Mobile Code Security

Aviel D. Rubin
AT&T Labs—Research,
Florham Park, New Jersey, USA
rubin@research.att.com

Daniel E. Geer
Certco
New York, NY USA
geer@world.std.com

Abstract

Mobile code is an exciting new technology. By its very nature, however, it is fraught with inherent security risks. In the paper, we give an overview of some of the techniques for securing mobile code environments that have been suggested and deployed. We examine the sandbox approach, code signing, hybrid approaches, firewalling techniques and proof carrying code. The relative merits of each approach are presented.

Keywords: Security, Mobile code, Java, ActiveX, Code signing

1 Introduction

Mobile code is the term used to describe general-purpose executables that run in remote locations. The concept is not new; people have been discussing distributed objects for years, and several object-based systems (e.g. Corba [12]) have been in place for some time. What's new and revolutionary about the current uses of mobile code is that web browsers now come with the ability to execute general-purpose executables. These programs can be written by anyone and execute on any machine that runs a browser. That is, the same code executes on any platform regardless of the operating system and hardware architecture. In this paper, 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 refering 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 its 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, the existence of mobile code interpreters with their inherent bugs, allows an attacker to run native code that is subject to neither restrictions nor access control on the executing machine [9]. That is, if the protection mechanism on the client side is somehow bypassed, attackers can include malicious machine code and cause the code 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 the machine. In fact, most users keep all of their files on the local disk drive such that they are 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, and can thus manipulate files, create network connections, etc. This is true because the mobile code interpreter, or virtual machine, runs with the user's permissions.

There are three practical techniques for securing mobile code. The first method is to limit the privileges of the executable to a small set of operations; this is known as the sandbox model. The

second technique is to obtain assurance that the source of the executable is trusted; this is the code signing approach. (A hybrid approach that combines these two techniques has been implemented in JDK 1.2 [4] and Netscape's Communicator [13]. The final approach to securing clients from mobile code is to examine executables as they enter a trusted domain and make a decision about whether or how to run them on the client based on specific properties of the executables; this is the firewalling approach. Besides these currently-used approaches, there is growing interest in a fourth technique called proof carrying code [10], where mobile programs carry with them a proof that certain properties are satisfied. As of this writing, this technique is limited to use with assembly language programs written by the developers of this approach.

In the following sections, we look at all four approaches in turn. We describe the approach and the trust model that it assumes.

2 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 [3].

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 that the applet may not access 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 that policy is said to be secure. An excellent reference is the Security Reference Model for the JDK 1.0.2 [http://www.javasoft.com/security/SRM.html].

The main components for securing the Java interpreter are the Classloader, the Verifier and the Security Manager. The Classloader is a special Java object that is responsible for converting remote bytecodes into data structures representing Java classes. Any class that is loaded from the network requires an associated Classloader that is a subtype of the Classloader class. This means that the only way for remote classes to be added to the local class hierarchy on a machine is via the Classloader.

The Verifier does 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
- does not covert 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, the definition of disaster.

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

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

Figure 1: Code segment to demonstrate how the security manager works. The security manager is invoked to determine whether or not to allow the operation XXX.

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.

The Security Manager is needed to provide flexible access to potentially dangerous system resources. Operations are classified as potentially harmful or safe. Safe operations are always allowed. However, the potentially harmful ones cause an exception and defer the decision to the security manger. 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 it is not allowed, a security exception is thrown. If it is allowed, then a private method is called. The private method is the one that actually performs the operation. A system administrator, or browser developer, can thus control the access of applets 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 a class is classified incorrectly as local, the Security Manager may not apply the right verifications. There have been repeated examples of shortcomings in the interpreters in Netscape Navigator and Internet Explorer. There are two types of applets to worry about, attack applets [2] and malicious applets [6]. Attack applets attempt to exploit software bugs in the virtual machine on the client. They have been shown to successfully break the type safety of JDK 1.0 and to cause buffer overflows in Hotjava. These are the most dangerous. Malicious applets are designed to monopolize resources, and cause inconvenience rather than actual loss.

It should be noted that Java does not involve trust except inasmuch as it involves trust in the design of the sandbox, rather than trust in the distant author of the applet. The trust model is that the design and implementation of the sandbox is trustworthy but mobile code is universally untrustworthy.

3 Code signing

In the code signing approach to securing mobile code, 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. An example implementation of code signing is Microsoft's Authenticode system for ActiveX.

If a user trusts the signer of the ActiveX content, then it 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, he/she can 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 [11].

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 is that it is possible to distinguish trustworthy authors of mobile code from untrustworthy and that trustworthy authors are incorruptible.

4 Hybrid approach: sandboxes and signatures

A hybrid scheme attempts to merge the benefits of the sandbox model with code signing. In the Java Development Kit (JDK) 1.1, a digitally signed applet is treated as trusted, local code if the signature key is recognized as trusted by the end system that receives the applet. 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 classloader tags the applet as local, thus giving it access to all system resources. This allows trusted applets access to the file system and to establish network connections and enables many distributed applications that are not possible because of the restrictions of the sandbox. However, it introduces the same security problems inherent in the ActiveX code signing approach.

The flexibility of the JDK 1.1 approach is limited. Applets are still either totally trusted or severely limited in functionality. A new, flexible security policy is introduced in the JDK 1.2 [5]. All classes, whether local, remote, signed or unsigned are subjected to access control decisions. Based on a policy, each piece of code has particular access to resources on the client. Besides access to resources, the 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 today is that all code is untrustworthy except for that which is from a trustworthy supplier who, once identified, is incorruptible.

5 Firewalling

The firewalling approach to securing mobile code 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 ones from the local domain, to run [8]. The firewalling approach assumes that somehow applets can 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 the applets for security properties. Only applets that are deemed safe are allowed to run. Unfortunately, both of these companies use proprietary techniques so the mechanisms they use are not known. This approach is fundamentally limited by the halting problem [1] 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. [7] (developed independently and marketed by

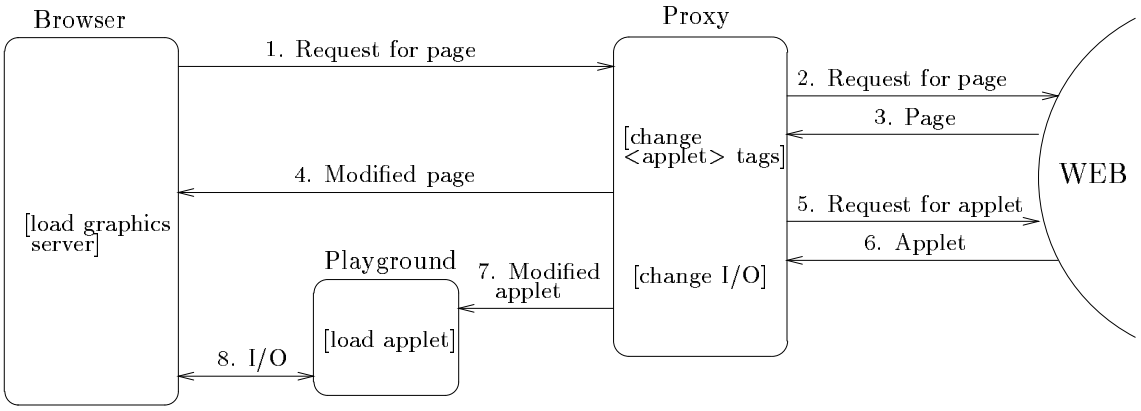

Figure 2: The playground architecture

Digitivity Inc [http://www.digitivity.com]) where Java applets are divided into graphics actions, run on the client machine, and all other actions, which are run on a sacrificial playground machine.

The playground works as follows (see Figure 2). 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 on the returning page, and for each `<applet>` tag so identified, the proxy replaces the named applet with the name of a trusted *graphics server* applet stored locally to the browser. The proxy then sends this modified page back to the browser (step 4), which loads the graphics server applet upon receiving the page. For each `<applet>` tag the proxy identified, the proxy retrieves the named applet (steps 5–6) 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 trust model is that the small graphics package is easy to analyze and well understood enough to trust while the more dangerous and untrustworthy mobile code has no access to meaningful resources. Note that this approach cannot be used in conjunction with the usual approach to code signing, as modification of the bytecodes is required.

6 Proof carrying code

Proof carrying code (PCC) [10] is a technique for statically checking code to make sure that it does not violate some safety policies. 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 security properties required to achieve assurance for downloadable executables are safety properties that can be checked. However, there are properties related to information flow and confidentiality that can never be achieved with proof carrying code. This is an active area of research so its trust model may change. At present, the trust model is that the design and implementation of the verifier are trustworthy but mobile code is universally untrustworthy.

7 Comparison of the techniques

So, which is the best technique for security mobile code? This is not really a fair question. Each of the 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 proof carrying code approach is not ready for prime time yet, but the ability to prove safety properties of code is an important element in the arsenal available to us for securing mobile code.

None of the techniques presented here can do much to protect users from social engineering attacks. For example, Javascript can be utilized to fool 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.

8 Conclusions

While mobile code in ubiquitous browsers (the universal client) presents an exciting new computational platform, it represents a challenge to the security community. We have presented several approaches here to protecting clients from potentially malicious mobile code. The first technique, called sandboxing, involves utilizing the language type restrictions of Java to limit the access of classes to system resources. The second approach, code signing, involves verifying digital signatures to obtain assurance about the identity of the author of a piece of code. Efforts are underway to combine these two approaches.

A third technique, called firewalling, involves intercepting mobile code at the firewall. Several companies are trying to screen mobile code at this point before allowing it to reach the end client. An alternative and more promising approach is to divert mobile code to a sacrificial machine, with only the user interface code running on the actual client, where valuable resources exist. Finally, proof carrying code represents a new approach to proving safety properties for mobile code, which has important implications for security.

Acknowledgements

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

References

[1] M. E. Davis and E. J. Weyuker. *Computability, Complexity, and Languages.* Academic Press, Inc, 1983.

[2] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From hotjava to netscape and beyond. *1996 IEEE Symposium on Security and Privacy*, pages 190–200, 1996. http://www.cs.princeton.edu/ ddean/java/.

[3] J. Steven Fritzinger and Marianne Mueller. Java security. *Sun Microsystems white paper*, 1996.

[4] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. *In Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 102–112, December 1997.

[5] Li Gong and Roland Schemers. Implementing protection domains in the java development kit 1.2. *Proc. Internet Society Symposium on Network and Distributed System Security*, March 1998.

[6] Mark Ladue. Hostile applets home page, 1996. http://www.rstcorp.com/hostile-applets.

[7] D. Malkhi, M. K. Reiter, and A. D. Rubin. Secure execution of java applets using a remote playground. *Proceedings of the 1998 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 40–51, May 1998.

[8] David Martin, S. Rajagopalan, and Aviel D. Rubin. Blocking java applets at the firewall. *Proc. Internet Society Symposium on Network and Distributed System Security*, pages 16–26, 1997.

[9] G. McGraw and E. Felten. *JAVA Security: Hostile Applets, Holes and Antidotes*. John Wiley & Sons, 1997.

[10] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. *Second Symposium on Operating Systems Design and Implementation*, pages 229–243, October 1996.

[11] P. G. Neumann. *Computer Related Risks*. Addison Wesley, 1995.

[12] R. Otte, P. Patrick, and M. Roy. *Understanding Corba*. Prentice Hall, October 1995.

[13] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. *16th Symposium on Operating Systems Principles*, pages 116–128, 1997.