification.

Servlets are Java programs running on a WAS and extend the Web server's capabilities. For example, servlets support generation of dynamic Web page content, provide database access, concurrently serve multiple clients, and filter data by MIME type. Servlets use the Java Servlet API. By analogy, servlets are the server-side equivalent of client-side browser applets.

JSP files enable the separation of the HTML coding from the business logic in Web pages, allowing HTML programmers and Java programmers to more easily collaborate in creating and maintaining pages. This process is described in greater detail in Section 4.1.2 on page 104.

3.2.3 Application Client Modules

Application clients are first-tier Java-based client programs. Even though it is a regular Java application, an application client depends on an *application client container* to provide system services. An *application client module* packages application client code in a JAR file. This JAR file includes a deployment descriptor XML file, which specifies the enterprise beans and external resources referenced by the application.

The security configuration of an application client determines how the application will access enterprise beans and Web resources. If the J2EE components that the client application accesses are secured, the client will be authenticated accordingly. In order for an application client to retrieve authentication data from an end user, configuration information must be specified in a deployment descriptor XML file, `application-client.xml`, associated with the client application. Application clients typically run in an environment that has a Java 2 security manager installed and the security policies enforced based on the J2SE security policy framework (see Chapter 8 on page 253).

3.3 Secure Interoperability between ORBs

J2EE applications are required to use RMI-IIOP when accessing EJB components. This allows enterprise beans to be portable between container implementations.

A *J2EE container* provides the runtime support for the J2EE components. A J2EE container vendor enables access to the enterprise beans via IIOP. This facilitates interoperability between containers by using the Common Secure Interoperability (CSI) protocol. Security is enabled and enforced by the ORBs, ensuring authenticity, confidentiality, and integrity. Version 2 of this protocol specification (CSIv2) is the accepted industry standard and is mandated by the J2EE specification.

3.4 Connectors

A *resource adapter* is defined in the J2EE Connector Architecture specification as a system-level software driver that a Java application uses to connect to an EIS. The resource adapter plugs into an application server and provides connectivity between the EIS, the J2EE application server, and the enterprise application.

The Java Connector Architecture (JCA) specification allows resource adapters that support access to non-J2EE systems to be plugged into any J2EE environment. Resource adapter components implementing the JCA API are called *connectors*.

The JCA specification describes standard ways to extend J2EE services with connectors to other non-J2EE application systems, such as mainframe systems and ERP systems. The JCA architecture enables an EIS vendor to provide a standard resource adapter for a J2EE application to connect to the EIS. A resource adapter is used by a Java application to connect to an EIS. For example, Web enablement of business applications, such as IBM's Customer Information Control System (CICS), would imply that the J2EE-based presentation layer would connect to a CICS application using a CICS connector. With this approach, protocol details of connecting to a CICS system are transparent to the Web application and are handled by the CICS connector implementation.

JCA defines a standard set of *system-level contracts* between a J2EE server and a resource adapter. In particular, these standard contracts include a *security contract* and enable secure access to non-J2EE EISs. The security contract helps to reduce security threats to the information system and protects valuable information resources managed by such a system. Given that most of these EIS systems have facilities to accept some form of authentication data representing an identity connecting to the system, the JCA security contract deals with the authentication aspects of the EIS. Essentially, it is about a J2EE application *signing on* to an EIS system. This means that the J2EE application accesses a connection to the EIS system by providing authentication information. As discussed in Section 3.9.4 on

page 87 and Section 3.10.3 on page 94, two organizational roles are involved in addressing this issue: the Application Component Provider and the Deployer. Specifically, the Application Component Provider can use either of two choices related to EIS sign-on: the declarative approach or the programmatic approach.

- The *declarative approach* allows the Deployer to set up the resource principal and EIS sign-on information. For example, the Deployer sets the user ID and password—or another set of credentials—necessary to establish a connection to an EIS instance.
- With the *programmatic approach*, the Application Component Provider can choose to perform sign-on to an EIS from the component code by providing explicit security information. For example, the user ID and password—or another set of credentials—necessary to establish a connection to an EIS instance are coded into the application code.

The Application Component Provider uses a deployment descriptor element, such as `res-auth` for EJB components, to indicate the requirement for one of the two approaches. If the `res-auth` element is set to `Container`, the application server sets up and manages EIS sign-on. If the `res-auth` element is set to `Application`, the component code performs a programmatic sign-on to the EIS.

Further details of the security aspects of a JCA-based connection to an EIS from a J2EE application are discussed in Section 3.9.4 on page 87 and Section 3.10.3 on page 94.

3.5 JMS

JMS is a standard Java messaging API that provides a common mechanism for Java-language programs to access messaging systems. Java clients and middle-tier components must be capable of using messaging systems to access the J2EE components that are enabled via a messaging layer.

- Application clients, EJB components, and Web components can send or synchronously receive a JMS message. Application clients can also receive JMS messages asynchronously.
- A new kind of enterprise bean introduced in EJB V2.0, the message-driven bean, enables the asynchronous consumption of messages. A message-driven bean can be accessed by sending a method invocation request over a messaging infrastructure. A JMS provider may optionally implement concurrent processing of messages by message-driven beans.

The J2EE specification requires JMS providers to implement both the reliable point-to-point messaging model and the publish/subscribe model. The *reliable*

point-to-point messaging model allows one entity to send messages directly to another entity that understands the format of the messages and the requests. The *publish/subscribe model* is event driven; a message is published, and the message is delivered to all subscribers of the event. One example is a StockQuote application, with multiple traders wanting to get the latest stock quote. In this scenario, the traders' applications subscribe to the stock-quote messaging service. When the stock values are published, the information is made available to all the subscribers. In a Java environment, both the StockQuote server and the traders' applications can use the JMS mechanism with a JMS provider to achieve the required messaging fucntionality.

However, JMS does not specify a security contract or an API for controlling message confidentiality and integrity. Security is considered to be a JMS-provider-specific feature. It is controlled by a System Administrator rather than implemented programmatically or by the J2EE server runtime.

3.6 Simple E-Business Request Flow

It would be helpful to understand a simple e-business request flow in an enterprise Java environment. Figure 3.3 presents a simple request flow that does not involve security.

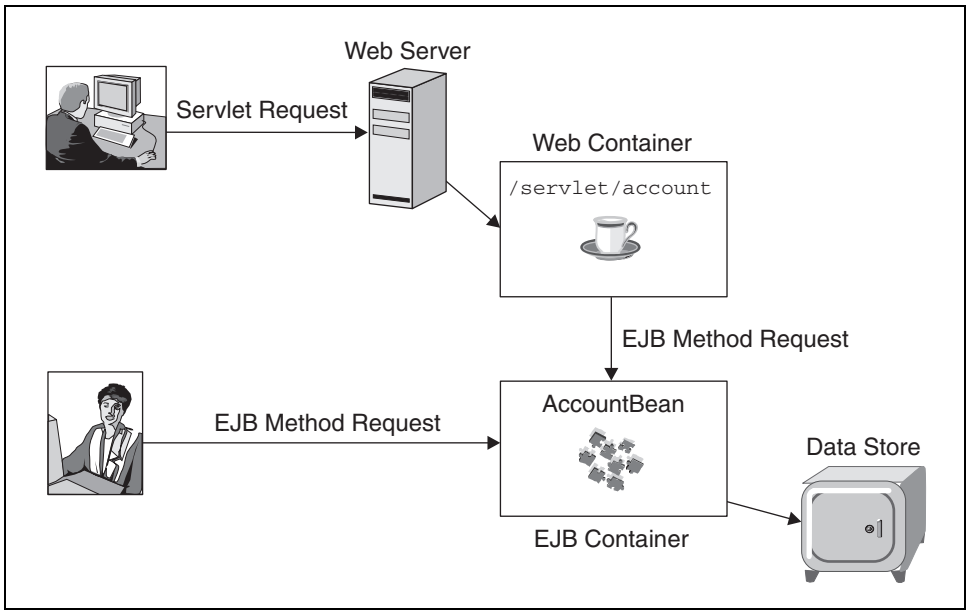

Figure 3.3. Simple E-Business Request Flow

Let us consider two types of clients: *HTTP clients*, such as Web browsers, and *IIOP clients*, regular applications capable of using IIOP to send requests and receive responses over the network. An HTTP client invokes a URL: for instance, `/servlet/account/`. The request from the user's browser gets handled by the Web server, which routes the request to a Web, or servlet, container serving the URL resource. The logic behind the URL is implemented as a Java servlet. This servlet, packaged in a Web module, is hosted in a J2EE Web container, which in turn invokes an enterprise bean, AccountBean, via IIOP. AccountBean is an entity bean, packaged in an EJB module, with its business data stored in a data store. The same enterprise bean is accessed directly from an IIOP client, packaged in an application client module. In this case, the request is not routed by the servlet but is directly accessed as a remote object from the Java client.

The request flow just described does not involve security considerations. The next sections in this chapter provide an overview of the J2EE specification as it pertains the security of an enterprise. The platform roles reflect the organizational responsibilities, from application development, application assembly, and application deployment, to administration.

3.7 J2EE Platform Roles

J2EE defines roles that reflect the responsibilities within an organization. Any person or software involved in the process of making an application available within an enterprise can usually be categorized into organization roles, called *J2EE platform roles*. The J2EE platform roles having security responsibilities are the Application Component Provider, Application Assembler, Deployer, System Administrator, J2EE Product Provider, and Tool Provider. The J2EE security model is defined with respect to these J2EE roles.

Figure 3.4 shows the interactions among the Application Component Provider, Application Assembler, Deployer, and System Administrator. These are the roles involved, from a security perspective, in the stages between development and deployment.

Figure 3.4 depicts the software process cycle from the perspective of J2EE platform roles.[1] In a typical J2EE software process cycle, application component developers build enterprise application components, such as servlets or enterprise

1. The J2EE V1.3 platform role called *Tool Provider* is responsible for supplying tools used for the development and packaging of application components. Because it depicts the software life cycle in development and deployment of a J2EE application and the users involved in the process, Figure 3.4 does not include the roles of Tool Provider and J2EE Product Provider.

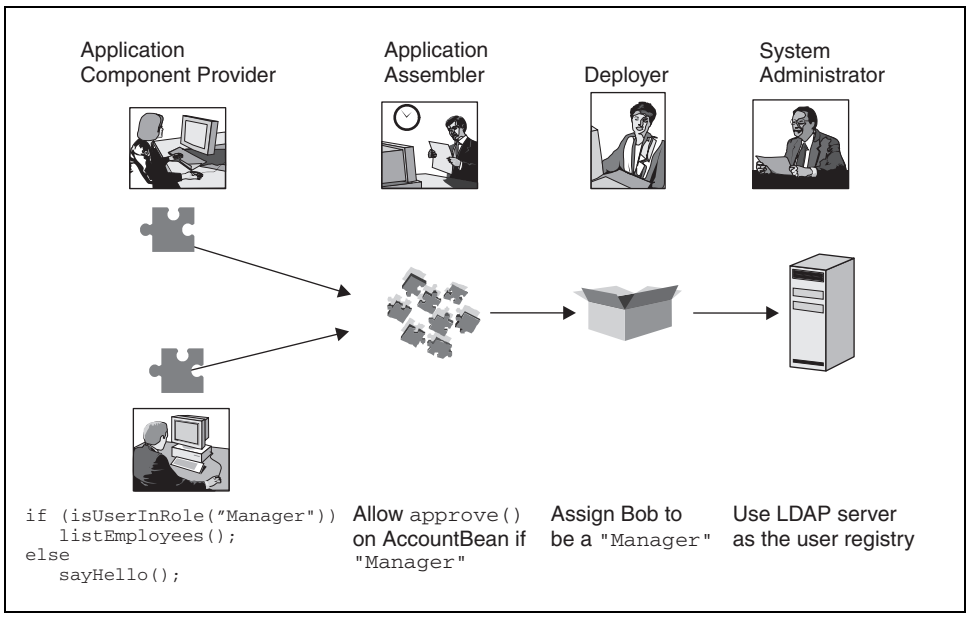

Figure 3.4. J2EE Platform Roles with Security Responsibilities in the Development and Deployment of a J2EE Application

beans. The greatest opportunity for component reuse and flexibility in reconfiguring security policy is when the components are written to be *security unaware*, meaning that they do not contain embedded security policy code. Conversely, components containing embedded security policy code are said to be *security aware*. Security-aware components are difficult to reuse, and flexibility is limited because it often requires changing the source code to reflect various security policies. For some applications, this may be unavoidable.

An Application Assembler integrates a set of components supplied by one or more Application Component Providers. The Application Assembler has the in-depth knowledge of the application. The Application Assembler specifies security policies as hints to the Deployer. For example, the Application Assembler can provide hints such that the `approve()` method of an enterprise bean should be accessed only by those principals granted the role of a Manager.

A Deployer deploys enterprise applications, assembled by Application Assemblers, into an operational environment. When tailoring the security policies to the operational environment, the Deployer consults the security policy hints provided by the Application Assembler. For example, the Deployer can assign the role of a Manager to a user named Bob.

A System Administrator is responsible for administering the system, including security. This may include configuring the J2EE product to use an LDAP server for managing security information, including user and group membership.

The following subsections provide a more detailed description of the four J2EE roles. The J2EE platform roles that we have listed are considered to be the typical roles, although in practice, the roles are adapted to better match the organization's application development and deployment work flow. The rest of this section summarizes the major responsibilities of the individual J2EE platform roles with respect to security management.

3.7.1 Application Component Provider

The *Application Component Provider* is the J2EE platform role responsible for implementing the business logic as a set of J2EE application components—enterprise beans, servlets, and/or JSP files. These components are packaged in an ejb-jar file containing one or more enterprise beans and/or aWAR file containing one or more servlets and/or JSP files, and/or a JAR file containing an application client module.

The Application Component Provider has numerous responsibilities in code development. These responsibilities range from resource access to programmatic access of the caller's security context. Following are the Application Component Provider's key security responsibilities:

3.7.1.1 Access of Resources in the Underlying Operating System

The J2EE architecture does not define the operating system principal—for example, the operating system user—under which EJB methods or servlets execute. Therefore, the Application Component Provider cannot rely on a specific principal for accessing the underlying operating system resources. The Application Component Provider should design the applications so that special privileges are not required to access system resources.

3.7.1.2 Security Recommendations

The Application Component Provider should avoid implementing security mechanisms or hard-coded security policies in the component but instead should rely on the security mechanisms provided by the J2EE container. The Application Component Provider should let the Application Assembler and the Deployer define the appropriate security policies for the application. The Application Component Provider can use the deployment descriptors to convey security-related information to the Application Assembler.

3.7.1.3 Programmatic Access to the Caller's Security Context

Programmatic-security APIs should be avoided when possible. However, they should be used when the J2EE component methods need access to security-context information because the J2EE declarative security model is insufficient to implement application security requirements.

3.7.1.4 Conveying the Use of Role References

A *security role* is a set of J2EE authorizations. The Application Component Provider may build a security-aware application and use *role references*—security role names within the application components. For example, a component may call `isUserInRole("Manager")` on a `javax.servlet.http.HttpServletRequest` object. When security role names are hard-coded in an application component, the Application Component Provider must identify these role names for the Application Assembler so that it can map component-defined security role references in each of the components in the deployment to a single application-level security role name. For example, two components in an application may use the security role references `Manager` and `Boss` within the component, whereas both of these roles imply the application security role of a Supervisor.

3.7.2 Application Assembler

The *Application Assembler* is the J2EE platform role responsible for combining J2EE components into deployable application units. The Application Assembler also simplifies the Deployer's job by providing a security view of the enterprise beans, servlets, and JSP files in the relevant deployment descriptors. A *security view* consists of a set of J2EE security roles. A security role is a semantic grouping of J2EE authorizations, or permissions—implemented as `java.security.Permission` objects—that a given type of application users must have in order to successfully use the application. The Application Assembler defines one or more security roles in the deployment descriptor and specifies and associates J2EE permissions with these roles. For example, the security role Manager could be granted the J2EE permissions to invoke an enterprise bean to grant loans and view the loan status of all the customers using Web applications. In contrast, the security role HelpDesk could be granted only a subset of these J2EE permissions—for example, only the J2EE permission to view the loan status of the customers—by having been granted access to the relevant URIs.

Following are some of the Application Assembler's security responsibilities.

3.7.2.1 Defining EJB Method Permissions

The home, local home, remote, and local interfaces of an enterprise bean are defined as part of the EJB specification.

- The *home interface* of an enterprise bean is a Java interface used to create, find, or delete an instance of the enterprise bean. The methods defined in the remote interface can be accessed from within the same container or remotely via RMI-IIOP.
- The *local home interface* of an enterprise bean is functionally similar to the home interface, but the methods defined in the local home interface are accessible only from within the same container.
- The *remote interface* of an enterprise bean is a Java interface that defines the operations that can be performed on the enterprise bean to access the business logic associated with the enterprise bean itself. The methods defined in the remote interface can be accessed from within the same container or remotely via RMI-IIOP.
- The *local interface* of an enterprise bean is functionally similar to the remote interface, but the methods defined in the local interface are accessible only from within the same container.

The home, local home, remote, and local interfaces of an enterprise bean define which methods the enterprise bean exposes to a client. An EJB method permission is defined by an XML `method-permission` element in an EJB module's deployment descriptor and is used to assign groups of methods of the home, local home, remote, and local interfaces of an enterprise bean packaged in that EJB module to the security roles. This way, the Application Assembler can define the security view of the enterprise bean.

An EJB method permission specifies the methods of the home, local home, remote, and local interfaces that each of the listed security roles is allowed to invoke. This implies that an EJB method permission may include a list of one or more security roles and a list of one or more methods. In addition, a security role or a method may appear in multiple XML `method-permission` elements. Users of particular security roles are granted access to all the methods listed in all the EJB method permission elements where those security roles appear. EJB method permissions and the deployment descriptor are discussed further in Chapter 5 on page 157.

3.7.2.2 Defining Web Resources Security Constraints

An Application Assembler uses a Web module's deployment descriptor to define security constraints for a Web application packaged in that module. The Web

module's deployment descriptor's `auth-constraint` element is used for this purpose. This element consists of one or more security roles and a list of URL patterns that users with any of those security roles are authorized to invoke. Specifically, deployment descriptors are used to assign groups of URL patterns to the security roles, thus defining security views of Web applications. Login configuration information, such as requiring a user to be authenticated using a form-based login mechanism, and transport guarantee constraints, such as requiring access to a URL pattern to be submitted only using an HTTPS connection, can also be specified in the deployment descriptor.

3.7.2.3 Declaring Security Roles within a J2EE Application

An Application Assembler defines each security role by using the `security-role` XML element in the relevant deployment descriptor.

- If the deployment descriptor belongs to an ejb-jar file, the `security-role` element is scoped to that ejb-jar file and applies to all the enterprise beans in that EJB module.
- If the deployment descriptor belongs to a WAR file, the security role element is scoped to that WAR file and applies to all the servlets and/or JSP files in that Web module.
- If the deployment descriptor belongs to an EAR file, the `security-role` element applies to all the JAR and WAR files that are packaged within that EAR file. Effectively, the set of security roles declared in the EAR file's deployment descriptor is the union of the security roles defined in the deployment descriptors of the JAR and WAR files packaged within that EAR file. Technically, however, the security roles described in the constituent modules of an EAR file are the ones that are used to enforce authorization, because those roles are associated with authorization policies. The roles declared in the EAR file's deployment descriptor are typically used for administration and management purposes only. For example, they can be used to assign security roles to principals with regard to the whole application packaged in the EAR file.

The deployment descriptor of an application client module does not contain security role elements, because security roles are specific to the server side of a J2EE application, not for the client.

Within each `security-role` element, the Application Assembler will use the `role-name` subelement to define the name of the security role and, optionally, will use the `description` subelement to provide a description of the security role.

3.7.3 Deployer

For each J2EE application, the Deployer takes the modules comprising that application and deploys the module components into a specific operational, or runtime, environment. The modules were produced by an Application Assembler. The operational environment in which the application is deployed includes a specific J2EE container. The Deployer is also responsible for ensuring the security of an assembled application when it is deployed in the target operational environment.

The Deployer has the following responsibilities with respect to security management.

3.7.3.1 Reading the Security View of the J2EE Application

The Deployer uses the deployment tools supplied by the J2EE Product Provider to read the security view of the application. The Deployer should treat the security policies specified in a deployment descriptor as hints and modify those policies as appropriate to the operational environment in which the application is being deployed.

3.7.3.2 Configuring the Security Domain

A *security domain* within an enterprise represents an instance of an authentication authority and relevant security infrastructure. For example, a security domain may point to a particular Kerberos domain for authentication and an LDAP user repository to deduce user and group membership to be used for authorization. In the case of multiple security domains within the enterprise, the Deployer is responsible for configuring the J2EE product to use the appropriate security domains.

3.7.3.3 Assigning of Principals to Security Roles

The Deployer is responsible for assigning principals and/or groups of principals used for managing security in the runtime to the security roles defined in the XML `security-role` elements of the deployment descriptors. The process of assigning the logical security roles defined in the J2EE application's deployment descriptor to the operational environment's security concepts is specific to the configuration capabilities of a particular J2EE product.

3.7.3.4 Configuring Principal Delegation

Delegation allows an intermediary to perform a task, initiated by a client, under an identity based on a *delegation policy*. The Deployer is responsible for configuring principal delegation for intercomponent calls by using the appropriate deployment descriptor elements, as follows.

- If the deployment descriptor belongs to an ejb-jar file, the Deployer uses the `security-identity` deployment descriptor element for this purpose.

When the value of the `security-identity` element is `use-caller-identity`, the identity of the caller of the enterprise bean will be used when calling other components from the enterprise bean. When the value specified is `run-as`, the identity of the caller to the enterprise bean will be propagated in terms of the security role name defined in the `run-as` element of the descriptor. For example, if the caller to an enterprise bean is user Bob, Bob's identity will be used if the `security-identity` element is set to `use-caller-identity`. If the `security-identity` element is set to `run-as` and the role name is `Teller`, the downstream calls from the enterprise bean will be performed in terms of the Teller role.

- If the deployment descriptor belongs to a WAR file, the Deployer uses the `run-as` deployment descriptor element for this purpose. If the `run-as` element is not declared in the WAR file, the identity of the servlet's caller will be used for components called from the servlet. If the `run-as` element is declared in the deployment descriptor, the identity passed when the servlet makes calls to other components will be that of the security role name defined in the `run-as` element of the descriptor. For example, if the caller to a servlet is user Bob, Bob's identity will be used if no `run-as` element is declared in the WAR file's deployment descriptor. If the `run-as` element is declared in the deployment descriptor and the role name is `Teller`, the downstream calls from the servlet will be performed in terms of the Teller role.

3.7.4 System Administrator

The System Administrator is responsible for the configuration and administration of the enterprise's computing and networking infrastructure, including the J2EE container. The System Administrator is also responsible for the overall management and operational aspects of the J2EE applications at runtime. The following list describes the security-related responsibilities of the System Administrator. Some of these responsibilities may be carried out by the Deployer or may require the cooperation of both the Deployer and the System Administrator.

3.7.4.1 Administering the Security Domain

The System Administrator is responsible for administering the security domain. This includes the principal administration, user account management, group membership assignment, deployment of J2EE products within an enterprise environment, including configuration of DMZs, firewalls, user registries, and so on. These are typically performed using the tools provided by the relevant product vendor; for example, user registry management is performed using an LDAP server product, firewall configuration using the firewall product, and so on.

3.7.4.2 Assigning Application Roles to Users and Groups

The System Administrator is responsible for assigning principals and/or groups of principals used for managing security in the runtime to the security roles defined in the XML `security-role` elements of the deployment descriptors. The process of assigning the logical security roles defined in the J2EE application's deployment descriptor to the operational environment's security concepts is specific to the configuration capabilities of a particular J2EE product. For example, using the tools provided by a J2EE product, the System Administrator can assign a Teller role scoped to a J2EE application FinanceApp to user Bob and group TellerGroup.

3.7.5 J2EE Product Provider

The J2EE Product Provider has the following areas of responsibility.

3.7.5.1 Supplying Deployment Tools

The J2EE Product Provider is responsible for supplying the deployment tools that the Deployer uses to perform all the deployment tasks, including the security-related tasks. For example, the J2EE Product Provider will supply tools to perform security-role-to-principal and/or -user and/or -group assignment.

3.7.5.2 Configuring Security Domains

The J2EE Product Provider is responsible for configuring the J2EE product to use appropriate security domains. For example, the J2EE Product Provider needs to supply facilities to configure the J2EE product to use a particular authentication mechanism—for example, to use a Kerberos domain.

3.7.5.3 Supplying Mechanisms to Enforce Security Policies

The J2EE Product Provider is responsible for supplying the security mechanisms necessary to enforce the security policies set by the Deployer. This includes authentication of principals, authorization to perform EJB/servlet calls, configuration of resource adapters defined in the JCA, and secure communication with remote clients, integrity, and confidentiality.

3.7.5.4 Providing Tools for Principal Delegation

The J2EE Product Provider is responsible for passing principals on EJB/servlet calls. In particular, the J2EE Product Provider is responsible for providing the deployment tools that allow the Deployer to configure principal delegation for calls from one J2EE component to another.

3.7.5.5 Providing Access to the Caller's Security Context

The J2EE Product Provider is responsible for providing access to the caller's security context information when programmatically queried from enterprise beans

and servlets using the J2EE-defined security APIs. For example, when a servlet calls `getUserPrincipal()` on a `javax.servlet.http.HttpServletRequest` object, the J2EE Product Provider must return the `java.security.Principal` object representing the caller of the servlet.

3.7.5.6 Supplying Runtime Security Enforcement

One of the most significant responsibilities of the J2EE Product Provider is to supply *runtime security enforcement*, as follows.

- Provide enforcement of the client access control as specified by the current security policy.
- Isolate an enterprise bean instance from other instances and other application components running on the server, thus preventing unauthorized access to privileged information.
- Provide runtime facilities to implement the principal-delegation policies set in the deployment descriptor.
- Allow a J2EE application to be deployed independently multiple times, each time with a different security policy.

3.7.5.7 Providing a Security Audit Trail

Optionally, the J2EE Product Provider may provide a *security audit trail* mechanism whereby secure access to enterprise beans and Web resources is logged. Such audit logs can be used to determine the information about the activity on the J2EE components. For example, these logs can be used to discover unauthorized attempts to access enterprise beans and Web resources.

3.8 J2EE Security Roles

The J2EE authorization model is based on the concept of security roles. Security roles are different from J2EE platform roles. As noted in Section 3.7.2 on page 67, a security role is a semantic grouping of permissions that a given type of application users must be granted to be authorized to use the application. In contrast, a J2EE platform role represents the organizational responsibility in making a J2EE application available to an enterprise, as described in Section 3.7 on page 64. Both declarative and programmatic security are based on the security roles.

Security roles are defined by the Application Component Provider and the Application Assembler. The Deployer then maps each security role to one or more security identities, such as users and groups, in the runtime environment. Listing 3.1 is an example of an XML description of two security roles declared within the deployment descriptor of an application's EAR file.

Listing 3.1. Description of Security Roles Teller and Supervisor

```
<assembly-descriptor>
   <security-role>
      <description>
         This role is intended for employees who provide
         services to customers (tellers).
      </description>
      <role-name>Teller</role-name>
   </security-role>
   <security-role>
      <description>
         This role is intended for supervisors.
      </description>
      <role-name>Supervisor</role-name>
   </security-role>
</assembly-descriptor>
```

Declarative authorization can be used to control access to an enterprise bean method. This contract is specified in the deployment descriptor. Enterprise bean methods can be associated with `method-permission` elements in EJB modules' deployment descriptors. As described in Section 3.7.2 on page 67, a `method-permission` element specifies one or more EJB methods that are authorized for access by one or more security roles. In Section 3.7.3 on page 70, we observed that the mapping of principals to security roles is performed by the Deployer. If the calling principal is in one of the security roles authorized access to a method, the caller is allowed to execute the method. Conversely, if the calling principal is not a member of any of the roles, the caller is not authorized to execute the method. Listing 3.2 is an example of an XML `method-permission` element in an EJB module's deployment descriptor.

Listing 3.2. Example of an XML `method-permission` Element in an EJB Module's Depolyment Descriptor

```
<method-permission>
   <role-name>Teller</role-name>
   <method>
      <ejb-name>AccountBean</ejb-name>
      <method-name>getBalance</method-name>
   </method>
   <method>
      <ejb-name>AccountBean</ejb-name>
      <method-name>getDetails</method-name>
   </method>
</method-permission>
```

Access to Web resources can be similarly protected. An action on a Web resource URI can be associated with an XML `security-constraint` element in a

Web module's deployment descriptor. The `security-constraint` element contains one or more URI patterns that can be authorized for access by one or more security roles. If the calling principal is a member of one or more of the security roles authorized to access an HTTP method on a URI, the principal is authorized to access the URI. Conversely, if the calling principal is not a member of any of the roles, the caller is not allowed to access the URI. Listing 3.3 shows a Web module's deployment descriptor fragment that defines access authorization requirements for a Web application.

Listing 3.3. Example of an XML `security-constraint` Element in a Web Module's Deployment Descriptor

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>
            Account servlet protected area
        </web-resource-name>
        <url-pattern>/finance/account/</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
        <description>Teller can access the URIs</description>
        <role-name>Teller</role-name>
    </auth-constraint>
</security-constraint>
```

In a J2EE environment, the security roles form the basis for the security provided by the containers that host the components. A container can provide two types of security: declarative and programmatic.

In the *declarative security model*, an application expresses its security policies through security constraints in a form external to the application. In J2EE, security constraints are specified in the deployment descriptors. This allows the application to be *security-mechanism agnostic*, or *security unaware*. Application code is not required to enable security or enforce the application security policy.

With declarative security, the application's logical security requirements are defined in the deployment descriptors and then mapped by the Deployer and the System Administrator to a deployment environment. The Deployer uses container-deployment tools to process the deployment descriptors. At runtime, the container uses the security policy configured by the Deployer and the System Administrator to enforce authorization.

Declarative security allows greater opportunities for application portability because all the security issues related to the underlying container and operating system are defined in configuration files external to the application. In addition, an application that makes use of declarative security is easier to develop because security and policy configuration issues are managed outside the application.

Application developers need not to be security experts, and an application based on declarative security can be more easily extended. Therefore, declarative security should always be used instead of programmatic security unless declarative security alone is insufficient to describe the security requirements of an application.

In the *programmatic security model*, the application programmer is responsible for explicitly writing the code that defines and enables security. The application security policy is an integral part of the application. An application conforming to this model is said to be *security aware*.

Programmatic security makes application development more difficult and severely limits the portability and extensibility of an application, because security issues related to the specific application, container, and operating system on which the application is running must be hard-coded. For these reasons, programmatic security should be used only when declarative security alone is insufficient to express the security model of an application. For example, the declarative security capabilities of J2EE V1.3 do not allow expressing a policy whereby a user cannot withdraw more than $1,000 from an automatic teller machine (ATM). Similarly, instance-level authorization to impose that only Bob can access Bob's account cannot be defined declaratively. In these cases, the application needs to enforce these rules programmatically.

3.9 Declarative Security Policies

Security policies associated with URIs and enterprise beans include the following:

- Login configurations associated with URIs: for example, use of form-based login
- Authorization policies associated with URIs and enterprise beans based on J2EE security roles
- Principal-delegation policies that apply to Web applications and enterprise beans
- Connection policies associated with JCA connectors that dictate how applications access EIS in a secure manner

Such authorization and delegation policies can be specified declaratively within the relevant deployment descriptors.

3.9.1 Login-Configuration Policy

Authentication is the process of proving the identity of an entity. Authentication generally is performed in two steps: (1) acquiring the authentication data of a principal and (2) verifying the authentication data against a user (principal) registry.

J2EE security authenticates a principal on the basis of the authentication policy associated with the resource the principal has requested. When a user requests a protected resource from a Web application server, the server authenticates the user. J2EE servers use authentication mechanisms based on validating credentials, such as digital certificates (see Section 10.3.4 on page 372), and user ID and password pairs. Credentials are verified against a user registry that supports the requested authentication scheme. For example, authentication based on user ID and password can be performed against an LDAP user registry, where authentication is performed using an LDAP bind request.

A Web server is responsible for servicing HTTP requests. In a typical J2EE environment, a Web server is a component of a J2EE WAS. In this case, the WAS hosts servlets, JSP files, and enterprise beans. The login—authentication—configuration is managed by the WAS, which drives the authentication challenges and performs the authentication. Similarly, if the Web server is independent of the WAS and the Web server is the front end for the WAS, the Web server acts as a proxy for J2EE requests. Again, the authentication is typically performed by the WAS.

The authentication policy for performing authentication among a user, a Web server, and a WAS can be specified in terms of the J2EE login configuration elements of a Web application's deployment descriptor. The authentication policy can specify the requirement for a secure channel and the authentication method. The requirement to use a secure channel when accessing a URI is specified through the `user-data-constraint` descriptor.

The authentication method is specified through the `auth-method` element in the `login-config` descriptor. There are three types of authentication methods:

1. **HTTP authentication method.** The credentials that the client must submit to authenticate are user ID and password, sent to the server as part of an HTTP header and typically retrieved through a browser's dialog window. The two modes of HTTP authentication are *basic* and *digest*. In both cases, the user ID is sent as cleartext.[2] In basic authentication, the password is transmitted in cleartext as well; in digest authentication, only a hash value of the password is transmitted to the server (see Section 10.2.2.4 on page 356).

2. More precisely, the cleartext is encoded in *base64 format*, a commonly used Internet standard. Binary data can be encoded in base64 format by rearranging the bits of the data stream in such a way that only the six least significant bits are used in every byte. Encoding a string in base64 format does not add security; the algorithm to encode and decode is fairly simple, and tools to perform encoding and decoding are publicly available on the Internet. Therefore, a string encoded in base64 format is still considered to be in cleartext.

2. **Form-based authentication method.** The credentials that the client must submit to authenticate are user ID and password, which are retrieved through an HTML form.
3. **Certificate-based authentication method.** The credential that the client must submit is the client's digital certificate, transmitted over an HTTPS connection.

3.9.1.1 Authentication Method in Login Configuration

The `auth-method` element in the `login-config` element specifies how a server challenges and retrieves authentication data from a user. As noted previously, there are three possible authentication methods: HTTP (user ID and password), form based (user ID and password), and certificate based (X.509 certificate).

With the *HTTP authentication method*, the credentials provided by the user consist of a user ID and password pair, transmitted as part of an HTTP header. When HTTP authentication is specified, a user at a Web client machine is challenged for a user ID and password pair. The challenge usually occurs in the following way:

1. A WAS issues an HTTP unauthorized client error code (401) and a `WWW_Authenticate` HTTP header.
2. The Web browser pops up a dialog window.
3. The user enters a user ID and password pair in this dialog window.
4. The information is sent to the Web server.
5. The WAS extracts the information and authenticates the user, using the authentication mechanism with which it has been configured.

With HTTP authentication, a realm name also needs to be specified. *Realms* are used to determine the scope of security data and to provide a mechanism for protecting Web application resources. For example, a user defined as `bob` in one realm is treated as different from `bob` in a second realm, even if these two IDs represent the same human user, Bob Smith.

Once specified, the realm name is used in the HTTP 401 challenge to help the Web server inform the end user of the name of the application domain. For example, if the realm is SampleAppRealm, the dialog window prompting the user for a user ID and password pair during authentication will include that the user ID and password are to be supplied for the SampleAppRealm realm.

HTTP authentication can be either basic or digest. In *basic authentication,* the credentials requested of the user are user ID and password, and both are transmitted as cleartext. In order for the authentication method to be basic, the `auth-method` element in the `login-config` descriptor must be set to `BASIC`. Listing 3.4

is a deployment descriptor fragment showing an example of login configuration requiring basic authentication.

Listing 3.4. Login Configuration for Basic Authentication

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

This scheme is not considered to be a secure method of user authentication, unless used in conjunction with some external secure systems, such as SSL.

In *digest authentication*, the user ID and a hash value of the password are transmitted to the server as part of an HTTP header. Therefore, the password does not appear in cleartext, which is the biggest weakness of basic authentication.

When digest authentication is specified, the Web server responds to the client's request by requiring digest authentication. A one-way hash of the password (see Section 10.2.2.4 on page 356), as specifed by the Request for Comments (RFC) 2617,[3] is computed by the client, based on a random number, called *nonce*, uniquely generated by the server each time a 401 response is made. The hash value of the password is sent to the server, which computes the digest of the password for the user ID and compares the resulting hash value with the one submitted by the client. The requesting user is considered to be authenticated if the hash values are identical.

This mode of authentication assumes that the server has access to the user's password in cleartext—a necessary requirement in order for the server to compute the hash of the password. However, this is rarely the case in most enterprise environments, as the password in cleartext is not retrievable from a user repository containing the user ID and password information. Rather, the server typically delegates responsibility to the user repository to validate a user's password. Therefore, digest authentication is not widely adopted in enterprise environments and hence is not required to be supported by a J2EE container.

J2EE servers that do support digest authentication can be configured to issue a digest authentication challenge by setting the value of the `auth-method` element in the `login-config` descriptor to `DIGEST`. Listing 3.5 is a deployment descriptor fragment illustrating how a J2EE server can be configured to require digest authentication.

3. See `http://www.ietf.org/rfc/rfc2617.txt`.

Listing 3.5. Login Configuration for Digest Authentication

```
<login-config>
    <auth-method>DIGEST</auth-method>
    <realm-name>SampleAppRealm</realm-name>
</login-config>
```

The second authentication method is form based. With this method, the `auth-method` element in the `login-config` element must be set to `FORM`. The form-based authentication method assumes that the server is configured to send the client an HTML form to retrieve the user ID and password from the Web user, as opposed to sending a 401 HTTP unauthorized client error code as in the basic challenge type.

The configuration information for a form-based authentication method is specified through the `form-login-config` element in the `login-config` element. This element contains two subelements: `form-login-page` and `form-error-page`.

- The Web address to which a user requesting the resource is redirected is specified by the `form-login-page` subelement in the Web module's deployment descriptor. When the form-based authentication mode is specified, the user will be redirected to the specified `form-login-page` URL. An HTML form on this page will request a user ID and password.
- If the authentication fails, the user is redirected to the page specified by the `form-error-page` subelement.

Listing 3.6 is a sample HTML page for the login form.

Listing 3.6. Login Page Contents

```
<HTML>
    <HEAD>
        <TITLE>Sample Login page.</TITLE>
    </HEAD>
    <BODY>
        <TR><TD>
        <HR><B>Please log in!</B><BR><BR>
        </TD></TR>
        <CENTER>
        Please enter the following information:<BR>
        <FORM METHOD=POST ACTION="j_security_check">
            Account <INPUT TYPE=text NAME="j_username"
                SIZE=20><BR>
            Password <INPUT TYPE=password
                NAME="j_password" SIZE=20><BR>
```

```
            <INPUT TYPE=submit NAME=action
               VALUE="Submit Login">
        </FORM><HR>
        </CENTER>
    </BODY>
</HTML>
```

Listing 3.7 is a deployment descriptor fragment showing an example of login configuration that requires form-based authentication.

Listing 3.7. Login Configuration for Form-Based Authentication

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

The third type of authentication method is *certificate based* (X.509 certificate). In order for the authentication method to be certificate based, the `auth-method` element in the `login-config` descriptor must be set to `CLIENT-CERT`. The certificate-based authentication method implies that the Web server is configured to perform mutual authentication over SSL. The client is required to present a certificate to establish the connection. When the `CLIENT-CERT` mode is specified, the client will be required to submit the request over an HTTPS connection. If the request is not already over HTTPS, the J2EE product will redirect the client over an HTTPS connection. Successful establishment of an SSL connection implies that the client has presented its own certificate and not anyone else's. The details of how the server ensures that the client certificate really belongs to the client are explained in Section 10.3.4 on page 372 and Section 13.1.2 on page 452. The certificate used by the client is then mapped to an identity in the user registry the J2EE product is configured to use.

Listing 3.8 is a deployment descriptor fragment showing an example of login configuration that requires certificate-based authentication.

Listing 3.8. Login Configuration for Certificate-Based Authentication

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

Note that the user registry is not specified in this XML deployment descriptor fragment because it is not part of the J2EE specification.

3.9.1.2 Secure-Channel Constraint

Establishing an HTTPS session between the client and the Web server is often a necessary requirement to provide data confidentiality and integrity for the information flowing between the HTTP client and the server. In a J2EE environment, the security policy can require the use of a secure channel, specified through the `user-data-contraint` deployment descriptor element. When the requirement for a secure channel is specified, the request to the URI resource should be initiated over an HTTPS connection. If access is not already via a HTTPS session, the request is redirected over an HTTPS connection.

Specifying `INTEGRAL` or `CONFIDENTIAL` as the value for the `transport-guarantee` element in the `user-data-constraint` descriptor will be treated as a requirement for the HTTP request to be over SSL. This requirement can be specified as part of the `user-data-constraint` element in a Web application's login configuration. In theory, `INTEGRAL` should enforce communitcation integrity, whereas `CONFIDENTIAL` should enforce communication confidentiality, and it could be possible to select different cipher suites to satisfy these requirements. However, a J2EE server typically does not differentiate `INTEGRAL` from `CONFIDENTIAL` but instead treats both of these values to indicate the need to require an SSL connection with a particular cipher suite, not based on whether `INTEGRAL` or `CONFIDENTIAL` was specified.

Listing 3.9 is a deployment descriptor fragment showing an example of login configuration that contains the `user-data-constraint` element. More details are provided in Section 4.6.6 on page 132.

Listing 3.9. Specifying the Requirement for a Secure Channel

```
<user-data-constraint>
   <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

3.9.2 Authorization Policy

The *role-permission interpretation* of the J2EE security model treats a security role to be a set of permissions. The security role uses the `role-name` label defined in the `method-permission` element of an EJB module's deployment descriptor and in the `security-constraint` element of a Web module's deployment descriptor as the name of the set of permissions. The set of permissions defines a number of *resources*—the enterprise beans and the Web resources to which the

`method-permission` and `security-constraint` elements refer, respectively—and a set of *actions*—the methods listed by the `method-permission` and the `security-constraint` descriptors. For example, in Listing 3.2 on page 74, the security role Teller is associated with the permissions to invoke the `getBalance()` and `getDetails()` methods on the AccountBean enterprise bean. Similarly, in Listing 3.3 on page 75, the security role Teller is associated with the permission to perform a `GET` invocation over HTTP to the `/finance/account/` URI. If multiple `method-permission` and `security-constraint` descriptors refer to the same security role, they are all taken to contribute to the same role permission. In other words, the sets of permissions associated with that security role are merged to form a single set.

This model has the advantage of dramatically reducing the number of objects in a *security object space*—a set of pairs (*subject*, <*target*, *operation*>), where the *subject* is an entity requesting to perform a security-sensitive *operation* on a given *target*. The Deployer and the System Administrator can define authorization policies, associated with EJB or URI targets and the operations of enterprise bean methods and HTTP methods, respectively, for the security roles in their applications. Then, they associate subjects to security roles; by extension, those subjects are granted the permissions to perform the operations permitted by the security roles.

Based on the J2EE security model, a protected action can be performed by a subject who has been granted at least one of the security roles associated with the action. The security roles associated with a protected action are the *required security roles*—the permissions necessary to perform the action itself. The roles associated with a subject are the *granted security roles*—the permissions that have been given to that subject. This means that the subject will be allowed to perform an action if the subject's granted security roles contain at least one of the required security roles to perform that action. For example, if the action consisting of accessing the EJB method `getDetails()` on the AccountBean enterprise bean can be performed only by the security roles Teller and Supervisor and if subject Bob has been granted the security role of Teller, Bob will be allowed to perform that action, even if Bob has not been granted the security role of Supervisor.

The table that represents the association of security roles to sets of permissions is called the *method-permission table*. A method-permission table (see Table 3.1) can be used to deduce the set of required security roles. The rows in the table represent security roles; the columns represent protected actions.

It can be inferred from Table 3.1 that in order to access the `getBalance()` method on AccountBean, the required security roles are Teller and Supervisor. In order to access any URI that matches the pattern `/public/*`, a PublicRole is required.

Table 3.1. Example of Method-Permission Table

	`/finance/ accountGET`	`/finance/ accountPUT`	`/public/*`	`AccountBean. getBalance()`	`AccountBean. getDetails()`
Teller	Yes	No	No	Yes	Yes
Supervisor	Yes	Yes	No	Yes	Yes
PublicRole	No	No	Yes	No	No

The table that represents the association of roles to subjects is called the *authorization table*, or *protection matrix*. In such a table, the security role is defined as the *security object*, and users and groups are defined as *security subjects*. An authorization table (see Table 3.2) can be used to deduce the set of granted security roles. The rows in the table refer to the users and user groups that are security subjects in the protection matrix; the columns represent the J2EE security roles that are security objects in the protection matrix.

The method-permission table and the protection matrix reflect the configuration specified in the deployment descriptors. For example, the first row in Table 3.1 reflects the deployment descriptor obtained from the deployment descriptor fragments of Listing 3.2 on page 74 and Listing 3.3 on page 75. It can be inferred from Table 3.2 that user Bob and group TellerGroup are granted the security role of Teller, everyone is granted the PublicRole, and only users in the ManagerGroup are granted the security role of Supervisor.

Combining Table 3.1 and Table 3.2, it follows that Bob can access the `getBalance()` and `getDetails()` methods on the AccountBean enterprise bean and can issue an HTTP `GET` request on the `/finance/account/` URI. Bob cannot, however, issue an HTTP `PUT` request on the `/finance/account/` URI. Note that Bob will be able to access any URI that matches `/public/*`, as everyone has been granted the role PublicRole, which is the role necessary to get access to `/public/*`.

In the J2EE security model, the Application Assembler defines the initial mapping of actions on the protected resources to the set of the required security

Table 3.2. Example of Authorization Table

	Teller	**Supervisor**	**PublicRole**
TellerGroup	Yes	No	No
ManagerGroup	No	Yes	No
Everyone	No	No	Yes
Bob	Yes	No	No

roles (see Section 3.7.2 on page 67). This can be done using the application assembly tool. Subsequently, the Deployer will refine the policies specified by the Application Assembler when installing the application into a J2EE environment (see Section 3.7.3 on page 70). The Deployer also can use the application assembly tool to redefine the security policies, when necessary, and then install the application into the J2EE container. The method-permission table is formed as a result of the required security roles getting specified through the process of application assembly and refinement during deployment.

Authorization policies can be broadly categorized into *application policies*, which are specified in deployment descriptors and map J2EE resources to roles, and *authorization bindings*, which reflect role to user or group mapping. As discussed in Section 3.7.2 on page 67, a set of security roles is associated with actions on J2EE protected resources. These associations are defined in the J2EE deployment descriptors when an application is assembled and deployed. The security roles specified in this way are the required security roles—the sets of permissions that users must be granted in order to be able to perform actions on protected resources. Pragmatically, before a user is allowed to perform an action on a protected resource, either that same user or one of the groups that user is a member of should be granted at least one of the required security roles associated with that protected resource. The authorization table that relates the application-scoped required security roles to users and user groups is managed within the J2EE Product Provider using the J2EE Product Provider configuration tools.

3.9.3 Delegation Policy

Earlier in this chapter, we defined *delegation* as the process of forwarding a principal's credentials with the cascaded downstream requests. Enforcement of delegation policies affects the identity under which the intermediary will perform the downstream invocations on other components. By default, the intermediary will impersonate the requesting client when making the downstream calls. The downstream resources do not know about the real identity, prior to impersonation, of the intermediary. Alternatively, the intermediary may perform the downstream invocations using a different identity. In either case, the access decisions on the downstream objects are based on the identity at the outbound call from the intermediary. To summarize, in a J2EE environment, the identity under which the intermediary will perform a task can be either

- The client's identity—the identity under which the client is making the request to the intermediary
- A specified identity—an identity in terms of a role indicated via deployment descriptor configuration

The application deployment environment determines whether the client or a specified identity is appropriate.

The Application Assembler can use the `security-identity` element to define a delegation identity for an enterprise bean's method in the deployment descriptor. Consider an example in which a user, Bob, invokes methods on a SavingsAccountBean enterprise bean. SavingsAccountBean exposes three methods—`getBalance()`, `setBalance()`, and `transferToOtherBank()`—and its delegation policy is defined as in Table 3.3. Figure 3.5 shows a possible scenario based on the delegation policy specified in Table 3.3.

The method `setBalance()` will execute under the client's identity because the delegation mode is set to `use-caller-identity`. The method `getBalance()` will execute under the client's identity as well because no delegation mode is specified, and the default is `use-caller-identity`. Therefore, if Bob invokes the method `getBalance()` on AccountBean, the method will execute under Bob's identity, `bob`. Suppose that the `getBalance()` method invokes a `lookup()` method on

Table 3.3. SavingsAccountBean Enterprise Bean's Delegation Policy

Method	Delegation Mode	Specified Role
`getBalance()`		
`setBalance()`	`use-caller-identity`	
`transferToOtherBank()`	`run-as`	Supervisor

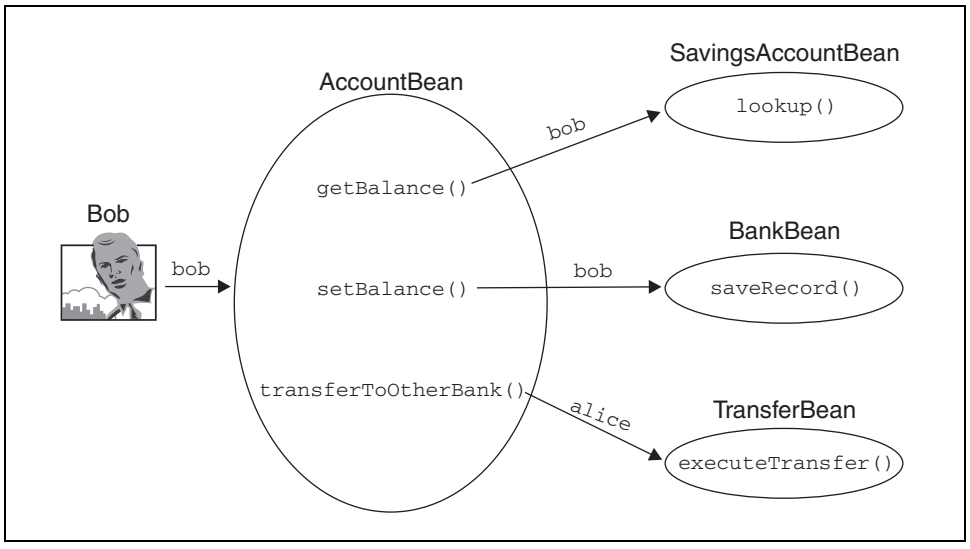

Figure 3.5. Delegation Policy Scenario

SavingsAccountBean. This invocation will still be executed under Bob's identity and will succeed only if Bob has been granted the permission to invoke `lookup()` on SavingsAccountBean.

Any downstream call from `transferToOtherBank()` will perform method calls on a TransferBean enterprise bean. These invocations will need to execute under a principal that has been granted the Supervisor role. The Deployer or the System Adminstrator needs to map the Supervisor role to a principal that has been granted the Supervisor role. This can be done by specifying a valid user ID and password pair corresponding to a user who has been granted that role. For example, if user Alice has been granted the Supervisor role and if the user ID and password pair for Alice is associated with the Supervisor role, the calls to `transferToOtherBank()` will occur under Alice's identity.

3.9.4 Connection Policy

Information in any EIS must be protected from unauthorized access. An EIS system is likely to have its own authorization model. At a minimum, most of these systems have facilities to accept some form of authentication data representing an identity connecting to the EIS. The JCA is designed to extend the end-to-end security model for J2EE-based applications to include integration with EISs. A WAS and an EIS collaborate to ensure the proper authentication of a resource principal when establishing a connection to a target EIS. As discussed in Section 3.4 on page 61, the JCA allows for two ways to sign on to an EIS: container-managed sign-on and component-managed sign-on.

With container-managed sign-on, the connection to an EIS is obtained through declarative security. In order for a connection to be container managed, the deployment descriptor will indicate that the `res-auth` element associated with a resource definition is declared as `Container`. If the connection is obtained by passing the identity information programmatically, the value for `res-auth` should be set to `Application`. Details of component-managed sign-on are discussed in Section 3.10.3 on page 94.

A deployment descriptor fragment that declares that the authentication facilitated by the resource adapter should be set to be `Container` is shown in Listing 3.10.

Listing 3.10. An XML `res-auth` Element in a Deployment Descriptor

```
<resource-ref>
    <description>Connection to myConnection</description>
    <res-ref-name>eis/myConnection</res-ref-name>
    <res-type>javax.resource.cci.ConnectionFactory</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

The container is responsible for obtaining appropriate user authentication information needed to access the EIS. The connection to the EIS is facilitated by the specified resource adapter. The JCA allows specifying the authentication mechanism. The `authentication-mechanism-type` element in the deployment descriptor is used to specify whether a resource adapter supports a specific authentication mechanism. This XML element is a subelement of the `authentication-mechanism` element. The JCA specification supports the following authentication mechanisms:

- **Basic authentication.** The authentication mechanism is based on user ID and password. In this case, the `authentication-mechanism-type` XML element in the deployment descriptor is set to `BasicPassword`.
- **Kerberos V5.** The authentication mechanism is based on Kerberos V5. In this case, the `authentication-mechanism-type` element in the deployment descriptor is set to `Kerbv5`.

Other authentication mechanisms are outside the scope of the JCA specification.

In a secure environment, it is likely that a J2EE application component, such as an enterprise bean, and the EIS system that is accessed through the component are secured under different security domains, where a *security domain* is a scope within which certain common security mechanisms and policies are established. In such cases, the identity under which the J2EE component is accessed should be mapped to an identity under which the EIS is to be accessed. Figure 3.6 depicts a possible scenario.

In this scenario, an enterprise bean in a J2EE container is accessed by a user, Bob Smith. The enterprise bean is protected in a way that it allows only users from a specified LDAP directory to access it. Therefore, the identity under which Bob Smith will access the enterprise bean must be registered in that LDAP directory. Bob Smith uses the identity of `bsmith` when he accesses the enterprise bean.

In a simplistic case, where the `run-as` policy of the enterprise bean is set to be the caller identity, the connections to the EIS will be obtained on behalf of Bob Smith. If the connections are obtained through user ID and password, when the enterprise bean obtains a connection to a back-end system, such as a CICS system, the J2EE container will retrieve a user ID and password to act on behalf of user `bsmith`. The application invokes the `getConnection()` method on the `javax.resource.cci.ConnectionFactory` instance (see Listing 3.10 on page 87) with no security-related parameters, as shown in Listing 3.11, a fragment of Java code.

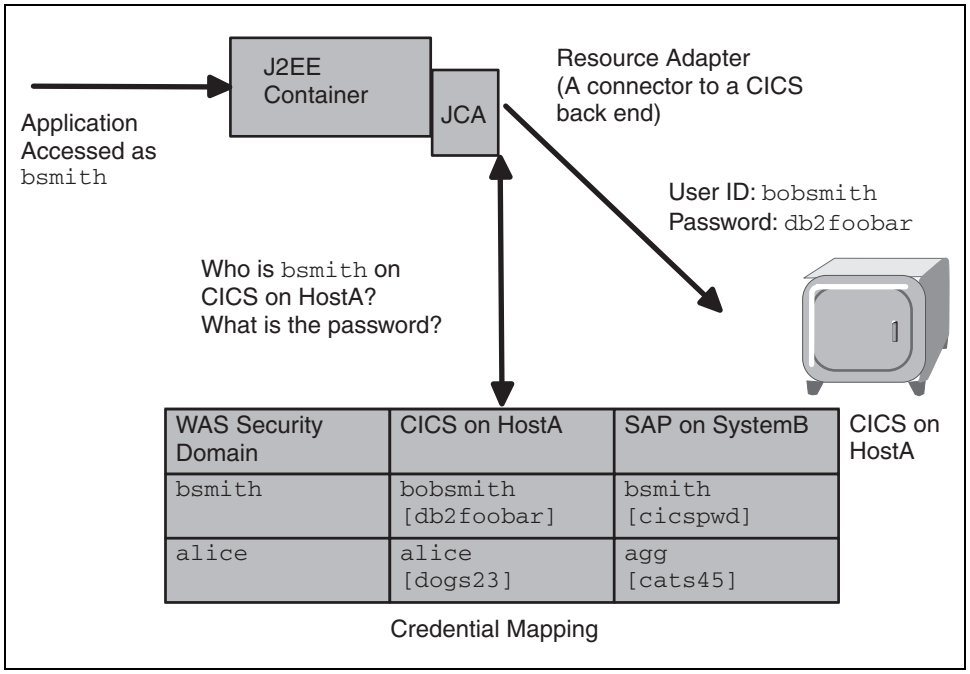

Figure 3.6. Credential Mapping when Accessing an EIS from a J2EE Container

Listing 3.11. Getting a Connection to an EIS with Container-Managed Sign-On

```
// Construct the InitialContext
Context initctx = new InitialContext();

// Perform a JNDI lookup to obtain a ConnectionFactory
javax.resource.cci.ConnectionFactory cxf =
    (javax.resource.cci.ConnectionFactory) initctx.lookup
    ("java:comp/env/eis/MyEIS");

// Invoke the ConnectionFactory to obtain a connection.
// The security information is not passed to the
// getConnection() method
javax.resource.cci.Connection cx = cxf.getConnection();
```

The application relies on the container to manage the sign-on to the EIS instance. This is possible in simple deployment scenarios in which the identity under which the EIS system is accessed is specified by the Deployer. This effectively means that all identities accessing the application are mapped to a single identity to access the EIS system: a *many-to-one identity mapping*.

In more sophisticated deployment scenarios, a many-to-one identity mapping may not be sufficient for security policy reasons. For example, it may be necessary for the EIS system to log all the identities that accessed it. For this logging facility to be useful, the identities accessing a J2EE application must not all be mapped to the same identity on the EIS system. A *one-to-one* or *many-to-many identity mapping* is recommended in this case. In particular, the container may use a credential mapping facility whereby `bsmith` is mapped to user ID `bobsmith` and password `db2foobar`, as shown in Figure 3.6.

If connections require Kerberos credentials or other generic credentials to be passed, the mapping facility is responsible for mapping one form of the credential to another that can be used by the target security domain. The manner in which these mappings happen and the level of sophistication in mapping available in J2EE application servers are server specific and not dictated by the J2EE specification.

In enterprise environments consisting of multiple departments, organizations, and even acquired companies, it is typical for systems to be interconnected and the applications shared. In such environments in which J2EE applications are deployed, it is a good architectural approach to design the application integration in a way that applications use JCA to obtain connections to other applications and to follow the declarative approach to define connection sign-on, as explained in this section. The use of JCA will make applications unaware of cross-security domains when accessing non-J2EE systems, and the use of declarative security will enhance application flexibility and portability. JCA with declarative security will also help manage the mapping of credentials and identities outside the application as enforced and facilitated by the enterprise-level mapping infrastructure.

3.10 Programmatic Security

Declarative security should always be used instead of programmatic security whenever possible. However, when declarative security is insufficient, it may be necessary to retrieve security-sensitive information programmatically from the container. This section explains how to retrieve the user's identity and privilege information programatically.

Applications that make use of programmatic security typically invoke the following EJB and servlet/JSP security APIs:

- **EJB method `isCallerInRole()` in interface `javax.ejb.EJBContext`.** This method is used to test whether the current caller, the client, has been assigned to a specified security role.

- **EJB method `getCallerPrincipal()` in interface `javax.ejb.EJBContext`.** This method is used to obtain a `Principal` object representing the current caller, the client.
- **Servlet/JSP method `isUserInRole()` in interface `javax.servlet.http.HttpServletRequest`.** This method, similar to the EJB method `isCallerInRole()`, returns a `boolean` indicating whether the authenticated user, the client, is a member of the specified security role.
- **Servlet/JSP method `getUserPrincipal()` in interface `javax.servlet.http.HttpServletRequest`.** This method, similar to the EJB method `getCallerPrincipal()`, returns a `Principal` object representing the current authenticated user, the client.

3.10.1 Retrieving Identity Information

The Java Servlet and EJB specifications provide mechanisms to programmatically obtain identity information about the user invoking a method on a servlet or an enterprise bean.

3.10.1.1 From a Servlet or JSP File

The `HttpServletRequest` object passed to a servlet method can be used to obtain information about the user invoking the method. Invoking the `getRemoteUser()` method on the `HttpServletRequest` object returns the name of the user if the user has been authenticated, `null` otherwise.

The `getRemoteUser()` method can be invoked as shown in Listing 3.12.

Listing 3.12. Retrieving the User Name from a Servlet

```
public void doGet(HttpServletRequest req,
    HttpServletResponse res)
{
    // other code...

    // obtain the user name
    String userName = req.getRemoteUser();

    // other code...
}
```

The `getUserPrincipal()` method in the `HttpServletRequest` object returns the `Principal` object corresponding to the user if the user has been authenticated, `null` otherwise. The name of the user can then be obtained by calling the `getName()` method on the `Principal` object, if this not `null`, as shown in Listing 3.13.

Listing 3.13. Retrieving the `Principal` Object and the User Name from a Servlet

```
public void doGet(HttpServletRequest req,
    HttpServletResponse res)
{
    // other code...

    // obtain the user Principal
    Principal userPrincipal = req.getUserPrincipal();

    // obtain the user name
    String userName;

    if (userPrincipal != null)
        userName = userPrincipal.getName();

    // other code...
}
```

3.10.1.2 From an Enterprise Bean

The `getCallerPrincipal()` method can be called on a `javax.ejb.EJBContext` object to obtain the `Principal` object corresponding to the user making the enterprise bean method invocation. The `Principal` object can then be used to obtain information about the user. A code example is shown in Listing 3.14.

Listing 3.14. Retrieving the User Name from an Enterprise Bean

```
public String getUserName(EJBContext context)
{
       // obtain and return the user name
       return context.getCallerPrincipal().getName();
}
```

3.10.2 Proactive Authorization

The Java Servlet and EJB specifications provide mechanisms to programmatically obtain information about the user's privileges by invoking a method on a servlet or an enterprise bean.

3.10.2.1 From a Servlet or JSP File

The `HttpServletRequest` object passed to a servlet method can be interrogated to obtain information about whether the user invoking the method has been granted a particular security role. Based on the result, the servlet may make decisions on how to proceed. For example, if the caller is granted the Boss role, the servlet redirects to a page that has managerial capabilities; otherwise, it might redirect to a different page. Note that when the servlet checks for the Boss role, this

role is scoped to the servlet. The Application Assembler performs the mapping of this role reference to an enterprise application role by using the `role-link` tag in the deployment descriptor, as shown in Listing 3.15.

Listing 3.15. An XML `role-link` Element in a Deployment Descriptor

```
<security-role-ref>
    <role-name>Boss</role-name>
    <role-link>Manager</role-link>
</security-role-ref>
```

A code example is shown in Listing 3.16.

Listing 3.16. Retrieving the User's Role Information from a Servlet

```
public void doGet(HttpServletRequest req,
    HttpServletResponse res)
{
    // other code...

    if (req.isUserInRole("Boss"))
    {
        // code to redirect to Manager's page...
    }
    else
    {
        // code to redirect to generic page...
    }

    // other code...
}
```

3.10.2.2 From an Enterprise Bean

The `isCallerInRole()` method on an `EJBContext` object can be used to obtain information about the roles granted to a particular user. This is similar to what we saw in Section 3.10.1.1 on page 91. A code fragment example is shown in Listing 3.17.

Listing 3.17. Retrieving the User's Role Information from an Enterprise Bean

```
public Object getOrganizationInfo(EJBContext context)
{
    // other code...

    // obtain the user name
    if (context.isCallerInRole ("Boss")
    {
```

(*continues*)

Listing 3.17. Retrieving the User's Role Information from an Enterprise Bean

```
        // code to access the Boss entity bean and get the
        // budget info...
    }
    else
    {
        // code to access the employee bean and get the
        // organization chart...
    }

    // other code...
}
```

3.10.3 Application-Managed Sign-On to an EIS

Section 3.9.4 on page 87 described container-managed sign-on, whereby a connection to an EIS is obtained through declarative security. An alternative approach is to use programmatic security by allowing the sign-on to an EIS to be managed directly by the application. In order for a connection to be application managed, the value of the `res-auth` XML element associated with a resource definition in the deployment descriptor must be set to `Application`, as shown in Listing 3.18.

Listing 3.18. Setting the `res-auth` Deployment Descriptor Tag to `Application`

```
<resource-ref>
    <description>Connection to myConnection</description>
    <res-ref-name>eis/myConnection</res-ref-name>
    <res-type>javax.resource.cci.ConnectionFactory</res-type>
    <res-auth>Application</res-auth>
</resource-ref>
```

In the case of application-managed sign-on, the application is responsible for retrieving appropriate user information, such as the user ID and password, necessary to connect to the EIS. The connection is facilitated by a resource adapter. The application invokes the `getConnection()` method on the `ConnectionFactory` instance with the security information: user ID and password. Specifying security information is dependent on the resource adapter type and the way in which the adapter accepts the user ID and password. For example, in order to connect to an EIS system called MyEIS, the application may be required to pass user ID and password through a `com.myeis.ConnectionSpecImpl` object. Listing 3.19 is a Java code fragment showing such a scenario, which is similar to the one discussed in Section 3.9.4 on page 87, except that the security information is coded into the application and is passed to the `getConnection()` method.

Listing 3.19. Getting a Connection to an EIS with Container-Managed Sign-On

```
// Method in an application component
Context initctx = new InitialContext();

// Perform a JNDI lookup to obtain a ConnectionFactory
javax.resource.cci.ConnectionFactory cxf =
    (javax.resource.cci.ConnectionFactory) initctx.lookup
    ("java:comp/env/eis/MyEIS");

// Insert here the code to get a new ConnectionSpec
com.myeis.ConnectionSpecImpl props = // ...

// Set user ID and password
props.setUserName("bobsmith");
props.setPassword("db2foobar");

// Invoke the ConnectionFactory to obtain a connection.
// The security information is passed explicitly to the
// getConnection() method.
javax.resource.cci.Connection cx = cxf.getConnection(props);
```

3.11 Secure Communication within a WAS Environment

A fundamental aspect of any distributed computing system is remote communication. In an enterprise environment, components, such as Web servers, plug-ins, and WASs; external servers, such as LDAP directory servers; and clients communicate with one another over multiple protocols.

- HTTP clients invoke URL requests to Web servers over HTTP.
- WASs communicate with one another over IIOP.
- Some WASs may communicate with external systems by using other protocols. For instance, a WAS can communicate with an LDAP directory server over LDAP.

Because these components can host and distribute security-sensitive information, it is necessary to provide secure communication channels. The quality-of-service (QoS) should include encryption, integrity, and, possibly, authentication. The SSL protocol is generally used to meet these QoS requirements. Typically, two modes of SSL connections are used in J2EE:

1. **Server-side SSL.** The client connects to the server and attempts to verify the authenticity of the server's identity. The server does not verify the client's identity. If the client can authenticate the server successfully, the client/server communication is performed over an encrypted channel.

2. **Mutual-authentication SSL.** The client connects to the server. Both client and server attempt to authenticate to each other. If the mutual-authentication process is successful, client and server communicate over an encrypted channel.

The SSL configuration on the server side dictates whether a client connecting to the server should connect over server-side SSL or mutual-authentication SSL. In both cases, the strength of encryption depends on the configured cipher suite. In a Java environment, JSSE-compliant SSL providers are used to establish secure communication using SSL between the end points (see Chapter 13 on page 449).

The following list shows possible combinations of securely communicating parties:

- **Web client to Web server.** Any Web browser can issue a request to a Web server over a secure connection. This communication can be over either server-side SSL or mutual-authentication SSL. The Web server should be configured to accept connections from Web browsers over a secure-socket port—typically, port 443. If a WAS requires a client to present a client certificate in order to be authenticated to access a servlet, the underlying Web server should be configured to require mutual-authentication SSL connections from Web browsers. This configuration is specific to the underlying Web server that is used, whereas the client certificate information is configured on the Web browser and is specific to the browser settings.
- **Web server to WAS.** In general, a Web server needs a plug-in to communicate with a back-end WAS, unless the Web server and the WAS are integrated to form a single component. In a typical scenario, a Web server plug-in may communicate with an application server over HTTP. This can be configured to be over SSL, in which case the resulting protocol is HTTPS. In order for this to happen, the WAS transport must be configured to accept only secure connections. In the case of mutual-authentication SSL, the WAS transport can also be configured to trust only a set of selected clients to connect to the WAS. By properly configuring the digital-key storage facility—for example, a key file—both the plug-in and the WAS can be configured to accept a list of trusted Certificate Authorities (CAs), so that a trusted communication link can be established with only the clients whose certificates have been authenticated by one of the trusted CAs (see Section 10.3.4 on page 372).
- **WAS to WAS.** WASs communicate to other WASs by using IIOP. In a secure environment, all these communications are over SSL. The digital-key databases can be configured to reflect the trust policy by using the

tools provided with J2EE. In general, it is also possible to configure the strength of encryption enforced in such a secure connection.

- **Application client to WAS.** Similar to a Web browser's making a request to a Web resource, Java clients can use the EJB programming model to invoke methods on an enterprise bean by connecting to the WAS that hosts the enterprise bean. In a secure environment, the communication from the client to the protected resources should be protected by the SSL protocol. At that point, an application client can securely communicate with a WAS over SSL. This is achieved by configuring the WAS with a list of trusted application clients. Alternatively, the WAS can be configured to accept a list of trusted CAs, so that a trusted communication link can be established with only the application clients whose digital certificates have been authenticated by one of the trusted CAs.
- **WAS to LDAP directory server.** A WAS may need to connect to external systems. For example, a WAS may be configured to use an LDAP directory server as a user registry. In this case, the WAS will make calls against the LDAP directory. The protocol of communication between these two entities is LDAP. User ID and password pairs are verified by performing LDAP bind operations against the LDAP directory. In a scenario like this, these values will flow over the wire unencrypted unless the communication is protected by the SSL protocol. For this reason, an LDAP directory server should be configured to require that all connections be over SSL. The set of digital certificates trusted by the directory server can be imported into the WAS's digital-key storage facility so that the WAS can successfully establish an SSL connection with the LDAP directory server.

3.12 Secure E-Business Request Flow

We are now ready to revisit the simple e-business request flow presented in Section 3.6 on page 63 with the security semantics added to the flow. The secure e-business request flow takes the security model and security technologies available in the J2EE environment into account and depicts the use of those components.

Figure 3.7 enhances the diagram depicted in Figure 3.3 on page 63 and shows how the J2EE security technologies presented in this chapter can play a role in a secure e-business request flow.

Let us consider the request flow discussed in Section 3.6 on page 63 and understand how security technologies play a role when that request is secured. A user, say, Bob, invokes URL `http://samples.com/servlet/account`, which is handled by the Web container that hosts the account servlet. Based on the security

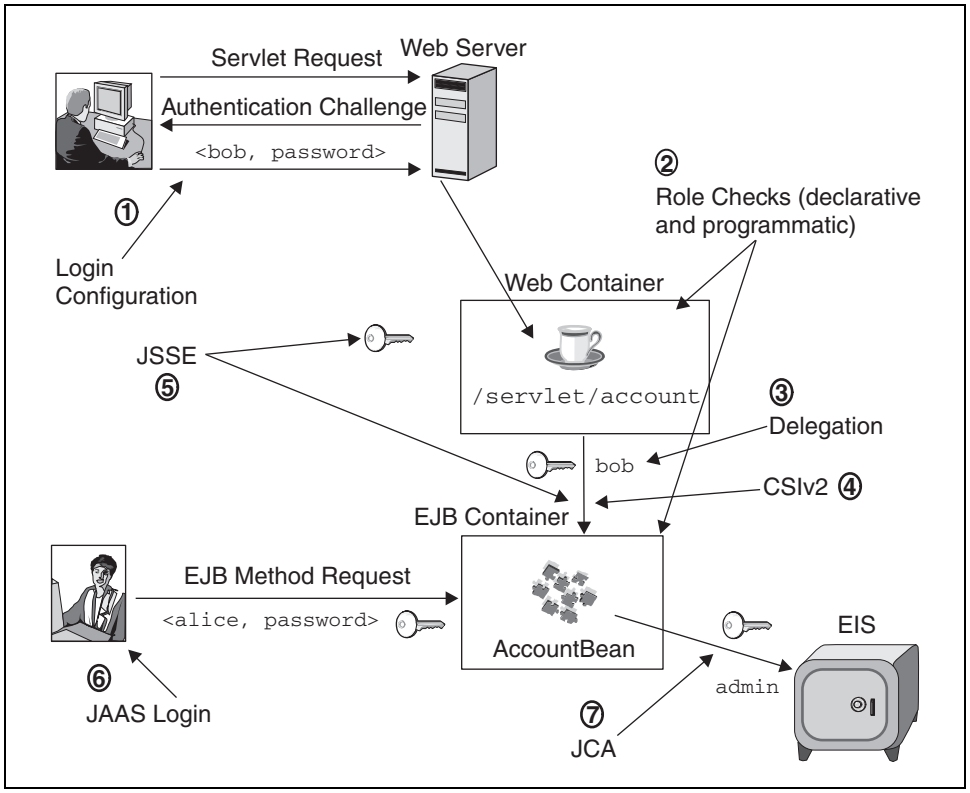

Figure 3.7. Secure E-Business Request Flow

constraints defined, the URI `/servlet/account` is a protected resource. The login configuration associated with the Web application is set to `BASIC`. Therefore, when the request arrives, the Web container issues an HTTP 401 response back to the browser. Bob resubmits his request by providing his user ID (`bob`) and his password (see step 1 in Figure 3.7). The Web container validates the user ID and password by authenticating the pair against the underlying user registry.

After the credentials are validated, the Web container performs an authorization check. The URI is protected in a way to grant access only to the Teller role. The container checks whether user Bob is granted the role (see step 2 in Figure 3.7). As Bob has been granted the Teller role, he is allowed to access the URI.

The servlet invokes the AccountBean enterprise bean, and the request is dispatched to the enterprise bean. The delegation policy on the servlet is not set to `run-as`, which means that the downstream requests will be performed by using the caller's identity. In this case, the caller is Bob, and therefore the identity under

which the enterprise bean is called consists of Bob's credentials, as depicted in step 3 in Figure 3.7.

The Web container dispatches the request to the EJB container when the account servlet makes a call to the AccountBean enterprise bean. The request is sent over IIOP. Given that this request is sent over a secure environment, the CSIv2 protocol is in effect (see Section 3.3 on page 61). The servers hosting the Web container and the EJB container establish a secure association using the CSIv2 technology as depicted in step 4 in Figure 3.7. Based on a successful establishment of the connection and the validity of Bob's credentials, the received identity at the EJB container is Bob's.

The connections between the Web server and the Web container, and the Web container and the EJB container are over SSL. This ensures confidentiality and integrity of the messages sent over the wire. It is essential for all communications in an enterprise to be over a secure connection. JSSE is used to establish the SSL connection as depicted in step 5 in Figure 3.7.

The AccountBean enterprise bean can also be invoked from a Java client. In this case, a user, say, Alice, sends the request directly over IIOP. The request is made from a J2EE client. In this case, the J2EE security technology used for authentication and authorization is JAAS, which is discussed in detail in Chapter 9 on page 289. The client is configured with a `javax.security.auth.callback.CallbackHandler` and performs a JAAS login against the server hosting the EJB container (step 6 in Figure 3.7). Alice needs to provide a valid user ID and password pair. Then, she can perform a method invocation on the AccountBean enterprise bean. J2EE supports a Java client to submit a user ID and password pair over an IIOP message. Using the CSIv2 protocol, the user ID and password pair is in a Generic Security Services Username Password (GSSUP) token within the CSIv2 `ESTABLISH_CONTEXT` message.[4] In order for this communication to be protected, transport-level SSL is recommended.

The AccountBean enterprise bean is protected by J2EE declarative security. A method permission definition declares the enterprise bean's methods to be accessible only by those who are granted the Teller role. Both users, Bob and Alice, are granted the Teller role. Therefore, the requests that come through the servlet and the one directly submitted are allowed to be invoked. The AccountBean enterprise bean is succesfully accessed after the authorization check.

The AccountBean enterprise bean needs to access the data source in order to retrieve the account information. The bean uses a JCA connection manager to obtain a connection to an EIS. When the connection is obtained, based on the connector security configuration, an identity is associated with the connection. The

4. Note that there is no standard declarative way to specify this type of authentication. The details are specific to the J2EE container.

`admin` identity is used to obtain the connection and access the back-end EIS. This access is enforced by the JCA capability and ensures that the EIS is accessed securely, based on the security configuration, as depicted in step 7 in Figure 3.7.

As illustrated in the simple request flow enhanced with security characteristics, the J2EE security model provides the capability and infrastructure to perform secure transactions in an e-business environment.

CHAPTER 10

The Theory of Cryptography

ONE of the essential ingredients of e-business and enterprise computing is cryptography. Cryptography plays a critical role in J2SE and J2EE security, as Part IV of this book demonstrates.

This chapter explains the theory of cryptography that will be used in Chapters 11, 12, and 13. First, this chapter describes secret-key cryptographic systems, as they are at the heart of most cryptographic services, including bulk-data encryption, owing to their inherent performance advantage. Next is an overview of public-key encryption, which is essential for conducting e-business, particularly across public networks, because of the relative ease of distributing cryptographic keys. In Chapter 11, secret- and public-key cryptography services are described in the context of the standard Java APIs: the Java Cryptography Architecture and the Java Cryptography Extension.

For readers who may feel intimidated by the mathematical jargon associated with cryptography, we have tried to explain the mathematics associated with cryptography in a clear and simple way. Our intent is to demystify the concepts and terms surrounding cryptography.

10.1 The Purpose of Cryptography

The purpose of cryptography is to protect data transmitted in the likely presence of an adversary. As shown in Figure 10.1, a *cryptographic transformation of data* is a procedure by which plaintext data is disguised, or *encrypted*, resulting in an altered text, called *ciphertext*, that does not reveal the original input. The ciphertext can be reverse-transformed by a designated recipient so that the original plaintext can be recovered.

Cryptography plays an essential role in

- **Authentication.** This process to prove the identity of an entity can be based on *something you know*, such as a password; *something you have*,

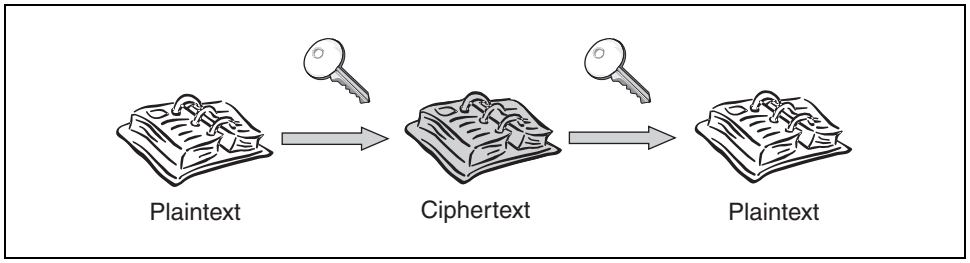

Figure 10.1. The Process of Encryption and Decryption

such as an encryption key or card; *something you are*, such as biometric measurements, including retinal scans or voice recognition; or any combination of these.

- **Data confidentiality.** With this property, information is not made available or disclosed to unauthorized individuals, entities, or processes. When two or more parties are involved in a communication, the purpose of confidentiality is to guarantee that only those parties can understand the data exchanged. Confidentiality is enforced by encryption.
- **Data integrity.** This property refers to data that has not been changed, destroyed, or lost in an unauthorized or accidental manner. The need for data integrity is especially evident if data is transmitted across a nonsecure network, such as the Internet, where a man-in-the-middle attack can easily be mounted. Integrity is enforced by mathematical functions applied to the message being transmitted.
- **Nonrepudiation.** *Repudiation* is the denial by one of the entities involved in a communication of having participated in all or part of the communication. *Nonrepudiation* is protection against repudiation and can be of two types.
  - *Nonrepudiation with proof of origin* provides the recipient of data with evidence that proves the origin of the data and thus protects the recipient against an attempt by the originator to falsely deny sending the data. Its purpose is to prove that a particular transaction took place, by establishing accountability of information about a particular event or action to its originating entity.
  - *Nonrepudiation with proof of receipt* provides the originator of data with evidence proving that data was received as addressed and thus protects the originator against an attempt by the recipient to falsely deny receiving the data.

In most cases, the term *nonrepudiation* is used as a synonym of *nonrepudiation with proof of origin*. Like integrity, nonrepudiation is based on mathematical functions applied to the data being generated during the transaction.

Keeping secrets is a long-standing tradition in politics, the military, and commerce. The invention of public-key cryptography in the 1970s has enabled electronic commerce to blossom in systems based on public networks, such as the Internet.

There are two primary approaches to cryptography (see Figure 10.2). In secret-key cryptography, the key used to decrypt the ciphertext is the same as the key that was used to encrypt the original plaintext. In public-key cryptography, the key used to decrypt the ciphertext is different from but related to the key that was used to encrypt the original plaintext.

Each approach has its strengths and weaknesses. Many of the cryptographic services enterprise applications need use both approaches. However, most application developers will not be aware of the underlying machinery that is deployed. For example, most users of SSL-enabled Web browsers are not aware that both public- and secret-key cryptography are essential parts of the SSL protocol.

Naively, we can think about cryptography primarily as a means for keeping and exchanging secrets. This is the confidentiality property that cryptography affords us. However, other essential cryptographic services are provided. When exchanging a message, whether encrypted or not, we often want to verify its integrity. Someone, particularly in public networks, may have modified the message. Data-integrity verification includes authenticating the origin of the message. Was the message from the source that we think sent the message? Once we accept that the message is from an authenticated entity and was not modified after being created, we also want to consider whether the sender can repudiate—deny sending—the message by claiming that someone stole the cryptographic key used to

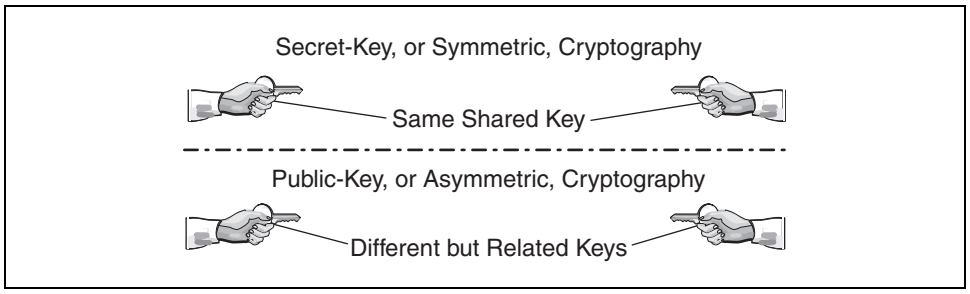

Figure 10.2. Secret-Key and Public-Key Encryption

authenticate the message. Therefore, nonrepudiation is an essential feature of cryptographic systems e-businesses use.

10.2 Secret-Key Cryptography

In *secret-key cryptography*, a sequence of bits, called the *secret key*, is used as an input to a mathematical function to encrypt a plaintext message; the same key is also used to decrypt the resulting ciphertext message and obtain the original plaintext (see Figure 10.3). As the same key is used to both encrypt and decrypt data, a secret key is also called a *symmetric key*.

10.2.1 Algorithms and Techniques

In this section, we examine the most common cryptographic algorithms that are based on the use of a secret key.

10.2.1.1 Substitutions and Transpositions

Some very early cryptographic algorithms manipulated the original plaintext, character by character, using the techniques of substitution and transposition.

- A *substitution*, or *permutation*, replaces a character of the input stream by a character from the alphabet set of the target ciphertext.
- A *transposition* replaces a character from the original plaintext by another character of that same plaintext. This results in shuffling yet still preserving the characters of the original plaintext.

An example of a substitution is the famous Caesar Cipher, which is said to have been used by Julius Caesar to communicate with his army. The Caesar

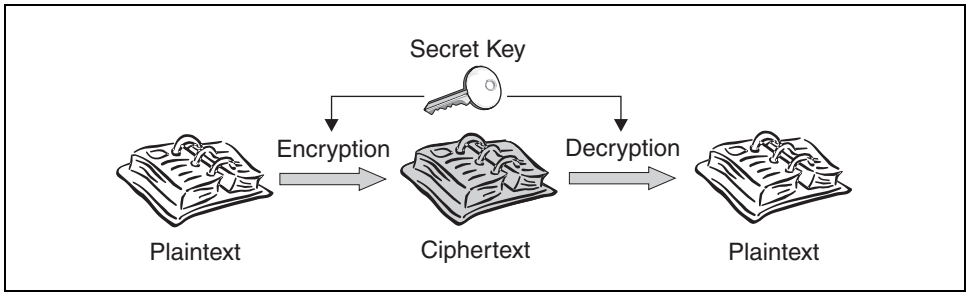

Figure 10.3. Secret-Key Encryption and Decryption

Cipher replaces each character of the input text by the third character to its right in the alphabet set. In Figure 10.4, the value 3 is added to the position of the input character; then modulo 26 is taken to yield the replacement character. If we assign numerical equivalents of 0–25 to the 26-letter alphabet A–Z, the transformation sends each plain character with position P onto the character with position $f(P) := P + 3 \pmod{26}$.

A *transposition cipher* consists of breaking the original plaintext into separate blocks first. A deterministic procedure is then applied to shuffle characters across different blocks. For example, a transposition can split the secret message `"PHONE HOME"` into the two separate blocks `"PHONE"` and `" HOME"`. Then, characters are cyclically shuffled across the two blocks to result in the ciphertext of `"POMHE HOEN"`. Another example of a simple transposition cipher consists of writing the plaintext along a two-dimensional matrix of fixed rows and columns and then simply transposing the matrix, as shown in Figure 10.5.

Original	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Substitution	D E F G H I J K L M N O P Q R S T U V W X Y Z A B C

Figure 10.4. The Caesar Cipher

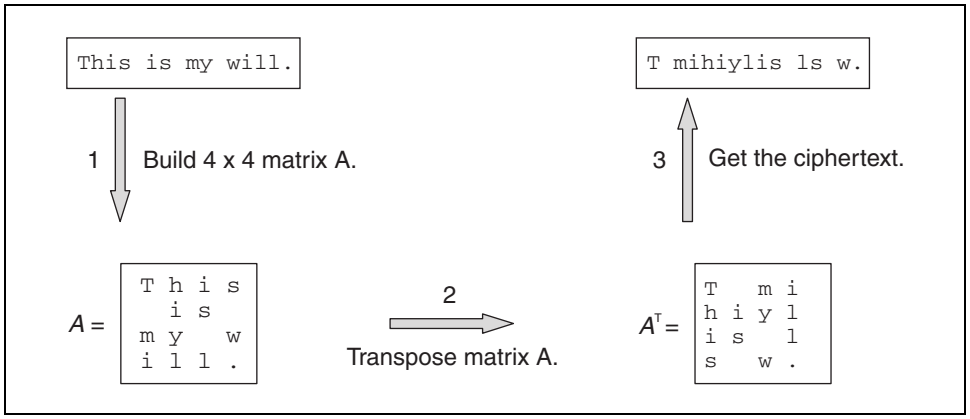

Figure 10.5. Transposition Matrix

Generally, transposition ciphers are easy to break. However, composing them by setting the result of one transposition as the input of another one greatly enhances the ciphering against attacks.

With the age of computers, early modern cryptography carried on these same concepts, using the various elementary transformations that we have listed. The primary difference is that these transformations now apply at the bit level of the binary representation of data instead of characters only.

10.2.1.2 The **XOR** *Operation*

A common transformation is the exclusive OR (XOR) operation, denoted by the symbols XOR, or $\oplus$. XOR is a bitwise function that maps an element of $\{0, 1\} \times \{0, 1\}$ onto the set $\{0, 1\}$, as shown in Figure 10.6. If we interpret the second operand as a key value, the XOR operation can be thought of as a bit-level substitution based on the bit values of the key. With such an assumption, XOR sends a 0 or 1 to itself when the corresponding key bit is 0 and inverts a 0 into a 1 and a 1 into a 0 when the corresponding key bit is 1.

The last property implies that when using a fixed-key value, the XOR operation can be applied to encipher a plaintext, which can then be recovered by simply applying the XOR operation to the ciphertext with the same key value. This property has led to the proliferation of many variants of weak encryption methods that rely solely on the simple XOR operation and thus are easily breakable.

Figure 10.7 shows how to XOR blocks of some plaintext P with a fixed-length key K, leading to ciphertext P'. The figure also shows that if P' is then XORed with K, the original plaintext P is produced.

Knowing a block of plaintext and its XOR transformation directly leads to K, by way of XORing the plaintext with the corresponding ciphertext, as shown in Figure 10.8. Similarly, by knowing two ciphertext blocks P' and Q' alone, one can XOR them together to yield the XOR of the corresponding plaintext blocks P and Q, as in Figure 10.9.

XOR:{0,1} × {0,1} → {0,1}

XOR	0	1
0	0	1
1	1	0

Figure 10.6. The XOR Operation Table

P = 1 0 1 0 1 1 0 0 0 1 0 1 1 1 1 0 0 1 0 1 0 0 0 1

K = 1 1 0 0 0 1 0 1

P	1 0 1 0 1 1 0 0	0 1 0 1 1 1 1 0	0 1 0 1 0 0 0 1
XOR K	1 1 0 0 0 1 0 1	1 1 0 0 0 1 0 1	1 1 0 0 0 1 0 1
= P'	0 1 1 0 1 0 0 1	1 0 0 1 1 0 1 1	1 0 0 1 0 1 0 0

P' = 0 1 1 0 1 0 0 1 1 0 0 1 1 0 1 1 1 0 0 1 0 1 0 0

P'	0 1 1 0 1 0 0 1	1 0 0 1 1 0 1 1	1 0 0 1 0 1 0 0
XOR K	1 1 0 0 0 1 0 1	1 1 0 0 0 1 0 1	1 1 0 0 0 1 0 1
= P	1 0 1 0 1 1 0 0	0 1 0 1 1 1 1 0	0 1 0 1 0 0 0 1

Figure 10.7. XORing Plaintext Blocks with a Fixed-Length Key

P block	1 0 1 0 1 1 0 0
XOR P' block	0 1 1 0 1 0 0 1
K	1 1 0 0 0 1 0 1

Figure 10.8. How to Get the Fixed-Length Key by XORing a Plaintext Block with Its Corresponding Ciphertext Block

P block	1 0 1 0 1 1 0 0
P' block	0 1 1 0 1 0 0 1

Q block	0 1 0 1 1 1 1 0
Q' block	1 0 0 1 1 0 1 1

P block XOR Q block	1 1 1 1 0 0 1 0
P' block XOR Q' block	1 1 1 1 0 0 1 0

Figure 10.9. Ciphertext-Block XOR and Plaintext-Block XOR Equality

Therefore, examining the bit patterns of $P \oplus Q$ can easily result in recovering one of the plaintexts by knowing some information about the other. The plaintext can then be XORed with its ciphertext to yield the keystream, where the *keystream* is the key used to encipher the plaintexts.

Despite the simplicity of the XOR operation and the weakness of encryption algorithms that use it with fixed keys, there is a way to make the sole use of such basic operation result in a perfect encryption scheme. A *one-time pad* is a key of randomly generated digits that is used only once. Use of such a key yields a perfect cipher. Such a cipher is provably secure against attacks in which a code breaker has knowledge of a set of ciphertexts.

The security of the one-time pad stems from the fact that the uncertainty in attempting to guess the keystream is equal to that of directly guessing the plaintext. Note, however, that the length of the keystream for the one-time pad is equal to that of the plaintext being encrypted. Such a property makes it difficult to maintain and distribute keys, which could be very long. This difficulty has led to the development of stream ciphers whereby the key is pseudorandomly generated from a fixed secret key.

10.2.1.3 Stream Ciphers

Stream ciphers are geared for use when memory buffering is limited or when characters are individually transformed as they become available for transmission. Because stream ciphers generally transform plaintext bits independently from one another, error propagation remains limited in the event of a transmission error. For example, the XOR operation lends itself to be used as a stream cipher.

10.2.1.4 Block Ciphers

Block ciphers divide a plaintext into identically sized blocks. Generally, the blocks are of length greater than or equal to 64 bits. The same transformations are applied to each block to perform the encryption.

All the widely known secret-key block-cipher algorithms exhibit the cryptographic properties desired in a block cipher. Foremost of these is the fact that each bit of the ciphertext should depend on all key bits. Changing any key bit should result in a 50 percent chance of changing any resulting ciphertext bit. Furthermore, no statistical relationships should be inferrable between a plaintext and its corresponding ciphertext. In the reminder of this section, we present the most common secret-key block-cipher algorithms.

Feistel Ciphers. A *Feistel cipher* uses a noninvertible function *f*, obtained as a sequence of substitutions and transpositions. A Feistel cipher consists of the following basic steps:

1. A plaintext message *m* is divided into two separate blocks of equal size: the *left block*, *L*, and the *right block*, *R*.

2. The original message, m, is transformed into an intermediate message, m', in which the left block, L', is the same as R, and the right block, R', is $L \oplus f(R)$, where the symbol $\oplus$, as usual, denotes the XOR operation.

These two steps are shown in Figure 10.10. Even though f is a noninvertible function, this design permits recovering m from m' by concatenating $R' \oplus f(L') = R' \oplus f(R) = L$ with $L' = R$.

Steps 1 and 2 must be iteratively repeated a number of times for a Feistel cipher to be secure. The number of iterations depends on the strength of the function f. It is possible to prove that, even with the strongest-possible function f, the iterations must be at least three in order for the Feistel cipher to be reliable.

DES. One of the most widely recognized secret-key block ciphers is the Data Encryption Standard (DES) algorithm. DES was developed by IBM cryptographers in the early 1970s and was adopted as a U.S. government standard in 1976. DES is intended for the protection of sensitive but unclassified electronic information. Because it uses the same key for both encryption and decryption, the algorithm is referred to as a *symmetric cipher*.

DES is a block cipher in which a 64-bit input plaintext block is transformed into a corresponding 64-bit ciphertext output. DES uses a 56-bit key expressed as a 64-bit quantity in which the least relevant bit in each of the 8 bytes is used for parity checking. DES is a Feistel algorithm that iterates over the data 16 times, using a combination of permutation and substitution transformations along with standard arithmetic and logical operations, such as XOR, based on the key value.

For many years, the DES algorithm withstood attacks. Recently, as the result of increased speed of computing systems, DES has succumbed to brute-force attack on several occasions, demonstrating its vulnerability to exhaustive searching of the key space.

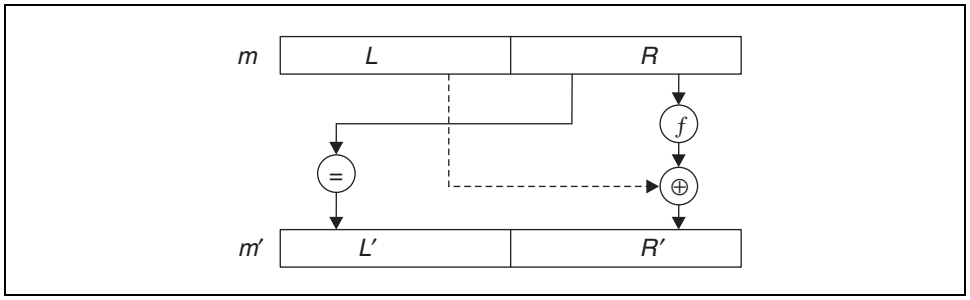

Figure 10.10. Basic Steps of a Feistel Cipher Algorithm

Triple-DES. *Triple-DES* is the DES algorithm applied three times, using either two or three keys.

- With two keys, Triple-DES proceeds by using the first key to encrypt a block of data. The second key is then used to decrypt the result of the previous encryption. Finally, the first key is once more used to encrypt the result from the second step. Formally, let us indicate the encrypting and decrypting functions based on a given key k with E_k and D_k, respectively. If k_1 and k_2 are the two Triple-DES keys and if m is the message to be encrypted, the encrypted message m' is obtained as

  $$E_{k_1}(D_{k_2}(E_{k_1}(m)))$$

  To decrypt m' and obtain the original plaintext m, it is necessary to compute

  $$D_{k_1}(E_{k_2}(D_{k_1}(m')))$$

- The three-key Triple-DES, stronger than the two-key Triple-DES, uses a separate key for each of the three steps described. With the notation that we have introduced, if k_1, k_2, and k_3 are three distinct keys, a plaintext message m is encrypted into its corresponding ciphertext message m' by

  $$E_{k_3}(D_{k_2}(E_{k_1}(m)))$$

  To decrypt m' and obtain the original plaintext m, it is then necessary to compute

  $$D_{k_1}(E_{k_2}(D_{k_3}(m')))$$

In Triple-DES, the second key is used for decryption rather than for encryption to allow Triple-DES to be compatible with DES. A system using Triple-DES can still initiate a communication with a system using DES by using only one key k. Formally, by choosing $k_1 = k_2 = k_3 = k$, the ciphertext m' corresponding to a plaintext message m is obtained from

$$E_k(D_k(E_k(m))) = E_k(m)$$

By contrast, m is obtained from m' by computing

$$D_k(E_k(D_k(m'))) = D_k(m')$$

This shows that Triple-DES with only one key reduces itself to DES.

IDEA. Although less visible than DES, the International Data Encryption Algorithm (IDEA) has been classified by some contemporary cryptographers as the most secure and reliable block algorithm. Like DES, IDEA encrypts plaintext data organized in 64-bit input blocks and for each, outputs a corresponding 64-bit ciphertext block. IDEA uses the same algorithm for encryption and decryption, with a change in the key schedule during encryption. Unlike DES, IDEA uses a 128-bit secret key and dominantly uses operations from three algebraic groups; XOR, addition modulo 2^{16}, and multiplication modulo $2^{16} + 1$. These operations are combined to make eight computationally identical rounds, followed by an output transformation resulting in the final ciphertext.

Rijndael. Recently chosen as the Advanced Encryption Standard (AES), a replacement of DES by the U.S. government, *Rijndael* is an iterated block cipher with a variable block length and a variable key length, both of which can independently be 128, 192, or 256 bits. The strong points of Rijndael are its simple and elegant design and its being efficient and fast on modern processors. Rijndael uses only simple whole-byte operations on single- and 4-byte words and requires a relatively small amount of memory for its implementation. It is suitable for implementations on a wide range of processors, including 8-bit hardware, and power- and space-restricted hardware, such as smart cards. It lends itself well to parallel processing and pipelined multiarithmetic logic unit processors.

A major feature of the Rijndael algorithm is that it presents a departure from the traditional Feistel ciphers. In such ciphers, some of the bits in the intermediate states of a cipher are transposed unchanged. The Rijndael algorithm does not adopt the Feistel structure. Instead, each round of transformations is composed of three distinct invertible subtransformations that treat each bit of the intermediate state of the cipher in a uniform and similar way.

10.2.1.5 Modes of Operation

Modes of operation are cryptographic techniques using block ciphers to encrypt messages that are longer than the size of the block. The most common modes of operation are electronic codebook (ECB) and cipher block chaining (CBC).

ECB. With the ECB mode of operation, a message is divided into blocks of equal size. Each block is then encrypted using a secret key. Figure 10.11 shows how ECB works, assuming the following.

1. The original message m is divided into n blocks $m_1, m_2, \ldots, m_n$.
2. For all $i = 1, 2, \ldots, n$, the plaintext block m_i is encrypted into a ciphertext block c_i with a secret key k. The encryption function associated with k is indicated with E_k. In ECB mode, the block-cipher algorithm typically used for encryption is DES.

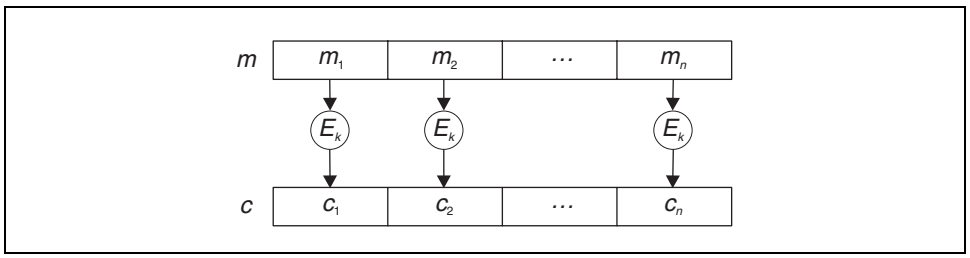

Figure 10.11. ECB Mode

3. The ciphertext blocks $c_1, c_2, \ldots, c_n$ are concatenated to form the ciphertext c corresponding to the message m.

ECB presents some limitations because each ciphertext block depends on one plaintext block only, not on the entire message.

CBC. Given a secret key k, the CBC mode of operation works as follows (see Figure 10.12).

1. The original message m is divided into n blocks $m_1, m_2, \ldots, m_n$.
2. A randomly chosen block of data is selected as the *initial vector* v. This initial vector must be known to the receiver as well. Therefore, a possibility is for both the sender and the receiver to be able to generate v independently as a function of the key k.
3. The first ciphertext block, c_1, is obtained by XORing v with m_1 and encrypting the result of the XOR operation with the secret key k. In other words,

$$c_1 = E_k(v \oplus m_1)$$

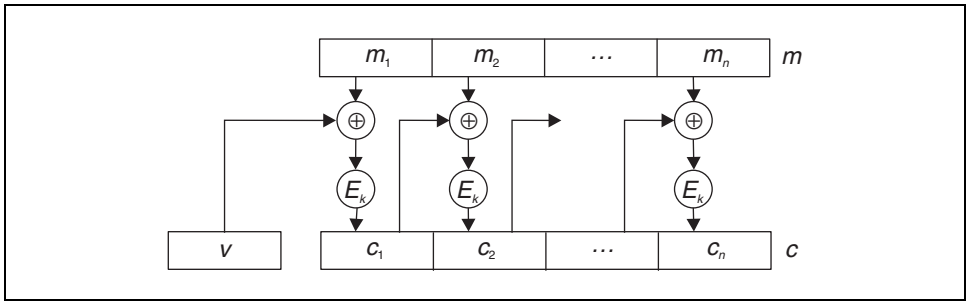

Figure 10.12. CBC Mode

where E_k is the encrypting function associated with the key k.

4. For all $i = 2, \ldots, n$, the ciphertext block c_i is obtained by XORing the plaintext block m_i with the ciphertext block c_{i-1} and encrypting the result of the XOR operation with the secret key k. In other words,

$$c_i = E_k(c_{i-1} \oplus m_i)$$

5. The ciphertext blocks $c_1, c_2, \ldots, c_n$ are concatenated to form the ciphertext c of the message m.

One of the key characteristics of CBC is that it uses a chaining mechanism that causes the decryption of a block of ciphertext to depend on all the preceding ciphertext blocks.

10.2.2 Secret-Key Security Attributes

This section examines the security implications of using secret-key cryptography.

10.2.2.1 Key Space

The strength of modern secret-key encryption methods no longer rests in the secrecy of the algorithm being used but rather in the secrecy of the encryption key. Breaking such cryptographic systems, therefore, can be achieved using a *brute-force attack*, the process of exhaustive searches over the *key space*. The latter is the set of all possible key values that a particular enciphering method can take.

For example, a generalization of the Caesar Cipher is an arbitrary permutation over the English alphabet. This results in 26! (factorial) possible keys corresponding to each of the permutations. Further constraining the permutation method to one that simply maps each letter in the alphabet to one at a fixed number of positions to its right (with a wraparound) and by enciphering each letter at a time (block length = 1), the key space narrows down to the much smaller set of the first 26 integers, $\{1, 2, \ldots, 26\}$. It should be noted, however, that the level of a secret-key encryption algorithm's security is not necessarily proportional to the size of the key space. For example, even though 26! is a very large number, it is possible to break the generalization of the Caesar Cipher by means of statistical analysis.

Most common secret-key cryptographic systems use unique, randomly generated, fixed-size keys. These systems can certainly be exposed to the exhaustive search of the key space. A necessary, although not sufficient, condition for any such cryptographic systems to be secure is that the key space be large enough to preclude exhaustive search attacks using computing power available today and for the foreseeable future. As ironic as it may sound, efficiency of enciphering methods will aid in the exhaustive brute-force search attacks.

10.2.2.2 Confidentiality

Using a secret-key algorithm to encipher the plaintext form of some data content allows only entities with the correct secret key to decrypt and hence retrieve the original form of the disguised data. Reliability of the confidentiality service in this case depends on the strength of the encryption algorithm and, perhaps more important, the length of the key used. The long lifetime of a secret key also might help diminish assurance in such a confidentiality service. Increasing the frequency with which a key is used increases the likelihood that an exhaustive key-search attack will succeed. Most modern systems make use of secret keys that remain valid for only the lifetime of a particular communication session.

10.2.2.3 Nonrepudiation

Secret-key cryptography alone is not sufficient to prevent the denial of an action that has taken place, such as the initiation of an electronic transaction. Although one can apply data privacy in such a scenario, the fundamental flaw of a nonrepudiation scheme based on secret-key cryptography alone is inherent in the fact that the secret key is dispensed to more than one party.

10.2.2.4 Data Integrity and Data-Origin Authentication

At a much lesser cost than encrypting the entirety of a plaintext, data integrity and data-origin authentication can be afforded by a secret cryptographic scheme using a message authentication code (MAC) function. The basic idea is to attach to each message m that is sent across a network the result $h(m)$ of a mathematical function h applied to the message m itself. If an error has occurred during the message transmission, such that the received message a is different from the message m that was originally sent, the message receiver will be able to detect the anomaly by independently computing $h(a)$ and comparing it to $h(m)$ (see Figure 10.13).

The main component of a MAC function is a hash digest function (see Figure 10.14). Hash digest functions are considered one of the fundamental primitives in modern cryptography. By definition, a *hash digest function* is a deterministic function that maps a message of arbitrary length to a string of fixed length n. Typically, n is 128 or 160 bits. The result is commonly known as a *message digest*. As the original data is often longer than its hash value, this result is sometimes also referred to as the original message's *fingerprint*.

Of course, a hash digest function is inherently *noninjective*. This simply means that multiple messages will be mapping to the same digest. In fact, the universe of the messages that can be digested is potentially unlimited, whereas the universe of all the message digests is limited by the set of the 2^n strings with n bits. However, the fundamental premise is that, depending on the strength of the hashing algorithm, the hash value becomes a more compact representation of the original data. This means that, although virtually possible, it should be computa-

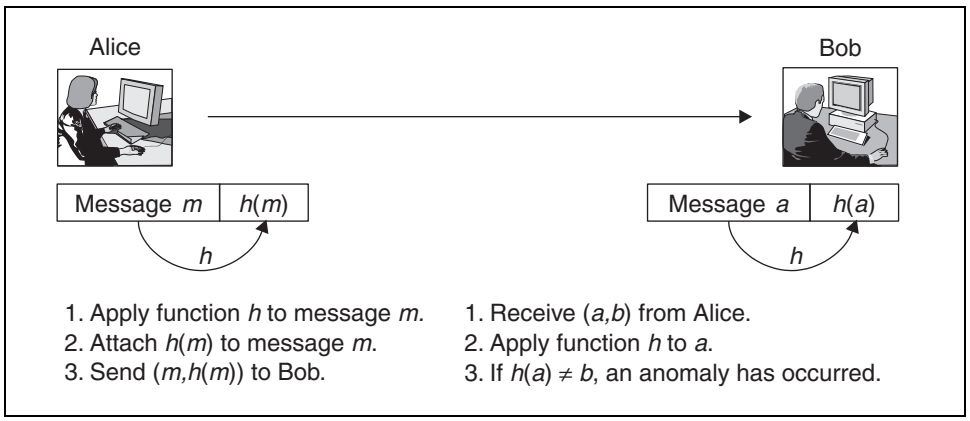

Figure 10.13. Data-Integrity Verification: Basic Scenario

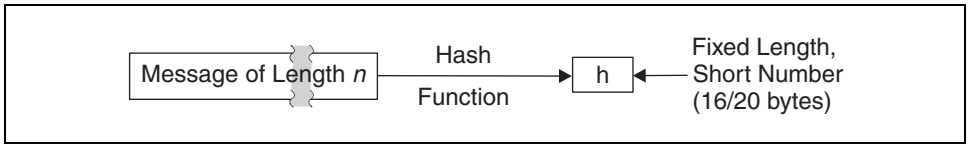

Figure 10.14. Producing a Message Digest with a Hash Function

tionally infeasible to produce two messages having the same message digest or to produce any message having a given, prespecified target message digest.

Message Digest V5 (MD5) and Secure Hash Algorithm V1 (SHA-1) are the most widely used cryptographic hash functions. MD5 yields a 128-bit (16-byte) hash value, whereas SHA-1 results in a 160-bit (20-byte) digest. SHA-1 appears to be a cryptographically stronger function. On the other hand, MD5 edges SHA-1 in computational performance and thus has become the de facto standard.

Hash functions alone cannot guarantee data integrity, because they fail in guaranteeing *data-origin authentication*, defined as the ability to authenticate the originator of a message (see Figure 10.15). The problem with digest functions is that they are publicly available. If a message m is intercepted by an adversary after being transmitted by Alice, the adversary can change m into a different message, m', compute $h(m')$, and send Bob the pair $(m', h(m'))$. By simply applying the function h to the received message m', Bob has no means of detecting that an adversary has replaced m with m'.

Data-origin authentication is inherently supported by secret-key cryptography, provided that the key is shared by two entities only. When three or more parties share the same key, however, origin authenticity can no longer be provided by

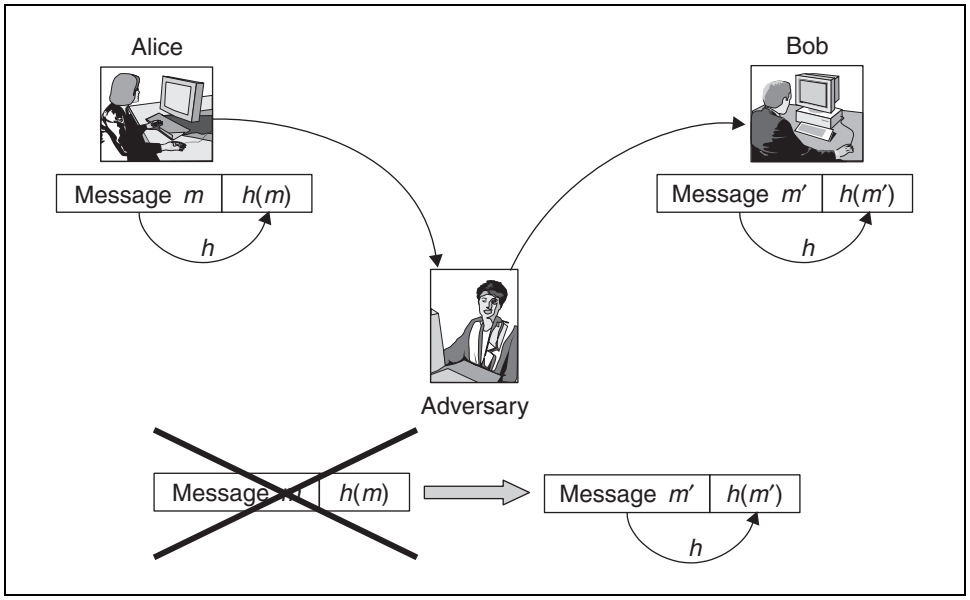

Figure 10.15. Data-Integrity Verification in the Presence of an Adversary

secret-key cryptography alone. Various secret-key-based authentication protocols have been developed to address this limitation. Public-key cryptography, described in Section 10.3 on page 359, provides a simpler and more elegant solution to this problem.

In contrast to using a pure and simple hash function to digest a message, a MAC function combines a hash digest fuction with secret-key encryption and yields a value that can be verified only by an entity having knowledge of the secret key. This way, a MAC function takes care of the problem described in Figure 10.15 and enables both data integrity and data-origin authentication.

Another simple solution to achieve data integrity and data-origin authentication is to apply a regular hash function h, such as SHA-1 or MD5, but rather than hashing the message m alone, the message is first concatenated with the key k, and then the result of the concatenation is hashed. In other words, the sender attaches to the message m the tag $h(k, m)$. This solution, however, exposes some theoretical weaknesses. A more reliable solution consists of attaching the tag $h(k, h(k, m))$.

A MAC can even be computed by using solely a secret-key block-cipher algorithm. For example, the last ciphertext block, encrypted in CBC mode, yields the final MAC value. This is a good choice for a MAC because one of the key characteristics of CBC is that it uses a chaining mechanism that causes the decryption of a block of ciphertext to depend on all the preceding ciphertext blocks. Therefore,

the MAC so defined is a *compact* representation of the *entire* message that can be computed *only* by an entity having knowledge of the secret key. Known instances of this procedure use DES and Triple-DES, resulting in DES-MAC and Triple-DES-MAC, respectively. A MAC mechanism that uses a cryptographic hash function is also referred to as HMAC. HMAC is specified in RFC 2104.[1]

10.3 Public-Key Cryptography

Public-key cryptography emerged in the mid-1970s with the work published by Whitfield Diffie and Martin Hellman.[2] The concept is simple and elegant yet has had a huge impact on the science of cryptography and its application. *Public-key cryptography* is based on the notion that encryption keys are related pairs, private and public. The *private key* remains concealed by the key owner; the *public key* is freely disseminated to various partners. Data encrypted using the public key can be decrypted only by using the associated private key and vice versa. Because the key used to encrypt plaintext is different from the key used to decrypt the corresponding ciphertext, public-key cryptography is also known as *asymmetric cryptography.*

The premise behind public-key cryptography is that it should be computationally infeasible to obtain the private key by simply knowing the public key. Toward achieving this premise, modern public-key cryptography derives from sophisticated mathematical foundations based on the one-way functions existing in the abstractions of number theory.

A *one-way function* is an invertible function that is easy to compute but computationally difficult to invert. A *one-way trapdoor function* is a one-way function that can be easily inverted only if one knows a secret piece of information, known as the *trapdoor.* Encryption is the easy one-way trapdoor function; its inverse, decryption, is the difficult direction. Only with knowledge of the trapdoor—the private key—is decryption as easy as encryption. Two of these currently known one-way functions, factoring large numbers and computing discrete logarithms, form the basis of modern public-key cryptography. Factoring large numbers is a one-way trapdoor function, whereas computing discrete logarithms is a one-way function with no trapdoors.

1. See `http://www.ietf.org/rfc/rfc2104.txt`.
2. W. Diffie and M. E. Hellman. "New Directions in Cryptography," *IEEE Transactions on Information Theory* 22, 6, (1976): 644–654.

10.3.1 Algorithms and Techniques

This section examines the most common cryptographic algorithms that are based on the use of a public- and private-key pair.

10.3.1.1 RSA

The most famous of the well-known trapdoor one-way functions is based on the ease of multiplying two large prime numbers; the reverse process, factoring a very large number, is far more complex. This consideration is at the basis of Rivest-Shamir-Adleman (RSA), certainly the most widely used public-key encryption algorithm.

Basic RSA Concepts. A *prime number*, by definition, is an integer that has no positive divisors other than 1 and itself. A nonprime integer is called *composite*. Two integers $a \geq 1$ and $b \geq 2$ are said to be *relatively prime* if their greatest common divisor GCD(a, b) is 1. The number of elements in the set

$$\{a \in \mathbf{Z} : 1 \leq a < b, \text{GCD}(a, b) = 1\}$$

where $\mathbf{Z}$ is the set of all integers, is often denoted by $\phi(b)$. The function ϕ is called the *Euler phi-function.*

Every integer $b \geq 2$ can be *factored* as a product of powers of primes in a unique way. For example, $60 = 2^2 \times 3 \times 5$. Factoring *large numbers*—numbers that expressed in binary format take 1,024 bits or more—is known to be computationally infeasible with current computing technology. Consequently, the one-way trapdoor problem is to make a very large number a public knowledge and secretly maintain its prime factors. With this in mind, we can now summarize the widely adopted RSA public-key algorithm.

How the RSA Algorithm Works. In simple terms, the *RSA algorithm* centers on three integer numbers: the *public exponent*, e; the *private exponent*, d; and the *modulus*, n. The modulus is obtained as the product of two distinct, randomly picked, very large primes, p and q. A well-known result from number theory implies that $\phi(n) = (p - 1)(q - 1)$. The two numbers e and d are characterized by the fact that they are greater than 1 and smaller than $\phi(n)$. In addition, e must be relatively prime with $\phi(n)$, and it must also be $de = 1 \pmod{\phi(n)}$, which means that d and e are the multiplicative inverse of the other modulo $\phi(n)$. The pair (e, n) is the RSA public key, whereas the pair (d, n) is the RSA private key.

A block of plaintext P whose numerical equivalent is less than the modulus is converted into a ciphertext block by the formula $P^e \pmod{\text{n}}$. Conversely, a ciphertext block C is converted back to its corresponding plaintext representation by the formula $C^d \pmod{\text{n}}$. These two formulas are the inverse of the other. Therefore,

whatever is encrypted with the public key can be decrypted only with the corresponding private key; conversely, whatever is encrypted with the private key can be decrypted only with the corresponding public key.

To better understand how RSA works, let us consider an example involving small numbers. We randomly pick two prime numbers, $p = 7$ and $q = 11$. This implies that $n = p \times q = 77$ and $\phi(n) = (p - 1)(q - 1) = 60$. A valid choice for the public exponent is $e = 13$. By solving the equation $13d = 1 \pmod{60}$, we get $d = 37$. Therefore, the RSA public key in this case is the pair (13, 77), and the corresponding RSA private key is the pair (37, 77). Let us now consider the plaintext message $P = 9$. By encrypting it with the RSA public key, we obtain the ciphertext message $C = 9^{13} \pmod{77} = 58$. To decrypt this message, we have to apply the RSA private key and compute $58^{37} \pmod{77} = 9$, which yields the original plaintext *P*.

To encrypt or decrypt a message, the RSA algorithm uniquely represents a block of data in either a plaintext or ciphertext form as a very large number, which is then raised to a large power. Note here that the length of the block is appropriately sized so that the number representing the block is less than the modulus. Computing such exponentiations would be very time consuming were it not for an eloquent property that the operation of exponentiation in modular arithmetic exhibits. This property is known as the *modular exponentiation by the repeated squaring method.*

Note that the one-way trapdoor function discussed in this section requires deciding on whether a randomly picked very large integer is prime. Primality testing, however, is a much easier task than factorization. Several methods have been devised to determine the primality of an odd number p, the most trivial of which is to run through the odd numbers starting with 3 and determine whether any of such numbers divides p. The process should terminate when the square root of p, $\sqrt{p}$, is reached, because if p is not a prime, the smallest of its nontrivial factors must be less than or equal to $\sqrt{p}$. Owing to the time complexity that it requires, in practice this procedure is stopped much earlier before reaching $\sqrt{p}$ and is used as a first step in a series of more complicated, but faster, primality test methods.

Security Considerations. Breaking the RSA algorithm is conjectured to be equivalent to factoring the product of two large prime numbers. The reason is that one has to extract the modulus n from the public-key value and proceed to factor it as the product of the two primes p and q. Knowing p and q, it would be easy to compute $\phi(n) = (p - 1)(q - 1)$, and the private key (d, n) could then be obtained by solving the equation $de = 1 \pmod{\phi(n)}$ for the unknown d. With the complexity of the fastest known factoring algorithm being in the order of $|n|$, where $|n|$ is the total number of the binary bits in the modulus n, this roughly means that, for example, every additional 10 bits make the modulus ten times more difficult to factor. Given

the state of factoring numbers, it is believed that keys with 2,048 bits are secure into the future. The fastest known factoring algorithm to date is the *number field sieve*.

10.3.1.2 Diffie-Hellman

The Diffie-Hellman (DH) key-agreement algorithm is an elegant procedure for use by two entities establishing a secret cryptographic key over a public network without the risk of exposing or physically exchanging it. Indeed, DH presents a critical solution to the secret-key distribution problem. The security of the algorithm relates to the one-way function found in the discrete logarithm problem.

Basic DH Concepts. Let q be a prime number. An integer α is called a *primitive root*, or *base generator* of q, if the numbers $\alpha \pmod q$, $\alpha^2 \pmod q$, ... , $\alpha^{q-1} \pmod q$ are distinct and consist of the integers from 1 to $q - 1$ in some permutations. For any integer y and a primitive root α of the prime number q, one can find a unique integer exponent x such that $y = \alpha^x \pmod q$. The exponent x is referred to as the *discrete logarithm* of y for the base α modulo q. This is a one-way function. In fact, computing y from x using this function is easy; for q about 1,000 bits long, this would take only a few thousand multiplications. However, the inverse function, $x = \log_\alpha y \pmod q$, which yields x from y, is computationally infeasible, as far as anyone knows; Diffie proved that with q still about 1,000 bits long and the best known algorithm, the discrete logarithm would take approximately 10^{30} operations.

How the DH Algorithm Works. The mathematics encompassed in the DH key-agreement algorithm is fairly simple. Let q and α be as explained previously. These two numbers are publicly available. Suppose that Alice and Bob want to agree on a secret key. Alice generates as her private key a secret random number x_A such that $1 \leq x_A < q$ and publishes the corresponding public key

$$y_A := \alpha^{x_A} \pmod q$$

Similarly, Bob generates as his private key a secret random number x_B such that $1 \leq x_B < q$ and publishes the corresponding public key

$$y_B := \alpha^{x_B} \pmod q$$

The secret key for Alice and Bob is

$$K_{AB} := \alpha^{x_A x_B} \pmod q$$

Alice can obtain this key by getting y_B from a public directory and then computing

$$y_B^{x_A} \pmod{q} = \alpha^{x_B x_A} \pmod{q} = \alpha^{x_A x_B} \pmod{q} = K_{AB}$$

Bob computes the same secret key in a similar way.

One problem in the algorithm that we have just described consists of finding a primitive root α of a given prime number q. The definition of primitive root does not help from a computational point of view, because it requires computing $q - 1$ powers in the worst case for every attempt to find a primitive root. However, a known algebraic theorem proves that an integer α is a primitive root of *9* if $\alpha^i \neq 1$ for any integer $i \in \{1, \ldots, q-1\}$ such that i is a divisor of $q - 1$. Therefore, the problem is reduced to factoring $q - 1$ and testing that $\alpha^i \neq 1$, where this time i varies only in the set of the divisors of $q - 1$. Unfortunately, as we discussed in Section 10.3.1.1 on page 360, factoring a large number is computationally infeasible too. In fact, this is exactly the one-way trapdoor function on which the security of the RSA algorithm relies. However, a solution to this problem for the DH algorithm consists of generating $q - 1$ before generating q itself. In other words, it is possible to generate $q - 1$ as the product of known primes—in which case, the factorization of $q - 1$ is known in advance—and subsequently test q for primality. As discussed in Section 10.3.1.1 on page 360, primality testing is a much easier task than factorization. An advantage of this algorithm is that its security does not depend on the secrecy of q and α. Once a pair of integers (q, α) has been found that satisfies the requirements described previously, the same pair can be published—in cryptography books, for example—and reused by algorithm implementors.

Security Considerations. With the algorithm described, Alice and Bob do not have to physically exchange keys over unsecure networks, because they can compute the same secret key independently of each other. An attacker would have to compute K_{AB} from the only public information available, y_A and y_B. No way to do this is known other than computing the discrete logarithm of y_A and y_B to find x_A and x_B, an operation that, as we said, is conjectured to be computationally infeasible even with the fastest known algorithm.

In order for Bob and Alice to be able to compute the same secret key independently of each other, they have to know each other's public keys. A general security problem that arises at this point is how to ascertain that the public key of an entity belongs to that entity. The DH algorithm does not offer a direct solution to this problem. However, we will see how to solve this problem in Section 10.3.4 on page 372.

10.3.1.3 Elliptic Curve

Recently, elliptic curves over finite fields have been proposed as another source of one-way trapdoor functions for use with existing public-key cryptographic systems.

Basic Elliptic-Curve Concepts. An *elliptic curve* in the plane x, y is the union of the singleton $\{\mathcal{O}\}$ with the set of the points (x, y) of the plane satisfying an equation of the form

$$y^2 + axy + by = x^3 + cx^2 + dx + e$$

where a, b, c, d, and e are real numbers, and x and y take on values in the real numbers. The element $\mathcal{O}$ is called *point at infinity.* For our purpose, it is sufficient to consider equations of the form

$$y^2 = x^3 + ax + b$$

Figure 10.16 shows the elliptic curve with equation $y^2 = x^3 - x$.

A form of *addition* can be defined over the set of points of an elliptic curve by imposing that if any three points on an elliptic curve lie on a straight line, their sum is $\mathcal{O}$. The operation of addition for an elliptic curve, indicated with the symbol +, is constructed on the following rules.

1. The point at infinity, $\mathcal{O}$, is the *additive identity.* This means that $\mathcal{O} = -\mathcal{O}$, and for any point P on the elliptic curve, $P + \mathcal{O} = \mathcal{O} + P = P$.
2. A vertical line meets the elliptic curve at two points with the same coordinates, say $P_1 = (x, y)$ and $P_2 = (x, -y)$. The vertical line also meets the curve at its infinity point, $\mathcal{O}$. This implies that $P_1 + P_2 + \mathcal{O} = \mathcal{O}$, and $P_1 = -P_2$. Therefore, the negative of a point is a point with the same x coordinate but negative y coordinate. This construction is illustrated in Figure 10.17.
3. If Q and R are two points with different x coordinates, draw a straight line between them and find the third point of intersection P_1. It is easily seen that P_1 exists and is unique, unless the line is tangent to the curve at either Q or R, in which case we take $P_1 = Q$ or $P_1 = R$, respectively. Because P_1, Q, and R lie on the same straight line, it must be $Q + R + P_1 = \mathcal{O}$, which implies $Q + R = -P_1$. This construction is illustrated in Figure 10.17.
4. To double a point Q, draw the tangent line in Q and find the other point of intersection S. Then $Q + Q = 2Q = -S$. This construction is illustrated in Figure 10.17.

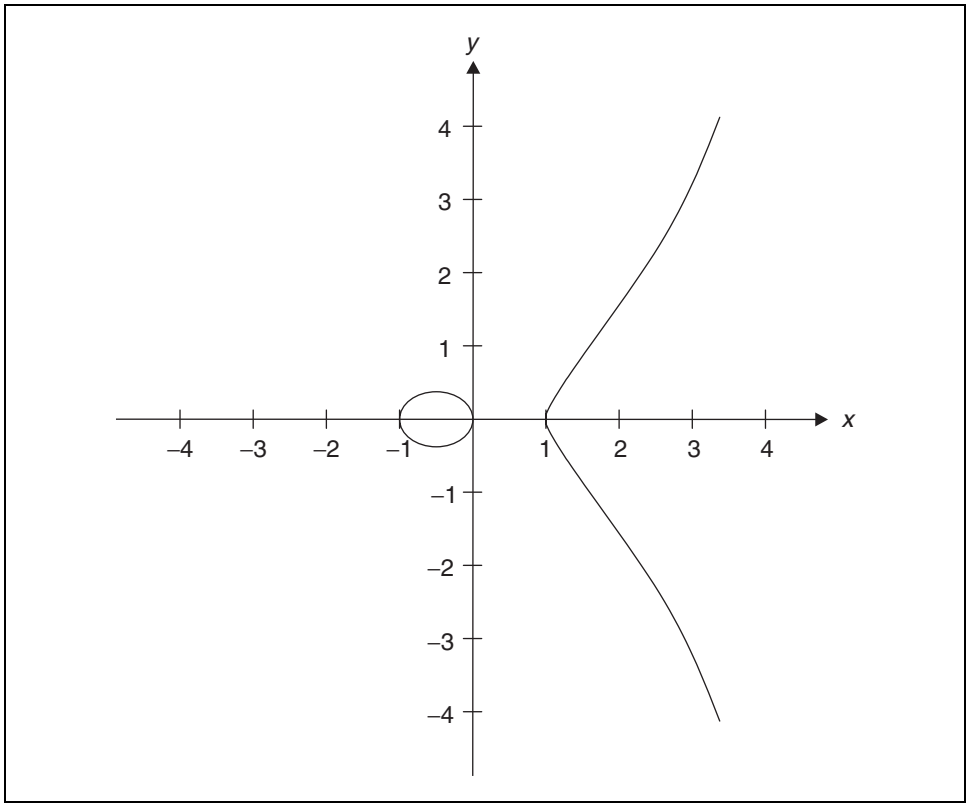

Figure 10.16. An Elliptic Curve

Figure 10.17 shows how to perform the addition operation on the elliptic curve $y^2 = x^3 - x$. It can be shown that if $4a^3 + 27b^2 \neq 0$, the operation of addition constructed on rules 1–4 has the following properties.

- **It is well defined.** Given any two points P and Q on an elliptic curve, their sum $P + Q$ is still a point on the same elliptic curve.
- **It is associative.** Given any three points P, Q, and R on an elliptic curve, $(P + Q) + R = P + (Q + R)$.
- **It is commutative.** Given any two points P and Q on an elliptic curve, $P + Q = Q + P$.
- **It possesses a unity element.** Rule 1 establishes that the unity element for the operation of addtion is the point at infinity, $\mathcal{O}$.
- **Every point on the elliptic curve has an inverse.** Given any point P on an elliptic curve, rules 1 and 2 show how to construct its inverse, $-P$.

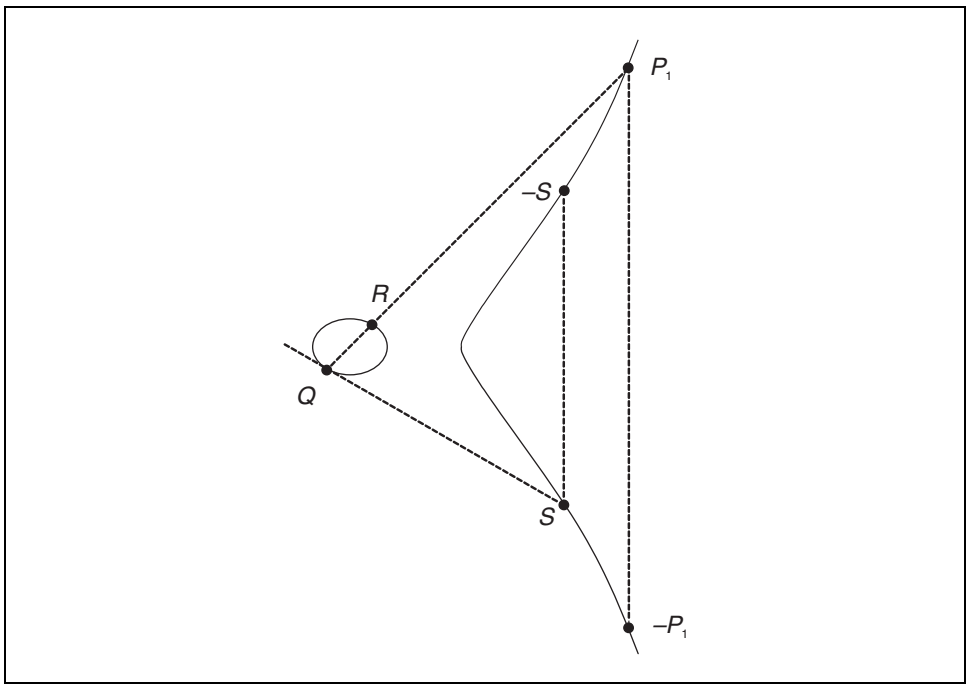

Figure 10.17. The Addition Operation on an Elliptic Curve

These properties can be summarized by saying that the set of the points of an elliptic curve, coupled with the operation of addition that we have just defined, is an *abelian group*. *Multiplication* of a point P on an elliptic curve by a positive integer k is defined as the sum of k copies of P. Thus $2P = P + P$, $3P = P + P + P$, and so on.

An elliptic curve can be defined on a finite field as well. Let $p > 3$ be a prime number. The *elliptic curve* $y^2 = x^3 + ax + b$ over $\mathbf{Z}_p$ is the set of solutions $(x, y) \in \mathbf{Z}_p \times \mathbf{Z}_p$ to the congruence $y^2 = x^3 + ax + b \pmod{p}$, where $a, b \in \mathbf{Z}_p$ are constants such that $4a^3 + 27b^2 \neq 0 \pmod{p}$, together with a special point $\mathcal{O}$, called the *point at infinity*. *Addition* of two points on an elliptic curve and *multiplication* of a point for an integer are defined in a way that is similar to elliptic curves over real numbers.

Note that the equation of an elliptic curve over the finite field $\mathbf{Z}_p$ is defined as for real numbers. The only difference is that an elliptic curve $\mathbf{Z}_p$ is not continuous. Rather, the points that belong in the curve are only the pairs of non-negative integers in the quadrant from (0, 0) to (p, p) that satisfy the equation modulo p.

Given an integer $k < p$ and the equation $Q = kP$, where P and Q are two points on an elliptic curve E over the finite field $\mathbf{Z}_p$, the one-way function here consists

of the easy operation of computing Q given k and P. The inverse problem of finding k given P and Q is similar to the discrete logarithm problem and is, in practice, intractable.

The Elliptic-Curve Algorithm. One straightforward application of the one-way function to DH is for two entities Alice and Bob to publicly agree on a point P on an elliptic curve E over a finite field $\mathbf{Z}_p$, where p is a very large prime number ($p \approx 2^{180}$). The criterion in selecting P is that the smallest integer value of n for which $nP = \mathcal{O}$ be a very large prime number. The point P is known as the *generator point*. The elliptic curve and the generator point are parameters of the cryptosystem known to all the participants.

To generate the key, the initiating entity, Alice, picks a random large integer $a < n$, computes aP over E, and sends it to the entity Bob. The integer a is Alice's private key, whereas the point aP is her public key. Bob performs a similar computation with a random large number b and sends entity Alice the result of bP. The integer b is Bob's private key, whereas the point bP is his public key. Both entities then compute the secret key $K = abP$, which is still a point over E.

Security Considerations. Given an elliptic curve E on a finite field $\mathbf{Z}_p$, where p is a very large prime number, the security of elliptic-curve cryptography depends on how difficult it is to determine the integer k given a point P on the curve and its multiple kP. The fastest known technique for taking the elliptic-curve logarithm is known as the *Pollard rho method*. With this algorithm, a considerably smaller key size can be used for elliptic-curve cryptography compared to RSA. Furthermore, it has been shown that for equal key size, the computational effort required for elliptic-curve cryptography and RSA is comparable. Therefore, there is a computational advantage to using elliptic-curve cryptography with a shorter key length than a comparably secure RSA.

10.3.2 Public-Key Security Attributes

This section examines the security implications of using public-key cryptography. Generally speaking, the strength of each algorithm is directly related to the type of the one-way function being used and the length of the cryptographic keys. Inverting the one-way functions we have discussed, namely, factoring a very large number and computing the discrete logarithm, is known to be practically infeasible within the computing means and the theoretic knowledge available today.

10.3.2.1 Confidentiality

The premise of the privacy service here is achieved by encrypting data, using the recipient's public key, and the fact that decryption can be done only by using the recipient's private key. For example, if Alice needs to send a confidential message

to Bob, she can encrypt it with Bob's public key, knowing that only Bob will be able to decrypt the ciphertext with his private key (see Figure 10.18).

Thus, only the recipient with knowledge of the private key is able to decrypt the enciphered data. It is worth noting that a privacy service strongly depends on the assurance that a public key is valid and legitimately belongs to the recipient.

One confidentiality problem that needs to be addressed by public-key encryption is the fact that in some cases, the plaintext corresponding to a given ciphertext can be easily understood. As an example, we consider the scenario in which Alice is a stock client and Bob a stockbroker, as shown in Figure 10.19.

Typically, Alice's messages are all likely to be of the type "Buy" or "Sell." Knowing this, an attacker could build a table mapping ciphertexts to plaintexts.

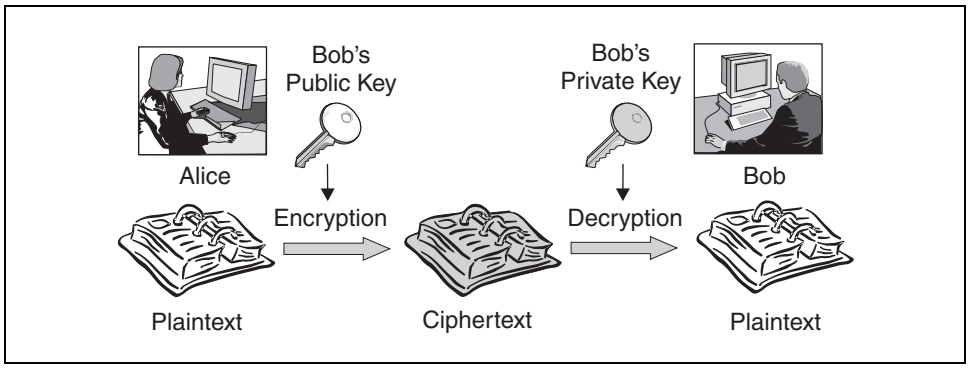

Figure 10.18. Public-Key Scenario

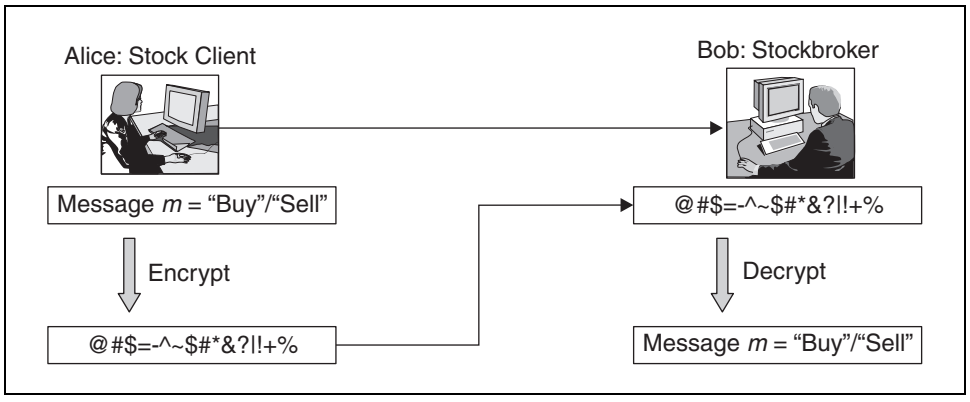

Figure 10.19. Scenario Requiring Message Randomization

This would break the confidentiality of the transmission. Even worse, the attacker could impersonate Alice and replace the ciphertext corresponding to "Buy" with the ciphertext corresponding to "Sell" and vice versa (see Figure 10.20).

This problem can be solved by *randomizing* the message. Before encrypting the plaintext message "Buy" or "Sell," the message-randomizing algorithm on Alice's side inserts a meaningless sequence of bits, which is randomly generated. As the ciphertext depends on the entire plaintext message, the ciphertexts produced by Alice are no longer recognizable. In addition, message randomization reduces the risks of message-prediction-and-replay-attacks (see footnote 6 on page 150).

10.3.2.2 Data Integrity, Data-Origin Authentication, and Nonrepudiation

As we said in Section 10.3.2.1 on page 367, privacy is provided by encrypting data, using a publicly available key, typically the recipient's public key. However, an eavesdropper may intercept the data, substitute new data, and encrypt it using the same public key. Simply applying a public-key algorithm to achieve privacy does not guarantee data integrity; nor does it guarantee data-origin authentication. In practice, digital signatures are the preferred method of achieving data integrity and data-origin authenticity. Another service that is inherently offered through digital signatures is nonrepudiation.

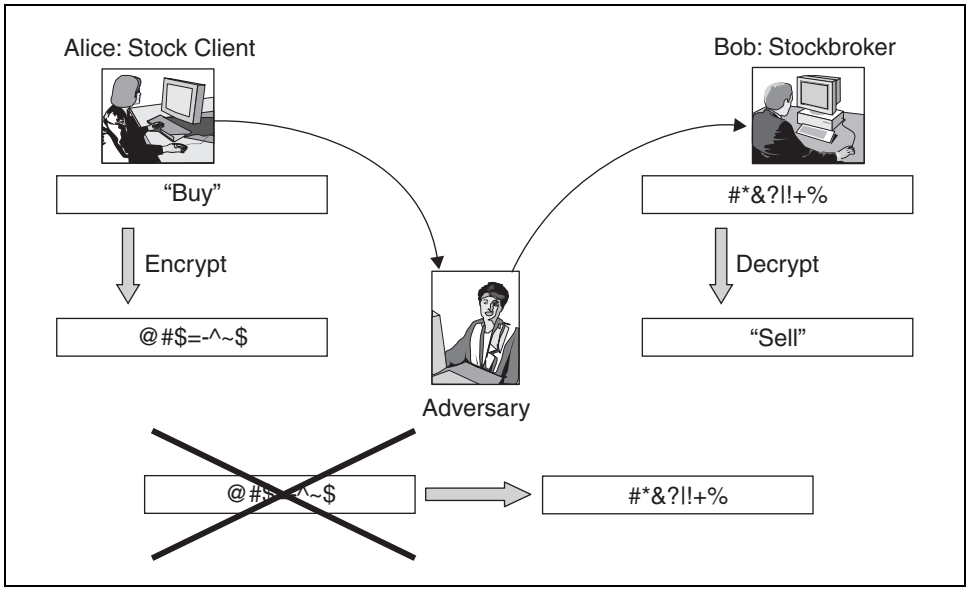

Figure 10.20. Message Randomization

10.3.3 Digital Signatures

The use of public-key cryptography combined with one-way hash functions enables the digital signing of documents. This process inherently enables data integrity and data-origin authentication and has the potential to withstand repudiation. In fact, using the private key of a public- and private-key pair to encrypt a data stream automatically binds the subject—a person or an entity—with the encrypted data.

The cost of encrypting an entire document in order to simply establish this binding can be prohibitive. Fortunately, digital signing of a document is a computationally affordable alternative, as it does not require encrypting the entire document.

If confidentiality is a requirement, the message originator, Alice, encrypts the message only once, with the public key of the receiver, Bob, thereby guaranteeing confidentiality, because the ciphertext can be decrypted only with the Bob's private key. However, data integrity, data-origin authentication, and nonrepudiation are not guaranteed, because anybody could have used Bob's public key to encrypt a different message, pretending that it was sent by Alice. To resolve this ambiguity, Alice attaches a digital signature to the encrypted message. The digital signature is obtained by applying a mathematical function to the plaintext message. This mathematical function depends on Alice's private key.

An eavesdropper who attempted to replace the transmitted data with new data could still encrypt the new data with Bob's public key but would not be able to use Alice's private key to generate Alice's digital signature. Once he receives the encrypted message and the digital signature, Bob decrypts the ciphertext with his own private key. Finally, he uses Alice's public key to verify Alice's digital signature. If the digital signature verifies, Bob knows that the original message has been sent by Alice and has not been compromised during transmission. Because Alice's private key has been used to compute the digital signature, this entire process guarantees data integrity, data-origin authentication, and nonrepudiation (see Figure 10.21).

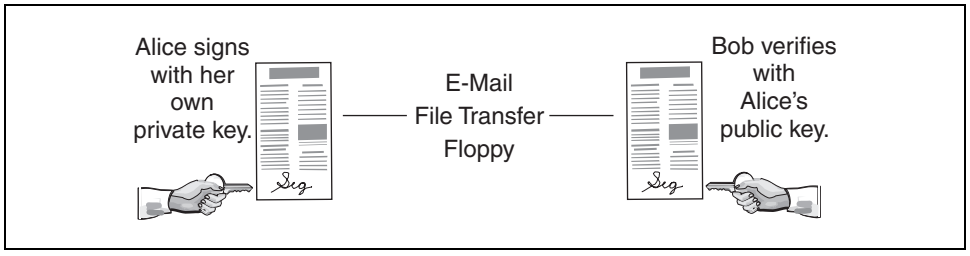

Figure 10.21. Digital-Signature Scenario

With the fundamental premise that the private key remains in the confines of its owner, verifying a digital signature using the associated public key certainly leaves no possibility for the originator to deny involvement. Denial, however, can always take place on the basis that a private key has been compromised. A strong nonrepudiation service never exposes the private keys it manages, even to the owner. Tamper-proof hardware modules for private keys become necessary for a legally binding nonrepudiation service.

If a confidentiality service is not needed, Alice can transmit the signed document to Bob in its cleartext form. The signature is provided to Bob for data-integrity verification, data-origin authentication, and nonrepudiation purposes.

The most well-known digital signature algorithms are RSA and Digital Signature Algorithm (DSA). These algorithms are discussed in the next two subsections.

10.3.3.1 RSA Signature

The RSA digital-signature algorithm proceeds along two main steps, as shown in Figure 10.22.

1. Using one of the common hashing algorithms, such as MD5 or SHA-1, a document is first digested into a much smaller representation: its hash value.
2. The hash value of the document, rather than the entire document itself, is then encrypted with the private key of the originator.

If confidentiality is needed, the document itself must be encrypted, as explained in Section 10.3.2.2 on page 369.

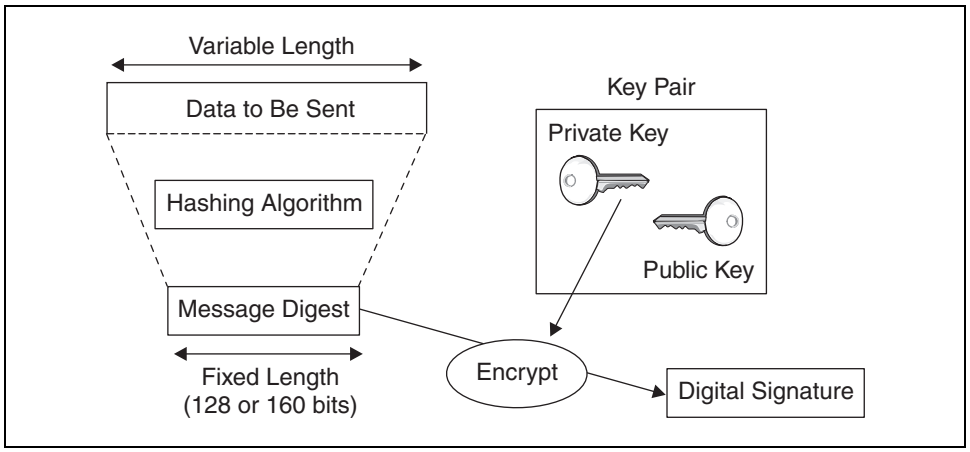

Figure 10.22. The Process of Computing a Message's RSA Digital Signature

10.3.3.2 DSA Signature

Other types of digital signatures rely on algorithms designed solely for signing but not encrypting. In other words, the digital signature is still obtained by encrypting the hash value of a document with the originator's private key, but the public and private key pair here can be used only for digital signing, not for encrypting arbitrary-size messages.

An example of this class of algorithms is the standard DSA, which computes a signature over an arbitrary-size input, using SHA-1 as a message digest, five public parameters, and a private key. DSA signatures have better performance characteristics than RSA does.

10.3.4 Digital Certificates

As we mention in Section 10.3.5 on page 375, authenticating the identity of a sending entity and protecting data to allow only authenticated and authorized receivers to view that data is an extremely important security requirement, especially for the exchange of security-sensitive data or when the nature of the transaction requires data-origin authentication and nonrepudiation. Encrypting a message with the receiver's public key guarantees confidentiality, whereas digitally signing a message by encrypting its hash value with the originator's public key guarantees data-origin authentication and nonrepudiation.

These scenarios are very attractive, but for them to work, it is necessary to have a means to bind a public- and private-key pair to its owner. To understand why, let us consider the following scenario. Alice wants to send Bob a confidential message in a secure manner over a public network. To do so, she needs to encrypt the message with Bob's public key. For sure, only Bob will be able to read the message once it is transmitted, because the message's ciphertext can be decrypted only with Bob's private key. However, how can Alice be sure that Bob is really Bob? Owning a public- and private-key pair does not give any assurance about the real identity of a person. Similarly, Bob may receive a signed message from Alice, and he can verify the digital signature's authenticity by decrypting it with Alice's public key, but how can he be sure that the entity that signed the message declaring to be Alice is really Alice?

A solution to this problem is to use digital certificates, which can be used to exchange public keys and to verify an entity's identity. An entity's *digital certificate* is a binary file that contains the entity's public key and Distinguished Name (DN), which uniquely identifies that entity, along with other pieces of information, such as the start and expiration dates of the certificate and the certificate's serial number (see Figure 10.23).

The international standard for public-key certificates is called X.509 (see Appendix B on page 553). This standard has evolved over time, and the latest version is V3. The most significant enhancement in X.509 V3 is the ability to add other,

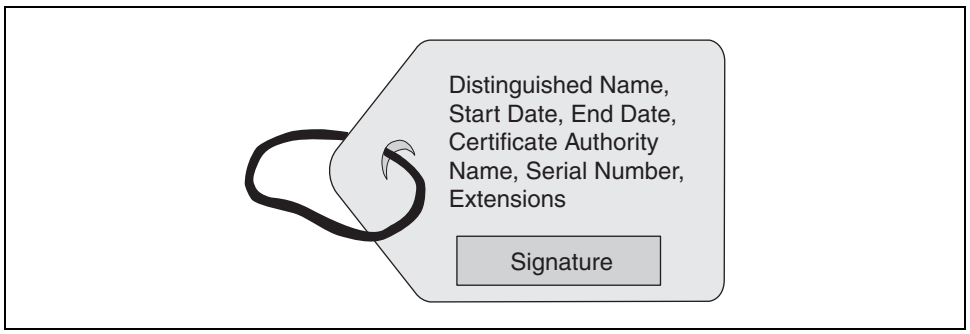

Figure 10.23. Information Contained in a Digital Certificate

arbitrary data in addition to the basic name, address, and organization identity fields of the DN. This is useful when constructing certificates for specific purposes. For example, a certificate could include a bank account number or credit card information.

Digital certificates are released by trusted third-party registry organizations called Certificate Authorities. These CAs are public organizations that are trusted by both the sender and the receiver participating in a secure communication. An entity, Alice, can receive her own certificate by generating a public- and private-key pair and by transmitting the public key along with a certificate request and proof of ownership of the public key to a CA. For serious applications, Alice can obtain a certificate only by applying in person and showing evidence of her identity. If Alice's request for a certificate is accepted, the CA wraps Alice's public key in a certificate and signs it with its own private key.

Alice can now convey her public key information to other entities by transmitting her certificate. A receiving entity, Bob, can verify the certificate's authenticity by verifying the CA's digital signature. This can be done without even contacting the CA, because CAs' public keys are available in all the most common client and server applications, such as Web browsers, Web servers, and other programs that require security. If the signature is verified, Bob is assured that the certificate really belongs to Alice. From this moment on, when he receives a message digitally signed by Alice, he knows that it is really Alice who signed it and transmitted it—data-origin authentication—and Alice will not be able to deny that the message originated from her—nonrepudiation. Similarly, by accessing Bob's certificate from a CA and by encrypting a message with Bob's public key, Alice is assured that only Bob, and no other person, will be able to decrypt the message—confidentiality.

As Figure 10.23 shows, certificates contain start and end dates. The validity of a certificate should not be too long, to minimize the risks associating with having inadvertently exposed the associated private key and to make sure that the current

key strength still makes it computationally infeasible to compute the private key from the public key. If the private key associated with the public key in a certificate gets inadvertently exposed, a certificate's owner should make an immediate request for suspending the certificate's validity. In this case, the CA will add an entry for that certificate in its certificate revocation list. A CRL also enumerates those certificates that have been revoked because their owners failed to comply with specific requirements. A CRL should always include data explaining why a certificate was suspended or revoked.

In the scenario that we have described in this section, there is only one CA that the sender and the receiver participating in a secure communication use to verify each other's public key's authenticity. In real-life situations, there are chains of CAs, whereby each successive CA verifies and vouches for the public key of the next identity in the chain. In this case, a public-key certificate embodies a chain of trust. Consider the situation shown in Figure 10.24.

A system has received a request containing a chain of certificates, each of which is signed by the next higher CA in the chain. The system has also a collection of *root certificates* from trusted CAs. The system can then match the top of the chain in the request with one of these root certificates, say, Ham's. If the chain of signatures is intact, the receiver can infer that Nimrod is trustworthy and has

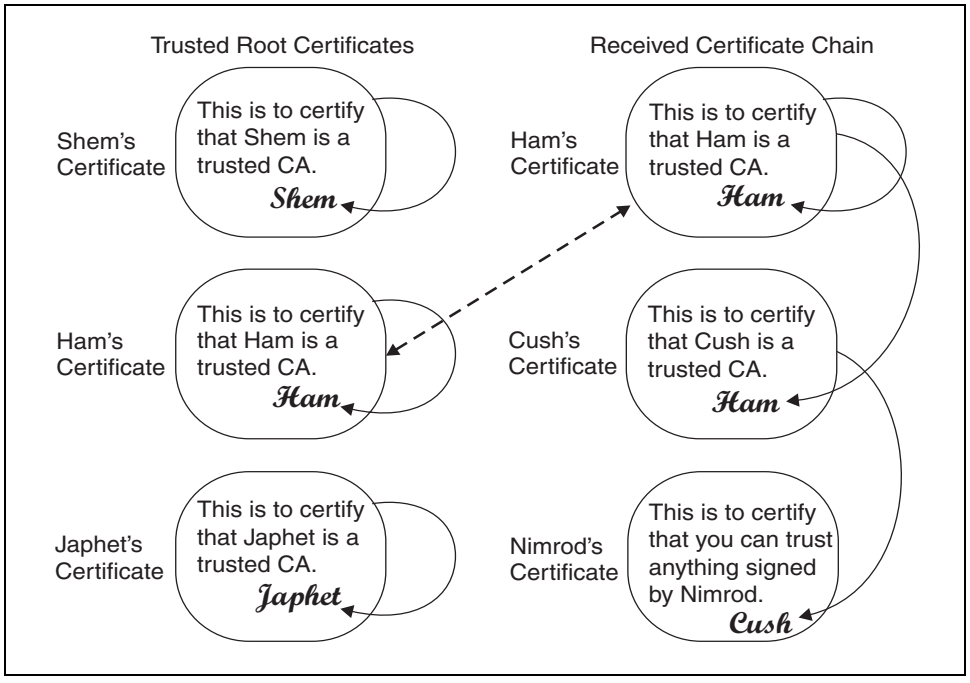

Figure 10.24. Certificate Hierarchy

inherited the trustworthiness from Ham. Note that one of the implications of a certificate chain is that the certificate at the top of the chain is *self-signed*.

10.3.5 Key Distribution

In public-key cryptography, an entity's private key never has to be exposed, whereas the corresponding public key is made publicly available. This makes it possible to obtain confidentiality, nonrepudiation, data-origin authentication, and data integrity without having to distribute the secret key. The main problem of public-key cryptography is that it is computationally expensive. Conversely, secret-key cryptography, described in Section 10.2 on page 346, offers better performance and scales well for Kerberos and distributed computing environment (DCE) security, even though its limitation lies in the fact that it becomes necessary to share the secret key across unsecure networks.

Combining public-key and secret-key cryptography yields the performance advantages of secret-key cryptography and the security enhancements of public-key cryptography, as shown in Figure 10.25. One algorithm that combines public-key and secret-key cryptography is DH (see Section 10.3.1.2 on page 362), which allows two parties to independently compute the same secret key. To do this, each

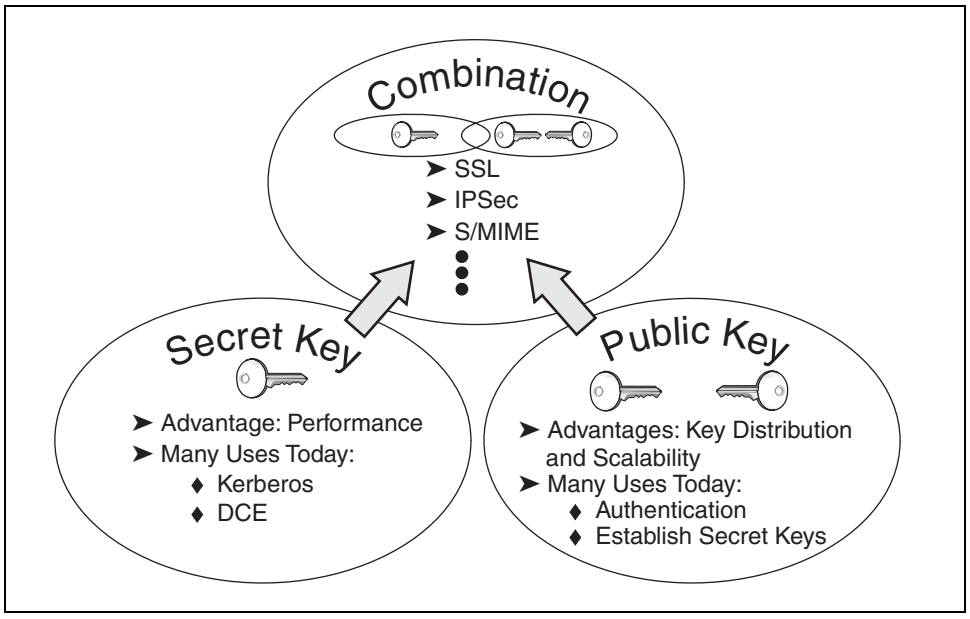

Figure 10.25. Combining Public-Key and Secret-Key Cryptography

entity uses its own private key and the other entity's public key. With Diffie-Helmann, the shared secret key is mathematically computed by the two parties, and there is no need to physically exchange it over the network.

Another way to use public-key cryptography for secure secret-key establishment over a public network is, essentially, to consider the secret key as the data that needs to be distributed with a privacy requirement. Thus, the secret key is encrypted using the public key of the target entity. The receiving entity uses its private key to decrypt the enciphered secret key and hence has established a common secret key with the sending entity. This is, for example, the approach used by the SSL and TLS protocols (see Section 13.1 on page 449). Other protocols that combine secret- and public-key cryptography are IPSec and S/MIME (see Section 12.2 on page 439).

Note that authenticating the identity of the sending entity is a strong security requirement. A breach in such a key-establishment mechanism risks exposing the entire cryptographic channel that follows key establishment.