Security Flaws in the HotJava Web Browser

Drew Dean
ddean@cs.princeton.edu
Dan S. Wallach
dwallach@cs.princeton.edu
Department of Computer Science,
Princeton University

November 3, 1995

Abstract

The growth of the Internet and the World Wide Web has led to demand for Web extensions, such as the ability to run server-supplied code on a Web client. We examine the HotJava Web browser and the Java language in which it is implemented. We demonstrate several attacks that compromise HotJava's security. Some of these attacks are made possible through browser code that fails to enforce access permissions, but can be easily fixed. Others point to underlying tension between the openness desired by Web application writers and the security desired by their users. We discuss the interaction of application requirements and security needs and suggest how they can both be accommodated.

1 Introduction

This paper describes several security flaws we found in Sun's HotJava Web browser and attacks that exploit those flaws. We have implemented denial of service and man-in-the-middle attacks. We have also found that HotJava applets can discover some potentially interesting information about the machine and its user, and that several channels exist for leaking this information out of the browser.

1.1 Background

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 fast as new tags are added, there will always be demand for more. In addition, many content providers wish to integrate interactive features such as chat systems and animations.

Rather than further HTML extensions, the notion of downloading a program (called an applet) which runs inside the Web browser has gained popularity. Such network code raises serious security issues; a casual Web reader shouldn't be concerned about malicious side-effects from visiting a Web page. Languages such as Java[10], Safe-Tcl[1], Phantom[3], Telescript[4], and ScriptX[1] [6] have been proposed for running downloaded code, and each has varying ideas of how to thwart malicious programs.

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

The HotJava Web browser is interesting because it both supports running untrusted Java code, and is written in Java itself. Its architecture is very distinct from other Web browsers, such as Netscape Navigator. HotJava has libraries of protocol and content

[1] ScriptX does not appear to address security concerns at all. With Kaleida's targeting of Web applications, we hope they will develop a comprehensive security policy.

handlers, which are meant to be extensible. New protocols and content types can be added to HotJava on the fly. On top of this infrastructure, HotJava constructs a typical browser user interface. In contrast, Netscape Navigator is a monolithic piece of software.[2] While Sun claims this architecture as an advantage for HotJava, it also exposes much more code and state to downloaded applets, increasing the chances of security problems.

In order to be secure, such a system must limit access to system resources such as the file system, the CPU, the network, the graphics display, and the browser's state. The type system should be "safe" – preventing forged pointers and checking array bounds. Additionally, the system should garbage-collect memory to prevent memory leakage, and carefully manage system calls that can access the environment outside the program.

The remainder of this paper introduces the Java language and its access-control semantics, discusses various security problems with the HotJava 1.0 alpha 3 browser, analyzes these failures in the context of secure systems design principles, and explores how Java applets can be provided with a sufficiently expressive API to accomplish their requirements yet preserve the user's security and privacy.

2 Java Semantics

Java is similar in many ways to C++. 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 Hoare-style monitor.

In support of downloaded code, Java introduces the concept of name spaces to distinguish downloaded code from local code. Each separate source[3] of Java bytecode is loaded into a separate name space. This prevents both accidental and malicious name clashes. Bytecode loaded from the local file system lives in a special system-wide name space. 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.

The Java runtime system only knows how to load bytecode from the local file system. To load code from other sources, the Java runtime system calls a subclass of the Java abstract class `ClassLoader`. `ClassLoader` defines an interface for the runtime system to ask a Java program to load a class. Classes are transported across the network as byte streams, and reconstituted into `Class` objects by subclasses of `ClassLoader`. It is possible, at runtime, to determine if a class has a `ClassLoader` in the current dynamic call chain. Java uses this to define different access permissions for local classes vs. downloaded classes. Many security-sensitive calls check if they are in the dynamic scope of a class with a `ClassLoader` (which implies that the current execution flow has passed through code loaded from the network) and throw an exception if so.

Java programmers can combine related classes into a package. These packages work like names spaces in C++[9], modules in Modula-2[11], or structures in Standard ML[7]. While packages can be named hierarchically, the hierarchy does not limit the scope of classes defined in a package. A (package, name space) pair defines the scope of a Java class, method, or instance variable that is not given a `public`, `private`, or `protected` modifier. (Public, private, and protected have the same meaning as in C++: Public classes, methods, and instance variables are accessible everywhere, private methods and instance variables are only accessible inside the class definition, and protected methods and variables are only accessible in the class or its subclasses.)

The Java bytecode runtime system enforces the language's access semantics. Unlike C++, it is not possible to forge a pointer to a function and invoke 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

[2] Netscape Navigator 1.0 is not extensible, while Navigator 2.0 is extensible only by the user manually installing shared libraries (DLLs).

[3] While the documentation[5] does not define "source", it appears to mean the machine of origin. Clearly this is inadequate as multiple organizations may have Web pages on the same machine. Sun has announced plans to include support for digital signatures in a future version.

assures application code cannot break these specifications.

While the Java documentation claims that the safety of Java byte codes can be statically determined at load time, the 1.0 alpha 3 release violates this in at least two places:

1. The type system is covariant[2], so certain operations require run time type checks.

2. The system does not detect access to a package-scoped variable from a different name space until the actual attempt.

These two violations of static typability are unfortunate, as the code verifier is no longer the only piece of the runtime system that must be correct to ensure security. The requirement for dynamic checks also introduces a performance penalty.

3 HotJava 1.0 alpha 3 Security

We analyzed the source code of the Java API and the browser itself, looking for exploitable flaws. We attempted accesses to the file system and the network. We examined denial-of-service and man-in-the-middle attacks. We also attempted to see how much an applet may learn about the system on which it's running.

These attacks won't necessarily apply to Netscape version 2.0. Netscape 2.0, which supports Java applets, has only recently become available in a beta-test version. Netscape's Java implementation appears to differ from Sun's in several aspects, and no documentation from Netscape is available at the time of this writing. In particular, the "pre-beta" Java release includes hooks to call a centralized `SecurityManager` class which implements a reference monitor. However, no policy is defined for the `SecurityManager`, so it is too early to say how much it improves Java's security.

3.1 Covert Channels

HotJava provides a number of security settings for network access. Typically, most users will use the default mode, which only allows an applet to connect to the host from which it was loaded. In fact, HotJava fails to enforce this setting through implementation errors and available covert channels. The `accept()` system call, used to receive a network connection initiated on another host, isn't protected by the usual security checks. 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 its port. Several covert channels are available for this message to be transmitted. If the Web server is running an SMTP mail daemon, the applet can connect to it and transmit an e-mail message to any machine on the net. Additionally, the Domain Name System (DNS) can be used as a two-way communications channel. 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 an arbitrary 32-bit IP number as a reply. Repeated DNS calls by the applet establish a channel between the applet and the attacker's DNS server. If the user asks for a more restrictive mode, such as preventing applets from having any network connections, DNS can still function as a low-bandwidth covert channel since the `InetAddress.getByName()` method is public and does not check the current security mode before making the DNS request. Java 1.0 alpha 3 even has a method where the applet can specify the DNS server, so that the attacker does not need his own domain name. If the Web server is cooperating, the ability for an applet to retrieve a URL is another covert channel. The format of a URL is *protocol://hostname/server-defined*, so any arbitrary information can be encoded in a URL.

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

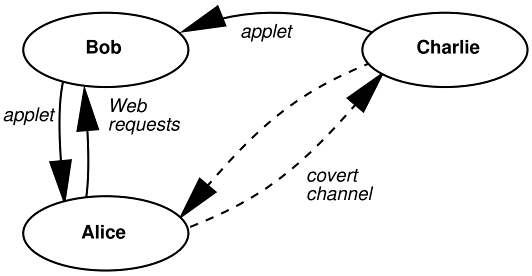

Figure 1: A Three Party Attack — Charlie produces a Trojan horse applet. Bob likes it and uses it in his Web page. Alice views Bob's Web page and Charlie's applet establishes a covert channel to Charlie. The applet leaks Alice's information to Charlie. No collusion with Bob is necessary.

attack. A three-party attack can originate from anywhere on the Internet, bypassing Java's network access controls. A three-party attack can spread if it is hidden in a useful class that gets copied onto many Web pages. The SMTP and DNS channels are examples of three-party attacks, while the URL channel is a two-party attack, assuming that an applet is only allowed to connect to the machine it came from. (See figure 1.)

3.2 Information Available to Applets

If a rouge applet can establish a channel with any Internet host, the next question is what can an applet learn about the user's machine to send over the channel. 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) specify where reading and writing of files or directories may occur without the user's 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 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 holes could be completely removed by editing the ACLs. While the user can do this, the system default should be less permissive.

We can also learn the user's login name and machine name, as well as other environment variables; `System.getenv()` has no access checks either. By probing environment variables and the `PATH` variable, we can often discover what software is installed on the user's machine. This information could be valuable either to corporate marketing departments, or to attackers desiring to break into a user's machine. Java also includes the ability to measure real time, which makes it possible to benchmark the user's machine. As a Java-enabled Web browser may well run on prerelease hardware and/or software, an attacker could learn valuable information.

3.3 Denial of Service

Another class of successful attack is denial of service. The obvious attacks are busy-waiting to consume CPU cycles and allocating memory until the system runs out, starving other threads and system processes. Nothing in Java prevents these attacks. Additionally, we can acquire locks on critical pieces of the HotJava browser to cripple it. For example, the code in figure 2 locks the status line at the bottom of the browser, effectively preventing it from loading any more pages. This attack can be programmed with a time delay before it locks the status line, causing the failure to occur when the user is viewing a different Web page, thereby masking the source of the attack. As with other attacks we've discussed, the `MeteredStream` class is public, allowing the attack to work. This could be fixed through replacing such critical classes with wrappers that don't expose the locks to outsiders. However, the CPU and memory attacks can't be easily fixed; many genuine applications may need large amounts of memory and CPU.

3.4 Man in the Middle

Perhaps the strongest attack is that we can change the browser's HTTP and FTP proxy servers. We can establish our 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 browser. All this is possible simply because the browser state was stored in public variables in public classes. While this attack destroys the users' privacy, its implementation is trivial (See figure 3). By using the property manager's `put()` method, we store our desired proxy in the property manager's database. If we can then entice the user to print a Web page, these settings will be saved to disk, and will be the default settings the next time the user starts HotJava. If the variables and classes were private, this attack would fail. Likewise, if the browser is running behind a firewall and relies on proxy servers to access the Web, this attack will also fail.

4 Security Analysis

While we found some interesting security problems in HotJava, this is hardly a surprising result considering that we were examining alpha-test quality code. More instructive than the particular bugs we found,

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

Figure 2: Java code fragment to deadlock browser by locking its status line.

```
hotjava.props.put("ftpProxyHost", "proxy.attacker.org");
hotjava.props.put("useFtpProxy", "true");
hotjava.props.put("proxyHost", "proxy.attacker.org");
hotjava.props.put("proxyPort", "8080");
hotjava.props.put("proxySet", "true");
FtpClient.useFtpProxy = true;
FtpClient.ftpProxyHost = "proxy.attacker.org";
FtpClient.ftpProxyPort = 8080;
HttpClient.cachingProxyHost = "proxy.attacker.org";
HttpClient.cachingProxyPort = 8080;
HttpClient.useProxyForCaching = true;
```

Figure 3: Code to redirect all HTTP and FTP retrievals.

though, is an analysis of possible causes. There are two primary issues: the availability of (interesting) information to leak, and the existence of channels to leak the information. While it is possible to deny the applet access to any interesting information, this eliminates the possibility of developing many interesting and useful applets. On the other hand, it would be extremely difficult, if not impossible, to remove all the covert channels available to Java applets. While we have discussed two channels, we are sure that many more exist. Thus, we feel it is more useful to attack the first problem.

4.1 Integrity

We note that the architecture of HotJava is inherently more prone to security problems than that of Netscape Navigator, because the HotJava browser's state is kept in Java variables and classes. Any errors in HotJava's variable and class definitions can be exploited for malicious purposes. Variables and methods that are public and protected are very dangerous: they give the attacker a toe-hold into HotJava. Static synchronized methods and public instances of objects with synchronized methods lead to easy denial of service attacks. These are all issues that can be addressed with good design practices, coding standards, and code reviews, but are also indications of deeper problems.

HotJava's architecture does not include an identified trusted computing base (TCB)[8]. Unfortunately, substantial, dispersed parts of the browser must cooperate to maintain security. The bytecode verifier, and interpreter or native code generator must properly implement all the checks that are documented. The Java interfaces to system services must manually check if they are being called in the dynamic scope of downloaded code. 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.

What we have discussed so far is specific to HotJava, in fact to the 1.0 alpha 3 release. These are implementation flaws in the realization of HotJava, not fundamental flaws in the Java language itself. However, we believe that there are also issues which need to be addressed at the Java language level, including: the lack of an explicit security policy, the lack of an audit trail, and some unfortunate choices in the definition of the Java language. After examining these issues, we will discuss some application requirements for Java-enabled browsers, tensions between applications and security policy, and some suggested mechanisms to help implement desired policies, without unduly restricting the expressive ability of Java.

4.2 Policy

The present documents on Java and HotJava do not define a clear 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."[8] Without such a policy, it is unclear how a secure implementation is supposed to behave. 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 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 an appropriate security policy.

4.3 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."[8] The Java language does not define any auditing capability. If we wish to trust a Java program that is extensible via downloading byte code across a network (i.e. HotJava), a reliable audit trail is a necessity. The level of auditing should be selectable by the user. At least files read and written from the local file system should be logged, along with network usage. Users may wish to log the bytecode of all the programs they download. This requirement exists because the user cannot count on the attacker to leave his Web site in place after a successful attack. The bottom line is that the Java runtime should provide a configurable audit system.

4.4 Language Semantics

The Java language definition could be altered to reduce accidental leaks of information, and encourage better program structure with a richer module system than Java's `package` construct. While Java's packages define multiple, non-interfering name spaces, richer interfaces and parameterized modules would be useful additions to the language. Public variables in public classes are so dangerous that it is hard to think of any application for them in their present form. At a minimum, public variables should not be writable across different name spaces. It may be better to simply remove them from 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. (In this case, we use the word "client" to mean a module, or in Java terms, a package.) The introduction of parameterized modules, like Standard ML's functors[7], should also be investigated. Parameterized modules are a solution to the program structuring problem that opened up our man-in-the-middle attack. By passing the transport protocols library as a parameter to the browser package, the proxy control variables would be accessible only to the browser, not to the applets.

5 Application Requirements

A major problem in defining a security policy for Java applets is making the policy general 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, but trustworthy, policy. We assume a HotJava style architecture, where the browser is implemented in Java, and running in the same virtual address space as the applets.

5.1 Web Browser

For a Web browser to attract any interest, it must support all the protocols and content types in current use today, including HTTP (the Web), FTP (file transfer), Gopher, SMTP (email), NNTP (Usenet news), and Finger. FTP presents the most problems, as the FTP specification requires the FTP client to accept a new connection from the server for each data transfer (including directory listings). To implement this, the language must provide access to the `listen()` and `accept()` calls, at least from trusted code. We would like an applet to be able to request a file via FTP, so simply checking whether we are in the dynamic scope of downloaded code is overly restrictive. This suggests that we need to trust the locally provided FTP implementation not to abuse its privileges if we allow it to call `accept()`. (Of course, the FTP client should be subjected to a thorough security audit.)

5.2 Distributed Applications

Other applications that would be convenient to provide as applets include audio/video conferencing, real-time multi-player games, vast distributed computations, like factoring, and HTML editors. Games require access to high-speed graphics libraries; many of these libraries trade speed for robustness and may crash the entire machine if they are called with bad arguments. The Java interfaces to the libraries may have to check function calls; with proper compiler support some of these checks could be optimized away. Games also require the ability to load images across the network (most likely by retrieving URLs), and the ability to measure real time, which makes it more difficult to close the benchmarking and URL retrieval holes. For teleconferencing, the applet needs both access to the network and to your 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 would provide sufficient protection for most users.

Providing a distributed computation as a Java applet would vastly increase the amount of cycles available: "Just click here, and you'll donate your idle time to computing ...." But this requires that a thread live after the user moves on to another Web page, which opens up opportunities for surreptitious information gathering and denial of service attacks. We consider the 1.0 alpha 3 release's not killing threads after leaving a page to be a bug. An HTML editor is another natural applet, but this may require access to the network after accessing the file system, which in general is a recipe for information leakage. The present file system read and write ACLs are not well suited for the HTML editor, as the user should only have to specify the file to read once, not twice (once for security, and once for the application).

5.3 User Interface

The security user interface is critical for helping the average user choose and live with a security policy. Unfortunately, a user may disable security if burdened with repeated authorization requests for the same applet. Applets should request capabilities when they're first loaded. The user's response could then be logged, alleviating the need for re-authorization in the future. Another useful feature would be trusted dialog boxes. An untrusted applet could call a trusted "File Save" dialog 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. A similar trusted dialog could be used for initiating network connections, as might be used in chat systems or games. A combination of formal policies with flexible user interfaces should be able to balance applet needs and user security.

6 Conclusion

Java is an interesting new programming language, designed to support the safe execution of applets on Web pages. While the fundamental mechanisms in Java appear to function as documented, the first Java-enabled browser, HotJava, has many implementation flaws which render it insecure. Our attacks export more information about the client machine and user than other Web browsers provide, show a potential integrity problem on Windows 95, and can provide near perfect eavesdropping on a user's Web browsing. While these particular flaws may be fixed in future releases, we are greatly disturbed by the absence of a well-defined security policy to verify an implementation against. We have attempted to analyze some application requirements to prepare a flexible, yet robust policy, and suggest several useful mechanisms along these lines.

Acknowledgments

Thanks to Andrew Appel and Ed Felten for reading this paper and making many helpful suggestions. Thanks to Sun Microsystems for providing full source to the HotJava browser, making this work possible.

References

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

[2] Castagna, G. Covariance and Contravariance: Conflict without a Cause. Tech. Rep. LIENS-94-18, Département de Mathématiques et d'Informatique, Ecole Normale Supérieure, October 1994.

[3] Courtney, A. Phantom: An Interpreted Language for Distributed Programming. In *Usenix Conference on Object-Oriented Technologies* (June 1995).

[4] General Magic, Inc. *Magic Cap Concepts,* `http://www.genmagic.com/MagicCapDocs/Concepts/*contents.html`, May 1995.

[5] Gosling, J., and McGilton, H. *The Java Language Environment,* `http://java.sun.com/whitePaper/javawhitepaper_1.html`. Sun Microsystems, May 1995.

[6] Kaleida Labs. *ScriptX Technical Overview,* `http://www.kaleida.com/products/sx-tech-overview.html`, 1995.

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

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

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

[10] Sun Microsystems. *The Java Language Specification,* `http://java.sun.com/1.0alpha3/javaspec/javaspec_1.html`, May 1995.

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