Java language was invented over 20 years5. What’s interesting and worth mentioning is that it was originally designed with a security in mind. The goal of Java was to provide an ubiquitous programming and secure execution environment for running untrusted, mobile code (Java programs).

Java runtime has a form of an abstract Java Virtual Machine of which goal is to provide a secure execution environment for Java programs. Java Virtual Machine is composed of several components as illustrated in Fig. 1. Those of a critical nature to the security of a Java VM environment are denoted below:

Class Loaders

Bytecode Verifier

Security Manager

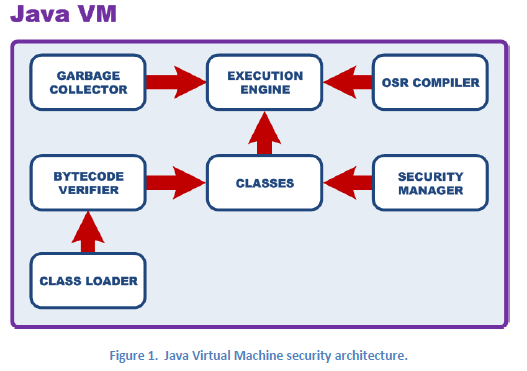
JVM Runtime

- Execution engine

- Classes definition (Java / native code)

OSR compiler

Garbage Collector

Class Loaders are special Java objects that always inherit from java.lang.ClassLoader class or its subclass. They provide Java Virtual Machine with Java programs to execute. These programs have a form of Class files, which may come from arbitrary locations such as remote hosts or local file system. Class Loader location denoting a base source of classes is called a codebase.

Multiple Class files being part of a given Java application can be packaged in a form of ZIP or JAR files. They can be also cryptographically signed in order to verify the authenticity of the application and its provider.

Class Loader objects implements several security critical methods. This includes the following:

- protected Class findClass(String name)

This method is invoked whenever a definition of a target class name is not found in the Java VM and when an attempt is made to load it from some external location.

- public Class loadClass(String name)

- protected Class loadClass(String name, boolean resolve)

These are the base methods that allows to load a given class (denoted by the name argument) to VM. The returned object is the instance of java.lang.Class class, which is a Java representation of VM classes. The second form of loadClass method contains additional argument that specifies whether class resolving (linking) should take place.

- protected final Class defineClass(String name, byte body[], int off, int size, ProtectionDomain protectiondomain)

This method is invoked whenever a given class (denoted by the name argument) is to be defined in the VM. The body argument holds the Class bytes to define. The protection domain argument will be discussed further in this document. It is sufficient to say that it denotes the permissions of a defined class.

The abovementioned methods are not accessible to untrusted code, except one argument loadClass() method.

Class Loader objects are quite powerful. They provide class definitions to the VM. They can specify permissions for loaded classes. Finally, they can also load native libraries into Java VM. These are just a few of the many reasons behind the requirement for the possession of a proper security privilege designating Class Loader creation. The checks for this privilege are implemented in ClassLoader instance initialization method (<init>).

Bytecode Verifier is the primary gatekeeper of Java VM security. This component is called during class loading process. It makes sure that a given sequence of bytes provided to the defineClass() method of Class Loader conforms to the Class file format. It also verifies the integrity and safety of bytecode instruction streams embedded in a Class definition.

Bytecode Verifier works in multiple passes during which it verifies VM constraints defined in Java Virtual Machine Specification. This is the Bytecode Verifier that verifies the type-safety of a target Java code. Any attempt to conduct an illegal type-cast from integer to object or vice-versa should be detected by this VM component.

Bytecode Verifier conducts a static analysis of a target bytecode instruction stream. It does this work by emulating the effect of a target instruction to the content of Java VM state, but solely with respect to the type information held in registers and on the stack. In the past, Bytecode Verifier algorithm inferenced all type information during the analysis of bytecode instruction flow.

**1.2.3 Security Manager**

Security Manager verifies and authorizes all security sensitive operations in a given VM environment. Security Manager objects are instances of java.lang.SecurityManager class or its subclass. There is one special object in each Java VM environment that denotes the Security Manager. A reference to it can be obtained by calling getSecurityManager() method of java.lang.System class. This is the reference that’s stored in a private static security field of this class.

Security Manager implements security checks verifying for the permissions required prior to conducting a given security sensitive operation. Its sample invocation is illustrated by Fig. 2.

****

**Figure 2. Sample invocation of Security Manager’s check.**

The absence of Security Manager is indicated by its null value. In such a case, no security checks are in place and Java program can run without any restrictions in a target Java VM environment.

The possibility to create an instance of a Security Manager does not lead to a direct compromise of VM security. The reason for it is that this is the security field of java.lang.System class that matters - it holds a reference to the Security Manager, which is in use by the environment.

Security Manager verifies whether a target class has specific permissions required for a given security sensitive operation. Java permissions are instances of java.security.Permission class or its subclasses.

Java SE has dedicated permissions for specific operations, such as network access, file system access, native library loading, specific API access, restricted package access, program execution, etc. There is also one special permission object that denotes ROOT privileges in Java. This is AllPermission permission.

It should be noted that many single permissions can be easily elevated to AllPermission.

It was already mentioned that defineClass() method of ClassLoader class can be used to provide Java VM with both Class definition and its permissions. The permissions that are used for the call are not provided in a direct form, but are encapsulated inside a Protection Domain object. The reason for it is because each class loaded into VM is defined in a specific Protection Domain (instance of java.security.ProtectionDomain class). Same Protection Domain (PD) is assigned to classes that come from the same location (CodeSource) and that share both a Class loader instance and a set of Permissions (permissions assigned to classes by this PD). Sample protection domain is illustrated on Fig. 3.

**Figure 3. Sample Protection Domain of untrusted Java Applet application.**

**3. 4 REMOTE, SERVER-SIDE CODE EXECUTION**

Vulnerabilities in Java are usually associated with the risk they pose to users of various web browsers. That’s completely natural taking into account the widespread use of Java Plugin software. There are however some other exploitation scenarios that are worth mentioning. This in particular concerns the possibility to exploit Java security issues on servers. Below, we present the idea behind two such scenarios that could facilitate the attack against server side Java software.

3.4.1 RMI protocol attack

RMI protocol12 is the base protocol used for communication between clients and servers during Java Remote Method Invocation13. RMI protocol implementation supports the concept of user provided codebases. A codebase is the URL value pointing to the remote resource where remote RMI server should look for unknown (non-system) classes. What’s interesting is that Codebase URL can be provided by the RMI client as part of the RMI call. It will be taken into account by the RMI server if java.rmi.server.useCodebaseOnly property is set to true. If true, RMI server will create RMIClassLoader instance with user provided Codebase URL. It will be further used as a base class loader during object deserialization by a MarshalInputStream.

RMI implementation does not verify whether a deserialized object is type compatible with the input argument of a target method call. RMI server reads and instantiates object provided as an argument to the call with the use of RMIClassLoader. If the object to read is of an unknown class, an attempt will be made to fetch class data from the Codebase URL provided by the user. That alone creates a possibility for remote loading and execution of user provided Java code.

**Reference:**

Security Explorations, “Security Vulnerabilities in Java SE”, Technical Report, Ver. 1.0.2, SE-2012-01 Project. URL: <http://www.security-explorations.com/materials/se-2012-01-report.pdf>