HAAGA-HELIA
University of Applied Sciences

Security Issues of the Sandbox inside Java Virtual Machine (JVM)

Mohammad Shouaib Hashemi

Bachelor‘s Thesis

Business Information Technology

2010

HAAGA-HELIA
University of Applied Sciences

Abstract

Degree Program in Business Information Technology

Authors Mohammad Shouaib Hashemi	**Group** X
Title Security Issues of the Sandbox inside Java Virtual Machine (JVM)	**Number of pages and appendices** 53
Supervisors Markku Somerkivi	

Nowadays most of our sensitive information is accessible through web browsers. Securing these systems is of great concern, because any web accessible system is continuously exposed to threats. Internet, despite its important role in our lives today, is suffering from many drawbacks such as threats to confidential information, scam, fraud and even endangering users' systems in the extreme.

Among the major concerns of large companies and private users regarding rogue java applets are destruction, modification and theft of their confidential documents. According to a Symantec Internet Security Threat Report there has been an overall increase in threats to confidential information. According to another Symantec Internet Security report published in October 2009 vulnerabilities in the Web browsers and Web browser plug-ins are causing attacks, attempting to gain access to a user system.

In this paper the basic security features of the Java platform, especially the Java Sandbox model security issues are discussed. Java Security Model (Sandbox) is one of the first and most popular security models that can be integrated into the Web browsers in order to prevent applets from doing anything destructive to the host system. Our ambition is ultimately to prove that the Java Sandbox Model is really secure, reliable and trustworthy security architecture.

The thesis is concluded by the findings that surely the Java Sandbox model provides mechanisms that can protect users' confidential information from being stolen, deleted, modified and any other destructive external threads. The Sandbox model allows the users to take advantages of ad-hoc applications while it surrounds the running application to protect users' system from malicious codes.

Key words

Java, Java Sandbox Model, Java 2 SDK, Java Virtual Machine

Table of contents

1 Introduction and Background 1

1.1 Introduction 1

1.2 The Java Virtual Machine 2

1.3 What is an applet 2

1.3.1 How to embed an applet into a Web page 3

1.4 Why Software Developers Choose Java 3

1.4.1 Java is Safe 4

1.5 Research Problem and Questions 4

1.6 Research Approach 5

1.7 Thesis Outline 5

2 Literature Review 6

2.1 Theoretical Background 6

2.2 Overview of Java Security Model 7

2.2.1 JDK 1.0 security 8

2.2.2 JDK 1.1 Security 9

2.2.3 Java 2 Security 11

2.3 Overview of Java 2 Security Architecture 16

2.4 The Java Sandbox Architecture 16

2.4.1 The Class Loader 19

2.4.2 The Bytecode Verifier 22

2.4.3 The Security Manager 29

3 Data Analysis 35

3.1 Threats of Malicious Codes 35

3.2 Design of Java Security Model 37

3.2.1 Is Java Bytecode Enhancing Security? 38

3.2.2 Turning off Java 39

3.3 Implementation of Java Security Model 40

3.3.1 Integrating Java Sandbox into a Web Browser 44

4 Conclusions 46

4.1 General Discussion 46

4.2 Research Questions 47

4.3 Conclusion 47

Bibliography 49

Table of Figures

Figure 1 JVM illustration 2
Figure 2 JDK 1.0.x Security Model 8
Figure 3 JDK1.1 Security Model 9
Figure 4 JDK1.2 Security Model 12
Figure 5 A machine has access to many resources 18
Figure 6 Class Loaders hierarchy 20
Figure 7 Control of Java bytecode 22

Tables

Table 1- The Check Methods of Security Manager

1 Introduction and Background

This chapter begins with a brief introduction; Problem Discussion concentrates which will help the reader to understand the insight of the research area, followed by description of the background of the selected area. The background description ends with research problems and specific research questions. Finally the methodology employed is presented.

1.1 Introduction

Internet is a technology that spreads faster than any other technology in the world and thus it has become part of our life. As nowadays most of our sensitive information is accessible through web browsers, securing these systems is of great concern, because there is a potential risk of attack to any computer hooks to networks. The risk is vast in an environment in which the software is downloaded across the network and executed locally as this is the case with the java applets. Applets are automatically (an applet may not give any visual signal of its existence or execution) downloaded across the network and run on the host machine when a user accesses in a browser a web page containing applet. A user coming across malicious applets is within the realms of possibility.

The Java Sandbox model provides you an environment, where you could welcome any code from any source. But when the code from an untrusted source runs Sandbox restricts the code from taking any actions that could possibly harm your system. It terms of applets the restrictions means that applets will be unable to determine information about each other; each applet is given its own memory space to run.

Java has solved the problem of security and portability in advanced in this way that the output of a Java compiler is not executable code. It is the Java Virtual Machine (JVM) which helps to solve the major problems associated with downloading programs over Internet. The fact that a Java program is interpreted into an intermediate language (byte-code) is a part of Java language security. Because the execution of every Java program comes under the control of JVM, the JVM prevent it from modifying sensitive information in the user machine and harming the user machine.

1.2 The Java Virtual Machine

Java Virtual Machine (JVM) provides a safe place for Java programs (applets) to run in the world of the Internet. It is a machine inside a machine or it is machine at the heart of the Java platform. As matter of fact it is a machine that does not physically exist or there is no hardware implementation of this microprocessor available, but it exists only in the memory of our computers. Figure1 is an illustration the location of JVM.

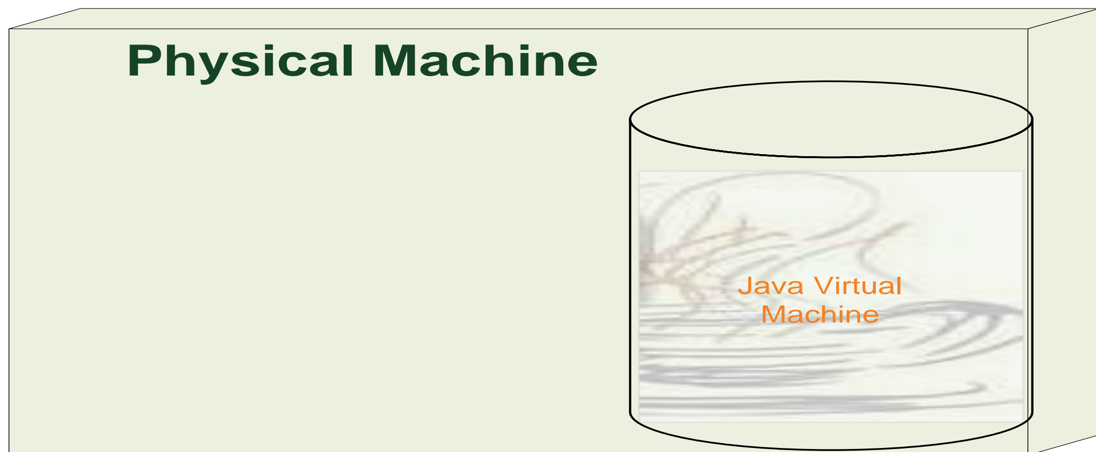

Figure 1 JVM illustration

The JVM has the main role to Java's portability because compiled Java programs run on the Java Virtual Machine. It is a platform-independent execution environment that converts Java byte code into machine language according to the operating system and executes it. It also allows you to run Java applets' inside your Web browser safely.

1.3 What is an applet

The introduction of Java applets has animated the development of World Wide Web and Internet applications. Before Java, HTML was only a static contents and World Wide Web was a read-only interface. The execution of applets in a browser while browsing web pages has made the browser to become the universal computer interface.

Applets are small programs written in Java language that can be embedded into HTML pages. Applets can run in a WWW browser using JVM or applet viewer (applet viewer is a standalone application provided by Sun). The Java applet is itself a fully functioning little program. They are designed to run inside a web browser and to perform some tasks such as animated graphics and interactive tools. Applets are supported by most web browsers such as Mozilla

and internet explorer. Applets can be downloaded from the Internet and run safely within a web browser sometimes even without noticing them.

1.3.1 How to embed an applet into a Web page

A Java applet can be embedded into a Web page by a <APPLET> tag. The following piece of HTML code illustrates its use.

```
<APPLET code="SomeJavaCode.class"
      width=300
      height=300>
<PARAM name=input1 value="value1">
<PARAM name=input2 value="value2">
Your browser does not know how to exec Java applications.
</APPLET>
```

The plain text inside <APPLET>, </APPLET> tags is displayed only in the browsers that do not support Java. Parameters can be delivered to Java applet by name/value pairs in the <PARAM> tag. (Michael Girdley, Kathryn A. Jones, quoted 20.01.2010)

1.4 Why Software Developers Choose Java

Java has been tested, refined, extended, and proven by a dedicated community [21]. Write software once in one platform and run it anywhere in any other platform is one of the most important key features of java language that makes java as the most powerful language and preferred language over other languages. As a software developer or programmers you don't have to think or be concerned with how or where your software will run instead you can concentrate on writing the software. Aside from being pre-eminent language of the Internet Java is a technology. Beside programming language it also is a development, an application and a deployment environment .As a programming language, Java can create all kinds of applications that you could create using any other programming language.

Aside from its best features secure, portable, platform independent and distributable over a network, Java is also simple, garbage collected, portable and robust according to Sun java definition; thus it has become extremely important to developers.

1.4.1 Java is Safe

Accepting the fact that there is no such thing as a 100-percent secure system, at least if we want that system to do anything useful. The security features within the Java language want to ensure that a program will be unable to modify sensitive information that may reside in the memory of a user's machine. In terms of applets, these protections also mean that applets will be unable to determine information about each other; each applet is given, in an essence, its own memory space in which to operate. In Java, objects cannot be directly manipulated by a programmer; instead, they can only be accessed through their public interfaces. Programmers cannot directly access memory either, but must use object references. This fact makes it much harder for a nasty program to search through memory looking for interesting data such as passwords and credit card numbers.

It is hard to fix Vulnerabilities with security patches[1]. Security problems should therefore be identified early in a development process. Ad hoc solutions to security problems may create new vulnerabilities and new security hole. Thus security was in minds of developer while developing Java.

1.5 Research Problem and Questions

Security is a critical part of any application. It would be impossible for this project to address security factors from throughout the entire Java range. The project will be focussing only on the Java Sandbox security issues. Thus this leads the research to the following questions:

1. How the sandbox protects our personal data and applications from external threats.

2. How the java security feature especially sandbox will minimize the likelihood that hackers will not be able to manipulate applications and access, steal, modify, or delete sensitive data

3. To investigate various security issues and options of Java sandbox.

[1] A patch is a piece of software designed to fix problems. (http://en.wikipedia.org)

1.6 Research Approach

Although research on Java Sandbox security issues is not something new, because most of the concepts in this study have been examined before, but mostly in the Java Virtual Machine security concepts as whole, hence the aim is not to make any simplification, but instead to provide step-by step analysis of java sandbox security processes, which intends to provide us a deeper understanding of java sandbox security issues.

My intention with this research is to describe and clarify, and find complete and detailed information about the security issues of the Java Sandbox. Thus to gain deeper understanding of the security issues in the Sandbox context, this research conducted as a qualitative study where focus lies on describing an event with the use of words.

1.7 Thesis Outline

This thesis is divided in four chapters, namely Introduction and Background, Literature Review, Data Analysis and Conclusion. Chapter 1 gave an introduction and background to the topic, research problem and questions, and research approach are also described there. Chapter 2 presents literature review. It provides the relevant literature in the filed that I selected for my research. Chapter 3 will analyze the data from preceding chapter. It will present the achieved results and the interpretation of results. Chapter 4 will bring the report to the end. It will present the conclusion of the study and summarizes the main findings of the research.

2 Literature Review

The previous chapter provided the introduction and research problem and research objective. In this chapter I will present literature review. The aim of this chapter is to provide the relevant literature in the field that I selected for my research.

2.1 Theoretical Background

To identify and describe the Java Sandbox security issues, this background study involved reviewing numerous literatures related to security issues of the Java Sandbox. It was used to give me a broader understanding of the Sandbox security issues in different environments. I read and used many books, theses, pdf files, PowerPoint slides and articles like, Java security evolution and concepts by (Raghavan N. Srinivas, JavaWorld.com, 28/07/00), Inside the Java virtual machine by (Bill Venners, 1998), Java™ 2 Platform Security Architecture Version 1.2, by (Li Gong, java.sun.com, 2002), Securing Java by (Gary McGraw and Edward Felten) and many more about Java security from the net.

Java is designed so that programs can be dynamically loaded over the network and run locally (Gary McGraw & Edward W. Felten, 37). Verifying a program's veracity before executing it is necessary, because malicious applets can steal or destroy information on user's machine. The Java Virtual Machine provides an environment where an untrusted application can be executed in a trusted environment. Discussion of Java's security model often centres on the idea of a sandbox model. The idea behind this model is that when you allow a program to be hosted on your computer, you want to provide an environment where the program can run, but you want to limit the program's play area within certain bounds. You may decide to let the program have access to certain system resources, but in general, you want to make sure that the program will not access local resources out of its sandbox. The essence of the sandbox model is that local code is trusted to have full access to vital system resources (such as the file system) while downloaded remote code (an applet) is not trusted and can access only the limited resources provided inside the sandbox.

2.2 Overview of Java Security Model

Traditionally, you had to trust software before you ran it. You achieved security by being careful only to use software from trusted sources and by regularly scanning for viruses just to make sure. After some software got access to your system, it had full reign. If it was malicious, it could do a great deal of damage because no restrictions were placed on it by the run time environment of your computer. So in the traditional security scheme, you tried to prevent malicious code from ever gaining access to your computer in the first place. (Bill, Venners, 34-35)

Java has taken a pro-active approach .Java platform's security model known as the sandbox model makes it easier to work with software that comes from sources you don't fully trust. The main goals of sandbox security model are: to focus on protecting users from hostile programs and to reduce; for the naive users who just surf the web, the necessity of becoming security expert. To accomplish this goal, the model provides a very restricted environment in which to run untrusted code or software. The advantage of the sandbox model is that you don't need to figure out what code you can and can't trust. The sandbox itself prevents any viruses or other malicious code from doing any damage to your computer.

Since the inception of Java sandbox model, the model has been extended and some significant new security features are available in the Java Development Kit (JDK) version 1.1 and 1.2 or java 2 platform security model. Since the Java sandbox model has been widely distributed to millions of users with their Web browsers, it is safe to say that the Java sandbox is today's most widely used sandbox (Gary McGraw and Edward W. Felten, 38). Before going deep into details of java sandbox I will briefly explain the different versions of sandbox security issues.

In order to remove confusion before jumping to more details:

- JDK stands for Java Development Kit
- SDK (JDK 1.2 was renamed SDK or Java 2 or Java 2 SDK) stand for Software Development Kit
- Don't confuse the JDK or SDK with the JRE (Java Runtime Environment). The JRE has everything needed for running Java programs, but does not have the tools needed for developing Java programs for example the compiler.

2.2.1 JDK 1.0 security

In the initial design of security model provided by the Java platform (Sandbox model) every single piece of code obtained from the open network was considered untrusted code. The model provided a very restricted environment to place untrusted code (applet). The core of the sandbox model is that local code is fully trusted and had access to all critical system resources for example the file system, but downloaded remote code (an applet) is not trusted and can only have limited access. Figure 2 is an illustration of the initial security architecture in JDK1.0 (Java Development Kit 1.0).

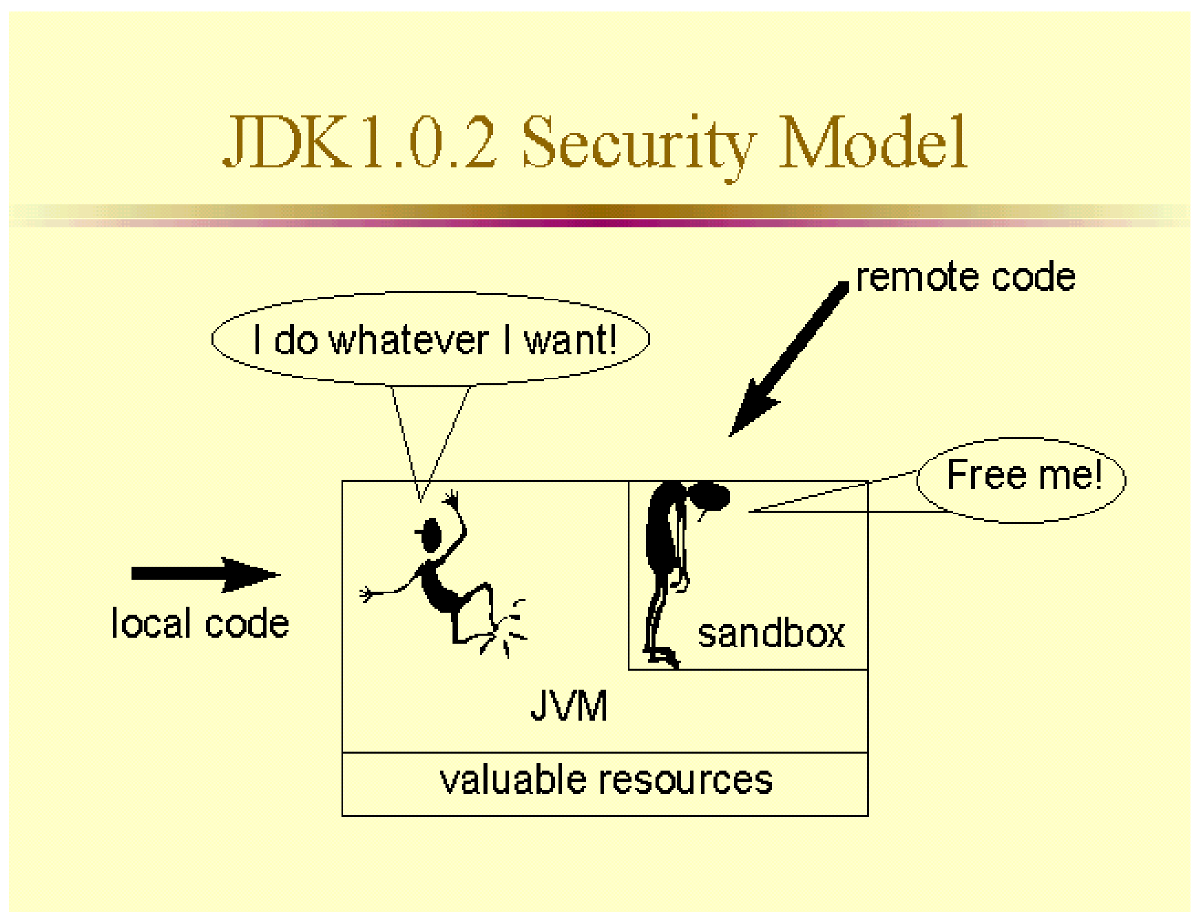

Figure 2 JDK 1.0.x Security Model (Li Gong, Marianne Mueller, Hemma Prafullchandra &Roland Schemers, 1997, 2)

The initial sandbox model provides a kind of rough security control: local code is considered trusted and can have full access to the local system and remote code (applet) is untrusted and can have very limited access.

2.2.2 JDK 1.1 Security

JDK 1.1 (Java Development Kit 1.1) introduced the concept of signed applet (Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers, 3). Figure 3 is an illustration of the signed applets security architecture in JDK1.1. A digitally signed applet is treated like local code, with full access to resources, if the public key used to verify the signature is trusted. In JDk1.1 unsigned applets are run in the sandbox with the same restriction as in JDK1.0. Signed applets are delivered, with their respective signatures, in signed JAR (Java Archive) files (Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers, 3).

The model is said to be too coarse (Michael Girdley & Kathryn A. Jones, quoted, 2.2.2010): End-user has only two options: signed the applet to gives the applet additional privileges or unsigned the applet and consequently diminished the usability of the applet. Both options have their own drawbacks. Signed applet is assumed to be safe or has come from a trustworthy source and given complete access to the local computing platform.

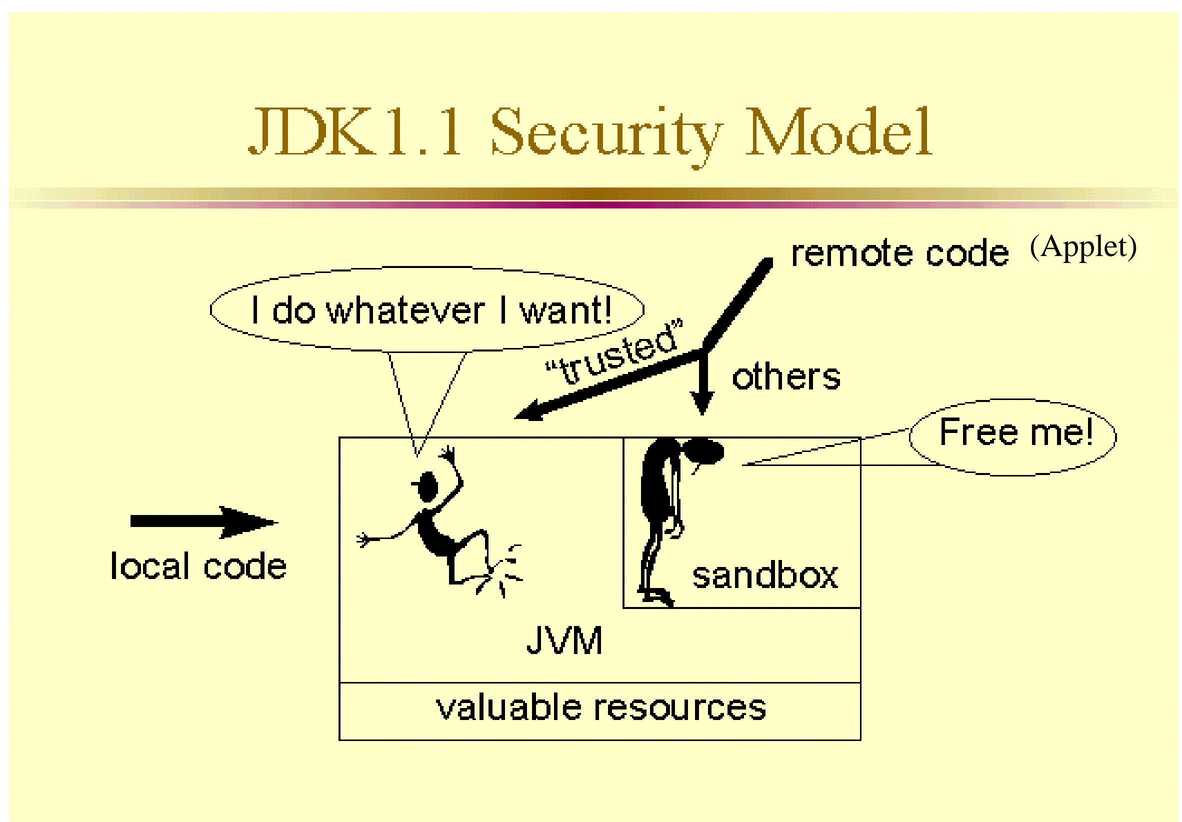

Figure 3 JDK1.1 Security Model (Li Gong, Marianne Mueller, Hemma Prafullchandra &Roland Schemers, 1997, 3)

In JDK 1.0 and 1.1 local Java applications are fully trusted and therefore could run with full privileges. In fact local Java applications should not be given full access to all parts of the system, because they may also be a potential risk to the system. For example it is common in software development world that a software developer test or installs a demo of the developed program on the local system and then tries the program outside. The demo program could cause a potential damage to the underlying system. Thus, it is prudent to give these types of programs less than full access to the system.

2.2.2.1 Signed Applets

Suppose that a customer of a brokerage firm uses a stock-trading applet loaded from the brokerage's Web site. This customer may want to let the applet update local files that contain her stock portfolio. However, access to the client-side file system is prohibited by the sandbox model (JDK1.0). Thus, this customer needs flexible access control, whereby certain applets can have access that is outside the sandbox. With JDK 1.1, the brokerage firm could sign the trading applet, and, assuming that the customer configured her Java runtime to recognize the brokerage firm to be trusted signer, the applet could access resource outside the sandbox. (Li Gong, Gary Ellison & Mary Dageforde, 2003, 32, quoted 18.02.2010)

However, the customer may have installed on her local desktop financial management software that handles income tax issues. She might not feel comfortable letting the brokerage firm's applet have free rein on her entire desktop system. In this case, it may be best to confine the applet to limited file system access, perhaps only to the brokerage firm's folder. What is needed is a model whereby the sandbox can be customized- for example, by the client system-to have flexible shapes and boundaries: in other words, fine-grained access control. (Li Gong, Gary Ellison & Mary Dageforde, 2003, 32, quoted 18.02.2010)

When Sun Microsystems and the major browser companies switched from the simple sandbox security model to fine-grained access control based on signed code, they sometimes tried to give the impression that this would "enhance" security. Code signing cryptographically binds a certain principal, such as an individual or company, to a piece of code. It is not feasible to

change a signed piece of code without being detected, nor is it possible to impersonate someone else by augmenting code with their signature. (Dirk Balfanz & Edward W. Felten, 3)

With this technology in place, it is always possible to find out who signed the code. The argument goes that now users can decide that code signed by untrusted or unknown parties should be denied privileges. (Dirk Balfanz & Edward W. Felten, 3)

However, the browser will still load “untrusted” code, the difference is only that the untrusted code will run with fewer privileges. Untrusted code is usually run within the old sandbox (i.e. no file system access, no network access except back to the originating host). But we have already seen that code can exploit implementation bugs and break out of that sandbox. Code signing provides no enhanced security at this point. (Dirk Balfanz & Edward W. Felten, 3)

Even worse, code signing introduces new system code which can have[2], and in fact has been found to have , new bugs that weren’t there before. This opens even more doors for malicious applets. On the trade-off continuum between security and functionality, code signing belongs on the functionality side, as it can be used to provide fine grained access control and provides the user with information about who endorsed a specific piece of code. Code signing does not “enhance” security. (Dirk Balfanz & Edward W. Felten, 3)

Thus necessity for a new and unqualified approach had to be taken in consideration about Java security model. The Java developers had realized the necessity too and as a result the JDK 1.2 was released.

2.2.3 Java 2 Security

The release of JDK 1.2 has brought a number of improvements over JDK 1.1. In JDK 1.2 every code, regardless of whether it is a local or remote (applet), can be under a security policy. As a user or a system administrator you could configure security policy to define different permission for different code from various signers or locations. Each permission specify a permitted access to a particular resource, such as read and write access to a specified file or directory or connect access to a given host and port. The runtime system organizes code into individual domains, each of which encloses a set of classes whose instances are granted the

[2] www.cs.princeton.edu/sip/news/april29.html

same set of permissions. A domain can be configured to be equivalent to the sandbox, so applets can still be run in a restricted environment if the user or the administrator so chooses. Applications run unrestricted, as before, by default but can optionally be subject to a security policy. (Mary Dageforde, quoted 22.01.2010)

The new security architecture in JDK 1.2 is illustrated in the following figure (figure 4). The arrow on the left end refers to a domain whose code is granted full access to resources; the arrow on the right refers to the opposite extreme: a domain restricted exactly the same as the original sandbox. The domains in between have more accesses allowed than the sandbox but less than full access. (Mary Dageforde, quoted 22.01.2010)

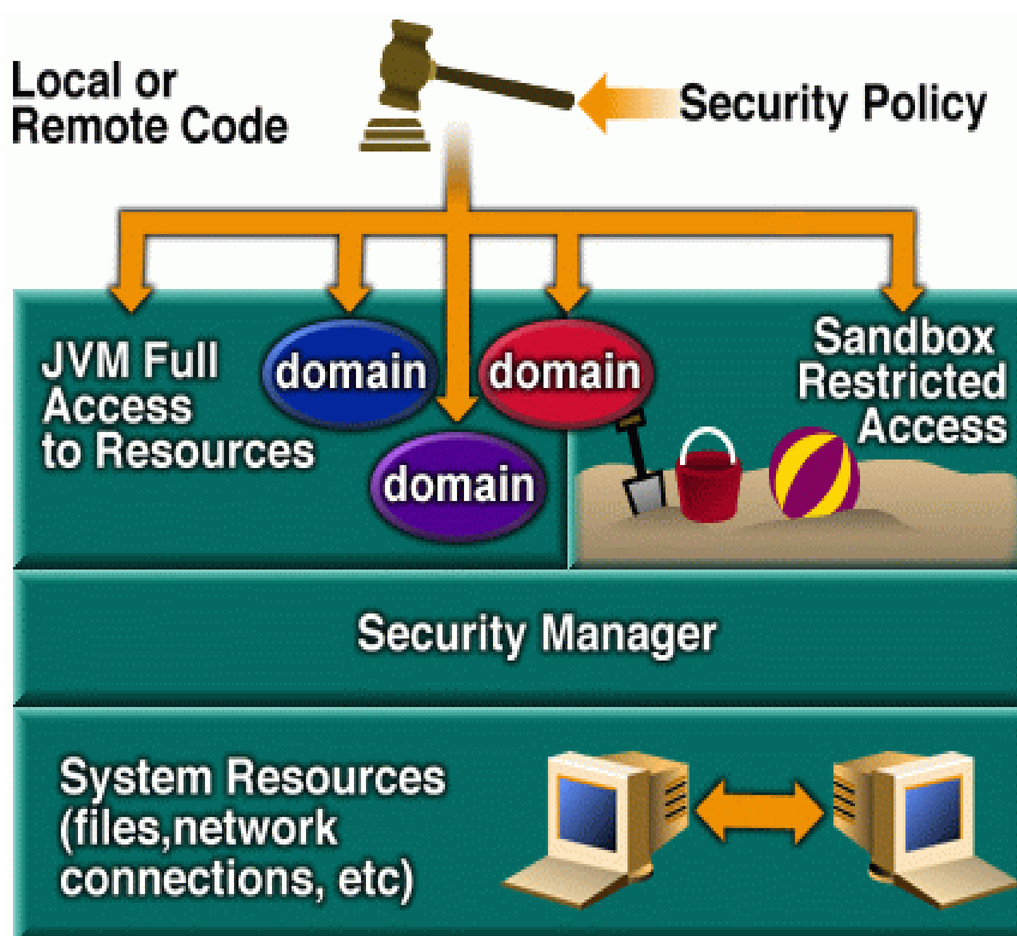

Figure 4 JDK1.2 Security Model (Mary Dageforde, Security in JDK1.2)

The new security architecture in SDK1.2 (the JDK has been renamed later and is now called a Software Development Kit (SDK), so we have the Java 2 SDK) let the users expand the original sandbox idea, create their own sandboxes, eliminate the sandbox entirely, and even the opportunity to run a java application (not applet) within a sandbox that the user or system administrator has constructed or configured.

The Java 2 security architecture eliminates the need to write custom security cod for all but the most specialized environment, such as the military, which may require special security properties, such as multilevel security. Even then, writing custom security code is simpler and safer than before. (Li Gong, Gary Ellison & Mary Dageforde, 2003, 32, quoted 18.02.2010)

The Java security architecture in SDK 1.2 is an expansion of the JDK 1.1 security model. Aside from other improved feature, the new security model allows configurable access controls for Java code. For better understanding of the new and improved security architecture in SDK 1.2, a more detailed explanation of improvement is needed.

- **Fine-grained access control**

Fine- grained access control is a mechanism of checking permissions before allowing access to certain protected system resources to increase system integrity. This capability existed in the JDK from the beginning (Li Gong, quoted 16.02.2010), but it was so much complicated to implement a more flexible and finer-grained access control on the Java platform. By sub classing and customizing the Security Manager and Class Loader the application writer had to accomplish this work. Thus, the application developer had to have a very deep knowledge of computer and internet security in order to be able to customize such security-sensitive programming.

- **Easily configurable security policy**

Once again, this capability existed previously in the JDK but was not easy to use. Moreover, writing security code is not straightforward, so it is desirable to allow application builders and users to configure security policies without having to program (Li Gong, quoted 16.02.2010). Configuring security policy is easy in SDK 1.2; all you have to do is to edit the appropriate

policy configuration file. SDK provides a number of ways to do this: by using either a text editor, command line or GUI tool. Configurable security policy of SDK 1.2 allows the users to specify a security policy for applets and applications and to have control over the access they allow to these programs based on their degree of trust in the source of the Java program they execute.

- **Easily extensible access control structure**

Prior to Java 2 SDK creating new access permission required adding new *check* method to the Security Manager class.

For example, to check whether a file can be opened for reading, you would call the *checkRead* method on the currently installed *Security Manager*. Such a design is not easily extensible, because it does not accommodate the handling of new type of checks that are introduced as after-marked add-on to the Java runtime. It is also not scalable. For example, to create a new access check, such as one that check whether money can be withdrawn from a bank account, you would have to add a new *checkAccountWithdraw* method to the Security Manager class or one of its subclasses. Thousands of various kinds of checks are possible. If methods were created for this large a number, they would clutter the Security Manager class. In fact, because many checks are application specific, not of all them can be defined within the JDK. What is needed is an easily extensible access control structure. (Li Gong, Gary Ellison & Mary Dageforde, 2003, 33, quoted 18.02.2010)

The new model (Java 2 SDK) does not require adding of new checks method to the security manager class; instead it looks into permissions in the policy file. The check permission method automatically handles all security checks.

- **Extension of security checks to all Java programs, including applets as well as applications**

Java application and signed applets are no longer fully trusted. Instead both local code and applets are treated equally and both are subjected to the same security control. If needed there

is a possibility to declare the policy on both local code or remote code (applet) as most trusted, thus the code will run as totally trusted.

- **Robust and simple Internal Security Mechanisms**

In JDK 1.0 and JDK 1.1, a number of internal security mechanisms were designed and implemented, using techniques that were rather fragile. Although they worked reasonably well, maintaining and extending them proved difficult. For Java 2, we made important internal structure adjustment to reduce the risk of creating subtle security holes in the Java runtime and application programs. This involved revising the design and implementation of the Security Manager and Class Loader classes, as well as the underlying access control mechanism. (Li Gong, Gary Ellison & Mary Dageforde, 2003, 34, quoted 18.02.2010)

2.2.3.1 The Access Controller

The release of SDK 1.2 (Java 2) introduced a new security policy-based class called Access Controller class which was not a part of the earlier versions. Access Controller is a much more simple and flexible mechanism for determining security policies. Access controller is the mechanism that the Security Manager actually uses to enforce its protections (Scott Oaks, 1998, 90). Access Controller gives the programmer the ability to check whether a user has the permission to perform a particular operation.

Before the existence of the Access Controller the Security Manager was a self-reliance security component because security policy was as an internal logic of the Security Manager. The Security Manager had to rely on its internal logic to determine the security policy that should be in effect, and changing the security policy required changing the Security Manager itself (Scott Oaks, 1998, 90). Therefore, in the earlier versions of the Java sandbox, implementation of a customized security policy with the Security Manager alone was not impossible, but it was too hard to implement and needed much effort. Starting with Java 2 the Security Manager can defer to Access Controller on access-control decisions.

Access Controller has brought flexibility in Security Manager for determining the security policy, because the policy needed to be enforced by the Security Manager can be specified in a file. Access Controller also provides a simple way of granting specific permission to a particular code. For back-ward-compatibility reason, the Java 2 API still calls the methods of

the Security Manger to enforce system security, but most of these methods call the AccessController (Marco Pistoia, Nataraj Nagaratnam, Larry Koved & Anthony Nadalin, 2004, 274).

2.3 Overview of Java 2 Security Architecture

The new security architecture in Java 2 platform unlike earlier versions is policy-based and easily configurable architecture. The new mechanism provides the ability of granting specific permission to a specific system resource for a specific piece of code, in simple words a fine-grained access control. As a Java system administrator you have the power to have control over any application, by defining what each application is permitted to do. For example giving read and write permission on a file (i.e. A), only read permission on another file (i.e. B) and no permission on another file (i.e. C), depending on who is the singer of the code and/or location of the code from where the code was loaded. The new mechanism of Java 2 looks into the security policy in order to be able to decide on granting access permission to running code. The permissions based on the properties of the code, such as the origin of the code, who is running the code, signer of the code if signed and so on.

Security checks is invoked by any attempt to access the protected resources, security check compares the granted permission with the one which is trying to access the resource. If access was not defined in the policy file, the default policy is the sandbox policy as implemented in JDK 1.0 and JDK 1.1.

2.4 The Java Sandbox Architecture

The Java sandbox exits in the memory of your machine outside the reach of Java programs. This prevents Java programs from being able to call low level system functions that may cause data corruption or other damages. The Java sandbox main job is to prevent malicious applets from accessing the resources in your machine. It applies several restrictions on the applets. The base Java Security sandbox is made of three major components:

- the Class Loader
- the byte code Verifier
- the Security Manager

One of the greatest strengths of Java's security model is that two of these components, the class loader and the security manager, are customizable. By customizing these components, you can create a customized security policy for a Java application. As a developer, you never need to create your own customized sandbox. You can often make use of sandboxes created by others. When you write and run a Java applet, for example, you make use of a sandbox created by the developer of the web browser that hosts your applet. (Bill Venners, 45)

Each of the three components must work properly in order for Java to perform in a secure fashion. The Security Manager depends on Class Loaders to correctly label code as trusted or not trusted. Class Loaders also shield or protect the Security Manager from spoofing attacks by protecting local trusted classes making up the Java API. On the other hand, the class loader system is protected by the Security Manager, which ensures that an applet cannot create and use its own Class Loader. The Verifier protects both the Class Loaders and the Security Manager against language-based attacks meant to break the VM. All in all, the three parts strongly connected to create a default sandbox.

In the past some thoughts referred to the Java security model as "three-layer" defence. The idea behind three layer defence was that if an applet succeed in crossing the first layer, there are still two other layers left to protect the system . In fact, the three parts of sandbox are dependent to each other tightly or tightly coupled. A bug in one of them could cause the entire security system to break.

The Java sandbox is responsible for protecting a number of resources, and it does so at a number of levels. Consider the resources of a typical machine as shown in figure 5. The user's machine has access to many things (Scott Oaks, 1998, 4):

- Internally, it has access to its local memory (the computer's RAM).
- Externally, it has access to its file system and to other machines on the local network.
- For running applets, it also has access to a web server, which may be on its local (private) net or may be on the Internet.
- Data flows through this entire model, from the user's machine through the network and (possibly) to disk.

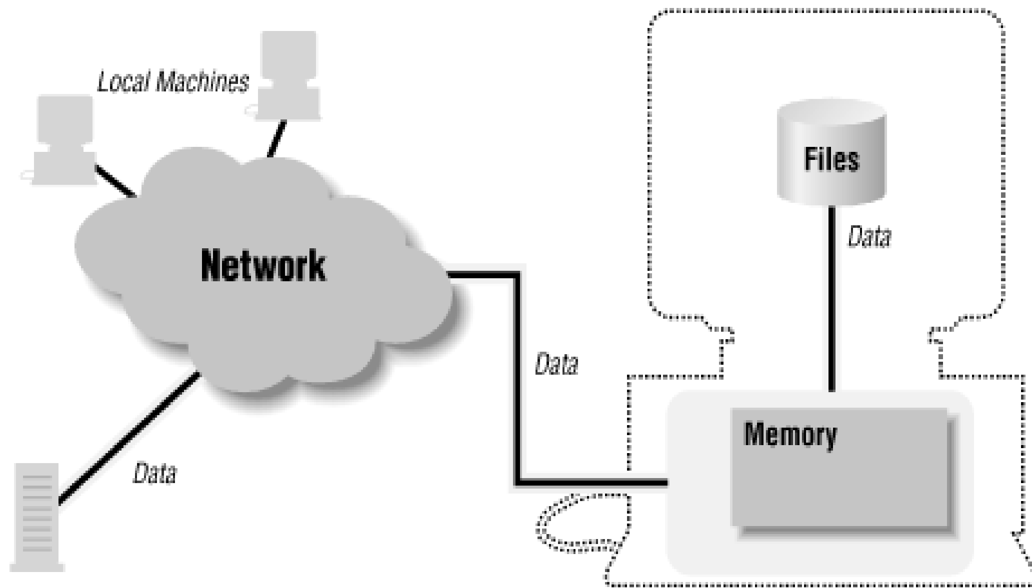

Figure 5 a machine has access to many resources (Scott Oaks, 1998, 5)

Each of these resources needs to be protected, and those protections form the basis of Java's security model. We can imagine a number of different-sized sandboxes in which a Java program might run (Scott Oaks, 1998, 4-5):

- A sandbox in which the program has access to the CPU, the screen, keyboard, and mouse, and to its own memory. This is the minimal sandbox -- it contains just enough resources for a program to run.
- A sandbox in which the program has access to the CPU and its own memory as well as access to the web server from which it was loaded. This is often thought of as the default state for the sandbox.
- A sandbox in which the program has access to the CPU, its memory, its web server, and to a set of program-specific resources (local files, local machines, etc.). A word-processing program, for example, might have access to the docs directory on the local file system, but not to any other files.
- An open sandbox, in which the program has access to whatever resources the host machine normally has access to.

The sandbox then, is not a one-size-fits-all model. Expanding the boundaries of the sandbox is always based on the notion of trust: in some cases, I might trust Java programs to access my

file system; in other cases, I might trust them to access only part of my file system; and in still other cases, I might not trust them to access my file system at all. (Scott Oaks, 1998, 5)

2.4.1 The Class Loader

The class loader is the first line of defence in Java security architecture. A class loader is an object that is responsible for loading or requesting classes, either from the web server or from the local system resources. It is the Class Loader, which gives Java its dynamic loading capabilities. The Class Loader is the part of the JVM that loads classes into memory or brings codes into the Java Virtual Machine. The class loader is important in Java's security model because initially, only the class loader knows certain information about the classes that have been loaded into the virtual machine (VM). Only the class loader knows where a particular class originated from. Java's class loader architecture is complex, but it is a central security issue. The original Class Loader architecture was meant to be extensible, thus a new Class Loader could be added to the running system. So it was clear at the beginning that a malicious Class Loader could break Java's type system, and hence it could be contributed a breach in Java's security. As a result, Java implementation does not allow applets to create Class Loader. It is the class loader's responsibility to determine when and how classes can be added to a running Java environment.

When the VM needs a byte code for a particular class, it request from a class loader (The VM knows which class loader to ask byte code from) to find and load the byte code for that particular class. It depends to the class loader to use its own method for finding requested byte code files: It can load them from the local resources, fetch them across the Net using any protocol, or it can just create the byte code on the spot. This flexibility is not a security problem as long as the class loader is a trusted object by the creator of the byte code that is being loaded. The class loaders also define the namespaces that be seen by different classes and how those namespaces relate to each other.

Name-spaces contribute to security because you can place a shield between classes loaded into different name-spaces. Inside the Java Virtual Machine, classes in the same name-space can interact with one another directly. Classes in different name-spaces, however, can't even detect each other's presence unless you explicitly provide a mechanism that allows them to interact. (Bill Venners, 46)

2.4.1.1 Class Loader Types

There are two basic varieties of class loaders: Primordial Class Loader and Class Loader Objects. There is only one Primordial Class Loader, which is an essential part of each Java Virtual Machine. It cannot be overridden. The Primordial Class Loader is involved in bootstrapping [3]the Java environment. Since most VMs are written in C, it follows that the Primordial Class Loader is typically written in C. This special class loads trusted classes, usually from the local disk. (Gary McGraw and Edward W. Felten, 61)

Class Loader objects load classes that are not needed to bootstrap the VM into a running Java environment. The VM treats classes loaded through Class loader objects as untrusted by default. Class Loaders are objects just like any other java objects- they are written in Java, compiled into byte code, and loaded by the VM (with the help of some other class loaders) These Class Loaders give Java its dynamic loading capabilities. Figure 6 shows the inheritance hierarchy of Class Loader available in Java 2. (Gary McGraw and Edward W. Felten, 62)

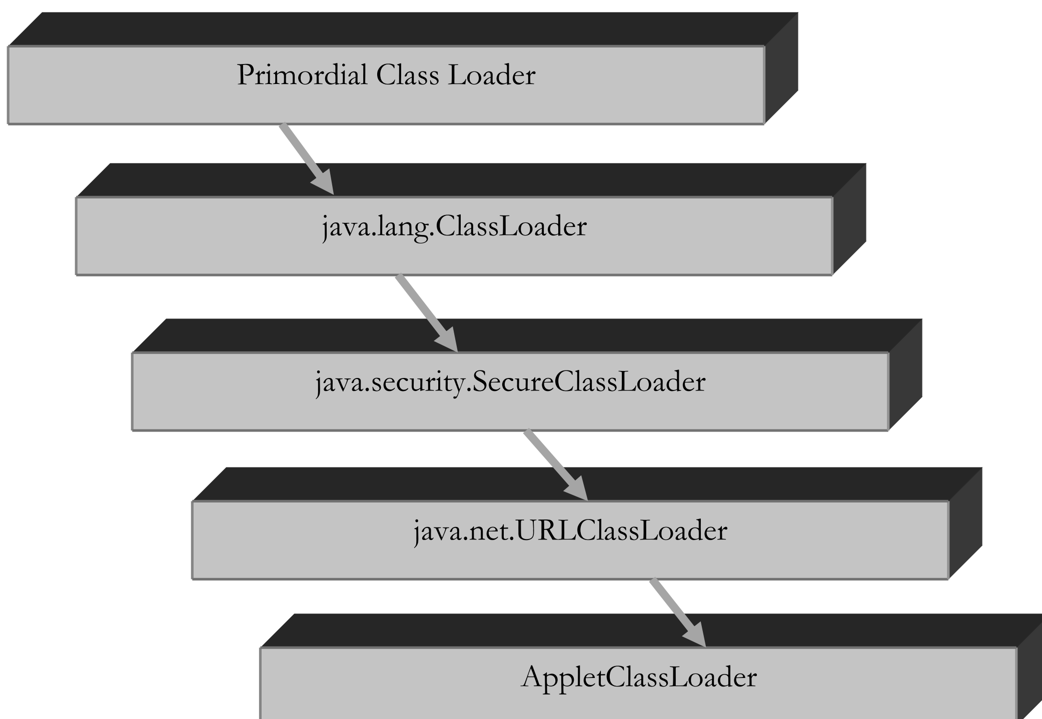

Figure 6 Class Loaders hierarchy (Gary McGraw & Edward W. Felten, 1999, 61)

[3] In computing, **bootstrapping** is a technique by which a simple computer program activates a more complicated system of programs. (www.wikipedia.org)

2.4.1.2 Class Loaders of a Web Browser

As indicated earlier that Class Loader is one of the customizable components of the Java Sandbox. It depends on the developer of a Web Browser, how to implement the class loader. A Web Browser is provided with two different class loaders: A class loader which has the responsibility to load class files over the network called Applet Class Loader and a class loader which loads class files from the local resources called System Class Loader. To every applet class loader assigns a private name-space and associated with the origin of the apple (The Java runtime can thus always determine the source of a given class and restrict the activities of the class accordingly (Lassi Lehto, quoted 20.01.2010), if the applets are from the same directory on a specific server class loader grouped them together in one name-space. To be able to maintain the trusted code separated from the untrusted code the Java runtime assigns a separate and unique name-space to all classes loaded from the local file system.

While browsing on a web page if an applet is encountered, a new Applet class Loader is created for that applet to load the applet if it is the first applet from the same place on this web page. In case an Applet class Loader was created for another applet from the same place on the same page, that class loader can be used to load the applet. If the execution of a java bytecode requires presence of a method, a field or a new class (because if there is no need for loading these class material, they are not loaded during loading and verification), the class Loader is invoked to fetch the class materials(method, field or class file). The Class Loader will first try to fetch the class materials from the local file system, only if the class material is not found in the local resources; it will fetch the class material across the network from origin of the applet.

Some criticism has been expressed because of the fact that every Web browser has its own implementation of the Java run time. Also critical security modules, like Applet Class Loader, have an individual implementation in each Java-enabled platform. Sun only provides a generic Class Loader to be extended as seen appropriate by each developer. The lack of central co-ordination in the security related features of the Java platform has been regarded as a major vulnerability of the system. (Lassi Lehto, quoted 20.01.2010)

2.4.2 The Bytecode Verifier

Despite the fact that Java compiler produces trustworthy class files and it makes sure that Java source code doesn't violate the safety rules, when an application such as the HotJava Browser imports a code fragment from anywhere, it doesn't actually know if code fragments follow Java language rules for safety: the code may not have been produced by a trustworthy Java compiler, but by some purposefully modified compiler for compromising the integrity of the Virtual Machine or the class file may have been change after the compilation process of Sun's compiler. In such a case, the Java run-time system on your machine can not trust the incoming bytecode stream. The loaded class files, loaded by a Web browsers are not source code, which then can be compiled, they are already compiled codes. For these reasons all the Java Virtual Machine implementations have Byte Code Verifier. Thus, the Java run-time system subjects class files to bytecode verification process to make sure if the class files are safe to use or execute. Figure 7 is an illustration of Java bytecode control by the Bytecode Verifier.

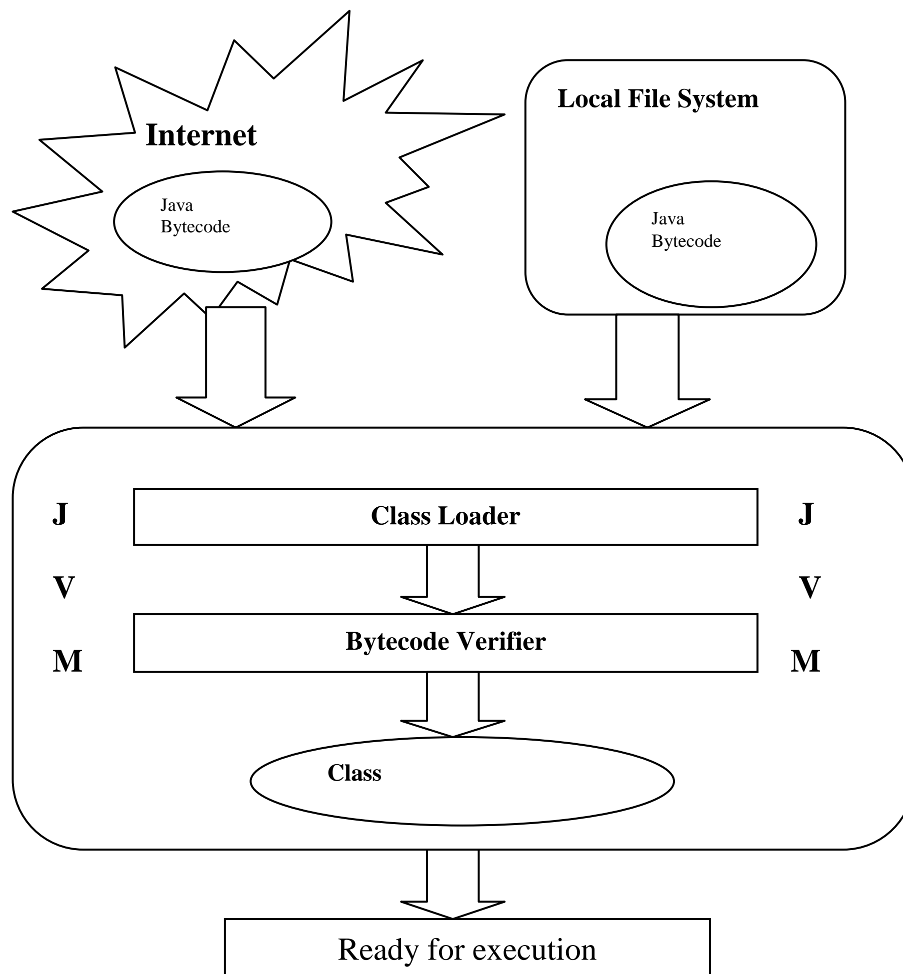

Figure 7 Control of Java bytecode

The illustration above shows the control of java Bytecode by the Bytecode Verifier .The important issue is that the Java class loader and the bytecode verifier make no assumptions about the primary source of the bytecode stream; the code may have come from the local system or from the internet which may have travelled halfway around the planet. The bytecode verifier ensures that code passed to the Java interpreter can be run without fear of breaking the Java interpreter. The Bytecode is not allowed to execute by any means until byte code passes through verification process or after it has passed the verifier's tests.

The Byte Code Verifier enforces the loaded applet to follow the strict security rules set by the Java specifications. Although the Java language enforces restrictions on class files, but the Verifier make sure that the class is not changed after compilations process. The Verifier is built in to the VM and cannot be accessed by Java programmers or Java users.

During the compilation process the Java source code is transformed to a platform independence machine code (byte code), which is not an executable code. The byte code then stored in a class file and it is loaded to the Web browser by the Applet class loader. The byte code then is checked by the Byte Code Verifier; only after the byte code passes the Byte Code Verifier without any problems then it will be ready for execution.

2.4.2.1 The Bytecode Verification Process

The Bytecode verifier starts checking the class file as early as possible. The performance takes place just after the class file is loaded. A class file is checked by the Bytecode verifier in four different levels or passes. The first three passes are internal checks of the Bytecode Verifier and the fourth pass is a runtime check or virtual pass where checking is done by appropriate Java Virtual Machine before execution. The verification passes are:

1. **Pass 1** occurs when the class is first read into the interpreter (Frank Yellin ,quoted 2.2.2010). In pass 1 in addition to verifying the Bytecode integrity the verifier makes sure that class file has a proper or basic format of a class file. Proper format means that every class file's first four bytes must contain the right magic number: 0XCAFEBABE and starts with the magic number. The purpose of magic numbers is to enable file parsers to easily recognize a certain type of file (Bill Venners, 46). All the attributes have a proper length is also checked in this pass. Proper length means that neither a class file is truncated nor it has extra bytes at the end.

2. **Pass 2** starts after the class file is linked[4]. In this pass verifier explores the class file format a bit more. This time verification is done without looking at the Bytecode.

   The checks perform by this pass include the following (Tim Lindholm & Frank Yellin, 1999, 141):

   - Ensuring that final classes are not subclassed and that final methods are not overridden.
   - Checking that every class (except object) has a direct supreclass.
   - Ensuring that the constant pool satisfies the document static constrains: for example, that each CONSTANT_Class_info structure in the constant pool contains in its name_index item a valid constant pool index for a CONSTANT_Utf8_info structure.
   - Checking that all filed references and method references in the constant pool have valid names, valid classes, and a valid type descriptor.

   Pass 2 checks only the well-formedness of the items. More detailed checking is done during Pass 3 and Pass 4.

3. **Pass 3** is the most complex pass of the Bytecode verification process. After the check for proper format and internal consistency in pass 1 and pass 2, the verifier turns its attention to the Bytecode. For better understanding of the Bytecode verifier, we should have some knowledge of bytecode and frames[5].

   The bytecode streams that represent Java method are a series of on byte instruction, called opcodes, each of which can be followed by one or more operands. The operands supply extra data needed by the Java Virtual Machine to execute the opcode instruction. The activity of executing bytecode, one opcode after another, constitutes a thread of execution inside the Java Virtual Machine. Each thread is awarded its own Java Stack, which is made up of discrete frames. Each method invocation gets its own

[4] Linking is the process of taking a binary form of a class or interface type and combining it into the runtime state of the Java Virtual Machine, so that it can be executed. (Li Gong, quoted 2.2.2010)
[5] A Java Virtual Machine *frame* is used to store data and partial results, as well as to perform dynamic linking, to return values for methods, and to dispatch exceptions. (Tim Lindholm & Frank Yellin, 71)

frame, a section of memory where it stores, among other things, local variables and intermediate results of computation. The part of the frame in which a method stores intermediate results is called the method's operand stack. An opcode and its (optional) operands may refer to the data stored on the operand stack or in the local variables of the method's frame. Thus, the virtual machine can use data on the operand stack, in the local variables, or both, in addition to any data stored as operands following an opcode when it executes the opcode. (Bill Venners, 51)

In this pass bytecodes of each method are verified. Data-flow analysis is performed on each method. The verifier ensures that at any given point in the program, no matter what code path is taken to reach that point (Tim Lindholm & Frank Yellin, 1999, 142):

- The stack is always the same size and contains the same types of objects.
- No register is accessed unless it is known to contain a value of the appropriate type.
- Methods are called with the appropriate arguments.
- Fields are modified with values of the appropriate type.
- All opcodes have appropriate type arguments on the stack and in the registers.
- Variables are properly initialized (Gary McGraw & Edward W. Felten, 57)

As indicated earlier, Pass 3 is the most complex pass of the Bytecode verification process. Hence the actual bytecod is verified in this pass.

The code of each method is verified independently. First the bytes that make up the code are broken up into a sequence of instruction, and the index into the code array of the start of each instruction is replaced in an array. The verifier then goes through the code a second time and parses the instructions. During this pass a data structure is built to hold information about each Java virtual machine instruction in the method. The operand, if any, of each instruction are checked to make sure they are valid. (Tim Lindholm & Frank Yellin, 1999, 143)

Pass 3 is a complex process that is carried out in two passes by Sun's Verifier. (Other vendors' Verifiers may behave differently.) The first pass identifies individual opcodes and stores them in a table. Once all opcodes are identified, the second pass parses each opcode's operands. During the second pass, a structure is built for each byte code

instruction. This structure is evaluated for syntactic correctness by checking that (Gary McGraw & Edward W. Felten, 57):

- Flow control related instructions branch to valid instructions.
- Local variable references are legal (associated with the proper method).
- Use of constant pool entries follows typing rules.
- Opcodes have the correct number of arguments.
- Exception handler start and end with valid instructions, and the start point comes before the end point.

These checks, along with data flow analysis that tracks behaviour of the operand stack and local variables, make up Pass 3. (Gary McGraw & Edward W. Felten, 57)

4. **Pass 4** occurs at run time. It performs some checks that were not done in pass 3 or delayed, for efficiency and implementation convenience, until the first time the code for the method is invoked. Some of these checks might have been impossible at verification time since some aspects of Java's type system cannot be statically checked, and some checks might have been deferred to runtime for implementation convenience (Gary McGraw & Edward W. Felten, 1999, 57). Unless it is necessary pass 3 avoid loading class file, but pass 4 makes sure that reference method or field exits in the given class if not they are loaded in this pass.

   For example, if a method invokes another method that returns an instance of class A, and that instance is assigned only to a filed of the same type, the verifier does not bother to check if the class A is actually exists. However, if it is assigned to a field of the type B, the definitions of both A and B must be loaded in to ensure that A is a subclass of B. (Tim Lindholm &Frank Yellin, 1999, 142)

   Pass 4 is a virtual pass whose checking is done by the appropriate Java Virtual Machine instructions. The first time an instruction that references a type is executed, the executing instruction does the following:

- Load in the definition of the referenced type if it has not already been loaded.
- Checks that the currently executing type is allowed to reference the type.

The first time an instruction invokes a method, or accesses or modifies a field, the executing instruction does the following (Tim Lindholm & Frank Yellin, 142- 143):

- Ensure that the referenced method or field exists in the given class.
- Checks that the referenced method or field has the indicated descriptor.
- Checks that the currently executing method has access to the referenced method or field.

Once byte code passes through verification, the following things are guaranteed (Gary McGraw and Edward W. Felten, 55):

- The class file has the correct format, including the magic number (0xCAFEBABE) and proper length
- Stacks will not be overflowed or underflowed
- Byte code instructions all have parameters of the correct type. For example, integers are always used as integers and nothing else
- No illegal data conversions (casts) occur. For example, treating an integer as a pointer is not allowed.
- Private, public, protected, and default accesses are legal. In other words, no improper access to restricted classes, interfaces, variables, and methods will be allowed.
- All register accesses and stores are valid.

After the excruciating test of the Bytecode verifier on the Bytecode, the Java interpreter can proceed, knowing that the code will run securely. Being sure about the secure execution of Bytecode makes the Java interpreter much faster, because it doesn't have to check anything. The interpreter can thus function at full speed without fear.

The important thing is not what the source code looked like or even what language it was written in, but what the byte code (the executable code) does. The Byte Code Verifier also insures that loaded class files have a proper internal structure. Verifier acts as the primary gatekeeper in the Java security model. In case it finds out something wrong in a class file, it throws an exception, loading ceases, and the class file never executes. This is obviously a

much more reasonable behaviour than running buggy or malicious code that crashes the VM. In order for the Verifier to succeed in its role as gatekeeper, the Java runtime system must be correctly implemented. Bugs in the runtime system will make byte code verification useless.

Sun's class file verifier is independent of any compiler. It should certify all code generated by Sun's compiler for the Java programming language; it should also certify code that other compilers can generate, as well as code that the current compiler could not possibly generate. Any class file that satisfies the structural criteria and static constrains will be certified by the verifier. (Tim Lindholm & Frank Yellin, 140-141)

The class file verifier is also independent of the Java programming language. Programs written in other languages can be compiled into the class file format, but will pass verification only if all the same constraints are satisfied. (Tim Lindholm, Frank &ellin, 141)

The Verifier disallows many obvious approaches to byte code manipulation. Nonetheless, a number of researchers have succeeded in creating byte code that should be illegal, but nevertheless passes the Verifier. The Princeton Secure Internet Programming team was the first to sneak attacks involving illegal byte code past the Verifiers**.** Mark LaDue, creator of the Hostile Applet Home Page (www.rstcorp.com/hostile-applets), performs a number of interesting experiments in which he creates byte code that does not play by the rules and yet passes verification. Other Java security researchers, most notably the Kimera group at the University of Washington, have discovered problems in commercial Verifiers. Their work places special emphasis on correct verification.

Java users and business people investigating the use of Java in their commercial enterprises often complain about the length of time it takes for a Java applet to get started running in a browser. E-commerce system designers paint start up delay as a business-side show stopper, citing the fact that consumers do not react well even to a 20-second delay in their shopping experience. Many people believe falsely that the main delay in starting applets is the time it takes to download the applet itself. But given a reasonably fast connection to the Internet, what takes the longest is not downloading the code, but verifying it. The inherent costs of verification fit the classic trade-off between functionality and security to a tee. As security researchers, we believe the security that byte code verification provides is well worth the slight delay. We also think it is possible to speed up the verification process so that its execution time is acceptable. (Gary McGraw and Edward W. Felten, 53).

2.4.3 The Security Manager

Security Manager is one the most important component of the Java Sandbox. Class Loader and bytecode Verifier are a part of Java Virtual Machine internal security: Class Loader prevents code loaded by different class loaders from interfering with one another inside the Java Virtual Machine and Verifier ensures that the code passed to Java interpreter can not break the Interpreter. Security Manager protects the resources external to the Java Virtual Machine. The security Manager defines the boundaries of the sandbox. The Java API (Application Programming Interface) refers to the security manager before it allows any access to the resource. The final decision, whether a particular operation is allowed or not, is done by the Security Manager. Because the Security Manager is one of the customizable components of the Sandbox, it lets you to define a custom security policy for an application.

The Java API enforces the custom security policy by asking the Security Manager for permission before it takes any action that is potentially unsafe. For each potentially unsafe action, the Security Manager has a method that defines whether that action is allowed by the Sandbox. Each method's name starts with *check*, so, for example, checkRead () defines whether a thread is allowed to read to a specified file, and checkWrite () defines whether a thread is allowed to write to a specified file. (Bill Venners, 58)

All methods in the Java API that can access resources outside of the Java environment call a Security Manager method to ask permission before doing anything. If the Security Manager method throws a SecurityException, the exception is thrown out of the calling method, and access to the resource is denied. The Security Manager class defines a number of methods for asking for permission to access specific resources. Each of these methods has a name that begins with the word "check." Table 2.1 shows the names of the check methods provided by the Security Manager class. (Mark Grand & Jonathan Knudsen, quoted 11.02.2010)

Table 1 The Check Methods of Security Manager (Mark Grand & Jonathan Knudsen, 1997)

Method Name	Permission
checkAccept()	To accept a network connection
checkAccess()	To modify a Thread or ThreadGroup
checkAwtEventQueueAccess()	To access the AWT event queue
checkConnect()	To establish a network connection or send a datagram
checkCreateClassLoader()	To create a ClassLoader object
checkDelete()	To delete a file
checkExec()	To call an external program
checkExit()	To stop the Java virtual machine and exit the Java environment
checkLink()	To dynamically link an external library into the Java environment
checkListen()	To listen for a network connection
checkMemberAccess()	To access the members of a class
checkMulticast()	To use a multicast connection
checkPackageAccess()	To access the classes in a package
checkPackageDefinition()	To define classes in a package
checkPrintJobAccess()	To initiate a print job request

checkPropertiesAccess()	To get or set the Properties object that defines all of the system properties
checkPropertyAccess()	To get or set a system property or Checks if the System properties can be accessed.
checkRead()	To read from a file or input stream
checkSecurityAccess()	To perform a security action
checkSetFactory()	To set a factory class that determines classes to be used for managing network connections and their content
checkSystemClipboardAccess()	To access the system clipboard
checkTopLevelWindow()	To create a top-level window on the screen
checkWrite()	To write to a file or output stream

The Security Manager class in the java.lang package is an abstract class that provides the programming interface and partial implementation for all Java security managers. Each Java application can have its own security manager object that acts as a full-time security guard.

Each VM can have only one Security Manager installed at a time, and once a Security Manager has been installed it cannot be uninstalled (except by restarting the VM). Java-enabled applications such as Web browsers install a Security Manager as part of their initialization, thus locking in the Security Manager before any potentially untrusted code or applets has a chance to run. (Gary McGraw and Edward W. Felten, 69)

Other kinds of applications (local applications) normally run without a security manager, unless the application itself defines one. To apply the same security policy to an application found on the local file system as to downloaded applets, either the user running the application must invoke the Java Virtual Machine with the new "-Djava.security.manager" command-line argument (which sets the value of the java.security.manager property), as in java -Djava.security.manager SomeApp or the application itself must call the setSecurityManager method in the java.lang.System class to install a security manager.

In a nutshell Java Security Manager prevents Java applets from accessing your local disk or local network. If an application or applet tries to access something outside of the sandbox without permission, Security Manager throws security exceptions. The checkPermission (java.security.Permission) method determines whether a request should be granted or an exception thrown. The job of the Security Manager is to keep track of who is allowed to do which dangerous operations. It is ultimately up to the security manager to determine whether a particular operation should be permitted or rejected. For example if a java program attempts to open a file, the security manager decides whether or not that operation should be permitted. If a java program wants to connect to a particular machine on the network, it must first ask permission of the security manager.

It's the security manager which prevents applets from the reading and writing of files, prevents the deletion or creation of files and prevents access to other computers. The security manager really allows applets to do only four things: They can run, they can access the screen, they can accept input and they can connect back to their originating host.

Most of the activities that are regulated by a "check" method are as follows. The classes of the Java API check with the security manager before they (Bill Venners, 58):

- Accept a socket connection from a specified host and port number
- Modify a thread (change its priority, stop it, and so on)
- Open a socket connection to a specified host and port number
- Create a new class loader
- Delete a specified file
- Create a new process
- Cause the application to exit
- Load a dynamic library that contains native methods
- Wait for a connection on a specified local port number
- Load a class from a specified package (used by class loaders)
- Add a new class to a specified package (used by class loaders)
- Access or modify system properties
- Access a specified system property
- Read from a specified file
- Write to a specified file

Because the Java API always checks with the security manager before it performs any of the activities listed above, the Java API will not perform any action forbidden under the security policy established by the security manager. (Bill Venners, 59)

2.4.3.1 Security Manager and Access Controller

The coming of Access Controller into existence in Java 2 has brought a big change into the role of the Security Manager. Before the existence of the Access Controller the Security Manager was a self-reliance security component and had to rely on it own internal logic. Now (after the release of Java 2), the Security Manager defer to Access Controller on access-control decisions.

However, the security Manager has still remained an important component in the Java Security Architecture. For enforcing system security the Java API still calls the methods of Security Manager but most of these methods of the security Manager call the AccessController. Thus, without existence of the Security Manager the Java API can never call the AccessController. The reason behind these method calls is upward compatibility.

For example, if you wrote a 1.1-based security manager that implements your desired security policy, you can still use that security manage with Java 1.2, your program will run exactly the same as it used to. In this case, you needn't worry about policy files and code sources and secure class loaders- the security model that you've already encapsulated into your security manger will b e respected. (Scott Oak, 1998, 125)

Unless the access controller is called it will not be initialized, but if it is called directly for a program-specific resource, it will automatically initialize itself. By default access controller will not be used by any Java application, because Java applications run without Security Manager by default.

Even though newly written library code could enforce access-control decisions by directly calling *AccessController.checkPermission* ( ), this is not considered to be a good programming choice. *SecurityManager.checkPermission*( ) should always be the primary interface between programs and *AccessController.checkPermission* ( ), for several reasons (Marco Pistoia, Nataraj Nagaratnam, Larry Koved & Anthony Nadalin, 2004, 274):

- The AccessController implementation may change in future Java release. By sticking to the SecurityManager's methods, programmers know that their code will not break.

- The SecurityManager can be made active from the command line; the AccessController cannot. As a consequence, an application can be run without an active SecurityManager, but security would still be partially enforced if *AccessController.checkPermission*( ) were invoked directly. This could lead to inconsistent security enforcements.

- Once it has been set as the system's current SecurityManager, a SecurityManager instance becomes the unique SecurityManager for the entire JVM system. This instance can, however, be replaced with a different SecurityManager instance by programs that are authorized to do so. A change in the active SecurityManager may imply a change in the measure the security policy is enforced; a new SecurityManager could be, in spite of the security policy, stricter that than the previous one, by denying specific Permissions, or more relaxed, by allowing other Permission. If some library code calls *AccessController.CheckPermission*( ) directly, without using *SecurityManager.checkPermission*( ) as an interface, a change in the SecurityManager will inconsistently reflect in a corresponding change in the policy enforcement.

2.4.3.2 Security Manager of a Web Browser

Security Manager is one of the customizable components of the Java Sandbox. In a Web Browser Implementation of a Security Manager is much more critical than implementation of a Class Loader, because after all it is the Security Manager, which allows or disallows access to critical system resources. A Web Browser developer defines and implements a security policy for downloaded bytecodes. Thus, a Web Browser has a very important role in the security of a system.

3 Data Analysis

In this chapter according to the methodology I have chosen the data will be analyzed and presented. At first, a brief explanation of malicious attacks types will be presented. Then the analysis of Java Security model for local and remote code (applet) will be done in two parts. In the first part, the design of Java Security Model will be discussed. In the second part, the adequacy and validity of the Java Sandbox against types of attacks, from malicious applet or programs can occur, will be discussed.

3.1 Threats of Malicious Codes

Malicious code is unlike viruses (the execution of a virus requires a user) is an auto-executable internet program. A malicious code is any code which will allow unauthorized access to the system, destroy, modify or steal data, damage a system and do something without the intention of a system user. It may not give any visual signal of its existence or execution to the victim. It can be written as any types of auto-executable content such as Java Applets. Malicious code does not replicate like viruses does, but the damage it brings to the system is an immediate damage. The main aim of programmers who intend to use malicious code is to gain access to files in a system or computer. It is also not far from the truth that sometimes the attacker may try to erase the system logs after doing something bad to the system in order to hide his/her footprint. Thus, it makes it sometime almost impossible to find who did something wrong to your system.

Threats or attacks are classified into different numbers of categories in different computer security literature that I have come across. My intention is not to find out the correct number of categories, but rather to briefly explain some of them with the types of problems that can arise from them. Categories of attacks are: Integrity Attacks, Availability Attacks (it is also called as denial-of -service Attacks) and Secrecy Attacks (in some other places it is called Disclosure Attacks). It is not intended to provide the complete list of all the potential attacks, but rather to give a flavour of some sample problems that can arise from them.

- **Integrity Attacks**

The attacker tries to delete, alter or modify the files or a part of the system in unauthorized way. The attack could also include modification of the current memory in use and killing of the processes or threads.

For example a college student breaks into the collage administration system to raise her examination score, thus, compromising data integrity. An attacker might also try to erase system logs in order to hide his footprint. (Li Gong, Gary Ellison & Mary Dageforde, 2003, 3).

- **Secrecy Attacks**

The attacker attempts to send personal or company files or mailing information of a host machine to someone or competitor over the network. The attacker may also try to steal confidential information such as password, medical records, email logs and etc. The methods of attack vary, from bribing the security guard to exploiting a security hole in the system or a weakness in the cryptography of algorithm (Li Gong, Gary Ellison & Mary Dageforde, 2003, 3).

- **Availability Attacks**

The attacker tries to prevent the intended user from accessing a computer resource by disrupting the normal operation of a system. These attacks are also called as denial-of-service attacks. The attacker or java applet can accomplish these attacks in any number of ways for example by allocating all available memory, by creating thousands of windows or by creating high priority processes or threads.

For example, bombarding a machine with a large number of IP (Internet Protocol) packets can effectively isolate the machine from rest of the network. A cyber terrorist might attempt to bring down the national power grid or cause traffic accidents by compromising the computer-operated control system. (Li Gong, Gary Ellison & Mary Dageforde, 2003, 3).

These categories of attacks are somehow related to each other in a very complicated way, because a successful attack in one category could assist attacker to attempt another attack. For example if an attacker could get a password by performing the secrecy attack, by using the password the attacker could access and alter the system which is an integrity attack and modifying system resources could lead to an availability attack or denial-of- service attack.

3.2 Design of Java Security Model

One of the most important parts of creating a safe environment for auto-executable content to be executed in is to identify the system resource and then to provide a simple way of granting limited access to these resources. Because, it would be impossible for a system, with a security policy in which an unknown program will have full access to all system resources, to survive in today's world.

Designers of the Java have been well aware of executable content or mobile code risks. Thus, while developing Java security was considered as part of the Java's design. It is important to know that unlike many computer and network systems (security through obscurity), Java security model with all the details was published when the model was released. The sandbox model design specification and full resource implementation was included in the publication. The purpose has been to encourage security researchers to examine the model and report the security flaws before it becomes a victim of an attack. When Java technology is available within an operating system, such as Solaris, Linux, or Windows, its presence does not alter the basic security characteristics of the underlying system (Li Gong, Gary Ellison and Mary Dageforde, 2003, 21).

I should remind you that Java security architecture is intended for both local and remote cod (applet) that are executed by Java enabled Web Browser. But my intention here is discussion of Java Security Mechanism for remote code or auto-executable content.

The design of Sandbox for remote codes (applets) is to have a Java enabled Web Browser. The embedded Java security model in Web Browsers allows a user to download and run applets (remote code or auto-executable content) either from the World Wide Web or from an intranet without risking user's machine. The Java Security mechanism confines the applet's

action to its sandbox: a dedicated area to that applet inside the Web Browser. The applet is allowed to do anything inside its sandbox, but not outside of the sandbox such reading, writing or modifying of data.

3.2.1 Is Java Bytecode Enhancing Security?

Bytecode is an intermediate code or language of a Java program, which is generated by Java compiler from the native code for the Java Virtual Machine. A Java program is compiled only once and it is transformed to bytecode which is a machine independent code, thus it can be run by Java interpreter on any platform or machine that has a Java interpreter.

There are a number of possible reasons why the Java developer has chosen using bytecode instead of source code or native code. Reasons include portability, security and size, but the main concern has been security.

Some criticism exists on portability, security and speed of the bytecode. If there is a possibility of portability with the java source code then why bytecode should be used instead. And if the Java source code was used instead of the bytecode the system would have been more secure. The speed of the process would have been also faster if the Java source code was used. Some people believe there is a delay in downloading the bytecode and some other believe that the delay is in verification process of the bytecode (if the Java source code was used instead there were no verification process, because the Java compiler is already trusted).

One of the problems with the Java Source code is the Java compiler. A compiler is a complex program. In today's world there are many computer platforms or computer types and day by day increasing in numbers, writing a new compiler for new platforms would need much effort On the other hand, an Interpreter is relatively small and simple program. Thus, writing an interpreter for a new platform wouldn't need much effort. It is important to know that there is no relative connection between Java and Java bytecode. Programs that are written in other languages can be compiled into Java bytecode.

For security reason the Java security model prevent loaded applets from taking any destructive action by disallowing any potentially dangerous operation of the applets. Before allowing the code to be run the verifier examines the code for any attempt of bypassing security. The verifier checks the internal consistency of the data: code that manipulates a data item as an

integer at one stage and then tries to use it as a pointer later will be caught and prevented from executing (Hank Shiffman, quoted 12.02.2010). Since the Java language does not allow pointer arithmetic, so it is not possible to do what is just described with the Java code. It is also not far from the truth that someone could write malicious or destructive bytecode by using a hexadecimal editor or even by some purposefully modified compiler. In general analyzing machine code of a program before execution and determining whether the program does anything destructive would be impossible, because there are some programming-tricks in which the destructive action of the program may appear while executing. For example self-modifying code is a code which changes its own instructions during the execution time. But Java byte code was designed for this kind of validation: it doesn't have the instructions a malicious programmer would use to hide their assault (Hank Shiffman, quoted 12.02.2010).

Even though there might be slight delay in either downloading or verifying of the bytecode, but the security bytecode provides is worth the slight delay. As a security researcher if I had made a simple questioner and asking people only one question: do you prefer security to speed or speed to security? There is no room for doubt that I would have got all the answer in favour of the security.

3.2.2 Turning off Java

Another very useful and important security feature of Java design is that it allows users to disable or switch off Java support in Web Browsers. Whenever users try to surf a web page, they don't fully trust or they don't know about the security problems related to that specific page, it is a recommendation to disable or switch off Java support in the Web Brower before surfing the site. This is one way of protecting web surfers from malicious applets, but it is not a satisfactory solution. Disabling Java could also eliminate some of very annoying popup windows of advertisements, which exploit the default settings of Java permissions. It is also recommended to switch off Java support in Web Browser, whenever there is a new security hole found in Java system until the bug is fixed. Thus, it is an effective way of protecting user's system or machine: the effectiveness of disabling Java support in a Web Brower is that the Java applets will not be fetched from across the network and install on the user's machine. Therefore a malicious code or applet could not cause any harm to the users' machine.

On the other hand switching off Java support in a Web Browser has some important side-effects and will deprive companies and private users of many of the advantages of the Java

platform. Although it is an effective way of protection against malicious codes or applets, but switching off Java support is not an effective solution for someone who uses Java technology in their intranets. For example some companies find Java applet technology very useful and use this technology within the intranet of the company for different purposes. These apples are written and use within the company, thus, they are trusted applets. But a great number of applets exist out there on the Internet, written with intentions that might be a sign of danger for the company. Consider if a company thinks that there are potential risks associated with the Java applets and decides to deploy an effective method against them, unfortunately this deployment would also affect useful Java technology altogether.

Fortunately some solutions exist out there for what was just described above. For example the solution from the Princeton's Secure Internet Programming team of Princeton University called Java Filter.

The Java Filter gives effective protection against unwanted execution of Java applets, without having to switch off Java support of your browser. For example, you could only allow applets from within your firewall onto your computer. (The Java Filter, quoted 01.03.2010)

3.3 Implementation of Java Security Model

In providing a safe environment for an auto- executable content, it is necessary to implement restrictions on host's system resources. In order to analyze the effectiveness of Java Security Model implementation in an effective way, it is necessary to find out whether there are adequate methods in the security model for controlling host's system resources such as file system, network, input devices (Mouse, Keyboard, Web Cam, etc), output device (Monitor, Sound Card, Speakers, Video Card, etc), random memory, and etc. No restriction on these system resources means there is a possibility for all types of attacks mentioned in section 3.1. The list of system resource just mentioned is not intended to be complete in terms of all possible attacks, but to give an example of types of attacks associated with every system resource. For example all types of attacks mentioned in section 3.1 are associated with file system. It means if there is no restriction on file system there is a possibility for all types of attacks. And if there is no restriction on network the secrecy attack might happen and as mentioned in section 3.1 these categories of attacks are somehow related to each other: A successful attack in one category could assist attacker to attempt another attack.

To understand the theory well, it is necessary to look at the some of the host's machine system resources in more detail with the types of problems associated with the availability of these resources and to loot at the Java Security Model if there are adequate methods for controlling these resources of the host machine. The resource names is not intended to be complete, but to provide you some example of the problems related to these resources and Java Security Model solutions for these problems.

File System - A policy with no restriction to the file system on all programs would be unacceptable to everyone, because then there is no protection of users' privacy. Attacking a system would not need much effort then. For example, if applets (remote code) had been give the permission to access the files on a user's machine, then writing an applet to destroy all the data on user's machine that downloaded the applet, would not be so difficult. Thus, all types of attacks mentioned in section 3.1 could be easily happened to a system.

Java Security Model does not give access permission for untrusted applets to the file system on host system or machine. Security Manager of Java Security Model provides specific checks for read and write to a particular file. Thus, access to the file system on client machine is well protected. Applets therefore cannot read, write, or alter the properties (*checkpropertyAccess* ( ) method of Security Manager checks if the system property can be accessed) of any files on a user system. The latest release of Java Security Model allows fairly easy implementing of extensible access controls.

Network Access - Without any restriction and security control on a system's network the system might be subjected to an attack. Though there are many different types of network attacks a few of which more commonly performed. Secrecy attack category of section 3.1 of this paper could be a result of a malicious code getting access to the network.

All the socket operations are managed by the Security Manager of the Java Security Model. It controls all access to operating system and processes. Security Manager has check methods like *checkConnect* ( ) and *checkAccept* ( ) for both creating and accepting of the sockets. Thus, by default security mode untrusted applets are not allowed to make network connections except to the host from which they were downloaded. It is important to remember that allowing access controls are flexible methods in Java Security Model.

System Properties - Applets are not allowed to access any sensitive properties of the system such as user account name or the current working directory. Despite many restrictions on applets, they can access or read specific set of system properties like the operating-system name and Java version number.

Input Devices - An applet can only access the keystrokes or mouse clicks of the user when the applet's window has been selected. Currently other input devices are not supported, although one would assume that any access to devices such as camera's or microphones would be through a Java library which could add security checks. Currently there are no explicit security checks involving input. (Joseph A. Bank, quoted 03.03.2010)

After analysis of Java security mechanism for controlling system resources of a host machine, it would be necessary to look at the adequacy of Java Security Model against malicious threads mentioned in section 3.1.

Integrity attacks - Java Sandbox approach is an important approach in preventing integrity violations. Access Control capabilities provided by Java

Security Model can easily prevents each of integrity attacks mentioned in section 3.1.

In the past some flaws in Java enabled browsers had been reported: attack applets gaining full access to the client system resources. All the flaws have been tested in the labs, but there had been no confirmed report of integrity attacks in the real Web environment. In most cases the reason behind the flaw had been implementing of some features of Java Security Model in an improper way in some versions of Java enabled Web browsers. All the flaws were fixed in the next release of those Web browsers.

Secrecy attacks - To prevent confidentiality violations the Java Sandbox approach is one of the best approaches. Access Control capabilities provided by Java Security Model can easily prevents most of secrecy attack mentioned in section 3.1. Restrictions on socket connection to other site on the internet by Java Security Manager prevent some of the Secrecy Attacks.

Availability attacks - Preventing of availability attacks in a way is much complicated. Java does have the ability to place some control on the creation of high priority threads (Joseph A. Bank, quoted 05.03.2010). Both preventing and not preventing has their side effects. A security policy which defines the amount of memory an applet can use will not work, because some trusted applet might need more memory than defined. Also if a policy prevents creation of more than 10 windows, this might seem as arbitrary restriction for some users. It is interesting to see how availability attack happens.

When Security manager finally give permission to a piece of code to access a system resource, the code will get full access privileges to the resource. To what extend the code uses the source is not

controlled by Security Manager. Consider if Security manager give only write permission to a specific directory for an applet (maybe untrusted applet). No mechanism exist to restrict the number of files the applet can create in the directory or how big in size a file can be or how long it can keep a file open for writing. The result would be denial-of-service which at least forces the victim to restart the system.

3.3.1 Integrating Java Sandbox into a Web Browser

To utilize remote codes (applets), a Web Browser with flexible and trustworthy security architecture is needed. By allowing the writer of a Web Browser to customize both Security Manager and Class Loader the Java Security Architecture has provided a flexible Security Architecture. From the original Sandbox model to policy-based model by refining the security model, Java has become a flexible and trustworthy Security Architecture. This has made the Java Security Model one of the most popular Security Architecture in the World.

By subclassing the Security Manager and certain methods, the developer of a Web Browser defines and implements a security policy, and then the new version of the Security Manager can be installed as the Browser or System Security Manager. Since the Security policy of a Java enabled Web browser is implemented by subclassing the Java Security Manager, it is of concern to end users of a Web browser that the Web browser's version of the Security Manager is implemented correctly. Thus, implementing of a Web browser's Security Manager is more critical than implementing of its Class Loader.

It is important to remember that despite being one of the most important components of the system, the Security Manager is not installed when the Java Runtime Environment (JRE) starts up. Whenever a Java enabled Web Brower encounters an untrusted code, it is the responsibility of the Web Browser to install Security Manager before allowing the code to be executed, because access to system's critical resource is controlled by the Security Manager. In case the Security Manager is not installed by the Web browser, while encountering an applet, the applet (remote code) will get the same access permissions as the local application or trusted codes.

Some criticism has been expressed because of the fact that the Java Security Model protects the system resources by providing a sandbox for every applet, but the model or Java Virtual Machine (JVM) does not protect the Web browser itself from being attack by malicious code. The malicious code might interact directly with the Web browser. It means if the malicious applet exploits bugs in the Java enabled Web browser, the applet might be able to avoid the Java Sandbox mechanisms and gain access to the system resources. In the extreme, the applet might replace the original Java Virtual Machine with rogue copy. If this is accomplished by a malicious applet, then all the remote codes will be able to gain access to the system resources without any restriction of the Java Sandbox. There had been reports and examples of attacks that exploit bugs in the Java enabled Web browsers instead of the Java Security Model. Thus, this means that the Java Sandbox security mechanisms built into the Java Virtual Machine is avoidable if the Java enabled Web browser is not bug free.

4 Conclusions

In this chapter, based upon the analysis derived in the pervious chapter, I will present the findings and conclusion. Findings will be presented in a general discussion and research questions. Then at the end of this chapter I will draw conclusion.

4.1 General Discussion

Despite Java and Java applets has increased security risks to the connected global Internet for example fraud, identity theft and even fear of users' system destruction, then what is the reason that Java is still desirable in the first place. There are many reasons behind; one of the most important reasons is that in most cases Java users probably finds the Java's advantages outweigh the risks. But for a system such as military base system where security is of paramount importance, it is recommended that it should be completely disconnected from the global Internet or has a separate network connected to the global Internet. Another reason is that Java, beside fairly good security, has also provided increased power and flexibility. For example Java applets have animated the internet and made the websites more dynamic and entertaining. Java applets provide functionalities in a Web browser that can not be perform by HTML such as games and chat applications. Java applet has also brought improvement in education systems for example improvement in practices of teaching and learning[6]. Some other advantages of applets include portability, speed, independent of browser, loaded when needed and no installation needed.

To restrict security breaches and combat vast numbers of security issues involved with the Java applets some of which is mentioned in the preceding paragraph and section 3.1 of this paper, Java has provided a security model with access controller, easily configurable security policy and easily extensible access control structure. Java has named the security model as Sandbox, a model which allows the users to take advantages of ad-hoc applications while it surrounds the running application to protect users' system from malicious codes.

[6] Java Applets in Education, URL: http://www.irt.org/articles/js151/index.htm, quoted 10.03.2010

The Java Security Model might not prevent some (minor things from my point of view) annoying behaviours or annoyance attacks for instance an applet may be able to play loud sound, reload page in new window, display something users don't like to look at and etc, but it is unable to steal users confidential information which it the most important thing from a privacy and system security point of view.

4.2 Research Questions

The main objective of this research (to summarize the research questions) has been to research, study and present the security edges (security advantages over others), important features and benefits of the Java sandbox model.

The findings reveal that Java applet does increase the security risks and the developers knew for well the risks, therefore they have provided the Sandbox model, a fairly well security mechanisms to protect users' confidential information from stealing, deleting, modifying and any other destructive external threads.

It is also revealed that some criticism has been expressed about the risks involved with the Java applets. But if we consider our normal life (by normal life I mean, no computer systems) situations, when benefits of something outweigh the risks of it people normally take the risks, that's what the Java developers did.

4.3 Conclusion

For the sake of a system security both the Web browser and the Java Security Model into the Web browser must be implemented properly. A bug in either of these can result a security hole that malicious code could exploit. For the Java Sandbox security mechanisms to work properly there many important points to be taken in consideration such as:

- The Web browser should be bug-free so that it can resist all attacks

- The Class Loader of the Java Sandbox should put all the codes into the right name spaces so that the untrusted codes cannot interact with each other

- The Class Loader should prevent interfering of the untrusted code with the trusted codes.
- The bytecode verifier should ensure that the bytecode follows the security rules set by Java specification

- The bytecode verifier should ensure that the code passed to the Java interpreter can be run without fear of breaking the Java interpreter.

- The Java API (Application Programming Interface) should refer to Security Manager in order to enforce the custom security policy by asking the Security Manager for permission before it takes any action that might be potentially unsafe

Lack of co-operation or very little co-operation between Web browser developers is a reason behind the flaws found in the Java enabled Web browsers. Flaws found in the Java enabled Web browsers were either because of implementation error or flaw in the Web browser itself not in the Java Sandbox Model. The result of some security organizations that had tested Java enabled Web browsers in the past had showed that a flaw found in one browser but not in the other because different Java enabled Web browsers treat threads differently. Personally, I think moving towards open source would be one of fairly good solutions to these kinds of problems.

I would personally recommend co-operation between Web browsers developers and some Internet security organizations for instance Princeton Secure Internet Programming team. The flaws found and solutions provided by these security organizations should be taken in consideration for the future versions of the Web browsers.

Bibliography

Almut Hezog, 2007, Usable Security Policies for Runtime Environments, Thesis, Linköping Studies in Science and Technology, Dissertation No. 1075, URL: http://www.ida.liu.se/~almhe/thesis/tek-dr-1075-full-version.pdf, Quoted 11.02.2010.

Aritma Developer, URL: http://www.artima.com, Quoted 15.9.2009

Aladdin Knowledge Systems, URL: http://www.aladdin.com, Quoted 23.02.2010

Beyond The Basics, Virginia Polytechnic Institute & State University, URL: http://ei.cs.vt.edu/~wwwbtb/book/index.html, Quoted 01.03.2010

Bill Venners, 1998, Inside the Java virtual machine, McGraw-Hill New York(NY)

Bill Venners, Java's Security Model and Built-In Safety Features, 1997, URL: http://www.artima.com/underthehood/overviewsecurity4.html, Quoted 5.9.2009

David Reilly, Inside Java: The Java Virtual Machine, 2006, URL: http://www.javacoffeebreak.com/articles/inside_java/insidejava-jan99.html, Quoted: 17.8.2009

Elliotte Rusty Harold, 2000, The Java Developer's Resource, URL: http://www.ibiblio.org/java/books/jdr/chapters/index.html, Quoted 20.8.2009

Frank Yellin , Low Level Security in Java, URL:
http://www.w3.org/Conferences/WWW4/Papers/197/40.html#1,
Quoted 23.02.2010

Gary McGraw and Edward W. Felten, 1999, Securing Java, Getting Down to Business with Mobile Code, Second Edition, John Wiley & Sons, Inc.

Java Security Architecture, 2002, URL:
http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-specTOC.fm.html,
Quoted 20.8.2009

Java Security API Overview,
http://journals.ecs.soton.ac.uk/java/tutorial/security1.1/overview/index.html,
Quoted, 25.09.2009

Java coffee break, 2006, free guide to learn the Java programming language, URL:
http://www.javacoffeebreak.com, Quoted: 17.8.2009.

Jay Heiser, Java Security Mechanisms, System self-protection against the invader applets, 2004, URL: http://www2.sys-con.com/itsg/virtualcd/java/archives/0203/heiser/index.html, Quoted 24.8.2009

Joseph A. Bank, Java Security, URL:
http://groups.csail.mit.edu/mac/users/jbank/javapaper/javapaper.html#resources,
Quoted, 10.02.2010

Hank Shiffman, Stackoverflow, programming Q&A website, URL: http://stackoverflow.com/, Quoted, 12.02.2010

Lassi Lehto, 1997, Java security, Thesis, Helsinki University of Technology, URL: http://www.tml.tkk.fi/Opinnot/Tik-110.551/1997/Java-Security.html#ref7, Quoted 01.03.2010

Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers, 1997, Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2

Li Gong, Gary Ellison and Mary Dageforde, 2003, Inside Java 2 Platform Security, Second Edition, Google books, URL: http://books.google.com, Quoted 18.02.2010.

Li Gong, JavaTM 2 platform Security Architecture, Version 1.2, URL: http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html, Quoted, 10.02.2010

Mikael Lagerkvist, 2005, Machine Assisted Reasoning for Multi-Threaded Java Bytecode, Thesis

Mark Grand and Jonathan Knudsen, 1997, Java Fundamental Classes Reference, First Edition, URL: http://docstore.mik.ua/orelly/java/fclass/index.htm, Quoted 11.02.2010.

Marco Pistoia, Nataraj Nagaratnam, Larry Koved, Anthony Nadalin, Enterprise Java™ Security, 2004, illustrated Edition, Addison-Wesley, Google books ,URL: http://books.google.com, quoted 20.02.2010.

Mary Dageforde, Security in JDK1.2, URL: http://pirlwww.lpl.arizona.edu/resources/guide/software/Java_tutorial/security1.2/overview/index.html, Quoted, 10.01.2010

Michael Girdley, Kathryn A. Jones, web programming with JavaTM, URL:http://docs.rinet.ru/JWP/index.htm , Quoted, 10.11.2009

Nitin Nanda, Sunil Kumar, Breaking the Sandbox Barrier, Part 1, 2001, URL: http://www.developer.com/java/article.php/934031/Breaking-the-Sandbox-Barrier-Part-1.htm, Quoted, 10.11.2009

Qun Zhong, Nigel Edwards, Security in the Large: Is Java's Sandbox Scalable, 1998, URL: http://www.hpl.hp.com/techreports/98/HPL-98-79.pdf, Quoted 12.9.2009

Raghavan N. Srinivas, Java security evolution and concepts, Part 1: Security nuts and bolts, 200o, URL:http://www.javaworld.com/jw-04-2000/jw-0428-security.html?page=1, Quoted, 6.10.2009

Scott Oaks, 1998, Java Security, First Edition, O'Reilly & Associates, USA

Symantec Report on Rogue Security Software, 2009, URL:http://eval.symantec.com/mktginfo/enterprise/white_papers/b-symc_report_on_rogue_security_software_WP_20100385.en-us.pdf, Quoted 01.03.2010

The Last Stage of Delirium Research Group, Poland, Java and Java Virtual Machine security vulnerabilites and their exploitation techniques, 2002, URL:

http://www.blackhat.com/presentations/bh-asia-02/LSD/bh-asia-02-lsd.pdf, Quoted, 12.10.2009

The Java Filter, URL: http://www.cs.princeton.edu/sip/JavaFilter/,
Quoted 01.03.2010

Tim Lindholm, Frank Yellin, 1999, The Java Virtual Machine Specification, Second Edition, Addison- Wesley, Reading (MA)

Wikipedia, URL: http://en.wikipedia.org, Quoted 23.02.2010