WHEN JAVA WAS ONE: THREATS FROM HOSTILE BYTE CODE

MARK D. LADUE

ABSTRACT. In Java's first year it has become clear that many of the problems posed by executable content have not been solved. The almost exclusive focus of the Java community on executable content has left numerous avenues unexplored for threats. It has been observed that there is no one-to-one correspondence between Java source code (programs) and Java byte code (class files). While every program written in Java can be compiled to byte code by a Java compiler, it is possible to create class files which no Java compiler can produce, and yet, which pass the Java Verifier with flying colors. This fact has one very serious implication - No matter what claims are made, and even formally demonstrated, for the security of the Java language, all bets are off when it comes to byte code running in the Java Virtual Machine. This paper will explore some of the implications of this curious lack of coherence between Java source code and byte code. It will also illustrate how easy it is to alter Java class files for malicious purposes.

1. THE STATE OF JAVA SECURITY

The Java programming language has recently turned one year old. In its first year, Java has had a number of spectacular holes punched in its security model by Ed Felten and the Safe Internet Programming Team at Princeton University [McGF]. Since February of 1996 the Hostile Applets Home Page at the Georgia Institute of Technology's School of Mathematics has featured a collection of evil applets (with complete source code), and yet Sun Microsystems and its corporate partners have shown little progress in combatting hostile applets [LaD1, LaD2]. A year ago, when Java was first gaining notoriety, few people imagined that so many serious flaws would surface so quickly, and even fewer believed that threats from hostile applets would persist. In Java's first year it has become clear that many of the problems posed by executable content have not been solved. The power and complexity of the language make it extremely likely that security holes will continue to appear in years to come.

It has been observed that there is no one-to-one correspondence between Java source code (programs) and Java byte code (class files) [McGF, LaD4]. While every program written in Java can be compiled to byte code by a Java compiler, it is possible to create class files which no Java compiler can produce, and yet, which pass the Java Verifier with flying colors. Such class files are said to be *deviant*. Not only is it possible to create deviant class files, it is a simple matter to do so, and the number of these non-compiler class files greatly exceeds the number of those producible by Java compilers. This fact has one very serious implication - No matter what claims are made, and even formally demonstrated, for the security of the Java language, all bets are off when it comes to byte code running in the Java Virtual Machine. Deviant class files that pass the Verifier and exploit unenforced, or improperly implemented, Verifier rules have the potential to reduce Java Security to rubble.

Note that this applies as well to the most untrusted of applets (which are Java programs downloaded and run automatically by most browsers) as it does to applications (which are programs set up and run in more traditional ways). While inadvertently trusting a hostile application can lead to ruin, so can accidentally downloading a hostile applet that exploits the increased power of Java byte code over Java source code. Thus the distinction between applications and applets is unimportant in the present context. Until this new threat posed by Java applets is more fully understood and explored, it is wise to regard applets with more suspicion than ever before.

This paper will explore some of the implications of this curious lack of coherence between Java source code and byte code. It will also illustrate how easy it is to alter Java class files for malicious purposes. Section 2 contains an overview of some salient facts about the Java class file format. It highlights the ease with which class files can be altered to become deviant and do things beyond the power of Java source code. Section 3 describes the problem of incoherence between Java source code and byte code. It points out several surprising properties of byte code as well as several rules unenforced by the Java Verifier, all of which could lead to security breaches. Section 4 then introduces a number of examples in order to illustrate the threats. One particularly interesting example that will be considered at length is the application HoseMocha.java, which can be applied to applications and applets, making them impervious to the celebrated Mocha decompiler. Finally, Section 5 recounts recent experience with some rudimentary Java Platform viruses, and it assesses the possibility of more virulent threats from hostile byte code.

2. An Overview of the Java Class File Format

When Java source code is compiled, the result is a *class file*, having a *.class* extension and containing platform-independent *byte code* in a very specific format. A class file should be regarded as a stream of 8-bit bytes, with 16-bit, 32-bit, and 64-bit quantities being constructed in big-endian order from two, four, and eight consecutive 8-bit bytes, respectively. The

Java Virtual Machine (JVM) Specification represents a class file in a C-like structure notation as follows [Lind]:

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count - 1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Here the notation `un` refers to an unsigned n-byte quantity. While this structure gives some idea of the nature of Java class files, it will be helpful to take a closer look at a few of the details.

The 4-byte quantity `magic` has the value `0xCAFEBABE` and identifies the class file as such. The 2-byte quantities `minor_version` and `major_version` specify which version of the Java compiler produced the class file. The `constant_pool` is a table of structures that represent an assortment of class, field, and method names as well as string and other constants used within the class file. The `constant_pool_count` specifies how many entries are present in the `constant_pool`, while each `cp_info` structure is one of eleven different types that may appear in the `constant_pool`. The 2-byte quantity `access_flags` is a mask of modifiers used to specify class and interface accessibility. An extended form of `access_flags` also occurs in the `field_info` and `method_info` structures, where it serves the same purpose. The 2-byte quantities `this_class` and `super_class` refer to the `constant_pool` entries containing precisely what their names indicate.

The remainder of the class file consists of four tables, with one table each for `interfaces`, `fields`, `methods`, and `attributes`. Each table is preceded by a 2-byte quantity specifying the number of entries in that table, and each entry in a particular table is a structure of a type appropriate for that table. While each of these tables is an integral part of the class file, the `methods` table contains the byte code to be run in the JVM, and so a closer look at it is in order.

The `methods` table of a Java class file contains `methods_count` entries, and each entry is a structure of type `method_info`, which has the following format:

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

The JVM class file specification offers six predefined types of `attribute_info` structures:

1. Code
2. ConstantValue
3. Exceptions
4. LineNumberTable
5. LocalVariableTable
6. SourceFile

The most important of these `attribute_info` structures is the `Code` attribute, which contains the JVM instructions for a single Java method and has the following format:

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    table_info exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

The `code` array contains the bytes of code actually run by the JVM. Each byte of the `code` array is either a legal Java *opcode*, of which there are 201 at the present time, or an operand of an *opcode*. The `code` array, like the class file as a whole, is subject to a multitude of static and structural constraints, all of which must be checked by the Java Verifier [Lind].

While the class file format greatly enhances Java's security by making the verification process much more tractable, it also raises some security concerns of its own [McGF]. The well-defined format and level of detail present in class files make it a straightforward, though tedious, task to recover source code from them. The justly celebrated Mocha decompiler does precisely that. Using the Mocha decompiler, for example, it is an easy matter for one to decompile class files to source code and scour them for security weaknesses, and it is just as easy for a Java developer to decompile a business competitor's work and search for trade secrets. What can be read can often be rewritten, but one need not go to all of the effort of decompiling class files to source code, editing that source code, and re-compiling it to obtain hacked class files.

The hacker who knows a bit of Java programming, the class file format, and Java opcodes can easily insert, delete, or otherwise alter code in class files, all without effect on the class files' verifiability. To insert some code, for instance, one need only append entries to the `constant_pool`, append the appropriate opcodes to a suitable method's `code` array, and use the `goto` instruction (167 or `0xa7`) to jump to and from the inserted code. One has only to be careful and adjust the appropriate counts in the class file to maintain its verifiability. When the class file attacked happens to be `SecurityManager.class` or `AppletClassLoader.class`, more dire consequences would be sure to follow. The class `java.io.RandomAccessFile` has handy methods for reading and writing Java primitive data types, including unsigned 1-, 2-, and 4-byte quantities, at arbitrary locations in files. With Java's power and ease of use, it takes a scant few hours to develop the knowledge and skills required for the task.

Thus it is extremely simple to read and manipulate Java class files for evil purposes. In particular, it is very easy to take byte code produced by a Java compiler and alter it to produce deviant class files. It is quite possible that deviant class files will provide a new avenue for the attack applets that the Princeton team developed last year. But one need not be so sophisticated in order to develop a devastating attack applet. An industrious hacker could just as easily produce and test randomly generated deviant classes until a suitably destructive one appears. Moreover, what can be done by a hacker can just as easily be done by a virus. In the future we should expect class files to become tempting targets for hackers and virus writers.

3. Java Source Code Versus Byte Code

The overview of the class file format in the previous section revealed how simple it is to systematically inspect and tamper with Java class files. That alone is cause for concern. But the lack of a one-to-one correspondence between Java source code and class files which pass the Java Verifier is cause for much greater concern. There is fundamental incoherence between the Java programming language and the byte code passed by the Verifier.

A Java compiler will take any valid Java program and produce a class file which will pass the Verifier, yet there are class files which pass the Verifier but correspond to no valid Java program. Such *deviant* class files contain code not derived from legitimate programming constructs. Thus the JVM allows byte code to extend the Java language far beyond its official boundaries. Since some of Java's security policies depend upon the language itself, this is a potential source of serious security breaches. To make matters worse, the ease with which class files can be altered entails that it is utterly simple to produce deviant byte code. It also entails that the quantity of deviant byte code is vastly greater than that of the legitimate byte code produced by Java compilers.

One source of the problem is that the class file format is entirely independent of the Java language. While the Java Verifier can check with 100% certainty whether or not a given file is a bona fide class file, the Verifier does not, and most likely cannot, determine whether or not that file was produced by a Java compiler. The Verifier's function is to perform a multitude of tests to make sure that a potentially hostile file is *consistent* with *some* of the Java language's most important constraints. Thus it should be no surprise that byte code is more powerful than source code. The words of the Sun programmer who wrote `check_code.c`, the heart of the Java Verifier, are particularly apt: "All currently existing code passes the test, but so does lots of bad code." How true.

To compound the problems, a number of significant constraints are not enforced by the Verifier, and some opcodes possess more functionality than is apparently used by Sun's Java compiler. Examples of this abound. *The Java Virtual Machine Specification* lists three specific properties of exception handlers which byte code produced by Sun's Java compiler always possesses, but which are *not* checked by its Verifier:

1. The ranges of instructions protected by distinct exception handlers must be disjoint, or one must contain the other;
2. An exception handler cannot occur in code protected by itself;
3. Control cannot be passed to an exception handler's code by any means other than an exception.

These constraints are not enforced because supposedly "they do not pose a threat to the integrity of the Java Virtual Machine" [Lind, p.133]. It remains to be seen whether or not such a cavalier view of Java's security is justified.

One need not rest content with such officially documented examples; others are readily found. An apparently harmless example is that arbitrary bytes can be appended to class files without effecting their verifiability, contradicting the JVM Specification [Lind, p.125]. A more interesting case study is the `goto` instruction (167 or `0xa7`). While the Java language studiously avoided the `goto` statement, one is built into the JVM instruction set. Although Sun's Java compiler seems to always employ it in the forward direction, passing `goto` a sufficiently large offset allows it to pass control backwards. This allows deviant byte code to achieve arbitrary transfer of control within a method. In particular, arbitrary branches into and out of `catch` and `finally` blocks work perfectly well, and the better controlled `jsr` (168 or `0xa8`) and `ret` (169 or `0xa9`) instructions can be bypassed. It is quite possible that a deviant use of `goto` together with highly non-standard exception handling could open the back door of the JVM for attack applets to enter. In any event, deviant byte code running in the JVM is unpredictable and makes a mockery of Java's claims to security.

The surprising flexibility of the `goto` instruction has another interesting use. It is especially easy to append opcodes to any `code` array, execute that code, and return. This is very good news indeed for virus writers. Moreover, the existence of such appended code exposes another weakness in the Verifier - appended code is not consistently verified. If control is transferred to it at some point, it almost certainly is, but dead opcodes appended are sometimes inspected and sometimes not. This is troubling and further raises the prospects of deviant byte code harboring nasty surprises. This combination of language-dependent security, rules not enforced by the Verifier, undocumented behavior of opcodes, and inconsisent code checking entails a great deal of risk. At the very least it implies that very little can be asserted with certainty about Java Security. Clearly much more work remains to be done before Java can be declared safe.

4. BATTLE OF THE BYTE CODE

The plea to ponder potential security threats always seems less convincing in the absence of concrete examples. In order to illustrate the threats posed by deviant and subversive byte code, the author created a number of examples. Some are lighthearted, while others are more serious. All of these examples are readily obtained over the World Wide Web [LaD5]. The main point of these examples is that it is very easy to alter Java class files and just as easy to create deviant byte code, but a corollary is that Java class files are also tempting targets for hackers and virus writers. In pondering these examples, one should recall Frederick Cohen's amusing tale of an expert's reaction to one of his early demonstrations [Coh2, pp.35-36]:

> ... and he got really upset. He said "I don't know why we had you here, you're the worst programmer I have ever seen. In 15 minutes, I could write a program to

do this, it would go through the system like a hot knife through butter, and nobody would even notice it." So I said, "That's the point!" He finally got the point.

As a first example, the author wrote and then attacked Beginner.java, which was contrived to `try` reading a nonexistent file, `catch` the resulting `IOException` and print an error message ("Oops!"), and `finally` print one last message ("Help!"). After inspecting Beginner.class, the author wrote Attacker.java, which inserted 3 bytes of code into Beginner.class at the end of its `finally` block: `goto` followed by the 2-byte offset necessary to return control to the beginning of the program. Attacker also adjusted the `attribute_length` and `code_length` of the proper `Code_attribute` structure to maintain verifiability. The altered Beginner.class readily passed the Verifier, and when run, it proceeded into an infinite loop of printing "Oops!" and "Help!" messages, as expected. This deviant class file could not have been produced by a Java compiler, and the Mocha decompiler failed to decompile it. A second version of Attacker.java was able to insert similar code into the `catch` block and achieved a similar effect, though it also had to alter the `Exceptions` table to preserve verifiability.

A more interesting example is PublicEnemy.java. Given a target directory, this Java application searches it and all of its subdirectories for Java class files. Once a class file is located, PublicEnemy alters the contents of its `access_flags` for the class, its fields, and its methods. The results are the following:

- The class becomes `public`.
- Any `final` fields and methods become non-`final`; any non-`public` fields and methods become `public`; and all `public` fields and methods remain so.

Attacked classes pass the Verifier and continue to run just as before, and their sizes do not change. But they are then open to the maximum amount of inspection and abuse by other Java classes. The system whose `netscape.applet.*` classes fell prey to PublicEnemy would allow Java applets to rampage through it. Thus PublicEnemy would be a fearsome payload for a Java Trojan horse or virus to carry. It also makes an important point about the dangers of trust for Java - Inadvertently trusting a single hostile class a single time can lead to a swift, silent, and thorough compromise of all Java-related security.

In the examples considered so far, Java class files have altered other class files to achieve dubious ends. But a class file is certainly capable of inducing mutations in itself, while remaining acceptable to the Verifier, and taking actions based upon its history. A very simple application, Mutator.java, was created to illustrate this capability. By altering a single byte of its own class file each time it runs, Mutator keeps track of the number of times it has been run and deletes itself, together with its source code, if present, on the sixth attempted run. This example shows that even simple class files are capable of adaptation and learning. The power of the Java language, and the greater power of byte code, when combined with the transparency of the class file format, make it feasible to create armies of intelligent class files that can attack, defend, and maneuver in file systems by exploiting the Java Virtual Machine.

The main point of these examples is not the obvious one, that Java class files are easy prey for Trojan horses (destructive programs that appear to be benign) and viruses (programs that reproduce and may or may not be overtly destructive). Rather the points here are that Java class files are easy to alter, that deviant class files are simple to manufacture, and that the power of byte code extends well beyond that of the Java language. These points are perhaps better illustrated by the application HoseMocha.java.

It has been observed that the Java class file format makes it a simple matter to recover valid source code using decompilers such as Mocha. As was the case with the altered Beginner.class, deviant byte code resists decompilation because it corresponds to no Java source code. This suggests that class files could be protected from decompilation by making them deviant, while preserving their functioning. The Java application HoseMocha.java does precisely that, and it protects the class files of both applications and applets equally well. Its operation is simple. It sifts through a given class file until it arrives at the `methods` table. Once there, it inspects each method's attributes and looks for the method's `Code_attribute`. After finding that, it increases the `attribute_length` by 1, increases the `code_length` by 1, and inserts a dead opcode (for the `pop` instruction in this case) at the end of the `code` array. It so happens that the inconsistent Verifier ignores this frivolous `pop`, so that the altered class files are successfully verified and continue to function as designed. But the Mocha decompiler, trying desperately to do the impossible, gets a segmentation violation and dies when fed such deviant class files. If a future release of Mocha were to start defending itself, new strategies could be devised to protect class files from decompilation into source code. More sophisticated tools could rewrite most byte code in non-standard ways and make class files much more resistant to decompilation. It should now be clear that the power of byte code greatly exceeds the power of Java source code.

5. More Virulent Threats

From the preceding examples it is but a short step to the realm of the virus. The threats posed by PC viruses are well understood, and ample tools for their prevention, detection, and elimination are readily available. There is a widely held myth, however, that UNIX systems are somehow immune to viruses [Rada]. Of course some of Frederick Cohen's pioneering work on viruses was carried out on UNIX systems [Coh1, Coh2], and in the late 1980's Tom Duff and M. Douglas McIlroy developed several strains of UNIX viruses at AT&T Bell Laboratories [Duf1, Duf2, McIl]. Java's "write once, run anywhere" capability offers the possibility of universal computer viruses, which would lay to rest the myth by putting UNIX systems on a par with DOS machines as havens for viruses. The examples of the preceding section illustrate numerous skills useful to *Java Platform Viruses*, and class files can offer fine mobile homes for them.

As proof-of-concept the author created several applications, which, when inadvertently trusted and run, infect the unfortunate user's system with malware or viruses:

- Homer.java generates and executes homer.sh, a Bourne shell script virus which would work on any UNIX platform by appending itself to all Bourne shell scripts in a user's home directory. It would be just as easy to write a Java Trojan horse to detect the user's platform and infect it with an appropriate virus.
- Hijacker.java is a Java Trojan that subverts Sun's `javac` by adding a hostile `main` class to the user's `CLASSPATH` ahead of `classes.zip`. In this case the subverted compiler simply announces its presence and appends the string "Hijacked!" to class files that it produces, but it could just as easily infect them with a Java Platform virus.
- CopyCat.java attacks Java source code in a user's home directory by inserting the necessary code to make it viral. When compiled and run, infected applications do likewise.
- VAppMaker.java also attacks Java source code. It compiles the infected source code and restores it to its original state, leaving only infected class files, which do likewise. It also makes applets able to run as applications, increasing the odds of further infection.

Of course these last two Java viruses are necessarily slow in their activities and obvious in their effects; they stand no chance of being successful in the wild and simply serve to illustrate the possibilities.

A more realistic example is provided by ClassHacker.java. This application attacks Java class files directly, inserting the necessary byte code to make them viral and making all of the adjustments needed to insure that infected class files continue to pass the Verifier and run as before.

ClassHacker uses the same techniques for manipulating class files as the examples of the preceding section, and the amount of code that it inserts is about 2K bytes. Though it acts swiftly, the viral code makes no effort to conceal or defend itself, and so it is unlikely to be very successful.

While the examples of Java Platform viruses considered here are not particularly threatening, more virulent threats are sure to arise in the future. The evil that hostile byte code can perpetrate is limited only by the power of the Java Virtual Machine and human ingenuity. At the very least, the JVM will require integrity maintenance mechanisms to defend its crucial classes from tampering, and users will need to be enabled to better defend the files in their `CLASSPATH`s. Additional levels of trusted code in the Java programming environment will complicate matters and may create more problems than they solve. Key management is sure to come under attack by hostile byte code, and the stakes will be higher. One misapplication of trust can open a back door to attack applets and lead to a total breakdown of all Java-related security.

What does this tell us about Java Security? As long as there remains such a vast gulf between the Java programming language and the byte code accepted by the Java Verifier, very little can be asserted with any certainty about Java Security. Given the complete independence of the Java language and the class file format, bridging this gulf is going to be a very difficult task. The power of byte code beyond the Java language and the ease with which class files can be attacked may become a source of recurring problems for Java Security. This entails that Java applications and applets should be accorded little if any trust until the problems are better understood and solutions are found. Thus it seems extremely likely that the problems will continue as Java grows up and moves from its childhood sandbox to a much harsher adulthood.

6. References

[Coh1] Frederick B Cohen. *Computer Viruses - Theory and Experiments.* Computers & Security, Volume 6, Number 1, pp.22-35. Elsevier Advanced Technology, Oxford, 1987.

[Coh2] Frederick B. Cohen. *A Short Course on Computer Viruses, 2nd Edition.* John Wiley & Sons, New York, 1994.

[Duf1] Tom Duff. *Viral Attacks on UNIX Systems.* Proceedings of the Winter 1989 USENIX Conference, pp.165-171. USENIX Association, January, 1989.

[Duf2] Tom Duff. *Experience with Viruses on UNIX Systems.* Computing Systems, Volume 2, Number 2, pp.155-171. USENIX Association, Spring, 1989.

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

[LaD1] Mark D. LaDue. *Pushing the Limits of Java Security.* Tricks of the Java Programming Gurus, Chapter 23. SAMS.net Publishing, Indianapolis, 1996.
Available at http://www.math.gatech.edu/~mladue/HostileArticle.html.

[LaD2] Mark D. LaDue. *Java Security: Whose Business is it?* The Online Business Consultant. May, 1996.
Available at http://www.math.gatech.edu/~mladue/OBCArticle/Article.html.

[LaD3] Mark D. LaDue. *Java Insecurity.* Computer Security Journal. Computer Security Institute, San Francisco. Spring 1997.
Available at http://www.math.gatech.edu/~mladue/Java_insecurity.html.

[LaD4] Mark D. LaDue. *Java Security: From Eggs to Applets.*
Available at http://www.math.gatech.edu/~mladue/eggs_to_applets.html.

[LaD5] Mark D. LaDue. *Hostile Java Source Code.*
Available at http://www.math.gatech.edu/~mladue/SourceCode.html.

[Lind] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, MA, 1997.

[McGF] Gary McGraw and Edward Felten. *Java Security: Hostile Applets, Holes and Antidotes.* John Wiley & Sons, New York, 1996.

[McIl] M. Douglas McIlroy. *Virology 101.* Computing Systems, Volume 2, Number 2, pp.173-181. USENIX Association, Spring, 1989.

[Rada] Peter V. Radatti. *The Plausibility of UNIX Virus Attacks.* CyberSoft, Incorporated. April, 1996.

[Thom] Ken Thompson. *Reflections on Trusting Trust.* Communications of the ACM, Volume 27, Number 8, pp.761-763. August, 1984.

School of Mathematics, Georgia Institue of Technology, Atlanta, GA 30332-0160
E-mail address: mladue@math.gatech.edu