Security in the Large: Is Java's Sandbox Scalable?

Qun Zhong, Nigel Edwards
Extended Enterprise Lab
Hewlett Packard Laboratories, Bristol
Filton Road, Stoke Gifford
Bristol BS34 8QZ
<qz@hplb.hpl.hp.com, nje@hplb.hpl.hp.com>

Abstract

Using Java security as an example, this paper tries to draw attention to the various issues of security in large scale distributed systems, some of which are often ignored when the security mechanisms are designed. Even though a lot of work has been done on Java security, we argue in this paper that due to weaknesses inherent in the Java approach to building sandboxes, Java security is not suitable when applied to large-scale distributed systems.

In addition, the paper also explains the impact of these issues on the security mechanisms and introduces some of our efforts to find the security mechanisms that address the issues of building large scale secure systems.

1 Introduction

Java is proposed as the most promising platform to develop large-scale distributed systems on the Internet. One of its main advantages over its rival technology ActiveX is the security. It adopts the sandbox model to encapsulate untrusted applets downloaded from the Internet. However, even though the security functionality offered by Java is acceptable at the current stage, its approach to building a sandbox is unlikely to keep pace with the development of Java based large-scale applications.

The security problems presented here are not associated with a particular version of HotJava/Netscape/Explorer implementation. Instead, the problems are classified into several categories. This helps us discover the cause of these problems and predict what problems could arise when application systems scale up. We believe without the understanding of the origin of the problem, trying to cure symptom phenomena is always exhausting and costly.

In the paper, we first present the Sandbox security model and explain why it is suitable to deploy in large-scale distributed systems. Then we discuss how Java's security mechanisms support this model. The weaknesses of this approach to building sandboxes are analyzed from three aspects of large-scale distributed systems. Finally, we briefly outline our approach to building sandboxes and demonstrate how our approach addresses these weaknesses.

2 Sandbox Model and Java's Mechanisms to support it

2.1 Sandbox Model

The sandbox concept was originally developed for fault tolerance [1]. The basic idea is to provide a restricted environment to confine the behavior of a process instead of eliminating process execution failures; therefore, mitigating the damage caused by the accidental or malicious process misbehavior to an acceptable degree.

The sandbox model provides a unified conceptual model to address two security issues in large scale distributed systems. Since it is almost impossible to eliminate bugs from any non-trivial engineering projects, a sandbox provides a restricted environment needed to confine the behavior of a potential buggy process. In addition, in the development of large-scale distributed systems, there is often the need to use Commercial-Off-The-Shelf (COTS) components or outsource parts of system development to third parties. A sandbox can be deployed to provide security even when parts of the system are in black-box form[2].

However, how the security mechanisms to support the sandbox model are designed and implemented can also determine whether the security solution provided is suitable to deploy in large-scale systems. As we will discuss in the paper, how the "top-down" approach employed by Java is not likely to scale up.

2.2 How Java's security mechanisms provide support for the sandbox

Java's security mechanism consists of three interlocking 'prongs' to guarantee that the security policies are enforced [3]. The byte-code verifier checks the untrusted code to make sure that the downloaded code has not violated the properties it is supposed to have. Most importantly, it should guarantee the type safety imposed by the language semantics. The purpose is to make sure that the code does not attempt to access memory it should not access, so that all the resource accesses is requested by the code itself. The Applet Class Loader makes sure that the Java classes are separated into correct name spaces and tagged properly with the security information such as its origin and digital signature. The security manager will use this information to authenticate the mobile code and decide whether to accept or reject an access request based on this authentication information. In addition, it must guarantee that the local trusted classes that access the system resources are not spoofed. Finally, the Security Manager, which the local trusted classes consult before actually performing the resource access, acts as the security reference monitor to enforce run time security policy.

In addition to the requirement of a perfectly built JVM, to achieve an overall system security also requires that an application using Java has to be security sound enough to fend off malicious mobile code which JVM thinks behaves perfectly. Figure 1. is a typical mobile code paradigm[4]that is employed by Java enabled browsers. It highlights the interactions between various parts of a system. As we can see from the figure, JVM protects system resources by asking every applet go through its security check. However, JVM doesn't protect the application such as a browser from being attacked by malicious mobile code as the applet can directly interact with the application. The situation of the application is quite similar to a network service that can be directly accessed by untrusted users. This means that if a malicious applet exploits bugs in the application, it can actually by-pass the security check, or sandbox mechanisms that are built into the JVM to gain access to system resources. It can even replace the original trusted JVM with a rogue copy. Once this is done, all the following mobile code can access system resources without the restriction of sandbox. Examples show that some attacks are through the bugs in the browser instead of Java Virtual Machine. In a more formal terminology, this means that the security reference monitor built into the Virtue Machine is by passable if the application is not bug free.

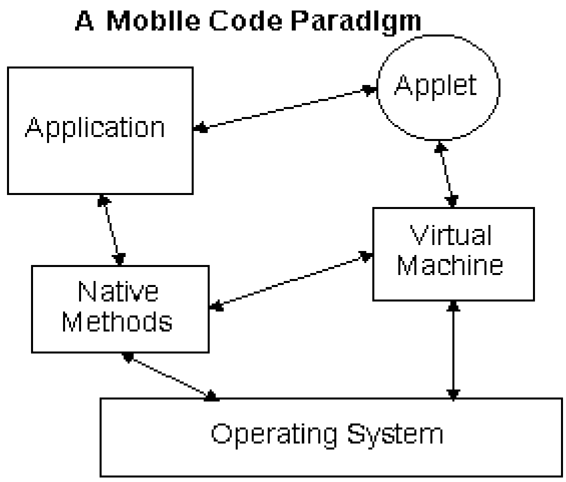

Figure 1. A Mobile Code Paradigm

From previous discussion, it is obvious that the methodology employed in implementing Java security is a top-down method:

1 The application (browser) has to be error-free so that it is immutable to all attacks.

2 The language (Java) has to be type-safe to guarantee that programs running on VM will provide exactly the original security information and only access their own resources.

3 The byte code verifier has to prove that the byte code is type safe before allowing it to pass.

4 The class loader has to put the untrusted code into the proper name spaces so that they will not interfere with each other and they have to use the local trusted class to access system resources.

5 The local trusted classes always consult the security manager, who applies the sandbox policies to decide whether to accept or reject a resource request, before accessing resources.

3. Weaknesses of Java's security mechanisms in large scale distributed systems

The approach of building security from top-down is not likely to scale up to large distributed systems. In addition, the security policy model supported by Java and conventional operating system is not likely to scale up either. In this section we look at these weaknesses from three points of view, i.e. from developing, managing and using the system security in the context of large-scale application.

3.1 Weakness in the development of Large-Scale Distributed Applications

Java's methodology of building sandbox suffers exactly the same problem as the top-down methodology in software engineering. The overall system security depends on perfect functioning of the application, the language, the three 'prongs' of JVM security mechanisms and the underlying operating system. It also depends on the proper interaction of these elements. Therefore, this kind of system security becomes very complicated and unstable if the system is large. The experience of Java security has shown that most of the security problems reported come from bugs in the implementation of the security mechanism and malicious applets attacking bugs in the application that uses the Java Virtual Machine [3].

3.1.1 Complex JVM security mechanisms

As we discussed, Java's security mechanisms are built by three "prongs". It is not an easy task to guarantee that these three prongs are bug-free. Since the three 'prongs' are tightly coupled, any bugs in one of them could render all the security effort useless. As the above model implies, the byte code verifier and class loader together should guarantee that all the resource access has to go through local trusted classes, which consult the security manager with un-tampered security information. Even if Java is type safe, which is in dispute [5], examples already show that an error in a particular implementation could have bugs that could result in the byte code verifier allowing non-type-safe bytecode through.

Even if the implementations of these three prongs are bug-free, the security inconsistency between various parts of JVM can also create a backdoor for attackers. For example, an attack named "slash and burn" used the cache to get the untrusted code into the local disk. Since code loaded from the local disk is trusted, allowing class names to start with backslashes (or slashes) can activate the impostor code and therefore, proceed to illegally access the local system. When the system is small, and the interactions between various parts are simple, it is possible to find all the delicate inconsistencies. However, when the system grows large and complex, it is very difficult, even if possible, to discover all the security inconsistencies. If one of them is unidentified, it can be deployed to compromise the whole system security.

In addition, the local classes that perform various system functions are likely to grow; therefore, the chances of containing bugs grow as well. As these classes are local trusted classes and not subject to the restriction of the sandbox, bugs in them can bring down the whole Java subsystem security as bugs in an operating system can bring down the whole system security.

3.1.2 By-passable Security

The most serious weakness of this top-down approach in developing large scale distributed applications lies in that the security it provides can be by-passed if the application contains bugs. Therefore, even if we can build perfect secure JVM, which is possible since JVM is relatively stable and simple compared with the actual application systems, the final application system's security cannot be assured as the mobile code can attack the application from inside. Once the application is compromised, the applet can do whatever the application is allowed to do, including replace a trusted JVM with a rogue copy and allow subsequent malicious mobile code to freely access system resources.

As we have already indicated, it is simply not practical to eliminate bugs from any non-trivial engineering applications. Most existing application systems are already big and complex, not to mention how complex they will become when we want to develop systems that operate at the Internet scale. The context where Java's security can be effective is simply impossible to maintain.

3.2 Inflexible security policy and complex security administration

Different organizations need different security policies to satisfy their own security requirements. Java's security policy is hard-wired into the JVM. This introduces two problems when the application it supports becomes large. One is that it can only offer very limited flexibility. For example, the original Java sandbox only allows an applet to open network connections to its original host and applets are prohibited from accessing local resources. Whether or not this policy can actually provide security is another question. The fact is that this policy actually limits applet functionality. Another problem is the introduction of new security policy would need substantial change to the JVM since the security policy is built into JVM; therefore, the chances of implementation faults are increased . A newer JVM from Javasoft allows some trusted applets to access some of the local resources, but to achieve this, the JVM has had to be modified. This could make Java's security mechanism prone to the problems discussed in the previous section.

Even when JVM provides the necessary support to specify various security policies such as in JDK 1.2, it is difficult to enforce a consistent system wide security policy since the system wide security policy has to be configured into each JVM and this policy is likely to change quite often. The security policy of using shared system resources has to be configured into and enforced by each JVM since the owners of system wide resources, such as network servers, can not distinguish whether a request comes from a trusted user or from a untrusted applet running under the user's name. This means any system wide security policy change will result in the change of every JVM's security configuration. It is not a difficult task when the system is small and the policy is simple. However, when Java is popularly used in large network applications as it is intended, this task is not easy to accomplish. In addition, the system wide security policy is likely to change quite often in a large distributed system as Java's security policy model is still the conventional Discretionary Access Control (DAC) model. This model defines the security policy of who can use the resource in which way directly, as opposed to the not so widely used Mandatory Access Control (MAC) model [6] which introduce an indirection between the users and the resources through security labels. When the shared resources and user pools are relatively small and static, the security policy based on the DAC model does not need to change often. However, this security policy is likely to be frequently changed when the application system is to operate at the global network scale where the users and the shared resources are dynamically added to or removed from the system. It is very difficult to maintain the policy consistency given the large amount of work that has to be done for each change and the frequency of the changes.

3.3 Cost of using the security in large-scale distributed systems

As we discussed in the previous section, the end-user has to be responsible for correctly configuring the security policies for the system wide resources. The cost of achieving this is very high in large complicated systems. One reason is that these end-users are normally not security experts and not aware of many subtle security implications of a particular operation. Therefore, they are more likely to be subject to so-called social engineering attacks and are more likely to hand out critical security privileges than experienced and well informed security experts. The cost of bringing them to an acceptable security qualification through education etc. cannot be ignored. Another cost is associated with the using of the security. As we said, the Java security policy model is based on the DAC model, which directly maps user's operation scope to the shared resources by employing some forms of access matrix. Therefore, the end-user who configures the JVM security policy has to be aware of every system wide policy change. For example, the code signed by CompanyX is no longer regarded trustable anymore or certain resources are now made available to an applet coming from a new partner's site. Obviously, the end-user has to spend a lot of time and effort configuring and maintaining the security policy when the system is large and there are so many shared resources and business partners.

4. Where and How should the Sandbox be Built

In contrast to the method adopted by Java, we seek support to build sandboxes from the operating system rather than from a particular language. As we can see in Figure 1, native operating system is the best place to provide some support to build non-by-passable security mechanisms since all the resource access has to go through it. In addition, operating system is normally more stable than application and protected by hardware, making it very hard to tamper with. Therefore, the sandbox model can be supported from bottom-up, i.e. from the lowest layer of the system that actually manage various system resources --- the operating system. Consequently, it is possible to reduce the system security dependencies between JVM, application and operating system that exist in top-down security approach deployed by Java to operating system only. Without this support, we would have to secure every application to achieve the overall system security.

Conventional operating systems do not provide the necessary facilities to confine the behavior of a process. Instead of building a set of new security mechanisms totally from scratch, we have found that we can reuse and refine some B-level operating system [7] security features, particularly Mandatory Access Control and privilege management mechanism to serve this purpose.

The initial stage of our work has yielded exciting results. MAC and related privilege management mechanism has been successfully deployed to sandbox COTS components[2] as well as mobile code[8]. The results have proved that these security mechanisms provide the support for the development of secure large applications through the ability of confining the behaviour of processes. The only things that the overall system security depends on are several very small and simple trusted programs, which can be thoroughly studied and act as reference monitors to the shared resources, and the operating system, which is much more tamper-proof and stable than most non-trivial application systems.

Additionally, our work on mobile code demonstrates that the cost of the using operating system security is significantly lower than using the security built by Java's approach since the MAC model is developed to provide administrative security. The task of security administration and management of shared resources is pushed to the system security administrator. Therefore, the end-user is liberated from complex security maintenance and can concentrate on his own work.

Finally, the MAC model also looks promising in solving the scalability and flexibility problem in security management as MAC introduce an indirection, i.e. the security label, that allows us to classify users and resources and make the security policy according to the classification of objects in the system. This ability has two advantages. One is that the relationships between the classifications of objects are much more static than the relationships between dynamic objects as objects in a system tend to have shorter lifetime. Therefore, a stable system wide security policy could be defined and enforced consistently. Another advantage is that depending on the classification of the users and the resources, different policies can be produced to satisfy different business security requirements. For example, the same mechanism proposed in our work[8] could be deployed by Internet Service Provider(ISP) to help parents filter out unwanted WWW site as well as used by a company to create different views of their IT system for different users. We are currently investigating this area.

Acknowledgements

The author would like to thank Dirk Kuhlmann and Liqun Chen for the very valuable comments about the paper. Thanks also go to Chris R. Dalton, Andy Norman, Owen Rees for discussions on the ideas presented in this paper.

---

1 Robert Wahbe, Steven Lucco, Thomas E. Anderson and Susan L. Graham, " Efficient Software-based fault isolation", Proceedings of the Sym. On Operating System Principles, 1993

2 Qun Zhong, Nigel Edwards, "Security Risk Control of COTS-based Applications", to appear in IEEE Computer, special issue on COTS, June, 1998

3 Gary Mcgraw and Edward W. Felten, Java Security: Hostile applets, Holes, and antidotes, Wiley Computer Publishing, 1997

4 Marvin Schaefer, Sylban Pinsky etc. "Ensuring Assurance in Mobile Computing", proceedings of 1997 IEEE Symposium on Security and privacy", May 1997, Oakland

5 S. Drossopoulou and S. Eisenbach, " Is the Java type system sound?", Proceedings of the fourth International Workshop on Foundations of Object-Oriented Languages, Paris, Jan. 1997

6 Ravi S. Sandhu, "Lattice-Based Access Control Models", IEEE Computer, Nov. 1993

7 TCSEC, Department of Defence, "Trusted Computer System Evaluation Criteria", DoD 5200.29-STD, Dec., 1985

8 Qun Zhong, Nigel Edwards, Owen Rees, “Operating system Support for the Sandbox Methods and its Application on Mobile Code Security”, HP Laboratories Technical Report HPL-97-153