Security challenges for Enterprise Java in an e-business environment

by L. Koved
A. Nadalin
N. Nagaratnam
M. Pistoia
T. Shrader

As e-business matures, companies require enterprise-scalable functionality for their corporate Internet and intranet environments. To support the expansion of their computing boundaries, businesses have embraced Web application servers. These servers support servlets, JavaServer Pages™, and Enterprise JavaBeans™ technologies, providing simplified development and flexible deployment of Web-based applications. However, securing this malleable model presents a challenge. Successful companies recognize that their security infrastructures need to address the e-business challenge. They are aware of the types of attacks that malevolent entities can launch against their servers and can plan appropriate defenses.

The Java** technology has established itself in the enterprise realm, both for the ease with which developers can create component software and for the platform-independence of the language. The Java 2 Platform, Standard Edition (J2SE**) introduced a fine-grained, policy-based security model that is customizable and configurable into numerous protection domains, which are well-suited to component-based software. To provide security for e-business, the Java platform builds upon a core set of technologies.

Enterprise security requires authentication to identify a principal user based on the enterprise user registry, authorization to enforce the enterprise security policies, encryption to keep information confidential, and pliable management of information. In the Java environment, these technologies manifest themselves through the J2SE security architecture, Java Authentication and Authorization Service (JAAS), Java Cryptography Architecture (JCA), Java Cryptography Extension (JCE), Java Secure Socket Extension (JSSE), Public-Key Cryptography Standards (PKCS), and support for the Public Key Infrastructure (PKI).

This paper describes the infrastructure and the security considerations that companies deploying a Web application server (WAS) must consider. We delve into the set of Java security technologies that can be applied. In addition, this paper discusses the current J2SE security architecture and security for the WAS environment, focusing on the challenges that lie ahead as these architectures evolve to address enterprise e-business needs.

Web application servers

Web application servers have recently come of age, supplementing and often surpassing traditional Web servers in their use and functionality in enterprise environments. A WAS differs from a traditional Web server because it provides a robust and flexible foundation for dynamic transactions and objects. Traditional Web servers are constrained to servicing standard HTTP (HyperText Transfer Protocol) requests, returning the contents of static HTML (HyperText Markup Language) pages and images or the output from executed CGI (Common Gateway Interface) scripts. Some Web servers have tried to extend their functionality by including Java servlet[1] support. Serv-

0018-8670/01/$5.00 © 2001 IBM

lets allow Java applications to be executed on the server side and return dynamically generated information, in much the same way that a CGI script can dynamically generate information. However, grafting servlet support onto a Web server does not provide a complete solution to meet e-business requirements.

A WAS provides a more scalable environment for modeling enterprise solutions, partly through the exploitation of Java technology. A WAS supports Java objects in their simple and compound manifestations. Servlet support plays an important role, but a WAS also supports Enterprise JavaBeans** (EJB) and JavaServer Pages** (JSP) technologies.[2] EJB technology provides an architecture for distributed transaction-based objects, allowing developers to concentrate on business logic, rather than its interaction with infrastructure. Similarly, JSP technology provides developers and administrators with a convenient way to generate HTML or XML (Extensible Markup Language) via Java programming to generate dynamic information.

A WAS supports other functionality, but the blend of EJB, servlets, and JSP provides the foundation for representing a model-view-controller (MVC)[3,4] architecture to developers. The modularity and focus of purpose in EJB represents the model. JSP and servlets represent the view through their ability to dynamically generate information (e.g., HTML, XML). The client system represents the controller.

WAS security environment

A WAS is more than just a grouping of Java objects. In the multiuser enterprise environment, a WAS also provides integrated authentication and authorization support for user transactions, whether they originate from Web browsers, client applications, or another WAS. This complex environment of both friendly and potentially malicious entities requires answers to a number of questions:

- How can users be authenticated to a WAS?
- After authentication, to which objects and actions are the users authorized?
- How can the details of users' transactions remain hidden from unauthorized entities?

These questions fundamentally deal with the security issues of authentication, authorization, and encryption.

Objects that play a major role in the WAS environment are client objects and server objects. Client objects include Web clients and application clients. Server objects include Web servers, WASs, security servers, and user registry servers.

The physical WAS executing the supported Java objects plays an important role, but it is just one member of the entire ensemble of processes that complete the WAS environment. Note the distinction between the WAS and the WAS environment. The WAS represents the server that handles requests for EJB objects, servlets, and JSP pages. The WAS environment encompasses all of the client and server objects that have a direct or indirect interaction with the WAS, including the WAS itself.

Clients to a WAS include Web browsers, Java client applications or applets, and pervasive devices, such as mobile appliances. Client objects can originate from many different sources, but they can be categorized into those from traditional Web browsers and those from stand-alone applications.

A WAS typically does not directly service client requests. Web servers respond to HTTP requests for HTML pages or to execute CGI scripts. For more complex tasks, such as the manipulation of EJB objects, Web servers pass the service requests to the WAS.

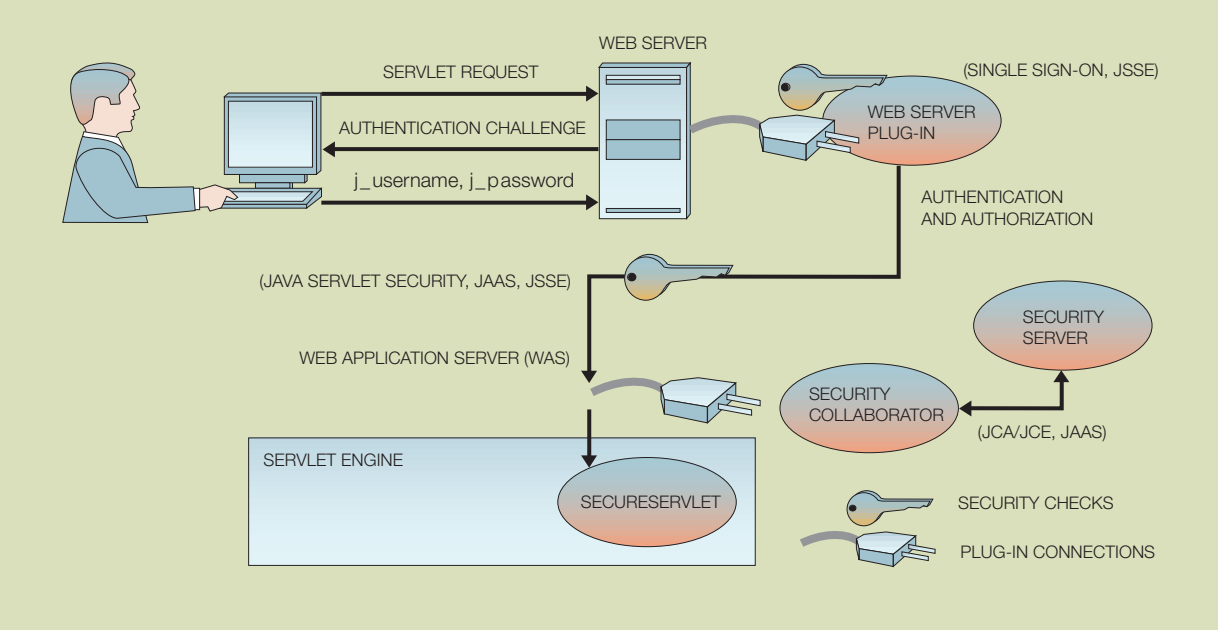
Playing a quiet but essential role in the WAS environment, the security server maintains a consistent security schema. It arbitrates user authentication and authorization access to objects. To authenticate users, as well as obtain the user's authorization attributes and digital certificate information, the security server obtains the information from an enterprise user registry (e.g., LDAP server, OS/390* RACF*, and SecureWay* Policy Director).[5]

Security in the WAS environment is not limited to authentication and authorization. Between the different server and client objects, security also comes into play through the use of encryption technologies.

Role of Java security technologies in a WAS enterprise environment

This paper describes a set of Java security technologies and how each of them plays a vital role in securing a WAS environment. This section describes the general flows within the WAS environment. Later sections discuss the role of security technologies within these flows in further detail.

Figure 1 Secure servlet invocation scenario



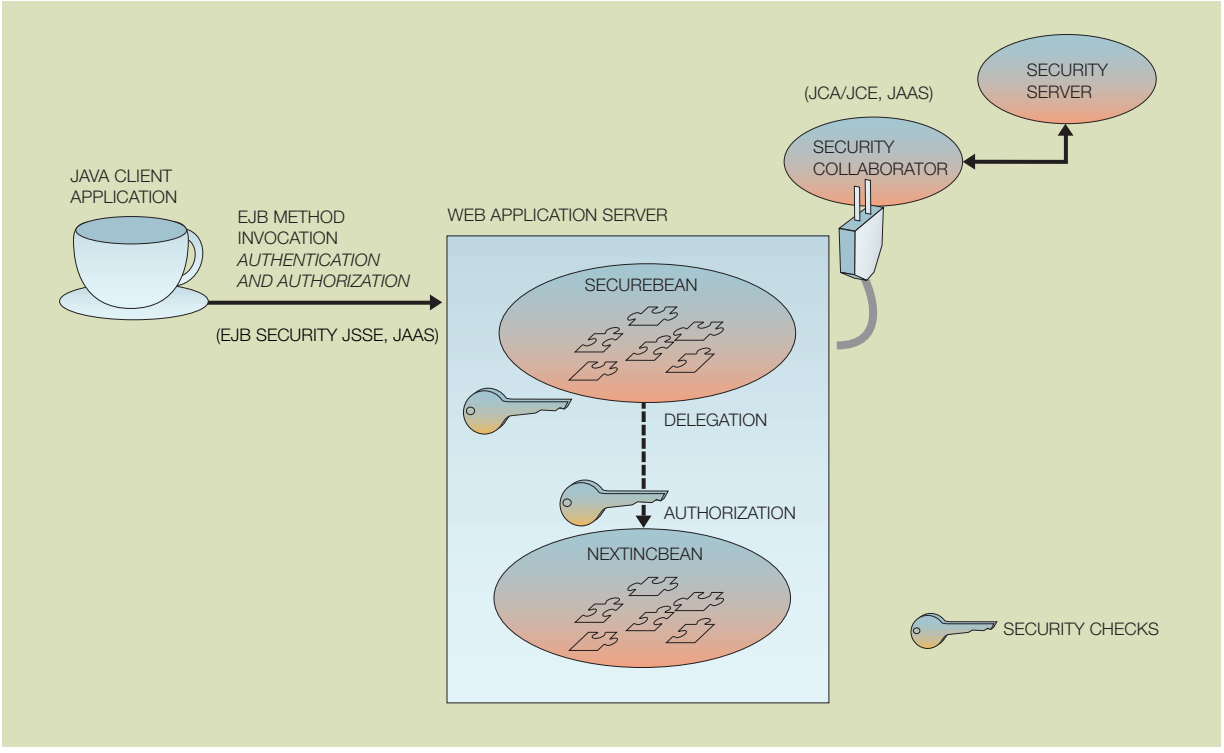
Invoking a secure servlet from a Web client. In a typical scenario, the user submits a request to a Web resource, such as a Java servlet, by specifying the uniform resource locator (URL) corresponding to that resource. Since it does not internally process servlets, the target Web server receives the request and forwards it on to the WAS (see Figure 1). The WAS determines that the requested resource is protected and the security collaborator must authenticate the user. The security collaborator resides at the server's entry point to help broker requests with the security server, based on security policies. If the policy requires a user ID (identifier) and password, the Web server may issue an HTTP basic challenge or post an HTML form to request authentication data from the user. Tighter security policies may dictate a PKI-based authentication, where the user is required to present a client digital certificate.

An administrator may configure a WAS with policies based on security specifications for Java servlets and manage authentication and authorization with Java Authentication and Authorization Service (JAAS) modules. An authentication and authorization service may be written in Java code or interface to an existing authentication or authorization infrastructure. For a cryptography-based security infrastructure, the security server may exploit the Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE). To present the user with a usable interaction with the WAS environment, the Web server may employ a form of "single sign-on" to avoid redundant authentication requests. A single sign-on preserves user authentication across multiple HTTP requests so that the user is not prompted many times for authentication data (i.e., user ID and password).

Invoking a secure EJB object from a Java client. Java client applications may invoke EJB methods by submitting a request to the WAS that hosts an EJB container. As with Web clients, Java security technologies can play a role in the security decisions made during this request flow from a client application (see Figure 2).

Based on the security policies, JAAS may be employed to handle the authentication process with the identity of the Java client. After successful authentication, the WAS security collaborator consults with the security server. Based on the result of the authentication process and EJB security policies, the security collaborator will make an EJB authorization decision. If the authorization succeeds, the collaborator will enforce the delegation policy associated with that

Figure 2 Secure EJB invocation scenario



method. For example, if the delegation policy on a method on SecureBean is set to enforce impersonation, when that method invokes a downstream NextIncBean method, the method call will be performed under the client's identity.

The WAS environment must ensure the confidentiality of information exchanged between the client and the environment, as well as between servers, by maintaining a mutually authenticated, secure communications channel for information flow. JSSE can be used to establish these secure channels. EJB objects can additionally exploit various security technologies, such as those available with JCA, JCE, PKCS, and S/MIME (Secure Multipurpose Internet Mail Extensions).

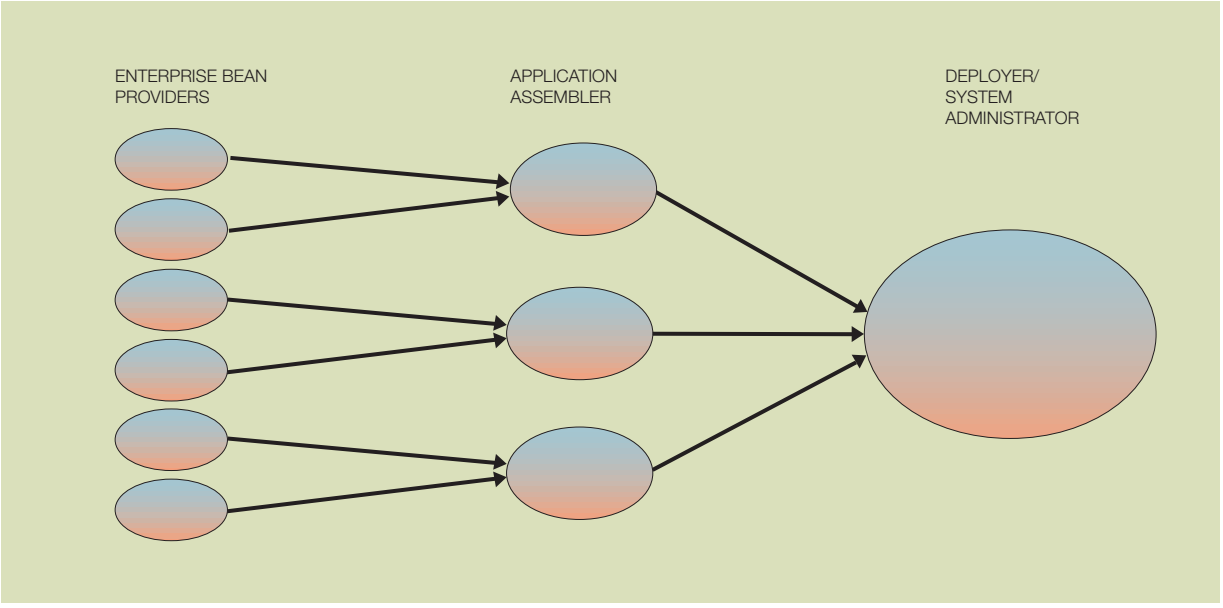
The environment and security flows just described are relevant to any enterprise. It is left to the implementation of such an environment to enforce various security policies at relevant checkpoints throughout the flow. The rest of this paper will describe these security technologies and how they can be used individually or combined to present a secure Java environment for the WAS enterprise. In particular, this paper will cover the following technologies:

- Enterprise JavaBeans (EJB)
- Java Authentication and Authorization Service (JAAS)
- Servlet security
- Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE)
- Java Secure Socket Extension (JSSE)
- Public-Key Cryptography Standards (PKCS) and Secure/Multipurpose Internet Mail Extensions (S/MIME)

Enterprise JavaBeans

Although they can service many heterogeneous objects, WASs specialize in providing EJB containers. As e-business applications progress from small-scale endeavors to the demands of enterprise-wide solutions, most incorporate EJB objects to model the business. This section introduces the EJB security model and

Figure 3 Relationship of EJB providers, application assembler, deployer, and system administrator



its support for authentication, authorization, and delegation.

EJB is a server-side component-programming model for distributed transaction processing.[6,7] The EJB 1.1 specification[6] prescribes a number of *roles*, including *enterprise bean provider*, *application assembler*, *deployer*, *system administrator*, *EJB server provider*, and *EJB container provider*. Each of these roles takes on specific responsibilities in managing and implementing security.

As shown in Figure 3, the EJB component programming model aggregates multiple EJB objects to create functionality for applications requiring transaction-oriented characteristics. Written by enterprise bean providers, these components are aggregated by application assemblers who associate transactional, security, and other properties with the components. These aggregated components are stored in an EJB Java archive (JAR) file. Included in the EJB JAR file is a *deployment descriptor* that describes the various properties of the EJB objects stored in the JAR file.

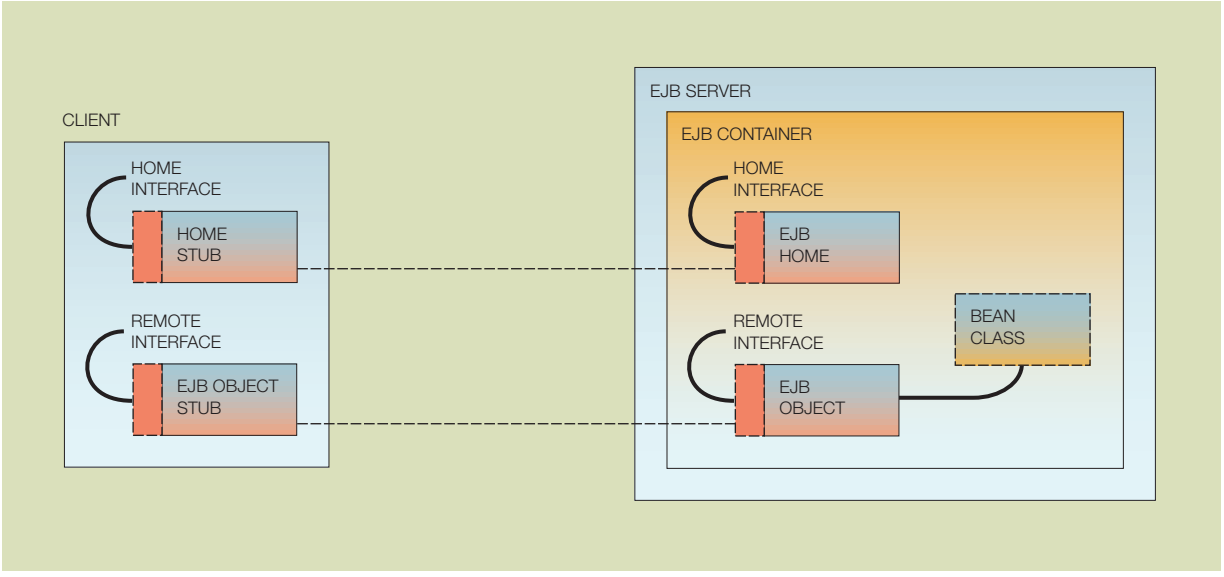
To execute an EJB object, an EJB JAR must be *deployed*. An EJB container provider is an organization that supplies an implementation of an EJB run-time environment (e.g., middleware). The EJB container provider is also responsible for providing tools that allow the deployer to:

- Process an EJB JAR file
- Configure an EJB server to install and execute an EJB object
- Configure the network environment to contain the Remote Method Invocation (RMI) stubs needed by the client to communicate with the EJB object via the EJB container
- Provide directory services, such as Java Naming and Directory Interface (JNDI)[8] to locate the deployed EJB bean

In addition, the EJB container provider supplies tools for security management at deployment time, as well as for the *system administrator* for ongoing security administration of the EJB object and container run-time environment.

Another key element of the EJB programming and run-time model is that it is a distributed programming model. In fact, the only architected means for communicating with an EJB object is through its *HomeInterface* object via an RMI call. That is, the EJB programming model is a distributed programming model where components and applications communicate with each other via remote method calls. The

Figure 4 The EJB programming and run-time model as a distributed run-time environment



EJB container mediates all communication between an EJB client and the EJB instance (see Figure 4).

Authentication. The EJB specification does not directly address the issue of authentication and the authentication process. Most of the specification deals with authenticated principals and the actions they can perform, such as invoking methods specified in an EJB HomeInterface or RemoteInterface object. In most respects, the execution model follows a simplified version of the Common Object Request Broker Architecture** (CORBA**) security model.[9,10] Authentication is the responsibility of the EJB container and could be implemented using JAAS.

In the Object Request Broker (ORB) authentication model, an ORB, such as RMI over Internet Inter-Orb Protocol (RMI-IIOP),[11,12] receives a method invocation request and examines the security attributes of the request prior to method dispatch.[13] If the principal(s) are authenticated, then the process moves on to the authorization phase.

Aside from the authentication process, the result of authentication appears in several places. In particular, EJB includes a notion of a *security role*. The security role is a grouping of principals to make it easier for the deployer and system administrator to administer authorization. When a principal is authenticated, the principal is logically assigned to zero or more security roles defined by the application (as configured by the deployer). Further details of security roles are covered later, in the subsection on authorization.

For the enterprise bean provider, authenticated user information is available in two methods in the EJBContext class. The first is the getCallerPrincipal( ) method, which returns a Principal object representing the authenticated principal on whose behalf the EJB object is executing. The result of getCallerPrincipal( ) is implementation-specific and will be discussed later in the subsection on delegation. The second method containing user information is the method isCallerInRole( ), which accepts a String object as an argument. After the requesting principal is authenticated and authorized, an EJB method is invoked. During the execution of the EJB method, the application code can determine if the calling principal possesses a specific EJB security role. The role name passed in the isCallerInRole( ) method call is compared to the security roles assigned to the authenticated user. If that name is one of the roles, the method returns `true`, otherwise, `false`.

From an authentication perspective, the EJB container provider is responsible for supplying tools and run-time support for:

- The javax.ejb.EJBContext methods getCallerPrincipal( ) and isCallerInRole( )
- Security role name mappings as defined in the EJB deployment descriptor
- Tools for the deployer and system administrator to perform security administration, including security role mapping and assignment of roles to principals or user groups
- Principal authentication and delegation support in the ORB

Authorization. Much of EJB security is concerned with authorization. As previously noted, EJB authorization is based on a simplified CORBA security model, which asks whether an authenticated principal (or group of principals) is authorized to invoke a method accessible via the ORB. Also, EJB security is about the *process* of deploying an application so that it can be secure. As such, EJB authorization will be discussed from the perspective of each EJB security role.

Enterprise bean provider. An enterprise bean provider writes business logic in Java code. The intent of the EJB architecture is to allow these providers to write application code without having to understand or be involved in security. Aside from the javax.ejb.EJBContext methods described above, there are no other methods or operations that an EJB object can use to influence EJB security.

However, if the EJB uses the isCallerInRole( ) method call, then the EJB deployment descriptor will need to contain entries that enumerate the security role names used in these method calls.

Application assembler. An EJB JAR file delivered to a deployer may contain EJB objects from multiple enterprise bean providers. It is the responsibility of the application assembler to aggregate objects and create the deployment descriptor that goes into this EJB JAR file. A deployment descriptor is written in XML and provides the road map for deploying the objects in an EJB JAR file. In particular, the application assembler defines security role names that apply to all EJB objects in the EJB JAR. Since the security role names used in each of the EJB objects may not be the same as those defined for the EJB JAR file, the application assembler must provide a mapping of the EJB object-specific security role names to the security role names used for the entire EJB JAR. The security role names defined by the application assembler will be the security role names that will be visible to the deployer.

The application assembler is also responsible for associating the EJB JAR security role names with each EJB method that is to be made accessible via the ORB. Each method can be assigned individually to a security role, or all methods in a class can be assigned to a security role. Also, methods can be associated with multiple security roles. This allows an authenticated principal associated with *any* of the assigned security roles to invoke the method. The method security role assignments are intended to be a hint to the deployer. It is the responsibility of the deployer to either accept the security role mappings or modify them as appropriate for the deployment environment.

Deployer and EJB container provider tools. A deployer receives an EJB JAR file from an application assembler and uses tools provided by the EJB container provider to process the deployment descriptor, including the security attributes (e.g., role names and mappings). Also, the deployment tools allow the deployer to modify any of the default method or security role mappings found in the EJB JAR file, and to assign enterprise-specific principals the EJB security roles. In addition, the deployer may adjust any method and security role mappings to suit the target environment.

The ability to modify mappings of roles to methods is required to adhere to the organization structure and security policies of the environment into which the beans are deployed. For example, teller and cashier may be two declared roles in an EJB JAR file. The organizational structure of the financial organization in which they are deployed may have one role, finance assistant, responsible for performing both the teller and the cashier tasks. In this case, the deployment tool should allow the deployer to modify both the role names, teller and cashier, to become finance assistant. As a result of this modification, all the methods related to teller and cashier will be associated with finance assistant.

EJB container provider. A WAS is responsible for enforcing the authorization constraints specified by the deployer on an EJB container. This includes the security role name mappings, method authorization, etc. Note that if any method in a deployed EJB JAR file is not associated with a security role, or if a security role is not associated with any principals in the deployed system, the WAS will prohibit access to the method. The implementation of an authorization mechanism is outside the scope of the EJB specification and will vary depending on the implemen-

tation of the WAS, EJB container, and RMI ORB. Implementations typically exploit existing infrastructure, such as OS/390 RACF, SecureWay Directory, or the Windows NT** Active Directory.

Delegation. Delegation is the process of forwarding a principal's credentials along with associated tasks that the principal originated or is having performed on its behalf.[14] The EJB specification does not define how delegation is to be handled since it is tied to a specific container implementation. In fact, there is nothing in the EJB JAR deployment descriptor to help a deployer manage delegation. Largely, delegation is an issue for the WAS environment and administration.

When a deployer configures an EJB object for a container, the container tools provide any necessary interfaces to configure delegation. One notable problem with delegation is that the javax.ejb.EJBContext.getCallerPrincipal( ) method is ill-defined. In general, the enterprise bean provider cannot know the delegation configuration in the deployment environment. Also, the EJB specification does not indicate who the "caller principal" really is—it could be the client that initiated the original call to the WAS, or it could be the immediate caller to the EJB instance. If delegation is enabled, the result of getCallerPrincipal( ) may not be the principal that the enterprise bean provider expected.

If administrators enable delegation,[15,16] they must configure their delegation policies so that the WAS environment conforms to security policies of the enterprise. Therefore, the identity of the initiator of the call is often closer to the desired semantics of the getCallerPrincipal( ) method than that of the immediate caller to the EJB instance.

Along with authentication, it is the responsibility of the EJB container implementation in a WAS to enforce the policies defined and supported within the environment. The WAS environment can use the JAAS technology to implement delegation policies. As later described in the JAAS section, the Subject.doAs( ) method can be used to execute a set of actions under the identity of the specified principal. Because this is not a part of the EJB specification, one cannot expect the same behavior when the EJB objects are deployed in different WAS environments, since they may have different authentication mechanisms and delegation support.

Security considerations. While not specifically a security consideration, the EJB programming model constrains an EJB object to a limited set of resources. Specifically, in a J2SE environment, an EJB object is prohibited from using many Java resources as a result of the EJB isolation architecture. This architecture scales well in a multiprocess and multithreaded WAS environment. Each EJB object is designed to run by itself and not interfere with other EJB objects. For example, EJB objects cannot include native method calls since the native code may not be portable and could adversely affect other EJB objects. Note that since the EJB container is middleware and acts as a broker, it can make these native calls or use restricted Java resources.

It is important to define and enforce a secure association mechanism between EJB components.

Given the isolation architecture of EJB components, each EJB object needs to interact with other EJB objects using remote method calls. Because the interaction will be location-independent and may cross machine boundaries, it is important to define and enforce a secure association mechanism between EJB components. The channel of communication should provide data confidentiality and integrity. Using SSL (secure sockets layer) communication between the EJB containers may address these requirements. It is also important that the security context associated with the invoking EJB object be passed on to the target EJB object, so that the identity of invocation gets propagated. A mechanism that achieves secure association between the source and target EJB components needs to address the quality of the service requirements. Some of these issues are addressed in the Java 2 Platform, Enterprise Edition (J2EE**) specification.[2]

Java Authentication and Authorization Service

The WAS environment authentication requirements can be fairly complex. In a given deployment environment all applications or solutions may not originate from the same vendor. In addition, these ap-

plications may run on different operating systems. Java is the language of choice for portability between platforms, but it needs to marry its security features with those of the containing environment. This section describes how JAAS accomplishes this union, and how it can be exploited in the WAS environment.

Authentication and authorization are key elements in any secure information handling system. Since the inception of Java technology, much of the authentication and authorization issues have been with respect to downloadable code running in Web browsers. In many ways, this had been the correct set of issues to address, since the client's system needs to be protected from mobile code obtained from arbitrary sites on the Internet. As Java technology moved from a client-centric Web technology to a server-side scripting and integration technology, it required additional authentication and authorization technologies.

Traditional computing systems perform authentication on a principal or accountable entity, typically through some sort of challenge-response mechanism. The most salient of these is a user ID and password combination that is often used for server or Web resources, such as HTTP basic authentication. However, the challenge may be more complex, including the encryption of information, the possession of a specific physical token (e.g., a key for a physical locking mechanism), or possession of specific information (e.g., mother's maiden name or value from a one-time keypad). The response must be valid based on the type of the challenge.

Similarly, most computing systems base authorization on an authenticated principal and a list of resources authorized for use by the principal. Most often, the authenticated principal is associated with an operating system process or thread of execution. When protected resources are accessed, the authorization mechanism verifies whether the currently executing principal is authorized for the resource.

Before JAAS, existing Java mechanisms did not provide the structure needed to support traditional authentication and authorization. Security in J2SE was based on the use of public key cryptography (digital signatures) on the *code* executing in the Java virtual machine (Jvm), not the principal making a request for computing or data resources. Similarly, authorization was based on the *code* attempting to use the computing or data resources.[17,18] JAAS was designed to address these shortcomings in a manner consistent with the existing J2SE infrastructure.

JAAS is divided into two major components: authentication and authorization. The authentication part is based on Pluggable Authentication Modules (PAMs),[19] with a framework designed to be used both on clients and servers. The authorization aspects were designed to be an extension of the authorization mechanisms already found in J2SE.

Authentication: LoginModule objects. The kind of proof required for authentication may depend on the security requirements of a particular resource and enterprise security policies. To provide such flexibility, the JAAS authentication framework is based on the concept of configurable authenticators. This architecture allows system administrators to configure, or *plug in*, the appropriate authenticators to meet the security requirements of the deployed application. The JAAS architecture also allows applications to remain independent from underlying authentication mechanisms. So, as new authenticators become available or as current authentication services are updated, system administrators can easily replace authenticators without having to modify or recompile existing applications.

The JAAS LoginContext class represents a Java implementation of an enhanced PAM framework. A LoginContext object consults a configuration table to determine the authenticators, or LoginModule objects, that are to be employed by an application.

JAAS, like the PAM framework, supports the notion of *stacked authenticators*. The JAAS authentication framework ensures that either *all* log-in modules succeed or none succeed. It is the responsibility of the LoginContext object performing the authentication to ensure this "all or nothing" behavior. This is achieved in two phases:

1. In the first phase of authentication, or the log-in phase, the LoginContext object invokes the specified log-in modules and instructs each to attempt, but not commit, the authentication. If all the required log-in modules successfully pass the first phase, the LoginContext object then can proceed to the second phase.
2. In the second phase, each configured LoginModule object is instructed to formally commit to the authentication process. During this phase, each LoginModule object associates the appropriate

authenticated Principal and Credential objects with the Subject object.

If either phase fails, the LoginContext object instructs the configured LoginModule object to abort the entire authentication process. Each LoginModule object is then required to clean up (e.g., discard) any state that had been associated with the attempted authentication.

An important feature in JAAS is the mechanism by which an authenticated context is established. Ordinarily, authentication is not part of the normal Java method dispatch path. Thus, if a Java program wishes to request authentication, it should construct a LoginContext object and call its login( ) method. The login( ) method will construct and annotate a Subject object with appropriate authenticated Principal and Credential objects. Subsequently, to assume the identity of that Subject object, the program calls the Subject.doAs(Subject, PrivilegedAction) method. This method call runs the specified PrivilegedAction object's run( ) method with the security attributes of the specified Subject object. After the PrivilegedAction. run( ) method terminates, the program returns to the security state in effect prior to the Subject.doAs( ) call. When the program no longer needs the authenticated identities, it can simply call the LoginContext object's logout( ) method.

To allow selection of an appropriate set of LoginContext objects, a java.lang.String object is passed to the LoginContext constructor. This string is used as an index into the configuration file, which selects the appropriate log-in module(s) to be used for authentication. Naming conventions for this index are entirely up to the Java program.

An important feature of JAAS is that it can be configured to support a wide variety of authentication mechanisms and the arguments that they might take. For example, authentication could require:

- An account name (or user name) and a password
- A distinguished name (DN) and the ability to prove identity through a digital signature
- A fingerprint or a retinal scan

Since one of the goals of JAAS is to have a pluggable authentication mechanism, the framework methods are generic enough to allow all authentication mechanisms to work, and simple enough to avoid complexity for authentication mechanism providers. This is handled by having the four authentication methods visible at the LoginModule application programming interface (API). The login( ), commit( ), abort( ), and logout( ) methods have no parameters, and a Java return type of `boolean` (`true` if the method succeeded, `false` if the log-in module should be ignored). This approach does not require authentication providers to add constraints to their current interfaces, but still leaves unresolved the issue of how to provide the additional configuration information when needed.

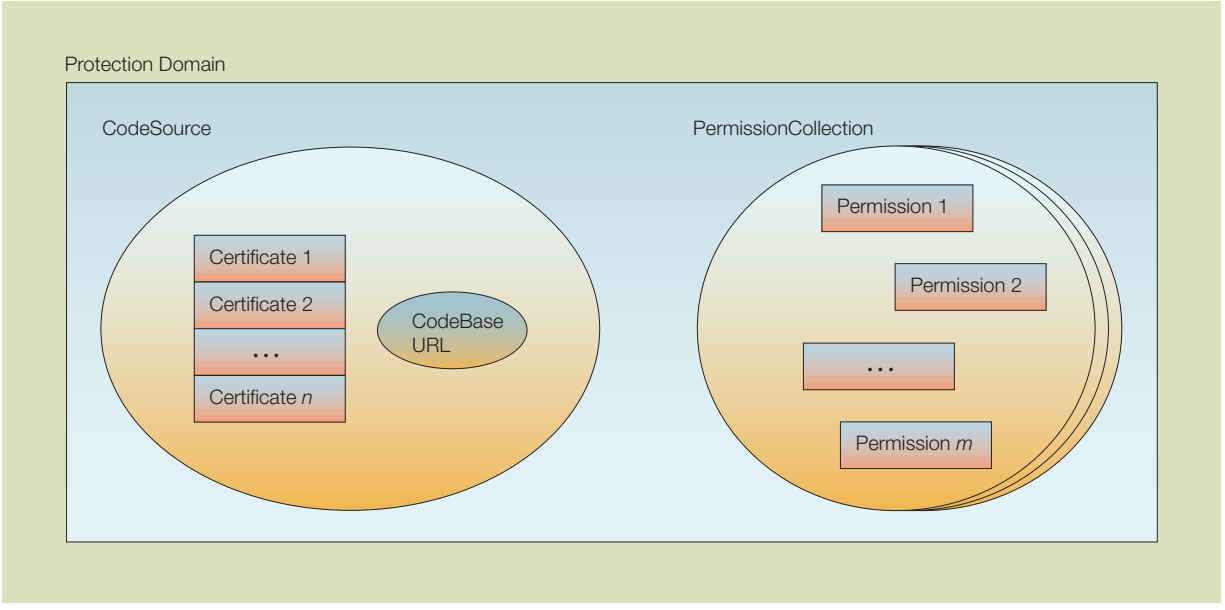
The mechanism for providing additional information is to have an initialize( ) method in each LoginModule object, where an optional CallbackHandler object can be specified. When employed, these objects can provide implementation- and environment-specific information needed to satisfy a particular LoginModule object. So, if a LoginModule object needs environment information to authenticate the user, it can examine the array of instantiated Callback objects to extract the necessary information. For example, if a user ID and password are needed, the LoginModule object can examine the callback array to see if it contains a NameCallback object and a PasswordCallback object. If so, it will use them to obtain the necessary information.

In addition to JAAS, the Generic Security Services Application Programmer's Interface (GSS-API)[20] and Simple Authentication and Security Layer (SASL) Application Programmer's Interface[21] define frameworks that provide additional support for pluggable authentication. Specifically, the GSS and SASL authentication frameworks were designed for network communication protocols.[22] As such, they provide additional support for securing network communications after authentication has completed. While JAAS accommodates general network-based authentication protocols, it also addresses the need to support pluggable authentication in stand-alone and nonconnection-oriented environments.

Authorization. J2SE authorization[17,18] is based on the notion of classes being members of protection domains, threads of execution each having an AccessControlContext object, and an authorization policy that is enforced by an AccessController object.

The Jvm loads classes and associates each class with a CodeSource object, which contains the base URL from which the code was loaded and the array of certificates associated with the entities that signed the code. Each class is loaded into the Jvm via an instance of the SecureClassLoader class, or one of its

Figure 5 Graphical representation of a ProtectionDomain object



subclasses, and is assigned to a ProtectionDomain object based on its CodeSource object. The ProtectionDomain object contains a set of Permission objects, and these objects indicate which resources the class is authorized to use. These resources include file and network access and the ability to print and execute code outside of the Jvm.[16,17]

The AccessControlContext object is a snapshot of all of the ProtectionDomain objects currently associated with a thread of execution (see Figure 5). That is, when a method is called, a new activation record is placed on the Java run-time stack. Each new method call results in a new activation record being placed on the stack, and when the method returns, the activation record is removed. When AccessController.getContext( ) is called, a search is made of the run-time stack to locate the Java class for each of the activation records. Since each Java class is associated with a ProtectionDomain object, a set is constructed that contains all of the unique ProtectionDomain objects associated with the classes in the current thread of execution. The AccessControlContext object is this set of ProtectionDomain objects.

To determine whether a thread of execution is allowed to access a protected resource, a call is made to AccessController.checkPermission( ), which creates an instance of an AccessControlContext object based on the activation records in the current thread. The AccessControlContext object is then given the Permission object passed to the AccessController.checkPermission( ) call and asked whether all of the ProtectionDomain objects that it contains are authorized for the specified Permission object. If any ProtectionDomain in the AccessControlContext object fails to contain the specified Permission object, the authorization test fails. A successful test of the Permission object indicates that all of the classes currently executing in the thread are authorized to use the resource.[6,17]

JAAS authorization. The challenge for JAAS designers was to add principal-based authorization to J2SE in a way that would not disturb the existing authorization mechanisms. They met the challenge by extending the J2SE authorization mechanism, associating a Subject object and its set of Principal objects with a thread of execution and logically extending the ProtectionDomain objects of executing code to include associated Permission objects.

To associate a Subject object with a thread of execution, the Subject.doAs( ) method is called with a Subject object containing authenticated Principal objects and a PrivilegedAction object containing the

code to be executed with the Subject object's Permission object(s). The PrivilegedAction interface is the same as that used for passing privileged code to an AccessController.doPrivileged( ) call.[17,18]

In J2SE version 1.3, the AccessControlContext object accepts a java.security.DomainCombiner object, which is called during the authorization process. The purpose of the DomainCombiner object is to modify, in some suitable fashion, the ProtectionDomain objects associated with the classes on the thread's activation stack. In the case of JAAS, a SubjectDomainCombiner object is used, which logically extends each ProtectionDomain object found on the stack so that it includes the ProtectionDomain objects of all of the Principal objects in the currently active Subject object in the thread.[23]

Security considerations. Developers have exploited JAAS in WAS environments to bridge their applications to a native authentication and authorization platform, such as an operating system or database application. By allowing a user to log in as a particular identity, JAAS can call upon its log-in modules to ensure that the authentication information is valid and thus use the Java security policy to determine for which resources the user's principal is authorized.[23]

In the Subject.doAs( ) method and SubjectDomainCombiner class, the J2SE authorization mechanism was extended, rather than replaced. Code that had previously been written to work with J2SE security continues to work, and new code designed to use JAAS can use the new authentication and authorization frameworks.

An EJB container can use JAAS for both authentication and method-level authorization. By using the JAAS subject and principal structures, it is possible to accommodate the EJB role-based authorization scheme in a straightforward manner.

For example, when a client requests an action on an EJB object, the WAS receives the request and the EJB container extracts principal identity information. After the EJB container performs a JAAS log in with the Principal object, the EJB container receives one or more authenticated Subject objects' Principal objects and credentials. The EJB container issues a Subject.doAs( ) call to establish the credentials before invoking the EJB object. Through the EJB object's internal actions (e.g., through the ORB), AccessController.checkPermission( ) is called to determine whether the caller is authorized for the method.

The Subject.doAs( ) method can execute a privileged action under a desired identity. This method, though useful for delegation, should be used with caution by enforcing a principle of "least privilege." The actions associated with the method should be limited to the ones requiring the privilege of the associated subject and should not be extended to perform more. If the subject represents a privileged user (e.g., the root), the action associated should be limited to that requiring the privileged authority.

Servlets have come of age as Java-based alternatives to CGIs and are essential objects in the WAS environment.

Servlet security

Servlets have come of age as Java-based alternatives to CGIs and have become essential objects in the WAS environment. Servlets provide the same functionality as CGIs, such as creating HTML layouts or servicing queries, but do so by exploiting Java technologies. As servlets become more general-purpose, administrators must consider their security aspects.

Servlets are platform-independent Java classes loaded dynamically into and run by a Web server.[24] Thus servlets interact with Web clients to provide access to enterprise back-end services. To secure back-end applications and enterprise data, it is essential to protect the entry point from security attacks. As servlet invocations are based on the HTTP request/response protocol, any security policy for protecting Web resources (servlets, JSP pages, and HTML files) should consider the HTTP security infrastructure as well.

The servlet container determines which server to invoke based on its internal configuration, calling the servlet with objects representing the request and response.[24] Security can be handled either by the servlet container or by the servlet itself. Declarative security policies are handled by the servlet container based on the WAS security configuration. Servlet writ-

ers can handle application-level security using security APIs provided by the servlet run-time environment.

In a typical scenario, a Web client (e.g., a Web browser) accesses a Web server and makes an HTTP request. If the request is for a servlet, the Web server delegates the request to the servlet container to handle and to send back a response. The container determines the level of protection required, and invokes a servlet if the requesting user has the required permissions to invoke the requested method. If the servlet is protected, the container will manage the authentication process based on the authentication policies. Once the user is authenticated, authorization policies are checked to verify that the user has the necessary privilege to invoke the requested method.

Authentication. A Web client can authenticate a user to a Web server using HTTP (basic or digest), HTTPS, or form-based authentication.

HTTP basic authentication. HTTP basic authentication is a widely used authentication mechanism. The servlet container issues an HTTP "401" authentication challenge and the user is prompted for a user ID and password by the Web browser. The WAS considers the authentication to be successful if the user registry confirms that the password is associated with the user ID provided. For example, if an LDAP directory is the user registry, then authentication succeeds if an LDAP bind request to the directory using the user's DN is successful and the password matches.

HTTP digest authentication. HTTP digest authentication ensures that a clear text password does not flow with the request. In this mode, a digest version (one-way hash) of the password is provided to the Web server. The WAS must be able to obtain the original password so that it can compute the hash and compare it with the digest. This may not always be possible, because the user registry dictates whether the clear text password can be accessed.

HTTPS client authentication. This mechanism makes use of the public key infrastructure in authenticating a user. A Web server can be configured to require mutual authentication when a request is issued over HTTPS (secure HTTP). Not only will the Web server provide its server-site digital certificate, but it can also require the Web client to present its client digital certificate. When such a session is established, it proves that the Web server trusts the certificate authority (CA) that issued the certificate and that the client digital certificate belongs to the user.

It is left to the WAS to perform any mapping from the digital certificate contents to a user in its user registry. WAS could utilize LDAP, an operating system registry, or another representation (such as a SecureWay directory) for its user registry. There are some basic ways in which this mapping can be accomplished:

1. The certificate can be compared to the one stored for the user in the registry.
2. The contents of the certificate can be matched against the attributes of the user entry in the registry.

Form-based authentication. In the case of basic authentication, a Web browser prompts the user for ID and password using a dialog window. However, for usability it is often desirable to present the user with an HTML form where the authentication values can be entered and the request submitted. When the form is submitted, the WAS will extract these values and perform appropriate authentication.

The Java Servlet API Specification version 2.2 standardizes this approach by specifying the names of the form fields and the associated action. The log-in form must contain fields for the user to specify user name and password. These fields must be named j_username and j_password, respectively.[24] For the authentication to proceed appropriately, the value of the action field on the log-in form must always be j_security_check.

At the end of a successful authentication, the request is associated with a user in the WAS user registry. The WAS associates the user with the security context of the servlet invocation.

Authorization. After a successful authentication, the WAS consults security policies to determine if the user has the required permissions to complete the requested action on the servlet. This policy can be enforced using the WAS configuration (declarative security) or by the servlet itself (programmatic security), or a combination of both.

In the case of declarative security, the WAS will perform an authorization check to determine if the user has the permission to perform the requested action on the servlet. This process takes into account the user's privilege attribute information, such as user

ID or group memberships. It will also take into account any security policies declared from the servlet container, such as the role relationships and the permissions granted to these roles. The authorization policy may be defined at the granularity of the Web application of which the servlet is a part.

The J2EE authorization model is based on security roles.[2] A user or a user group is associated with one or more roles in the application domain. The WAS determines the authorization policy for a user based on the authorization associated with a security role and the association between the user and the security role.

In the case of programmatic security, a servlet can obtain information about the authenticated user using the servlet APIs. The servlet can then make business decisions based on the user's name or the associated role. Programmatic security consists of the following methods of the HttpServletRequest interface:[24]

- getRemoteUser( ). This method returns the authenticated user name, if the user is authenticated; otherwise it returns the keyword anonymous.[25]
- isUserInRole( ). This method queries the underlying security mechanism of the container to determine if a particular user is in a given security role.
- getUserPrincipal( ). This method returns a java.security.Principal object.

Delegation. All method invocations "downstream" from a servlet are performed on behalf of the user invoking the servlet. The WAS may provide a set of programmatic APIs where the servlet can set up an identity on the security context programmatically, so that the downstream method requests can be performed under that identity. JAAS may be used to handle such delegation requirements if the downstream calls can be made using the Subject.doAs( ) method.

Security considerations. HTTP basic authentication is a popular way to authenticate servlets. However, the user ID and password flows "over the wire" in (near) clear text. Anyone who can obtain the data flowing over the wire can also obtain the password with minimal effort by decoding the base64-encoded user ID and password pair.[26] Also, the target Web server is not authenticated. Therefore, if basic authentication is the desired mechanism, access to the protected resource should occur over HTTPS to ensure that the target server is authenticated and that no "man-in-the-middle" attacks will be able to obtain the confidential data flowing between the browser and the Web server.

Java Cryptography Architecture and Java Cryptography Extension

The public key infrastructure has solved many vexing problems for today's enterprise environments, from authentication to the publication of user information. Just as with other successful standards, Java technology needed to incorporate public key technologies into its functionality in a standard and extendable way.

From the Java 2 Platform, Standard Edition, version 1.2 onward, Java technology has provided general-purpose APIs for cryptographic functions, collectively known as the Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE). This section introduces the JCA and JCE and shows how these layers can be exploited at a higher level by JSSE, PKCS, and S/MIME in the WAS environment.

Java Cryptography Architecture framework. JCA is a framework for accessing and developing core cryptographic functionality for the Java platform. It encompasses the parts of the J2SE security API related to cryptography. JCA was designed around two principles. The first principle is implementation independence and interoperability. The second principle is algorithm independence and extensibility.

Implementation independence is achieved using a provider-based architecture. For each cryptographic service, such as message digest and digital signature, JCA supplies a service provider interface (SPI) abstract class as part of the API. Examples of SPI classes are MessageDigestSpi and SignatureSpi. The term *cryptographic service provider* (CSP) refers to a package or a set of packages that supply a concrete implementation of a subset of the SPI classes that are part of the Java security API. In other words, these packages must implement one or more cryptography services. This allows providers to be updated or replaced in a manner that is transparent to applications. This allows faster or more secure versions to be installed when they become available.

Implementation interoperability means that various implementations can work with each other, use each other's keys, or verify each other's signatures. This means, for example, that for the same algorithms,

a key generated by one provider is usable by another, and a signature generated by one provider is verifiable by another.

Algorithm independence is achieved by defining types of cryptographic services and classes that provide the functionality of these cryptographic services. Such classes are called engine classes. Examples include the MessageDigest, Signature, and KeyFactory classes. For each engine class, there is a corresponding SPI class.

Algorithm extensibility means that new algorithms that fit in one of the supported engine classes can easily be added.

The security-related classes of JCA shipped with J2SE provide for only message digest and digital signature. This allows developers to provide their own classes for public key pairs, certificates, and other primary security objects. Developers can also perform reliable authentication, which, in turn, can be used as a basis for implementing access controls that relax restrictions. However, J2SE does not provide the general-purpose encryption needed to send confidential data.

JCE extends the cryptography-related classes shipped with J2SE. The JCE package uses the same structure as JCA, with engine classes that expose the algorithms in a generic way and SPI abstract classes that are subclassed to provide a concrete implementation of the cryptographic service they represent. JCE provides engine classes for symmetric key encryption and for generating and manipulating the secret keys that such algorithms require.

Java cryptography standard API. JCE extends the JCA API to include classes for encryption, key exchange, and message authentication code (MAC). Together, JCE and the cryptography aspects of J2SE provide a platform-independent cryptography API.

Fundamentals of JCA. The JCA framework is based on the concepts of *engine*, *algorithm*, and *provider*.

Engine is the term used to depict an abstract representation of a cryptographic service that does not have a concrete implementation. A cryptographic service is always associated with a particular algorithm or type, and it can have one of the following functions:

- To provide cryptographic operations (like those for digital signatures or message digests)
- To generate or supply the cryptographic material (keys or parameters) required for cryptographic operations
- To generate data objects (key stores or certificates) that encapsulate cryptographic keys (that can be used in a cryptographic operation) in a secure fashion

Message digests and signatures are examples of engines. JCA encompasses the cryptography-related classes of the J2SE security package, including the engine classes. Users of the JCA API request and utilize instances of the engine classes to carry out corresponding operations.

An algorithm can be looked upon as an implementation of an engine. For instance, the Message Digest 5 (MD5) algorithm is one of the implementations of the message digest engine. The internal implementation of the MD5 algorithm can differ depending on the vendor that provides the MD5 algorithm class.

Each set of algorithm classes from a particular source (a provider) is managed by an instance of the java.security.Provider class. Installed providers are listed in the java.security properties file, which contains Java security configuration entries.[17] A provider does not know the actual implementation of the cryptographic algorithms. Rather, a provider knows which algorithm class can provide a particular algorithmic implementation.

Java Cryptography Extension. JCE has been provided as an extension to the Java platform. It supplies a framework and implementations for encryption, key generation, key agreement, and MAC to supplement the interfaces and implementations of message digests and digital signatures provided by J2SE.

The provider architecture of JCA aims to allow algorithm independence. The design principles behind JCE share this philosophy of implementation and algorithm independence. In addition to making it possible to use newer algorithms for generating keys, JCE also introduces new interfaces and classes that facilitate the implementation of these concepts.

JCE provides for symmetric bulk key encryption through the use of secret keys. With a secret key, the sender and receiver share the same key to encrypt as well as to decrypt data. Associated concepts of MAC and key agreements support symmetric bulk encryption and symmetric stream encryption.

Security considerations. Within the WAS environment, the security server is the primary user of JCA to validate signatures on transactions and certificates. However, all primary and secondary objects within the WAS environment can exploit the capabilities of JCA and JCE. For example, an EJB object could sign data using a JCA Signature instance or encrypt data using a JCE Cipher instance. The JSSE and PKCS sections demonstrate how JCA and JCE functionality can be combined to create more complex technologies. JCA plays a fundamental role to any Java application that implements public key security.

Export control restrictions by the United States Commerce Department currently prohibit such a cryptography framework from being exported outside the United States or Canada, unless appropriate mechanisms have been implemented in the framework that allow the framework to control the type of encryption algorithms and their cryptographic strength available to applications. The lack of exportability has significantly affected the usability and deployment of JCE.

Exportability has been accorded to the new version, JCE version 1.2.1, in which JCE, not its CSPs, enforces export restrictions. IBM's distribution of J2SE contains an implementation of JCE with a suite of commonly used cryptographic functions.[27]

Java Secure Socket Extension

Through the cryptographic APIs provided in J2SE and in the standard JCE package, developers can invoke cryptographic functions from within Java code. However, most developers and application designers would prefer to use ready-built cryptographic protocols, rather than having to create them from the basic elements of encryption and digital signatures. *Secure sockets layer* (SSL) is the most widely used protocol for implementing encrypted channels over the Web.[28] Almost all e-business Web sites use SSL to ensure that their own or their clients' personal information, such as a credit card number, can flow securely over the unsecured Internet. This section describes SSL, its implementation in Java code through JSSE, and its use among objects and entities in the WAS environment.

What is SSL? SSL is a standard protocol proposed by Netscape Communications Corporation for enabling secure transmission on the Web.[29] The primary goal of the SSL protocol is to provide privacy and integrity between two communicating parties. As the name suggests, SSL provides a secure form of the standard TCP/IP (Transmission Control Protocol/Internet Protocol) sockets protocol. In fact, SSL is not a drop-in replacement because the application has to specify additional cryptographic information. Nonetheless, it is not a large step for an application that uses regular sockets to convert to SSL. Although the most common implementation of SSL is for HTTP, several other application protocols have also been adapted.

SSL has two security aims:

1. To authenticate the server and the client using public key signatures and digital certificates as required
2. To provide an encrypted connection for the client and server to exchange messages securely

The SSL connection is private and reliable, using encryption after an initial handshake to define a secret key. The SSL connection also maintains message integrity checks. Note that in SSL, symmetric cryptography is used for data encryption, while asymmetric or public key cryptography is used to authenticate the identities of the communicating parties and encrypt the shared encryption key when an SSL session is established. This way, the shared encryption key can be exchanged in a secure manner, and client and server can be sure that only they know the shared secret key. In addition, the client and server have the advantage of encrypting and decrypting the communication flow with a single encryption key, which is much faster than using asymmetric encryption.

In this way, SSL is able to provide:

- Privacy. The connection is made private by encrypting the data to be exchanged between the client and the server. In other words, only they can decrypt and make sense of the data. This allows for secure transfer of private information such as credit card numbers, passwords, secret contracts, and the like.
- Data integrity. The SSL connection is reliable. The message transport includes a message integrity check based on a secure hash function. There is practically no possibility of data corruption without detection.
- Authentication. Optionally, the client can authenticate the server and an authenticated server can authenticate the client. This means that, when authentication is required, the information is guaranteed to be exchanged only between the intended

parties. The authentication mechanism is based on the exchange of digital certificates.
- Nonrepudiation. Digital signatures and certificates together imply nonrepudiation. This establishes accountability of information about a particular event or action to its originating entity, and the communications between the parties can be proved later.

The SSL protocol can use different digital signature algorithms for authenticating the communicating parties. SSL provides various key exchange mechanisms that allow the sharing of secret keys used to encrypt the communicated data. Furthermore, SSL can make use of a variety of algorithms for encryption and hashing. SSL cipher suites describe the cryptographic options defined by SSL and whether or not the cipher strength can be exported outside the United States or imported to other countries.

JSSE package. Fetching data using the URL technique is a very simple approach, but it limits the Java program capabilities, because client/server communications can exploit only the capabilities offered by CGI (or another, similar, server interface). Even if this is adequate for the function, it imposes some performance overhead. A direct SSL socket connection between client and server allows more sophisticated and responsive applets to be created. This can be done by using a package that provides SSL function.

Java Secure Socket Extension (JSSE) is a standard extension to the Java platform. It consists of a set of packages that enable secure Internet communications. JSSE implements a Java version of SSL and Transport Layer Security (TLS) protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication. Using JSSE, developers can provide for the secure passage of data between a client and a server running any application protocol (such as HTTP, Telnet, NNTP [Network News Transfer Protocol] and FTP [File Transfer Protocol]) over TCP/IP.

Once an extension providing SSL support, such as JSSE, has been installed on a Java system, an SSL-protected client/server communication can be implemented in Java code using the SSL protocol with the SSL Java APIs.[17]

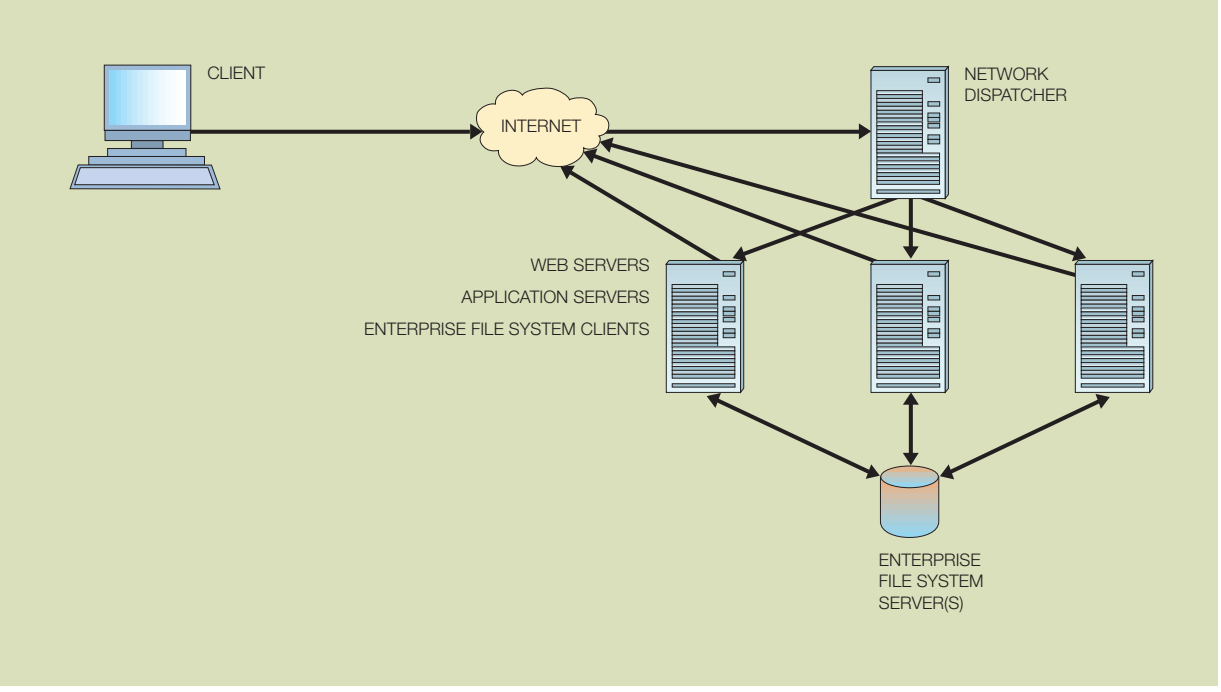
JSSE in a WAS environment. In the WAS environment, JSSE has an immediate application—to provide enhanced authentication. For example, if a digital certificate is the authentication data and if the user can establish an SSL connection between the Web client and Web server, the WAS trusts that the certificate belongs to the user. The user information present in the certificate (for example, the distinguished name) is then mapped onto the user registry to find a matching user entry. The certificate challenge implies that the Web server is configured to perform mutual authentication over SSL.[30,31] In other words, both the client and the server must present a certificate to establish the connection. In other circumstances, the authentication policy can include a *secure channel constraint*. In other words, an SSL session may be required along with a challenge mechanism to provide data confidentiality and integrity for the information flowing between the server and a client.[32]

SSL may also be used when single sign-on is required. A way to implement single sign-on in a WAS environment is by setting a "cookie" on the client system.[33] The safest approach in this case is to ensure that an SSL connection is established every time the cookie is created and stored in the browser. An SSL connection prevents anyone from acquiring the cookie from the network flow.

The SSL API provided by the JSSE standard extension allows the creation of a direct SSL socket connection between a Java client and a Java server.[17] This way, client and server applications do not need to rely on the support of a Web browser and a Web server to exploit the advantages offered by SSL in terms of security. In particular, SSL support is useful when the client application is the WAS itself. Many times, two application servers need to exchange confidential information. For example, there may be clustered WASs sharing the same Web contents and "load balanced" by a dispatcher machine, as shown in Figure 6.[34–36] In this case, client session information needs to be shared across the nodes or authentication information will be lost. To obtain data confidentiality and integrity on untrusted networks, SSL can be turned on.[30,31]

Security considerations. The history of the World Wide Web is based on pragmatism. For example, no one would argue that sending uncompressed ASCII (American National Standard Code for Information Interchange) text data on sessions that are set up and torn down for every single transaction is efficient in any way. However, this is what HTTP does, and it is very successful. The reason for its success is that it is simple enough to allow many different systems to interoperate without problems of differ-

Figure 6 Using SSL to exchange private information in a load-balancing scenario



ing syntax. The cost of simplicity is in network overhead and a limited transaction model.

Using cryptography in Java code offers a similar dilemma. It is possible to write secure applications using a toolkit of basic functions. Such an application can be very sophisticated, but it will also be complex. Alternatively, SSL URL connections offer a way to simplify the application, but at the cost of application function. SSL Java packages, such as JSSE, provide a middle way, retaining simplicity but allowing more flexible application design for applications in the WAS environment.

When using JSSE and cookies to implement a single sign-on solution, the WAS administrator should specify a time period after which the user should be required to reauthenticate. This restricted time period prevents replay attacks using the cookie. To ensure that cookies are not persistent, so that off-line attacks can be prevented, the cookie should be set to expire at the end of the browser's session. A WAS should also provide a mechanism by which the single sign-on cookie can expire programmatically so that in a kiosk scenario a user leaving a terminal can explicitly log out and not worry about the session being used by subsequent users. Indicating that a cookie should be stored only for the session also eliminates the security risk of having the cookie values stored on a hard drive, for example.

To prevent replay attacks that retrieve the cookie from the network flow, developers should restrict the cookie to flow only over HTTPS protocol. Setting the "secure" field of the cookie ensures that the cookie will flow only over SSL connections, which are used by HTTPS.

PKCS and S/MIME

Just as JSSE leveraged JCA and JCE to create and maintain secure connections between entities, there must also be a way for developers to represent objects that use public key technologies in a standard fashion. The industry has adopted PKCS and S/MIME as the standard techniques through which these types of security objects can be created, packaged, and delivered.[37] Many Java implementations of these standards have emerged, but what are they, and how can they address the security needs of the WAS environment? This section introduces those elements of the

standards that can be exploited by objects in the WAS environment.

First developed by RSA Data Security, Inc. and a consortium of companies in 1991, PKCS has evolved to encompass a wide set of functionality from encryption to the use of smart cards and object packaging. At the core of the standards is the use of public key technologies, whether these technologies are explicitly used in signing data or implicitly used in the bundling of certificate information within a token, for example.

By overlaying PKCS over the WAS environment, the Cryptographic Message Syntax Standard (PKCS #7) consolidates the transactions and objects used by WAS and its associated servers. RFC 2315 contains information about PKCS #7 version 1.5[38] with enhancements to the standard detailed in RFC 2630.[39]

Earlier sections of this paper have identified the importance of signing and encryption for transactions from objects, such as EJB objects within the WAS, and between client and server entities in a WAS environment, such as a Java client and a Java server using JSSE. Recipients of a signed message or transaction can verify the contents and gain assurance that the message was not changed in transit and that it actually originated from the sender. Encryption helps protect the message from prying eyes and ensures that only the intended recipient can unlock its contents. Signing and encrypting can be used alone or in combination to provide a transaction hardened against security threats.

When using JCA and JCE objects, developers are compelled to maintain different initialization attributes and objects to accomplish their tasks, such as the signature algorithm, the contents to be signed, and signature bytes. These objects can become scattered and do not lend themselves to a free and standard exchange across a heterogeneous environment. PKCS #7 defines how objects, such as SignedData objects, should be packaged so that the creator of the object includes all of the subobjects and attributes that are needed for the recipient to verify, decrypt, or perform some other action upon the aggregate object.

The PKCS standards use the Abstract Syntax Notation (ASN.1)[40] to formally describe the objects, for example, whether or not an attribute is optional or if it should consist of a set of other objects. ASN.1 is an international standard for specifying data used in communication protocols. It is a powerful and complex language—its features are designed to describe accurately and efficiently communications between homogeneous or heterogeneous systems. Since Java PKCS implementations model objects that represent the ASN.1 notation, these objects need to be encoded in a standard form that senders and recipients agree upon. For PKCS, encoding is performed using the Distinguished Encoding Rules (DER), a subset of the Basic Encoding Rules (BER) described in ASN.1.

Signing and encrypting can be used, alone or in combination, to harden a transaction against security threats.

Two classes of objects in PKCS #7 play an active role in WAS environments, SignedData and EnvelopedData. As the name suggests, SignedData objects represent a signed message and consist of a number of essential objects, including the signature algorithm, signer's certificate, encapsulated contents, and signer information, such as the signature bytes and the time the signer generated the signature. Signature algorithms include MD5withRSA and SHA1withDSA.[41] PKCS cleverly represents SignedData objects; the object definition allows only a single instance of the encapsulated contents to be included, whereas the contents can have many signers with information for each signer stored as a separate subobject.

The EnvelopedData object hides the specified contents by introducing encryption into the mix. Java PKCS implementations first generate a secret key for the designated cipher algorithm as part of constructing an EnvelopedData object. Cipher algorithms include RC2, DES, and TripleDES.[41] Both the contents to be encrypted and the secret key are passed into the cipher object, which calculates and returns the encrypted bytes. Since the secret key cannot be sent "in the clear" with the encrypted contents, the EnvelopedData object goes one step further, encrypting the secret key with the public key of each recipient. It uniquely encrypts the secret key for each recipient, since each recipient's public key is different. When each recipient receives the Envel-

opedData object, the private key is used to decrypt the secret key and the decrypted secret key is used to decipher the encrypted message.

The introduction and progression of the S/MIME[42] standards helps marry the benefits of encapsulated PKCS objects with a popular mechanism for representing different content types across the Internet. For example, MIME messages with a content type of text/plain are a popular way for users to send general messages to each other. Since these messages travel across an unsecured network—the Internet—MIME messages lacked uniform security until the incorporation of S/MIME. The S/MIME standards bring to MIME the security benefits of PKCS—protecting message integrity, authenticating the originator of messages, and enciphering the content of messages. All of the popular e-mail applications support S/MIME, ensuring that users can send messages securely with all the benefits of authentication, integrity, and encryption.

S/MIME allows senders to encode PKCS #7 SignedData and EnvelopedData objects within the message.[43] Since DER-encoded binary data cannot be sent in their raw form, the binary data are base64 encoded, which results in 7-bit US-ASCII text[44] that can be sent in an Internet message. S/MIME recipients take the message and work backward, decoding the base64 message, decoding the DER-encoded message, and instantiating the encoded object. In addition to the PKCS #7 SignedData and EnvelopedData objects, S/MIME also supports PKCS #10 CertificationRequest objects, which allow entities to request a certificate from a CA. The SignedData object is versatile enough that CAs can respond to a PKCS #10 request by sending back the issued certificate in a SignedData object contained within a S/MIME message.

PKCS and S/MIME in the WAS environment. PKCS and S/MIME play a key role in guaranteeing the authenticity of a message and hiding sensitive contents between client and server and between server and server entities within the WAS environment. As part of the authentication architecture, every entity has a certificate associated with it, whether a client whose certificate was retrieved by the security server from the LDAP database or a Web server with an embedded certificate. This certificate forms the foundation for sending and receiving secure messages or transactions.

Transactions within the WAS environment can be divided into two levels, *primary* and *secondary*. Primary-level transactions occur between the major entities, such as between client and server or server and server. Security between the major entities can involve the security server and could be initiated by a security plug-in in a Web browser or security collaborator in a WAS. JSSE is a major component of security at the primary level. Transactions at the secondary level occur within a major entity, such as within an EJB object that is invoked by a WAS. The secondary objects still interact with the security server for authentication and authorization, but they are able to invoke the security requests at their level. PKCS can play roles at either level, but it is especially important within the secondary level.

Consider a secondary security example. A WAS Java application client sends a request to the WAS to invoke an EJB object. The security collaborator within the WAS ensures that the client has the authorization to utilize the EJB object. In this example, the EJB object requires secondary security to add to the user registry the set of certificates that were passed by the client to the EJB object.

Signing transactions with PKCS. The EJB object encapsulates the request as part of a PKCS #7 SignedData object. To create the object, the EJB object constructs a SignedData object with the EJB object's certificate, private key, signature algorithm, and the attributes of the request as input. The resulting SignedData object contains the contents, the EJB object's certificate, and signer information containing the signature. (The signature algorithm requires the private key to compute the signature bytes, but the SignedData object never bundles the private key as part of its attributes.) With a socket to the security server, the EJB object can DER-encode the SignedData object and send it to the security server as part of the request.

Upon receipt, the security server decodes the DER-encoded SignedData object to prepare for verification. Before the verification method can be called, however, the security server must first ensure that the signing certificate can be trusted. The security server performs this check by tracing the EJB object's certificate to a trusted root CA, stored within the user registry. Assuming the certificate path holds true and the signing certificate is still valid, the security server provides the signing certificate as an input to the verification method. The original contents—the set of certificates to add—need not be

passed to the method since they are already part of the SignedData object.

In reply, the verification method returns a Boolean response. A positive response gives the security server the confidence that the message was not changed in transit from the EJB object and that it truly originated from that object. The security server can now add each of the content certificates within the SignedData object to the user registry.

Encrypting transactions with PKCS. In the scenario just described, it would have been just as easy for the EJB object to encrypt the message. In this case, the EJB object would require access to the security server's certificate. To construct an EnvelopedData object, the EJB object would use the message alone or the SignedData object along with the security server's certificate and the encryption parameters. The encryption parameters would include the algorithm and the key size so that construction of the EnvelopedData object would automatically generate the secret key "under the covers." As described earlier, the secret key is encrypted with the public key from the security server's certificate. The EJB object would send the created EnvelopedData object through the same routes that were described earlier for the SignedData object.

The security server receives the EnvelopedData object and decodes it. The security server passes its certificate and private key to the decrypt method of the EnvelopedData object. The decrypt method uses the security server certificate to determine which encrypted secret key should be decrypted, since the EnvelopedData object may contain many uniquely encrypted secret keys, one for each recipient. The decrypt method uses the private key associated with the security server certificate to unlock the encrypted secret key associated with the certificate. Next, the method uses the secret key to decrypt the cipher text of the message.

Note that inclusion of the SignedData object within the EnvelopedData object provides the additional benefit of signing the primary contents, which are encapsulated within the SignedData object. After decrypting the message, the security server could verify the contents of the message using this compound object. Thus with an EnvelopedData object wrapped around a SignedData object, the sender and receiver gain the benefits of three main security facets—integrity, authentication, and encryption.

Asynchronous transactions with S/MIME. Instead of sending the request and the DER-encoded PKCS #7 object as binary data across a socket, S/MIME provides a standard way for these objects to be sent using a universal format. As a standard extension to the Java run-time environment (JRE), the JavaMail[45] package could be extended to provide security to MIME messages. With S/MIME, clients and servers, as well as secondary objects (like EJB objects) within the WAS environment can readily send and queue up asynchronous requests.

Security considerations. As they have matured over the years, the PKCS and S/MIME standards have been examined for completeness and security. Although powerful, they do not address authorization, an important part of security. Entities that hide private keys are the only ones with authorization to the key, and this controlled access provides an explicit authorization to decrypt an encrypted message. However, PKCS does not address who can receive a signed message, for example, or who can act upon the contents of a signed message. A WAS or a secondary entity, using responses from the security server, must determine whether or not the client has authorization to perform the signed action, and authorization typically stems from a policy established by a physical entity, such as an administrator. Additionally, administrators and developers must determine the appropriate type of cryptography for their WAS environment.[46]

The blend of JCA and JCE technologies gives rise to key agreement algorithms within the SSL protocol. Within the Java environment, the JSSE implements SSL and allows two parties to authenticate and send secure transactions to each other. As discussed earlier, JSSE provides a sanctioned and secure communication link between two entities, and this technology is the fundamental way in which signed and encrypted communication can occur synchronously between primary entities in the WAS environment. However, if the overhead of SSL is not needed and if communication can occur asynchronously, PKCS and its companion S/MIME can readily fill the security needs of a WAS environment. The encapsulation of PKCS objects allows these objects to be easily stored and retrieved for later evaluation or processing.

Conclusion

The WAS environment pulls together many different technologies to service the enterprise. Because of the heterogeneous nature of the client and server

entities, Java technology is a natural choice for administrators and developers. However, to service the diverse security needs of these entities and their tasks, many Java security technologies must be used, not only at a primary level between client and server entities, but also at a secondary level, from served objects, such as EJB objects. By using a synergistic mix of the various Java security technologies, administrators and developers can make not only their Web application servers, but their WAS environments secure as well.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc., Object Management Group, or Microsoft Corporation.

Cited references and notes

1. J. Hunter and W. Crawford, *Java Servlet Programming*, O'Reilly & Associates, Sebastapol, CA (1998).
2. B. Shannon, M. Hapner, V. Matena, J. Davidson, E. Pelegri-Llopart, and L. Cable, *Java 2 Platform, Enterprise Edition: Platform and Component Specifications*, Addison-Wesley Publishing Co., Reading, MA (2000).
3. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Publishing Co., Reading, MA (1983).
4. G. Krasner, *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley Publishing Co., Reading, MA (1983).
5. Acronym definitions: LDAP is Lightweight Directory Access Protocol; RACF is Resource Access Control Facility.
6. EJB 1.1 specification, http://java.sun.com/ejb.
7. R. Monson-Haefel, *Enterprise JavaBeans*, O'Reilly & Associates, Sebastapol, CA (1999).
8. Java Naming and Directory Interface specifications, http://java.sun.com/products/jndi/.
9. CORBA security specification, http://www.omg.org.
10. B. Blakley, *CORBA Security: An Introduction to Safe Computing with Objects*, Addison-Wesley Publishing Co., Reading, MA (1999).
11. Java language mapping to Object Management Group's Interface Definition Language, http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm.
12. RMI over IIOP, http://java.sun.com/products/rmi-iiop/.
13. OMG Common Secure Interoperability, version 2 specifications, http://www.omg.org/.
14. WebSphere security overview, http://www-4.ibm.com/software/webservers/appserv/security.pdf.
15. N. Nagaratnam and D. Lea, "Secure Delegation for Distributed Object Environments," *Proceedings, USENIX Conference on Object-Oriented Technologies and Systems*, Santa Fe, NM (April 27–30, 1998), pp. 101–116.
16. N. Nagaratnam, *Practical Delegation for Secure Distributed Object Environments*, Ph.D. dissertation, computer engineering degree program, Syracuse University (April 1998).
17. M. Pistoia, D. F. Reller, D. Gupta, M. Nagnur, and A. K. Ramani, *Java 2 Network Security*, Prentice Hall, Englewood Cliffs, NJ (2000).
18. L. Gong, *Inside Java 2 Platform Security: Architecture, API Design, and Implementation,* Addison-Wesley Publishing Co., Reading, MA (1999).
19. V. Samar and C. Lai, "Making Login Services Independent of Authentication Technologies," http://java.sun.com/security/jaas/doc/pam.html/.
20. GSS-API Security Attribute and Delegation Extensions, The Open Group.
21. RFC 2222, Simple Authentication and Security Layer (SASL).
22. N. Nagaratnam, B. Maso, and A. Srinivasan, *Java Networking and AWT API SuperBible: The Comprehensive Reference for the Java Programming Language*, Macmillan USA, Indianapolis, IN (1996).
23. B. Rich, A. Nadalin, and T. Shrader, "All that JAAS: An Overview of the Java Authentication and Authorization Services," *IBM Developer Connection* (March 2000); http://www.developer.ibm.com/devcon/mag.htm.
24. Java Servlet API Specification version 2.2, http://java.sun.com/products/servlet.
25. IBM WebSphere Standard/Advanced 3.02 Security Overview, http://www-4.ibm.com/software/webservers/appserv/security.pdf.
26. MIME (Multipurpose Internet Mail Extensions), http://www.ietf.org/rfc/rfc1521.txt?number=1521.
27. IBM J2SE specifications, http://www.ibm.com/java/.
28. A. O. Freier, P. Karlton, and P. C. Kocher, *SSL 3.0 Specification*, Technical Report, Netscape Communications Corporation (November 1996); available at http://home.netscape.com/eng/ssl3/index.html.
29. SSL 3.0 Specifications, http://home.netscape.com/eng/ssl3/.
30. B. Nusbaum, M. Pistoia, G. Rochester, and T. Liu, *Network Computing Framework Component Guide*, SG24-2119-00, IBM Corporation (1997).
31. B. Nusbaum, M. Pistoia, G. Rochester, and T. Liu, *IBM Network Computing Framework for e-business Guide*, SG24-5296-00, IBM Corporation (1998).
32. M. Pistoia, K. Kojima, and N. Raghu, *Internet Security in the Network Computing Framework*, SG24-5220-00, IBM Corporation (1998).
33. RFC 2109, HTTP State Management Mechanism, http://www.ietf.org/rfc/rfc2109.txt?number=2109.
34. M. Pistoia and C. Letilley, *IBM WebSphere Performance Pack: Load Balancing with IBM SecureWay Network Dispatcher*, SG24-5858-00, IBM Corporation (1999).
35. M. Pistoia, T. Menner, C. Milligan, and B. G. Pham, *IBM WebSphere Performance Pack: Web Content Management with IBM AFS Enterprise File System*, SG24-5857-00, IBM Corporation (1999).
36. M. Pistoia, V. Iovine, and S. Pischedda, *IBM WebSphere Performance Pack Usage and Administration*, SG24-5233-00, IBM Corporation (1998).
37. T. Shrader, B. Rich, and A. Nadalin, *Java and Internet Security*, iUniverse, http://www.iuniverse.com (2000).
38. RFC 2315, PKCS #7: Cryptographic Message Syntax Version 1.5, ftp://ftp.isi.edu/in-notes/rfc2315.txt.
39. RFC 2630, Cryptographic Message Syntax, ftp://ftp.isi.edu/in-notes/rfc2630.txt.
40. B. S. Kaliski, Jr., *A Layman's Guide to a Subset of ASN.1, BER, and DER*, RSA Laboratories (November 1993).
41. B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd Edition, John Wiley & Sons, Inc., New York (1995).
42. RFC 2311, S/MIME Version 2 Message Specification, ftp://ftp.isi.edu/in-notes/rfc2311.txt.
43. T. Shrader, A. Nadalin, and B. Rich, "Understanding Cryptographic Messages in e-business," *IBM DeveloperToolbox Technical Magazine* (March 2000); http://www.developer.ibm.com/devcon/mag.htm.
44. ASCII was first defined by the American National Standards

Institute (ANSI) in ANSI Standard X3.4 in 1968. The ASCII code is also described in ISO 636 (1973) and CCITT V.2, which calls the standard IA5 (International Alphabet #5). ASCII is a 7-bit code, resulting in a maximum of 128 characters.
45. JavaMail version 1.1.3 specifications, http://www.javasoft.com/products/javamail/index.html.
46. T. Shrader, "Choosing the Right Cryptography for Your e-business Application," *IBM DeveloperToolbox Technical Magazine*, on-line edition at http://www.developer.ibm.com/devcon/mag.htm.

General references

For information on the Pluggable Authentication Module, see http://java.sun.com/products/jaas/.

RFC 1510, The Kerberos Authentication System, http://info.internet.isi.edu/in-notes/rfc/files/rfc1510.txt.

C. Neuman and T. Ts'o, "Kerberos: An Authentication Service for Computer Networks," *IEEE Communications* **32**, Number 9, 33–38 (September 1994).

Accepted for publication October 10, 2000.

Larry Koved *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: koved@us.ibm.com).* Mr. Koved is a research staff member in the Network Security and Cryptography department and a Java security architect for IBM. His primary interests are the security of mobile code, component software, and object-oriented languages. He is a member of ACM's SIGSAC and SIGCHI and of the IEEE Computer Society.

Anthony Nadalin *Tivoli SecureWay Business Unit, 9442 Capital of Texas Highway North, Arboretum Plaza One, Suite 500, Austin, Texas 78759 (electronic mail: drsecure@us.ibm.com).* Mr. Nadalin is the lead architect for the IBM Java Security project. As senior architect, he is responsible for infrastructure design and development across IBM. He serves as the primary security liaison to Sun Microsystems' JavaSoft Division for Java security design and development collaboration.

Nataraj Nagaratnam *IBM Application & Integration Middleware Division, P.O. Box 12195, 3039 Cornwallis Drive, Research Triangle Park, North Carolina 27709-2195 (electronic mail: natarajn@us.ibm.com).* Dr. Nagaratnam is the technical lead for the IBM WebSphere™ security team. He received his Ph.D. degree from Syracuse University; his thesis addresses secure delegation in distributed object environments. He has authored and edited books on Java networking and JavaBeans™ and has published in numerous journals and conferences.

Marco Pistoia *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: pistoia@us.ibm.com).* Mr. Pistoia is an advisory Java security specialist in the Network Security and Cryptography department. He has written nine books and taught classes worldwide on all areas of Java, WebSphere, and e-business security. His latest book, *Java 2 Network Security*, was published by Prentice-Hall. Mr. Pistoia's interests are in mobile code security and object-oriented technology.

Theodore Shrader *IBM Software Group, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: tshrader@us.ibm.com).* Mr. Shrader was a feature lead on the IBM Java Security project and currently is the Editor-in-Chief of the IBM *DeveloperToolbox* magazine. He has written numerous patents and technical articles dealing with Internet and Java development, distributed computing, object-oriented design, database architecture, and Web design. He also is a coauthor of an operating systems programming guide published by John Wiley & Sons and the lead author of the *Java and Internet Security* book published by iUniverse and cited in this paper.