

***Security Guidelines
for Vetting and Fielding
Java Solutions***

Vic Winkler
CTO Staff
Sun Microsystems, Federal

Contents

Abstract
Introduction
Section 1 - Policy and Enterprise Security
 The Importance of Digital Signatures
 Limitations of Digital Signatures
Section 2 - Guidelines for adopting Java
 Define and Implement a Security Policy
 Enforce Security Policy
 Software Vetting
Section 3 - Overview of Java 'Vetting'
 JDK 1.2 Signatures and Signed Code
 Certificate Verification
 Enterprise Specific Code Signing
 High-Assurance Software Vetting
Conclusion
Appendix: High-Assurance Java Vetting

Abstract

This paper presents a practical Java-based approach to enterprise security. It is well known that security depends on excluding any software that has the potential to compromise a platform. However, such "vetting" of software - examining and approving it - involves effort and administrative cost. Java runtime checking enforces a site security policy, and greatly reduces the need for additional software vetting in most enterprises. We discuss several factors involved in fielding Java solutions, these include: transparent downloading and execution of Java software; security of Java applications and applets; and the fundamental distinctions between Java security and enterprise security. These factors are common to typical Java security questions. If these issues are understood, and appropriate measures are taken, then Java has significant security advantages for cost-effective, network-centric computing.

INTRODUCTION

Many of Sun's customers recognize that Java offers compelling advantages for platform independent, network-centric computing. Java enables clients to download executable content from servers along with

information. The rapid development of Java applications and applets, along with their simplified fielding or ease of downloading, makes Java a powerful enabler for network computing. However, these same features may raise questions about potential security consequences.

Without an enforced site security policy, many users have concerns with software that is not approved, or is downloaded without constraint. This is especially so if there are few evident signs of its activity, or if there are no means to intervene in the execution. Because policy is difficult to enforce, this situation is typical for most sites.

Consider the figures below. Figure 1 depicts several security concerns that are common in enterprises that lack the means to perform either runtime validation of software or source verification. Retrofitting enterprise security to address these challenges is cumbersome, especially if the computing model does not directly support security. Without basic enterprise trust, the threat of malicious code is constant.

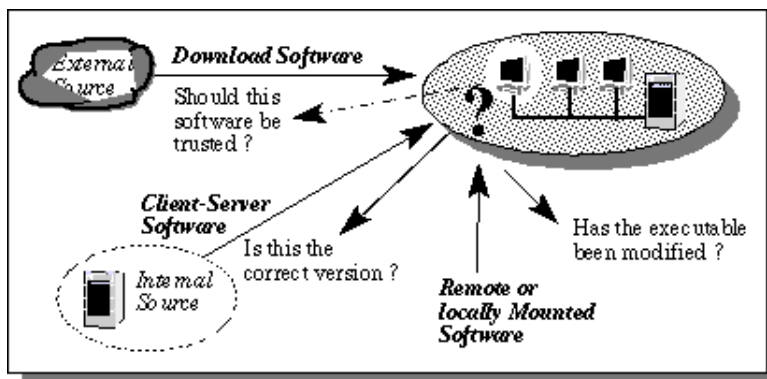


Figure 1 - Security Concerns Without Runtime Validation

But Java inherently offers exactly such security protections. It was designed to meet the security realities of network computing, both as a language and as an execution environment. The Java approach marries network computing with security features that allow a fine-grained security policy to be enforced, both before and while software is executed within a Java Virtual Machine (JVM).

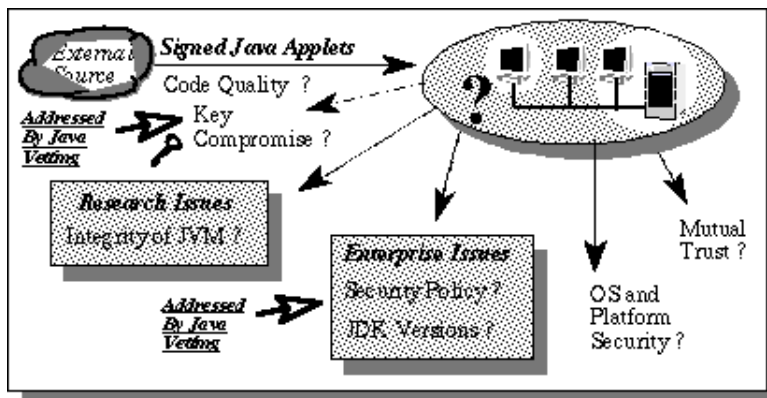


Figure 2 - Residual Security Concerns With Java

Java directly addresses fundamental security concerns, thereby minimizing the need for additional enterprise and platform security measures. Runtime validation within the JVM mitigates the need for

enterprise virus vigilance, and Java's signature and policy mechanisms address concerns with source and pedigree of software.

While it is not a replacement for enterprise security, Java enables security for software, platforms, and enterprises. Figure 2 depicts several residual security concerns that are not addressed by Java, as well as those that can be addressed by a combination of software vetting and Java signing.

The following sections discuss these issues and offer guidance on what can be done today toward leveraging Java for greater security than otherwise possible. Section 1 is a discussion on security policy and the distinctions between Java security and enterprise security. Section 2 includes specific guidelines for the safe adoption of Java in enterprises. Section 3 is an overview of our recommendations for achieving higher assurance via Java vetting. The Appendix outlines what we refer to as a high-assurance software vetting facility.

SECTION 1: POLICY & ENTERPRISE SECURITY

In this section, we discuss the key issues behind achieving enterprise security for Java environments. Establishing and maintaining security in an enterprise is dependent on enforcement of security policies, trusted software and platforms, and effective security management.

A *security policy* is generally an informal, English description of desired system behavior. It identifies information assets and specifies the roles and responsibilities for properly managing those assets. Unfortunately, there is generally a huge gap between policy and enforcement.

First, it is difficult to translate security policy into procedures and rules that can be enforced by systems. Most security countermeasures have limited functionality and do not fully enforce security policies. More effective countermeasures are expensive to implement and maintain. Second, software requires privileged operations to perform many useful functions. Yet granting permissions for privileges should be limited to software that is trusted. Effective enforcement of policy is critical to software and platform trust.

Java goes to the heart of the *software trust* problem. The Java language, as implemented in the Java Development Kit (JDK) 1.2 or higher, minimizes the set of typical language bugs. This greatly reduces potential software vulnerabilities. But Java also enforces security at runtime, before and while Java code is executed. The JVM subjects code to a series of rigorous tests. As Java classes are loaded, the JVM enforces permissions and security on the basis of a fine-grained and configurable security policy. For instance, the JVM can restrict applet execution within a protected space to prevent it from tampering with the JVM or client. If the applet is trusted, then it can be granted greater client privileges.

Java permissions for applets and applications are defined in a Java security policy. This is configurable for a site as well as for a given user. Further, the set of permissions that are allowed for code can be specified on the basis of permitted operations and resources, code location (e.g., file, directory, server, and URL), and code source attributes.

The Java Cryptography Architecture (JCA), and Java Cryptographic Extensions (JCE), provide an

extensible framework for cryptographic functionality. This includes platform-independent APIs for encryption, key exchange, and message authentication.

But Java security can only be achieved - and it is only relevant - if *platform and enterprise trust* is assured. Overall, security is only as good as the weakest link. If it is correctly implemented and fielded, Java is more than adequate as a component of overall security. Compared to alternative security strategies, a Java solution is easier to implement and enforce. With Java, a greater degree of trust and security can be achieved in a simpler and more cost-effective manner.

Effective management is necessary to maintaining enterprise security. Vulnerabilities must be addressed as they come to light. To this end, effecting security patches and revising procedures are activities that should be timely throughout the life-cycle and across the enterprise. Java does offer advantages here as well, in that JVM-enforceable policies can be propagated throughout an enterprise.

The act of monitoring security is usually transparent to end users. In general, these activities are at best minimal, and involve some combination of enterprise level firewalls, activity monitoring, and static integrity checking of executable images or file systems. Because Java security is dependent on platform security, these activities are no less important for a Java-enabled enterprise than for non-Java environments. And, since the integrity of the JVM is critical for runtime security, it is especially important to establish the means to control the introduction of rogue or unapproved JVMs.

The Importance of Digital Signatures

Digital signatures are fundamental to JVM policy enforcement, and also software integrity. A digital signature on a Java Archive file (JAR) is analogous to tamper-proof shrink-wrap. It is unique for the combination of signer (given a valid certificate) and the JAR contents. This assures a strong degree of trust in the origin of the signature as well as the integrity of the contents - subsequent to signing. After Java code is signed by a trusted party it can be safely downloaded, even through a hostile network path. The client JVM can then verify the downloaded code for source and integrity.

Digital signatures, along with other JDK 1.2 features, are fundamentally important to successful Java solutions. Signatures are a component for superior and cost effective security. Prior to JDK 1.2, while digital signatures were available, their use was mostly limited to signing JAR files. JDK 1.2 will expand the uses of digital signatures in the following ways:

- Administrators and users can define a fine-grained security policy for the enterprise and individual users, based on source signature verification;
- Finer grained JVM access controls will enforce the security policy, thus minimizing the need to develop specialized JVM Security Managers; and
- Security checking will be consistent for downloaded as well as local applets and applications. (Prior to JDK 1.2, local code was always trusted, whereas applet code was not.)

Limitations of Digital Signatures

Although digital signatures will become an important enabler for security, enterprise security depends on more than just signing and subsequent policy enforcement. Because both the signing process and the

policy specification can be subverted, it is not reasonable to solely base code trust on its origin. Even software from trusted sources can be flawed in ways that can be exploited. Several strategies can be employed to overcome these potential weaknesses; these are discussed later in this paper.

All environments should establish a sound enterprise security policy. This should be enforced throughout the enterprise. The following section, identifies general guidelines that should be followed in order to achieve this with Java.

SECTION 2: GUIDELINES FOR ADOPTING JAVA

Like all software, Java should not be allowed free reign with privileged operations. Privilege permissions are necessary, and yet they represent vulnerabilities if misused. The use of privilege should be limited to code that is trusted.

As discussed below, the use of privilege within an enterprise should be described by a security policy. This policy should be implemented for use by all JVMs. The use of the policy should be enforced by monitoring at the platform and enterprise level. And, for higher trust environments, all software, including Java software, should be vetted. Following are specific guidelines for assuring enterprise Java security:

Define and Implement a Security Policy

As stated earlier, security in Java is predicated on a Java security policy. Today, Java inherently supports a sound basis upon which to implement and enforce such a policy. The following should be performed:

- Define a site security policy. The policy should expressly define the parameters for trusting code that requires permissions;
- Translate the site policy into a Java security policy file. The policy file is an ASCII text file and can be maintained by Java's graphical "Policy Tool", or via a text editor;

While multiple policy files are possible, there are two default files, one for the system and one for each user. Each entry in the policy file can constrict permissions on the basis of "CodeBase" (the code source location), and "SignedBy" (an alias for a certificate stored in a keystore). Each of these entries also defines the permissions that are granted to conforming software;

- Specify the location of these policy files. For Solaris, this information is in:

```
java.home/lib/security/java.security
```

(Where "java.home" is the directory where the JDK is installed). That file identifies system and user policy files that will be used.

The policy files can be maintained as network resources, or on each machine;

Enforce Security Policy

While a JVM can today enforce a security policy that is expressed in policy files, it is still necessary to minimize the potential for circumventing policy. At the platform and enterprise level, it is necessary to:

- Prohibit the use of additional policy files that can be passed to the runtime system. This will prevent overriding the site security policy;
- Prohibit the use of Java applications without policy restrictions. By default, Java applications like most software, run without policy restrictions¹. Enforcing the use of a policy with Java applications can be achieved if they are installed - via command line argument - with an off-the-shelf or custom security manager;
- Verify that JVMs and policy files are correctly configured, and that their use is not circumvented. This can be supported by platform auditing and enterprise-level monitoring.

Software Vetting

While Java will likely be a strong link in overall security, some high-trust environments may have specific concerns over potential vulnerabilities in signing or policy specification. In such cases, we recommend that:

- All software, including Java software, should be vetted before it is granted execution privileges. Vetting addresses:
 - Establishing trust in Java code and associated privileges; and
 - Assuring the trust and integrity of the code-signing process.

The following section further discusses the advantages that can be gained from Java software vetting.

SECTION 3: OVERVIEW OF JAVA 'VETTING'

For higher assurance environments, vetting may be employed. Vetting of Java code can be implemented in a range of ways. At its simplest, this can be done by enforcing an enterprise-wide digital signature policy and explicit specification of trusted code locations. For higher assurance environments, Java vetting could entail additional enterprise support, such as signing all approved executables. At the extreme end, high assurance software would require vendor cooperation, or a significant development effort. A variety of such vetting strategies are discussed below.

JDK 1.2 Signatures and Signed Code

This is the simplest scheme, it is accomplished by the standard capabilities of JDK 1.2, and is more than ample for most environments. To be effective, this entails enterprise-level policy files. With a policy in place that identifies approved signers, locations and permissions, each JVM can reliably enforce

privilege use.

Certificate Verification

Trust in the validation of a digital signature requires access to the signing entity's legitimate certificate. To assure valid certificates, they should be obtained by the enterprise through trusted channels, and cached in a trusted manner. Achieving this requires public key infrastructure (PKI) support (Figure 3).

Enterprise Specific Code Signing

Where higher trust is needed, the vetting process can be extended in two simple directions. First, the enterprise can wrap its own signature around all Java code. This would be used by JVMs to only execute code that was signed by a trusted enterprise authority. Code would undergo some degree of validation before it is signed. This affords an additional degree of trust and privilege limitation.

Second, if the enterprise signing process is performed in a separate and trusted facility, then one can establish a great degree of assurance over possible signature subversion.

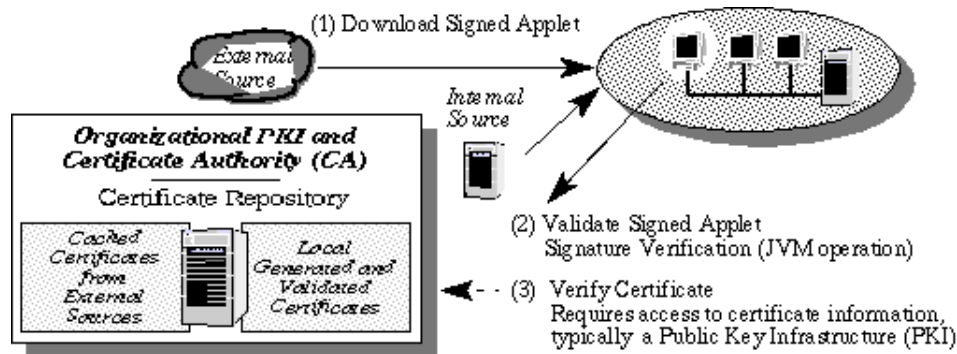


Figure 3 - Validation of Signed Applets via Local PKI Support

High-Assurance Software Vetting

For environments that require high-assurance software, Java vetting would be more involved and requires a significant implementation effort. An overview of a hypothetical implementation of a full vetting process is depicted in Figure 4. This shows that Java code is introduced into a stand-alone facility where it would be vetted and, if appropriate, signed and distributed to the enterprise. (High-assurance vetting is discussed further in the Appendix.)

Regardless of the vetting strategy, vetting code should be done in a manner consistent with the enterprise security policy. There is little benefit from this degree of vetting if the enterprise does not enforce a policy that precludes non-vetted browsers or JVMs.

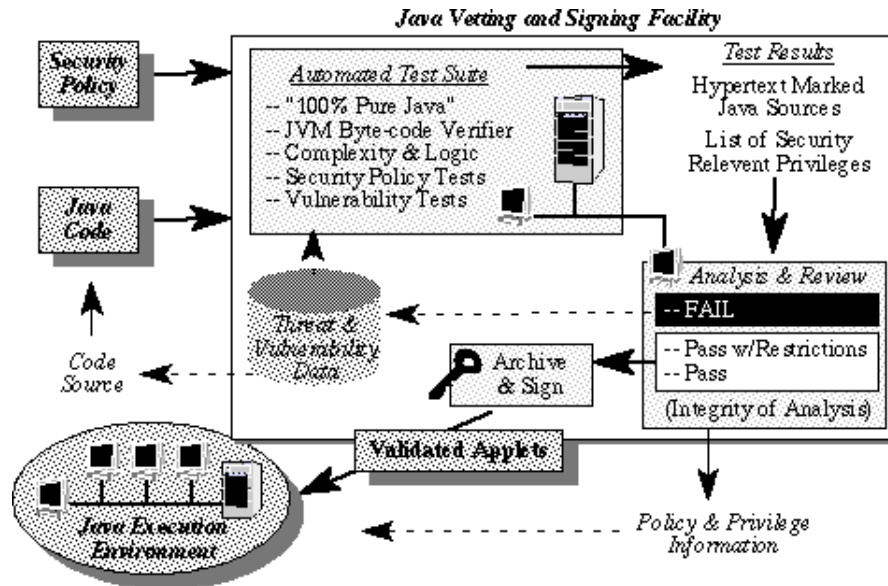


Figure 4 - Java Vetting in an Enterprise

CONCLUSION

As with any software, the security of Java is dependent on the security of the underlying platform. The JVM is only as trusted as the machine and operating system that it runs on. Network and enterprise security are a function of the overall degree of trust in the environment.

Java applications and applets can be made to conform to a sound security policy. With Java, policy and privilege use can be enforced within each JVM, throughout the environment. This is more achievable with Java than with alternatives.

A common question is: "Is it reasonable and safe to implement and field Java solutions ?"

As discussed above, there are three parts to the answer. First, Java security can only be achieved if enterprise security is assured. In other words, Java security is only as good as the weakest link. Second, Java security is more than adequate as a foundation for achieving overall network security. This is clearly the case when we contrast the security advantages of Java vs. alternatives. With Java, you can implement greater security in a significantly more cost-effective manner. And third, both Java and enterprise security must be maintained and monitored. To this end, an enterprise security policy should be defined, and potential vulnerabilities must be addressed as they arise. Java offers inherent and distinct security advantages in fielding solutions.

Today, enterprises can begin to benefit from currently available Java solutions and technology. As computing enterprises transition toward a network-centric model, with more complete security infrastructure support, the security advantages of Java will accrue greater payoffs.

Appendix: High-Assurance Java Vetting

In this section we discuss a potential implementation for vetting Java sources. Such a high-assurance vetting system would serve as a means for testing Java source code for security vulnerabilities. This goes well beyond current practices for almost all software. However, high-assurance environments would benefit from such testing.

Ultimately, a broad range of tests can be developed for the vetting process. To begin with, one can put off developing such tests and simply vet code on the basis of signatures. The benefit of this would be to implement a greater degree of enterprise control and assurance over the execution of foreign code. This could be done via a lookup of approved sources against a controlled list of approved sources. Conceivably, only those sources that meet the list requirements would be granted an additional enterprise signature.

Actual code testing could be implemented by building a JVM-derived framework that is tailored to examine the code, as described below. The JVM framework would be modified to produce appropriate security-relevant outputs that are then incorporated into a copy of the source code for review. Thus, the source code is annotated with JVM derived information.

The framework could be augmented with further tests for known vulnerabilities, code complexity, and other security aspects. The test results could also be incorporated into the annotated source code.

The framework would serve as an overall test harness, with all source code being introduced into the framework, tests performed, and outputs being annotated into a hypertext linked copy of the source code. The principal remaining components would be a web browser (for analyst review of the annotated source code), and a signing process.

The following sections present a functional overview.

STEP 0 Policy Definition *Vetting requires verifying code and achieving a reasonable degree of assurance that there are no known security vulnerabilities. This should be performed against a defined security policy.*

A natural-language policy must be defined and then translated into a set of rules that can be used to test and vet code. For complex and involved testing, these rules will probably not be expressed in a single unified statement. Rather it is likely that they would be distributed throughout the testing process in the form of discrete tests, etc. On the other hand, for a simplified vetting process that is limited to lookup and verification of code sources, the policy relationship to rules could be made very directly. Regardless of the complexity of testing, the results of these tests should be organized in a manner that is consistent with the enterprise policy.

STEP 1 Code Introduction *Software is introduced into the Java vetting facility for code verification and signing. This requires a protected environment that is separate from the execution environment.*

This can be achieved by use of a physically protected computing environment that is not directly network connected to the enterprise or other potential threats. Firewall and guard technology are appropriate for use when external network connectivity is necessary.

STEP 2 Code examination Over time, newly discovered vulnerabilities should be added to the vetting knowledge-base.

At minimum, the testing and vetting process can be based on code origin and uniqueness. If one simply relies on external signatures and certificates, then you have no means to revoke execution privileges based on new knowledge over vulnerabilities or security problems associated with any specific vendor, signer, or applet/application.

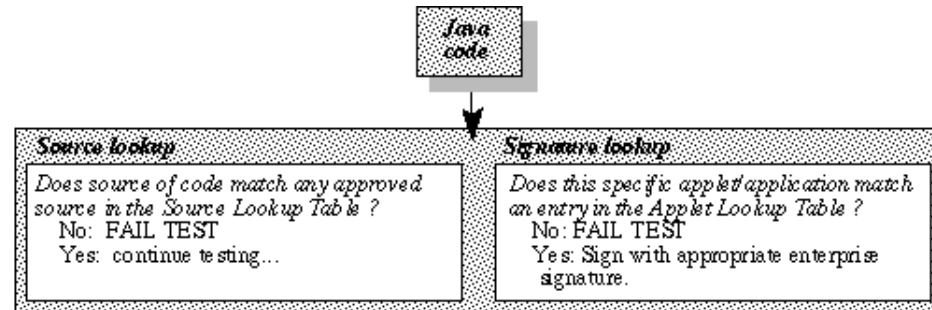


Figure A.1 - Possible Approaches to Code Examination

As indicated in Figure A.1, at least two tests are possible:

- Source lookup - Code source is validated via a lookup against approved signers. Tested code can be granted an additional enterprise signature. This achieves several things. First, one can tightly control the use of the enterprise signature and thereby gain a greater degree of control over what is allowed to execute at each JVM. Second, one can revoke the certificate that is used to sign any enterprise signature under your control. Finally, one can implement a simplified model for signature enforcement within the JVMs within the enterprise.
- Signature lookup - An additional lookup can be performed on the basis of signature information. This test would allow greater granularity over approving specific applets/applications for execution than would a simple source-based-on-signature test. Part of the signature of a piece of signed code is a unique checksum of the contents of the code. Since each checksum/signature pair is unique, one can discriminate among different applets/applications that are signed by the same source. This process also includes a verification of the signer's certificate.

As mentioned earlier, more complex and sophisticated code validation and testing could be based on a framework that is derived from the JVM. For this purpose, the JVM source could be modified and pared down to serve as an extensible framework (see Figure A.2).

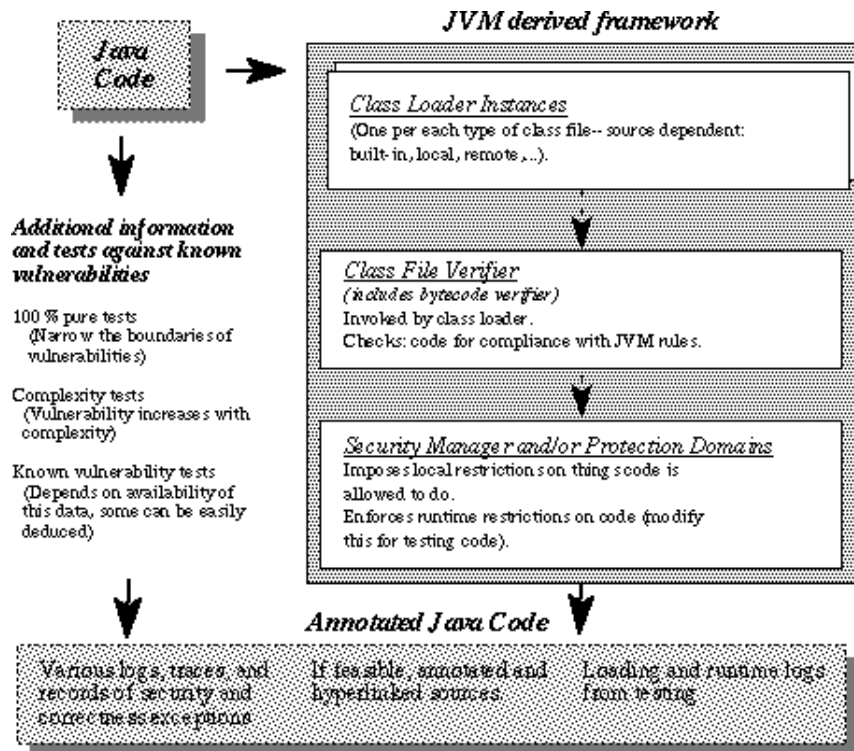


Figure A.2 - Use of Modified JVM as Vetting Framework

For the framework, existing JVM components might include: Class loader, class file verifier, security manager, and protection domains. During vetting, Java code would be operated on by these (see Figure A.2), and annotated appropriately. The output of this process would include HTML Java code (for ease of review) and logs, traces, and other information regarding JVM deduced issues, vulnerabilities, etc.

In addition, a prototype framework could be constructed which would include information about specific or known vulnerabilities. To begin with, this might only contain minimal vulnerability data, just enough to demonstrate the relationship between code vulnerabilities that are privilege-based and access controls that are enforceable via runtime signature checks. Finally, 100% Pure Java tests would also have utility for this testing as they would decrease the problem space in vulnerability testing.

In the future, a knowledge-based approach could have merit, as testing requires specialized expertise and is tedious. This could allow vetting to assess more sophisticated exploit scenarios than single Applets, red flags, and specific actions. For instance, one might identify interactions between sets of factors for not just one Applet, but between Applets. (Other vulnerabilities may be discovered that manifest exploitation not during one execution, but over time or throughout an enterprise.)

In summary, code enters the framework where it would be tested against existing JVM checks as well as against additional rules. Test results would then be output in the form of hypertext-marked Java sources.

STEP 3 Manual Analysis and Review of Annotated Code

Given such future tools for vetting and annotating Java code, it should still be reviewed by a security analyst. This could be facilitated by use of a web browser. Each suite of tested code could be packaged

in a manner that allows for the addition of an index of source components. The analyst would browse through the index according to security annotations and warnings about the vulnerabilities in the code or potential exploits of privilege requests. The analyst accepts or denies each warning via a simplified survey-like facility that is part of the hypertext-annotated code.

STEP 4 Code Signing

Since private cryptographic keys can be compromised, signing code proves little in terms of security, other than that a specific key was used. Assurance in signing is predicated on maintaining private key confidentiality as well as integrity of code through the signing process. Code signing should be performed using existing facilities.

STEP 5 Code Distribution to Enterprise

Once code is vetted and signed it can be distributed via network-safe channels (e.g., one-way guards, or media).

STEP 6 Execution Environment

There is little benefit to vetting and signing code without the means to enforce associated execution restrictions in the enterprise. There are several dimensions to this problem. First, approved signature knowledge is specified in Java security policy files for enforcement. If JVMs are consistent with JDK 1.2 then accepted signatures can be determined via a JDK identity database (a file) on either the JVM platform or on a network server.

Second, only approved and properly configured JVMs should be allowed. Otherwise it is trivial to circumvent security policy and enforcement of policy. Enforcing only approved JVMs can be done in a variety of ways. Executables can be limited by use of permission and access control lists (ACLs) in conjunction with permission masks. A common technique is the periodic use of directory or filesystem checksum checks to identify altered or new executables. It is also possible to perform more active monitoring, such as scanning system audit logs.

E-mail comments to:

Vic Winkler (vic.winkler@east.sun.com)

(go to top of document)