1 Introduction

Since the inception of Java technology, there has been strong and growing interest around the security of the Java platform as well as new security issues raised by the deployment of Java technology.

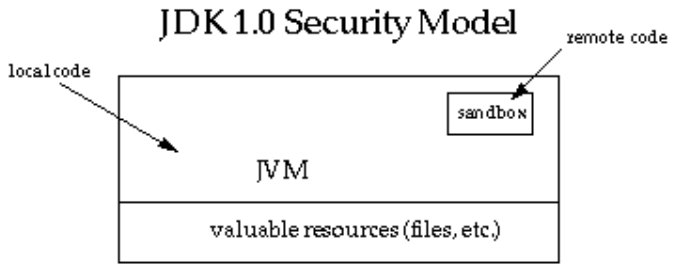
From a technology provider's point of view, Java security includes two aspects:

- Provide the Java platform as a secure, ready-built platform on which to run Java-enabled applications in a secure fashion.
- Provide security tools and services implemented in the Java programming language that enable a wider range of security-sensitive applications, for example, in the enterprise world.

This document discusses issues related to the first aspect, where the customers for such technologies include vendors that bundle or embed Java technology in their products (such as browsers and operating systems).

1.1 The Original Sandbox 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.

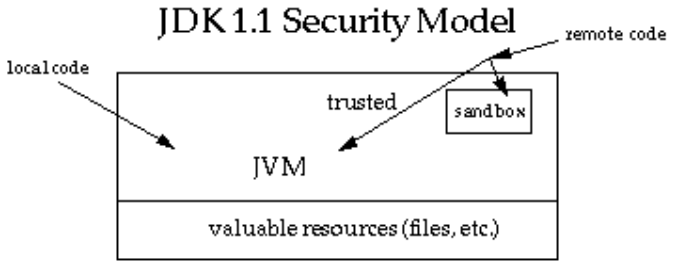
Overall security is enforced through a number of mechanisms. First of all, the language is designed to be type-safe and easy to use. The hope is that the burden on the programmer is such that the likelihood of making subtle mistakes is lessened compared with using other programming languages such as 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 bytecode verifier ensure that only legitimate Java bytecodes are executed. The bytecode verifier, together with the Java Virtual Machine, guarantees language safety at run time.

Moreover, a classloader defines a local name space, which can be used to ensure that an untrusted applet cannot interfere with the running of other programs.

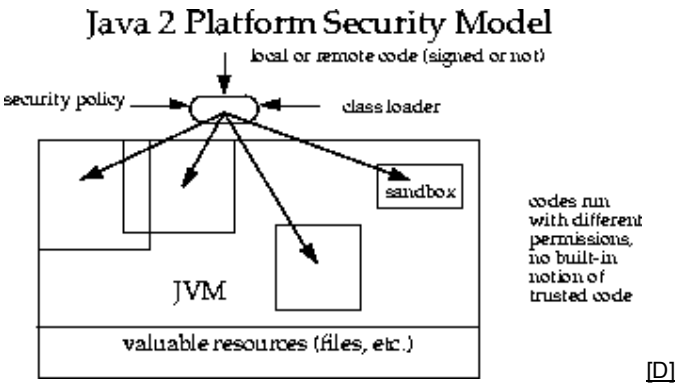
Finally, access to crucial system resources is mediated by the Java Virtual Machine and is checked in advance by a SecurityManager class that restricts the actions of a piece of untrusted code to the bare minimum.

JDK 1.1 introduced the concept of a "signed applet", as illustrated by the figure below. In that release, a correctly digitally signed applet is treated as if it is trusted local code if the signature key is recognized as trusted by the end system that receives the applet. Signed applets, together with their signatures, are delivered in the JAR (Java Archive) format. In JDK 1.1, unsigned applets still run in the sandbox.



1.2 Evolving the Sandbox Model

The new Java SE Platform Security Architecture, illustrated in the figure below, is introduced primarily for the following purposes.



[D]

- 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 subclassing and customizing the SecurityManager and ClassLoader classes). The HotJava 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 SecurityManager 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 SecurityManager 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 SecurityManager and ClassLoader classes) to reduce the risks of creating subtle security holes in future programming.

CONTENTS | PREV | NEXT