Java Security: Web Browsers and Beyond

Drew Dean
ddean@cs.princeton.edu

Edward W. Felten
felten@cs.princeton.edu

Dan S. Wallach
dwallach@cs.princeton.edu

Dirk Balfanz
balfanz@cs.princeton.edu

Department of Computer Science
Princeton University
Princeton, NJ 08544

February 24, 1997

Abstract

The introduction of Java applets has taken the World Wide Web by storm. Java allows web creators to embellish their content with arbitrary programs which execute in the web browser, whether for simple animations or complex front-ends to other services. We examined the Java language and the Sun HotJava, Netscape Navigator, and Microsoft Internet Explorer browsers which support it, and found 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 Microsystems 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[21], Safe-Tcl[6], Phantom[10], Juice[14] and Telescript[16] have been proposed for running untrusted 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 their Netscape Navigator web browser, beginning with version 2.0. Microsoft has also licensed Java from Sun, and incorporated it into Microsoft Internet Explorer 3.0. With the support of many influential companies, 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.

The original version of this paper was written in November, 1995 — after Netscape announced it would use Java. Since that time, we have found a number of bugs in Navigator through its various beta releases and later in Microsoft's Internet Explorer. As a direct result of our investigation, and the tireless efforts of the vendors' Java programmers, we believe the security of Java has significantly improved since its early days. In particular, Internet Explorer 3.0, which shipped in August, 1996, had the benefit of nine months of our investigation into Netscape's Java. Still, despite all the work done by us and by others, no one can claim that Java's security problems are fixed.

Netscape Navigator and HotJava[1] are examples of two distinct architectures for building web browsers. Netscape Navigator is written in an unsafe language, C, 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 Navigator can suffer from being implemented in an unsafe language (buffer overflow, memory leakage, etc.), but provides a well-defined interface to the Java subsystem. In Netscape Navigator, 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 private browser state to applets.

In order to be secure, such systems must limit applets' access to system resources such as the file system, the CPU, the network, the graphics display, and the browser's internal state. The language should be *memory safe* – preventing forged pointers and checking array bounds. Additionally, the system should garbage-collect memory to prevent both malicious and accidental memory leakage. Finally, the system must manage system calls and other methods which allow applets to affect each other as well as the environment beyond the browser.

Many systems in the past have attempted to use language-based protection. 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. Jones and Liskov describe language support for secure dataflow[26]. Rees describes a modern capability system built on top of Scheme[40].

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 Sun's HotJava, Netscape's Navigator, and Microsoft's Internet Explorer web browsers, 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 presents our conclusions. A more complete discussion of some of these issues can be found in McGraw and Felten's book[34].

2 Java Semantics

Java is similar in many ways to C++[42]. 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[30].

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 name spaces to prevent both accidental and malicious name clashes. Bytecode loaded from the local file system is visible to all applets. The documentation[22] 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.

[1]Unless otherwise noted, "HotJava-Alpha" refers to the 1.0 alpha 3 release of the HotJava web browser from Sun Microsystems, "Netscape Navigator" refers to Netscape Navigator 2.0, "Internet Explorer" refers to Microsoft Internet Explorer 3.0, and "JDK" refers to the Java Development Kit, version 1.0, from Sun.

[2]While the documentation[22] does not define "source", it appears to mean the URL prefix of origin. Sun and Netscape have announced plans to include support for digital signatures in future versions of their products. Microsoft has some support for digital signatures. See section 5.4.

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[3] `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 internally tagged with the `ClassLoader` that loaded it, and that `ClassLoader` is used to resolve any future unresolved symbols for the class. Additionally, 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 in JDK 1.0.x and derived systems (including Netscape Navigator and Internet Explorer).

Java programmers can combine related classes into a `package`. These packages are similar to name spaces in C++[43], modules in Modula-2[44], or structures in Standard ML[35]. 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. 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, origin of code) pair. A (package, origin of code) pair defines the scope of a Java class, method, or instance variable that is not given a `public`, `private`, or `protected` modifier[4]. 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 {
  protected int x;
  void SetFoo(Foo obj) { obj.x = 1; } // Legal
  void SetBar(Bar obj) { obj.x = 1; } // Legal
}

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

The definition of `protected` was the same as C++ in some early versions of Java; it was changed during the beta-test period to patch a security problem[37] (see also section 4.2).

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 is designed to ensure that 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[7] rule for subtyping arrays, so array stores require run time type checks[5] in addition to the normal array bounds checks. Cast expressions also require runtime checks. Unfortunately, this means the bytecode verifier is not the only piece of the runtime system that must be correct to ensure type safety. Dynamic checks also introduce a performance penalty.

[3]An `abstract` class is a class with one or more methods declared but not implemented. Abstract classes cannot be instantiated, but define method signatures for subclasses to implement.

[4]Colloquially, methods or variables with no access modifiers are said to have *package scope*.

[5]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.

2.1 Java Security Mechanisms

In HotJava-Alpha, 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[29]. The `SecurityManager` defines and implements a security policy, centralizing all access control decisions. All potentially dangerous methods first consult the security manager before executing. Netscape and Microsoft also use 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 current browsers typically make their access control decisions by examining the contents of the call stack, looking for the presence of a `ClassLoader`, indicating that they were called, directly or indirectly, from an applet.

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 more easily embeddable in other software, and potentially performs better because protection boundaries can be crossed without a context switch[3].

3 Taxonomy of Java Bugs

We now present a taxonomy of known 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.

3.1 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-Alpha browser, effectively preventing it from loading any more pages. In Netscape Navigator, this attack can lock the `java.net.InetAddress` class, blocking all hostname lookups and hence most new network connections. HotJava, Navigator, and Internet Explorer all have classes suitable for this attack. The attack could be prevented by replacing such critical classes with wrappers that do not expose the locks to untrusted code. However, the CPU and memory attacks cannot be easily fixed; many genuine applications may need large amounts of memory and CPU. Another attack, first implemented by Mark LaDue, is to open a large number of windows on the screen. This will sometimes crash the machine. LaDue has a web page with many other denial of service attacks[28].

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.

```
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.

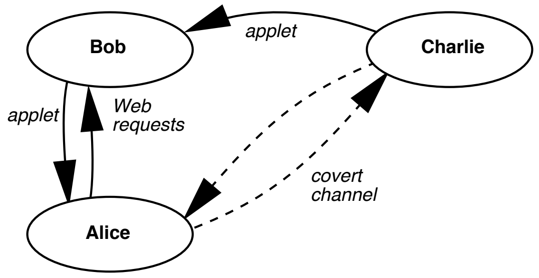

Figure 2: A Three Party Attack — Charlie produces a Trojan horse applet. Bob likes it and uses it in his web page. Alice views Bob's web page and Charlie's applet establishes a covert channel to Charlie. The applet leaks Alice's information to Charlie. No collusion with Bob is necessary.

Sun has said that they consider denial of service attacks to be low-priority problems[20].

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 HotJava, Navigator, and Internet Explorer, 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 to the host from which it was loaded. This is the only security mode available to Navigator and Internet Explorer users[6]. In fact, the browsers have failed 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-Alpha. 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 current Java implementations.

If the web server which provided 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[9]. In HotJava-Alpha, the DNS channel was available even with the security mode set to "no network access," although this was fixed in later Java versions. DNS has other security implications; see section 3.5.1 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[36], but in fact it was still available in JDK and Netscape Navigator. Netscape has since issued a patch (incorporated into

[6]Without using digitally signed code.

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.