Security Techniques for Mobile Code

Nathan Macrides

Security Techniques for Mobile Code
SANS Security Essentials (GSEC) Practical Assignment Version 1.4
Nathan Macrides
July 11, 2002

1. Abstract

From a security point of view mobile code entities extend the potential of (stationary) distributed systems through the possibility of programs being executed on computers that are often not maintained by the employer of that program. Here two parties are involved in running a program, and thus guarantees have to be given that one party will not harm the other. This paper discusses the various techniques and trust models needed to enforce a level of security that prevents malicious mobile code from infiltrating and running on an unsuspecting users system.

2. Introduction

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). Mobile code is revolutionary in that Web browsers come with the ability to run general-purpose executables. The beauty of this is that code can be written once and run anywhere on any hardware or operating system provided they have a suitable browser. The ability to run general-purpose code on any machine on the Internet opens a new world for distributed applications. However, such potential is implemented at a great cost, especially from the perspective of security where there is nothing more dangerous than a global, homogenous, general-purpose interpreter. The implementation of the interpreter as part of a browser, a large, continuously modified and hence notoriously buggy software package increases the risks. Especially when you consider that Internet Explorer is a fundamental part of the Windows family of Operating systems. 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. Consequently, by somehow bypassing any protection mechanisms in place on the client side, 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/2000 or 95/98/ME. Windows 9X 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. 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 was implemented in version 1.2 of Sun's Java Development Kit and in Netscape's Communicator. 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 fire-walling. All these approaches are in widespread use at this time. A fourth technique, called proof-carrying code, 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.

3. 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. Usually this involves restricting file system access and limiting network connectivity. The Java Virtual Machine, Suns Java interpreter is the most widespread sandbox implementation, and is found inside Internet browsers. [1]

Sun has several security policies, and gives classifications of their execution environments. Applications that implement these policies correctly are said to be secure. Obviously, this is dependent on the policy not being flawed or inconsistent. An excellent overview of the fundamental security requirements of the Java environment is provided by the JDK (Java Development Kit) 1.0.2 Security Reference Model. [2]

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. Classes loaded from the network require an associated class loader that is they are required to be a subtype of Classloader class. Therefore to add a remote class to a machines local class hierarchy the class loader must be used. 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

[1] M. Erdos, B. Hartman and M. Mueller, "Security Reference Model for the Java Developer's Kit 1.0.2," white paper, Sun Microsystems, Palo Alto, Calif., 1996; available at http://java.sun.com/security/SRM.html.

[2] ibid.

remote classes cannot overwrite local names. Without this restriction, an applet could redefine the class loader itself.

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. [3]

The Security Manager has been used since JDK 1.0 and in all sandbox implementations. This provides downloaded classes flexible access to potentially dangerous system resources, unlike local classes, which are unrestricted.

Operations are classified as safe or potentially harmful by the Class loader. 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.

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

}
```

Figure 1. Code s egment demonstrating how the Java interpreter's security manager w orks: A public method call invokes the s ystem manager, which determines whether th e operation XXX is allowed.

Figure 1 shows how the security manager is invoked when a caller attempts to execute a method that is restricted by the security policy. A call to a public method, results in the security manager checking to ensure such a method is allowed to run. 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. [4]

[3] Joseph A. Bank, "Java Security", MIT., 1995, available at http://www-swiss.ai.mit.edu/~jbank/javapaper/javapaper.html

[4] Bank, op. cit.

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

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 the JDK's since version 1.0 and to cause buffer overflows in Hotjava.[5] These are the most dangerous. Malicious applets are designed to monopolize resources, and cause inconvenience rather than actual loss.[6]

Examples of such shortcoming in the Microsoft Internet Explorer browser have been discovered recently. Microsoft release a security bulletin on March 4, 2002, outlining vulnerabilities in the Microsoft Java VM (Virtual Machine) that affected how Java requests for proxy resources are handled. A malicious Java applet could exploit this flaw to re-direct web traffic once it has left the proxy server to a destination of the attacker's choice. [7]

The attacker could then:

- Forward the information on to the intended destination, giving the appearance that the session was behaving normally.
- Send his own malicious response, making it seem to come from the intended destination, or could discard the session information, creating the impression of a denial of service
- The attacker could capture and save the user's session information. This could enable him to execute a replay attack or to search for sensitive information such as user names or passwords. [8]

The second vulnerability is a problem with the security checks on casting operations (allows casting of data types by the Java language) within the VM. A vulnerability results because it is possible for an attacker to exploit this flaw and use it to execute code outside of the sandbox. This code would execute as in the context of the user, and would only be limited by those constraints which govern the user.[9]

---

[5] D. Dean, E. W. Felten and D. Wallach, "Java Security: From HotJava to Netscape and Beyond", Proceedings of 1996 IEEE Symposium on Security and Privacy (Oakland, California), May 1996; available at http://www.cs.princeton.edu/sip/pub/secure96.php3

[6] ibid.

[7] Microsoft Corp, "Microsoft Security Bulletin MS02-013. 04 March 2002 Cumulative VM Update", Microsoft TechNet, March 18, 2002; available at http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS02-013.asp

[8] ibid.

[9] Microsoft Corp, "Microsoft Security Bulletin MS02-013. 04 March 2002 Cumulative VM Update", op cit.

More examples of attacks caused by malicious code are:

Integrity Attacks

- Deletion/Modification of files.
- Modification of memory currently in use.
- Killing processes/threads.

DOS Attacks

- Allocating large amounts of memory.
- Creating thousands of windows.
- Creating high priority processes/threads.
- Remote DOS attacks on machines.

Disclosure Attacks

- Mailing information about your machine (i.e. /etc/passwd).
- Sending personal or company files to an adversary or competitor over the network.

Annoyance Attacks

- Displaying obscene pictures on your screen.
- Playing unwanted sounds over your computer. [10]

Overall 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.

4. CODE SIGNING

In code signing, the client manages a list of entities that it trusts. The client verifies that the mobile code it has received was signed by an entity on the list. Once verified the code is executed, usually with the full rights of the user executing it. Microsoft uses a system called Authenticode to determine if the ActiveX content is trusted and should be run with full privileges, or not at all. [11]

There is a problem with this system, which can render ActiveX useless. A malicious ActiveX control could modify the policy; usually this is stored in a text file on a user's machine. The new policy can then enable the acceptance of all ActiveX content. In fact legitimate ActiveX content has allowed malicious code to run because it has access to the entire system. Such attacks have been demonstrated.[12]

---

[10] Bank, op. cit.

[11] Microsoft Corp, "MSDN – Creating, Viewing, and Managing Certificates" , MSDN Library May 2002, available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/security/Security/creating_viewing_and_managing_certificates.asp

[12] D. Hopwood, "A Comparison between Java and ActiveX Security", Network Security; available at http://www.users.zetnet.co.uk/hopwood/papers/compsec97.html

More problems with signed code exist. Examples include delayed attacks from installed ActiveX content. The attack cannot be traced back to an Active control run in the past. Code signing works on a trust model that assumes that trustworthy and untrustworthy authors of mobile code can be distinguished and those authors are incorruptible.

5. HYBRID: SANBOXES & 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 upon downloading an applet; the client consults a policy table of all signed applets to determine if the signer is trusted. The class loader then tags the applet as local if the applet is trusted, therefore giving the applet access to all system resources. Functionality such as file system access and network connectivity, that are usually restricted by the sandbox are enabled. Consequently the same security issues inherent in ActiveX code signing are introduced. [13]

Because of the limitations of the JDK 1.1 approach JDK 1.2 introduced a flexible approach that subjects all classes local, signed, remote or unsigned to access control decisions. Access to client resources is defined through a security policy. Such a security mechanism allows for an extensible architecture. The result is an environment that users can fine-tune to meet their functionality-to-security trade-off and allows signed code to run with different privileges based on the key that is used. [14]

One such example of this is JAR signing. The JAR file format is a convention for storing Java classes and other resources that may be signed. The format allows the contained files to be signed by different principals. This allows different people to sign different classes inside the JAR file. Consequently an attacker could add unsigned classes to a JAR, and use them to exploit the signed classes in an effort to break security. A successful attack of this kind is dependent on the care taken by the class writer to ensure that his/her code cannot be exploited. The problem therefore lies in ensuring that if a large number of signed controls are produced, that they are bug free, an almost impossible task. [15]

As discussed in section 4 (Code Signing), Microsoft uses a system called Authenticode to determine if the ActiveX content is trusted and should be run with full privileges, or not at all.[16] A CAB (short for cabinet file) file containing

[13] M. Erdos, B. Hartman and M. Mueller, op cit.

[14] M. Erdos, B. Hartman and M. Mueller, op cit.

[15] D. Hopwood, op cit.

[16] Microsoft Corp, "MSDN – Creating, Viewing, and Managing Certificates", op cit.

Java classes can be signed using Authenticode. CAB signing is similar to JAR signing, in that the CAB is a convention for storing either Java classes or ActiveX controls. While CAB files are usually signed by both trusted sources, test keys can be generated by virtually anyone with the right tools. A test certificate can then be generated from the key and used to sign the CAB. Once signed with a test certificate and the CAB downloaded and accepted by the client, the applet has full access to all client resources.[17] Being a generated test certificate client users are told that the certificate is from an untrustworthy source and prompted to accept or reject the code. Most people are tricked into accepting and so unwittingly and without thinking the malicious code is executed.

The necessary tools for signing a CAB file are available at

http://msdn.microsoft.com/MSDN-FILES/027/000/219/codesign.exe

The Figure 2 shows the steps performed in signing a CAB file with a test certificate using a batch file.

```
@echo off

rem Create a new test key
makecert -sk testkey -n "CN=Nathan says that you must click on YES below." testcert.cer

rem Convert to a software publishers certificate
cert2spc testcert.cer testcert.spc

rem Create the CAB file with the java class files
cabarc n myapplet.cab *.class

rem Sign and timestamp the CAB file
signcode -spc testcert.spc -k testkey myapplet.cab
```

Figure 2. Batch file demonstrating how to sign a CAB file using a test certificate.

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.

6. FIREWALLING

Firewalling takes the approach of selectively choosing whether or not to run a program at the point where it enters the client domain. Organisations running a firewall or Web proxy may try to identify Java applets, analyse them, and decide whether or not to serve them to the client. Research has shown that this it is not always easy to do.[18]

---

[17] Microsoft Corp, "INFO: Steps for Signing a .cab File (Q247257)" , MSDN Library July 2000, available at http://support.microsoft.com/default.aspx?scid=kb;EN-US;q247257

[18] 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://www.cs.bu.edu/~dm/pubs/java-firewalls.pdf.

Finjan Software (http://www.finjan.com) has 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, this company uses proprietary techniques, so the mechanisms by which they operate are not known. This approach is fundamentally limited, however, by the halting problem,[19] which states that there is no general-purpose algorithm that can determine the behaviour of an arbitrary program.

Another approach is taken by Malkhi et al.[20] (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.

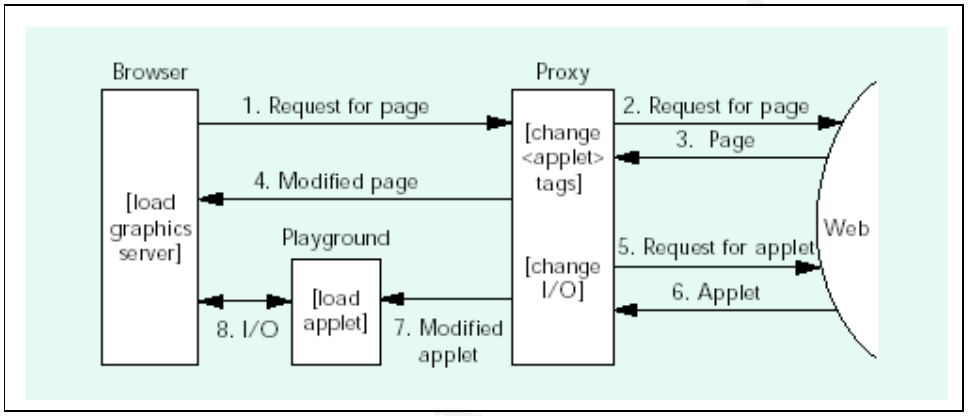

Figure 3. The playground architecture separates Java classes that prescribe graphics actions from those prescribing all other actions. The former are loaded on the client machine; the latter are loaded on a sacrificial playground machine.[21]

Figure 3 shows how the playground works.

Step 1: When a browser requests a Web page, the request is sent to a proxy.

Step 2: The proxy forwards the request to the end server.

Step 3: The requested page is received. 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.

Step 4: The proxy sends this modified page back to the browser.

---

[19] M.E. Davis and E.J. Weyuker, "Computability, Complexity,and Languages", Academic Press, New York, 1983.

[20] 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. Available at http://citeseer.nj.nec.com/cache/papers/cs/14965/http:zSzzSzwww.avirubin.comzSzplayground.pdf/malkhi98secure.pdf

[21] D. Malkhi, M.K. Reiter, and A.D. Rubin, op cit.

Step 5: The browser then loads the graphics server applet.

Step 6: For each <applet> tag the proxy identified, the proxy retrieves the named applet and modifies its bytecode to use the graphics server in the requesting browser for all input and output.

Steps 7 & 8: the proxy forwards the modified applet to the playground. It is the executed using the graphics server in the browser as an I/O terminal. The untrustworthy and dangerous mobile code is run where it has no access to meaningful resources. The small graphics package is trusted by the playground architecture because it is easy to analyse and well enough understood to trust. Due to the need for the playground to modify the bytecode, code signing techniques cannot be used in conjunction with it.[22]

7. PROOF-CARRYING CODE

The technique of Proof Carrying Code (PCC) involves statically checking untrustworthy code to make sure it does not violate some safety policies. Typically, the receiver of the code defines a set of safety rules that guarantee safe behaviour of programs. The developer of the untrustworthy code also constructs a safety proof that adheres to the safety rules. The code is then validated, simply and efficiently proving that it is safe for execution. However, there are properties related to information flow and confidentiality that can never be proved in this way. PCC's trust model may change in the future, because it is still very much in the stages of research and development. It can be said that the verifier's implementation is trustworthy and that mobile code is considered universally untrustworthy.[23]

[22] ibid.

[23] Andrew W. Appel, Edward W. Felten, Zhong Shao, "Scaling Proof-Carrying Code to Production Compilers and Security Policies", Princeton University 2002, available at http://www.cs.princeton.edu/sip/projects/pcc/whitepaper/

8. CONCLUSIONS

All of the techniques discussed in this paper offer different approaches to combating malicious mobile code. However the best approach is probably a combination of security mechanisms. The sandbox and code signing approaches are already hybridized. Combining these with firewalling techniques such as the playground gives an extra layer of security. PCC is still very much in the research and development phase at present. However the ability to prove the safety properties of code is an important weapon in the fight to securing mobile code. Diligence is also needed on the part of systems administrators. They need to be aware of any exploits and bugs that could be exploited by mobile code. Consequently any fixes supplied by the software vendors should be applied as soon as possible to maintain security.

None of these measures can do much to protect users from social engineering attacks. Users can be fooled into revealing something that they shouldn't. Passwords could be revealed with the use of JavaScript or even Java applets and then sent to remote server. The strictest of security policies will not be able to prevent such an attack. Educating users in social engineering attacks based around mobile code is usually the best way to prevent a security breach.

9. REFERENCES

1. M. Erdos, B. Hartman and M. Mueller, "Security Reference Model for the Java Developer's Kit 1.0.2," white paper, Sun Microsystems, Palo Alto, Calif., 1996; available at http://java.sun.com/security/SRM.html.

2. Joseph A. Bank, "Java Security", MIT., 1995, available at http://www-swiss.ai.mit.edu/~jbank/javapaper/javapaper.html

3. D. Dean, E. W. Felten and D. Wallach, "Java Security: From HotJava to Netscape and Beyond", Proceedings of 1996 IEEE Symposium on Security and Privacy (Oakland,California), May 1996; available at http://www.cs.princeton.edu/sip/pub/secure96.php3

4. Microsoft Corp, "Microsoft Security Bulletin MS02-013. 04 March 2002 Cumulative VM Update", Microsoft TechNet, March 18, 2002; available at http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS02-013.asp

5. Microsoft Corp, "MSDN – Creating, Viewing, and Managing Certificates", MSDN Library May 2002; available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/security/Security/creating_viewing_and_managing_certificates.asp

6. D. Hopwood, "A Comparison between Java and ActiveX Security", Network Security; available at http://www.users.zetnet.co.uk/hopwood/papers/compsec97.html

7. Microsoft Corp, "INFO: Steps for Signing a .cab File (Q247257)", MSDN Library July 2000; available at http://support.microsoft.com/default.aspx?scid=kb;EN-US;q247257

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://www.cs.bu.edu/~dm/pubs/java-firewalls.pdf.

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. Available at http://citeseer.nj.nec.com/cache/papers/cs/14965/http:zSzzSzwww.avirubin.comzSzplayground.pdf/malkhi98secure.pdf

11. Andrew W. Appel, Edward W. Felten, Zhong Shao, "Scaling Proof-Carrying Code to Production Compilers and Security Policies", Princeton University 2002, available at http://www.cs.princeton.edu/sip/projects/pcc/whitepaper/