Towards Secure Downloadable Executable Content: The JAVA Paradigm

J. Iliadis[1,2], S. Gritzalis[1,2], and V. Oikonomou[2]

[1] Department of Information & Communication Systems, University of the Aegean, Research Unit, 30 Voulgaroktonou St., Athens GR-11472, GREECE
tel: +30-1-6456688, fax: +30-1-6448428, email: {jiliad, sgritz}@aegean.gr

[2] Department of Informatics, Technological Educational Institute (T.E.I.) of Athens, Ag.Spiridonos St., Aegaleo, GR-12243, GREECE
tel: +30-1-5910974, fax: +30-1-5910975, e-mail: sgritz@acm.org

Abstract. Java is a programming language that conforms to the concept of downloadable, executable content. Java offers a wide range of capabilities to the application programmer, the most important being that a program may be executed remotely, without any modification, on almost any computer regardless of hardware configuration and operating system differences. However, this advantage raises a serious concern : security. When one downloads and executes code from various Internet sources, he is vulnerable to attacks by the code itself. A security scheme must be applied in order to secure the operations of Java programs. In this paper, the Java security scheme is examined and current implementations are evaluated on the basis of their efficiency and flexibility. Finally, proposed enhancements and upcoming extensions to the security model are described.

1. Introduction

The concept of downloadable executable content is not a new one. The explosion in use of the World Wide Web we are currently experiencing coupled with rapid evolution of the JAVA programming language has given insights into downloadable executable content and at the same time raised some important security issues.

JAVA is an object-oriented language that has been developed and distributed by Sun Microsystems. The main advantage of this language over other object-oriented languages is that the JAVA binary code may be executed on a wide variety of systems, without any modification. This is achieved through the *JAVA Virtual Machine* (JVM), a software program that simulates an imaginary virtual machine [Sun, 1997a], [Sun, 1997b]. The virtual machine insulates the application from differences between underlying operating systems and hardware and ensures cross platform compatibility among all implementations of the JAVA platform. This capability has offered JAVA wide acceptance from the community of application programmers, especially those who implement applications to be deployed in the Internet. However, security issues arise when a user downloads and executes a piece of code from a (possibly untrusted) network source. The user's system is vulnerable to any kind of attack, deriving from the executable content. JAVA has been surrounded by a specific security scheme, from the first release. The latter has undergone major alterations with respect to both its content (libraries, capabilities, structure) and its security mechanisms, the past few years. Now, in the new versions, JAVA promises that through its new designed security mechanism, it provides a secure environment for downloadable executable content that under specific circumstances can make use of a system's resources without compromising their availability and integrity.

This paper addresses the security problems associated with the JAVA programming language and provides some solutions for them. Section 2 provides a brief overview of the problems that would arise when running untrusted programs without providing a secure environment are investigated, section 3 takes a closer look at the solution provided by the JAVA security model. Current implementations as well as upcoming extensions are presented in section 4, where their efficiency and flexibility are evaluated, and proposed enhancements are discussed in section 5. Finally in section 6 concluding remarks are presented.

2. General Security Issues

Downloadable, executable content is the idea of downloading data that is actually code to be executed. The execution of downloadable content should be surrounded by a thorough security and safety scheme because downloadable, executable content (in contrast with traditional applications) could derive from untrusted or even unknown sources and as such it could misuse system's resources. The essence of the problem is that downloading and running JAVA code without placing any restrictions in resources availability, can provide a malicious program with the same ability to mischief as a hacker who had gained access to the host machine.

Hostile applets are a kind of Web-embedded JAVA programs that perform such hostile activities against Web users. An important part of creating a secure and safe environment for a program to run in is identifying the assets that we are concerned about and providing the classes of potential attacks that may occur. Therefore any program can do each of the following [Bank, 1995] [Gritzalis, 1991]:

- attack the *Integrity* of the system
- violate the user's *Privacy*
- limit system resources *Availability*
- achieve user's *Annoyance*

An extreme solution to the above mentioned problems would be to completely confine any downloaded executable content within the Web browser that is running it, hence not permitting any usage of the underlying system's resources. Nevertheless, such a solution is not a feasible one since one grants access to system's resources in order to make a program useful. By implication, one has to carefully consider what system resources and to what extent may be made available to a downloaded executable content, from within a browser, without endangering the system's security and at the same time guaranteeing the usefulness of the executable content.

3. JAVA Security

The Java Development Kit (JDK) is the official JAVA implementation from Sun. In JDK1.1 the downloaded code is executed in a customisable "sandbox", in order to protect the users from hostile applets. This model restricts the actions of an applet in a dedicated area of the web browser. Within its "sandbox" the applet may do anything it wants but cannot gain access to the user's file systems, network connections or other resources. The *sandbox* is made up of several systems that range from the applet security manager to the basic features of the JAVA language and JAVA Virtual Machine (JVM).

The JAVA language enhances safety [Sun, 1997c] by eliminating features like pointers and runtime casting to prevent illegal access to memory. In addition, security is provided by the JAVA simulator (interpreter) during the load and verification of JVM code. The JAVA interpreter inside JVM has three main tasks:

- code loading, performed by the class loader
- code verification, performed by the bytecode verifier
- code execution, done by the runtime system.

Applets are loaded from the network by the applet *Class Loader* which receives the bytecode instruction stream and converts it into internal data structures that represent the applet's classes. It then calls the verifier to check the class files, and creates a namespace where places all the applet's classes. A unique namespace exists for each network source, thus preventing untrusted applets from gaining access to more privileged, trusted parts of the system.

The Class Loader invokes the *Bytecode Verifier* before running a newly imported applet. The verifier subjects each applet class to a number of tests: Checks the bytecode to ensure that it does not forge pointers, violate access restrictions or access objects using incorrect type information and performs any other actions needed to prove that the applet will not be allowed to corrupt part of the security mechanism or to replace part of the system with its own code.

The *Security Manager* is an abstract class that enforces the boundaries around the sandbox. Whenever an applet tries to perform an action which could corrupt the local machine or access information (like the above mentioned actions), the JVM first asks the security manager if this action can be performed safely. If the security manager approves the action, the virtual machine will then perform it. Otherwise, the virtual machine raises a security exception and writes an error message to the JAVA console.

JDK1.2 includes a set of new features [Gong, 1997a], [Gong, 1997b] which introduce another modus operandi for the JAVA security system. JDK 1.2 will consist of the following new protection mechanisms: security policy, access permissions, protection domains, access control checking, privileged operation and JAVA class loading and resolution.

The security policy introduced by JDK1.2 is instantiated at JVM startup and may be altered a posteriori via secure methods. If no policies have been defined, a policy that conforms with the original “sandbox” operation is instantiated. Policies may be loaded either from the local file system or even from the Web. The policy comprises of a mapping between properties of the running code (the URL of the code and the code signature) and a set of permissions granted to that code. If a piece of code carries more than one signatures then the permissions it is entitled to are computed as the sum of permission each signature is entitled to. The permissions of an execution thread are calculated as an intersection of the permissions of all the callers of that thread.

There is a complete set of typed and parameterized access permissions contained in the JDK1.2 abstract class java.security.Permission. Furthermore, there are two abstract classes named java.security.PermissionCollection and java.security.Permissions which contain homogeneous and heterogeneous collections of permissions, respectively. One of the most important methods in this class is the "implies" method. a.implies(b)=true means that if one is granted the permission a then it is also granted the permission b.

JDK1.2 introduces the concept of *protection domains* [Gong, 1998]. The latter form the cornerstone of the new security features included in JDK1.2. A protection domain consists of all the objects that correspond to a principal who has been authorised by the system. Permissions may no longer be granted in JDK1.2 to classes but to protection domains.

Every class may belong to one domain only. The JAVA runtime maintains the mappings between classes and domains as well as between domains and permissions. It is possible to prevent the communication between different domains. When such a communication is necessary it may be performed either indirectly through system code, or directly if all the participating domains allow it. One of the future adjustments pertinent to the protection domains is the inclusion and usage of user authentication and delegation information, in order to be able for a piece of code to have different permissions when executed by different principals.

If a thread transverses more than one domain while executing, the permissions it is entitled to are computed based on the principle of least privilege. According to the latter, when the thread enters a domain which possesses fewer permissions then the previous one, it is entitled to these rights, while when it enters a domain which possesses more permissions than the previous one, it is entitled to the permissions of the previous domain.

Until JDK1.1, the code that performed access controls had to know the status of all it's callers; these checks had to be performed by the programmer. JDK1.2 introduces a new class called AccessController which simplifies this process. JDK1.2 uses “lazy evaluation” of the permissions of an object. When an object requests access to a resource, the programmer may call the checkPermisssion method of this class, thus having the system itself perform the access control on behalf of the programmer. For backward compatibility reasons, the SecurityManager usage is still allowed.

There are cases when a piece of code wishes to use the permission it is entitled to itself, although the status of the code’s callers prohibit it. In such a case, the programmer may use the beginPrivileged and endPrivileged methods in the

AccessController class and have his code use the aforementioned permissions. If in such a case the code calls another piece of code, the latter loses the privilege of the former.

The Classloader class has been replaced by the SecureClassloader. The latter can distinguish system classes from others and impose the designed security policy to the latter. In JDK1.2 policy restrictions apply both to applets and applications. In order to impose the security policy to locally installed applications too, a new class has been introduced, called java.security.Main.

As we have already mentioned, it is the purpose of Javasoft to include in following JDK revisions, user authentication information in the security policy. This will provide the means to run the code with different permissions, depending on who's behalf the code is running. User authentication information inclusion and usage has already been tested and performed to a certain degree [Balfanz, 1997].

JDK1.2 and most of the emerging JAVA security frameworks applications depend on the existence of a public key infrastructure. It is an obvious necessity for these frameworks to support the widely accepted standards for Public Key Infrastructures (PKI) and for the security mechanisms that surround them, such as X.509v3 certificates, Secure Socket Layer (SSL), HTTP Secure (HTTPS), Secure MIME (S/MIME) and Secure Lightweight Directory Access Protocol (LDAPS).

4. Security Extensions

The JDK 1.2 includes support for digital signatures (authentication, integrity) and message digests, key management, certificate management and access control. The *JAVA Cryptography Architecture* refers to the framework for accessing and developing cryptographic functionality for the JAVA platform. The official JAVA implementation by Sun includes an implementation of the NIST DSA algorithm, the MD5 and SHA message digest algorithms.

JDK 1.2 provides a security tool, the *javakey*, whose primary use is to generate digital signatures for archive files (JAR files) which enable the packaging of class files. To sign an applet the producer first creates a JAR file and then creates a digital signature based on the contents of the JAR. JDK1.2 also provides a tool called *keytool*, which is used to manage the *keystore*. The latter is a database for private and public keys and their respective certificates. The keystore may resides at the same repository with the policy; it is either stored on a local drive or at a remote repository, accessible via the Web. There is also a new tool available in JDK1.2, called PolicyTool, which is a graphical user interface that one may use in order to generate, import or export a security policy.

Since the ability to encrypt the data prior to being transferred is very important, APIs for data encryption are contained in a *JAVA Cryptography Extension*, as an add-on package to JDK.

Recently, new policy enforcement methods, secure code distribution and JAVA firewall blocking methods have been found and some of them implemented. These are presented in the following paragraphs.

Policy Enforcement Methods

Once the code is authenticated to the system, that is associated with a principal, then the code is subject to the policy defined for that principal. To enforce this policy, three secure methods have been found [Wallach, 1997]:

- *Capabilities:* Unforgeable pointers which can be safely given to user code
- *Extended stack introspection:* Information about the principals can be included in the stack
- *Name space management:* Restricting or changing an applet's namespace

Capabilities

Fundamentally, a capability is an unforgeable pointer to a controlled system resource. To use a capability, a program must have been first explicitly given that capability, either as part of its initialisation or as the result of calling another capability. Every top level class of an applet could be given by the system an array of capabilities (pointers to objects). Then the applet would be able to use whenever it wishes these capabilities. The objects that are referenced by the capabilities may implement and enforce their own, internal security policy, by checking the parameters with which they are called. The JAVA runtime would have to be modified in order to make private all the methods which handle system resources directly or indirectly. It is only the capabilities themselves (the referenced objects) that should have access to these methods.

Capabilities are easy to implement in JAVA, because of the underlying infrastructure. Systems that support "capabilities" have been researched. One of these is the JAVA Electronic Commerce Framework [Goldstein, 1996], which provides a security platform, complimentary to the one that JAVA is using right now, able to implement complex trust relationships between entities.

Extended Stack Introspection

This policy enforcement method has already been implemented in commercial browsers, such as Netscape Communicator 4 and Microsoft Internet Explorer 4 and in the JDK1.2. There are three primitives that must be implemented for this method:

- enablePrivilege(target)
- disablePrivilege(target)
- checkPrivilege(target).

Every system resource must be associated with a target. Before the system allows the use of a system resource, it should perform a checkPrivilege on the specified target. When a thread, associated with a principal, asks for a system resource it should perform an enablePrivilege on the specified target. The security policy will then decide whether that principal is entitled to use this target and the system will or will not grant the privilege to the code. After the thread has finished using this system resource it must execute a disablePrivilege. If the programmer omits the latter, the privilege should be discarded automatically. This can be done by storing the enabled privilege in a hidden and protected field of the method that created it. The checkPrivilege primitive should check the stack and judge whether a privilege must be given or not based on the *least privilege* rule, in order to prevent trusted code from calling untrusted code and passing the latter the privileges of the former.

Name Space Management

It is a method to enforce a security policy by replacing or hiding the classes that are visible by a program. There has to be a mapping between the applets (and their respective signers) and the namespaces that each applet will be able to see. Hiding a class from the namespace means that the applet may not be able at alto use it, while replacing the class with another (possibly a subclass of the latter) provides a means for controlling even more the way that the applet will call and use the methods contained in the aforementioned class.

Let's assume that we created a subclass S of class C and a method M which overrides the respective method of class C. Method M of subclass S may be providing limited use of the resources that method M of class C provides normally to an applet. If we bind the name of S to C, then the applet that will call method M will have at it's disposal a limited, functional set of resources, although it will believe it has gained access to the full set of resources it has asked for.

Secure Code Distribution

Secure code distribution [Zhang, 1997] is one of the security issues that concern the JAVA community. There are various approaches to authentication for secure code distribution. One of them is the signed applets. The JAVA class file format is extensible. It is possible to add new attributes to the latter, without influencing the current structure or workflow of the classloaders. These new attributes can contain signatures for the applet which can be verified, provided that the user possesses the certificate of the entity who signed the applet. After the verification the security manager may grant certain rights to the applet, depending on the trustworthiness of the author or of the person who has signed the applet. The level of trust, and therefore the amount and species of rights that should be granted to the applet, that each user preserves for each applet author or signer can be defined a priori by forming a local security policy, accessible from the user through the browser and protected by all means available (e.g. filesystem rights) to the Operating System. Secure code distribution is already supported by Javasoft's JAR specification. A JAR is an archive encapsulating, with proprietary methods, signed or unsigned classes and other files (e.g. sounds).The signature can be verified by the JDK infrastructure, providing thus a secure authentication means for the distribution of applets.

The success of the JAR secure code distribution scheme heavily depends on the existence of a public key infrastructure, based on widely accepted standards. Code signing provides a way for securely authenticating the source or even the author of the applet to the final user. However, the choice of rights or the choice of "trusted" signers or authors is still a decision that must be taken by the user, prior to executing the applet. One should not exclude the possibility of hostile code, deriving from a source or being signed by someone that a user considers to be "trusted". Furthermore, code signing provides no additional security layer for those applets that are considered to be "untrusted". They are still executed in the limited sandbox environment, and the local system’s safety depends on the existing security infrastructure.

Confining the Use of JAVA in a Network Domain

An organisation may wish to ban all JAVA incoming traffic from the Internet, but wishes also to deploy and use JAVA applets internally, in the domain of the organisation. This may be achieved by blocking the JAVA applets at the firewall [Martin, 1997]. However it is a complex task and up to now no solution has proven to be complete, apart from locking all the browsers' preferences in the organisation’s internal domain in such a way so they will not be able to interpret JAVA. However, the latter would render the browsers incapable of interpreting JAVA while operating in the organisation’s internal domain too.

Blocking JAVA applets at the firewall can be performed by implementing a number of strategies in order to assure that applets do not penetrate the internal network, or if they do, they are not executed at all at the local machines. The first strategy should be the rewriting of the <applet> tag by a proxy, wherever this is found. It could be in an HTTP transmission, or in mail or news messages (HTML enabled mail and news clients exist in the market currently). Rewriting the <applet> tag will cause the client to ignore the existence of the java applet in the message. It should be mentioned though that Javascript may be used in order to create the <applet> tag just before the page is viewed in the browser, rendering the above blocking useless. Another strategy that must be implemented is the blocking of the CAFEBABE signature. Every JAVA class file begins with this 4-byte signature, [Sun, 1997b] so it is possible to block all java classes as soon as they reach the proxy. However, if SSL or JAR files are used, these countermeasures are rendered useless, since the <applet> tag or the CAFEBABE signature will pass undetected, through the proxy. In addition, another strategy that has to be implemented is to set the proxy to reject all files which have an extension ".class". However, if classes arrive at the firewall archived (JAR, Zip) the ".class" extension would not be detected.

If one decides to implement the aforementioned strategies, he should take under consideration the possibility of accidentally blocking useful information, e.g. files whose first 4-byte signature is CAFEBABE but are not JAVA classes or files that contain the "<applet>" string but do not call any JAVA applets (could possibly be a JAVA security paper, in HTML format) or even blocking files that end in ".class" but do not consist of JAVA classes.

5. Proposed Enchancements

The runtime should provide a configurable audit system which would allow system administrators to study the circumstances under which each type of attack has been occurred. As a minimum, files read and written from the local file system should be logged, along with network usage.

Limiting system resources (degradation of service) is still an open issue. The JAVA community is not so concerned about it, though it is one of the easiest to exploit

JAVA security leaks. The JDK should limit the level to which an applet may use system resources.

The security user interfaces we have witnessed so far demand too frequently from the user to take security-related decisions, based on security and system resources information he is provided with upon asked. The number of questions towards the user should be limited as much as possible. This can be achieved by having the site administrator define a system-wide policy and leave the user with a few questions to be asked. Asking the user too many questions sometimes presents an impact quite opposite than the one we wish. The user starts pressing “OK” to any question that comes up, either because he is tired of answering (“authorisation fatigue”) to all these questions or because he might not understand the nature of a security related question. The latter imposes new limits on the way that JAVA security providers have to ask questions to end users. These questions should be simple enough for an end user to understand, but at the same time they have to encapsulate in these all the security-related information that has to be contacted to the user. For instance, instead of asking the user whether he wishes to grant the permission to this *word processing* applet to open a system dictionary, use system fonts or grant write permission to the filesystem (at least to a directory of the filesystem), the question could be “Grant this application with *word processing* rights?”. Finally, privileges granted or denied to applets (signers) should be stored in order to be looked up automatically the next time an applet arrives that carries a signature by the same signer.

Secure code distribution has evolved; one may identify securely the signer of the code he downloads. Furthermore, JAVA has the ability to enforce a security policy, by using the capabilities model, extended stack introspection and name space management. However, the methods and tools needed in order to define a system-wide policy have not yet been developed. It is up to the user to define the policy used against a certain piece of JAVA code, based on the digital signatures it carries. The end-user should be able to see and define only a part of the policy, enforced upon a piece of JAVA code. In an organisation, a network-wide policy should be designed by the network administrator and the administrators of the local networks.

A central repository, possibly a Directory or a Web Server, could be used in order to store the security policy. The later will be formed by three entities : a system-wide policy by the network administrator, a local policy by the administrators of the local networks of the organisation and the end-user. The policy each of these entities will define will be *based on the signer of JAVA code and on the Certificate Authority which certifies the signer’s signature*. The network administrator must be given the right to decide on matters such as whether JAVA code may be downloaded to computers inside the organisation’s network and whether it may access computers other than the one it is downloaded to. The local network administrators should be able to define in which computers JAVA has the right to access the hard disks or other magnetic storage devices and also define quotas, when access to hard disks is allowed; he should also have the right to define which users should be able to download and execute JAVA and to which computers. Finally, the end-user should be left with few choices to make on the JAVA-related security policy, because he may not have the necessary technological background. He should have to answer to questions such as

“Grant game-related privileges to this code?”, forming thus the lower layer of the security policy. The inclusion of a configurable security policy in JDK1.2 provides with the capability of grouping certain permissions. The PolicyTool can be used in order to enable the network administrator and the local administrator to store their JAVA-related security policy decisions in the central repository. The PolicyTool may be used by the end-user too in order to define the lower level of the security policy, for his machine. The PolicyTool has to be modified in order to allow to the above three individuals to have access to a specific set of permissions only, as described in the previous paragraphs. As far as the end-user is concerned, the permissions he must select to grant to specific principals should be have a high-level nature, such as “Typical game-related privileges”. The latter may be formed by a collection of carefully selected low-level permissions.

6. Conclusions

Remote execution of code has been an attractive and popular approach. JAVA programming language revolutionised the programming world three years ago within nine adjectives; the third one was *Secure*. With the JavaSecurity API, JAVA 1.1 is building on the foundation of previous versions and afford us more flexibility to do what we want, even though it is widely accepted that security, unfortunately but not surprisingly, can never be completely guaranteed.

Digital signatures have been a major contribution to the development of JAVA security. They provide the necessary infrastructure in order to define and enforce a JAVA-related security policy, based on the signature carried by a piece of JAVA code and on the Certificate Authority that certifies that signature.

The flexible security policy introduced in JDK1.2 provides an integrated method in order to grant specific permissions to applets, based on the signatures carried by the latter.

A fundamental research topic [McGraw, 1996] is a new approach to the way the Access Control List model works. The perspective idea is to provide a secure and safe way to implement mature Remote Method Invocation (RMI) facility. RMI will be possible with a strong ACL system in place.

The security scheme that surrounds the remote execution of Java code has undergone a significant number of modifications and adjustments. However, the underlying JAVA architecture has only been altered to a minor degree. This is due to the open architecture that characterises the Java programming language. This is essential in downloadable, executable content. The security scheme may evolve; it may be modified or adjusted according to new findings. The underlying architecture and the applications themselves should not be altered.

References

[Balfanz, 1997] D.Balfanz, L.Gong, (1997) "Secure Multi-Processing in Java".

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

[Felten, 1997] E.W.Felten, D.Balfanz, D.Dean, D.S.Wallach, (1997) "Web Spoofing: An Internet Con Game",
Proceeedings of the 20th National Information Systems Security Conference.

[Goldstein, 1996] Goldstein T., (1996) The Gateway Security Model in the Java Electronic Commerce Framework, JavaSoft, available at *http://www.javasoft.com/products/commerce/jectf_gateway.ps*

[Gong, 1997a] L.Gong, M.Mueller, H.Prafullchandra, R.Schemers, (1997) "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2", *Proceedings of the USENIX Symposium on Internet Technologies and Systems.*

[Gong, 1997b] L.Gong, (1997)"New Security Architectural Directions for Java", *Proceedings of IEEE COMPCON.*

[Gong, 1998] L. Gong, R. Schemers (1998) “Implementing Protection Domains in the Java Development Kit 1.2”, Proceedings of the 1998 Network and Distributed Systems Security Symposium, Internet Society

[Gritzalis, 1991] Gritzalis D., (1991) *Information Systems Security*, Greek Computer Society Publications (in Greek).

[Martin, 1997] Martin D., Rajagopalan S., Rubin A., (1997) Blocking Java Applets at the Firewall, *Proceedings of the SNDSS 1997 Symposium on Network and Distributed System Security*, pp.123-133, IEEE Computer Society Press.

[McGraw, 1996] McGraw G., Felten E., (1996) *Java Security Hostile Applets, Holes and Antidotes*, J. Wiley & Sons Inc.

[Sun, 1997a] Sun Microsystems, (1997) Secure Computing with Java: Now and the Future, at *http://java.sun.com/marketing/collateral/security.html*

[Sun, 1997b] The Java Virtual Machine Specification, (1997) available in the Web at
http://java.sun.com/docs/books/vmspec/

[Sun, 1997c] Sun Microsystems, (1997) Frequently Asked Questions - Applet Security, at
http://java.sun.com/sfaq/

[Wallach, 1997] D.S.Wallach, D.Balfanz, D.Dean, E.W.Felten, (1997) "Extensible Security Architectures for Java", *Proceedings of the 16th Symposium on Operating Systems Principles.*

[Zhang, 1997] X.N.Zhang, "Secure Code Distribution", (1997) IEEE Computer.