The evolution of Java security

by L. Koved
A. J. Nadalin
D. Neal
T. Lawson

This paper provides a high-level overview of the development and evolution of Java™ security. Java is a maturing technology that has evolved from its commercial origins as a browser-based scripting tool. We review the various deployment environments in which Java is being targeted, some of its run-time characteristics, the security features in the current releases of the base technology, the new Java Development Kit (JDK™) 1.2 policy-based security model, limitations of stack-based authorization security models, general security requirements, and future directions that Java security might take. IBM initiatives in Java security take into account our customers' desire to deploy Java-based enterprise solutions. Since JDK 1.2 was entering beta test at the time this paper was written, some operational changes and enhancements may result from industry feedback by the time JDK 1.2 becomes generally available.

The software industry is focused on providing support for developing and deploying mission-critical applications written in Java**. The Java environment encompasses a broad spectrum from enterprise servers to embedded devices. A range of Java-based systems, including JavaOS**, EmbeddedJava**, and PersonalJava**, among others, will become available, providing potentially different levels of underlying services. This situation will result in requirements for varying levels of security strength.

The initial focus of Java security has been in the support of downloaded applets (small programs) within World Wide Web browsers. To a large extent, the security features in Java reflect this heritage. As Java matures, it will increasingly support additional security features to address the needs of the target application environments. Included is the addition of security features generally found in large-scale server applications.

We have seen examples of how e-business (business conducted electronically via the Web) increases a customer's reach by orders of magnitude. As the customer base increases, the absolute magnitude of losses from malicious behavior can become great enough to warrant improved security products deployed in information technology systems. However, should a security exposure become widely publicized, a customer's reputation can become tarnished. IBM's customers demand systems implementations that are nearly flawless and that address the needs of their enterprise. They look to IBM to ensure that risks are known and to respond quickly with action when exposures are uncovered.

Gartner Group's December, 1996, report, *Java—Good Start, but Not Yet Secure*,[1] highlights several areas of concern regarding Java security that are based on earlier versions of the Java Development Kit (JDK**). Realistic expectations are important: no system is 100 percent secure. However, it is crucial to recognize three things about Java security:

1. Java enables a function that was never before commercially deployed on a broad scale: dynamic loading of code from a source outside the system. This important feature aggravates a significant "Trojan horse" security problem. However, it also provides extremely valuable function.
2. Java security was not designed to solve the same problems as the Resource Access Control Facility (RACF*) function in the Operating System/390 (OS/390*) Security Server addresses on Multiple Virtual Storage (MVS*), or similar traditional enterprise security technologies. RACF is designed to protect the enterprise and its resources against hostile users. The security for Java is designed to protect the user's workstation and resources against hostile code.
3. RACF and other traditional enterprise security technologies work well because they assume—and their environments provide—strong operating system integrity as a foundation. Java also assumes this integrity; however, the predominant desktop operating systems do not provide it sufficiently.

Security technologies are needed to prevent or mitigate the following types of threats:

- *Unauthorized resource usage*, including theft of software or CPU usage, corruption of data and software, disclosure of information, and unaccountable action
- *Abuse of privilege*, including misrepresentation of identity, affiliation, value of items exchanged, and entitlement of services; impersonation; fraud; and extortion
- *Malicious code*, including protection from viruses, worms, Trojan horses, and logic bombs
- *Wiretapping*, including active and passive measures
- *Denial of service*, including destruction of resources, saturation of services, and interruptions of communications

Trojan horse and spoofing attacks have been the most common Java security threats publicized to date. Once identified, it has been relatively easy to provide fixes as the majority of attacks have occurred due to implementation errors in browsers or in Java rather than fundamental design flaws.

Java security foundation and evolution. Since initial commercial deployments of Java were in Web browsers, much of the focus of Java security has been in providing features for protecting against hostile applets; that is, against hostile code downloaded from Web sites on the Internet. Java security builds upon three fundamental aspects of the Java run-time environment: the ByteCode Verifier, the Security Manager, and the ClassLoader.

The ByteCode Verifier ensures that downloaded code is properly formatted, that bytecodes (Java Virtual Machine instructions) do not violate the safety restrictions of the language or virtual machine (no illegal data conversions), that pointer addressing is not performed, that internal stacks cannot overflow or underflow, and that bytecode instructions will have the correct typed parameters.[2]

The Security Manager initiates run-time access controls on attempts to perform file I/O and network I/O, create new ClassLoaders, manipulate threads or thread groups, start processes on the underlying platform (operating system), terminate the Java Virtual Machine (JVM), load non-Java libraries (native code) into the JVM, perform certain types of windowing system operations, and load certain types of classes into the JVM. For example, the Java applet sandbox[3] severely constrains downloaded applets to a limited set of functions that are considered to be relatively safe.

The ClassLoader determines how and when applets can load code and ensures that applets do not replace system-level components within the run-time environment.

In addition, a number of features in the Java programming language and run-time environment, including automatic memory management and strong data type safety, facilitate writing safe code.

Through the use of digital signature services provided in JDK 1.1, trusted applets can be treated in a manner similar to applications written in Java. That is, these trusted applets have much greater access to JVM resources than applets that run in the restricted Java sandbox. Improved and much more flexible access control features are the major security addition in JDK 1.2 and are described in greater detail later in this paper.

Today's computing environments have a number of security weak points that are addressed by features available in Java prior to JDK 1.2:

- Strong memory protection—Java removes the possibility of either maliciously or inadvertently reading or corrupting memory locations outside boundaries of the program. As a result, Java applications

SECURITY FUNCTIONS IN JDK RELEASES

The evolution of Java has also significantly changed the availability of security function from release to release. Following is a summary of the major security functions found in three of the Java releases.

- Applet resource access: resource access given to code downloaded from the Web (i.e., applets).
- Application resource access: resource access given to code that is local to the system running the JVM (i.e., an application).
- Lexical scoping of privilege modification: enforces the "least privileged model"; that is, only the piece of code that needs the privilege will be enabled.
- Cryptographic services for data confidentiality and integrity: allows for the encryption and verification of data.
- Digital signature services for code signing: provides the ability to have authentication of the origin of the code (e.g., who signed it)

The table below summarizes the significant security functionality found in the various major JDK releases.

Java Security Functionality

Functionality	JDK 1.0.x	JDK 1.1.X	JDK 1.2
Applet resource access	Constrained access given to applets (Java sandbox)	Constrained access given to unsigned applets (Java sandbox) Signed applets unconstrained	Policy-based access to resources
Application resource access	Unconstrained access given to applications	Unconstrained access given to applications	Policy-based access to resources
Lexical scoping of privilege modification	Not available	Not available	Stack annotation based with beginPrivileged( ) and endPrivileged( )
Cryptographic services for data confidentiality and integrity	Not available	Java Cryptographic Extensions 1.1	Java Cryptographic Extensions 1.2
Digital signature services for code signing	Not available	Java Cryptographic Architecture DSA signature	Java Cryptographic Architecture DSA signature

or applets cannot gain unauthorized memory access to read or change contents.

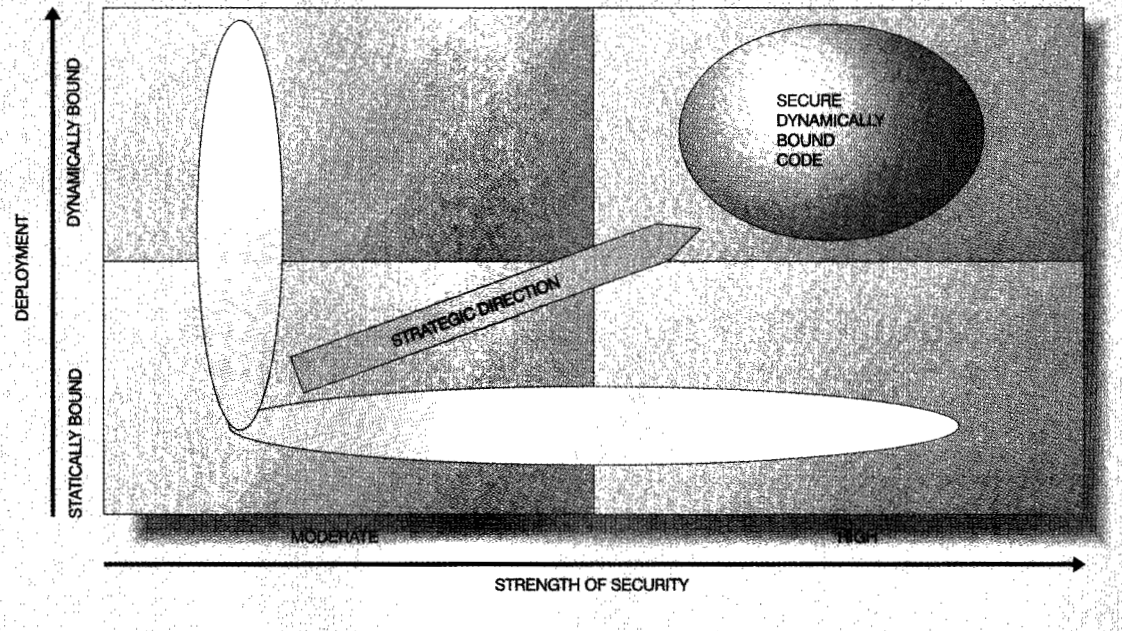
- Encryption and digital signatures—Java supports the use of powerful encryption technology to verify that an applet came from an identifiable source and has not been modified.
- Rules enforcement—Java is completely object-based. By using Java objects and classes to represent corporate information entities, it is possible to explicitly state the rules governing the use of such objects.

Java run-time environment. Java, as an object-oriented language, can be used to develop applications in much the same way as C and C++ are used to improve programmer productivity. In contrast to many other programming languages, Java provides a standard set of libraries, including a broad range

JAVA SECURITY WITH RESPECT TO COMPONENTS

The improvements needed in Java security are aimed at assuring that e-business applications are secure. The following schematic highlights the needed movement from an initial applet-only deployment scenario, with minimal protection requirements, to an environment where any program type (e.g., applet, servlet, etc.) has the same protection primitives and a policy enforcement mechanism capable of uniform policy enforcement available on all platforms. In addition, the functionality improvements and assurance quality from release to release will improve the strength of protection provided.

Note: The term "Secure Dynamically Bound Code" in the following schematic refers to a collecton of distributed applets assembled within a single HTML page or a collection of JavaBeans or Enterprise JavaBeans composed in an application builder, or dynamically during run time, to construct an application.



of communications and security capabilities, thus simplifying construction and deployment of client/server and distributed systems applications. It also provides data type safety and performs bytecode verification when the code is loaded into the JVM runtime environment. This catches bugs that arise from programming errors, especially with pointer arithmetic and array out-of-bounds indexing errors.

Additionally, Java-based support environments exist for the following:

- Applets—downloadable code with restricted access, usually coupled with a browser
- Aglets—code pushed to the end device rather than the pull model of applets
- Servlets—Java code on the server
- Orblets—code utilizing object request broker communication mechanisms

Any of these environments may utilize a component model for reusability, such as JavaBeans**[4] and Enterprise JavaBeans**.[5]

Java is evolving to support a multitude of system configurations:

- Embedded systems destined to be found in many consumer devices in the home and elsewhere
- Java-based smartcards
- Personal management devices such as pagers and PDAs (personal digital assistants)
- Network computers—thin and flat clients
- Mobile systems
- Highly scalable enterprise servers

To support this range of configurations, a family of products—including JavaOS, PersonalJava, Enterprise Java, and JVMs hosted in application builder environments—is evolving to meet the unique needs found in the respective environments.

Security requirements vary depending on the unique characteristics of these substantially different environments. Initial work indicates that a common security model is likely and that the environmental differences can be supported by providing extensions, rather than deploying significantly different security models. The use of a common security model will simplify and reduce the cost of application and library development and deployment.

JDK 1.2 permissions model

The discussion below is based on beta-level code; some operational changes may result from industry feedback prior to general availability.

JDK 1.2 introduces a number of new security features that make it easier to enforce access control of protected resources.[6] In earlier versions of Java, JVM resource access was enforced by the "sandbox" security model. Extensions were usually limited to features implemented by the platform provider (e.g., browser, Web server). The new JDK 1.2 permission model is much more flexible and even permits application-defined resources to be added to the access control system. Java programs now have the ability to define access restrictions on sensitive resources without requiring the writing of a new Security Manager or modifying the underlying platform. This means that applets downloaded into a browser, or servlets loaded into a Java server, can add resource access controls to a JVM without having to modify the underlying browser or server implementation.

One of the notable features of the new security model is that most of the access control implementation is contained in the Java security subsystem. Typically, Java programs (e.g., applets, servlets) and components or libraries (e.g., packages, beans) do not need to contain any access control code. When a program wants to add protected resources to the JVM, a method call can be added that will check whether the restricted operation is permissible. One general technique employed in JDK 1.2 is to create a *guarded* object, whereby access to an object or operations on an object are restricted by an access control call. Examples of how to use the access control features are shown later in this paper.

The JDK 1.2 access control subsystem introduces new concepts. The first is CodeSource, which is the combination of a set of signers (digital certificates) and a codebase URL (uniform resource locator). The CodeSource is the basis for many permission and access control decisions. The second concept is the security policy. The policy contains a number of *grant* entries that describe the permissions granted to a particular CodeSource (see the "Policy Database" sidebar, later). A grant entry may contain one or more Permissions, which is the right to access or use a protected resource or guarded object. Lastly, a ProtectionDomain is an aggregation of a CodeSource and the Permissions granted for the CodeSource as specified in the policy database. Each class file loaded into the JVM via a ClassLoader is assigned to a ProtectionDomain, as determined by the CodeSource of the class.

Loading Java programs. The three legs of JVM security are the ByteCode Verifier, the Security Manager, and the ClassLoader. Prior to JDK 1.2, each application had to write its own subclasses of SecurityManager and ClassLoader. JDK 1.2 simplified the development process by creating a subclass of ClassLoader called SecureClassLoader. SecurityManager no longer is abstract and can be instantiated or subclassed. Most of its methods now make calls to methods in class AccessController, which provides the access control function in the JDK 1.2. Since most of the SecurityManager methods call AccessController, this greatly simplifies the writing of new SecurityManager subclasses.

To automatically invoke the new security subsystem, a Java application is started from the command line of a native operating system. The Java run time creates an instance of SecureClassLoader, which in turn is used to locate and load the class file of the application. A subclass of SecurityManager is created and installed in the Java run time. The main() method

of the application is then called with the command line arguments.

The purpose of the change in the Java run time for starting Java applications is twofold. First, a simple SecurityManager is installed in the system that uses the new Java security access control subsystem. Second, a SecureClassLoader is used to safely and correctly load classes into the Java run time.

SecureClassLoader has several important purposes. The first is to make sure that searching for classes is done in the correct order. When the JVM needs a class, SecureClassLoader first looks for files referenced by the *classpath* of the JVM to see whether it is available. Files in the classpath are intended to be the completely trusted classes that are part of the Java run time. For example, all of the code shipped with the JVM is included in the classpath, and is therefore considered trusted code. If not found in the classpath, an application-defined location can be searched (e.g., a Web server via a URL request). Finally, code may be part of Java Standard Extensions, which is a set of classes that are available in the host file system but are not part of the JVM classpath. Classes in the Standard Extensions are typically located on the disk drive of the host system (e.g., the workstation or personal computer), but the classes are not part of the fully trusted run-time classes of the JVM.

The second important purpose of SecureClassLoader is to create and set the ProtectionDomain information for classes loaded into the JVM. When the SecureClassLoader loads a class into the JVM, the codebase URL and the digital certificate used to sign the class file (if present) are used to create a CodeSource. The CodeSource is used to locate (or instantiate) the ProtectionDomain for the class. The ProtectionDomain contains the Permissions that have been granted to the class. Once the class file has been loaded into the JVM, SecureClassLoader assigns the appropriate ProtectionDomain to the class. This ProtectionDomain information, and, in particular, the Permissions in the ProtectionDomain, is used in determining access control during run time.

Once a Java program starts to run, the SecureClassLoader assists the JVM in loading other classes required to run the program. These classes are also assigned the appropriate ProtectionDomains based on their CodeSource.

Run-time access controls. At various points during the execution of a Java program, access to protected resources is requested. Such access includes, but is not limited to, network I/O attempts, local file I/O, or attempts to create a new ClassLoader or to access a program-defined resource. To verify whether the running program is allowed to perform the operation, the library routine makes a call to the SecurityManager's checkPermission(permissionToCheck) method, which subsequently calls AccessController.checkPermission (permissionToCheck). These method calls are responsible for determining whether the current thread has sufficient permissions. checkPermission( ) takes a Permission object as an argument. The AccessController method checkPermission( ) walks back through the stack frames of the current thread, obtaining the ProtectionDomain for each of the classes on the thread's stack (see the section on "Thread stack frames"). As each ProtectionDomain in the thread stack is located, the permissionToCheck is compared to the Permissions contained in ProtectionDomain. For each stack frame, if permissionToCheck matches one of the Permissions in the ProtectionDomain, testing of the Permissions continues with the ProtectionDomain of the next stack frame (class) on the stack. This testing repeats until the end of the stack is reached. That is, all of the classes in the thread have permission to perform the operation. Thus, the access control check succeeds, typically meaning that the requested operation is able to proceed. If permissionToCheck is not granted to all classes on the stack (there is no appropriate Permission in all of the ProtectionDomains of the classes), then a SecurityException is thrown, and access to the resource is denied.

A flaw in the above scenario is when a class has a set of Permissions and does not care who its callers may be, as for example, a JavaBean installed on a desktop computer needing to read files from the local disk drive. The ProtectionDomain of the bean's class has a Permission to read these local files. However, the program loaded from a Web server that calls the bean has a ProtectionDomain that does *not* have local file read permission. Normally, if the bean were called by the program loaded from the Web server, the bean would be denied access to the files on the local disk drive because the program from the Web server does not have a local file read Permission. However, if the bean calls AccessController. beginPrivileged( ), an annotation is made on the stack frame of the thread, indicating that when AccessController.checkPermission(permissionToCheck) searches for ProtectionDomains, the search stops at this stack frame. The bean may make any number

of method calls, but when AccessController.checkPermission(anotherPermissionToCheck) is called, the search back through the stack frames to find ProtectionDomains stops at this stack frame. Based on the above scenario, the ProtectionDomains for the bean will be checked, but the ProtectionDomains for the program from the Web server are not checked since the search stopped at the stack frame for the bean. Therefore, the file read operation will succeed. To turn off this privileged mode of operation, a call to AccessController.endPrivileged( ) removes the stack annotation. If the application were to forget to call AccessController.endPrivileged( ), the JVM gracefully recovers because the beginPrivileged( ) and endPrivileged( ) call are associated with the stack frames. That is, once the method that called beginPrivileged( ) exits, the privileged mode is automatically turned off.

A subtle aspect of the above beginPrivileged( )/endPrivileged( ) operations is that programs creating new threads would lose ProtectionDomain information when a new thread is created. That is, each new thread creates a new run-time stack. The classes on the stack of the parent thread are not present in the new thread. Important ProtectionDomain information is no longer available when a checkPermission( ) operation is performed. This would give new threads more permissions than the threads that created them. To get around this apparent loss of security information, the ProtectionDomains of the parent thread are attached to (inherited by) a child thread when it is created. So, unless a beginPrivileged( ) operation is performed in the child thread, the ProtectionDomains of the parent thread are also checked during a checkPermission( ) operation.

POLICY DATABASE

The default JDK 1.2 access control subsystem implementation uses a policy database (e.g., a flat file) to describe access control policies. Each *grant* entry contains two parts: a CodeSource and a list of Permissions.

The CodeSource is also comprised of two parts:

- Code base URL—indicating where the classes are residing. If omitted, the code can be from any source.
- Digital certificate used to sign the classes/JAR file. If omitted, AccessController will not check whether the code is signed or not.

The Permissions consist of one or more entries, each of which is comprised of four parts:

- The fully qualified name of the permission class (includes the package name), e.g., java.util.PropertyPermission.
- The first quoted argument for the permission constructor, e.g., "java.version".
- The second quoted argument is an optional modifier, e.g., "read".
- The last argument is an optional name that has been assigned to a digital certificate for the permission class; e.g., the permission class (see the first bullet above) was loaded over the network and was signed with a digital certificate.

To gain an appreciation of these capabilities, a few examples are given.

Example 1: Applets from www.NominallyWidgets.com, signed by a digital certificate called LarrysCertificate can read files in directory /tmp and write files in directory /tmp/koved on the local disk drive. Note the wild card "*" as the last character of the first quoted argument. Many permissions support the wild card as the last character of the first argument.

```
grant codeBase "http://www.NominallyWidgets.com" signedBy "LarrysCertificate" {
    permission java.io.FilePermission "/tmp/*" "read";
    permission java.io.FilePermission "/tmp/koved/*", "write";
}
```

Example 2: Servlets from any location, digitally signed or unsigned, are allowed to read some additional system properties java.version and java.vendor.

```
grant {
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.vendor", "read";
}
```

Example 3: Programs signed by LarrysCertificate are allowed to execute a program called LocalStatus on the local operating system, regardless of where the class files originated.

```
grant signedBy "LarrysCertificate" {
    permission java.io.FilePermission "user/local/bin/LocalStatus", "execute";
}
```

Thread stack frames

Each thread in the JVM contains a number of stack "frames." Simply stated, these frames contain the method instance variables for each method called in the current thread. If a program debugger were used, the debugger would be able to show the instance variables for each of the methods on the stack. For further clarification, a couple of examples are offered.

Example 1: Simple check of the current thread. Figure 1 shows a snapshot of a program, called MyProgram, with a codebase URL of http://www.NominallyWidgets.com. After the security subsystem has been initialized, the program tries to get a system property by calling System.getProperty("java.home"). The getProperty method calls the Security Manager method, checkProperty(), to see whether the current thread is allowed to read a system property. In turn, the Security Manager calls the method AccessController.checkPermission with an argument of PropertyPermission("java.home", "read") to see whether all of the classes on the stack have the appropriate permissions.

Note that as of the writing of this paper, classes loaded via the JVM classpath are not assigned to a ProtectionDomain. These classes logically are assigned to the system domain, which has unrestricted access to all Java and application-defined resources. That is, when the AccessController does its checking, it assumes system domain classes have all permissions.

The access control in this example works as follows:

- Class java.security.AccessController is in the system domain. By default, the system domain has implicit permission to read the properties; checking is allowed to proceed to the next stack frame.
- The class java.lang.SecurityManager is in the system domain. By default, the system domain has implicit permission to read the properties; checking is allowed to proceed to the next stack frame.
- Class java.lang.System is in the system domain. By default, the system domain has implicit permission to read the properties; checking is allowed to proceed to the next stack frame.
- MyProgram has a ProtectionDomain with a codebase of http://www.NominallyWidgets.com. The permissions for this ProtectionDomain are checked. If the permission is not granted, a security exception would be thrown, and the getProperty() method call would fail. If the permission is granted, checking is allowed to proceed to the next stack frame. In this example, the permission is granted, so checking is allowed to proceed to the next stack frame.
- MyProgram has a ProtectionDomain with a codebase of http://www.NominallyWidgets.com. The permissions for this ProtectionDomain are checked. The permission is granted; checking is allowed to proceed to the next stack frame.
- The class java.lang.Thread is in the system domain. Since system domain has the implicit permission to read the properties, checking is allowed to proceed to the next stack frame.

In this example, the operation is permitted since the checks succeeded for all stack frames.

If this thread had been created by another thread with ProtectionDomains in any of its stack frames, the inherited ProtectionDomains from the parent thread would also be checked.

Obviously, there is room for optimization because many of the ProtectionDomains on the thread's stack are not unique. In practice the Permissions in each unique ProtectionDomain are checked only once per call to checkPermission().

Example 2: beginPrivileged() was called. In this example, a thread was created, and a program calls a bean (MyBean) containing a protected resource. MyBean calls AccessController.beginPrivileged() and then calls a method in MyBeanProtected that will check to see whether the thread has the appropriate permissions.

The thread's stack is presented in Figure 2.

The access control works as follows:

- java.lang.SecurityManager is part of the system domain, so it has implicit permission to access the resource. Proceed to the next stack frame.
- MyBeanProtected is part of the ProtectionDomain that has access to the resource. Proceed to the next stack frame.
- MyBean is part of the ProtectionDomain that has access to the resource. Since AccessController.beginPrivileged() was called from this stack frame, stop here; do not check any more stack frames.

The following two-stage algorithm describes how AccessController computes permissions.

Figure 1 Simple check of the current thread

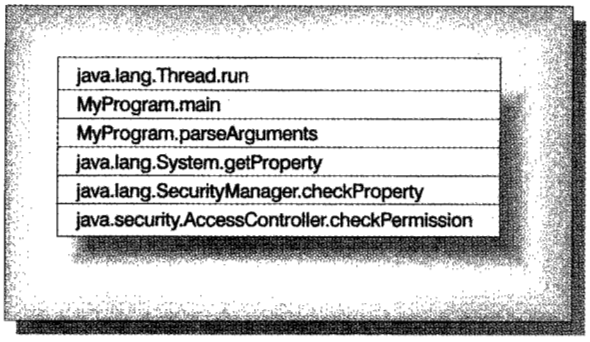


Figure 2 beginPrivileged( ) was called

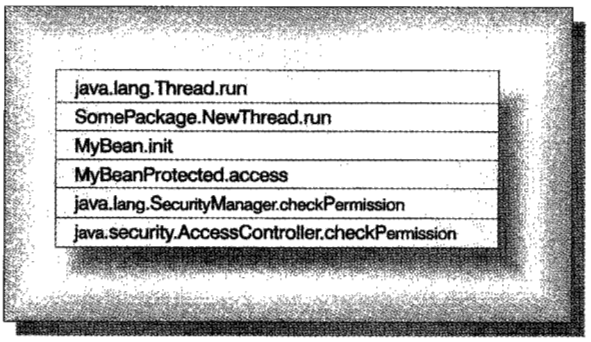


Figure 3 AccessController algorithm stage one

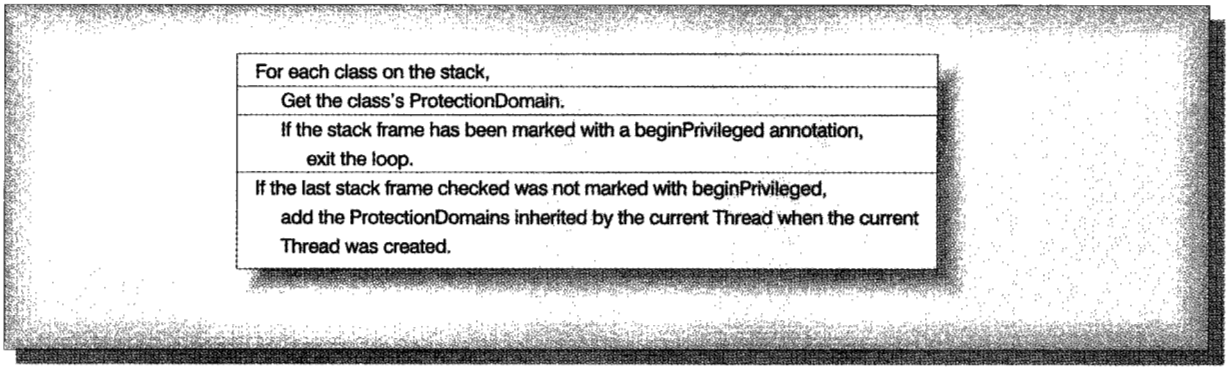


Figure 4 AccessController algorithm stage two

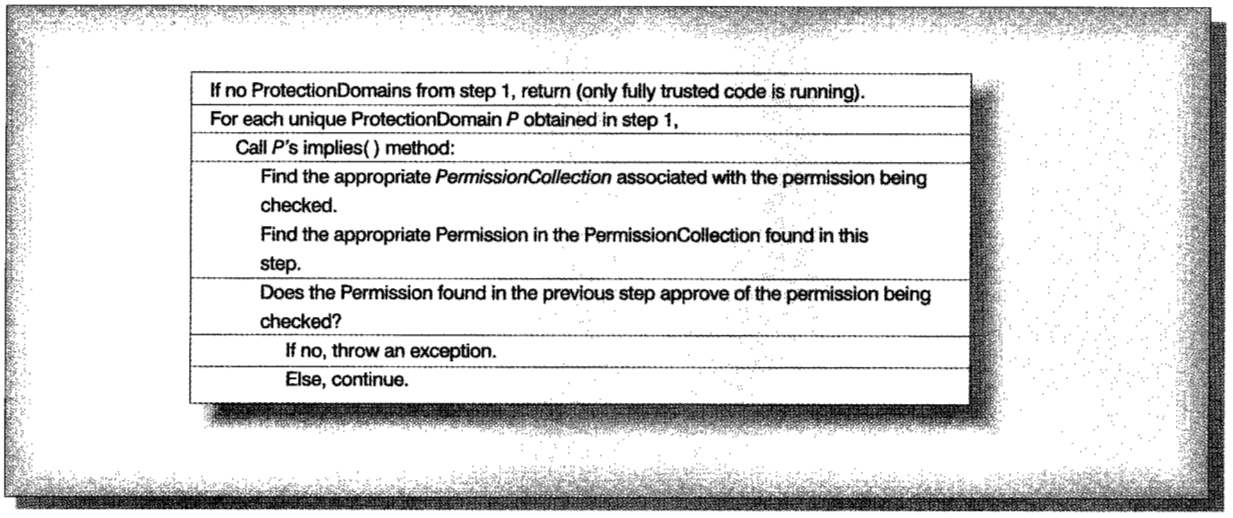


Figure 5 JDK 1.2 Permissions model relationships

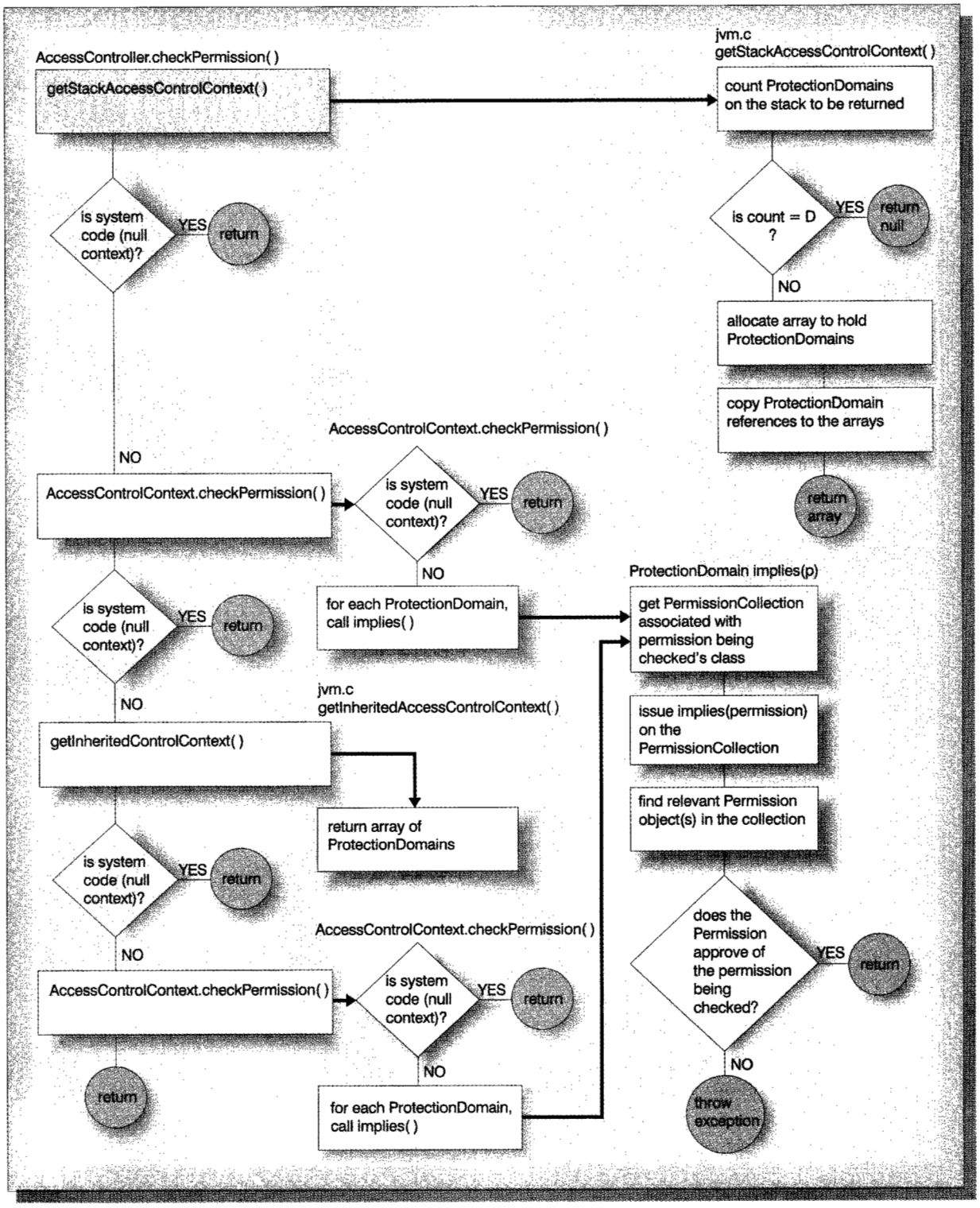
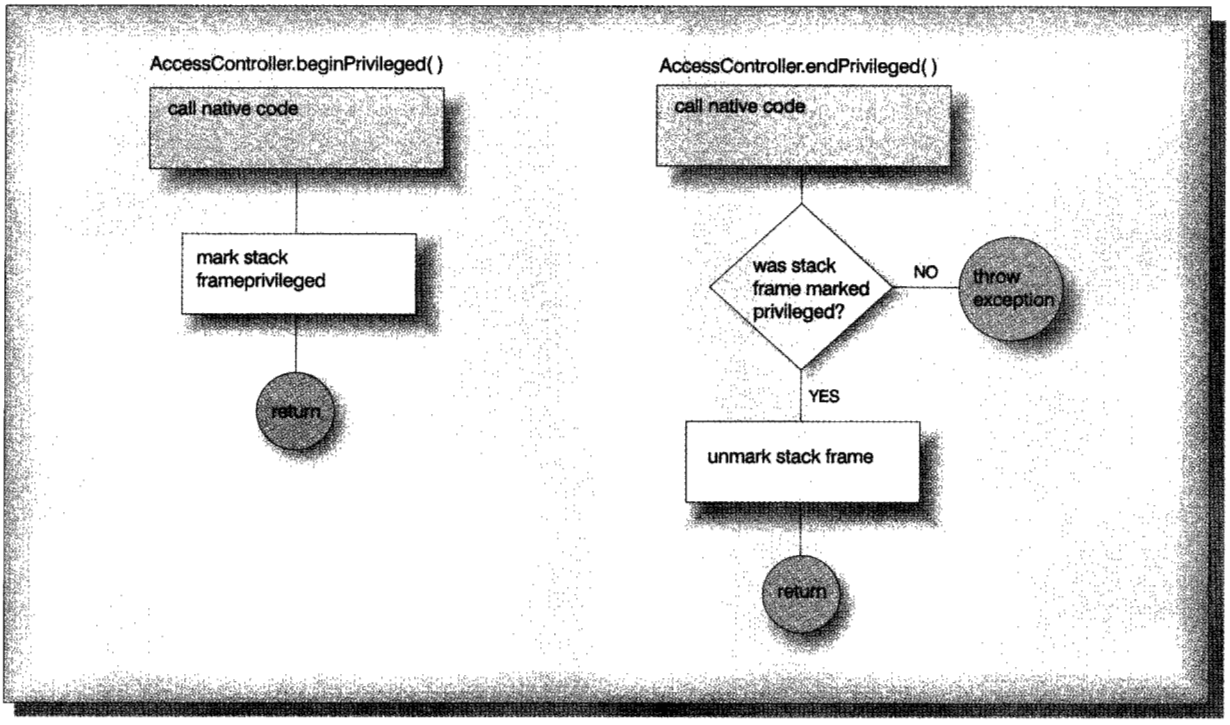


Figure 6 JDK 1.2 Permissions model relationships (continued)



The first step is to obtain a list of ProtectionDomains used by the second step as shown in Figure 3.

The second step is to check with each of the ProtectionDomains to see whether it contains the permission being checked (Figure 4).

If an exception is not thrown, the requested operation is permitted.

The flowcharts in Figure 5 and Figure 6 graphically indicate how the new functions just described relate.

Tools

There are three important security tools and two data repositories in JDK 1.2. These tools are primarily oriented for users of the JDK rather than for end users of applications that incorporate Java. They are described below.

The Java archive (JAR) utility tool (here represented as *jar*) was designed mainly to facilitate the packaging of Java object files and resources into a single file, or *archive*. When Java components (class, image, audio, etc.) are placed in an archive, they can be downloaded via a single HTTP (HyperText Transfer Protocol) transaction with a server rather than requiring a separate HTTP connection for each downloaded component. Network download performance is generally improved, especially since the jar tool can compress the contents of the archive.

Cryptographic key management, *keytool*, is a cryptographic key and certificate management utility. It, along with jarsigner (see below), replaces the JDK 1.1 *javakey* tool. The keytool utility allows developers to administer their own public or private cryptographic key pairs and associated certificates for use in client authentication, or for data integrity and authentication services, requiring digital signatures. This utility also allows the caching of the public key of a communicating peer (e.g., a Web server).

Keytool manages a *keystore* (repository) of private keys and the associated X.509 certificate chains authenticating the corresponding public keys. The keystore may be protected with a passphrase (as in the

default implementation from JavaSoft) or by a stronger protection mechanism (e.g., cryptography). There are two basic entries in the keystore:

- Key or certificate entries—consist of a private key and a certificate chain
- Trusted certificate entries—multiple single certificates with public key entries

Keytool can create a keystore, clone or delete entries in a keystore, import certificates (trusted and nontrusted), export certificates, display the contents of the keystore, and generate self-signed certificates (including public or private key pairs).

Both Java 1.1 and 1.2 implementations of keytool only support the DSA (Digital Signature Algorithm) key and the DSA/SHA-1 (Secure Hash Algorithm) signature algorithms. The key size is limited to a maximum of 1024 bits.

For backwards compatibility, it is expected that JDK 1.2 will be able to parse or process the JDK 1.1 keystore format.

The utility for signing JAR files digitally, *jarsigner*, has two major functions: sign jar files, and verify the signature(s) and integrity of signed jar files.

The certificate(s) contained in a jar file are used by jarsigner to verify the digital signature(s) and to verify whether or not the public key of the certificate(s) is contained within the specified keystore. Also, jarsigner verifies that the jar file has not been tampered with in any way. Currently jarsigner can only sign jar files that were created with the jar utility.

Jarsigner can sign jar files using either the DSA key algorithm, with the SHA-1 digest algorithm (the default cryptographic engine provider supplies the DSA/SHA-1 algorithms) or the RSA** (derived from the original founders: Rivest, Shamir, and Adleman) key algorithm with the MD5 (modification detection) digest algorithm. That is, if the signer's public and private keys are DSA, jarsigner uses the DSA/SHA-1 algorithms. If RSA keys are provided, the RSA/MD5 algorithms are used.

A jar file may be signed more than once by simply running jarsigner more than once, specifying a different signer each time.

Access control policy database management is handled by *policytool*, a graphical user interface that assists a user, such as a system administrator, in specifying, generating, editing, exporting, or importing a security access control policy for the JVM. The tool creates a *policyfile* as described below.

As described above, the *keystore*, cryptographic key storage, is a repository of private keys and the associated X.509 certificate chains authenticating the corresponding public keys. The keystore is a concrete implementation of the keystore class provided in the java.security package. All three utility programs described above use the keystore.

The policy for a Java run time, specifying which permissions are available for code from various code sources, is represented by a *Policy* (access control policy database) object. More specifically, it is represented by a Policy subclass providing an implementation of the abstract methods in the Policy class (which is defined in the java.security package). A default implementation of Policy, called PolicyFile, reads the policy information from flat ASCII files. The policy framework allows policy information to be stored on the local system or anywhere in the network.

The policy configuration file(s) for Java 1.2 installation specify what permissions (which types of system resource accesses) are allowed by code from specified code sources. The information stored in the policy database is described in the "Policy Database" sidebar.

Java and browser-based security models

Browsers and other Internet technologies were in the marketplace prior to the broad introduction of Java; consequently, it is not surprising that mismatches exist in the security models provided by the rapidly evolving Java related technologies. Major progress toward synchronization can occur if agreement can be reached on selected parts of the models in an orderly fashion. The mismatches result primarily from the early limitations of Java security and the desire to fill these voids to meet customer requirements. One such area where there is a concerted effort in model alignment is in the access control model.

This section provides a high-level view of the capability classes of a representative browser with comparable features found in the Java 1.2 security architecture. Both use a stack-based approach to authorization. However, they employ significantly

different access control models and authorization mechanisms. The browser's proposal is more ambitious, but the added functionality may be more difficult to manage.

It should be recognized that mismatches also exist among browser security models as they expand support to cover the hosting of HyperText Markup Language (HTML) pages, scripts, and applets. In some environments, signed Java applets will have reduced access to some Java elements because the containing page or script calling the applet is not signed. Only the Java access control mechanisms of the browser will be highlighted in the following discussion.

Alternative access control model. A commercially available Web browser's Java Virtual Machine implementation that uses an alternative access control model based on targets and explicit activation or deactivation of permissions was examined.[7,8] A target is a mapping of a principal to an operation on an object (i.e., roughly a traditional permission representation). Whether a permission can be enabled or not depends on a three-valued logic for "enabling" policy. Essentially, if at least one principal (user, systems administrator, target class definer) permits the target and no principal forbids the target, the target can be enabled. Java developers can enable such targets at run time. Also, developers can disable (forbid) or revert (undo enabling of) targets at run time. This model enables the policy of multiple principals to be combined and less than maximal rights to be granted to an applet. It is up to the applet code to manage this subset of enabled permissions.

At run time, targets are activated by enablePrivilege(), prohibited by disablePrivilege(), and deactivated by revertPrivilege(). Reverting only affects the calling stack frame, so targets enabled in previous stack frames are still active and supersede the reversion. Therefore, it is not possible for a descendant method to remove a privilege, possibly enabling security exposures.

The browser authorization mechanism checks the stack for an enabled privilege for the request. If one is found, the operation is permitted regardless of the trust of the classes in the thread's stack (the method's callers). Therefore, explicit prohibition or proper permission reversion of rights is necessary to prevent an unauthorized principal from using another's "enabled" privilege. Note that there may be other methods that enabled the permission higher in the thread stack.

If a method in the Java core classes (the basic Java run time) requires the addition of access controls to close a security hole, the browser model described in this section will cause trouble for existing code. It is straightforward to add a permission to allow the code to enable the permission. However, the code does not already contain code to enable the permission since the permission was not required when the code was originally written. The same would be true for library and bean writers who discover in subsequent versions of their software that they need to add access controls. The implication is that application writers would have to update their code to support (i.e., enable) any new permissions added to the base Java classes or libraries and beans they employ. This results in severe version management problems. This problem does not exist with the JDK 1.2 access control model. In JDK 1.2, only an update to the policy database is required.

The browser model provides more functionality than the Java model.

In summary, the browser model provides more functionality than the Java model, but this functionality places more responsibility on programmers to track granted and retracted rights and to be aware of browser, JVM, or library/bean version differences.

Stack-based authorization

The current access control mechanisms in Java are based on "stack introspection," or logically walking the stack frames of the thread to see whether the calling methods or classes have sufficient permissions to perform a requested operation. Given the coarse granularity of the current permission structure, the performance of stack-based authorization appears to be acceptable. However, since all security issues cannot be reduced to stack introspection, it is possible for one object to pass rights to another object and obtain information that it could not otherwise directly obtain. For example, it may be possible to

induce another object to pass a right (e.g., a file descriptor) to an unauthorized object. The unauthorized object was not on the stack when the file descriptor was created, yet it still received the object reference ("off-the-stack" spoofing). This style of security attack is inherent to stack-based authorization techniques because the technique does not track all unsafe interactions between objects.

In environments such as those with stringent communication requirements (e.g., requiring hierarchical, lattice-like, communication protocols regarding information flow) the Java security model may not be adequate. However, for most of the envisioned uses of Java, stack introspection-based access control features are adequate for implementing mission-critical applications.

Security requirements—a high-level view

The following requirements address needs identified beyond the level of function currently planned to be delivered in JDK 1.2; discussions are underway with JavaSoft on many of them, and some may be addressed by the time of JDK 1.2 or in follow-on JDK releases. These are representative categories and examples of requirements within each category, not necessarily a complete compilation of known requirements within each category.

- *Java Virtual Machine high-integrity computing environment*: The requirement is to support concurrent applet or servlet execution with multiple sets of security credentials. Because authentication and credentials requirements vary between systems, and sometimes between subsystems, it is necessary for the applets or servlets within a JVM to support handling multiple sets of security credentials. Simplified APIs (application programming interfaces) will make it easier for application writers to exploit these security features.

  Satisfaction of this requirement should enable secure interactions between clients and servers and between server subsystems as is needed for e-commerce applications.

- *Policy-driven Java security model and security services*: Customers should be able to define and deploy a security policy. The underlying systems (e.g., Java) should have sufficient mechanisms to implement and enforce that policy. Mechanisms for enforcing policy need to include support for: access control, cryptographic and quality of protection, trust, secure delivery of policy statements to the JVM, policy administration, and JVM support of a policy engine.

  Satisfaction of this requirement may reduce the total cost of ownership through simpler configuration and policy administration by making the JVM a single point of policy enforcement for all Java applications.

- *Simple security programming models*: Require simple high-level APIs for quality of protection (privacy, integrity, and nonrepudiation) to allow security-unaware applications to obtain default security protection for such functions as secure communications, secure documents or mail, secure streams, and secure remote method invocation (RMI) and Internet Inter-Orb Protocol (IIOP).[9]

  This requirement may simplify the programming model, making security permissions readily available to any application and reducing potential mistakes made by security-naive programmers.

- *Standards for secure deployment of applications*: A manifest format and single signature standards (such as W3C) for applet and application delivery are needed.

  This requirement should establish a single and low-cost way of ensuring the delivery of applications both to the server and client systems.

- *Standardized programming models*: Create standards for establishing trust and integrity of applications consisting of multiple applets embedded in HTML or XML (Extensible Markup Language) documents.

  This requirement may provide a single model for the allocation of access control or other policy to an application that may consist of a number of componentized elements.

- *Native security services support*: Utilize the security features of the underlying platform.

  This requirement should maximize platform security functionality, improve performance, and allow for consistency.

- *Standards for development and deployment of (cryptographic) service providers*: Cryptographic service

providers should be signed and provide enumerated descriptions of the services within the provider.

This requirement should ensure that international deployment requirements are met and needed information is accessible to applications requiring these services. Cryptographic services can be deployed internationally with control of the strength of cryptographic operations (e.g., key length, signature length, algorithm strengths, etc.).

- *Maintainability, scalability, and interoperability*: Provide centralized administration and the ability to react to changes, the ability to interoperate and utilize non-Java security capabilities, and the ability to support full security functionality in a distributed manner as needed between clients and servers.

  This requirement should allow for the controlled deployment of Java and improves performance and migration.

- *Removal of security as an impediment to performance*: Allow hardware-supported or native-supported implementations of algorithms to be used during validation of class files, and allow policy to define where a combination of trusted signer and usage of a trusted compiler will permit the override of dynamic bytecode verification at class load time.

  Many security functions are highly performance-intensive (e.g., hashing, key generation); improvements are needed to approach performance found in non-Java environments.

It should be noted that Java does not run in isolation; it runs in the context of the operating system platform on top of which it has been implemented. In addition, Java is frequently embedded inside another application, such as a Web browser or Web server. Each of these operating systems and subsystems has an impact on the JVM and Java run-time vulnerability to security attacks.[10–13] When deploying Java programs, care should be taken in configuring the JVM and application files to minimize vulnerability to security attacks; this discussion is outside the scope of this paper.

Future directions

With continued strong support from the software industry, many enhancements required to bring the Java environment in line with the most stringent needs generally provided in the non-Java environment today are possible. These features include strong encryption, sophisticated access control, and the ability to provide centralized security policy management. Some of these capabilities are likely to become available beginning in mid-1998 and throughout 1999. Much work is also underway to provide Java extensions for accessing existing industrial-strength security mechanisms, thus improving system integrity, performance, and functionality. In the future, we will also see high-level e-commerce and other types of applications migrate from individually provided security capabilities to utilize the capabilities in the latest release of the JDK. By late 1998 or early 1999, significant initial security functionality should be expected in all Java environments.

IBM, Lotus, and Tivoli are working vigorously with the Java software industry to bring the business-critical enterprise security requirements forward and ensure that they are met. IBM's Java initiative is acutely aware of the need to focus on securing the Java environment.

Acknowledgments

The authors would like to thank Li Gong, the Java security architect at Sun Microsystems, for in-depth reviews of JDK 1.2 security, as well as reviewing this paper. We would also like to thank the following people from IBM who are actively involved in improving the Java security environment and to whom much insight and original thinking influenced ideas in this paper: Bob Blakley III, Trent Jaeger, Paul Karger, Peter Thull, and anonymous reviewers.

Thanks also to JavaSoft's security group for development insights for JDK 1.2 and for the ongoing relationship between IBM/Lotus and JavaSoft in evolving the Java functionality to meet enterprise class security needs.

Cited references

1. M. Zboray, *Java—Good Start, but Not Yet Secure*, Gartner Group, Information Security Strategies (ISS) (December 1996).
2. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Publishing Co., Reading, MA (1997).
3. D. Flanagan, *Java in a Nutshell: A Desktop Quick Reference*, O'Reilly & Associates, Sebastopol, CA (1997).
4. "JavaBeans (1.0)," http://www.javasoft.com, Sun Microsystems (1996).
5. V. Matena and M. Hapner, "Java Enterprise Beans (0.79)," http://www.javasoft.com, Sun Microsystems (1997).
6. S. Oaks, *Java Security*, O'Reilly & Associates, Sebastopol, CA (1998).
7. J. Roskind, "Evolving the Security Model for Java from Navigator 2.x to Navigator 3.x: setScopePermission," http://developer.netscape.com/library/documentation/security/sectn1.html (1997).
8. J. Roskind, "Security Tech Note #2: Activating Codebase Principals," http://developer.netscape.com/library/documentation/security/sectn2.html (1997).
9. *The Common Object Request Broker: Architecture and Specification, Version 2.2*, Chapter 13, OMG, Object Management Group (February 1998).
10. J. S. Rothfuss and J. W. Parrett, "Go Ahead, Visit Those Web Sites, You Can't Get Hurt . . . Can You?," *20th National Information Systems Security Conference*, sponsored by NIST and the National Computer Security Center, Baltimore, MD (October 7–10, 1997), pp. 80–94.
11. E. W. Felten, D. Balfanz, D. Dean, and D. S. Wallach, "Web Spoofing: An Internet Con Game," *20th National Information Systems Security Conference*, sponsored by NIST and the National Computer Security Center, Baltimore, MD (October 7–10, 1997), pp. 95–103.
12. W. Cooke, "Stupid JavaScript Security Tricks," *20th National Information Systems Security Conference*, sponsored by NIST and the National Computer Security Center, Baltimore, MD (October 7–10, 1997), pp. 116–127.
13. R. Kemmerer, F. De Paoli, and A. L. Dos Santos, "Vulnerability of 'Secure' Web Browsers," *20th National Information Systems Security Conference*, sponsored by NIST and the National Computer Security Center, Baltimore, MD (October 7–10, 1997), pp. 488–497.

Accepted for publication March 20, 1998.

Larry Koved *IBM Research Division, T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: koved@us.ibm.com).*
 Mr. Koved joined the Research Center in 1982 and has worked in a number of areas that involve network computing. He has built multiuser collaborative systems, including multiuser virtual reality systems incorporating visualizations of numerical simulations. He has also worked on a number of mobile computing projects, including user interfaces for mobile computing and algorithms for data replication. His current work is in component-based software, including security issues that arise with the deployment of *mobile,* or downloadable, software.

Anthony J. Nadalin *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: drsecure@us.ibm.com).*
 Mr. Nadalin joined the former IBM Federal Systems Division in 1983, where he worked on secure projects for the government. In 1987 he began working on secure operating systems design. The work included evaluations of MVS™ and VM (virtual machine) operating systems, in support of IBM development laboratories, with the goal of producing commercial "off-the-shelf" secure operating systems. In 1992 he transferred to the Application Systems/400 Division to complete an evaluation of secure operating systems and databases. While on special assignment to the Personal Software Products (PSP) Division, he worked on the specification and prototype Object Security Services (OSS). In 1995 Mr. Nadalin joined the PSP division where he was part of the security design team for base operating system and distributed computing. He assisted in the transfer of the OSS prototype to the SOMobjects™ group for use in Component Broker. In 1996 he joined the Internet Division and continued to work on distributed object security.

Don Neal *IBM Network Computing Software Division, 4205 S. Miami Boulevard, Research Triangle Park, North Carolina 27709 (electronic mail: dhneal@us.ibm.com).*
 Mr. Neal, a Senior Technical Staff Member, joined IBM in 1973, and has worked on a number of technical assignments in the areas of the telecommunications environment, networking architecture and network design, high-speed communications, multivendor interoperability, and systems management. He joined the I/T Security Programs area in 1997 with responsibilities for security strategy and architecture, with a particular focus on Java security.

Tim Lawson *Lotus Development Corporation, 1 Rogers Street, Cambridge, Massachusetts 02142.*
 Mr. Lawson joined Lotus in 1984 and has worked on the testing and development of Lotus 1-2-3®, Symphony®, SmartSuite®, and other business applications for the worldwide market. He is currently the development manager for the eSuite™ Infrastructure—a Java-based abstraction layer for the eSuite Devpack applets. The eSuite Infrastructure provides platform- and browser-independent persistence services and the user interface for the applets. Immediately prior to this, Mr. Lawson worked as architect for eSuite security technologies in collaboration with IBM and JavaSoft.

Reprint Order Number G321-5681.