Java Bytecode Modification and Applet Security*

Insik Shin
ishin@cs.stanford.edu

John C. Mitchell
mitchell@cs.stanford.edu

Computer Science Department
Stanford University
Stanford, CA 94305

Abstract

While the Java Virtual Machine includes a bytecode verifier that checks bytecode programs before execution, and a bytecode interpreter that performs run-time tests such as array bounds and null-pointer checks, Java applets may still behave in ways that are annoying or potentially harmful to users. For example, applets may mount denial-of-service attacks, forge email or display misleading information in order to trick users. With these concerns in mind, we present techniques that may be used to insert additional run-time tests into Java applets. These techniques may be used to restrict applet behavior or, potentially, insert code appropriate to profiling or other monitoring efforts. The main techniques are class-level modification, involving subclassing non-final classes, and method-level modification, which may be used when control over objects from final classes is desired.

Subject Areas: language design and implementation, analysis and design methods, software engineering practices, experienced with object-oriented applications and systems, security.

1 Introduction

The Java Language [12] has proven useful for a variety of purposes, including system development and the addition of "active content" to web pages. Although previous language implementations, such as Pascal and Smalltalk systems, have used intermediate bytecode, the use of bytecode as a medium of exchange places Java bytecode in a new light. To protect against execution of erroneous or potentially malicious code, the Java Virtual Machine verifies code properties before execution and performs additional checks at run time. However,

---

*submitted to OOPLSA '98

1

these tests will not protect against certain forms of undesirable run-time behavior, such as denial-of-service attacks, irritating audio sounds, or violation of conventions regarding hypertext links. Moreover, users cannot easily customize the tests that are performed since these are built into the Java Virtual Machine.

The goal of our work is to develop methods for enforcing applet properties, in a manner that may be customized easily. In this paper, we propose a technique, called *bytecode modification*, through which we put restrictions on applets by inserting additional bytecode instructions that will perform the necessary run-time tests. These additional instructions may monitor and control resource usage, limit applet functionality, or provide control over inaccessible objects. While our techniques bear some relation to software fault isolation [13], we check different properties and our code operations are specifically tailored to the file structure and commands of the Java Language. Our technique falls into two parts: *class-level* modification and *method-level* modification. In class-level modification, references to one class are modified to refer to another class. Since this method relies on class inheritance and subtyping, it is simple and fast, but can not be applied to final classes and interfaces. In these cases, method-level modification is used since it may be applied on a method-by-method basis without regard to class hierarchy restrictions.

We have implemented our techniques in an HTTP proxy server that modifies classes before they are received by the browser. Although the proxy structure provides an easy way to test our approach, it is not completely comprehensive since, for example, encrypted applets from secure pages are not accessible to the proxy. For this reason, it might also be useful to implement bytecode modification facilities in the browser or virtual machine, or by an extension of the class loader. Our proxy server is controlled by a user interface that runs as a Java applet and may be configured to block access to a specific sites, redirect requests for special Java classes or eliminate tagged advertisements.

In Section 2, we discuss several example Java applet attacks that require techniques beyond the current Java verifier and security model. We explain the bytecode modification techniques in Section 3 and present some examples in Section 4. Experimental performance data appears in Section 5, with comparison to related work on safe execution of Java applets in Section 6. We conclude in Section 7.

2   Java Applet Safety

Before describing a series of techniques for modifying Java bytecode programs, we give some motivating examples of hazardous or undesirable Java applets. Each of the problems outlined in this section can be eliminated or contained using our approach.

2.1 Denial of Service Attack

The Java Virtual Machine provides little protection against denial of service attacks. An applet can make the system unstable by monopolizing CPU time, allocating memory until the system runs out, or starving other threads and system processes. For example, an applet may create huge black windows on the screen in such a way that the users cannot access other parts of the screen, or it may open a large number of windows [5]. Many machines have limits on the number of windows that can be open at one time and may crash if these limits are exceeded. Since the safety of the Java runtime system may be threatened by inordinate system resource use, it is useful to have some mechanism to monitor and control resource usage.

2.2 Disclosure of Confidential Information Attack

There are many ways that an applet may communicate information to another site on the network. For example, browsers such as Navigator, Internet Explorer and HotJava provide a network security mode which allows an applet to connect to the web server from which it was loaded. In addition, there are a variety of covert channels that are difficult to detect. One example involves 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 [3]. Time-delayed access to files also can be used as a covert channel [10]. Specifically, if an applet, A, with access to private information is prohibited from accessing the net, information can still be sent out by another applet, B, which shares a file with applet A.

While it seems difficult to eliminate all possible covert channels, some may be prevented using our techniques. An example we discuss later in the paper is forged email. If a web server is running an SMTP mail daemon, an applet can interact with sendmail after connecting to port 25 on the web server. This allows a hostile applet to forge email. This form of email forgery may be prevented by disallowing connections to port 25. Inter-applet communications using storage channels can be detected by monitoring the actions of applets through logging facilities.

2.3 Spoofing Attack

In a spoofing attack, an attacker creates a misleading context in order to trick a user into making an inappropriate security-relevant decision [4]. Some applets display the URL that will be accessed when the the mouse is held over a graphic or link. By convention, the URL is shown in a specific position on the status line. If an applet displays a fake URL, the user can be misled. This could allow an applet to mislead a user into connecting to a site that is hazardous in some way. Fortunately, this spoofing attack can be controlled by enforcing conventions about the URL displayed on the status line.

2.4   Annoyance Attack

An applet can annoy users with a very noisy sound which never ends. This form of sound attack exploits a useful feature of Java, the ability to play sound in the background. To eliminate an annoying sound, however, users typically must kill the thread playing sound, disable the audio, or quit the browser. All of these can be inconvenient. Another possible annoyance attack is to make the browser visit a given web site over and over, popping up a new copy of the browser each time. One way to manage annoying sounds is to provide the ability to turn the sound off. To do so, the Java runtime system must monitor and control objects with sound. Similar methods can be used to handle repeated visits to a certain site, and other annoyance attacks.

3   Java Bytecode Modification

This paper presents a safety mechanism for Java applets that is sufficient to solve the problems summarized above. The basic idea is to put restrictions on applets by inserting safeguarding code. In the examples we have implemented and tested, safeguarding code may monitor and control resource usage as well as limit the functionality of applets. Our techniques bear some relation to software fault isolation [13], but involve different run-time properties. In addition, our code operations are specifically tailored to the file structure and commands of the Java Language.

Our safety mechanism substitutes one executable entity, such as a class or a method, with a related executable entity that performs additional run-time tests. For instance, a class such as `Window` can be replaced with a more restrictive class `Safe$Window` that performs additional security and sanity checks. (We use the prefix `Safe$` to indicate one of our safe classes.) This safety mechanism must be applied before the applet is executed. For convenience in developing a proof of concept, applets are currently modified within an HTTP proxy server that sits between a web server and a client browser. This implementation does not require any changes in the web server, Java Virtual Machine or web browser. Since applets and the browser are not notified of changes in the applet, subsequent requests for safeguarded executable entities may be issued to the web server, which does not have them. This problem is handled by having the proxy redirect these requests to cites where the safeguarding entities are actually stored.

The following sections explain how modified executable entities are inserted in Java bytecode. The modifications may be divided into two general forms, class-level and method-level modifications.
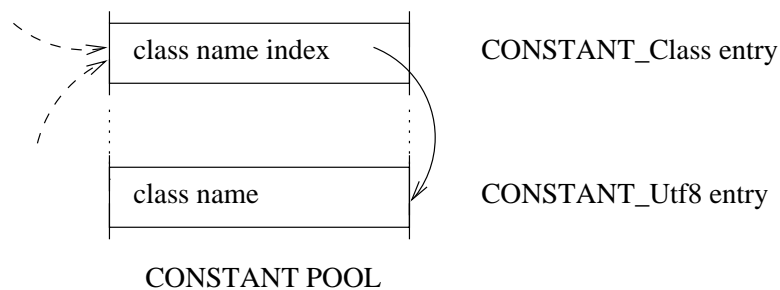
Figure 1: A class is represented with two entries in the constant pool

3.1   Class-level Modification

A class such as `Window` can be replaced with a subclass of `Window` (which will be called `Safe$Window` in this example) that restricts resource usage and functionality. For example, `Safe$Window`'s constructor method can put a limit on how many windows can be open on the screen. The method allows new windows to be created until the number of windows exceeds the limit. If the limit is exceeded, the method throws an exception indicating that too many windows are open. Since `Safe$Window` is a subtype of `Window`, type `Safe$Window` can appear anywhere type `Window` is expected. Hence, the applet should not notice the change, unless it attempts to create windows exceeding the limit.

This example of class-level substitution is done by merely substituting references to class `Window` with references to class `Safe$Window`. In Java, all references to strings, classes, fields, and methods are resolved through indices into the constant pool of the class file [6], where their symbolic names are stored. Therefore, it is the constant pool that should be modified in a Java class file. In more detail, two entries are used to represent a class in the constant pool. A class is represented by a constant pool entry tagged as CONSTANT_Class which refers to a CONSTANT_Uft8 entry for a UTF-8 [1] string representing a fully qualified name of the class, as shown in Figure 1.

If we replace a class name of a CONSTANT_Uft8 entry, `Window`, with a new class name, `Safe$Window`, the CONSTANT_Class entry will represent the new class, `Safe$Window`, as shown in Figure 2.

Class-level substitution requires a simple modification of a constant pool entry, since it takes advantage of the property of class inheritance. Obviously, however, the use of class inheritance prevents this approach from being applied to final classes or interfaces.

---

[1] The Unicode Standard, version 1.1, and ISO/IEC 10646 1:1993 jointly define a 16 bit character set which encompasses most of the world's writing system. UTF 8, one of UCS transformation formats, has been developed for the compatibility between the 16 bit characters and many applications and protocols for the US ASCII characters. For more information regarding the UTF 8 format, see *File System Safe UCS Transformation Format (UTF-8)*, X/Open Preliminary Specification, X/Open Company Ltd., Document Number: P316.

DOCKET
A L A R M

Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

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

Real-Time Litigation Alerts

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

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

Advanced Docket Research

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

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

Analytics At Your Fingertips

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

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

API

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

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

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

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

fastcase
Smarter legal research.