IJRIT International Journal of Research in Information Technology, Volume 1, Issue 11, November, 2013, Pg. 524-532

International Journal of Research in Information Technology (IJRIT)

www.ijrit.com ISSN 2001-5569

JAVA VIRTUAL MACHINE AND JAVA SECURITY ARCHITECTURE

[1]Latika Kapur , [2]Kanika Ahuja , [3]Abhay Kaul

[1]Dronacharya College of Engineering
Gurgaon
latika992@gmail.com

[2] Dronacharya College of Engineering
Gurgaon
kanika.ahuja14@yahoo.co.in

[3] Dronacharya College of Engineering
Gurgaon
abhay.kaul@gmail.com

Abstract

A virtual machine (VM) is a software implementation of a machine (i.e. a computer) that executes programs like a physical machine. Originally, Java was designed to run based on a virtual machine separated from a physical machine for implementing WORA (Write Once Run Anywhere), although this goal has been mostly forgotten. Therefore, the JVM runs on all kinds of hardware to execute the Java Bytecode without changing the Java execution code.

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. 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

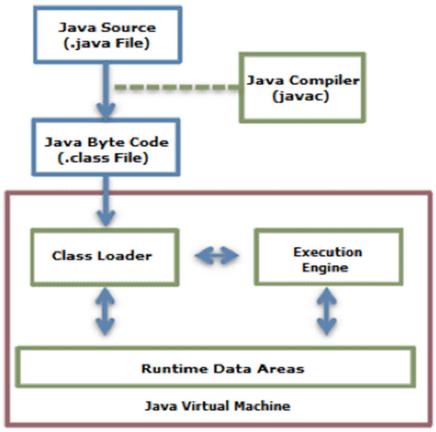
1. INTRODUCTION

Java virtual machine has following features:

Stack-based virtual machine: The most popular computer architectures such as Intel x86 Architecture and ARM Architecture run based on a *register*. However, *JVM runs based on a stack*.

Symbolic reference: All types (class and interface) except for primitive data types are referred to through symbolic reference, instead of through explicit memory address-based reference.
Garbage collection: A class instance is explicitly created by the user code and automatically destroyed by garbage collection.
Guarantees platform independence by clearly defining the primitive data type: A traditional language such as C/C++ has different int type size according to the platform. The JVM clearly defines the primitive data type to maintain its compatibility and guarantee platform independence.
Network byte order: The Java class file uses the network byte order. To maintain platform independence between the little endian used by Intel x86 Architecture and the big endian used by the RISC Series Architecture, a fixed byte

2. JVM Architecture



The code written in Java is executed by following the process shown in the figure below. A class loader loads the compiled Java Bytecode to the Runtime Data Areas, and the execution engine executes the Java Bytecode.

2.1 Class Loader

Java provides a dynamic load feature; it loads and links the class when it refers to a class for the first time at runtime, not compile time. JVM's class loader executes the dynamic load. The features of Java class loader are as follows:
Hierarchical Structure: Class loaders in Java are organized into a hierarchy with a parent-child relationship. The Bootstrap Class Loader is the parent of all class loaders.
Delegation mode: Based on the hierarchical structure, load is delegated between class loaders. When a class is loaded, the parent class loader is checked to determine whether or not the class is in the parent class loader. If the upper class loader has the class, the class is used. If not, the class loader requested for loading loads the class.
Visibility limit: A child class loader can find the class in the parent class loader; however, a parent class loader cannot find the class in the child class loader.
Unload is not allowed: A class loader can load a class but cannot unload it. Instead of unloading, the current class loader can be deleted, and a new class loader can be created.
Each class loader has its namespace that stores the loaded classes. When a class loader loads a class, it searches the class based on FQCN (Fully Qualified Class Name) stored in the namespace to check whether or not the class has been already loaded. Even if the class has an identical FQCN but a different namespace, it is regarded as a different class. A different namespace means that the class has been loaded by another class loader.
The following figure illustrates the class loader delegation model.

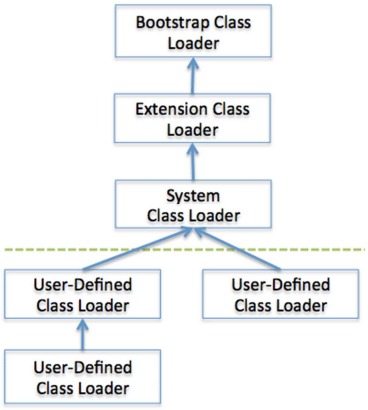


Figure 2: Class Loader Delegation Model.

When a class loader is requested for class load, it checks whether or not the class exists in the class loader cache, the parent class loader, and itself, in the order listed. In short, it checks whether or not the class has been loaded in the class loader cache. If not, it checks the parent class loader. If the class is not found in the bootstrap class loader, the requested class loader searches for the class in the file system.

Bootstrap class loader: This is created when running the JVM. It loads Java APIs, including object classes. Unlike other class loaders, it is implemented in native code instead of Java.
Extension class loader: It loads the extension classes excluding the basic Java APIs. It also loads various security extension functions.
System class loader: If the bootstrap class loader and the extension class loader load the JVM components, the system class loader loads the application classes. It loads the class in the $CLASSPATH specified by the user.
User-defined class loader: This is a class loader that an application user directly creates on the code.

Frameworks such as Web application server (WAS) use it to make Web applications and enterprise applications run independently. In other words, this guarantees the independence of applications through class loader delegation model. Such a WAS class loader structure uses a hierarchical structure that is slightly different for each WAS vendor.

If a class loader finds an unloaded class, the class is loaded and linked by following the process illustrated below.

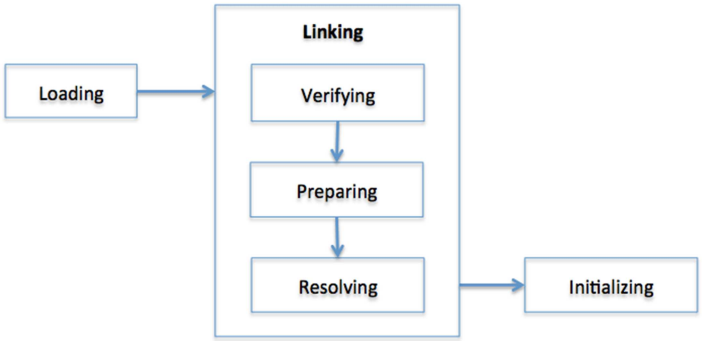


Figure 3: Class Load Stage.

2.2 Each stage is described as follows.:

Loading: A class is obtained from a file and loaded to the JVM memory.

Verifying: Check whether or not the read class is configured as described in the Java Language Specification and JVM specifications. This is the most complicated test process of the class load processes, and takes the longest time. Most cases of the JVM TCK test cases are to test whether or not a verification error occurs by loading wrong classes.

- **Preparing**: Prepare a data structure that assigns the memory required by classes and indicates the fields, methods, and interfaces defined in the class.
- **Resolving**: Change all symbolic references in the constant pool of the class to direct references.
- **Initializing**: Initialize the class variables to proper values. Execute the static initializers and initialize the static fields to the configured values.

The JVM specification defines the tasks. However, it allows flexible application of the execution time.

2.3 Runtime Data Areas

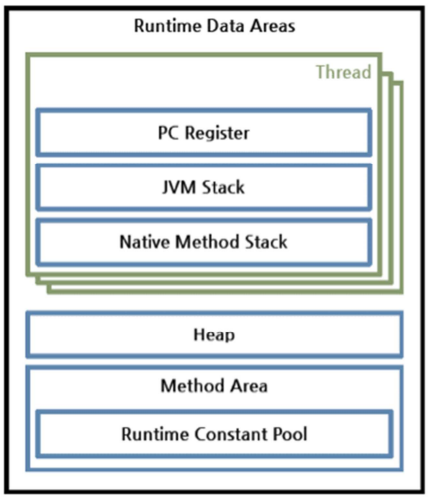


Figure 4: Runtime Data Areas Configuration.

Runtime Data Areas are the memory areas assigned when the JVM program runs on the OS. The runtime data areas can be divided into 6 areas. Of the six, one PC Register, JVM Stack, and Native Method Stack are created for one thread. Heap, Method Area, and Runtime Constant Pool are shared by all threads.

- **PC register**: One PC (Program Counter) register exists for one thread, and is created when the thread starts. PC register has the address of a JVM instruction being executed now.
- **JVM stack**: One JVM stack exists for one thread, and is created when the thread starts. It is a stack that saves the struct (Stack Frame). The JVM just pushes or pops the stack frame to the JVM stack. If any exception occurs, each line of the stack trace shown as a method such as printStackTrace() expresses one stack frame.

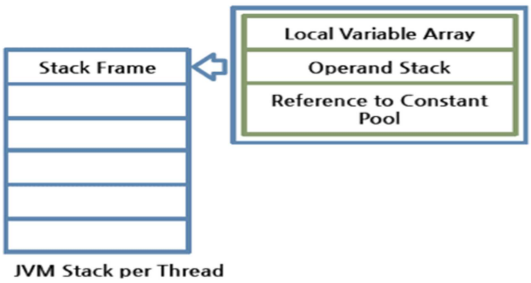


Figure 5: JVM Stack Configuration.

− Stack frame: One stack frame is created whenever a method is executed in the JVM, and the stack frame is added to the JVM stack of the thread. When the method is ended, the stack frame is removed. Each stack frame has the reference for local variable array, Operand stack, and runtime constant pool of a class where the method being executed belongs. The size of local variable array and Operand stack is determined while compiling. Therefore, the size of stack frame is fixed according to the method.

− Local variable array: It has an index starting from 0. 0 is the reference of a class instance where the method belongs. From 1, the parameters sent to the method are saved. After the method parameters, the local variables of the method are saved.

− Operand stack: An actual workspace of a method. Each method exchanges data between the Operand stack and the local variable array, and pushes or pops other method invoke results. The necessary size of the Operand stack space can be determined during compiling. Therefore, the size of the Operand stack can also be determined during compiling.

- **Native method stack**: A stack for native code written in a language other than Java. In other words, it is a stack used to execute C/C++ codes invoked through JNI (Java Native Interface). According to the language, a C stack or C++ stack is created.
- **Method area**: The method area is shared by all threads, created when the JVM starts. It stores runtime constant pool, field and method information, static variable, and method bytecode for each of the classes and interfaces read by the JVM. The method area can be implemented in various formats by JVM vendor. Oracle Hotspot JVM calls it Permanent Area or Permanent Generation (PermGen). The garbage collection for the method area is optional for each JVM vendor.
- **Runtime constant pool**: An area that corresponds to the constant_pool table in the class file format. This area is included in the method area; however, it plays the most core role in JVM operation. Therefore, the JVM specification separately describes its importance. As well as the constant of each class and interface, it contains all references for methods and fields. In short, when a method or field is referred to, the JVM searches the actual address of the method or field on the memory by using the runtime constant pool.

Heap: A space that stores instances or objects, and is a target of garbage collection. This space is most frequently mentioned when discussing issues such as JVM performance. JVM vendors can determine how to configure the heap or not to collect garbage.

2.4 Execution Engine

The bytecode that is assigned to the runtime data areas in the JVM via class loader is executed by the execution engine. The execution engine reads the Java Bytecode in the unit of instruction. It is like a CPU executing the machine command one by one. Each command of the bytecode consists of a 1-byte OpCode and additional Operand. The execution engine gets one OpCode and execute task with the Operand, and then executes the next OpCode. But the Java Bytecode is written in a language that a human can understand, rather than in the language that the machine directly executes. Therefore, the execution engine must change the bytecode to the language that can be executed by the machine in the JVM. The bytecode can be changed to the suitable language in one of two ways.
Interpreter: Reads, interprets and executes the bytecode instructions one by one. As it interprets and executes instructions one by one, it can quickly interpret one bytecode, but slowly executes the interpreted result. This is the disadvantage of the interpret language. The 'language' called Bytecode basically runs like an interpreter.
JIT (Just-In-Time) compiler: The JIT compiler has been introduced to compensate for the disadvantages of the interpreter. The execution engine runs as an interpreter first, and at the appropriate time, the JIT compiler compiles the entire bytecode to change it to native code. After that, the execution engine no longer interprets the method, but directly executes using native code. Execution in native code is much faster than interpreting instructions one by one. The compiled code can be executed quickly since the native code is stored in the cache.

3. JAVA Security Architecture

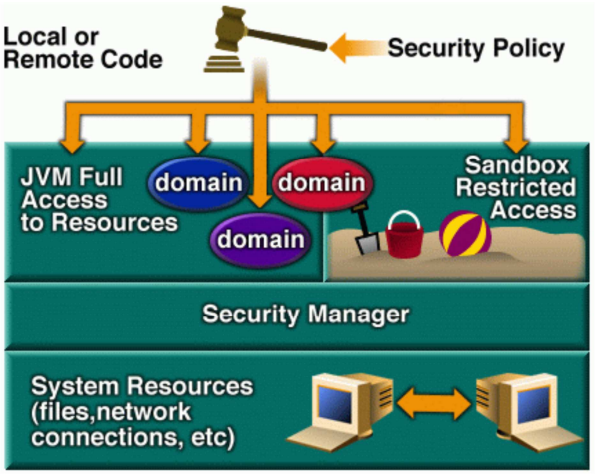
The new security architecture in SDK1.2 (the JDK has been renamed later and is now called a Software Development Kit (SDK), so we have the Java 2 SDK) let the users expand the original sandbox idea, create their

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.

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



3.1. 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.

3.2 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.

3.3 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.

4. Implementation of security architect

The java security model is used to deal with following vulnerabilities:

4.1 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.

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).

 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).

5. SUMMARY

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.

6. REFERENCES

[1] 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.

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

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

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

[5] 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

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

[7] Elliotte Rusty Harold, 2000, The Java Developer's Resource, URL: http://www.ibiblio.org/java/books/jdr/chapters/index.html**,** Quoted 20.8.2009 50

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

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

[10] 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