JAVA SECURITY

Ritu Yadav
Dronacharya College of Engineering
Khentawas, Haryana

Abstract :- **Web browsers, Web servers, Java application servers all are instances of Java execution environments that run more or less entrusted Java applications. In all these environments, Java applications can come from different sources. Consequently, the application developers rarely know which other applications exist in the target Java execution environment. This paper investigates the requirements that need to be imposed on such a container from a security point of view and how the requirements have been implemented by different Java applications. More specifically, we show a general risk analysis considering assets, threats and vulnerabilities of a Java programming. This risk analysis exposes generic Java security problems and leads to a set of security requirements. These security requirements are then used to evaluate the security architecture of existing Java programming for Java applications, applets, servlets, and Enterprise Java Beans. For comparison, the requirements are also catechize for a C++ applications.**

INTRODUCTION

Java security technology includes a large set of APIs, tools, and implementations of commonly used security algorithms, mechanisms, and protocols. The Java security APIs spans a wide range of areas, including cryptography, public key infrastructure, and secures communication, authentication, and access control. Java security technology provides the developer with a comprehensive security framework for writing applications, and also provides the user or administrator with a set of tools to securely manage applications. Underlying the Java SE Platform is dynamic, extensible security architecture, standards-based and interoperable. Security features — cryptography, authentication and authorization, public key infrastructure, and more — are built in. The Java security model is based on a customizable "sandbox" in which Java software programs can run safely, without potential risk to systems or users.

SECURITY FEATURES IN JAVA

The JVM

The binary form of programs running on the Java platform is not native machine code but an intermediate byte code.
The JVM performs verification on this byte code before running it to prevent the program from performing unsafe operations such as branching to incorrect locations, which may contain data rather than instructions. It also allows the JVM to enforce runtime constraints such as array bounds checking. This means that Java programs are significantly less likely to suffer from memory safety flaws such as buffer overflow than programs written in languages such as C which do not provide such memory safety guarantees.
The platform does not allow programs to perform certain potentially unsafe operations such as pointer arithmetic or unchecked type casts. It also does not allow manual control over memory allocation and DE allocation; users are required to rely on the automatic garbage collection provided by the platform. This also contributes to type safety and memory safety.

SECURITY MANAGER

The platform provides a security manager which allows users to run untrusted bytecode in a "sandboxed" environment designed to protect them from malicious or poorly written software by preventing the untrusted code from accessing certain platform features and APIs. For example, untrusted code might be prevented from reading or writing files on the local filesystem, running arbitrary commands with the current user's privileges, accessing communication networks, accessing the internal private state of objects using reflection, or causing the JVM to exit. The security manager also allows Java programs to be cryptographically signed users can choose to allow code with a valid digital signature from a trusted entity to run with full privileges in circumstances where it would otherwise be untrusted.
Users can also set fine-grained access control policies for programs from different sources. For example, a user may decide that only system classes

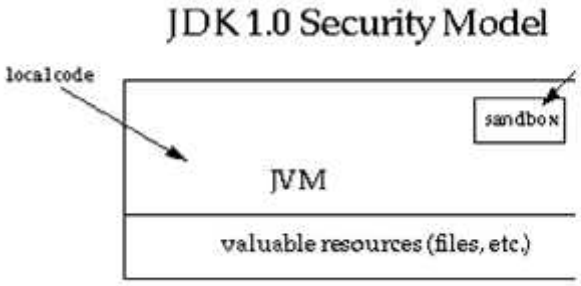
should be fully trusted, that code from certain trusted entities may be allowed to read certain specific files, and that all other code should be fully sandboxed.

SECURITY APIs

The Java Class Library provides a number of APIs related to security, such as standard cryptographic algorithms, authentication, and secure communication protocols.

JAVA SECURITY MODEL

The original security model provided by the Java platform is known as the sandbox model, which existed in order to provide a very restricted environment in which to run untrusted code obtained from the open network. 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. This sandbox model is illustrated in the figure below.



The sandbox model was deployed through the Java Development Kit (JDK), and was generally adopted by applications built with JDK 1.0, including Java-enabled web browsers.

Overall security is enforced through a number of mechanisms. First of all, the language is designed to be type-safe and easy to use. The hope is that the burden on the programmer is such that the likelihood of making subtle mistakes is lessened compared with using other programming languages such as C or C++. Language features such as automatic memory management, garbage collection, and range checking on strings and arrays are examples of how the language helps the programmer to write safe code. Second, compilers and a byte code verifier ensure that only legitimate Java byte codes are executed. The byte code verifier, together with the Java Virtual Machine, guarantees language safety at run time. Moreover, a class loader defines a local name space, which can be used to ensure that an untrusted applet cannot interfere with the running of other programs. Finally, access to crucial system resources is mediated by the Java Virtual Machine and is checked in advance by a Security Manager class that restricts the actions of a piece of untrusted code to the bare minimum.

CONCLUSION

- Fine-grained access control.

This capability existed in the JDK from the beginning, but to use it, the application writer had to do substantial programming (e.g., by sub classing and customizing the Security Manager and Class Loader classes). The Hot Java browser 1.0 is such an application, as it allows the browser user to choose from a small number of different security levels.

However, such programming is extremely security-sensitive and requires sophisticated skills and in-depth knowledge of computer security. The new architecture will make this exercise simpler and safer.

- 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.

- Easily extensible access control structure.

Up to JDK 1.1, in order to create a new access permission, you had to add a new check method to the Security Manager class. The new architecture allows typed permissions (each representing an access to a system resource) and automatic handling of all permissions (including yet-to-be-defined permissions) of the correct type. No new method in the Security Manager class needs to be created in most cases. (In fact, we have so far not encountered a situation where a new method must be created.)

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

There is no longer a built-in concept that all local code is trusted. Instead, local code (e.g., non-system code, application packages installed on the local file system) is subjected to the same security control as applets, although it is possible, if desired, to declare that the policy on local code (or remote code) be the most liberal, thus enabling such code to effectively run as totally trusted. The same principle applies to signed applets and any Java application.

Finally, an implicit goal is to make internal adjustment to the design of security classes (including the Security Manager and Class Loader

classes) to reduce the risks of creating subtle security holes in future programming.

REFERANCES

- **https://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-spec.doc1.html**

- **http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html**