6.857 Computer and Network Security	Fall Term, 1997
Lecture 17 : 30 October 1997	
Lecturer: Ron Rivest	*Scribe: Alex Hartemink*

1 Preliminaries

1.1 Handouts

- Graded Problem Set 5 returned
- Solutions to Problem Set 5 handed out
- Midterm Examination distributed

1.2 Comments

The midterm is to be turned in at the end of class Thursday, 6 November 1997. References to class notes from this year may be made, and textbooks which were recommended as class reading may also be consulted. No other references are permitted. In particular, no discussion of any problem with another person is permitted.

This lecture will be abbreviated so that students can attend the 3:30 lecture being given by James Gosling, one of the developers of Java. Attendance is encouraged, though not required. Class will break at 3:15.

The following book is recommended for further reading on this topic should any student wish to delve deeper: *Java Security*, by McGraw and Felton (Wiley, 1997).

2 Security of Mobile Code

Java was designed (at least partially) with the intention of allowing developers to create extremely portable code, capable of being run on a diversity of operating system platforms and machine architectures. Consequently, Java has become a popular language for the development of Internet applets because it allows code written in

one location to be downloaded to a user's machine and run remotely, independent of the user's operating system and architecture. Such code is called *mobile code*.

Java implements a uniform *Virtual Machine* (VM) interface which is pre-compiled for the specific operating system and architecture on which the VM is being run. Code developers compile their Java programs into a *bytecode* representation which can thereafter be interpreted or compiled by any Java VM. Portability is achieved by the use of this uniform bytecode and VM interface.

The remote execution of mobile code raises a number of security issues, however. Since a particular remote user most likely has not written the mobile code about to be run, how can he or she ensure that it will do what it purports to do? How can the user be convinced that the code will not misbehave in any particular way?

Or is this really just an old question, reconstituted? How does the user know that *any* application is doing what it is supposed to be doing? An overwhelming majority of all code is executed by users who did not write the code – so how is the issue of mobile code accessed by users via their web browsers any different?

For the remainder of this lecture, we will not consider how Java addresses the issue of the security of mobile code (perhaps Gosling will touch on this [he did not]), but instead, we will ask some more general questions: what are the issues one needs to consider in implementing mobile code execution, who worries about the security of mobile code and when, and how are mobile applets different from any other kind of distributed code?

3 Concerns

3.1 User Concerns

At this point in the lecture, we will try to determine possible concerns regarding mobile code. Below we present a number of behaviors of mobile code which users might want to scrutinize, limit, or restrict [the order of presentation is unrelated to the order of importance]:

- The applet could try to upload the contents, or some subset of the contents, of the local hard drive to another node on the Internet.
- The applet could try to delete the contents, or some subset of the contents, of the local hard drive.

- The applet could try to make modifications to the contents of the local hard drive. These include modifying file permissions, encrypting files and holding them hostage, changing secret keys (PGP, RSA, e.g.), or altering the security restrictions imposed on applets.

- The applet could record keystrokes (in an attempt to snoop passwords, for example) and upload them to another node on the Internet.

- The applet could send email from the local machine so that it seems to originate from the current user of the local machine. This could be used to perform some type of masqueraded spam, for example.

- The applet could try to fork off a ridiculous number of processes, thereby clogging the system and slowing all threads (a denial of service attack, of sorts).

- The applet could leverage its position inside a firewall to perform tasks which might otherwise not be permitted by the firewall, thereby violating firewall boundaries.

- The applet could try to access parts of the memory outside of its memory space, thereby spying on other processes.

- The applet could try to reconfigure the browser. Presumably, this would entail modifying some state, either a configuration or cookie file on either the hard drive or in memory.

- The applet could try to directly access some hardware which it should not. Some possibilities include frying the monitor (hopefully, this is not a possibility for any process, trusted or untrusted), dialing the modem (perhaps bypassing a firewall), printing something, damaging the hard drive (locking it, parking it, hitting a resonant frequency, moving the read/write head too far), accessing a serial port, rebooting the machine, packet sniffing on the network interface card, playing a sound file, or perhaps most insidiously, watching or listening to the user through a video camera or microphone (imagine a specious video-conferencing applet which might very easily be granted access to both of these!).

- The applet could try to steal cycles from the processor to work on some auxiliary computation (cracking DES comes to mind). If too many cycles are stolen, this might be classified as a denial of service attack.

- The applet could try to act on behalf of the user, impersonating the user to perform a banking transaction, database access, *etc.*

- The applet could pop up windows spoofing the browser and asking the user to enter his or her password.
- The applet could pop up so many windows that the user becomes unable to use any other application (a denial of service attack).
- The applet could watch the URL's being requested by the user and track his or her movements on the Internet.
- The applet could scan the hard drive to look for particular applications and modify its behavior as a consequence. Microsoft's Windows 95 scanned the user's machine configuration during the registration process – is this a problem?
- The applet could download files to the local hard drive, either for running a service unbeknownst to the user (an FTP archive of pornographic images, for example) or to fill up the user's disk space (a denial of service attack).
- The applet could interfere with other processes, including the Java VM, the Java Security Manager or ClassLoader, or processes belonging to a competitor's application.
- The applet could be modified *en route*. We might therefore want applets to be able to authenticate themselves in order to prevent such a switch.
- The applet could try to load and run other applets or link against libraries which the user desires to be "off limits".

It is difficult to set a security policy for such behaviors because in many cases the behaviors may be desirable. For example, there are certainly legitimate reasons for an applet to access a machine's sound card (to play background music or voice clips), use the video camera or microphone (to enable teleconferencing), save information to the disk (to store a cookie, or other state information), load and run other applets (to complete an installation or perform an automatic update), use a large number of cycles (to compute an exponentiation for an encryption), or reconfigure the user's browser (to patch a security hole in its configuration defaults). So disallowing certain behaviors may not always be the solution.

Moreover, disallowing some behaviors may provide a false level of security if applets can work in conjunction with one another. For example, if one applet has read privileges but cannot write and another has write privileges but cannot read, perhaps the two can work in conjunction to compromise a system's security. For example, the two can set up a covert communication channel and cooperate to foil the security the user believes is in place.

3.2 Developer Concerns

Applet users are not the only agents who have security concerns regarding mobile code; applet developers have security concerns of their own. For example, a number of legal liability issues arise if an applet is widely distributed and has the capacity for serious abuse. How responsible is an applet developer for possible uses of his or her code? Can he or she act irresponsibly without consequence? If not, how much does a developer need to thwart possible abuse?

Another developer concern arises in the intellectual property domain. How does the developer protect his code from reuse or theft? This is particularly relevant in the context of Java, which has the property that its bytecode is generally easily decompiled into source code. How does the developer protect his code and restrict its use to the proper context? Intellectual properties issues frequently arise in the arena of font development, for example.

4 Policy Determination

In order to address security concerns, users typically want to define a security policy for their browsers to implement when downloading and executing mobile code. The following questions remain: who sets policy, how often is it set, and how is it modified and negotiated in the presence of new mobile code which requests greater latitude in order to perform its duties? In particular, it is a nuisance to have to consider each of the above concerns whenever a new bit of mobile code appears. Once again, security butts heads with ease of use.

In closing, in their book *Java Security* (reference above), McGraw and Felton classify security concerns as falling into four general categories (note that these are all user concerns and do not include developer concerns) :

- System modifications
- Privacy violations
- Denial of service
- Antagonisms (from annoyances to intentional harm)

We will turn to the specific question of how Java implements security (and how it fails to do so) in the next lecture.