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.

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

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.

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

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

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

```
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.

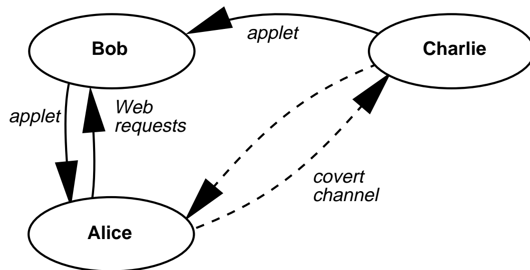

Figure 2. A Three Party Attack — Charlie produces a Trojan horse applet. Bob likes it and uses it in his Web page. Alice views Bob's Web page and Charlie's applet establishes a covert channel to Charlie. The applet leaks Alice's information to Charlie. No collusion with Bob is necessary.

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.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase® Smarter legal research.