ELE548 Research Essay: Topic 2

Java Security

Zhengqiang Shan

April 12, 1999
Abstract
The initial focus of Java security has been in the support of downloaded applets (small programs) within World Wide Web browsers. The original security model provided by the Java platform is known as the sandbox model. The essence of the sandbox model is that local code is trusted to have full access to vital system resources (such as the file system) while downloaded remote code (an applet) is not trusted and can access only the limited resources provided inside the sandbox. In addition to the safety-related characteristics of language, Java security relies on three throns of defense: the Byte Code Verifier, the Class Loader, and the Security Manager. JDK 1.1 implement the concept of a "signed applet". In this release, a correctly digitally signed applet is treated as if it is trusted local code if the signature key is recognized as trusted by the end system that receives the applet. Unsigned applets still work on the sandbox model. JDK 1.2 introduces new architecture to the security model that make it easier to enforce access control of protected resources. There is no longer a built-in concept that all local code is trusted. Instead, local code is subjected to the same security control as applets, so do applications. The combination of a set of signers (digital certificates) and a codebase URL are codeSource. The CodeSource is the basis for many permission and access control decisions. Security policy contains a number of grant entries that describe the permissions granted to a particular CodeSource. 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. 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 new architecture focus on this issue to provide Java as a secure, ready-built platform on which to run Java enabled applications in a secure fashion.
1 Introduction to Java Security

Java is designed so that programs can be dynamically loaded over the network and run locally. This very powerful new paradigm changes the face of computing greatly but it also introduce a lot of security worries. There are four basic categories of potential attacks Java applets can facilitate:

- System modification: attacks in which an applet damages or alters the client's machine
- Invasion of privacy: attacks in which private information is stolen or the applet misappropriates the client’s identity
- Denial of service: attacks in which the applet consumes CPU cycles, memory, or other resources that make the response slow or hangs a machine
- Antagonism: attacks by obnoxious applets that have annoying behavior

The first two categories of attack are the most dangerous, and Java provides strong defense against them. But Java has weaker control of the latter two types of attack. Java's language feature were designed with security in mind. Many safety-related characteristics of the Java language are: strict type checking, subscript checking, removal of pointer arithmetic and so on. In addition to these safety-related characteristics, Java security relies on three throngs of defense: the Byte Code Verifier, the Class Loader, and the Security Manager.

2 The SandBox model for JDK 1.0

The original security model provided by the Java platform is known as the sandbox model, which existed in order to provide a very restricted environment in which to run untrusted code obtained from the open network. The essence of the sandbox model is that local code is trusted to have full access to vital system resources (such as the file system) while downloaded remote code (an applet) is not trusted and can access only the limited resources provided inside the sandbox. This sandbox model is illustrated in the figure 1. 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.

### 2.1 Byte Code Verifier

The Java byte code downloaded by web browser from web sever is verified before it can run. This verification scheme is meant to ensure that the byte code following the rules. The Verifier checks a number of important properties and at a number of different levels. In basic level, the format of a code fragment is test to be correct; further, a built-in theorem prover is used to make sure that byte code does not forge pointer, violate access restrictions, or access object using incorrect type information. Once passed, following things are guaranteed:

- Stacks will not be overflowed or underflowed.
- Byte code instructions all have parameters of the correct type
- No illegal data conversions occur.
• Private, public, and protected class accesses are legal.

• All register accesses and stores are valid.

2.2 Class Loader

The second security defense is the Java Applet Class Loader. The Applet Class Loader determines when and how an applet can add classes to a running Java environment. In general, a running Java environment can have many Class Loaders active, each defining its own name space. Name spaces allow Java classes to be separated into distinct kinds according to where they originate. Applets are forbidden to install a new Class Loader. The Applet Class Loader install each applet in a separate name space, so classes of different applet are invisible to each other. But a Java application can set up its own Class Loader.

Internally, the Java Virtual Machine tags each class with the Class Loader which installed it. This label is used in the VM to make decisions of security. When a class is imported from the network, the Applet Class Loader places it into a private name space labeled with information about its origin. Whenever one class tries to reference another, the search path the Applet Class Loader follows is: first, local name space; then name space the class making reference to. This prevent imported classes from pretending to be built-in classes and protect fundamental primitives from outside corruption.

2.3 Security Manager

The Security Manager performs run time checks on dangerous methods, is the third prong of the Java security model. This part of security model restricts the ways an applet uses visible interfaces. Instead of allowing applets to define Security Manager, a default Security Manager is provided as a template from Sun. The Security Manager can be customized through sub-classing. Each Java-enable application fills in the template to meet security requirements for the application. The Java run time library forces all access requests to refer to Security Manager.
The duties of Security Manager include:

- Preventing installation of new Class Loader
- Protecting threads and thread groups from each other.
- Controlling the creation of OS programs.
- Controlling access of OS processes.
- Controlling file system operations
- Controlling socket operations
- Controlling access to Java packages

3 Possible holes in the sandbox model

There are three possible paths that Java Byte Code may follow through the security model. The paths chosen depend on where the byte code originates. External byte code (loaded across the network) must be verified, and is subject to the Class Loader and Security Manager. In addition the check during compilation, locally developed byte code is subject to same checks if it is not included in CLASSPATH directory. Byte code from JDK distribution and others in the CLASSPATH directory don’t pass through the Verifier and may be subject to further optional security checks.

In such a sandbox model, all the protection effort of a foreign code (applet, servlet, etc.) viewer is devoted exclusively to make impossible direct attacks from untrusted foreign code. And no protection at all is provided against indirect attacks that could be performed by the means of additional modules that could be placed in the CLASSPATH, because it is assumed that only trustworthy code has been stored in the local file system and that, thanks to the sandbox model, a foreign untrusted applet cannot write by its own in that local file system to store hostile files. The sandbox model makes a priori no distinction between trustworthy
additional modules and secure ones and does not define any kind of internal security policy between local modules.

The activation of the byte code verifier is, by default, reserved to foreign classes loaded by a class loader. The security policies related to the class loaders and security manager that can be used by a Java application are completely defined in Java by the application itself. The integrity of packages is not guaranteed by the runtime system. The standard APIs are not built in the body of the runtime system program but are just a set of classes stored independently in the CLASSPATH and that can be easily modified or masked. All above weakness can introduce serious security threats.

4 Improvement in JDK 1.1

The addition of authentication and access-control mechanisms that rely on the use of cryptography is the possible solution for the holes discussed in previous section. The key to certification and authentication is digital signatures. The idea is simple: to provide a way for people to sign electronic documents so that these signature can be used in the same way as in paper document. It should be verifiable, unforgeable, non-reusable unalterable and non-deniable. The digital signature used for Java code are based on public-key cryptography. One want to sign documents must first use a special mathematical technique to generate two large numbers: public key and private key. He keep the private to generate the signature and others use the public to verify his signature.

JDK 1.1 implement the concept of a ”signed applet”, as illustrated by the figure 2. In this release, a correctly digitally signed applet is treated as if it is trusted local code if the signature key is recognized as trusted by the end system that receives the applet. Signed applets, together with their signatures, are delivered in the JAR (Java Archive) format. In JDK 1.1, unsigned applets still run in the sandbox.
5 New Architecture in JDK1.2

JDK 1.2 introduces a number of new security features that make it easier to enforce access control of protected resources. In earlier versions of Java, JVM resource access was enforced by the "sandbox" security model described in section 2. 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.

According to the Java Security Architecture specification, the new security architecture, illustrated in the figure 3, is introduced primarily for the following purposes:

- Fine-grained access control
- Easily configurable security policy
- Easily extensible access control structure
JDK 1.2 Security Model

- Extension of security checks to all Java programs, including applications as well as applets.

There is no longer a built-in concept that all local code is trusted. Instead, local code (e.g., non-system code, application packages installed on the local file system) is subjected to the same security control as applets, although it is possible, if desired, to declare that the policy on local code (or remote code) be the most liberal, thus enabling such code to effectively run as totally trusted. The same principle applies to signed applets and any Java application.

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. 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. 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.

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.

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. 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 runtime. To verify whether the running program is allowed to perform the operation of accessing to protected resource, 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.

6 Discussion

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. The introduction of the new security architecture in JDK1.2 is a big step in such a direction. Table 1 summarized the evolution of Java security model from JDK1.0 to JDK1.2.

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.

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
Table 1: Java Security Functionality

<table>
<thead>
<tr>
<th>Functionality</th>
<th>JDK1.0</th>
<th>JDK1.1</th>
<th>JDK1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applet resource access</td>
<td>Constrained access for applets</td>
<td>Constrained for unsigned unconstrained for signed</td>
<td>Policy-based access to resources</td>
</tr>
<tr>
<td>Application resource access</td>
<td>Unconstrained access for applications</td>
<td>Unconstrained access for applications</td>
<td>Policy-based access to resources</td>
</tr>
<tr>
<td>Lexical scoping of privilege modification</td>
<td>Not available</td>
<td>Not available</td>
<td>stack annotation</td>
</tr>
<tr>
<td>Cryptographic service for data integrity &amp; confidentiality</td>
<td>Not available</td>
<td>Java Cryptographic Extensions 1.1</td>
<td>Java Cryptographic Extensions 1.2</td>
</tr>
<tr>
<td>Digital signature service for code signing</td>
<td>Not available</td>
<td>Java Cryptographic Architecture</td>
<td>Java Cryptographic Architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSA signature</td>
<td>DSA signature</td>
</tr>
</tbody>
</table>
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.

7 Reference


4. Li Gong, Marianne Mueller, et al. Going Beyond the sandbox: an overview of the new security architecture in the Java development kit 1.2

5. Gray MaGraw and Edward W. Felten, JAVA SECURITY, John Wiely&Sons, 1997

