MOBILE CODE SECURITY

Aviel D. Rubin
AT&T Labs-Research
Daniel E. Geer, Jr.
Certco

Sandboxes, code signing, firewalls, and proof-carrying code are all techniques that address the inherent security risks of mobile code. This survey summarizes the relative merits of each.

Mobile code is a term used to describe general-purpose executables that run in remote locations. The concept is not new. Distributed objects have been an important topic in computer science for years, and several object-based systems are well established (CORBA, for example). What's new and revolutionary about mobile code is that Web browsers now come with the ability to run general-purpose executables. The executables can be written by anyone and execute on any machine that runs a browser. This means that the same code can execute on any platform regardless of the operating system and hardware architecture. (Note that we use the term "mobile code" in the context of a program that runs in one place, such as a Java applet. We are not referring to mobile agents that move from machine to machine, run, and then move again.)

The ability to run general-purpose scripts on any machine on the Internet opens up a world of possibilities for distributed applications. However, such functionality is not without costs. In fact, from a security perspective, there is nothing more dangerous than a global, homogeneous, general-purpose interpreter. The fact that the interpreter is also part of a browser (a large, continuously modified and hence notoriously buggy software package) increases the risks. In the worst case, mobile code interpreters, with their inherent bugs, allow an attacker to run native code that is subject to neither restrictions nor access control on the executing machine.[1] That is, if the protection mechanism on the client side is somehow bypassed, attackers can include malicious machine code in executables and cause it to be executed.

The dominant platform on the Internet is an Intel PC with Windows NT or 95. Windows 95 provides little protection from native code running on a machine. In fact, most users keep all their files on the local disk drive in a way that is completely accessible to manipulation by any program they run. Even on Unix and NT systems, which were designed with security in mind, code executed by a user runs with that user's permissions. This gives the mobile code interpreter, or virtual machine, potential access to system files and network connections.

 1089-7801/98/$10.00 ©1998 IEEE

There are three practical techniques for securing mobile code. The first is to limit the privileges of the executable to a small set of operations; this is the *sandbox* model. The second technique is to obtain assurance that the source of the executable is trusted; this is known as *code signing*. A hybrid approach that combines these two techniques has been implemented in version 1.2 of Sun's Java Development Kit[2] and in Netscape's Communicator.[3]

The third approach is to examine executables as they enter a trusted domain and to decide whether or how to run them on the client based on specific executable properties; this is *firewalling*.

All these approaches are in widespread use at this time. A fourth technique, called *proof-carrying code*,[4] is currently limited to use with assembly language programs written by the developers of the approach. In this technique, mobile code carries with it a proof that it satisfies certain properties.

In the following sections, we look at all these approaches in turn, describing them briefly, along with the trust model that each one assumes.

THE SANDBOX

The idea behind the sandbox is to contain mobile code in such a way that it cannot cause any damage to the executing environment. This usually involves restricting access to the file system and limiting the ability to open network connections. The most widespread implementation of a sandbox is in the Java interpreter inside Internet browsers.[5]

Each implementation of an interpreter attempts to adhere to a security policy. The policy explicitly describes the restrictions that should be placed on remote applets. For example, a policy may state that an applet is allowed to access the machine that delivered it but not the files in the local file system. Sun gives a classification of several execution environments and what their stated security policies are. Assuming that the policy itself is not flawed or inconsistent, then any application that truly implements the policy is said to be secure. The **Security Reference Model for JDK 1.0.2** provides an excellent overview of the fundamental security requirements for the Java environment.

Three main components secure the Java interpreter: the class loader, the verifier, and the security manager. The *class loader* is a special Java object that converts remote bytecodes into data structures representing Java classes. Any class loaded from the network requires an associated class loader that is a subtype of the Classloader class. This means that the only way to add remote classes to a machine's local class hierarchy is via the class loader.

```
Public boolean XXX(Type arg1) {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
                security.checkXXX(arg1);
        }
}
```

Figure 1. Code segment demonstrating how the Java interpreter's security manager works: A public method call invokes the system manager, which determines whether the operation XXX is allowed.

The *verifier* performs static checking on the remote code before it is loaded. It checks that the remote code

- is valid virtual machine code,
- does not overflow or underflow the operand stack,
- does not use registers improperly, and
- does not convert data types illegally.

These checks attempt to verify that remote code cannot forge pointers or access arbitrary memory locations. This is important because if an applet could access memory in an unrestricted fashion, it could run native machine code on the client machine—an ultimate hacker goal and the definition of disaster.

In addition, the class loader creates a name space for the downloaded code and resolves classes against the local name space. Local names are always given priority, so remote classes cannot overwrite local names. Without this restriction, an applet could redefine the class loader itself.

In the JDK 1.0 and 1.1 sandbox implementations, local classes are unrestricted but remote classes are passed along to the next verification stage, the *security manager*, which is needed to provide flexible access to potentially dangerous system resources. The Classloader classifies operations as safe or potentially harmful. Safe operations are always allowed, but potentially harmful ones cause an exception and defer a decision to the security manager. In effect, the security manager classes represent a security policy for remote applets.

Figure 1 shows how the security manager is invoked when a caller attempts to execute a method that is restricted by the security policy. When a public method is called, the security manager checks to see if such a call is allowed. It consults the policy module for this. If the call is not allowed, the

security manager throws a security exception. If it is allowed, then the security manager calls a private method, which actually performs the operation. Thus, a system administrator or browser developer can control an applet's access to resources by changing the Security Manager.

The biggest problem with the Java sandbox is that any error in any security component can lead to a violation of the security policy. The risks are exacerbated by the complexity of the interaction between components. For example, if the class loader has incorrectly identified a class as local, the security manager might not apply the right verifications.

Running native machine code on the client— the ultimate hacker goal.

There have been repeated examples of shortcomings in the Netscape Navigator and Internet Explorer interpreters. Two types of applets cause most of the problems. *Attack applets* try to exploit software bugs in the client's virtual machine; they have been shown to successfully break the type safety of JDK 1.0 and to cause buffer overflows in Hotjava.[1] These are the most dangerous. *Malicious applets* are designed to monopolize resources, and cause inconvenience rather than actual loss.[1]

Something is said to be *trusted* if it is believed that it will behave correctly. Java does not involve trust except as a function of the design of the sandbox; it does not address matters of trust in the distant author of the applet. The trust model is therefore that the sandbox is trustworthy in its design and implementation but mobile code is universally untrustworthy.

CODE SIGNING

In code-signing, the client manages a list of entities that it trusts. When a mobile executable is received, the client verifies that it was signed by an entity on this list. If so, then it is run, most often with all of the user's privileges.

Microsoft's Authenticode system for ActiveX is an example implementation. If a user trusts the signer of the ActiveX content, then the content runs with full privileges. Otherwise, it does not run at all. Unfortunately, there is a class of attacks that render ActiveX useless. If an intruder can change the policy on a user's machine, usually stored in a user file, the intruder can then enable the acceptance of all ActiveX content. In fact, a legitimate ActiveX program can easily open the door for future illegitimate traffic, because once such a program is run, it has complete access to all of the user's files. Such attacks have been demonstrated in practice.[6]

There are more problems with signed code. For example, malicious code can plant all manner of delayed attacks. Later, when problems occur, there is no way to tie them back to a given ActiveX control run at some point in the past.

The trust model for code-signing assumes that it is possible to distinguish trustworthy from untrustworthy authors of mobile code and that trustworthy authors are incorruptible.

HYBRID: SANDBOXES AND SIGNATURES

A hybrid scheme attempts to merge the benefits of the sandbox model with code signing.

In the JDK 1.1, a digitally signed applet is treated as trusted local code if the signature key is recognized as trusted by the client system that receives it. That is, the client downloads an applet and then consults a policy table for every signed applet to determine if the signer is trusted. If so, the class loader tags the applet as local, thus giving it access to all system resources. This allows trusted applets to access the file system and to establish network connections, thus enabling many distributed applications that are otherwise restricted by the sandbox. However, it introduces the same security problems inherent in the ActiveX code-signing approach.

The JDK 1.1 approach has limited flexibility. Applets are still either totally trusted or severely limited in functionality. JDK 1.2 introduces a flexible approach that subjects all classes—whether local, remote, signed, or unsigned—to access control decisions.[7] A security policy defines the access each piece of code has to resources on the client. In addition, a security mechanism provides an extensible architecture whereby, for example, signed code can run with different privileges based on the key that is used. Thus, users can fine-tune their functionality-to-security tradeoff to suit their needs.

The trust model for current hybrid approaches is that all code is untrustworthy except for code from a trustworthy supplier who, once identified, is incorruptible.

FIREWALLING

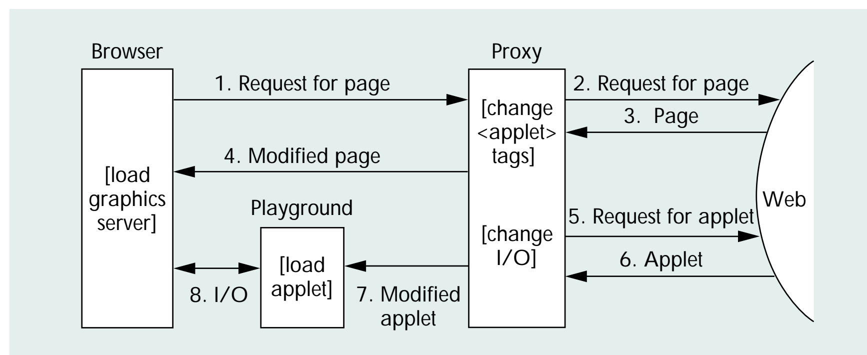

Figure 2. The playground architecture separates Java classes that prescribe graphics actions from those prescribing all other actions. The former are loaded on the client machine; the latter are loaded on a sacrificial playground machine.

The firewalling approach to mobile code security involves selectively choosing whether or not to run a program at the very point where it enters the client domain. For example, if an organization is running a firewall or Web proxy, it may be useful to try to identify Java applets, examine them, and decide whether or not to serve them to the client. Research shows that it may not always be easy to block unwanted applets while allowing other applets—say, those from the local domain—to run.[8] The firewalling approach assumes that applets can somehow be identified.

Several commercial ventures have been established that use the firewalling approach. For example, Finjan Software and Security 7 have several products that attempt to identify applets and then examine them for security properties. Only applets that are deemed safe are allowed to run. Unfortunately, both of these companies use proprietary techniques, so the mechanisms by which they operate are not known. This approach is fundamentally limited, however, by the halting problem,[9] which states that there is no general-purpose algorithm that can determine the behavior of an arbitrary program.

Another approach is taken by Malkhi et al.[10] (developed independently and marketed by Digitivity Inc.) where Java applets are divided into graphics actions and all other actions. The former run on the client machine; the latter run on a sacrificial playground machine.

Figure 2 shows how the playground works. When a browser requests a Web page, the request is sent to a proxy (step 1). The proxy forwards the request to the end server (step 2) and receives the requested page (step 3). As the page is received, the proxy parses it to identify all <applet> tags and, for each <applet> tag so identified, replaces the named applet with the name of a trusted graphics server applet stored locally to the browser. The proxy sends this modified page back to the browser (step 4), which then loads the graphics server applet. For each <applet> tag the proxy identified, the proxy retrieves the named applet (steps 5 and 6) and modifies its bytecode to use the graphics server in the requesting browser for all input and output. The proxy forwards the modified applet to the playground (step 7), where it is executed using the graphics server in the browser as an I/O terminal (step 8).

The playground architecture entrusts the small graphics package because it is easy to analyze and well enough understood to trust, while the more dangerous and untrustworthy mobile code has no access to meaningful resources. Note that this approach requires bytecode modification and therefore cannot be used in conjunction with the usual approach to code signing.

PROOF-CARRYING CODE

Proof-carrying code (PCC) is a technique for statically checking code to make sure that it does not violate some safety policies.[4] For some programs, it is possible to construct a proof that they do not contain any buffer overflows. This has applications to secure mobile code in that many of the safety properties required to achieve assurance for downloadable executables can be achieved with PCC. However, there are properties related to information flow and confidentiality that can never be proved in this way.

PCC is an active area of research so its trust model may change. At present, the design and implementation of the verifier are considered trustworthy but mobile code is universally untrustworthy.

CONCLUSIONS

Each of these techniques offers something different, and the best approach is probably a combination of security mechanisms. The sandbox and code-signing approaches are already being hybridized. Combining these with firewalling techniques such as the playground gives an extra layer of security. The PCC approach is not yet ready for prime time, but the ability to prove safety proper-

URLs for this article

Authenticode • www.microsoft.com/workshop/security/authcode/signing.htm

Digitivity, Inc. • www.digitivity.com/

Finjan Software • www.finjan.com/

Security 7 • www.security7.com/

Security Reference Model for JDK 1.0.2 • www.javasoft.com/security/SRM.html

ties of code is an important element in the arsenal available for securing mobile code.

None of the techniques can do much to protect users from social engineering attacks, where a user is somehow fooled into revealing something they shouldn't reveal. For example, JavaScript can be employed in a way that fools a user into revealing passwords to a remote server. Java applets could be used to do this as well, even under the strictest security policy. User education is the only way to combat mobile code attacks that are based on social engineering. ■

ACKNOWLEDGMENTS

We thank Steve Bellovin, Dahlia Malkhi, and Gary McGraw for helpful comments.

REFERENCES

1. G. McGraw and E. Felten, *Java Security: Hostile Applets, Holes and Antidotes*, John Wiley & Sons, New York, 1997.
2. L. Gong et al., "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2," *Proc. Usenix Symp. Internet Technologies and Systems*, Usenix Assn., Berkeley, Calif., 1997, pp. 102-112.
3. D.S. Wallach et al., "Extensible Security Architectures for Java," *Proc. ACM 16th Symp. Operating Systems Principles*, ACM Press, New York, 1997, pp. 116-128.
4. G.C. Necula and P. Lee, "Safe Kernel Extensions Without Run-Time Checking," *Proc. Second Symp. Operating Systems Design and Implementation*, Usenix Assn., Berkeley, Calif., 1996, pp. 229-243.
5. J.S. Fritzinger and M. Mueller, " Java Security," white paper, Sun Microsystems, Palo Alto, Calif., 1996; available at http://java.sun.com:81/security/whitepaper.txt.
6. P.G. Neumann, *Computer Related Risks*, Addison Wesley, Reading, Mass., 1995.
7. L. Gong and R. Schemers, "Implementing Protection Domains in the Java Development Kit 1.2," *Proc. Internet Society Symp. Network and Distributed System Security*, 1998; available online at http://www.javasoft.com:81/people/gong/papers/pubs98.html
8. D. Martin, S. Rajagopalan, and A.D. Rubin, "Blocking Java Applets at the Firewall," *Proc. Internet Society Symp. Network and Distributed System Security*, 1997; available online at http://cs-www.bu.edu/students/grads/dm/.
9. M.E. Davis and E.J. Weyuker, *Computability, Complexity, and Languages*, Academic Press, New York, 1983.
10. D. Malkhi, M.K. Reiter, and A.D. Rubin, "Secure Execution of Java Applets Using a Remote Playground," *Proc. IEEE Computer Society Symp. Research in Security and Privacy*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 40-51.

Aviel D. Rubin is a senior technical staff member at AT&T Labs–Research and an adjunct professor of computer science at New York University. He is coauthor of the *Web Security Sourcebook*. Rubin holds a BS (1989), MSE (1991), and PhD (1994) in computer science and engineering from the University of Michigan in Ann Arbor. He served as program chair for the 1998 Usenix security conference and is program chair for the 1999 Usenix technical conference.

Daniel E. Geer, Jr., is vice president of Certco, LLC, a market leader in digital certification for electronic commerce. He is coauthor of the *Web Security Sourcebook*.

Readers may contact Avi Rubin at rubin@research.att.com or Dan Geer at geer@world.std.com.