

CHAPTER 3

Java 2 Security Architecture

*The state is nothing but an instrument of
oppression of one class by another.*
—Friedrich Engels

The need to support flexible and fine-grained access control security policies, with extensibility and scalability, called for an improved security architecture. The architecture introduced with the Java 2 Platform, Standard Edition (J2SE), 1.2, fulfills this goal. This chapter details why the security architecture changes were needed and then gives an overview of the Java 2 security architecture. The three subsequent chapters provide architecture details. Chapter 4 describes secure class loading, Chapter 5 tells how a security policy is specified and represented, and Chapter 6 tells how the security policy is enforced.

3.1 Security Architecture Requirements of Java 2

As discussed in the previous chapter, it was critical that the original release of JDK 1.0 consider security seriously and provide the sandbox security model. Not many technologies have security as a design goal, so Java technology, together with the Internet and the promise of e-commerce, helped to finally move security technology into the mainstream of the computer industry. Doing so was a significant achievement. The next step was to improve on the enhancements incorporated in JDK 1.1 to make the security solutions on the Java platform easy to use and more robust. The Java 2 security architecture corrects limitations of earlier platform versions.

3.1.1 Flexible Access Control

By default, the sandbox model severely restricts the kind of activities that an applet may perform. Although it was the catalyst that created the atmosphere for safe Internet computing, this model treats all applets as potentially malicious. Thus, some applets, such as those created by a corporation's finance group to handle internal transactions, are also limited in what they can do, even though they are likely to be more trustworthy than an arbitrary applet downloaded from an unfamiliar Web site.

Such a blanket restriction on all applets can be limiting. For example, suppose that a customer of a brokerage firm uses a stock-trading applet loaded from the brokerage's Web site. This customer may want to let the applet update local files that contain her stock portfolio. However, access to the client-side file system is prohibited by the sandbox model. Thus, this customer needs *flexible access control*, whereby certain applets can have access that is outside the sandbox. With JDK 1.1, the brokerage firm could sign the trading applet, and, assuming that the customer configured her Java runtime to recognize the brokerage firm to be a trusted signer, the applet could access resources outside the sandbox.

However, the customer may have installed on her local desktop financial management software that handles income tax issues. She might not feel comfortable letting the brokerage firm's applet have free rein on her entire desktop system. In this case, it may be best to confine the applet to limited file system access, perhaps only to the brokerage firm's file folder. What is needed is a model whereby the sandbox can be customized—for example, by the client system—to have flexible shapes and boundaries: in other words, *fine-grained access control*.

Prior to Java 2, one could, in theory, implement a more flexible and finer-grained access control on the Java platform. To accomplish this, however, someone, such as an application writer, had to do substantial programming work by, for example, subclassing and customizing the `SecurityManager`, `ClassLoader`, and other classes. The HotJava browser was an example of such efforts; it had a limited range of user-definable security properties. However, such extremely security-sensitive programming requires in-depth knowledge of computer security and robust programming skills.

The Java 2 security architecture eliminates the need to write custom security code for all but the most specialized environments, such as the military, which may require special security properties, such as multilevel security [72]. Even then, writing custom security code is simpler and safer than before.

3.1 Security Architecture Requirements of Java 2

3.1.2 Separation of Policy Expression from Policy Enforcement

As codified by the `java.lang.SecurityManager` class, the sandbox model implements a specific security policy that is expressed in the implementation of the software that does the policy enforcement. This means that to enforce a different security policy, a customized version of the software must be used—clearly, this is not desirable. Instead, what is needed is an infrastructure that supports a range of easily configurable security policies.

The Java 2 security architecture cleanly separates the enforcement mechanism from the description of security policy. In this way, application builders can configure security policies without having to write special programs.

3.1.3 Flexible and Extensible Policy Enforcement

Prior to Java 2, the Java security architecture hard coded the types of security checks performed. For example, to check whether a file can be opened for reading, you would call the `checkRead` method on the currently installed `SecurityManager`. Such a design is not easily extensible, because it does not accommodate the handling of new types of checks that are introduced as after-market add-ons to the Java runtime. It is also not very scalable. For example, to create a new access check, such as one that checks whether money can be withdrawn from a bank account, you would have to add a new `checkAccountWithdraw` method to the `SecurityManager` class or one of its subclasses. Thousands of various kinds of checks are possible. If methods were created for this large a number, they would clutter the `SecurityManager` class. In fact, because many checks are application specific, not of all them can be defined within the JDK. What is needed is an easily extensible access control structure.

To that end, the Java 2 architecture provides typed access control permissions and an automatic permission-handling mechanism to achieve extensibility and scalability. In theory, no new method ever needs to be added to the `SecurityManager` class. Thus far, throughout the multiple J2SE releases, we have not encountered a situation requiring a new method. Instead, the more general `checkPermission` method added in Java 2 has proved sufficient to handle all security checks. See Chapter 6 for more information.

3.1.4 Flexible and Customizable Security Policy

JDK 1.x had the built-in assumption that all locally installed Java applications were fully trusted and therefore should run with full privileges. As a result, the sandbox model applied only to downloaded unsigned applets. However, software installed locally should not be given full access to all parts of the system. For

example, often a user installs a demo program on the local system and then tries it out. It is prudent to limit the potential damage such a demo program could cause, giving it less than full system access. In another example, caching applets on the local file system will improve performance, but caching should not change the security model by treating cached applets as trusted code, even though they now reside on the local system. Furthermore, the distinction between what is local code and what is remote code is blurry. In the modern world of software components, one application could use multiple components, such as JavaBeans, that reside in all corners of the Internet. So security checks must be extended to all Java programs: to include applications as well as applets.

In Java 2, all code—whether local, remote, signed, or unsigned—is subjected to the same security controls. Thus, users can choose to give full or limited system access, based on the properties of the code and who is running the code. Such a choice is expressed by configuring a suitable security policy.

3.1.5 Robust and Simple Internal Security Mechanisms

In JDK 1.0 and JDK 1.1, a number of internal security mechanisms were designed and implemented, using techniques that were rather fragile. Although they worked reasonably well, maintaining and extending them proved difficult. For Java 2, we made important internal structural adjustments to reduce the risks of creating subtle security holes in the Java runtime and application programs. This involved revising the design and implementation of the `SecurityManager` and `ClassLoader` classes, as well as the underlying access control mechanism.

3.2 Overview of the Java 2 Security Architecture

The security architecture introduced in the Java 2 platform uses a security policy to decide which individual access permissions are granted to running code. These permissions are based on the code's characteristics, such as who is running the code, where it is coming from, whether it is digitally signed, and if so by whom. Attempts to access protected resources invoke security checks that compare the granted permissions with the ones needed for the attempted access. If a security policy is not explicitly given, the default policy is the classic sandbox policy as implemented in JDK 1.0 and JDK 1.1. The various caveats, refinements, and exceptions to this model are discussed in later chapters.

The Java 2 security architecture does not invent a new computer security theory, even though we had to design new ways of dealing with many subtle security issues that are unique to object-oriented systems. Instead, it offers a real-world

3.3 Architecture Summary

example in which well-known security principles [36, 96, 127, 111] are put into engineering practice to construct a practical and widely deployed secure system.

Chapters 5 and 6 describe the details of the policy enforcement implementation classes. The major components of the security model are security policy, access permissions, protection domain, access control checking, privileged operation, and class loading and resolution. Security policy and access permissions define what actions are allowed, whereas protection domain and access control checking provide the enforcement. Privileged operation and class loading and resolution are valuable assistants in the overall protection mechanisms.

Beyond the security model itself, we describe the implementation classes that support authentication, integrity protection, and confidentiality controls. Additionally, we provide the details of how to customize the security architecture to enhance or modify the context evaluated when making a security decision. We describe best practices to minimize the risk of introducing a security vulnerability in custom code, and we tell how to deploy the Java runtime to ensure that it too is secure from unauthorized modification.

3.3 Architecture Summary

As a summary of the overall process of how the Java 2 security architecture works, this section takes you through the handling of an applet or application. The following steps occur when viewing an applet, either through a Web browser or appletviewer or when running a Java application, possibly from the command line by invoking the program called `java`.

1. A class file is obtained and accepted if it passes preliminary bytecode verification.
2. The class's code source is determined. If the code appears to be signed, this step includes signature verification.
3. The set of static permissions, if any, to be granted to this class is determined, based on the class's code source.
4. A protection domain is created to mark the code source and to hold the statically bound permission set. Then the class is loaded and defined to be associated with the protection domain. *Note:* If a suitable domain has previously been created, that `ProtectionDomain` object is reused, and no new permission set is created.

5. The class may be instantiated into objects and their methods executed. The runtime type-safety check continues.
6. When a security check is invoked and one or more methods of this class are in the call chain, the access controller examines the protection domain. At this point, the security policy is consulted, and the set of permissions to be granted to this class is determined, based on the class's code source and principals, specifying who is running the code. In this step, the `Policy` object is constructed, if it has not been already. The `Policy` object maintains a runtime representation of the security policy.
7. Next, the permission set is evaluated to see whether sufficient permission has been granted for the requested access. If it has been granted, the execution continues. Otherwise, a security exception is thrown. (This check is done for all classes whose methods are involved in a thread. See Chapter 6 for the complete algorithm.)
8. When a security exception, which is a runtime exception, is thrown and not caught, the Java virtual machine aborts.

The delaying tactics described earlier help to reduce start-up time and the footprint of the runtime because objects are not instantiated until they must be used.

There are variations to this flow of actions. For example, in an eager approach, the creation of the `Policy` object and permissions can happen when classes are loaded into the runtime. This was the approach taken prior to Java 2 version 1.4.

The fundamental ideas adopted in the Java 2 security architecture have roots reaching into the past 40 years of computer security research: for example, the overall idea of the access control list [69]. We followed some of the UNIX conventions in specifying access permissions to the file system and other system resources. Most significantly, the design was inspired by the concept of protection domains and the work on handling mutually suspicious programs in Multics [110, 116] and “rights amplification” in Hydra [57, 134].

One novel feature not present in such operating systems as UNIX or Windows is the implementation of the least-privilege principle by automatically intersecting the sets of permissions granted to the protection domains involved in the call sequence. In this way, a programming error in system or application software is less likely to have an exploitable security hole.

Note that although it typically runs over a host operating system, such as Solaris, the Java virtual machine also may run directly over hardware, as in the case of the network computer JavaStation running JavaOS [101]. In general, to

3.3 Architecture Summary

maintain platform independence, the Java 2 architecture does not depend on security features provided by an underlying operating system.

Furthermore, this 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. This is effective, however, only if the underlying operating system itself has granted the user those permissions.

The protection mechanisms in Java 2 are language based and carried out within a single address space. This departure from more traditional operating systems is related to work on software-based protection and safe kernel extensions—for example, [12, 21, 117]—whereby research teams have targeted some of the same goals but by using different programming techniques. In a typical operating system, a cross-domain call tends to be quite expensive. In Java 2, a cross-domain call is just another method invocation.

The following are significant benefits of Java 2 platform security:

- ◆ The content of the security policy is totally separated from not only the implementation mechanism but also the interfaces. This leaves maximum room for evolution. It also allows the policy to be configured entirely separately from the runtime environment, thus reducing the complexity of system administration.
- ◆ The access control algorithm is cleanly separated from the semantics of the permissions it is checking. This allows the reuse of the access controller code with—perhaps application-specific—permission classes that are introduced after Java 2's release.
- ◆ The introduction of a hierarchy of permission classes brings the full power of object orientation, and especially encapsulation, to bear. This means that access control permissions can be expressed both statically and dynamically and that each `Permission` class may define its own semantics: for example, how it relates to a permission of its own type or of a different type or how to interpret wildcard symbols and other peculiarities that are specific to it.
- ◆ The secure class loading mechanism, coupled with the delegation mechanism, extends security coverage to Java applications, thus resulting in a uniform security architecture and policy for any and all Java code, whatever its origin or status.

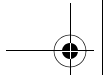
3.4 Lessons Learned

Over the course of developing Java 2 platform security, great attention has been given to interface design, proper division of labor among classes, minimizing the number of classes and APIs, and, whenever possible, implementing private classes and methods for the sake of clarity and protection. Typically, we do not start to prototype code until we have a good grasp of the APIs. This was especially true when we first began developing Java 2. Throughout the development life cycle, we have responded to comments and suggestions and have made extensive revisions to APIs throughout the project, all without much difficulty and without jeopardizing code quality or project delivery. When we made the initial transition from JDK 1.x we superseded fragile features, such as those methods we deprecated in the `SecurityManager` class, with a more robust architecture.

We did encounter in JDK 1.0 two artifacts that, although inconvenient, were not changed. First, system classes have been traditionally loaded with a “primordial” class loader, which is now formally referred to as the bootstrap class loader. If you asked for the class loader of a system class, you were given a `null`. This became a sort of de facto API; that is, a class having a `null` class loader was a system class. Some programmers started to test for the existence or nonexistence of class loaders as a way to distinguish between system and nonsystem classes, especially as part of the security decision-making process. For backward compatibility, Java 2 provides that system classes are still loaded by the bootstrap class loader; if a `null` is returned as the class loader for a class, it means that the bootstrap class loader loaded the class. *Note:* System classes are now sometimes referred to as bootstrap classes, but we will continue to refer to them as system classes in this book.

This association between system classes and the `null` class loader, coupled with the difference in treatment of classes based on their class loader types, however, makes it difficult to subdivide system classes into various packages or JAR files and then give them separate sets of permissions. Such a subdivision can reduce the amount of code you need to trust completely, as well as reduce the amount of trust in that code. In Java 2, application classes residing on the local file system must now be loaded with non-`null` class loaders. Further, a class being loaded with a non-`null` class loader does not say anything about its status, as the class might have been granted `AllPermission`. Hindsight tells us that it would have been much easier to evolve the design if all system classes had been originally loaded with a special, but non-`null`, class loader.

The second JDK 1.0 artifact that is not changed is that the runtime system does not always have a security manager installed, and in this case, a call to `System.getSecurityManager` results in a `null` security manager. Again, for backward compatibility, we have not changed this in Java 2. However, this oddity



3.4 *Lessons Learned*

causes a few unnecessary complexities. For example, each invocation of a security check must be preceded with a test for a `null` security manager; this clutters code. Historically, programmers have tested for a `null` security manager as a means of determining the state of the universe, rather like trying to distinguish the world before and after the so-called big bang. This has led to unwarranted assumptions of how a virtual machine should behave when the security manager is `null`, partly because no security checks can be invoked on a `null` security manager. These assumptions should not have been permitted at a general level; nevertheless, they are being made by some programmers. The presence of such assumptions creates pressure on maintaining backward compatibility.

The `AccessController` class makes it possible to invoke security checks in the absence of a security manager, but such checks may need to be deployed gingerly for fear of breaking backward compatibility. It would have been easier for us if the security manager had always been installed—that is, immediately after the bootstrap process—even though its behavior might change over time.

The lesson learned from these two artifacts is that one cannot easily evolve the interface design of something that is `null`. Further, you definitely cannot invoke method calls on something that is `null`.

