Journal Of Harmonized Research (JOHR)

Journal Of Harmonized Research in Engineering
1(2), 2013, 154-160

ISSN 2347 – 7393

Original Research Article

SECURITY IN JAVA

Nisha Yadav, Shikha Yadav, Preeti Dhanda

Department of Computer Science and Engineering,
Dronacharya College of Engineering, Khentawas,
Farukhnagar, Gurgaon, India

Abstract

The **Java programming language** collectively with its runtime environment is well known to provide a lot of security features for applications written in Java and headed for the environment, in which Java applications are deployed. Certainly, under the frequently used slogan "**Java Security**", quite unusual aspects regarding the security of Java applications are addressed. Java has been designed to be an intrinsic safe programming language. In particular Java does not permit to directly access or control memory. While not being element of the innermost security features, "Java Security" also encompasses some standard libraries and extensions that are shipped with SUN's Java Development Kits (JDKs) and that are predominantly intended for the usage in security critical tasks. While the Java standard API does only classify appropriate interfaces for most of these purposes, the JDKs also embrace implementations of cryptographic and security providers.

The runtime environments provided by application-level virtual machines, for instance, the Java Virtual Machine are attractive for Internet application providers because the applications can be deployed on any platform that supports the target virtual machine. By means of Internet applications, organizations in addition to end users face the threat of viruses, trojans, and denial of service attacks. Virtual machine providers are responsive of these Internet security risks and thus provide, for example, runtime monitoring of untrusted code and access control to susceptible resources.

1. Introduction

The term **security** means different things to different people. Computer network communities often talk on the subject of firewalls and intrusion detection systems when asked if their computer systems are protected. Folks in management sometimes will argue that their organization's systems are secure because they implement the Secure Socket Layer (SSL) protocol.

Security is not a feature; it is indeed a system property. In building secure systems for the

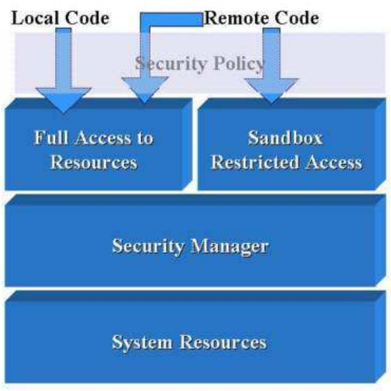
For Correspondence:
jazzynishuATgmail.com
Received on: November 2013
Accepted after revision: December 2013
Downloaded from: www.johronline.com

Internet, one has to realize that nothing can be considered 100% secure. Computer professionals actually need to take the opposite view: consider all client systems as fundamentally insecure. As a consequence, security-minded developers apply preeminent practices in order to build as robust systems as feasible. In addition, security must be integrated into the software development life-cycle in order to craft reliable software. A security review at the end of the software process is not sufficient to produce good quality programs. Security needs to be considered in a much wider sense, as a set of non-functional goals. A thorough treatment of the matter includes procedures for prevention, traceability and auditing, monitoring, privacy and confidentiality, ambiguity, authentication, and integrity.

Security is the practice by which individuals and organizations shield their physical and intellectual property from all forms of attack and pillage. Although security concerns are not new, there is revived interest in the entire area of security in computing systems. This is because today's information systems have become the repositories for both personal and corporate assets and computer networks are providing new levels of access for users.

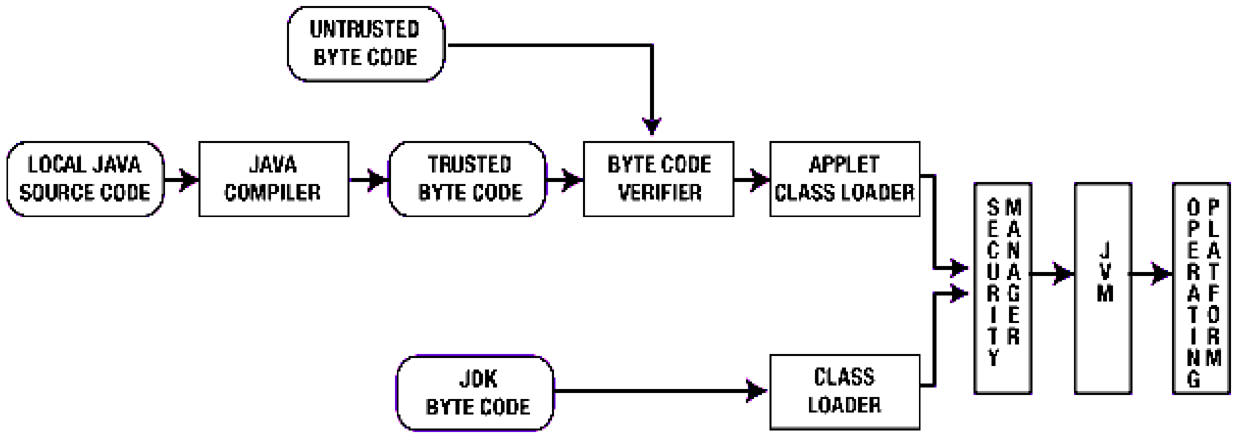
Consequently latest opportunities for unauthorized interaction and possible abuse may occur. In order to combat potential security threats, users need programs they can rely on. Furthermore, developers are looking for a development platform that has been intended with built-in security capabilities. This is where the Java stage comes in. As a matter of fact, Java is intended from the ground up for network-based computing, and security dealings are an integral part of Java's design.

Java Security Architecture-



- **Security** can have varying levels of difficulty for implementation. One factor in determining the difficulty is the number and distribution of the systems. When only individual systems need to be protected, such as one computer with all files residing locally and with no need to connect to any outside resources, security is not as complex as with distributed systems. With distributed systems architecture, there are different nodes and resources. One major issue with distributed systems is application security. The Java architecture for distributed systems computing was designed taking security requirements into consideration.

Developers need to create programs that are executed on remote distributed systems. An architecture needed to be put in place, however, that would not leave these systems vulnerable to malicious code. This was accomplished through the Java architecture. The source code is written and then converted to byte code and is stored as a class file, which is interpreted by the Java Virtual Machine (JVM) on the client. Class loaders then load any additional classes that are needed by the applications. Several security checks are put between the remote server distributing the program, and the client executing it, such as the "sandbox" security model, the byte code verifier, the applet class loader, the security manager, and through other security measures that can be implemented through Java's security APIs.



Java Security Model

In the rest of this section, we will briefly describe three parts of the Java security model, which are byte code verifier, class loader and security manager-

1. Java Bytecode Verifier

Java compiler compiles source programs into byte codes, and a trustworthy compiler ensures that Java source code does not contravene the safety rules. At runtime, a compiled code section can come from anywhere on the net, and it is unknown if the code fragment comes from a trustworthy compiler or not. So, basically the Java runtime simply does not trust the incoming code, and instead subjects it to a series of tests by **byte code verifier**.

The byte code verifier looks at the class files that are to be executed and analyzes them based on specific checks. The code will be verified by three or four passes depending on whether or not any methods are invoked. Gollmann states that some of the checks performed are to make certain that the proper format is used for the class, to prevent stack overflow, to maintain type integrity, to verify that the data does not change between types, and that no illegal references to other classes are made. Hartel and Moreau further state that the byte code verifier ensures that jumps do not lead to illegal instructions, that method signatures are valid, access control, initialization of objects, and that "subroutines used to employ exceptions and synchronized statements are used in FIFO order"

2. Java Class Loader

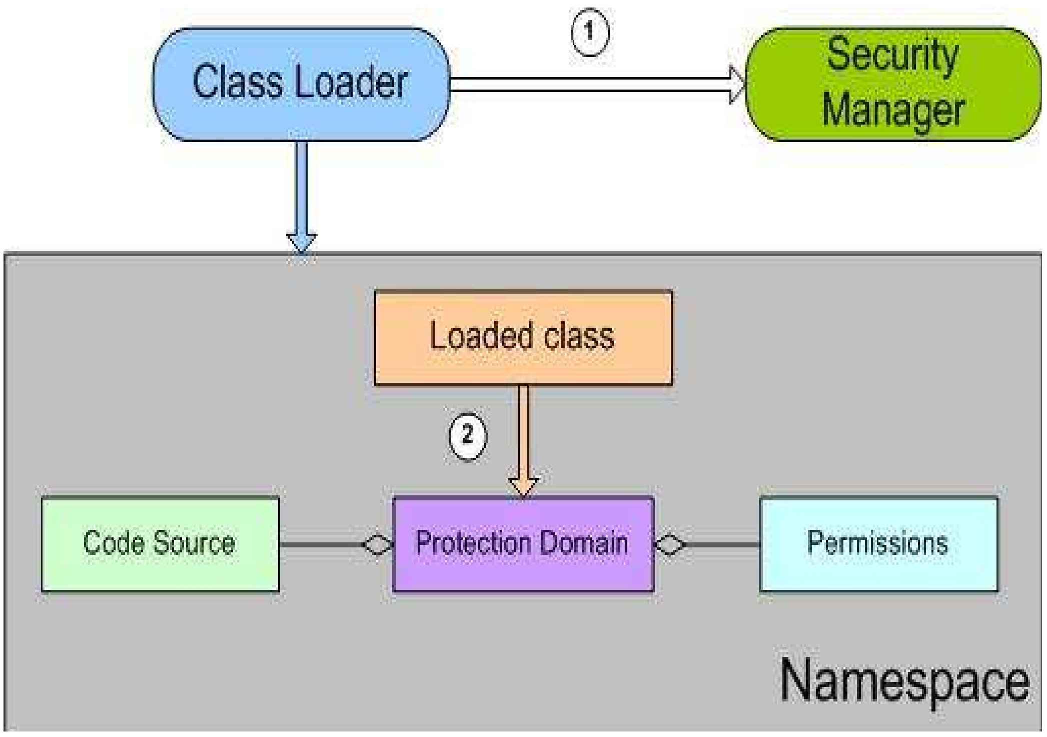
The class loader is basically defined in Java by an abstract class, **Class Loader**. As an interface, it can be used to define a policy for loading Java classes into the runtime environment. The major functions of the class loader are :-

· It fetches the applet's code from the remote machine.

· It creates and enforces a namespace hierarchy. One of its more important functions is to ensure that running applet do not replace system-level components within the runtime environment. In particular, it prevents applets from creating their individual class loader.

· It prevents applets from invoking method that are part of the system's class loader.

As a Java application is executed, further classes may be called. These classes are not loaded until they are needed. When they are called the applet class loader is liable for loading the specified applets. Classes in Java are structured by name spaces, and each class loader is responsible for one name space. The class loaders are therefore responsible to protect the integrity of the classes in its name space (Gollmann, 2001). Java has built-in classes that reside locally, though, that are loaded automatically without any security checks. The path to these classes is indicated by the CLASSPATH environment variable.



3. Java Security Manager

The **Security Manager** contains a number of methods which are anticipated to be called to check specific types of actions. The Security Manager class itself is not projected to be used directly, instead it is intended to be sub classed and installed as the System Security Manager. The sub classed Security Manager can be used to instantiate the preferred security policy. The Security Manager provides an enormously flexible and powerful mechanism for conditionally allowing access to resources.

The Security Manager methods which ensure access are passed arguments which are necessary to implement conditional access policies, as well as having the ability to check the execution stack to determine if the code has been called by local or downloaded code.

Some of the Security Manager's duties include:-

· Managing all socket operations.

· Guarding access to protected resources including files, personal data, etc.

· Controlling the creation of, and all access to, operating system programs and processes.

· Preventing the installation of new Class Loaders.

· Maintaining thread integrity.

· Controlling access to Java packages (i.e., groups of classes).

When writing applications, developers often wish to protect variables and methods from being modified by classes that do not belong to the group of classes they have written. In order to create this division, classes are grouped into packages. When a variable or method is declared in a class, it can be private (access only through same class), protected (access through class or subclass), public (any class can access), or they may chose not to use any of the former, in which case only classes within the same package will have access. Depending on the package that a class belongs to, the class will have different access to the other classes in the package, so security could be compromised if an unauthorized class attaches itself to the package. The security manager makes sure that only classes that actually belong to the package in question are able to declare themselves in this package. The security settings are configured through a security policy.

Java has provided developers the means to create their own security manager. To create it, the developer must create a subclass of the Security Manager class, and override whichever methods are necessary to implement the required security. For example, the developer may decide to impose a stricter policy for reading and writing files. This could be attained through overriding the read and write methods already defined in the super class.

Java Security Fundamentals

The following section describes the underlying security foundation of the Java platform-

1. The Java language rules

The basic building blocks in the Java security model are a set of language specific rules.

Their primary purpose is to deny access to or modification of random locations in the memory of the hosting machine. The rules are-

1. Access levels are strictly enforced.
2. Code cannot access arbitrary memory locations.
3. Entities marked with the final identifier cannot be changed.
4. Variables may not be used before they have been initialized.
5. You cannot access data outside your initial data set. E.g., attempts to access an array index that does not exist will result in an ArrayIndexOutOfBounds.
6. Objects cannot be arbitrarily cast into other objects.

2. Enforcement of the Java language rules

The constructs liable for enforcing these rules are the *compiler*, *byte-code verifier*, and the *Java Virtual Machine* (JVM). The first line of defence is the compiler. During compilation, every rule is checked; the compiler cannot enforce checking of array bounds or all cases of illegal casts. These checks will be completed at runtime. The problem with casting arises when two objects are not known to be unrelated:

```
Object maybeCar = myVector.elementAt(0);
Car ferrari = (Car) maybeCar;
```

There is no way for the compiler to know whether the object returned from the vector is a car, or just something posing as a car.

When classes are loaded in Java, the byte-code verifier provides a means to check the rules on the list above. In addition, the byte-code verifier also makes sure that:

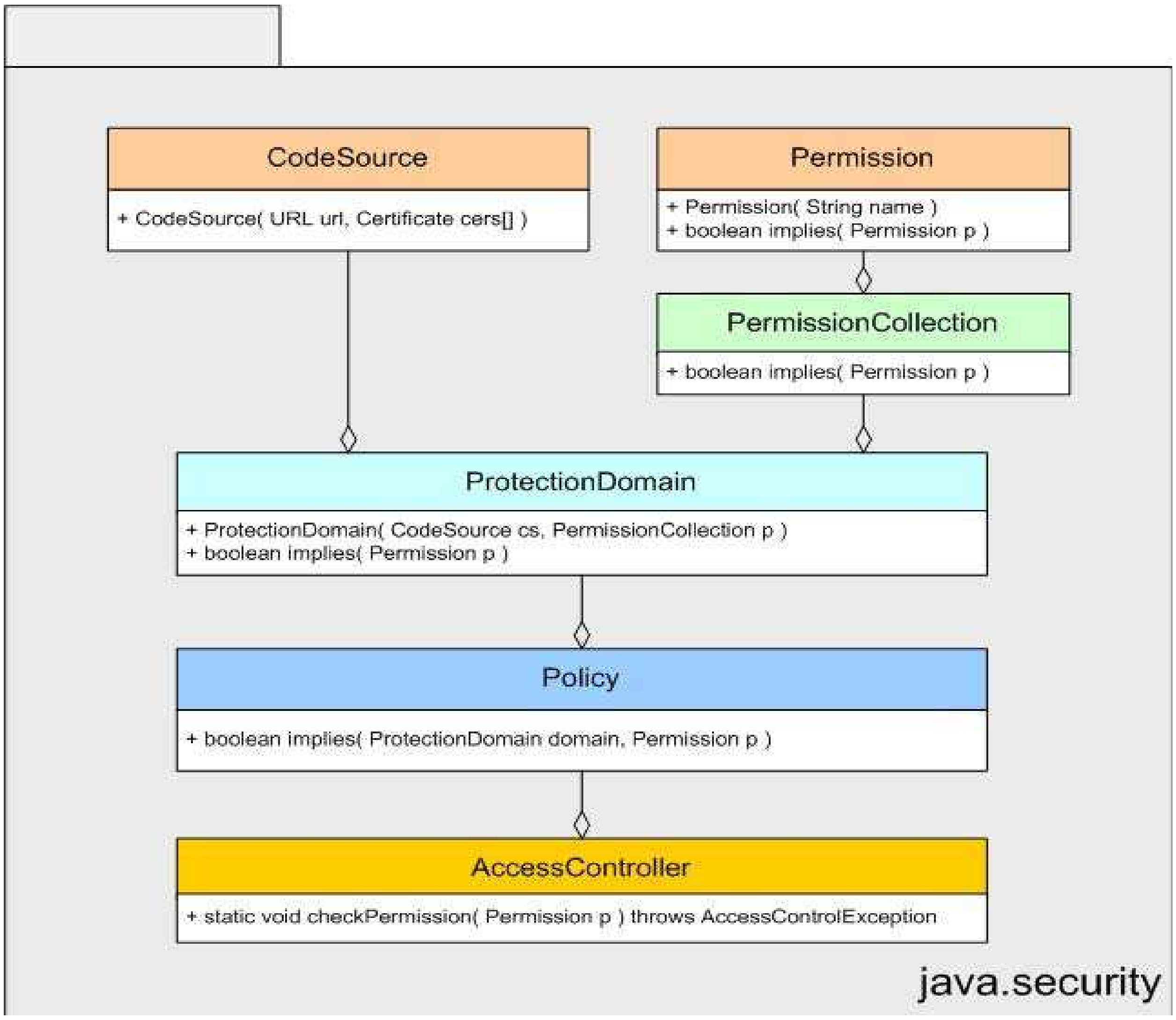
• The format of the class file is correct.
• Every class has a single superclass.2
• There are no operand stack overflows or underflows.

The compiler and the byte-code verifier have overlapping tasks: they perform some of the same checks. The double-checking is crucial when dealing with code that has been compiled by somebody else, whom you probably don't trust.

To exercise the byte-code verifier, you run your program with the -verify alternative from the command line. In upcoming releases of the Java platform, the byte-code verifier will most likely be running by default. But for now, using the -verify option is highly recommended when running code compiled by others.

The JVM is responsible for scrutiny of array bounds and the validity of object casts.

Non-compliance with the former check results in a java.lang.ArrayIndexOutOfBounds Exception. Likewise, illegal object casts end with the throwing of a java.lang.ClassCastException at runtime.


CodeSource
+ CodeSource(URL url, Certificate cers[])
Permission
+ Permission(String name)
+ boolean implies(Permission p)
PermissionCollection
+ boolean implies(Permission p)
ProtectionDomain
+ ProtectionDomain(CodeSource cs, PermissionCollection p)
+ boolean implies(Permission p)
Policy
+ boolean implies(ProtectionDomain domain, Permission p)
AccessController
+ static void checkPermission(Permission p) throws AccessControlException
java.security

The Advantages of Software Protection-
Historically, memory protection and privilege levels have been implemented in hardware: memory protection via base / limit registers, segments, or pages; and privilege levels via user / kernel mode bits or rings [Schroeder and Saltzer 1972]. Recent mobile code systems,
however, rely on software rather than hardware for protection. The switch to software
mechanisms is being driven by two needs: portability and performance.-

1. **Portability -** The first argument for software protection is portability. A user level software product like a browser must coexist with a variety of operating systems. For
a Web browser to use hardware protection, the operating system would have to provide accesses to the page tables and system calls, but such mechanisms are not available universally across platforms. Software protection allows a browser to have platform-independent security mechanisms.

2. **Performance-** Second, software protection offers significantly cheaper cross domain calls. The magnitude of the performance difference is estimated first. The results should not be considered highly accurate, since they mix measurements from different architectures. However, the effect measured is so large that these small inaccuracies do not affect conclusions. For example, first, the performance of a null call between Common Object Model (COM) objects on a 180 MHz Pentium Pro PC running Windows NT 4.0 is measured. When the two objects are in different tasks, the call takes 230 sec; when the called object is in a

dynamically linked library in the same task, the call takes only 0.2 sec — a factor of 1000 difference. While COM is a very different system from Java, the performance disparity exhibited in COM would likely also appear if hardware protection were applied to Java. This ratio appears to be growing even larger in newer processors [Ousterhout 1990; Anderson et al. 1991].

The time difference would be acceptable if cross-domain calls were very rare. But modern software structures, especially in mobile code systems, are leading to tighter binding between domains.

To illustrate this fact, the Sun Java Virtual Machine (JVM) (JDK 1.0.2 interpreter running on a 167 MHz Ultra Sparc) is instrumented to measure the number of calls across trust boundaries.

In UNIX, a system-call crosses domains between user and kernel processes. In Java, a method call between applet and system classes also crosses domains because system classes have additional privileges.

References-

1. Anderson, t. E., levy, h. M., bershad, b. N., and lazowska, e. D. 1991. The interaction of architecture and operating system design. In Proceedings of the Fourth ACM Symposium on Architectural Support for Programming Languages and Operating Systems.
2. Badger, l., sterne, d. F., sherman, d. L., walker, k. M., and haghighat, s. A. 1995. Practical domain and type enforcement for UNIX. In Proceedings of the 1995 IEEE Symposium on Security and Privacy 66–77.
3. Carlisle Adams and Steve Lloyd, Understanding PKI—Concepts, Standards, and Deployment Considerations. Pearson Education, Inc, second edition 2003.
4. Ross Anderson, "Why Cryptosystems Fail," ACM 1st Conf.- Computer and Comm.Security 1993.
5. Ross Anderson, Alan Blackwell, Alashdair Grant, and Jeff Yan, "Password Memorability and Security—Empirical Results," IEEE Security and Privacy, September/October 2004.
6. BankID.no. Retrieved January 21, 2005, from bankid.no. http://www.bankid.no/ Building Secure Software—Best practices in software security. Retrieved May 16,2005, from Cigital.http://www.cigital.com/presentations/roots/bss05/index_files/frame.html
7. Joseph A. Bank. Java security, PMG group at MIT LCS, December, 1995,http://swissnet.ai.mit.edu/~jbank/javapaper/javapaper.html
8. Rich Levin. Security grows up: the Java 2 platform security model, October 1998,http://www.javasoft.com/features/1998/11/jdk.security.html
9. Executive Summary. Secure computing with Java: now and the future, 1998, http://www.javasoft.com/marketing/collateral/security.html
10. Chizmadia, D. (1998). A quick tour of the CORBA security service. Retrieved August 27, 2002, from http://www.itsecurity.com/papers/corbasec.htm
11. COM+ security programming, part 1: declarative role-based security. (2001, JuneRetrieved August 28, 2002, from http://www.itworld.com/nl/windows_sec/06112001/
12. COM+ security programming, part 2: programmatic role-based security. (2001, June 18). Retrieved August 28, 2002, from http://www.itworld.com/nl/windows_sec/06182001/
13. Emmerich, W., & Kaveh, N. (2002). Component technologies: Java Beans, COM, CORBA, RMI, EJB and the CORBA component model. Software Engineering, 691-692.
14. Gollmann, D. (2001). Computer security. West Sussex, England: John Wiley & Sons Ltd. Gong, Li. (1997).