Java Security: From HotJava to Netscape and Beyond*

Drew Dean
ddean@cs.princeton.edu

Edward W. Felten
felten@cs.princeton.edu

Dan S. Wallach
dwallach@cs.princeton.edu

Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

The introduction of Java applets has taken the World Wide Web by storm. Information servers can customize the presentation of their content with server-supplied code which executes inside the Web browser. We examine the Java language and both the HotJava and Netscape browsers which support it, and find a significant number of flaws which compromise their security. These flaws arise for several reasons, including implementation errors, unintended interactions between browser features, differences between the Java language and bytecode semantics, and weaknesses in the design of the language and the bytecode format. On a deeper level, these flaws arise because of weaknesses in the design methodology used in creating Java and the browsers. In addition to the flaws, we discuss the underlying tension between the openness desired by Web application writers and the security needs of their users, and we suggest how both might be accommodated.

1. Introduction

The continuing growth and popularity of the Internet has led to a flurry of developments for the World Wide Web. Many content providers have expressed frustration with the inability to express their ideas in HTML. For example, before support for tables was common, many pages simply used digitized pictures of tables. As quickly as new HTML tags are added, there will be demand for more. In addition, many content providers wish to integrate interactive features such as chat systems and animations.

Rather than creating new HTML extensions, Sun popularized the notion of downloading a program (called an applet) which runs inside the Web browser. Such remote code raises serious security issues; a casual Web reader should not be concerned about malicious side-effects from visiting a Web page. Languages such as Java[9], Safe-Tcl[3], Phantom[8], and Telescript[10] have been proposed for running downloaded code, and each has varying ideas of how to thwart malicious programs.

After several years of development inside Sun Microsystems, the Java language was released in mid-1995 as part of Sun's HotJava Web browser. Shortly thereafter, Netscape Communications Corp. announced they had licensed Java and would incorporate it into version 2.0 of their market-leading Netscape Navigator Web browser. With the support of at least two influential companies behind it, Java appears to have the best chance of becoming the standard for executable content on the Web. This also makes it an attractive target for malicious attackers, and demands external review of its security.

Netscape and HotJava[1] are examples of two distinct architectures for building Web browsers. Netscape is written in an unsafe language, and runs Java applets as an add-on feature. HotJava is written in Java itself, with the same runtime system supporting both the browser and the applets. Both architectures have advantages and disadvantages with respect to security: Netscape can suffer from being implemented in an unsafe language (buffer overflow, memory leakage, etc.), but provides a well-defined interface to Java. In Netscape, Java applets can name only those functions and variables explicitly exported to the Java subsystem. HotJava, implemented in a safe language, does not suffer from potential memory corruption problems, but can accidentally export too much of its environment to applets.

In order to be secure, such systems must limit applets'

*To appear in the 1996 IEEE Symposium on Security and Privacy, Oakland, CA, May 6–8, 1996. Copyright 1996 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

[1] Unless otherwise noted, "HotJava" refers to the 1.0 alpha 3 release of the HotJava Web browser from Sun Microsystems, "Netscape" refers to Netscape Navigator 2.0, and "JDK" refers to the Java Development Kit, version 1.0, from Sun.

access to system resources such as the file system, the CPU, the network, the graphics display, and the browser's internal state. The language's type system should be *safe* – preventing forged pointers and checking array bounds. Additionally, the system should garbage-collect memory to prevent memory leakage, and carefully manage system calls that can access the environment outside the program, as well as allow applets to affect each other.

The Anderson report[2] describes an early attempt to build a secure subset of Fortran. This effort was a failure because the implementors failed to consider all of the consequences of the implementation of one construct: assigned `GOTO`. This subtle flaw resulted in a complete break of the system.

The remainder of this paper is structured as follows. Section 2 discusses the Java language in more detail, section 3 gives a taxonomy of known security flaws in HotJava and Netscape, section 4 considers how the structure of these systems contributes to the existence of bugs, section 5 discusses the need for flexible security in Java, and section 6 concludes.

2. Java Semantics

Java is similar in many ways to C++[31]. Both provide support for object-oriented programming, share many keywords and other syntactic elements, and can be used to develop standalone applications. Java diverges from C++ in the following ways: it is type-safe, supports only single inheritance (although it decouples subtyping from inheritance), and has language support for concurrency. Java supplies each class and object with a lock, and provides the `synchronized` keyword so each class (or instance of a class, as appropriate) can operate as a Mesa-style monitor[21].

Java compilers produce a machine-independent bytecode, which may be transmitted across a network and then interpreted or compiled to native code by the Java runtime system. In support of this downloaded code, Java distinguishes remote code from local code. Separate sources[2] of Java bytecode are loaded in separate naming environments to prevent both accidental and malicious name clashes. Bytecode loaded from the local file system is visible to all applets. The documentation[15] says the "system name space" has two special properties:

1. It is shared by all "name spaces."

2. It is always searched first, to prevent downloaded code from overriding a system class.

[2]While the documentation[15] does not define "source", it appears to mean the machine and Web page of origin. Sun has announced plans to include support for digital signatures in a future version.

However, we have found that the second property does not hold.

The Java runtime system knows how to load bytecode only from the local file system. To load code from other sources, the Java runtime system calls a subclass of the abstract class `ClassLoader`, which defines an interface for the runtime system to ask a Java program to provide a class. Classes are transported across the network as byte streams, and reconstituted into `Class` objects by subclasses of `ClassLoader`. Each class is tagged with the `ClassLoader` that loaded it. The `SecurityManager` has methods to determine if a class loaded by a `ClassLoader` is in the dynamic call chain, and if so, where. This nesting depth is then used to make access control decisions.

Java programmers can combine related classes into a package. These packages are similar to name spaces in C++[32], modules in Modula-2[33], or structures in Standard ML[25]. While package names consist of components separated by dots, the package name space is actually flat: scoping rules are not related to the apparent name hierarchy. A (package, source of code) pair defines the scope of a Java class, method, or instance variable that is not given a `public`, `private`, or `protected` modifier. In Java, `public` and `private` have the same meaning as in C++: Public classes, methods, and instance variables are accessible everywhere, while private methods and instance variables are only accessible inside the class definition. Java `protected` methods and variables are accessible in the class or its subclasses or in the current (package, source of code) pair; `private protected` methods and variables are only accessible in the class or its subclasses, like C++'s `protected` members. Unlike C++, `protected` variables and methods can only be accessed in subclasses when they occur in instances of the subclasses or further subclasses. For example:

```
class Foo {
  private protected int i;
  void SetFoo(Foo o) { o.i = 1; } // Legal
  void SetBar(Bar o) { o.i = 1; } // Legal
}

class Bar extends Foo {
  void SetFoo(Foo o) { o.i = 1; } // Illegal
  void SetBar(Bar o) { o.i = 1; } // Legal
}
```

The definition of `protected` was different in some early versions of Java; it was changed during the beta-test period to patch a security problem.

The Java bytecode runtime system is designed to enforce the language's access semantics. Unlike C++, programs are not permitted to forge a pointer to a function and invoke it directly, nor to forge a pointer to data and access it directly. If a rogue applet attempts to call a private method, the runtime system throws an exception, preventing the errant access.

Thus, if the system libraries are specified safely, the runtime system assures application code cannot break these specifications.

The Java documentation claims that the safety of Java bytecodes can be statically determined at load time. This is not entirely true: the type system uses a covariant[5] rule for subtyping arrays, so array stores require run time type checks[3] in addition to the normal array bounds checks. Unfortunately, this means the bytecode verifier is not the only piece of the runtime system that must be correct to ensure security. Dynamic checks also introduce a performance penalty.

2.1. Java Security Mechanisms

In HotJava, all of the access controls were done on an ad hoc basis which was clearly insufficient. The beta release of JDK introduced the `SecurityManager` class, meant to be a reference monitor[20]. The `SecurityManager` defines and implements a security policy, centralizing all access control decisions. Netscape also uses this architecture.

When the Java runtime system starts up, there is no security manager installed. Before executing untrusted code, it is the Web browser's or other user agent's responsibility to install a security manager. The `SecurityManager` class is meant to define an interface for access control; the default `SecurityManager` implementation throws a `SecurityException` for all access checks, forcing the user agent to define and implement its own policy in a subclass of `SecurityManager`. The security managers in both JDK and Netscape typically use the contents of the call stack to decide whether or not to grant access.

Java uses its type system to provide protection for the security manager. If Java's type system is sound, then the security manager should be tamperproof. By using types, instead of separate address spaces for protection, Java is embeddable in other software, and performs better because protection boundaries can be crossed without a context switch.

3. Taxonomy of Java Bugs

We now present a taxonomy of Java bugs, past and present. Dividing the bugs into classes is useful because it helps us understand how and why they arose, and it alerts us to aspects of the system that may harbor future bugs.

[3] For example, suppose that `A` is a subtype of `B`; then the Java typing rules say that `A[]` ("array of `A`") is a subtype of `B[]`. Now the following procedure cannot be statically type-checked:

```
void proc(B[] x, B y) {
  x[0] = y;
}
```

Since `A[]` is a subtype of `B[]`, `x` could really have type `A[]`; similarly, `y` could really have type `A`. The body of `proc` is not type-safe if the value of `x` passed in by the caller has type `A[]` and the value of `y` passed in by the caller has type `B`. This condition cannot be checked statically.

3.1. Denial of Service Attacks

Java has few provisions to thwart denial of service attacks. The obvious attacks are busy-waiting to consume CPU cycles and allocating memory until the system runs out, starving other threads and system processes. Additionally, an applet can acquire locks on critical pieces of the browser to cripple it. For example, the code in figure 1 locks the status line at the bottom of the HotJava browser, effectively preventing it from loading any more pages. In Netscape, this attack can lock the `java.net.InetAddress` class, blocking all hostname lookups and hence all new network connections. Both HotJava and Netscape have several other classes suitable for this attack. The attack could be prevented by replacing such critical classes with wrappers that do not expose the locks to outsiders. However, the CPU and memory attacks cannot be easily fixed; many genuine applications may need large amounts of memory and CPU.

There are two twists that can make denial of service attacks more difficult to cope with. First, an attack can be programmed to occur after some time delay, causing the failure to occur when the user is viewing a different Web page, thereby masking the source of the attack. Second, an attack can cause *degradation of service* rather than outright denial of service. Degradation of service means significantly reducing the performance of the browser without stopping it. For example, the locking-based attack could be used to hold a critical system lock most of the time, releasing it only briefly and occasionally. The result would be a browser that runs very slowly.

Sun has said that they consider denial of service attacks to be low-priority problems[14].

3.2. Two vs. Three Party Attacks

It is useful to distinguish between two different kinds of attack, which we shall call two-party and three-party. A two-party attack requires that the Web server the applet resides on participate in the attack. A three-party attack can originate from anywhere on the Internet, and might spread if it is hidden in a useful applet that gets used by many Web pages (see figure 2). Three-party attacks are more dangerous than two-party attacks because they do not require the collusion of the Web server.

3.3. Covert Channels

Various covert channels exist in both HotJava and Netscape, allowing applets to have two-way communication with arbitrary third parties on the Internet.

Typically, most HotJava users will use the default network security mode, which only allows an applet to connect

```
synchronized (Class.forName("net.www.html.MeteredStream")) {
    while(true) Thread.sleep(10000);
}
```

Figure 1. Java code fragment to deadlock the HotJava browser by locking its status line.

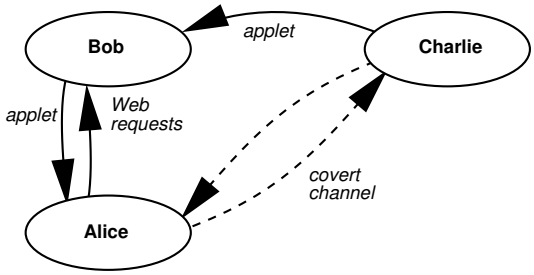

Figure 2. A Three Party Attack — Charlie produces a Trojan horse applet. Bob likes it and uses it in his Web page. Alice views Bob's Web page and Charlie's applet establishes a covert channel to Charlie. The applet leaks Alice's information to Charlie. No collusion with Bob is necessary.

to the host from which it was loaded. This is the only security mode available to Netscape users. In fact, HotJava and Netscape fail to enforce this policy through a number of errors in their implementation.

The `accept()` system call, used to receive a network connection initiated on another host, is not protected by the usual security checks in HotJava. This allows an arbitrary host on the Internet to connect to a HotJava browser as long as the location of the browser is known. For this to be a useful attack, the applet needs to signal the external agent to connect to a specified port. Even an extremely low-bandwidth covert channel would be sufficient to communicate this information. The `accept()` call is properly protected in Netscape, but the attack described in section 3.7 allows applets to call `accept()`.

If the Web server which served the applet is running an SMTP mail daemon, the applet can connect to it and transmit an e-mail message to any machine on the Internet. Additionally, the *Domain Name System* (DNS) can be used as a two-way communication channel to an arbitrary host on the Internet. An applet may reference a fictitious name in the attacker's domain. This transmits the name to the attacker's DNS server, which could interpret the name as a message, and then send a list of arbitrary 32-bit IP numbers as a reply. Repeated DNS calls by the applet establish a channel between the applet and the attacker's DNS server. This channel also passes through a number of firewalls[7]. In HotJava, the DNS channel was available even with the security mode set to "no network access," although this was fixed in JDK and Netscape. DNS has other security implications; see section 3.5 for details.

Another third-party channel is available with the *URL redirect* feature. Normally, an applet may instruct the browser to load any page on the Web. An attacker's server could record the URL as a message, then redirect the browser to the original destination.

When we notified Sun about these channels, they said the DNS channel would be fixed[26], but in fact it was still available in JDK and Netscape. Netscape has since issued a patch to fix this problem.

As far as we know, nobody has done an analyis of storage or timing channels in Java.

3.4. Information Available to Applets

If a rogue applet can establish a channel to any Internet host, the next issue is what the applet can learn about the user's environment to send over the channel.

In HotJava, most attempts by an applet to read or write the local file system result in a dialog box for the user to grant approval. Separate access control lists (ACLs)[4] specify where reading and writing of files or directories may occur without the user's explicit permission. By default, the write ACL is empty and the read ACL contains the HotJava library directory and specific MIME `mailcap` files. The read ACL also contains the user's `public_html` directory, which may contain information which compromises the privacy of the user. The Windows 95 version additionally allows writing (but not reading) in the `\TEMP` directory. This allows an applet to corrupt files in use by other Windows applications if the applet knows or can guess names the files may have. At a minimum, an applet can consume all the free space in the file system. These security concerns could be addressed by the user editing the ACLs; however, the system default should have been less permissive. Netscape does not permit any file system access by applets.

In HotJava, we could learn the user's login name, machine name, as well as the contents of all environment variables; `System.getenv()` in HotJava has no security checks.

[4] While Sun calls these "ACLs", they actually implement profiles — a list of files and directories granted specific access permissions.

By probing environment variables, including the `PATH` variable, we can often discover what software is installed on the user's machine. This information could be valuable either to corporate marketing departments, or to attackers desiring to break into a user's machine. In JDK and Netscape, `System.getenv()` was replaced with "system properties," many of which are not supposed to be accessible by applets. However, the attack described in section 3.7 allows an applet to read or write any system property.

Java allows applets to read the system clock, making it possible to benchmark the user's machine. As a Java-enabled Web browser may well run on pre-release hardware and/or software, an attacker could learn valuable information. Timing information is also needed for the exploitation of covert timing channels. "Fuzzy time"[18] should be investigated to see if it can be used to mitigate both of these problems.

3.5. Implementation Errors

Some bugs arise from fairly localized errors in the implementation of the browser or the Java subsystem.

DNS Weaknesses A significant problem appears in the JDK and Netscape implementation of the policy that an applet can only open a TCP/IP connection back to the server it was loaded from. While this policy is sound (although inconvenient at times), it was not uniformly enforced. This policy was enforced as follows:

1. Get all the IP-addresses of the hostname that the applet came from.
2. Get all the IP-addresses of the hostname that the applet is attempting to connect to.
3. If any address in the first set matches any address in the second set, allow the connection. Otherwise, do not allow the connection.

The problem occurs in the second step: the applet can ask to connect to any hostname on the Internet, so it can control which DNS server supplies the second list of IP-addresses; information from this untrusted DNS server is used to make an access control decision. There is nothing to prevent an attacker from creating a DNS server that lies. In particular, it may claim that any name for which it is responsible has any given set of addresses. Using the attacker's DNS server to provide a pair of addresses *(machine-to-connect-to, machine-applet-came-from)*, the applet can connect to any desired machine on the Internet. The applet can even encode the desired IP-address pair into the hostname that it looks up. This attack is particularly dangerous when the browser is running behind a firewall, because the malicious applet can attack any machine behind the firewall. At this point, a rogue applet can exploit a whole legion of known network security problems to break into other nearby machines.

```
hotjava.props.put("proxyHost",
    "proxy.attacker.com");
hotjava.props.put("proxyPort", "8080");
hotjava.props.put("proxySet", "true");
HttpClient.cachingProxyHost =
    "proxy.attacker.com";
HttpClient.cachingProxyPort = 8080;
HttpClient.useProxyForCaching = true;
```

Figure 3. Code to redirect all HotJava HTTP retrievals. FTP retrievals may be redirected with similar code.

This problem was postulated independently by Steve Gibbons[11] and by us. To demonstrate this flaw, we produced an applet that exploits an old `sendmail` hole to run arbitrary Unix commands as user `daemon`.

As of this writing, Sun and Netscape have both issued patches to fix this problem. However, the attack described in section 3.7 reopens this hole.

Buffer Overflows HotJava and the alpha release of JDK had many unchecked `sprintf()` calls that used stack-allocated buffers. Because `sprintf()` does not check for buffer overflows, an attacker could overwrite the execution stack, thereby transferring control to arbitrary code. Attackers have exploited the same bug in the Unix `syslog()` library routine (via `sendmail`) to take over machines from across the network[6]. In Netscape and the beta release of JDK, all of these calls were fixed in the Java runtime. However, the disassembler was overlooked all the way through the JDK 1.0 release. Users disassembling Java bytecode using **javap** are at risk of having their machines compromised if the bytecode has very long method names.

Disclosing Storage Layout Although the Java language does not allow direct access to memory through pointers, the Java library allows an applet to learn where in memory its objects are stored. All Java objects have a `hashCode()` method which, unless overridden by the programmer, casts the address of the object's internal storage to an integer and returns it. While this does not directly lead to a security breach, it exposes more internal state than necessary.

Public Proxy Variables Perhaps the strongest attack we found on HotJava is that we can change the browser's HTTP and FTP proxy servers. We can establish our own proxy

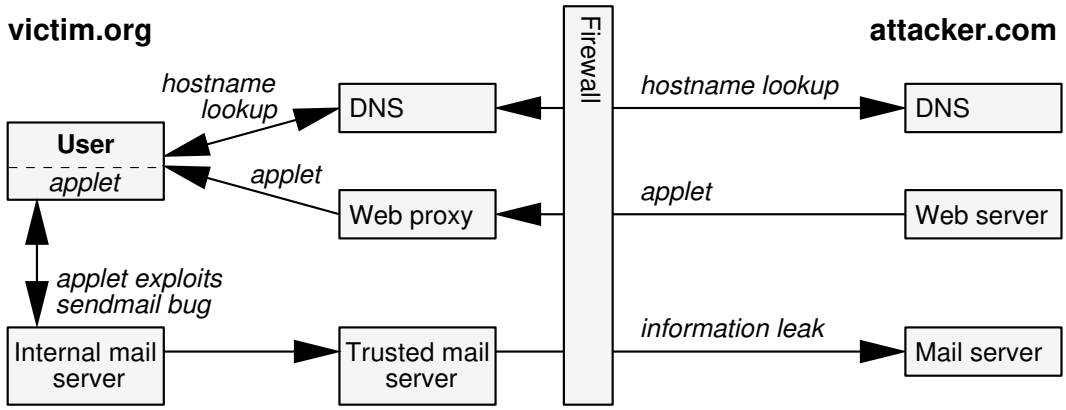

Figure 4. DNS subversion of Java: an applet travels from attacker.com to victim.org through normal channels. The applet then asks to connect to foo.attacker.com, which is resolved by attacker.com's DNS server to be victim.org's internal mail server which can then be attacked.

server as a man-in-the-middle. As long as the client is using unencrypted HTTP and FTP protocols, we can both watch and edit all traffic to and from the HotJava browser. All this is possible simply because the browser state was stored in public variables in public classes. While this attack compromises the user's privacy, its implementation is trivial (see figure 3). By using the property manager's `put()` method, we store our desired proxy in the property manager's database. If we can then entice the user to print a Web page, these settings will be saved to disk, and will be the default settings the next time the user starts HotJava. If the variables and classes were private, this attack would fail. Likewise, if the browser were running behind a firewall and relied on proxy servers to access the Web, this attack would also fail.

We note that the same variables are public in JDK, although they are not used. This code is not part of Netscape.

3.6. Inter-Applet Security

Since applets can persist after the Web browser leaves the page which contains them, it becomes important to separate applets from each other. Otherwise, an attacker's applet could deliberately sabotage a third party's applet. More formally, the Java runtime should maintain noninterference[12, 13] between applets. In many environments, it would be unacceptable for an applet to even learn of the existence of another applet.

In Netscape, `AppletContext.getApplets()` is careful to only return handles to applets on the same Web page as the caller. However, an applet may easily get a handle to the top-level `ThreadGroup` and then enumerate every thread running in the system, including threads belonging to other arbitrary applets. The Java runtime encodes the applet's class name in its thread name, so a rogue applet can now learn the names of all applets running in the system. In addition, an applet can call the `stop()` or `setPriority()` methods on threads in other applets. The `SecurityManager` only checks that applets cannot alter system threads; there are no restraints on applets altering other applet threads.

An insidious form of this attack involves a malicious applet that lies dormant except when a particular target applet is resident. When the target applet is running, the malicious applet randomly mixes degradation of service attacks with attacks on the target applet's threads. The result is that the user sees the target applet as slow and buggy.

3.7. Java Language and Bytecode Differences

Unfortunately, the Java language and the bytecode it compiles to are not as secure as they could be. There are significant differences between the semantics of the Java language and the semantics of the bytecode. We discuss superclass constructors and David Hopwood's attack[17] based on package names as examples of language versus bytecode differences. We then discuss related security weaknesses.

Superclass Constructors The Java language[9] requires that all constructors call either another constructor of the same class, or a superclass constructor as their first action. The system classes `ClassLoader`, `SecurityManager`, and `FileInputStream` all rely on this behavior for their security. These classes have constructors that check if they are called from an applet, and throw a `SecurityException` if so. Unfortunately, while the Java language prohibits the following code, the bytecode verifier readily accepts its bytecode equivalent:

```
class CL extends ClassLoader {
    CL() {
        try { super(); }
        catch (Exception e) {}
    }
}
```

This allows us to build (partially uninitialized) `ClassLoaders`, `SecurityManagers`, and `FileInputStreams`. ClassLoaders are the most interesting class to instantiate, as any code loaded by a `ClassLoader` asks its `ClassLoader` to load any classes it needs. This is contrary to the documentation[15] that claims the system name space is always searched first; we have verified this difference experimentally. Fortunately from the attacker's viewpoint, `ClassLoaders` don't have any instance variables, and the actual code in `ClassLoader`'s constructor only needs to run once — and it always runs before the first applet is loaded. The result of this attack, therefore, is a properly initialized `ClassLoader` which is under the control of an applet. Since `ClassLoaders` define the name space seen by other Java classes, the applet can construct a completely customized name space.

We have recently discovered that creating a `ClassLoader` gives an attacker the ability to defeat Java's type system. Assume that classes A and B both refer to a class named C. A `ClassLoader` could resolve A against class C, and B against class C'. If an object of class C is allocated in A, and then is passed as an argument to a method of B, the method in B will treat the object as having a different type, C'. If the fields of C' have different types or different access modifiers (`public, private, protected`) than those of C, then Java's type safety is defeated. This allows an attacker to get and set the value of *any* non-`static` variable, and call *any* method (including native methods). This attack also allows an applet to modify the class hierarchy, as it can set variables normally only writable by the runtime system. Java's bytecode verification and class resolution mechanisms are unable to detect these inconsistencies because Java defines only a weak correspondence between class names and `Class` objects.

We discovered this attack just before the submission deadline for this paper. We have implemented all of the type-system violations described above, but have not had time to investigate the full ramifications of this attack.

Illegal Package Names Java packages are normally named `java.io`, `java.net`, etc. The language prohibits '.' from being the first character in a package name. The runtime system replaces each '.' with a '/' to map the package hierarchy onto the file system hierarchy; the compiled code is stored with the periods replaced with slashes. David Hopwood found that if the first character of a package name was '/', the Java runtime system would attempt to load code from an absolute path[17], since absolute pathnames begin with a '/' character. Thus, if an attacker could place compiled Java in any file on the victim's system (either through a shared file system, via an incoming FTP directory, or via a distributed file system such as AFS), the attacker's code would be treated as trusted, since it came from the local file system rather than from the network. Trusted code is permitted to load dynamic link libraries (DLLs, written in C) which can then ignore the Java runtime and directly access the operating system with the full privileges of the user.

This attack is actually more dangerous than Hopwood first realized. Since Netscape caches the data it reads in the local file system, Netscape's cache can also be used as a way to get a file into the local file system. In this scenario, a normal Java applet would read (as data) files containing bytecode and DLL code from the server where the applet originated. The Java runtime would ask Netscape to retrieve the files; Netscape would deposit them in the local cache. As long as the applet can figure out the file names used by Netscape in its cache, it can execute arbitrary machine code without even needing prior access to the victim's file system.

3.8. Java Language and Bytecode Weaknesses

We believe the the Java language and bytecode definitions are weaker than they should be from a security viewpoint. The language has neither a formal semantics nor a formal description of its type system. The module system is weak, the scoping rules are too liberal, and methods may be called on partially initialized objects[16]. The bytecode is in linear form rather than a tree representation, it has no formal semantics, it has unnaturally typed constructors, and it does not enforce the `private` modifier on code loaded from the local file system. The separation of object creation and initialization poses problems. We believe the system could be stronger if it were designed differently.

Language Weaknesses The Java language has neither a formal semantics nor a formal description of its type system. We do not know what a Java program means, in any formal sense, so we cannot formally reason about Java and the security properties of the Java libraries written in Java. Java lacks a formal description of its type system, yet the security of Java relies on the soundness of its type system. Java's package system provides only basic modules, and these modules cannot be nested, although the name space superficially appears to be hierarchical. With properly nested modules, a programmer could limit the visibility of security-critical components. In the present Java system, only access to variables is controlled, not their visibility. Traditional capability systems[24] treat capabilities as hidden names with associated access control rights. Java object references are less

flexible than capabilities — they must give either all access rights or no access rights to an object. Java also allows methods to be called from constructors: these methods may see a partially initialized object instance.

Bytecode Weaknesses The Java bytecode is where the security properties must ultimately be verified, as this is what gets sent to users to run. Unfortunately, it is rather difficult to verify the bytecode. The bytecode is in a linear form, so type checking it requires global dataflow analysis similar to the back end of an optimizing compiler[34]; this analysis is complicated further by the existence of exceptions and exception handlers. Type checking normally occurs in the front end of a compiler, where it is a traversal of the abstract syntax tree[28]. In the traditional case, type checking is compositional: the type correctness of a construct depends upon the current typing context, the type correctness of subexpressions, and whether the current construct is typable by one of a finite set of rules. In Java bytecode, the verifier must show that all possible execution paths have the same virtual machine configuration — a much more complicated problem, and thus more prone to error. The present type verifier cannot be proven correct, because there is not a formal description of the type system. Object-oriented type systems are a current research topic; it seems unwise for the system's security to rely on such a mechanism without a strong theoretical foundation. It is not certain that an informally specified system as large and complicated as Java bytecode is consistent.

Object Initialization Creating and initializing a new object occurs in an interesting way: the object is created as an uninitialized instance of its class, duplicated on the stack, then its constructor is called. The constructor's type signature is *uninitialized instance of class* $\rightarrow$ *void*; it mutates the current typing context for the appropriate stack locations to initialized instance of their class. It is unusual for a *dynamic* function call to mutate the *static* typing context.

The initialization of Java objects seems unnecessarily baroque. First, a newly-allocated object sets all instance variables to either null, zero, or false. Then the appropriate constructor is called. Each constructor executes in three steps: First, it calls a another constructor of its own class, or a constructor of its superclass. Next, any explicit initializers for instance variables (e.g. `int x = 6;`) written by the programmer are executed. Finally, the body of the constructor is executed. During the execution of a constructor body, the object is only partially initialized, yet arbitrary methods of the object may be invoked, including methods that have been overridden by subclasses, even if the subclasses' constructors have not yet run. It seems unwise to have the system's security depend on programmers' understanding of such a complex feature.

Information Hiding We also note that the bytecode verifier does not enforce the semantics of the `private` modifier for code loaded from the local file system. Two classes loaded from the local file system in the same package, have access to all of each other's variables, whether or not they are declared private. In particular, *any* code in the `java.lang` package can set the system's security manager, although the definition of `System.security` and `System.setSecurityManager()` would seem to prevent this. The Java runtime allows the compiler to inline calls to `System.getSecurityManager()`, which may provide a small performance increase, but with a security penalty.

The Java language definition could be altered to reduce accidental leaks of information from public variables, and encourage better program structure with a richer module system than Java's `package` construct. Public variables in public classes are dangerous; it is hard to think of any safe application for them in their present form. While Java's packages define multiple, non-interfering name spaces, richer interfaces and parameterized modules would be useful additions to the language. By having multiple interfaces to a module, a module could declare a richer interface for trusted clients, and a more restrictive interface for untrusted clients. The introduction of parameterized modules, like Standard ML's functors[25], should also be investigated. Parameterized modules are a solution to the program structuring problem that opened up our man-in-the-middle attack (see section 3.5).

4. Security Analysis

We found a number of interesting problems in both HotJava, an alpha release, and Netscape 2.0, a released product. More instructive than the particular bugs we and others have found is an analysis of their possible causes. Policy enforcement failures, coupled with the lack of a formal security policy, make interesting information available to applets, and also provide channels to transmit it to an arbitrary third party. The integrity of the runtime system can also be compromised by applets. To compound these problems, no audit trail exists to reconstruct an attack afterward. In short, the Java runtime is not a high assurance system.

4.1. Policy

The present documents on Netscape[29] and HotJava do not formally define a security policy. This contradicts the first of the Orange Book's Fundamental Computer Security Requirements, namely that "There must be an explicit and well-defined security policy enforced by the system."[27] Without such a policy, it is unclear how a secure implementation is supposed to behave[22]. In fact, Java has two

entirely different uses: as a general purpose programming language, like C++, and as a system for developing untrusted applets on the Web. These roles will require vastly different security policies for Java. The first role does not demand any extra security, as we expect the operating system to treat applications written in Java just like any other application, and we trust that the operating system's security policy will be enforced. Web applets, however, cannot be trusted with the full authority granted to a given user, and so require that Java define and implement a protected subsystem with an appropriate security policy.

4.2. Enforcement

The Java `SecurityManager` is intended to be a reference monitor[20]. A reference monitor has three important properties:

1. It is always invoked.
2. It is tamperproof.
3. It is verifiable.

Unfortunately, the Java `SecurityManager` design has weaknesses in all three areas. It is not always invoked: programmers writing the security-relevant portions of the Java runtime system must remember to explicitly call the `SecurityManager`. A failure to call the `SecurityManager` will result in access being granted, contrary to the security engineering principle that dangerous operations should fail unless permission is explicitly granted. It is not tamperproof: the original beta implementation of the `SecurityManager` had a `protected` variable, which could be modified by any subclass of `SecurityManager`. This error was fixed by changing the semantics of `protected`. Using the superclass constructor attack to create a `ClassLoader`, an attacker can change *any* variable in the system, including the `SecurityManager`'s private variables. The attacker can also change the variable used by the `SecurityManager` to determine where a class was loaded from, thereby tricking the `SecurityManager` into believing a class is trusted. Finally, it is not verifiable: it is written in a language, Java, that does not have precisely defined semantics. Unfortunately, the JDK and Netscape implementations of the `SecurityManager` had faulty logic (see section 3.5).

4.3. Integrity

The architecture of HotJava is inherently more prone than that of Netscape to accidentally reveal internal state to an applet because the HotJava browser's state is kept in Java variables and classes. Variables and methods that are public or protected are potentially very dangerous: they give the attacker a toe-hold into HotJava's internal state. Static synchronized methods and public instances of objects with synchronized methods lead to easy denial of service attacks, because any applet can acquire these locks and never release them. These are all issues that can be addressed with good design practices, coding standards, and code reviews.

Java's architecture does not include an identified trusted computing base (TCB)[27]. Substantial and dispersed parts of the system must cooperate to maintain security. The bytecode verifier, and interpreter or native code generator must properly implement all the checks that are documented. The HotJava browser (a substantial program) must not export any security-critical, unchecked public interfaces. This does not approach the goal of a small, well defined, verifiable TCB. An analysis of which components require trust would have found the problems we have exploited, and perhaps solved some of them.

4.4. Accountability

The fourth fundamental requirement in the Orange Book is accountability: "Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party."[27] The Java system does not define any auditing capability. If we wish to trust a Java implementation that runs bytecode downloaded across a network (i.e. HotJava or Netscape), a reliable audit trail is a necessity. The level of auditing should be selectable by the user or system administrator. As a minimum, files read and written from the local file system should be logged, along with network usage. Some users may wish to log the bytecode of all the programs they download. This requirement exists because the user cannot count on the attacker's Web site to remain unaltered after a successful attack. The Java runtime system should provide a configurable audit system.

5. Flexible Security for Applets

A major problem in defining a security policy for Java applets is making the policy flexible enough to not unduly limit applets, while still preserving the user's integrity and privacy. We will discuss some representative applications below and their security requirements. We will also suggest some mechanisms that we feel will be useful for implementing a flexible and trustworthy policy.

5.1. Networking

The Java runtime library must support all the protocols in current use today, including HTTP (the Web), FTP (file transfer), Gopher, SMTP (email), NNTP (Usenet news), and Finger (user information). Untrusted applets should be able to use network services only under restricted circumstances.

FTP presents the most difficulties. While FTP normally has the server open a connection back to the client for each data transfer, requiring the client to call `listen()` and `accept()`, all FTP servers are required to support passive mode, where the client actively opens all the connections. However, a FTP client must be carefully designed to ensure an applet does not use it to perpetrate mischief upon third parties. In particular, an applet should not be able to control the `PORT` commands sent on its behalf.

5.2. Distributed Applications

Other applications that would be desirable to implement as applets include audio/video conferencing, real-time multi-player games, and vast distributed computations like factoring. Games require access to high-speed graphics libraries; many of these libraries trade speed for robustness and may crash the entire machine if they are called with bad arguments. The Java interfaces to the libraries may have to check function calls; with proper compiler support some of these checks could be optimized away. Games also require the ability to measure real time, which makes it more difficult to close the benchmarking hole. For teleconferencing, the applet needs access to the network and to the local video camera and microphone — exactly the same access one needs to listen in on a user's private conversations. An unforgeable indicator of device access and an explicit "push to talk" interface would provide sufficient protection for most users.

Providing a distributed computation as a Java applet would vastly increase the amount of cycles available: "Just click here, and you'll donate your idle time to computing ...." But this requires that a thread live after the user moves on to another Web page, which opens up opportunities for surreptitious information gathering and denial of service attacks. While many applets have legitimate reasons to continue running after the Web browser is viewing a new page, there should be a mechanism for users to be aware that they running, and to selectively kill them.

5.3. User Interface

The security user interface is critical for helping the average user choose and live with a security policy. In HotJava, an applet may attempt any file or network operation. If the operation is against the user's currently selected policy, the user is presented an *Okay / Cancel* dialog. Many users will disable security if burdened with repeated authorization requests from the same applet. Worse, some users may stop reading the dialogs and repeatedly click *Okay*, defeating the utility of the dialogs.

Instead, to minimize repetitive user interaction, applets should request capabilities when they are first loaded. The user's response would then be logged, alleviating the need for future re-authorization. To associate the user's preferences with a specific applet or vendor will likely require digital signatures to thwart spoofing attacks.

Another useful feature would be trusted dialog boxes. An untrusted applet could call a trusted *File Save* dialog with no default choice which returns an open handle to the file chosen by the user. This would allow the user to grant authorization for a specific file access without exposing the full file system to an untrusted applet[19]. A similar trusted dialog could be used for initiating network connections, as might be used in chat systems or games. An applet could read or write the clipboard by the user selecting *Cut from Applet* and *Paste to Applet* from the *Edit* menu, adjacent to the normal cut and paste operations. By presenting *natural* interfaces to the user, rather than a succession of security dialogs, a user can have a controlled and comfortable interaction with an applet. By keeping the user in control, we can allow applets limited access to system resources without making applets too dangerous or too annoying.

6. Conclusion

Java is an interesting new programming language designed to support the safe execution of applets on Web pages. We and others have demonstrated an array of attacks that allow the security of both HotJava and Netscape to be compromised. While many of the specific flaws have been patched, the overall structure of the systems leads us to believe that flaws will continue to be found. The absence of a well-defined, formal security policy prevents the verification of an implementation.

We conclude that the Java system in its current form cannot easily be made secure. Significant redesign of the language, the bytecode format, and the runtime system appear to be necessary steps toward building a higher-assurance system. Without a formal basis, statements about a system's security cannot be definitive.

The presence of flaws in Java does not imply that competing systems are more secure. We conjecture that if the same level of scrutiny had been applied to competing systems, the results would have been similar. Execution of remotely-loaded code is a relatively new phenomenon, and more work is required to make it safe.

7. Acknowledgments

We wish to thank Andrew Appel, Paul Karger and the referees for reading this paper and making many helpful suggestions. We are grateful to Paul Burchard, Jon Riecke and Andrew Wright for useful conversations about this work. We also thank Sun Microsystems for providing full source to

the HotJava browser and the Java Development Kit, making this work possible.

Edward W. Felten is supported in part by an NSF National Young Investigator award.

References

[1] S. R. Ames, Jr., M. Gasser, and R. G. Schell. Security kernel design and implementation: An introduction. *Computer*, pages 14–22, July 1983. Reprinted in *Tutorial: Computer and Network Security*, M. D. Abrams and H. J. Podell, editors, IEEE Computer Society Press, 1987, pp. 142–157.

[2] J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force, Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA 01730 USA, Oct. 1972. Volume 2, pages 58–69.

[3] N. S. Borenstein. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *IFIP International Working Conference on Upper Layer Protocols, Architectures and Applications*, 1994.

[4] D. F. C. Brewer and M. J. Nash. The Chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[5] G. Castagna. Covariance and contravariance: Conflict without a cause. Technical Report LIENS-94-18, Département de Mathématiques et d'Informatique, Ecole Normale Supérieure, Oct. 1994. `ftp://ftp.ens.fr/pub/reports/liens/liens-94-18.A4.ps.Z`.

[6] CERT Coordination Center. Syslog vulnerability - a workaround for sendmail. CERT Advisory CA-95:13, Oct. 1995. `ftp://ftp.cert.org/pub/cert_advisories/CA-95%3A13.syslog.vul`.

[7] W. R. Cheswick and S. M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[8] A. Courtney. Phantom: An interpreted language for distributed programming. In *Usenix Conference on Object-Oriented Technologies*, June 1995.

[9] D. Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Inc., 1st edition, Feb. 1996.

[10] General Magic, Inc., 420 North Mary Ave., Sunnyvale, CA 94086 USA. *The Telescript Language Reference*, Oct. 1995. `http://www.genmagic.com/Telescript/TDE/TDEDOCS_HTML/telescript.html`.

[11] S. Gibbons. Personal communication, Feb. 1996.

[12] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[13] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 75–86, 1984.

[14] J. Gosling. Personal communication, Oct. 1995.

[15] J. Gosling and H. McGilton. *The Java Language Environment*. Sun Microsystems Computer Company, 2550 Garcia Avenue, Mountain View, CA 94043 USA, May 1995. `http://java.sun.com/whitePaper/javawhitepaper_1.html`.

[16] L. Hasiuk. Personal communication, Feb. 1996.

[17] D. Hopwood. Java security bug (applets can load native methods). *RISKS Forum*, 17(83), Mar. 1996. `ftp://ftp.sri.com/risks/risks-17.83`.

[18] W.-M. Hu. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 8–20, 1991.

[19] P. A. Karger. Limiting the damage potential of discretionary trojan horses. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 32–37, 1987.

[20] B. W. Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, Mar. 1971. Reprinted in *Operating Systems Review*, 8(1):18–24, Jan. 1974.

[21] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, Feb. 1980.

[22] C. E. Landwehr. Formal models for computer security. *Computing Surveys*, 13(3):247–278, Sept. 1981.

[23] L. Lemay and C. Perkins. Yes, Java's Secure. Here's Why. *Datamation*, 42(5):47–49, March 1, 1996.

[24] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[25] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[26] M. Mueller. Regarding java security. *RISKS Forum*, 17(45), Nov. 1995. `ftp://ftp.sri.com/risks/risks-17.45`.

[27] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria*. National Computer Security Center, 1985.

[28] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[29] J. Roskind. Java and security. In *Netscape Internet Developer Conference*, Netscape Communications Corp., 501 E. Middlefield Road, Mountain View, CA 94043 USA, Mar. 1996. `http://home.netscape.com/misc/developer/conference/`.

[30] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 12–21, Dec. 1981.

[31] B. Stroustrup. *The C++ Programming Langauge*. Addison-Wesley, 2nd edition, 1991.

[32] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[33] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 2nd edition, 1983.

[34] F. Yellin. Low level security in Java. In *Fourth International World Wide Web Conference*, Boston, MA, Dec. 1995. World Wide Web Consortium. `http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html`.

Online For more information related to this paper, please visit our Web page:
`http://www.cs.princeton.edu/~ddean/java/`