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.

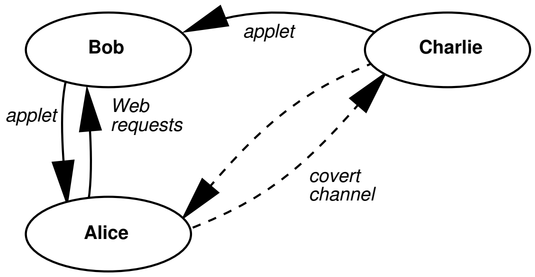

Figure 2: A Three Party Attack — Charlie produces a Trojan horse applet. Bob likes it and uses it in his web page. Alice views Bob's web page and Charlie's applet establishes a covert channel to Charlie. The applet leaks Alice's information to Charlie. No collusion with Bob is necessary.

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.

Netscape Navigator 2.01) to fix this problem.

As far as we know, nobody has done an analysis of covert storage or timing channels in the Java runtime system.

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-Alpha, 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)[7] 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. Navigator and Internet Explorer do not permit any file system access by applets (without digital signatures).

In HotJava-Alpha, we could learn the user's login name, machine name, as well as the contents of all environment variables; `System.getenv()` in HotJava-Alpha had no security checks. By probing environment variables, including the `PATH` variable, we could 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 later Java versions, `System.getenv()` was replaced with "system properties," many of which are not supposed to be accessible by applets. However, there have been implementation problems (see Section 3.7.2) that allowed 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"[25] should be investigated to see if it can mitigate these problems.

3.5 Implementation Errors

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

3.5.1 DNS Weaknesses

A significant problem appeared in the JDK and Netscape Navigator 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 reasonable, since applets often need to load components (images, sounds, etc.) from their host, 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 occurred 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 was used to make an access control decision. There is nothing to prevent an attacker from

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

```
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-Alpha HTTP retrievals. FTP retrievals may be redirected with similar code.

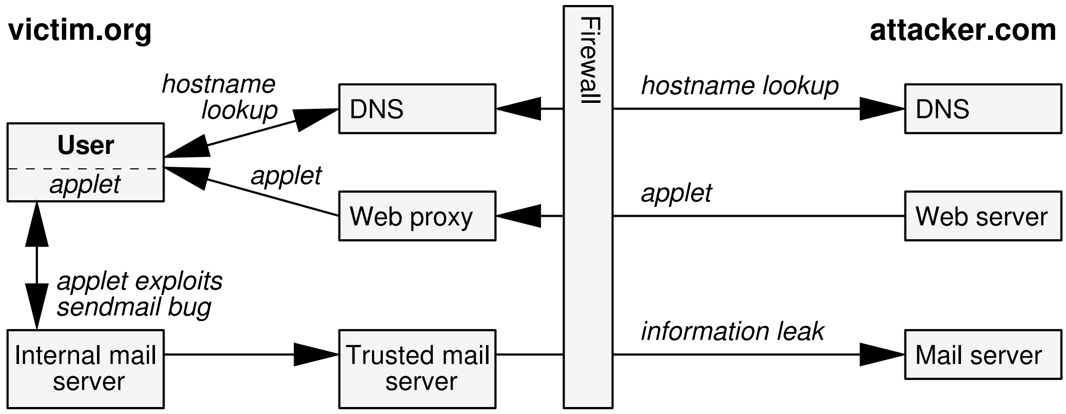

Figure 4: DNS subversion of Java: an applet travels from `attacker.com` to `victim.org` through normal channels. The applet then asks to connect to `foo.attacker.com`, which is resolved by the DNS server for `attacker.com` to be mail server inside `victim.org` which can then be attacked.

creating a DNS server that lies[4]. 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 could connect to any desired machine on the Internet. The applet could 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.

This problem was postulated independently by Steve Gibbons[17] 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`.

Sun (JDK 1.0.1) and Netscape (Navigator 2.01)[8] have both issued patches to fix this problem.

3.5.2 Buffer Overflows

HotJava-Alpha 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[8]. In later Java releases, all of these calls were fixed in the Java runtime. However, the bytecode disassembler was overlooked all the way through the JDK 1.0 release. Users disassembling Java bytecode using **javap** were at risk of having their machines compromised if the bytecode had very long method names. This bug was fixed in JDK 1.0.2.

[8]Netscape solved the problem by storing the results of all DNS name lookups internally, forcing a given hostname to map to exactly one IP address. Netscape Navigator also stores the applet source as a function of its IP address, not hostname. This solution has the added property that it prevents time-varying DNS attacks. Previously, an attacker's name server could have returned different IP addresses for the same hostname each time it was queried, allowing the same attacks detailed above.

3.5.3 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.

3.5.4 Public Proxy Variables

An interesting attack on HotJava-Alpha is that an attacker can change the browser's HTTP and FTP proxy servers. An attacker can establish their own proxy 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-Alpha 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, an attackers stores a desired proxy in the property manager's database. If the attacker 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 Navigator or Internet Explorer.

3.6 Inter-Applet Security

Since applets can persist after the web browser leaves the page which contains them, it becomes important to protect applets from each other. Otherwise, an attacker's applet could deliberately sabotage a third-party's applet. More formally, the Java runtime should maintain non-interference[18, 19] between unrelated applets. In many environments, it would be unacceptable for an applet to even learn of the existence of another applet.

In Netscape Navigator, `AppletContext.getApplets()` is careful to only return handles to applets on the same web page as the caller. However, an applet could 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 could call the `stop()` or `setPriority()` methods on threads in other applets. The `SecurityManager` only checked that applets could not alter the state of system threads; there were no restraints on applets altering other applet threads. Netscape Navigator 4.0 prevents an attacker from seeing threads belonging to applets on other web pages, in the same way it protects applets. Internet Explorer allows an applet to see those threads, but calls to `stop()` or `setPriority()` have no effect.

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

3.7 Java Language Implementation Failures

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. First, we discuss David Hopwood's attack[24] based on package names. Next, we present our attack that runs arbitrary machine code after compromising the type system. Several flaws in the type system are examined, including two first noted by Tom Cargill.

3.7.1 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[24], since absolute pathnames begin with a "/" character on Unix or Windows. 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 Navigator caches the data it reads in the local file system, Netscape Navigator'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 Navigator to retrieve the files; Navigator would deposit them in the local cache. As long as the applet can figure out the file names used by Navigator in its cache, it can execute arbitrary machine code without even needing prior access to the victim's file system.

3.7.2 Superclass Constructors

The Java language[21] 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 accepted its bytecode equivalent:

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

This allowed an attacker 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 resolve any classes it references. This is contrary to the documentation[22] that claims the system name space is always searched first; we have verified this difference experimentally. Fortunately for an attacker, `ClassLoaders` did not have any instance variables, and the `ClassLoader` constructor only needs to run once, to initialize a variable in the runtime system. This happens before any applets are loaded. Therefore, this attack resulted in 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. A fix for this problem appeared in Netscape Navigator 2.02, which was later broken (see Section 3.7.5). Netscape Navigator 3.0 and JDK 1.0.2 took different approaches to fix this problem.

We 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 (e.g., `Object` and `int`) 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 read and write variables normally only visible to the runtime system.

Any attack which allows object references to be used as integers, and *vice versa*, leads to complete penetration of Java (see section 3.7.4). 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.

Netscape Navigator 3.0 and Microsoft Internet Explorer fix the superclass constructor issue and take other measures to prevent applets from instantiating `ClassLoaders`. JDK 1.1-Beta initially offered "safe" `ClassLoaders` to applets, but the feature was withdrawn from the final release because they could, in fact, still be abused.

Fundamentally, the job of a `ClassLoader` is to resolve names to classes as part of Java's dynamic linking. Dynamic linking has subtle interactions with static typechecking. For a formal analysis of this process and some necessary conditions for correctness, see Dean[11] for details.

3.7.3 Attacking the SecurityManager

Unfortunately, a `ClassLoader` can load a new `SecurityManager` that redeclares the `SecurityManager`'s variables as `public`, violating the requirement that reference monitors be tamperproof. There are four interesting variables in the JDK `AppletSecurity` class: `readACL, writeACL, initACL,` and `networkMode`. The `readACL` and `writeACL` variables are lists of directories and files that applets are allowed to read and write. The `initACL` variable tracks whether the ACLs have been initialized. The `networkMode` variable determines what hosts applets are allowed to make network connections to. By setting the `networkMode` variable to allow connections anywhere, the ACLs to `null`, and the `initACL` variable to true, we effectively circumvent all JDK security.

Java's `SecurityManagers` generally base their access control decisions on whether they are being called in the dynamic scope of code loaded from the network. The default `ClassLoader` in the runtime system only knows how to load classes from the local file system, and appears as the special value `NULL` to the runtime system. Any other `ClassLoader` is assumed to indicate untrusted code. However, a `ClassLoader` can provide an implementation of `Class` that makes certain runtime system data structures accessible as `ints`. Setting the `ClassLoader` field to zero causes the Java runtime system to believe that the code came from the local file system, also effectively bypassing the `SecurityManager`.

3.7.4 Running Machine Code from Java

Netscape Navigator 2.0 protected itself from the attacks described above with additional checks in the native methods which implement file system routines that applets would never have any reason to invoke. However, the type system violations (i.e., using `Objects` as `ints` and *vice versa*) make it possible, but non-trivial, to run arbitrary machine code, at which point an attacker can invoke any system call available to the user running the browser without restriction, and thus has completely penetrated all security provided by Java.

While Java does not guarantee a memory layout for objects[33], the current implementations lay out objects in the obvious way: instance variables are in consecutive memory addresses, and packed as in C. An attacker can clearly write machine code into integer fields, but there are two remaining challenges: learning the memory address of our code, and arranging for the system to invoke our machine code. All an attacker can do is use object references as integers, but note that object references in JDK and Netscape Navigator are pointers to pointers to objects, and we can only doubly dereference them. Internet Explorer uses a single pointer. Below, we describe how Netscape Navigator can be attacked, but this attack has been modified to work with Microsoft Internet Explorer as well.

Netscape Navigator's object references are pointers to a structure which contains two pointers: one to the object's private data, and another to its type information. Thus, while a malicious `ClassLoader` allows an attacker to cast object references to integers and back, the attacker can only doubly dereference the pointers. This complicated the process of learning the actual machine address of an object's data.

To solve this problem, observe that `Class` objects, i.e., instances of the class `java.lang.Class` are not implemented as normal Java objects. `Class` has no Java-visible variables, but its internal representation stores direct pointers to data, rather than the usual indirect Java references. This allows an attacker to directly edit the method table to point to our own machine code. The only remaining problem is to learn

```
lui    a0, ((a+20) div 65536)      ; Most significant 16 bits of string pointer
addi   a0, a0, ((a+20) mod 65536)  ; Least significant 16 bits of string pointer
li     v0, 1010                    ; System call number for unlink() in IRIX 5
syscall
nop
.asciz "/tmp/JavaSafe.NOT"         ; File to delete
```

Figure 5: MIPS assembly code to delete a file.

the code's memory address. To do this, an attacker can follow a chain of pointers through the method table and back to the `Class` structure. Because the chain has odd length and two steps are taken every time, the attacker can cycle around the loop, gaining access to each and every pointer.

The final attack overwrites the beginning of the `Class` structure with the machine code in figure 3.7.4 and changes an entry in the method table to point to it. Since the attack alters the runtime system's internal data structures, care must be taken not to crash the runtime system. Invoking that method from Java causes the native machine code to run, allowing us to then make arbitrary system calls. This attack can be implemented on *any* machine using the a variant of the Sun or Microsoft Java runtime systems. We implemented the attack on a Silicon Graphics workstation running version 5 of the IRIX operating system running Netscape Navigator.

3.7.5 More Type System Attacks

The `ClassLoader` attack described above was the first of many holes found in the Java type system. Any hole in the type system that allows integers to be used as object references is sufficient to run arbitrary machine code. Many such problems have been found.

Cargill's Interface Attack Tom Cargill noted that Java's `interface` feature could be used to call `private` methods. This works because all methods declared in an `interface` are `public`, and a class was allowed to implement an interface by inheriting a `private` method from its parent. Netscape Navigator 2.02 fixed the `ClassLoader` attack by making the native methods `private`, and wrapping them inside methods that checked a boolean variable initialized by the `ClassLoader` constructor before calling the `private`, `native` methods to do the real work. Since an attacker could now call the `private` methods directly, the fix was defeated.

Hopwood's Interface Attack David Hopwood found that `interfaces` were matched by name, and not `Class` object reference. By passing items across multiple name spaces (where a name space corresponds to a unique code base) via `System.out`, a `public`, non-`final` variable, Hopwood could also treat object references as integers, and vice versa. We found that exceptions were also identified by name, and thus had the same problem.

Arrays Java defines all arrays to be objects. The system normally provides array classes that inherit directly from `Object`, but use the covariant subtyping rule. However, we found that the user was able to define his own array classes because of a bug in the `AppletClassLoader`. When a class is loaded via `Class.forName()`, it will ask the `ClassLoader` of the code that invoked `Class.forName()` to supply the class, unless it's an array class, in which case the system will supply an appropriate definition. However, `AppletClassLoader` did not check that the name of the class it actually loaded is the same as what was requested until after it has called `defineClass()`, which had the side-effect of entering the class into the system class table. By misnaming an array class, it was entered into the system class table, and could be used as an array. By calling `Class.newInstance()`, an attacker could allocate an instance of their class, and cast it into an array. When the class definition is needed to check the cast, the system first looks in the system class table, but only for array classes. If our definition has an integer as its first instance variable, and the array is `Class[]`, then necessary conditions exist to run arbitrary machine code.

Tom Cargill noted that the *quick* variants of the method invocation instructions (which do not perform typechecking) do not interact properly with arrays of `interfaces`. Recall that the method invocation instruction is rewritten at runtime, using the actual type of its argument the first time it is executed. However, not all elements of the array need have the same type. Using this, an attacker can call private methods of `public`, non-`final` classes. This was fixed in JDK 1.1, Navigator 3.0, and Internet Explorer 3.0.

Packages The pre-release versions of Internet Explorer 3.0 did not separate packages with the same name loaded from different origins. An applet could declare classes belonging to system packages such as `java.lang`. These classes would be accepted due to a error in the `SecurityManager`'s method that was supposed to prevent this. This error was also present in JDK 1.0.2. As a result, an applet could access package-scope variables [9] and methods of system packages. This bug was fixed in the final release of Internet Explorer 3.0. Since Sun and Netscape's implementation of the Java Virtual Machine (JVM) detected this elsewhere, and did not in fact consult the `SecurityManager`, HotJava and Navigator were not vulnerable to this attack.

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.[10] The module system is weak, the scoping rules are too liberal, and methods may be called on partially initialized objects[23]. The bytecode is in linear form rather than a tree representation, has no formal semantics[11], has unnaturally typed constructors, and 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 had been designed differently.

Language Weaknesses The Java language definition[21] has neither a formal semantics nor a formal description of its type system. We do not know what a Java program means, in a 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 fundamentally 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. Java also allows methods to be called from constructors: these methods may see a partially initialized object instance[23].

One nice feature of Java is that an object reference is roughly equivalent to a traditional capability [32]. Because pointers cannot be forged, the possession of an object instance (such as an open file) represents the capability to use that file. However, the Java runtime libraries are not generally structured around using objects as capabilities. Used as capabilities, Java objects would have all the traditional problems of capability systems, e.g., difficulty tracking and controlling who has access to various system resources.

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[45]; 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[39]. (The Juice system[14] works in the same way.) In the traditional case, type checking is compositional: the type correctness of a construct depends upon the current typing context, the type correctness of its 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 lead to the same virtual

[9] such as `readACL` and `writeACL`, see section 3.7.3.

[10] Drossopoulou and Eisenbach are developing a formal semantics for Java[12].

[11] A formal definition of the bytecode is under development by Computational Logic, Inc. Similar work is also being done at Digital's Systems Research Center.

machine configuration — a much more complicated problem, and thus more prone to error. The present bytecode verifier cannot be proven correct, because there is not a formal description of the bytecode. 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.

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. The first time a class is used, its static constructors are executed. Then, for each instance of the class, a newly-allocated object sets all of its instance variables to either null, zero, or false, as appropriate for the type of the variable. Then the appropriate constructor is called. Each constructor executes in three steps: First, it calls 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. Since Java's security partly relies on some classes throwing exceptions during initialization (to prevent untrusted code from creating an instance of a dangerous class), 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 bytecode 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 improvement, 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 naming environments, 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[35], 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).

4 Security Analysis

We found a number of interesting problems in an alpha version of HotJava, and various commercial versions of Netscape Navigator and Microsoft Internet Explorer. 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 system is not a high assurance system.

4.1 Policy

The present documents on Netscape Navigator[41], Microsoft Internet Explorer, 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."[38] Without such a policy, it is unclear how a secure implementation is supposed to behave[31]. 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[29]. Recall that 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: attacks which compromise the type system can alter information that the `SecurityManager` depends on. Finally, the `SecurityManager` code is the only formal specification of policies. Without a higher-level formal specification, informal policies may have incorrect implementations which go unnoticed. For example, the informal policies about network access were incorrectly coded in JDK 1.0 and Netscape Navigator 2.0's `SecurityManager` (see Section 3.5.1).

4.3 Integrity

The architecture of HotJava is inherently more prone than that of Netscape Navigator or Microsoft Internet Explorer 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` 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)[38]. 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."[38]

The Java system does not define any auditing capability. If we wish to trust a Java implementation that runs bytecode downloaded across a network, 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.[12]

In support of distributed computation, desired features include remote procedure calls (RPCs) with remote object references, distributed garbage collection, automatic object marshalling and unmarshalling, and process migration (i.e. agents). While these features may greatly help programmers developing applications, they also create a whole new arena in which to attack the Java runtime. For example, the JVM must protect itself against unmarshalling an object with an inconsistent state. Likewise, an attacker may deliberately reference an object which is known to have been garbage collected. To maintain correctness, the JVM cannot make assumptions about the structure of RPC messages from untrusted sources.

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. 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 these 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 covert 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, denial of service, and cycle-stealing attacks. While many applets have legitimate reasons to

[12] In particular, an applet should not be able to control the `PORT` commands sent on its behalf. Some dynamic packet-filtering firewalls monitor FTP command connections, and open a hole in the firewall when the for each FTP data connection. Normally, this is safe, because a trusted user agent on the inside of the firewall is sending the `PORT` commands. However, with untrusted applets, this is no longer true. Steven Bellovin independently rediscovered this problem, and noted that it is an instance of the general problem of security policies not composing[5].

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[13].

5.3 User Interface

The security user interface is critical for helping the average user choose and live with a security policy. In HotJava-Alpha, 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[27]. 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.

5.4 Signed Applets

In Internet Explorer 3.0, Microsoft introduced Authenticode – a method for applying a digital signature to native machine code or Java bytecode. This signature acts as an *endorsement* of the code. While a complete description of Authenticode is beyond the scope of this paper (see Felten[13] or Garfinkel and Spafford[15] for more information), several issues are worth discussing here.

Authenticode allows only a binary trust model. Either the user grants the applet or signed ActiveX control full access to their machine (making it trivial to install a Trojan horse or mount any other imaginable attack), or the code is not allowed to run at all. Additionally, Authenticode has no tamper-proof logging facility (see section 4.4). Thus, once trust has been granted to a rogue applet, the user or organization would not have sufficient evidence to demonstrate who attacked them.

Despite these problems, Authenticode makes it easy for well-known software publishers to create web browser extensions which can be downloaded from any web page. The user need only trust the software-publisher, instead of every web page author who uses the extension. A mechanism like this for Java would help address complaints that Java's security is *too strong*, allowing only for "toy" programs.

As we discussed earlier, the user interface component is critical to the success of digitally signed code. We believe applets should request subsets of the full system privileges. Privileges need to be grouped into meaningful subsets for users to manage. For example, "typical game privileges" might grant limited file system access (for saving state) and full-screen graphics without exposing general network or file system access. Use of terms like "typical game privileges" is less likely to be misunderstood by users than lists of primitive system resources.

Additionally, it would be useful to digitally sign Java libraries, which could be used by unsigned Java applets. This could allow features such as trusted dialog boxes (see above) to be written by a trusted authority and distributed with any applet.

Digital signatures should have a number of interesting uses in future Java systems. With careful user-interface design and appropriate low-level support, trusted applets should be able to exercise privileges

[13]This feature appears in the HotJava-Beta release, available from JavaSoft. See `http://java.sun.com/products/HotJava/index.html`

beyond the least-common-denominator “sandbox” approach while controlling the user’s exposure to dangerous code.

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 Sun’s HotJava, Netscape’s Navigator, and Microsoft’s Internet Explorer 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[1]. 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 *1996 IEEE Symposium on Security and Privacy* referees for reading earlier versions of this paper and making many helpful suggestions. We are grateful to Paul Burchard, Jon Riecke, Andrew Wright, and Jim Roskind for useful conversations about this work. We also thank Sun Microsystems for providing full source code 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. Dan Wallach and Drew Dean are supported by fellowships from Bellcore. Princeton’s Secure Internet Programming group is supported by Bellcore, Microsoft, and Sun Microsystems.

Portions of this work were done while Drew Dean was a visitor at the SRI International Computer Science Laboratory and Dan Wallach was an intern at Netscape Communications Corp. We thank them for their support.

References

[1] Stanley R. Ames, Jr., Morrie Gasser, and Roger 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] James 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, October 1972. Volume 2, pages 58–69.

[3] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, 1991.

[4] Steven M. Bellovin. Using the domain name system for system break-ins. In *Proceedings of the Fifth Usenix UNIX Security Symposium*, pages 199–208, Salt Lake City, Utah, June 1995. Usenix.

[5] Steven M. Bellovin, July 1996. Personal communication.

[6] Nathaniel 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.

[7] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

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

[9] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

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

[11] Drew Dean. The security of static typing with dynamic linking. In *Fourth ACM Conference on Computer Communications Security*, Zurich, Switzerland, April 1997. `http://www.cs.princeton.edu/sip/pub/ccs4.html`.

[12] Sophia Drossopoulou and Susan Eisenbach. Is the Java type system sound? In *Proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages*, Paris, January 1997.

[13] Edward W. Felten. Inside risks: Webware security. *Communications of the ACM*, 40(4), April 1997.

[14] Michael Franz and Thomas Kistler. A tree-based alternative to Java byte-codes. In *Proceedings of the International Workshop on Security and Efficiency Aspects of Java '97*, 1997. Also appears as Technical Report 96-58, Department of Information and Computer Science, University of California, Irvine, December 1996.

[15] Simson Garfinkel and Gene Spafford. *Web Security and Commerce*. O'Reilly & Associates, Inc., 1997.

[16] General Magic, Inc., 420 North Mary Ave., Sunnyvale, CA 94086 USA. *The Telescript Language Reference*, June 1996. `http://www.genmagic.com/Telescript/Documentation/TRM/index.html`.

[17] Steve Gibbons. Personal communication, February 1996.

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

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

[20] James Gosling. Personal communication, October 1995.

[21] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[22] James Gosling and Henry McGilton. *The Java Language Environment*. Sun Microsystems Computer Company, 2550 Garcia Avenue, Mountain View, CA 94043 USA, May 1996. `http://java.sun.com/doc/language_environment.html`.

[23] Lee Hasiuk. Personal communication, February 1996.

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

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

[26] Anita K. Jones and Barbara H. Liskov. A language extension for controlling access to shared data. *IEEE Transactions on Software Engineering*, SE-2(4):277–285, December 1976.

[27] Paul 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.

[28] Mark LaDue. Hostile applets home page. `http://www.prism.gatech.edu/~gt8830a/HostileApplets.html`.

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

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

[31] Carl E. Landwehr. Formal models for computer security. *Computing Surveys*, 13(3):247–278, September 1981.

[32] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[33] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[34] Gary E. McGraw and Edward W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley and Sons, 1996.

[35] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[36] Marianne Mueller. Regarding Java security. *RISKS Forum*, 17(45), November 1995. `ftp://ftp.sri.com/risks/risks-17.45`.

[37] Marianne Mueller. Personal communication, January 1996.

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

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

[40] Jonathan A. Rees. A security kernel based on the lambda-calculus. Technical Report A.I. Memo No. 1564, Massachusetts Institute of Technology, Artificial Intelligence Labortory, March 1996.

[41] Jim Roskind. Java and security. In *Netscape Internet Developer Conference*, Netscape Communications Corp., 501 E. Middlefield Road, Mountain View, CA 94043 USA, March 1996. `http://developer.netscape.com/misc/developer/conference/proceedings/j4/index.html`.

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

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

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

[45] Frank Yellin. Low level security in Java. In *Fourth International World Wide Web Conference*, Boston, MA, December 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/sip/`

Biographies

Drew Dean received his bachelor's degree in Mathematics and Computer Science from Carnegie Mellon University, and currently is a graduate student in the Department of Computer Science at Princeton University. His research interests are in the foundations of language-based protection.

Edward W. Felten is Assistant Professor of Computer Science at Princeton University. He received his Ph.D. in Computer Science from the University of Washington in 1993. His research interests include computer security and distributed computing.

Dan Wallach currently studies security issues in remote code systems as a graduate student in the Department of Computer Science at Princeton University. He earned his B.S. in Electrical Engineering / Computer Science at the University of California, Berkeley, focusing on computer graphics and digital video.

Dirk Balfanz received his diploma from Humboldt University, Berlin, where he studied Computer Science, Artificial Intelligence, and Psychology. He is currently a graduate student in Computer Science at Princeton University. His research interests are in the challenges of large computer networks.