Java Security

Joseph A. Bank
jbank@mit.edu

December 8, 1995

1 Introduction

Java is a new programming language from Sun Microsystems (currently in beta release). The Java language has a number of interesting properties. One property is that it is intended to be portable, even to the extent that programs can be dynamically loaded over the network and run locally. In particular, small programs called applets can be loaded and run by a user's WWW browser while the user is "surfing" the Web (HotJava is such a browser written in Java, and Netscape2.0 will support Java applets). While this idea is very powerful, it is also an invitation to security problems. The Java language and runtime system (which includes libraries, the compiler, and the bytecode interpreter) attempt to address these security issues, with the result that Sun claims Java will be secure.

This paper evaluates the security issues raised by the Java language and its intended uses in Java enabled Web browsers and Java's proposed solutions. After a brief discussion on the background of executable content, this paper moves on to discuss the potential security risks of executable content, what Java's proposed solutions are, and finally an analysis of the effectiveness of those solutions.

1.1 Background on Executable Content

Executable content is the idea of sending around data that is actually code to be executed. Why is the idea of executable content so exciting? The answer is fairly simple. Power and expressiveness. Use of the World Wide Web has exploded over the past few years, along with this growth there have been many attempts to retrofit applications to the Web. While the Web has adapted to allow more interesting uses through forms and scripts that run on the server, these methods are extremely limiting. The ability to have users locally run a program written in a full-fledged programming language allows applications to be used directly over the Web.

Not surprisingly Java is not alone with its idea of shipping around programs to run. For example, the Safe-Tcl language [1] (an extension of the Tcl language) attempts to provide for "Enabled Mail" which would allow users to send email with embedded Safe-Tcl programs. Safe-Tcl goes through great troubles to make sure that the language satisfies strong security and portability constraints. Telescript from General Magic also provides many of the same features for executable content as Java, with similar claims regarding safety and security.[3] So why has Java received so much attention? The main reason is the Java is being supported by both Sun and Netscape. The consensus seems to be that if any language that supports executable content will make it, Java will.

2 The Problem

Before moving on to a discussion of Java security, one should have an understanding of the potential problems raised by executable content. The advantages of executable content come from the increase in power and flexibility provided by software programs. The increased power of Java applets (the Java term for executable content) is also the potential problem. When a user is "surfing" the Web, they should not have to worry that an applet may be deleting their files or sending their private information over the network surreptitiously.

The essence of the problem is that running programs on a computer typically gives that program access to certain resources on the host machine. In the case of executable content, the program that is running is untrusted. If a Web browser that downloads and runs Java code is not careful to restrict the access that the untrusted program has, it can provide a malicious program with the same ability to do mischief as a hacker who had gained access to the host machine. Unfortunately, the solution is not as simple as completely restricting a downloaded programs access to resources. The reason that one gives programs access to resources in the first place is that in order to be useful a program needs to access certain resources. For example a text editor that cannot save files is useless. Thus, if one desires to have useful and secure executable content, access to resources needs to be carefully controlled. The next section takes the first step, which is to identify the resources that we are concerned about. After the resources have been identified, some example scenarios which illustrate the problems with not providing sufficient limitations are presented.

2.1 What Needs Restrictions?

An important part of creating a safe environment for a program to run in is identifying the resources and then providing certain types of limited access to these resources. Table 1 provides a partial list of a typical host's resources along with a classification of some of the types of attacks which can be associated with availability of that resource. The four types of attacks are:

- disclosure of information about a user or the host machine

Resource	Disclosure	Availability	Integrity	Annoyance
File system	x	x	x	x
Network	x			x
Random Memory	x	x	x	x
Output Devices (CRT, Speaker, etc)				x
Input Devices (Keyboard, Microphone, etc)	x	x		x
Process Control		x		x
User Environment	x		x	x
System Calls	x	x	x	x

Table 1: Host Resources

- denial of service attacks make a resource unavailable for legitimate purposes (i.e. filling the file system)
- damaging or modifying of data, this could include data in use by other programs or by the file system
- annoyance attacks such as displaying obscene pictures on a user screen.

Note that the table is not intended to be complete in terms of possible types of attacks, but merely provides an example of the types of problems associated with a given resource. For example, for a spoofing program (a program that appears to the user to be a different program) may desire to utilize any given resource for its attack since it should appear to the user to use the same resources as the original program.

Some of the given resources are clearly more “dangerous” to give full access to than others. For example it is hard to imagine any security policy in which an unknown program should be given full access to the file system. On the other hand, most security policies would not limit a program from almost full access to the display (assuming the program was limited in other ways).

2.2 Scenarios

This section gives a few examples of scenarios that show what types of attacks from malicious programs can occur. The classifications of attacks will be the same as those mentioned in Section 2.1. Of course, this list of potential attacks is not intended to be complete, but rather give a flavor of the types of problems that can arise.

- Integrity Attacks
  - Deletion/Modification of files.

  - Modification of memory currently in use.
  - Killing processes/threads.

- Availability Attacks
  - Allocating large amounts of memory.
  - Creating thousands of windows.
  - Creating high priority processes/threads.

- Disclosure Attacks
  - Mailing information about your machine (i.e. /etc/passwd).
  - Sending personal or company files to an adversary or competitor over the network.

- Annoyance Attacks
  - Displaying obscene pictures on your screen.
  - Playing unwanted sounds over your computer.

3 Java's Approach

Java is a programming language. It is important to remember that Java is intended to be used for both stand alone applications and applets that are executed by Java enabled Web Browsers. Thus, the discussion here of Java security is really a discussion of the intended use of the Java language as providing a mechanism for executable content.

The Java approach to providing executable content is to have Web Browsers with an embedded Java interpreter and runtime library. These Web Browsers can download Java programs called applets, and have the Java interpreter execute the program. With this model, there are three fundamental layers: the Java language itself, the standard set of Java libraries, and the Web Browser itself. The security of the system depends fundamentally on the security of each of these three layers.

3.1 The Language

Java is an object oriented language with a syntax that is very similar to that of C++. The important features of the language from a security standpoint are the use of access control for variables and methods within classes, the safety of the type system, the lack of pointers as a language data type, the use of garbage collection (automatic memory deallocation), and the use of packages with distinct namespaces.

Java, like C++, has facilities for controlling the access to the variables and methods of objects. These access controls allow objects to be used by non-trusted code with the guarantee that they will not be used improperly. For example, the Java library contains a definition for a File object. The File object has a `public` method (callable by anyone) for reading and a low level `private` method (only callable by the objects methods) for reading. The public read call first performs security checks and then calls the private read. The Java language ensures that non-trusted code can safely manipulate a File object, providing only access to the public methods. Thus, the access control facilities allows programmers to write libraries which are guaranteed by the language to be safe by correctly specifying the library's access controls.

A second facility for providing access control is the ability to declare classes or methods as `final`. This provides the ability to prevent a malicious programmer from subclassing a critical library class or overriding the methods of a class. Thus, the language guarantees that the actual method that is invoked on an object is the finalized method that was written for the object's compile time type.[1] This provides a guarantee that certain parts of an object's behavior have not been modified.

The Java language is also designed to be a type-safe language. This means that the compile time type and the runtime type of variables are guaranteed to be compatible. This ensures that casts (operations that coerce a runtime type to a given compile time type) are checked at either compile time or runtime to make sure that they are valid. This prevents the forging of access to objects to get around access control. Using our File example from before, this prevents the malicious code from casting a File object to the malicious code's MyFile type which has the same layout as the File type, but with all methods public.

Another safety feature is the elimination of pointers as a data type. This means that pointers cannot be directly manipulated by user code (no pointer arithmetic). This prevents both malicious and accidental misuse of pointers (running off the end of an array for example). Again using our File example, this prevents the malicious code from simply accessing the private method directly by using pointer arithmetic starting with the File object's pointer. Clearly this type-safety is a necessary part of the access control facilities of objects, preventing forging (note that this safety is clearly lacking in C++).

The Java language uses garbage collection to recover unused memory instead of relying on explicit user deallocation. This not only eliminates an extremely common class of bugs, but eliminates potential security holes. For example, if Java had manual deallocation, this could provide a round-about way of illegally casting. First, the malicious code creates a new object of type MyFile, and then deallocates the memory used by that object, keeping the pointer. Then, the malicious code immediately creates a File object which happens to have the same size. If this is done carefully (with knowledge of how the allocation and deallocation of memory works), the new pointer to the File object is the same as the original MyFile pointer. The malicious code can now access the private methods of the File object with the MyFile pointer.

Finally, the Java language uses packages (similar to modules in Modula-3, or pack-

[1] Java uses method dispatch based on the runtime type of objects, so without final methods this is not guaranteed.

ages in Common Lisp) to provide namespace encapsulation. From a safety standpoint, packages are useful because they allow downloaded code to be easily distinguished from local code. In particular, this prevents downloaded code from shadowing system library code with malicious code. The Java language guarantees that when a class is referenced the system first looks in the local namespace, and then in the namespace of the referencing class. This also guarantees that a local class cannot accidently reference a downloaded class.

3.2 The Libraries

The standard Java runtime environment comes with a variety of useful libraries, providing file system access, network access, a window toolkit, and a variety of other tools. The correct specification of the libraries is of critical importance. The language itself can provide the ability to create secure libraries, but if the library code is not specified and written correctly the system is not secure. Since the libraries are the part of the Java runtime that provides access to the system resources mentioned in Section 2.1, the correct implementation of the libraries is of fundamental importance.

The access restrictions of the libraries are based on three mechanisms. The first is the Java language mechanism of providing access restrictions to object methods and variables mentioned in Section 3.1. The second mechanism is the use of specialized `ClassLoader`s to load imported code. The final mechanism is the use of explicit calls to a global `SecurityManager` to check the validity of certain specific operations.[2]

3.2.1 ClassLoader

The Java runtime has two distinct ways of loading a new class. The default mechanism is to load a class from a file on the local host machine. This mechanism does not need a ClassLoader. Any other way of loading a class, such as over the network, requires an associated ClassLoader (i.e. a subtype of the `ClassLoader` class which has some specialized methods). The ClassLoader is responsible for converting the raw data of a class (e.g. bytes transmitted over the network) into an internal data structure representing that class.

In order to have Java applets be as portable as possible, the Java compiler does not compile to machine code, instead it compiles to bytecodes for an architecture independent virtual machine. The Java interpreter runs a program by interpreting these bytecodes. Thus, applets are transmitted in bytecode form instead of source code or machine code. This means that the ClassLoader only deals with bytecode.

For security reasons, the ClassLoader cannot make any assumptions about the bytecode. The bytecode could have been created from a Java program compiled with the

[2]The global SecurityManager variable defaults to null. It is only set by applications such as Web Browsers which need to have a security policy. Thus, stand alone applications pass the SecurityManager checks automatically.

Java compiler, or it could have been created by a C++ program compiled with a special compiler for the virtual machine.[3] This situation means that the ClassLoader must verify that the bytecode does not violate the safety that Java guarantees. Aside from simple format checks, the bytecode verifier checks:[5][7]

- that it doesn't forge pointers
- that it doesn't violate access restrictions
- that it accesses objects as what they are
- that it calls methods with appropriate arguments of the appropriate type
- that there are no stack overflows

Along with checking the validity of the bytecode, the ClassLoader has the responsibility of creating a namespace for downloaded code, and resolving the names of classes referenced by the downloaded code.

3.2.2 SecurityManager

In the current release of Java (beta), the SecurityManager is not well documented. Nevertheless, from examining the code released with the Java beta release and reading what documentation does exist, a good deal can be determined about what the SecurityManager is intended to do. The SecurityManager contains a number of methods which are intended to be called to check specific types of actions. Figure 3.2.2 provides a full list of the public methods with their intended uses. The SecurityManager class itself is not intended to be used directly (each of the checks defaults to throwing a security exception), instead it is intended to be subclassed and installed as the System SecurityManager. The subclassed SecurityManager can be used to instantiate the desired security policy.

The SecurityManager provides an extremely flexible and powerful mechanism for conditionally allowing access to resources. The SecurityManager methods which check access are passed arguments which are necessary to implement conditional access policies, as well as having the ability to check the execution stack to determine if the code has been called by local or downloaded code.

The standard metaphor for creating library code for a potentially dangerous system resource is to only provide access to operations that are not dangerous, and to wrap a security check (via the SecurityManager) around calls that access is provided to on a limited basis.

[3] This is actually useful since it allows other languages to compile to Java bytecode as long as the other languages bytecode compiler ensures that they "play by the rules".

Method	Description
getInCheck	Determine whether a security check is in progress
checkCreateClassLoader	Check to prevent the installation of additional ClassLoaders.
checkAccess	Check to see if a thread or thread group can modify the thread group.
checkExit	Checks if the Exit command can be executed.
checkExec	Checks if the system commands can be executed.
checkLink	Checks if dynamic libraries can be linked (used for native code).
checkRead	Checks if a file can be read from.
checkWrite	Checks if a file can be written to.
checkConnect	Checks if a network connection can be created.
checkListen	Checks if a certain network port can be listened to for connections.
checkAccept	Checks if a network connection can be accepted.
checkProperties	Checks if the System properties can be accessed.
checkTopLevelWindow	Checks whether a window must have a special warning.
checkPackageAccess	Checks if a certain package can be accessed.
checkPackageDefinition	Checks if a new class can be added to a package.
checkSetFactory	Check if an applet can set a networking-related object factory.

Table 2: SecurityManager public methods

```
public boolean mkdir(String path) throws IOException {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkWrite(path);
    }
    return mkdir0();
}
```

Figure 1: Example of security check. This example shows the basic metaphor: the public method `mkdir` checks the system `SecurityManager` (which will throw an exception if the check does not pass) and then calls the low level private method `mkdir0`.

3.3 Java Enabled Browsers

The Web browser itself plays a large role in the security of the system. The Web browser defines and implements a security policy for running downloaded Java code. A Java enabled Web browser will include a Java interpreter and runtime library along with classes added to implement a SecurityManager and various ClassLoaders. From a security standpoint, the Web browser's implementation of the SecurityManager is much more critical than the implementation of the ClassLoaders.[4] Some discussion of how to increase security via the ClassLoader is discussed in Section 4.2.2.

The SecurityManager controls the access to critical system resources. This allows the writer of a Web browser to implement a specific security policy by subclassing the SecurityManager and overriding certain methods, and then installing the new version as the system SecurityManager. Since the subclassed SecurityManager implements the security policy, it is critical that the Web browser's version of the SecurityManager is implemented correctly. In the extreme, if a Java enabled Web browser did not install a system SecurityManager, an applet would have the same access as a local Java application.

The Web browser's security policy can be made arbitrarily complex since the SecurityManager hooks provide a flexible interface. Any policy that can be programmed can be used. For example, the policy can have the SecurityManager query the user with information regarding any particular requested access.

4 Analysis

The analysis of the security of the model for executable content provided by Java will be done in two sections. In the first section, the high level design of Java will be discussed. In the final section, the validity of more specific parts of the implementation will be discussed.

4.1 Design

The Java security model is based almost entirely on the ability to verify the downloaded bytecode, the ability to specify and write libraries that prevents undesired access to resources, and the ability of Web browser developers to specify and write code that implements good security policies.

4.1.1 Why bytecodes?

The decision to transfer Java programs in compiled bytecode form is a questionable decision from a security standpoint. The system might be more secure if Java source

[4] The security critical calls of the ClassLoader class are declared as final, so they cannot be overridden by a subclass.

code was used instead of bytecode. The use of bytecodes requires the process of bytecode verification to make sure that the bytecode does not violate the requirements of the Java language. If instead Java source code was used, the safety of the code could be ensured by interpreting the source directly or compiling the Java code locally with a trusted compiler. This would be a simpler approach, since the Java compiler is already trusted (the Java runtime system does not perform verification on local bytecodes). The addition of a program that checks if programs in another language (the bytecode) is compatible adds another point of failure to the system.

There are a number of possible reasons for using bytecode. First, it provides a certain obfuscation, preventing reverse engineering of Java programs. While this is certainly true, the bytecodes are not impossible to decompile, and there are other code obfuscation techniques that manipulate the source directly. Second, the bytecode representation may be somewhat smaller. Currently, this is not true. In every current example, including the entire HotJava browser source, the source and bytecode are almost identical in size. Third, the process of bytecode verification may be faster than the process of compilation. Thus, the process of downloading and running programs is less computationally intensive and provides lower latency. This argument is fairly compelling, although the Java team has suggested that they might eventually use a "Just in Time" compiler for computationally intensive programs, suggesting the compilation is not prohibitive.[5] A final argument can be made that the bytecode verifier is a less complex program than a full compiler, thus the verification of the correctness is a simpler process. In fact, the compiler does not have to be trusted since all bytecodes could be verified with the trusted bytecode verifier (the local code could be verified only once when it is compiled for efficiency purposes).

4.1.2 Writing Secure Libraries

The process of writing and specifying good libraries cannot be overemphasized in the importance of Java security. The paper *Security Flaws in the HotJava Web Browser*[2] by Drew Dean and Dan Wallach gives a number of good examples of how the security of the HotJava Web browser is compromised by errors in implementation of the libraries. Dean and Wallach point out flaws in the implementation of the 1.0 alpha 3 release libraries. They show that a number of variables and methods are made publicly accessible and do not have security checks, allowing certain breaches of security. While these specific problems have been fixed in the beta version of the Java runtime libraries, they show that correct specification is not a simple task.

The problem of security is made more difficult by the fact that even if the libraries are written properly, the writer of a Java enabled Web browser must correctly implement and specify their part of the SecurityManager. The SecurityManager is an improvement over the previous versions more spread out mechanisms, providing a specific and well-

[5] Since a JIT compiles bytecode to machine code, this process may be far simpler than normal compilation.

encapsulated way of flexibly setting up a security policy. Nevertheless, the fact that a Web browser has direct hooks into the security of the Java system increases the potential for errors in specification and implementation. While Java itself may undergo fairly rigorous scrutiny, it is quite possible the Java enabled browsers may not be as careful, relying on Java's built in security.

The security of Java thus relies upon the correct implementation of a fairly large code base. This situation is a result of Java's design choice to provide a very flexible security model. By giving application writers hooks into the systems security, they have offered flexibility at the price of opening potential holes.

4.2 Implementation

In analyzing the effectiveness of Java security, it is necessary to check if there are adequate methods of controlling each of the resources specified in Section 2.1.

- **File system -** Access to the file system is well protected, with specific SecurityManager checks for read and write access to a given file. This allows flexible policies such as access control lists to be fairly easily implemented.

- **Network -** Access to the network is well protected. There are SecurityManager checks on the methods necessary for both accepting and creating of sockets, as well as protection of calls to other network related methods. Again, these methods are flexible, allowing access controls.

- **Random Memory -** Protection of memory is done by the language specification itself. There is protection against access to the already allocated memory, but there is no protection against an applet allocating all of the current memory available (by creating objects).

- **Output Devices -** One protection provided for output devices is that any applet windows can be forced to have a special marking noting that they are unsafe. Additionally, an applet cannot directly access devices, instead it must use the mechanisms provided by the Java libraries.

- **Input Devices -** An applet can only access the keystrokes or mouse clicks of the user when the applet's window has been selected. Currently other input devices are not supported, although one would assume that any access to devices such as camera's or microphones would be through a Java library which could add security checks. Currently there are no explicit security checks involving input.

- **Process Control -** Access to control of threads is fairly limited. There are SecurityManager checks of access the threads and threadgroups. Additionally, the maximum priority of threads can be set to make sure an applet's thread does not dominate.

- **User Environment -** Access to environment variables is protected by SecurityManager checks. Additional access control to the Java language environment is provided by the languages access control mechanisms for classes.

- **System Calls -** The SecurityManager checks any attempted system calls, including attempts to exit.

The control of a few of these resources stand out. First, the current SecurityManager does not have a method for controlling the creation of top level windows (aside from forcing them to be marked as unsafe), or control of what can be displayed or played back audibly. Second, there is no mechanism for controlling an applet's access to user input. There are certainly situations it would be desirable to have a more specific security policy regarding various input devices. Finally, an applet can currently allocate an arbitrary amount of memory by creating new objects. The problem of allocating memory is difficult because it does not provide a very direct threat; there is no single operation or set of operations that can be controlled. The problem is not horrible since the browser can limit the amount of memory available to Java. The browser could also provide a method of killing the current Java applet, causing the memory to be recovered.

4.2.1 Scenarios

Given the analysis of the access controls to resources, it is interesting to see how effective Java could be against the various scenarios mentioned in Section 2.2.

- **Integrity Attacks -** Each of the mentioned integrity attacks can easily be prevented by the access control capabilities. The malicious modification of files, memory, and threads can be prevented.

- **Availability Attacks -** The availability attacks are much harder to prevent. As was previously mentioned, there is no current limitation to prevent the allocation of all the memory available to Java or the creation of thousands of windows. Java does have the ability to place some control on the creation of high priority threads.

- **Disclosure Attacks -** Each of the mentioned disclosure attacks can easily be prevented by the access control capabilities. Java provides mechanisms that both prevent an applet from accessing sensitive information, as well as preventing the creation of channels to deliver data. Since either one of these would be sufficient to stop disclosure attacks, the combination is sufficient.

- **Annoyance Attacks -** Since graphics and audio are currently impossible to screen based on content, annoyance attacks cannot be prevented without taking the extreme position that no downloaded data will be shown or heard. Java provides this particular alternative (don't use it to download anything), but does not provide anything more flexible.

The given analysis shows that Java is effective at preventing the more dangerous types of attacks. It should be noted that the annoyance attacks which were mentioned are just as applicable to current Web browsers which do not use Java. The problem of denial of service attacks is also fairly difficult to prevent entirely. One can imagine a security policy that prevents the creation of more than 10 windows, or prevents the use of more than 100Kbytes of memory, but these types of restrictions seem arbitrary. Instead, it would be desirable to have Web browsers that allowed the user to explicitly kill an applet and all of the resources that it is using. Hopefully such a mechanism will be implemented.

4.2.2 Digital Signatures

The use of digital signatures as a mechanism for verifying that code comes from a trusted source could play an important part in the future of Java security. Currently there is no built in mechanism for allowing code that is verified to have come from a trusted source have special access to resources. The current Java security mechanism does seem flexible enough to allow the addition of digitally signed applets. The ClassLoader class can be subtyped to create a SignedClassLoader which first does the digital signature verification, and then does the actual loading of the class. The various SecurityManager methods can then check if the call is in the dynamic scope of a SignedClassLoader in order to determine whether access should be allowed or denied. Thus, the current mechanism certainly allows the writers of Web browsers to add special access for digitally signed code. One might hypothesize that it is only for legal rather than technical reasons that this scheme is not part of the current release.

5 Conclusion

Some of the discussion about the security of Java needs to be put into perspective. The reason that Java was desirable in the first place was that it provided increased power and flexibility. There is an inevitable trade off between this increase in power and the security risk of a system using Java. The security measures of Java provide the ability to tilt this balance whichever way is preferable. For a system where security is of paramount importance, using Java does not make sense; it is not worth the added security risk. For a system such as a home computer, many people are likely to find that the benefits of Java outweigh the risks. By this same token, a number of systems are not connected to the Internet because it is a security risk that outweighs the benefits of using the Internet. Anyone that is considering using Java needs to understand that it does increase the security risk, but that it does provide a fairly good “firewall” (to extend the Internet connection example).

References

[1] Nathaniel S. Borenstein, *Email With a Mind of Its Own: The Safe-Tcl Language for Enabled Mail.* In *Proceedings of ULPAA* (1994).

[2] Drew Dean and Dan S. Wallach, *Security Flaws in the HotJava Web Browser*, November 3, 1995.
Available via `ftp://ftp.cs.princeton.edu/reports/1995/501.ps.Z`

[3] General Magic, Inc. Available via *An Introduction to Safety and Security in Telescript.*
Available via `http://cnn.genmagic.com/Telescript/TDE/security.html`

[4] James Gosling and Henry McGilton, *The Java Language Environment: A White Paper*, Sun Microsystems, May 1995.
Available via `ftp://java.sun.com/docs/JavaBook.ps.tar.Z`

[5] Sun Microsystems, *HotJava(tm): The Security Story.*
Available via `http://java.sun.com/1.0alpha3/doc/security/security.html`

[6] Sun Microsystems, *The Java Language Specifications: Version 1.0 Beta.*
Available via `http://java.sun.com/JDK-beta/psfiles/javaspec.ps`

[7] Frank Yellin, *Low Level Security in Java.*
Available via `http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html`