6.857 Computer and Network Security November 15, 2001

Lecture Notes 20 : Java Security

Lecturer: Ron Rivest *Scribe: Krishnamurthy*

Outline

- Introduction
- The Java Language
- Compiling Java
- Java vs. ActiveX
- Sandbox Model
- Security Policy
- Bad Applets

1 Introduction

One of the underlying themes in computer security is the fact that computer programs can be leveraged against one another. In 6.857, we have talked about ways in which computer systems have been broken, such as viruses and the Internet worm. Since the 1980s, it has become increasingly easier to transfer a program from one computer to another. In fact, the transfer of files between computers is *invited* these days. What was once considered a "no-no" is now the norm and as a result, the job of virus designers has become much simpler.

The Java programming language, since its inception in 1995, has grown at an alarming rate. According to the Sun Microsystems Web site, Java is based on one simple idea: code that can work just about everywhere. The platform-independence of Java has essentially made mobile code possible. One of the neat features of Java is that Java applets can be embedded into HTML code, allowing Internet users anywhere to download code and run it locally.

Naturally, executing foreign code at the local level introduces a variety of security risks. For instance, the code could have a virus attached to it. Moreover, the code could "appear" to be doing what it's supposed to do, when in fact it is doing something malicious. One of Sun's commonly-used marketing slogans for Java has been "Write once, run everywhere." However, because of the security issues involved in running foreign code, security experts are quick to remind us that in actuality it should be "Write once, *test* everywhere."

The scope of this problem is rather large. The fact that all of today's Web browsers are Java enabled puts all Internet users at risk. As a result, Java security remains a hot topic in computer science today.

2 The Java Language

According to Sun Microsystems, Java is a "simple, object-oriented, distributed, interpreted, robust, **secure**, architecture neutral, portable, high-performance, multi-threaded, and dynamic language." (Whew!)

- *Object Oriented:* Java exhibits a class structure, in which each class consists of data objects and the methods that manipulate them.

- *Strongly Typed:* This feature is central to Java's security model, as it prevents arbitrary access to memory and protects against illegal typecasting. Figure 1 below illustrates the essence of type safety. This figure is modeled after the one in Chapter 2, Section 10 of *Securing Java* by McGraw and Felten.

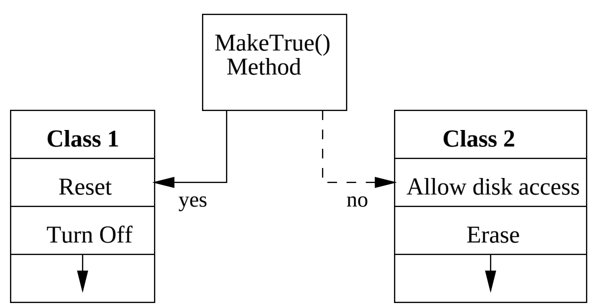

Figure 1: The figure is an illustration of two classes, Class 1 and Class 2, along with two of their fields. Type safety prevents a program from applying the "reset" method to an object of Class 2.

- *Multithreaded:* Programs can execute multiple tasks simultaneously.

- *Garbage Collection:* Rather than have the program allocate memory itself (like C and C++), Java's memory management scheme runs as a thread that frees up memory locations that are no longer needed by the program. This protects against "dangling pointers," a common security problem often associated with C.

- *No Pointers:* Memory is managed by reference. In C, pointers can be manipulated by arithmetic, resulting in a variety of bugs. The fact that Java has no pointers makes it more reliable and thus safer.

- *Exception Handling:* Through exception handling, errors that might normally crash the system are handled at runtime with relative ease in Java.

Question: What exactly is the adversary model for Java?
Answer: In general, when it comes to Java, the adversary is a bad applet running off a Web site that uses holes in the language to get sensitive information from the user, such as credit card numbers, passwords, or social security numbers. The bad guy writes the program; the good guy runs it.

Question: What about Javascript or other scripting languages?
Answer: Javascript has no relation to Java – it was just given the name as a marketing ploy. In any case, no languages are perfect. The objective is to create a sophisticated language that people can download *and* run safely. To say the least, it is a very hard problem.

Question: How does Java's hierarchical namespace impact its security?
Answer: This is a problem of ambiguity. When two parts of a system are referring to the same name, but different objects, a lot of confusion can occur, sometimes to the point of compromising security.

3 Compiling Java

Arguably the most distinguishing feature of the Java language is its platform independence. Java's Virtual Machine (VM) is what makes this possible.

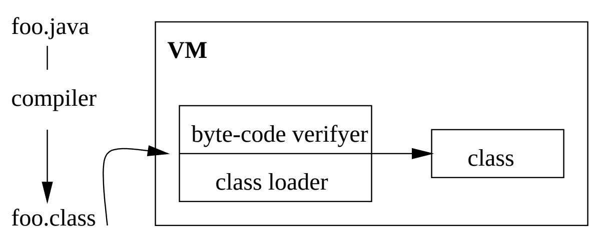

Figure 2: When foo.java is compiled, the compiler converts it to platform independent bytecode, which is then run on a virtual machine.

Figure 2 illustrates the process in which a Java program is transformed into an executable. The program "foo.java" is compiled and the resulting object code is scrutinized by the verifier. Then, the class loader outputs the result of the compilation, a series of class files. Both the verifier and class loader are covered in detail in Section 5.

4 Java vs. ActiveX

Although Java is probably the most popular mobile code system around, it certainly isn't the only one. Many consider Microsoft's ActiveX the direct competitor to Java. Thus, it is worth our time to see how their security models compare.

4.1 Java : Sandbox

Java uses an approach called *sandboxing*. The idea is to make untrusted code run inside a box (or sandbox, take your pick) and give it very limited capabilities. In other words, all code is treated skeptically. The sandbox model will be investigated further in the sections that follow.

4.2 ActiveX : Code Signing

ActiveX uses a very different approach from Java. In ActiveX, code is digitally signed by some trusted party. "Code signing" is a black and white, trust-based model. In other words, you either trust the code or you don't. If you trust the code, it can run with its full capabilities on your machine. If you don't trust the code, it cannot run at all. This model hinges on trusting the proper parties. Trusting the wrong piece of code could have disastrous consequences.

Question: It's really frustrating that the sandbox model by default gives code very little capabilities, especially when compared to ActiveX.
Answer: Well, it gets harder for users as security needs increase. User interface is a huge research area. Can someone from the street implement a secure system with relative ease? Honestly, this is as much a problem of human engineering as it is computer science.

5 Sandbox Model

Java's sandbox model is made up of three interrelated parts: the verifier, class loader, and security manager. Together, they perform the compile-time and run-time checks necessary to implement Java's security protocol.

5.1 Sandboxing Applets

Applets are probably the most important form of mobile code in Java's security model. Mobile code downloaded from the Web cannot be trusted and thus must face a variety of restrictions.

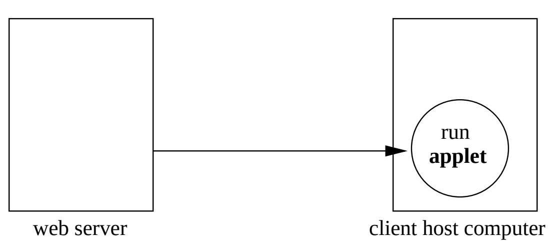

Figure 3: When an applet is downloaded from the Web, it is sent from the Web server to the client host machine, where it is actually run.

These restrictions are enforced by the Java sandbox. When an untrusted applet is downloaded, the default sandbox calls for a variety of restrictions. For instance, the applet cannot:

- read, write, or delete files
- rename files
- create a directory or list the contents of a directory
- check to see if a file exists
- obtain information about a file
- create a network connection, except back home, where the applet originated
- listen for or accept network connections on any port
- create top-level window without the "untrusted window" banner
- obtain the username, or the home directory name of a user
- run any program
- exit the interpreter
- load dynamic libraries
- manipulate any threads other than its own thread group
- create a class loader or a security manager

Some restrictions are implemented at run-time, while others are implemented at compile-time.

5.2 The Verifier

As we pointed out in Section 3, a Java program is compiled into platform-independent object code, which is then run on the Java VM. Before the bytecode can run, however, it must be verified because the compiler cannot be trusted. For efficiency sake, the verifier performs as much checking as possible without actually executing the program. The verifier makes the following checks on the bytecode:

- The class file has the correct format.
- Stack parameters all have the proper type and stacks are not overflowed. (1 for recursive calls, 1 for operands).
- No illegal type casts (remember, Java is a strongly-typed language).
- All register operations (LOAD and STORE) are legal.
- Variables are initialized.
- Classes have superclasses.

Rice's Theorem: Let x be any nontrivial property of the *behavior* of a program. Then x is undecidable.

5.3 The Class Loader

What do Java class loaders do? Two main functions:

1. *Find the bytecode requested by the VM.* When the VM is in need of a certain segment of bytecode, the class loader either loads it from disk, fetches it from the Web server, or creates it on its own.

2. *Manage namespaces.* Class loaders define how namespaces from different classes are related.

There are two essential properties that you should know about class loaders. First, class loaders are arranged hierarchically. At the top of the hierarchy is the "primordial" class loader and the remaining class loaders are class loader "objects." The key difference here is that class loader objects are initially untrusted objects as far as the VM is concerned, and need to be passed through the bytecode verifier.

Second, as we touched upon above, each class loader has its own namespace (one per codebase). Namespaces are unique names of classes loaded by a particular class loader. This is often a point of confusion, as problems with managing namespaces have led to ambiguity problems.

5.4 The Security Manager

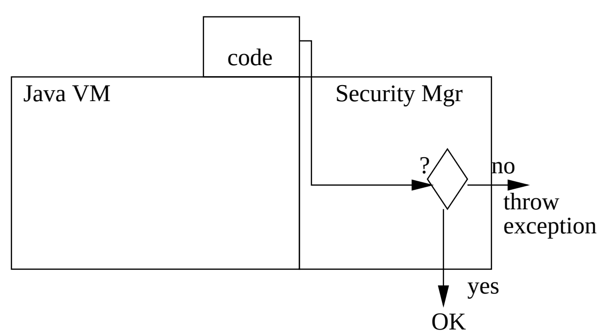

Figure 4: The security manager is invoked by the API when a Java program makes a call to a potentially dangerous operation. If the security manager denies the operation, it throws the security exception. Otherwise, the API performs the operation.

The security manager is the third and final piece to Java's security puzzle. Its job is simple: allow trusted Java code to do whatever it wants; prevent untrusted code from causing harm. The security manager is a Java object that performs run-time checks on potentially dangerous methods. For untrusted applets, the security manager does the following:

- Prevents installation of new class loaders.

- Protects threads from one another.

- Controls access to system resources.
- Controls access to other applications.
- Controls network socket operations.
- Controls the ability to shut down VM.

6 Security Policy

When Java was first created back in 1995 (in computer time 7 years is a LONG time), Java employed a simple, black and white security policy. Since then, security needs have increased and Java's security policy has become more complex.

1. **JDK 1.0:** Code was either trusted or untrusted. If the classes were built-in, code was trusted and allowed to run. If the classes were downloaded from elsewhere, it was untrusted code, subject to the restrictions of the sandbox.
2. **JDK 1.1:** JDK 1.1 is very similar to its predecessor, with the addition of code signing. Code can also be trusted if it is digitally signed and deemed trustworthy by the user.
3. **JDK 2.0:** JDK 2.0, like the versions that preceded it, trusts all built-in classes. The key difference here is that some digitally signed classes can become partially trusted at the user's discretion. Stack inspection is used to determine whether or not a call to a potentially dangerous function is allowed.

7 Bad Applets

There are some problems with mobile code doing malicious things at the local level. Here are a few examples.

- *Noisy Dog Applet:* Contains a thread that never dies.
- *Business Assassin:* Kills the threads of its competitors.
- *Denial of Service:* Hogs CPU time so that other tasks cannot be completed.

Question: Don't modern OS's have something built-in that prevents applets from taking up say X percent of the CPU?
Answer: Not at this point, but that makes sense. Actually, in a way, Java is it's own OS.

8 Source

The content in these lecture notes were written with the help of *Securing Java*, by Gary McGraw and Ed Felten.