

[About Us](#)[Consulting Services](#)[CrossTalk](#)[Conference](#)[Resources](#)[Search](#)

CrossTalk Only

## Software Technology Support Center

[Home](#) > [CrossTalk Jul 2006](#) > [Article](#)

Jul 2006 Issue

**CROSSTALK**

The Journal of Defense Software Engineering

### About CrossTalk

- Mission
- Staff
- Contact Us

### Current Issue

### Subscription

- [Subscribe Now](#)
- [Update](#)
- [Cancel](#)
- [RSS 2.0](#)

### Theme Calendar

### Author Guidelines

### Back Issues

### Article Index

### Your Comments

## The New Java Security Architecture

**Idongesit Mkpong-Ruffin, Department of Computer Science and Software Engineering, Auburn University****Dr. John A. Hamilton, Jr., Department of Computer Science and Software Engineering, Auburn University****Dr. Martin C. Carlisle, Department of Computer Science, United States Air Force Academy**

*Java's original security architecture was designed to facilitate the execution of software from remote systems while simultaneously preventing downloaded code from performing unauthorized operations on host machines. The sandbox model of the Java Development Kit (JDK) 1.0's security architecture was found to be too restrictive; therefore the model was modified so that remote code could be allowed as trusted code. In the Java 2 platform, also known as JDK 1.2, the notion of trusted code was removed and security control mechanisms were implemented that could be applied to both application and applet code so the code could be run with configurable trust. Creating applications in Java does not guarantee that they are secure. Java developers need to understand and incorporate the new Java security architecture into their development process to make certain their applications are secure. This article looks at the implementation of the new architecture and the new mechanisms provided for ensuring security for Java code. It details the motivation for the security changes in a security architecture, gives a general overview of the architecture added, and then looks at some of the details of the mechanisms either changed or provided by the new architecture.*

Java's original security architecture was designed around the following four security elements:

1. **Language and Compiler.** Pointers were removed from the Java language; memory allocation and layout decision were moved to runtime.
2. **Bytecode Verifier.** A methodology for validating the safety of the code was established.
3. **Class Loader.** A separate namespace was allocated and the class loader was utilized to make sure that classes (applets) loaded from the network executed within their own namespace and did not interfere with the running of other programs.
4. **Interface-Specific Security.** Tools for implementing different levels of security were provided.

In the sandbox model (see Figure 1a), all the remote code was placed within a fixed, boundary-protected domain without access to host machine file systems, network connection, or other resources. So an applet within the sandbox could not read or write to a local disk, create a new process, load a new dynamic library, directly call a native method, or make a network connection to any host other than the one from which the applet came [1].

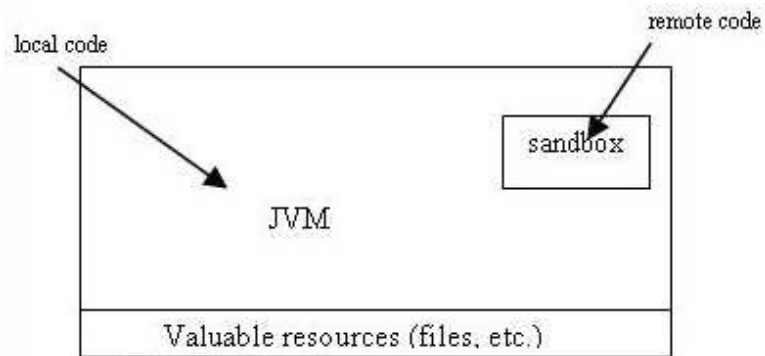


Figure 1a Original Sandbox

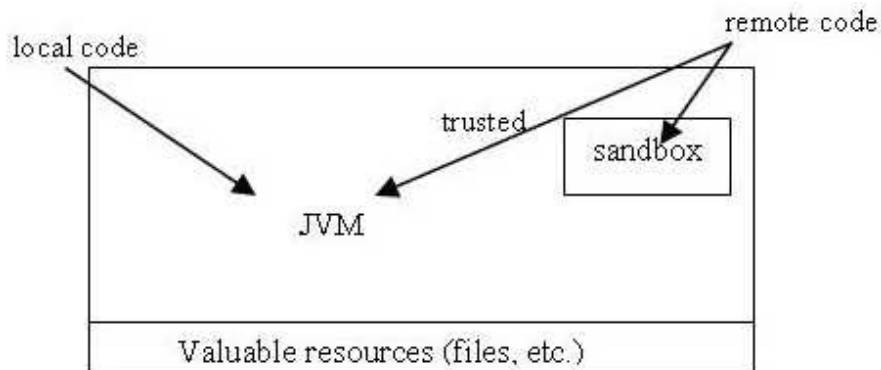


Figure 1b JDK 1.1 Security Model

As the language evolved, modifications and enhancements were made to the security model. In JDK 1.1, the concept of a *signed applet* was implemented. This allowed *correctly digitally signed* applets to be treated as trusted local code if the signature key was recognized as trusted by the end system that received the applet (see Figure 1b). In Java version 1.2, major security features were added and the basic security architecture changed, as shown in Figure 2. The built-in notion of trusted code was removed and all code is subject to the same security control applied to applets with the ability to change policy to allow for code (application or applet) to run with additional trust [2] [3].

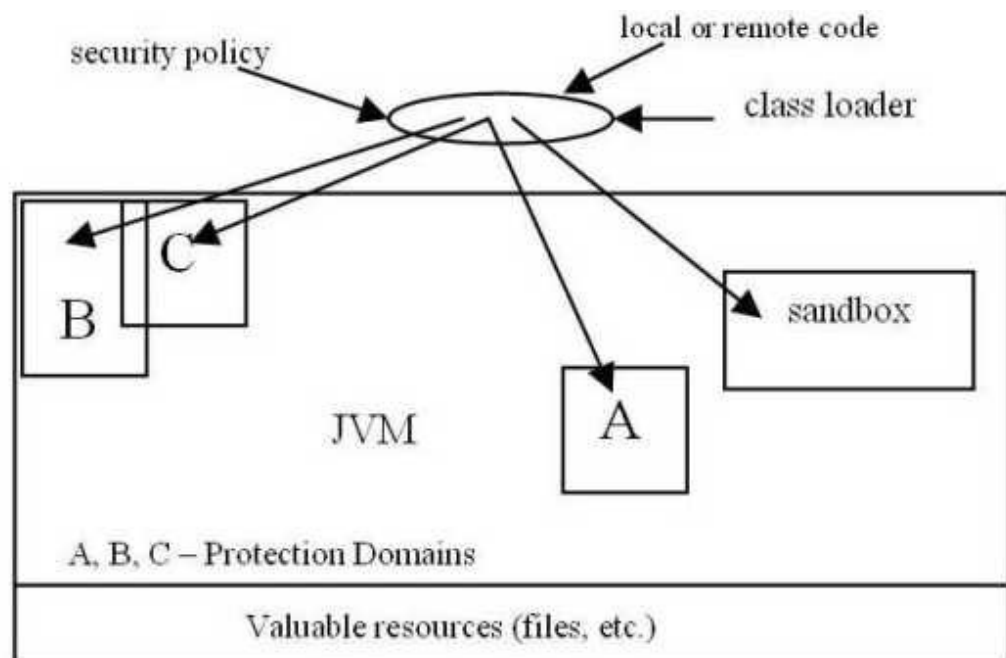


Fig 2 - Java 2 Security Model

## Major Components of the Java 2 Security Model

The following are the major components of the security model:

- Security policy.
- Access permission.
- Protection domain.
- Access control checking.
- Privileged operation.
- Class loading and resolution.

Security policy and access permissions define what actions are allowed, whereas protection domain and access control checking provide the enforcement.

## Security Policy

In Java 2, a security policy is utilized in deciding the individual access permissions to be given to running code. Deciding what permissions to grant depends on characteristics of the code: Where is the code coming from? Who is running it? Is it digitally signed? And if so, by whom? Security checks are invoked when attempts are made to access protected resources. These checks compare the permissions given to the required permissions for the attempted access. Where no security policy exists, the classic sandbox policy is used [4].

A policy object is used to represent a system security policy. It is usually instantiated during start up of the Java virtual machine (JVM), though it can be changed at a later time via a secure mechanism. The system security policy can be set by either the user or a system administrator. It is instantiated from the class `java.security.Policy`, and it maintains a runtime representation of the policy. Though multiple instances of the policy object can exist, only one can be in effect at any time. The `getPolicy` method allows the currently installed policy object to be obtained and the policy can be changed by issuing a call to the `setPolicy` method, assuming the code issuing the call has permission to reset the policy [2].

At the conceptual level, the security policy maps a set of properties that characterize running code to a set of access permissions granted to that code. The running code is characterized by its origin (the location of which is specified by a URL) and the set of public keys that match the set of private keys used to sign the code (the signature being based on one or more digital signature algorithms).

These properties are stored in the class `java.security.CodeSource`. A programming Application Program Interface (API) is used to set the policy within the Java runtime.

The policy information is dependent on the policy implementation, the configuration of which may be stored in a database, the serialized binary file of the policy class, or as a flat ASCII file. By default, a single system-wide and a single user policy file exist. When neither a system nor a user policy file exists, a built-in policy is used. This built-in policy is the classic sandbox policy. The policy file contains information about the permissions granted and the authentication of public keys.

## Permissions

To represent access to system resources, permission classes are used. An abstract class, `java.security.Permission`, is subclassed as needed for specific access. For example, `perm = new java.io.FilePermission("/tmp/abc", "read");` could be used to produce a permission to read file `abc` in the `/tmp` directory. Subclassed permissions apart from `BasicPermission`, which is part of the `java.security.BasicPermission`, usually belong to their packages.

In the example above, `FilePermission` (for file system access) is found in the `java.io` package. Other permission classes included in this security architecture are: `java.net.SocketPermission` (to access network resources), `java.lang.RuntimePermission` (used to allow access to runtime system resources, e.g. properties), and `java.awt.AWTPermission` (for access to windowing resources).

It is crucial that each new permission class implements the `implies` method. The `implies` method is utilized when access control decisions are made. Basically, `a implies b` means that if one is granted permission `a`, one is naturally granted permission `b` [2]. If an applet is granted permission to write to the entire file system, then it can overwrite the system binary, including the JVM runtime. In this scenario, the applet has been granted all permissions. Granting an applet runtime permission to create class loaders, has the effect of granting a lot of other permissions, since a class loader can perform sensitive operations. Care should be taken when granting permissions, as the Java security architecture does not prevent malicious behavior at the native code level. Therefore, permissions that allow either the setting of system properties or runtime permissions for loading native code libraries or defining packages are *dangerous*.

## Protection Domains

The main building block for Java's system security architecture is the concept of protection domains. Domain boundaries are established by the *set of objects that can be directly accessed by a principal at the present time* where a principal is *an entity in the computer system to which permissions (and as a result, accountability) are granted* [2]. Protection domains serve as a mechanism for grouping and isolating units of protection. The sandbox approach utilized the fixed boundary protection domain. In the new architecture, it is possible, though not a built-in feature, to separate protection domains so that they do not interact with each other. Any permitted interaction has to be through trusted system code or explicitly allowed by the domains in question.

There are two discrete groups to which protection domains generally belong: system and application. All protected external resources (for example, the screen, keyboard, files system, or networking facility) should be accessible via system domains.

A domain encapsulates *a set of classes whose instances are granted the same set of permissions* [2]. The security policy in effect determines the protection domain in effect. A mapping from code to domain and their permissions is kept by the application environment

A thread of execution can occur either solely within a single protection domain or involve an application domain and the system domain. The application domain should not be able to gain additional permissions by calling the system domain. Also, if a system domain invokes a method from an application domain, it is also pertinent that the effective access rights are the same as the rights currently enabled in the application domain. Consequently, a *less powerful* domain is not able to gain additional permission because it calls or is called by a more powerful domain.

The *doPrivileged* method allows a piece of trusted code to temporarily have access to more resources made available directly to the application that called the trusted code. When a critical system resource (e.g. file and/or network input/output) is requested during execution, an *AccessController* class method is invoked by the resource-handling code, either directly or indirectly. This method evaluates the request and decides whether the request will be granted or denied. The evaluation entails examination of the call history and the permissions that have been granted to the pertinent protection domains. Upon completion of the evaluation, if the request is granted, the *doPrivileged* method returns normally; otherwise, a security exception is thrown.

## Domain-Based Access Control

The JDK 1.2 utilizes a new class: *java.security.AccessController* for the automation of the access checking process. The following are the uses for the access controller class:

1. Decides whether to allow or deny access to a critical system resource based on the security policy currently in effect.
2. Marks code as *privileged*, which affects subsequent access determination.
3. Obtains a *snapshot* of the current calling context which allows access-control decisions from a different context to be made with respect to the saved context [2].

In JDK 1.2, the burden of discovery of the history of callers is given to the *AccessController*. Any code can now query the access controller to determine whether permission would succeed if performed at the present time by calling on the *checkPermission* method.

### For example:

```
FilePermission perm2 = new FilePermission("path/file", "read");
AccessController.checkPermission(perm);
```

Access rights are determined in Java 2 by using the permission class hierarchy; the root of the class hierarchy is the abstract class *java.security.Permission*. The default permissions are access *approval* rather than *denial* [5].

Though *AccessController* automates the access checking process, the determination of access rights is still left to the discretion of the user. The access rights, so code may execute, have to be determined before an application or library code is deployed in Java. Presently this is done by experimentation.

The developer reads documentation for libraries used (including the Java run-time libraries) and deduces the required access rights. Unfortunately, this documentation is often missing, misleading, or out of date. In the absence of reliable documentation, the developer executes the new code and observes authorization failures. The developer then grants additional access rights and retests. The developer repeats this process, possibly many times, until there are no authorization failures. However, required access rights

requirements can remain undiscovered due to an insufficient number of test cases, rendering the code unstable. [5]

A similar situation occurs when mobile code is dynamically installed and the user/client has to decide the set of access rights to provide. To do this, the user usually assumes the recommendations of the developer/distributor can be relied upon and grants those permissions. Also, the user could run code with fewer permissions, look at failures, and determine whether to add access rights incrementally or not. One can see that this approach is tedious and error prone [5].

Security in language design is of great concern due the increased usage of the internet and the ease with which code can be transported. As code from unreliable sources is used on a local system, it is vital that the local system be protected from the non-local code, as these sources could be hostile [6]. The Java Security Architecture is implemented via a set of methods. It is a dynamic security checking system. It checks access restrictions at run-time as opposed to compile-time. This is accomplished via stack inspection, which is at the heart of the JDK security architecture. To accomplish this, the programmer adds *doPrivileged* and *checkPrivileged* commands to the code.

The *doPrivileged* command adds a flag to the caller's stack frame, which is eliminated when the frame returns. When a privilege is checked via *checkPrivilege* command, the stack frames are searched most to least recent. If a frame is encountered with the desired flag, the check succeeds. Also, all programs in JDK 1.2 come with a specified owner. This means that stack frames have owners, which may be unreliable for some privilege; if such an owner is encountered before success, the check fails [6].

## Revised Security Manager

The security manager has been revised so that the *AccessController* is invoked whether or not there is a class loader associated with the calling class. Prior to JDK 1.2, access control could have been invoked as follows:

```
ClassLoader loader = this.getClass().getClassLoader(); if (loader != null) {
  SecurityManager security = System.getSecurityManager(); if (security !=
  null) { security.checkRead("path/file"); } }
```

Now, with the new security manager, this can be written much more simply as:

```
FilePermission perm = new FilePermission("/temp/abc", "read");
AccessController.checkPermission(perm) [2]
```

The revised security manager still has backward compatibility, so that all *check()* methods within security manager are still supported, but the default implementations for the *check()* methods have been changed to invoke *accesscontroller* with the correct permission object when possible.

Since the security manager was used prior to Java 2 for protection of applets, it is automatically installed when an applet is running. When an application is running, the user running the application or the application itself must install the security manager. There are two ways to apply the same restrictive downloaded applet security policy to applications found on the local file system. Either the user invokes the JVM with *-Djava.security.manager* (e.g. *java -Djava.security.manager SomeApp*) or the application must call the *setSecurityManager* method, which is in the *java.lang.System* class, to install a security manager.

## Security Manager versus AccessController

The security manager allows for a central point of access control while *AccessController* facilitates a specific access control algorithm, using special features like the *doPrivileged* method. The security manager class provides flexibility (for example, for those wanting to customize the security model for either multilevel security or to allow for mandatory access control) and maintains backward compatibility (for example, those applications that were written with their own security manager classes based on previous versions of the JDK). *AccessController* is provided to alleviate the burden of having to write *extensive security code* for the typical programmer in many cases.

## Secure Class Loading

The class loader provides security in Java by being responsible for finding and getting the class file, verifying the security policy, and defining the class object with the appropriate permissions. The class loading mechanism is dynamic and its dynamic nature gives the Java platform ability to install software components at run-time.

Since there can be many instances of class loader objects, and in some cases these class-loader classes have distinct properties; it becomes important to decide which class loader to use and what type of class loader that should be used. In Java 2, class

*java.security.SecureClassLoader* has been introduced as a concrete implementation of the abstract class *java.lang.ClassLoader* that was originally defined in JDK 1.0.

All applets and applications load classes either directly using the provided *SecureClassLoader* or by asking another class loader to perform the operation. To accomplish this, the class loader's local cache or something functionally equivalent to that can be checked to see if a loaded class matches the target class. If the current class loader has a stated delegation parent, then delegate to the parent to try to load the class. Where there is no parent, then it will delegate to the system class loader. If there is no stated delegation parent, a check is made on whether a customizable method for finding the class exists. If there is a customizable method for finding the class, then the custom class loader could override this method and specify how a class should be looked up.

The process given above answers the question – how do we find the class to load? With many instances of class loader objects, it is important that a way to determine which class loader to use be established, and the steps given above do not address this issue. Rules to determine the class loader are the following:

- When loading the first class of an application, a new instance of the *URLClassLoader* is used.
- When loading the first class of an applet, a new instance of the *AppletClassLoader* is used.
- When *java.lang.Class.forName* is directly called, the base class loader is used.
- If the request to load a class is triggered by a reference to it from an existing class, the class loader for the existing class is asked to load the class.

Note that rules about the use of *URLClassLoader* and *AppletClassLoader* instances have exceptions and can vary depending on the particular system environment. For example, a web browser may choose to reuse an existing *AppletClassLoader* to load applet classes from the same web page.

Due to the power of class loaders, Java 2 severely restricts what code can create class-loader instances. It provides a mechanism for applications or applets to indicate URL locations from which classes can be loaded. This is provided in the form of static methods that allow programs to create instances of the *URLClassLoader* but not other types of class loaders [2].

## Architecture Summary

To better understand how Java 2 security architecture works, let's look at how an applet or application is handled. We start by viewing an applet through a web browser or applet viewer or by running an application using the *java* program. The following steps will occur:

1. The class file is put through preliminary bytecode verification. If it passes the bytecode verification, the class file is gotten and accepted.
2. The class' 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' code source. Also, 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.
4. class may be instantiated into objects and their methods executed.
5. 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.
6. 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.
7. When a security exception, which is a run-time exception, is thrown and not caught, the JVM aborts.

Even though Java usually runs over a host operating system, such as Solaris, the Java virtual machine may run directly over hardware. The Java 2 architecture does not depend on security features provided by an underlying operating system. This allows for maintenance of platform independence.

## Conclusion

In creating the new security model, the new Java 2 architecture provides more flexible

access control, separates policy expression from policy enforcement, and makes policy enforcement extensible and flexible, thereby allowing for a flexible and customizable security policy. With the Java 2 security architecture, the need to write custom security code for all but the most specialized environments is removed. The Java 2 security architecture separates policy expression/description from the enforcement mechanism thereby giving an avenue to support "easily configurable security policies" [4]. The Java 2 architecture provides an easily extensible access control structure by the provision of typed access control permissions and an automatic permission-handling mechanism. In addition, a more general *checkPermission* method has been added in Java 2 to handle security checks. The new security policy subjects all code —whether local, remote, signed, or unsigned—to the same security controls. The client then has the ability to decide when and to whom to give full or limited system access. This choice is given by giving the client/user the ability to configure the desired/suitable security policy. The design and implementation of the security manager and class-loader classes and the underlying access control mechanism have been revised, so as to reduce the risks of creating *subtle* security holes in the Java runtime of application programs.

## References

1. Gritzalis, Stefanos, and George Aggelis. "Security Issues Surrounding Programming Languages for Mobile Code: JAVA vs. Safe-Tcl." *ACM SIGOPS Operating Systems Review* Apr. 1998:16-32.
2. Gong, L. "Java 2 Platform Security Architecture" 1999 <<http://java.sum.com/j2se/1.5.0/docs/guide/security/spec/security-spec.doc.html>>.
3. McGraw, G., and E. Felten. *Securing Java: The Base Java Security Model: The Original Applet Sandbox*. John Wiley and Sons, 1999 <[www.securingsjava.com/chapter-two](http://www.securingsjava.com/chapter-two)>.
4. Gong, L. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. 2nd ed. Addison-Wesley, 2003.
5. Koved, Larry, Marco Pistoia, and Aaron Kershenbaum. "Access Rights Analysis for Java." *Proc. of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Nov. 2002: 359-372.
6. Skalka, C., and S. Smith. "Static Enforcement of Security with Types," *Proc. of the Fifth ACM SIGPLAN International Conference on Functional Programming*. 2000: 34-45.

---

## About the Authors



**Idongesit Mkpog-Ruffin** has an extensive background as an instructor in Computer Science and Business and as a technology consultant. She is currently a Ph.D. student enrolled at Auburn University. She has a Bachelor of Science in Computer Information Science from Freed-Hardeman University, a Master of Business Administration from Tennessee State University and a Master of Science in Computer Information Science from Troy University, Montgomery campus.



**John A. "Drew" Hamilton, Jr., Ph.D.**, is an associate professor of computer science and

software engineering at Auburn University and director of Auburn University's Information Assurance Laboratory. He has a Bachelor of Arts in journalism from Texas Tech University, masters degrees in systems management from the University of Southern California and in computer science from Vanderbilt University, and a doctorate in computer science from Texas A&M University.

Auburn University  
107 Dunstan Hall  
Auburn, AL 36849  
Phone: (334) 844-6360  
Fax: (334) 844-6329  
E-mail: [hamilton@eng.auburn.edu](mailto:hamilton@eng.auburn.edu)



**Martin C. Carlisle, Ph.D.**, is an associate professor of computer science at the United States Air Force Academy and an affiliate member of the Auburn University Information Assurance Laboratory. He has a Bachelor of Science in mathematics and computer science from the University of Delaware, and a Master of Arts and a doctorate in computer science from Princeton University.



---

[Privacy and Security Notice](#) · [External Links Disclaimer](#) · [Site Map](#) · [Contact Us](#)

Please [E-mail](#) or call 801-775-5555 (DSN 775-5555) if you have any questions regarding your [CrossTalk subscription](#) or for additional STSC information.

**Webmaster:** 517th SMXS/MDEA, 801-777-0857 (DSN 777-0857), [E-mail](#)

**STSC Parent Organizations:** [309SMXG](#) Ogden Air Logistics Center, Hill AFB