

Is Java still secure?

David M. Chess, John F. Morar
IBM TJ Watson Research Center
Yorktown Heights, NY

Abstract

Java 2 (version 1.2) has now been released, and with it a new Java security architecture. We discuss the differences between the initial Java security architecture, the interim architectures in popular browsers, and the Java 2 model, including the implications of the new model for Java viruses and Trojan horses. Which previous problems have been solved, what are the security aspects of the new Java features, and what possible holes still remain? We address these and other timely questions.

Active Content

Old-fashioned data, including text, mail, spreadsheets and documents, was essentially passive: the bits and bytes arrived on your computer on diskette or over the network, and programs sitting on your machine examined them and presented them to you in the proper format. Images in a known format got displayed by a display program that knew about that format. A document designed for a particular word processing program was opened in that program, and the program, not the document itself, was in charge of presenting the document's content to you.

Active content is a new paradigm, in which data objects themselves, including documents, mail, spreadsheets and Web pages, contain the knowledge necessary to correctly present their content to the user, and if necessary interact with the user (and the user's computer!) to process that content. Macros in Word documents are a primitive form of active content; when you open a Word document, a Basic program contained in the document can run, perhaps welcoming you to the document and offering you a number of different viewing options depending on what parts of the document you want to see first. When you visit a JavaScripted Web page using a JavaScript-enabled browser, a program contained on that page will get downloaded and executed, enabling Web authors to enhance their pages with greater responsiveness and interactivity. If a Web page contains a type of data your computer doesn't currently know how to display, the page can offer an "ActiveX control"; a particular kind of program that your browser can download and incorporate in order to let you see and interact with the new kind of data.

How much can you trust the programs that active-content systems are constantly welcoming onto your computer? Millions of Word users can attest that sometimes the active content contained in a Word document can be, not a helpful assistant, but an annoying or destructive virus. Thousands of Word macro viruses are now known; they exploit the fact that the original version of Word's active content system had no security at all: any document could contain macros, and those macros could do anything at all to your system once you opened the document. When a page offers you an ActiveX control, you will generally be told who (if anyone) digitally signed the control, and given the chance to refuse it; but if you do accept it, the control is just like any other program, and can do anything it wants to your system and its files. Office2000 provides a similar sort of security for macros in Word and other Office applications: macros can be digitally signed, and a system can be set up to run only macros signed by trusted parties.

The Java Platform

The Java(tm) platform[1] is a set of languages and technologies designed with the security needs of active content in mind. Broadly speaking, there are two types of Java programs, general purpose applications (referred to as applications) and limited scope applications (referred to as applets). Applets usually run inside a host environment such as a web browser where it is desirable to protect the local system by limiting the range of functions the applet can perform.¹ Web pages using Java can do the sorts of things mentioned above, downloading special viewers for the data offered by a Web page, interpreters for new image or movie formats, and a host of other special services that old-fashioned passive content could not have provided so conveniently. At the same time, Java provides a richer security model than the all-or-nothing decision required by ActiveX and Office2000. Java applets are run under the watchful eye of a security manager: a piece of system software that prevents them from doing anything that they are not authorized to do. So untrusted programs can be run, but not allowed to read or write any files or access dangerous system services, while programs signed by parties that you trust can be given wider permissions.

Since it was first released by Sun(tm) in 1995, the Java platform has become more flexible, and also more complex[2]. Different browsers and other applications have incorporated Java components in different ways, and different (and incompatible!) extensions have been made to the security model. Version 1.1 of the Java platform included digital-signing technology that allowed users to declare certain parties to be trusted, and allowed applets signed by those parties to execute without supervision by the security manager. The 1.1 technology was not broadly adapted by browser makers, however; both Netscape and Microsoft independently designed their own versions of program-signing and security enforcement for their browser products, in different and very incompatible ways. The most recent major release of the Java platform from Sun, in December 1998, includes an attempt to standardize a very flexible code signing and security model; while this should make things simpler in the long run, the present situation is quite complex, since there are now two different Sun implementations of Java signed-program enablement, and two more from the major browser manufacturers.

In the next sections, we will examine the various Java security models that have been widely deployed, and examine the implications of them for the overall security of systems that use the Java platform. We will not present a detailed tutorial on Java code signing, certificate management, or applet development, and we will gloss over some complex issues where they are not directly relevant to our overall investigation. For a thorough comparison of the detailed steps involved in actually utilizing the various security models, we recommend chapter 12, "Java Gets Out of its Box", in the recent IBM Redbook "Java 2 Network Security" [3].

The original Java security model -- the sandbox

In the first releases of the Java platform, all programs loaded across the network were assumed to be untrusted, and were prevented by the Java security manager from doing anything that might, in the judgement of the Java implementors, damage the system running them, or reveal private or confidential information. While the details differed slightly between implementations and configurations, the usual set of things untrusted applets could not do included:

- Interacting with the local file system: reading, writing, deleting, or renaming files, creating or listing directories, finding the properties of a file (or even whether or not a file with a given name existed),
- Connecting over the network to any computer except the one the applet was loaded from,
- Accepting network connections from any other computer,
- Altering or creating any system properties (the Java equivalent of environment variables) on the local system, or reading any system properties except for a fixed list of "safe" properties (such as the version of Java in use),

¹ The distinction between applications and applets is actually not this simple. Whether a Java program is an applet or an application actually depends not on where the program is loaded from, but on the internal structure of the program. However, for the purposes of this discussion we will refer to trusted locally-loaded Java programs as "applications", and all other Java programs as "applets".

- Invoking any program (such as Microsoft Word, or the FORMAT command) on the local system, or loading any dynamic-link libraries,
- Causing the Java interpreter to exit,
- Prompting the user to enter information, without an obvious "untrusted applet" warning appearing in the prompt window,
- Doing anything that might indirectly avoid any of the above restrictions, such as creating a new security manager or class loader, or altering any standard Java system functions.

A program that can do none of these things will be relatively harmless; it can display lots of annoying messages, but it can't erase your files or steal your secrets. On the other hand, these restrictions (the inability to read or write files especially) also limit the good and useful things that an untrusted applet can do. This set of limits is often called the "sandbox", reflecting the fact that the area to which untrusted applets are restricted is mostly harmless, but also less than ideally useful.

Not that nothing useful can happen in the sandbox; an untrusted applet restricted to the sandbox can interact with the user, and can make connections back to the server it was loaded from. So even from within the sandbox an applet could, for instance, gather your interests and viewing preferences on a particular subject and communicate them back to the server, which could store them in its own files for later use, to present the Web site to you in your preferred way next time. An applet could also let you interact in powerful ways with new forms of highly-compressed 3D data in a "virtual world", administer a physics quiz, or just allow a Web site to present you with an interactive site-map. This sort of function, doable even within the sandbox restrictions, was enough to keep interest in Java high when it was first released. Before long, however, the need to get beyond the sandbox was felt strongly enough that various escape-hatches were opened in it.

Escaping from the sandbox

Even in the first releases of the Java platform, not all programs were untrusted. Programs loaded from the local system ("applications" rather than "applets") were not subject to the control of the security manager, and could do all the things that programs in any other language can, including reading and altering files and connecting to other hosts on the network. But what was needed was a way to allow certain trusted applets to escape from certain restrictions of the sandbox, without sacrificing the convenience of one-click access offered by applets on Web pages, and without opening up serious security holes for less-trusted code.

Release 1.1 of the Java Development Kit from Sun offered a simple escape from the sandbox. By using command-line tools provided by Sun, a developer could package up his Java applet into a "JAR" file,² and sign that JAR file using a digital signature. Users, similarly, could use the command-line tools to indicate that certain signers should be trusted. Using these features, it is possible to deploy a signed applet within an intranet, for instance, and instruct all users how to add the signer to the list of signers their systems trust. (Version 1.1 of the Hot Java browser, distributed by Sun, allows more detailed distinction among signers: each signer can be either fully trusted, untrusted, or trusted to an intermediate degree, where the system will prompt the user every time a program signed by that signer tries to do anything outside the sandbox.)

The JDK 1.1 security enhancements were a good start toward flexible security in Java, but they were never picked up by the browsers which most people actually use, and the command-line tools provided by Sun were complex and beyond the abilities of most end users. Running a signed applet in such a way that it could get outside the sandbox involved obtaining an obscurely-named certificate file, and entering a pair of complicated-looking commands on the command line.

The Netscape and Microsoft browsers took different approaches to allowing applets out of the sandbox. In both cases, once the applet developer has packaged up and signed his program, the user who attempts to run the applet (by visiting a page where the applet is embedded) will be presented with a prompt, indicating

² "JAR" stands for Java ARchive file; a JAR file is really just a ZIP archive containing a few special files within it.

who the applet is signed by, what authority vouches for the validity of the signature, and what extra outside-the-sandbox privileges the applet is asking for. The user can then decide whether or not to allow the applet to run with those extra privileges. The means by which the developer packages and signs the program, the format of the file the signed program is stored in, and the actual code used to request the privileges, are all complex, and very different between the two browsers. But the basic interaction with the user is similar, and is based on the usual point-and-click that end users are used to.

Function	Java 2 SDK	Netscape Communicator	Microsoft Internet Explorer
Mechanism for delivering Signed Java	Java 2 signed JAR	Netscape signed JAR	Microsoft signed CAB
Signature Verification Algorithm	DSA/SHA1	RSA/SHA1	RSA/MD5
Certificate handling at client	Sun "keystores" for keys and certificates	Netscape proprietary key and certificate management	Microsoft proprietary key and certificate management
Request for privileges	Applet attempts privileged action. An exception is thrown when action is not allowed.	Programmer defines the privileges required by calling privilege manager methods.	Code signer defines the privileges required when signing the CAB file.
Mechanism for storing policy information	Policy configuration files (can be located remotely) that are manually edited.	User prompted the first time privileges are requested. Proprietary method for storing persistent settings.	Privileges are configured in security zones that are stored in a Microsoft proprietary format.

The simplicity of the end-user's view of applet security in the Netscape and Microsoft browsers, as compared to the more complex command-line methods in version 1.1 of the Sun platform, is both an advantage and a disadvantage. The advantage, of course, is that it is more likely that end-users will actually be able to use it, and therefore that it will be feasible to deploy signed out-of-the-sandbox Java programs when the simpler interface is available. On the other hand, years of security experience show that users are all too ready to push "OK" on a prompt that they don't understand, just to get it out of the way so they can get on with their jobs. So there is some danger that a malicious user could create an applet with some destructive payload, sign it with an uncertified (technically, a "self-certified") signature, and trick users into running it, counting on at least some users to ignore all warnings from the browser, and push the "yes" or "grant" button just to get the prompt to go away. While we know of no actual attacks by this method, we do know of many infections by Word macro viruses that required users to push roughly-similar buttons despite roughly-similar warnings.

Java 2

The most recent release of a Java Software Development Kit from Sun comes with a new name and a new version number. The underlying technology and environment are now referred to as the Java 2 Platform (Standard Edition, Enterprise Edition, and so on). The version number of the most recent release is 1.2, which follows along nicely after versions 1.0 and 1.1 of the Java Development Kit. Java 2 is the next logical step in the evolution of the Java technology; don't let the name confuse you.

Java 2, version 1.2, exists in a number of forms. The programmer's toolkit is the Java 2 SDK (Software Development Kit); this is Sun's implementation of the Java 2 technology, and includes the compiler that produces Java ".class" files from human-readable Java programs, the "jar" tool for creating and manipulating JAR files, and so on. Also available is the Java 2 Runtime Environment (JRE), which

includes the interpreter that runs Java programs, but not the programmer's tools that create them; the Runtime Environment can be provided to users to allow them to run Java 2 programs (Java 2 programs that don't use any new Java 2 features will also generally run in older Java engines, such as those in existing releases of Web browsers). In order to allow users to take advantage of new Java 2 features in existing browsers, the Java 2 Runtime Environment also includes a "plugin" that allows the Netscape and Microsoft browsers to run Java 2 programs.

There are two ways the Java 2 plugin can be installed. The install program that contains the Java2 runtime environment can be obtained (for instance from the Sun website) and used to install Java 1.2 plugins for both Internet Explorer and Netscape Navigator. In order for a web page to use these plugins it must have HTML that explicitly directs an embedded Java applet to the plugins. Installation also happens semi-automatically when a Java 2 enabled web page is viewed by a Windows-based browser that does not have the Java 2 plugin installed. Assuming the web page uses the standard HTML sequence for invoking Java 2, the user is presented with a dialog indicating that she needs a plugin to view the web page. This dialog has a convenient "yes" button that more or less automatically downloads and installs the Java 2 plugin for that browser. After the Java 2 plugin is installed, the browser contains two Java virtual machines: the original built-in engine that is invoked by traditional web pages, and the Java plugin that is invoked by pages specifically targeting the Java 1.2 plugin.

While the Java 2 plugin is the primary way that users can get and use a Java 2 engine today, future releases of Web browsers and other active content systems will presumably come with Java 2 already built in. Netscape, for instance, has announced that version 5 of Netscape Communicator will include a "pluggable" Java interface, which will among other things allow replacing the standard Netscape Java engine with the Java 2 engine.

When the Java 2 security manager starts up, it reads security policy information from one or more policy files (for the gory details of policy file syntax and placement, we recommend references [3] and [4]). Each policy file can describe a number of principals, and for each principal specify a number of privileges that should be granted applets associated with those principals. A principal can be everyone, a particular signer, a particular place that a program can be loaded from, or a combination of a signer and a place. Some simple fictional examples:

```
// Everyone can do these things
grant {
    permission java.util.PropertyPermission "user.haircolor";
    permission java.io.FilePermission "c:\\temp\\*", "read,write";
};

// These guys can read and write the directory we set up for them
grant signedBy "GoodGuys Inc." {
    permission java.io.FilePermission "c:\\GoodGuys\\*", "read,write";
};

// Things signed by the security administrator and loaded
// from his server can do various magic
grant signedBy "secadmin", codeBase http://secadmin.example.com/admin/* {
    permission java.security.SecurityPermission "Security.insertProvider.*";
    permission java.security.SecurityPermission "Security.removeProvider.*";
    permission java.security.SecurityPermission "Security.setProperty.*";
};
```

Straightforwardly, these give any applet permission to access the (fictional) system property "user.haircolor", and permission to read and write files in a temporary directory, give any applet signed by GoodGuys Inc. permission to read and write files in a certain directory, and give programs signed by the

administrator and loaded from a particular place on a particular server permission to do some low-level security functions. There is considerable technology behind all this, of course; the Java 2 engine will not simply take an applet's word that it was written by the security administrator, for instance; the applet must be signed with a digital signing key that is present in a keystore under the name "secadmin", and that key must further be certified by a recognized signature-certifying authority. Again we refer the reader to references [3] and [4] for the gory details; we are more concerned with the higher-level implications.

The steps that a developer takes when constructing a signed applet that needs to get outside the sandbox in the Java 2 platform are similar to those he would take in Java 1.1. Some of the command names and switch details are different, but the essential steps of using a command-line program to create a JAR file and sign it remain the same. In the current (June, 1999) release of the Java 2 platform, the steps that the client has to carry out are also similar; a command-line program is used to register a certificate that verifies the signer's signature as really belonging to him, and the user edits his policy file (either with a text editor or with a simple Sun-provided GUI program called "policytool") to grant the signer the permissions that the applet requires.

As in previous versions of Sun's Java platforms, the steps required at a client machine to indicate that a signer is trusted are more cumbersome, but also less prone to unthinkingly trusting an attacker, than the mechanisms in the Netscape and Microsoft web browsers. Efforts are underway to find a good compromise between convenience and security; a way to allow users to trust those signers that they ought to trust, without encouraging them to trust anyone who asks indiscriminately. Sun security architect Li Gong writes in reference [4], "when the plug-in encounters previously unknown applets..., it needs a solution to give the applets their needed privileges in order to run them as they are designed. One approach to this deployment issue is to design a way for the developer to specify the required permissions and a way to help the browser user to understand the meaning and implications of granting those permissions."

Currently, there is little support for central security management, or enterprise-wide deployment of trusted applets, in the existing Java 2 implementations. It is possible to set up the Java 2 engine on a system in such a way that the security policy files used by the security manager are loaded from a Web site, rather than from the local system. This is a start toward centralized management, and we can expect to see more in that direction from Sun and other technology developers over the next year. Enterprise-wide deployment will require a more automatic way to add signatures and certificates to the client's keystores, without requiring the user to run confusing command-line tools; in reference [4], Li Gong suggests that the Java Plug-In should be enhanced to share the certificate database of the browser that it is running under, rather than using a separate database of its own. Similarly, there is currently no support for backward compatibility with the sandbox-escape mechanisms offered by the native Netscape and Microsoft Java engines; while those mechanisms are not widely used, there are certainly some enterprises that have come to depend on them, and some sort of conversion utility or compatibility layer would be desirable. The Java 2 Platform is still a new technology set as of this writing, and many of the issues that arise with respect to security implementation are still in flux.

Despite the lack of fully developed centralized administration tools, it is possible to configure a system so that both Java 2 plugins and Java applications use the same security manager and security policy. Details of how to tell the Java 2 security manager how to use a particular policy file, and how to apply the security manager and security policies even to locally-loaded applications, can be found in references [3] and [4]. Those responsible for Java 2 security (for a group of users or for themselves) should know how to check which security manager and what security policy files are being used on any particular system; hints on how to do this in the most common configurations can be found in the Appendix of this paper.

Java viruses?

The Java 2 security system provides more granular control than did previous versions of Java, but the range of available function in the sandbox is essentially the same. Consequently, there is no fundamentally new danger of viral spread in the new environment. In [5], we discuss the Trojan horse and Java virus threat,

and explain in some detail why the Java environment with appropriate security controls is not friendly either to viruses or to Trojan horses. The discussion presented there flows along two related paths, one for viruses and one for Trojan horses.

The basis of protection against Trojan horses comes from limiting what an untrusted Java applet can do to only those actions you feel cannot hurt you beyond a threshold you are comfortable with. So for instance, an arbitrary untrusted Java applet may be able to display something you don't want to look at, or make a sound you don't like, but it cannot steal your private information or make purchases on your behalf. Java has well-thought-out and well-executed limitations on what untrusted code can do, which historically has been effective at limiting such abuse to the nuisance category. There are always dangers associated with bugs in the JVM (Java Virtual Machine) that can be exploited by applets to obtain more privileges than the host intends. The best defense against bugs is to keep JVMs up to date and for responsible parties to be aware of possible exploits published by the JVM providers.

Last year we reported on the only known example of a Java virus; a demonstration called "Strange Brew". Although Strange Brew was released, it was never observed to spread in the wild. There have been a handful of similar attempts since then, but none have been viable viruses. We credit the lack of virus activity with Java applets to the environment in which applets execute. Unlike programs, document files, and spreadsheets, people do not share Java applets often, and when they do, the environment that the applets execute in virtually never allow them write access to applets that other people use. This environment makes Java viruses unlikely candidates for worldwide spread. As long as these environmental conditions exist, Java viruses are unlikely to be a factor. As future uses for Java technology appear, it is important for the industry to ensure that this continues to be the case.

Calling All Java Engines

As we have implied above, there are opportunities for many different Java engines to coexist on a single machine. For example, imagine a Windows machine with Internet Explorer, Netscape Navigator, and a recent release of Lotus Notes installed. Suppose you've run the Java 1.2 install program provided by Sun. In that case you have a JVM developed by Microsoft inside Internet Explorer, a JVM developed by Netscape inside Navigator, a JVM developed by Sun inside Lotus Notes, and the Java plugins from Sun inside both browsers for a grand total of four different JVMs in five different locations using four different signature databases and four different sets of security settings. Needless to say, security related issues would be far easier to handle if all of these different implementations shared a common security configuration. As of this writing, we know of no generally-available tools to tie all of these related systems into a common management structure.

So What?

So is Java still secure? The short answer is yes. For unsigned or untrusted applets, the Java security manager still enforces the same sandbox controls that it always has. The occasional security hole in a Java implementation is still discovered, but no exploits are known. There are a number of demonstration "hostile applets" that make annoying sounds, or may cause your browser to crash, but there are no viable Java viruses or actively malicious Java-based attacks known to the security community.

The longer answer is that the Java platform is still evolving, and at the moment it is in a stage of considerable complexity, and some confusion. The fact that the same Web page can host two different Java applets, running under two different Java engines, with two different and incompatible security and signature technologies, increases the possibility that somewhere, something is misconfigured. The fact that security policy in some implementations is determined by the individual user, rather than centrally

administered, and that it sometimes involves using obscure command-line tools and editing delicately-formatted files, also increases the likelihood of misconfiguration. On the other hand, the fact that a user can tell his Web browser to trust applets signed by John Q. Badguy with just a couple of mouse clicks is also cause for concern.

The news is not all bad, though! Few of these things actually happen in real life. The vast majority of applets on the Web run strictly within the sandbox, and when an enterprise needs to grant more permissions to some key applets in the intranet, the barriers to deployment can be overcome. Most users don't realize that they could potentially change the security configuration of the Java engine, and so never have the opportunity to do it wrong. Over the next several months, we can expect to see improved user interfaces to the security settings of the Java 2 platform, and better integration between the Java 2 engine and the popular Web browsers. Java still provides a powerful and secure active-content system, and Java 2 represents the next step in its evolution.

Acknowledgements

Thanks are due to the support folks at eng.sun.com for their quick answers to some last-minute questions, and to Tracy Russ at Addison-Wesley for sending us information that we needed before we even knew we needed it. All errors are, of course, our own.

References

[1] See <http://java.sun.com/>

[2] Koved, Nadalin, Neal, and Lawson, "The Evolution of Java Security", http://www.ibm.com/security/html/wp_javaevol.html

[3] Marco Pistoia et al, "Java 2 Network Security", IBM order number SG24-2109-01, <http://www.redbooks.ibm.com/abstracts/sg242109.html>,
<http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg242109.pdf>

[4] Li Gong, "Inside Java 2 Platform Security", Addison-Wesley, Reading Mass., 1999, ISBN 0-201-31000-7, <http://www.javasoft.com/docs/books/security/>

[5] Morar and Chess, "Web Browsers: Threat or Menace", Proceedings of the 1998 Virus Bulletin Conference, <http://www.av.ibm.com/InsideTheLab/Bookshelf/ScientificPapers/Chess/Threat/Threat.html>

Appendix - Java 2 security settings

Important Java 2 security settings for the Java 2 platform's security manager are stored in the file "java.security", which is in the "lib/security" directory under the directory in which the Java 2 runtime environment is installed. For instance, the security settings file might be in "c:\jdk1.2\src\lib\security\java.security" on a Windows system on which the Java Software Development Kit is installed, and in "c:\Program Files\javasoft\JRE\1.2\lib\security\java.security" on a system where only the JRE is installed. A typical java.security file will contain lines like:

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

These lines list security-policy files, from which the security manager will take lists of principals and the privileges that should be granted to them. "\${java.home}" refers to the directory in which the Java 2 runtime environment is installed. "\${user.home}" refers to the user's home directory, which will be the usual home directory on a Solaris or Unix-derived system, and will be something like "c:\windows" or "d:\users\default" or equivalent on a Windows system.

In determining the current security settings on a system, you should read and understand the contents of the java.security file, and all files (or URLs) listed on the "policy.url.n" lines within it. On a typical Windows installation where the file c:\Program Files\javasoft\JRE\1.2\lib\security\java.security contains the two lines above, the policy files in effect would be c:\Program Files\javasoft\JRE\1.2\lib\security\java.policy, and c:\windows\java.policy. The latter file, designed to hold user customizations of the security policy, will often not exist. See the main text of this paper for a few examples of security policy entries, and see references [3] and [4] for detailed treatments of the syntax and semantics of both the java.security and security-policy files.