Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2

Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers
JavaSoft, Sun Microsystems, Inc.

Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java™ Development Kit 1.2

Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers

JavaSoft, Sun Microsystems, Inc.
{gong,mrm,hemma,schemers}@eng.sun.com

Abstract

This paper describes the new security architecture that has been implemented as part of JDK1.2, the forthcoming Java™ Development Kit. In going beyond the sandbox security model in the original release of Java, JDK1.2 provides fine-grained access control via an easily configurable security policy. Moreover, JDK1.2 introduces the concept of protection domain and a few related security primitives that help to make the underlying protection mechanism more robust.

1 Introduction

Since the inception of Java [8, 11], there has been strong and growing interest around the security of Java as well as new security issues raised by the deployment of Java. From a technology provider's point of view, Java security includes two aspects [6]:

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

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

It is worth emphasizing that this work by itself does not claim to break significant new ground in terms of the theory of computer security. Instead, it offers a real world example where well-known security principles [5, 12, 13, 16] are put into engineering practice to construct a practical and widely deployed secure system.

1.1 The Original Security Model

The original security model provided by Java is known as the sandbox model, which exists in order to provide a very restricted environment in which to run untrusted code (called applet) obtained from the open network. The essence of the sandbox model, as illustrated by Figure 1, is that local code is trusted to have full access to vital system resources (such as the file system) while downloaded remote code is not trusted and can access only the limited resources provided inside the sandbox.

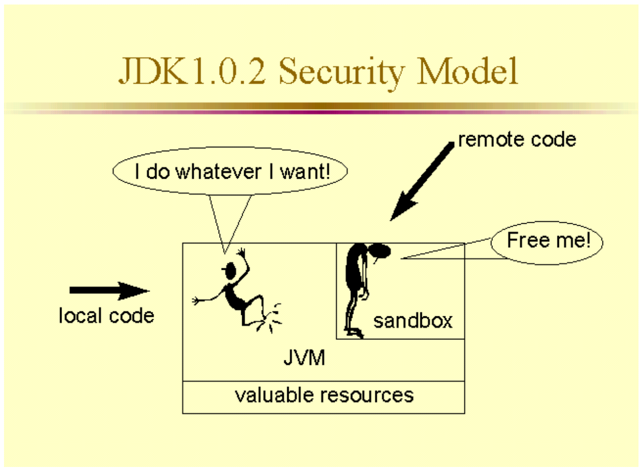

Figure 1: JDK1.0.x Security Model

This sandbox model is deployed through the Java Development Toolkit in versions 1.0.x, and is generally adopted by applications built with JDK, 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 it is less likely to make subtle mistakes, 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 safer code.

Second, compilers and a bytecode verifier ensure that only legitimate Java code is executed. The bytecode verifier, together with the Java virtual machine, guarantees language type safety at run time.

Moreover, a class loader defines a local name space, which is used to ensure that an untrusted applet cannot interfere with the running of other Java 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 to the minimum the actions of untrusted code.

JDK1.1.x introduced the concept of signed applet. In this extended model, as shown in Figure 2, 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.

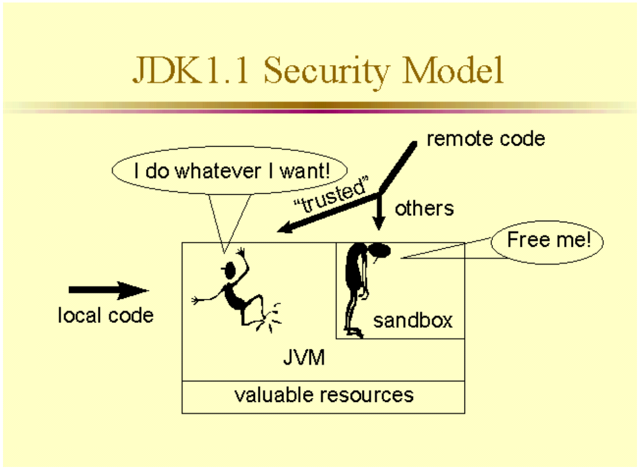

Figure 2: JDK1.1 Security Model

The rest of this paper focuses on the new system security features. Discussion of various language safety issues can be found elsewhere (e.g., [3, 4, 19, 21]).

1.2 Evolving the Sandbox Model

The new security architecture in JDK1.2, as illustrated in Figure 3, is introduced primarily for the following purposes.

- Fine-grained access control.

  This capability has existed in Java from the beginning, but to use it, the application writer has to do substantial programming (e.g., by subclassing and customizing the `SecurityManager` and `ClassLoader` classes).

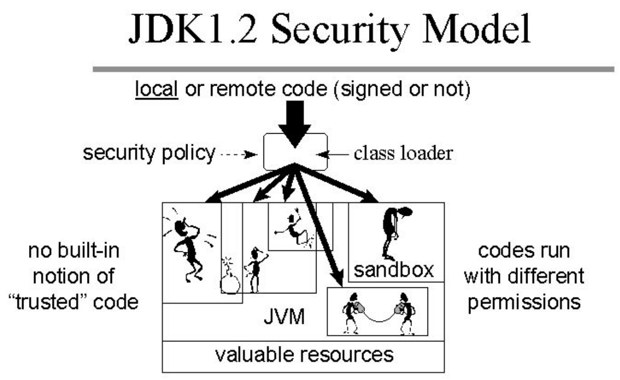
JDK1.2 Security Model
local or remote code (signed or not)
security policy
class loader
no built-in notion of "trusted" code
sandbox
codes run with different permissions
JVM
valuable resources

Figure 3: JDK1.2 Security Model

  HotJava is such an example application. However, such programming is extremely security sensitive and requires sophisticated skills and in-depth knowledge of computer security. The new architecture makes this exercise simpler and safer.

- Easily configurable security policy.

  Once again, this feature exists in Java but is not easy to use. This design goal implies that the security and its implementation or enforcement mechanism should be clearly separated. Moreover, because writing security code is not straightforward, it is desirable to allow application builders and users to configure security policies without having to program.

- Easily extensible access control structure.

  Up to JDK1.1, to create a new access permission, one has to add a new `check()` method to the `SecurityManager` class. The new architecture allows typed permissions and automatic handling. No new method in the `SecurityManager` class needs to be created in most cases. (Actually, we have not encountered a situation where a new method must be created.)

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

  There should not be a built-in concept that all local code is trusted. Instead, local code should be subjected to the same security controls as applets, although one should have the choice to declare that the policy on local code (or remote code) be the most liberal (thus local code effectively runs as totally trusted). The same principle applies to signed applets and applications.

Finally, we also take this opportunity to make internal structural adjustment in order to reduce the risks of creating subtle security holes in programs. This effort involves revising the design and implementation of the `SecurityManager` and `ClassLoader` classes as well as the underlying access control checking mechanism.

1.3 Related Work

The fundamental ideas adopted in the new security architecture have roots in the last 40 years of computer security research, such as the overall idea of access control list [10]. We followed some of the Unix conventions in specifying access permissions to the file system and other system resources, but significantly, our design has been inspired by the concept of protection domains and the work dealing with mutually suspicious programs in Multics [17, 15], and right amplification in Hydra [9, 20].

One novel feature, which is not present in operating systems such as Unix or MS-DOS, is that we implement the least-privilege principle by *automatically* intersecting the sets of permissions granted to protection domains that are involved in a call sequence. This way, a programming error in system or application software is less likely to be exploitable as a security hole.

Note that although the Java Virtual Machine (JVM) typically runs over another hosting operating system such as Solaris, it may also run directly over hardware as in the case of the network computer JavaStation running JavaOS [14]. To maintain platform independence, our architecture does not depend on security features provided by an underlying operating system.

Furthermore, our architecture does not override the protection mechanisms in the underlying operating system. For example, by configuring a fine-grained access control policy, a user may grant specific permissions to certain software, but this is effective only if the underlying operating system itself has granted the user those permissions.

Another significant character of JDK is that its protection mechanisms are language-based, within a single address space. This feature is a major distinction from more traditional operating systems, but is very much related to recent works on software-based protection and safe kernel extensions (e.g., [2, 1, 18]), where various research teams have lately aimed for some of the same goals with different programming techniques.

2 New Protection Mechanisms

This section covers the concept and implementation of some important new primitives introduced in JDK1.2, namely, security policy, access permission, protection domain, access control checking, privileged operation, and Java class loading and resolution.

2.1 Security Policy

There is a system security policy, set by the user or by a system administrator, that is represented by a policy object, which is instantiated from the class `java.security.Policy`. There could be multiple instances of the policy object, although only one is "in effect" at any time. This policy object maintains a runtime representation of the policy, is typically instantiated at the Java virtual machine start-up time, and can be changed later via a secure mechanism.

In abstract terms, the security policy is a mapping from a set of properties that characterize running code to a set of access permissions that is granted to the concerned code.[1]

Currently, a piece of code is fully characterized by its origin (its location as specified by a URL) and the set of public keys that correspond to the set of private keys that have been used to sign the code using one or more digital signature algorithms. Such characteristics are captured in the class `java.security.CodeSource`, which can be viewed as a natural extension of the concept of a code base within HTML. (It is important not to confuse `CodeSource` with the `CodeBase` tag in HTML.) Wild cards are used to denote "any location" or "unsigned".

Informally speaking, for a code source to match an entry given in the policy, both the URL information and the signature information must match. For URL matching, if the code source's URL is a prefix of an entry's URL, we consider this a match. For signature matching, if one public key corresponding to a signature in the code source matches the key of a signer in the policy entry, we consider it a match.

[1] In the future, the security policy can be extended to include and consider information such user authentication and delegation.

When a code source matches multiple policy entries, for example, when the code is signed with multiple signatures, permissions granted are additive in that the code is given all permissions contained in all the matching entries. For example, if code signed with key A gets permission X and code signed by key B gets permission Y, then code signed by both A and B gets permissions X and Y.

Verification of signed code uses a new package of certificate `java.security.cert` that fully supports the processing of X.509v3 certificates.

The policy within the Java runtime is set via a programming API. We also specify an external policy representation in the form of an ASCII policy configuration file. Such a file essentially contains a list of entries, each being a pair, consisting of a code source and its permissions. In such a file, a public key is signified by an alias – the string name of the signer – where we provide a separate mechanism to create aliases and import their matching public keys and certificates.

2.2 Permission

We have introduced a new hierarchy of typed and parameterized access permissions that is rooted by an abstract class `java.security.Permission`. Other permissions are subclassed either from the `Permission` class or one of its subclasses, and generally should belong to their own packages.

For example, the permission representing file system access is located in the Java I/O package, as `java.io.FilePermission`. Other permission classes that are introduced in JDK1.2 include: `java.net.SocketPermission` for access to network resources, `java.lang.RuntimePermission` for access to runtime system resources such as properties, and `java.awt.AWTPermision` for access to windowing resources. In other words, access methods and parameters to most of the controlled resources, including access to Java properties and packages, are represented by the new permission classes.

A crucial abstract method in the `Permission` class that needs to be implemented for each new class of permission is the `implies` method. Basically, `a.implies(b) == true` means that, if one is granted permission `a`, then one is naturally granted permission `b`. This is the basis for all access control decisions.

For convenience, we also created abstract classes `java.security.PermissionCollection` and `java.security.Permissions` that are subclasses of the `Permission` class. `PermissionCollection` is a collection (i.e., a set that allows duplicates) of `Permission` objects for a category (such as `FilePermission`), for ease of grouping. `Permissions` is a heterogeneous collection of collections of `Permission` objects.

Not every permission class must support a corresponding collection class. When they do, it is crucial to implement the correct semantics for the `implies` method in the corresponding permission collection classes. For example, FilePermission can get added to the `FilePermissionCollection` object in any order, so the latter must know how to correctly compare a permission with a permission collection.

Typically, each permission consists of a target and an action thus, informally, a permission implies another if and only if both the target and the action of the former respectively implies those of the latter.

Take `FilePermission` for example. There are two kinds of targets: a directory and a file. There are four ways to express a file target: `path`, `path/file`, `path/*`, and `path/-`. `path/*` denotes all files and directories in the directory `path`, and `path/-` denotes all files and directories under the subtree of the file system starting at `path`. The actions include `read`, `write`, `execute`, and `delete`.

Therefore, "read file /tmp/abc" is a permission, and can be created using the following Java code:

```
p = new FilePermission("/tmp/abc", "read");
```

Permission (`/tmp/*`, `read`) implies permission (`/tmp/abc`, `read`), but not vice versa. Permission (`/home/gong/-`, `read,write`) implies permission (`/home/gong/public_html/index.html`, `read`).

In the case of `SocketPermission`, a net target consists of an IP address and a range of port numbers. Actions include `connect`, `listen`, `accept`, and others. One SocketPermission implies another if and only if the former covers the same IP address and the port numbers for the same set of actions.

Applications are free to add new categories of permissions. Note that a piece of Java code can create any number of permission objects, but such actions do not grant the code the corresponding access rights. What matters is that permission objects the Java runtime system associates with the Java code through the concept of protection domains.

2.3 Protection Domain

A new class `java.security.ProtectionDomain` is package-private, and is transparent to most Java developers. It serves as a useful level of indirection in that permissions are granted to protection domains, to which classes and objects belong, and

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase® Smarter legal research.