In Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, Dec., 1997

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.

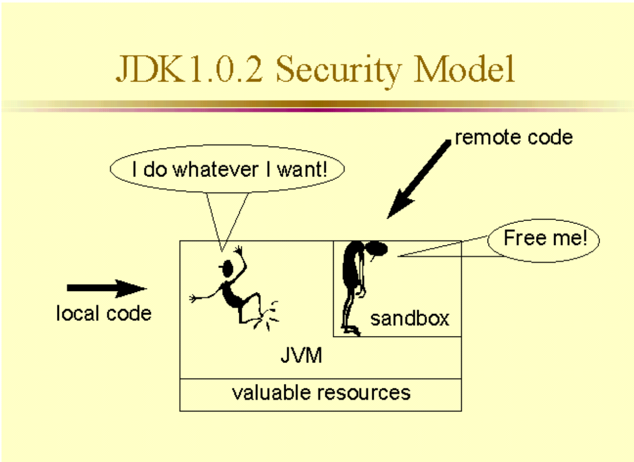

Figure 1: JDK1.0.x Security Model

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.

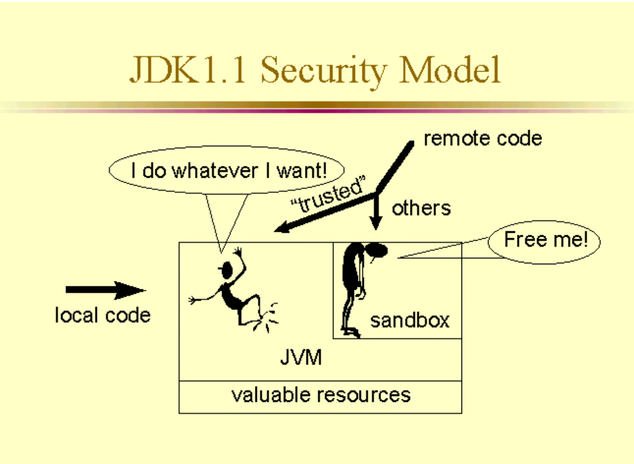

Figure 2: JDK1.1 Security Model

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).

  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

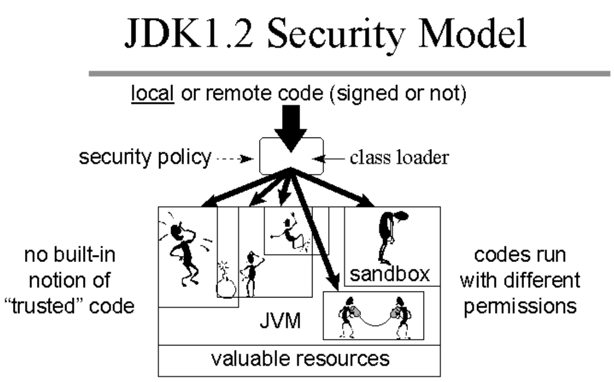

Figure 3: JDK1.2 Security Model

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 not to classes and objects directly.[2] In other words, a domain can be scoped by the set of objects that correspond to a principal, where a principal is an entity in the computer system to which authorizations (and as a result, accountability) are granted [16]. The Java sandbox in JDK1.0.2 is one example of a protection domain with a fixed boundary.

In JDK1.2, protection domains are created "on demand", based on code source. Each class belongs to one and only one domain. The Java runtime maintains the mapping from code (classes and objects) to their protection domains and then to their permissions.

Protection domain also serves as a convenient point for grouping and isolation between units of protection within the Java runtime. For example, it is possible to separate different domains from interacting with each other. Any permitted interaction must be either through system code or explicitly allowed by the domains concerned.

The above point brings up the issue of accessibility, which is orthogonal to security. In the Java virtual machine, a class is distinguished by itself plus the class loader instance that loaded the class. In other words, a class loader defines a distinct name space and can be used to isolate and protect code within one protection domain if the loader refuses to load code from different domains (and with different permissions).

On the other hand, it is sometimes desirable to allow code from different domains to interact with each other – for example, in the case of an application made up from Java Beans signed by different public keys, the beans should be able to access each other (which is the purpose of the application) although the runtime environment may insist that different beans are loaded into different domains. The `AppletClassLoader` class used by the `appletviewer` in JDK1.2 will load classes from different domains.

One protection domain is special: the system domain, which consists of system code that is loaded with a `null` class loader (basically all classes located on `CLASSPATH`) and is given special privileges. It is important that all protected external resources, such as the file system, the networking facility, and the screen and keyboard, are directly accessible only via system code.

[2] In the future, protection domains can be further characterized by user authentication and delegation so that the same code could obtain different permissions when running "on behalf of" of different principals.

2.4 Domain-Based Access Control

The decision of granting access to controlled resources can only be made within the right context, which must provide answers to questions such as "who is requesting what, on whose behalf". Often, a thread is the right context for access control. Less frequently, access control decisions have to be carried out among multiple threads that must cooperate in obtaining the right context information. This section focuses on the former, as it is the most common case encountered in building JDK1.2.

A thread of execution may occur completely within a single protection domain (i.e., all classes and objects involved in the thread belong to the identical protection domain) or may involve multiple domains such as an application domain and also the system domain.

For example, an application that prints a message out will have to interact with the system domain that is the only access point to an output stream. In this case, it is crucial that at any time the application domain does not gain additional permissions by calling the system domain. Otherwise, there can be security serious implications.

In the reverse situation where a system domain invokes a method from an application domain, such as when the AWT system code calls an applet's `paint` method to display the applet, it is again crucial that at any time the effective access rights are the same as current rights enabled in the application domain.

In other words, a less "powerful" domain cannot gain additional permissions as a result of calling a more powerful domain; whereas a more powerful domain must lose its power when calling a less powerful domain. This principle of least privilege is applied to a thread that transverses multiple protection domains.

Up to JDK1.1, any code that performs an access control decision relies on explicitly knowing its caller's status (i.e., being system code or applet code). This is fragile in that it is often insufficiently secure to know only the caller's status but also the caller's caller's status and so on. At this point, placing this discovery process explicitly on the typical programmer becomes a serious burden, and can be error-prone.

To relieve this burden by automating the access checking process, JDK1.2 introduces a new class `java.security.AccessController`. Instead of trying to discover the history of callers and their status within a thread, any code can query the access controller as to whether a permission would succeed if performed right now. This is done by calling the `checkPermission` method of the `AccessController` class with a `Permission` object that represents the permission in question.

By default, the access controller will return silently only if all callers in the thread history (e.g., all classes on the call stack) belong to domains that have been granted the said permission. Otherwise, it throws a `java.security.AccessControlException`, which is a subclass of `java.lang.SecurityException`, usually printing the reason of denial.

This default behavior is obviously the most secure but is limiting in some cases where a piece of code wants to temporarily exercise its own permissions that are not available directly to its callers. For example, an applet may not have direct access to certain system properties, but the system code servicing the applet may need to obtain some properties in order to complete its tasks.

For such exceptional cases, we provide a primitive, via static methods `beginPrivileged` and `endPrivileged` in the `AccessController` class. By calling `beginPrivileged`, a piece of code is telling the Java runtime system to ignore the status of its callers and that it itself is taking responsibility in exercising its permissions.

To summarize, a simple and prudent rule of thumb for calculating permissions is the following:

- The permission of an execution thread is the intersection of the permissions of all protection domains transversed by the execution thread.
- When some code calls the `beginPrivileged` primitive, the permission of the execution thread includes a permission if it is allowed by the said code's protection domain and by all protection domains that are called or entered directly or indirectly subsequently.
- When a new thread is created, it inherits from its parent thread the current security context (i.e., the set of protection domains present in the parent at child creation time). This inheritance is transitive.

In following the above rule, the access controller examines the call history and the permissions granted to the relevant protection domains, and to return silently if the request is granted or throw a security exception if the request is denied.

There are two obvious strategies for implementing this access control rule. In an "eager evaluation" implementation, whenever a thread enters a new protection domain or exits from one, the set of effective permissions is updated dynamically. The benefit is that checking whether a permission is allowed is simplified and can be faster in many cases. The disadvantage is that, because permission checking occurs much less frequently than cross-domain calls, a large percentage of permission updates may be useless effort.

JDK1.2 employs a "lazy evaluation" implementation where, whenever a permission checking is requested, the thread state (as reflected by the current thread stack or its equivalent) is examined and a decision is reached to either deny or grant the particular access requested. One potential downside of this approach is performance penalty at permission checking time, although this penalty would have been incurred anyway in the "eager evaluation" approach (albeit at earlier times and spread out among each cross-domain call). In our implementation, performance of this algorithm is quite acceptable[3], so we feel that lazy evaluation is the most economical approach overall.

2.5 Revised SecurityManager

Up to JDK1.1., when access to a critical system resource (such as file I/O and network I/O) is requested, the resource handling code directly or indirectly invokes the appropriate `check` method on the installed `java.lang.SecurityManager` to evaluate the request and decide if the request should be granted or denied.

JDK1.2 maintains backward compatibility in that all `check()` methods in `SecurityManager` are still supported, but we have changed their default implementations to invoke `AccessController`, whenever feasible, with the appropriate permission object. This class, which has been abstract up to JDK1.1.x, is made concrete in JDK1.2.

To illustrate the usage of the new access control mechanism, let us examine a small example for checking file access. In earlier versions of the JDK, the following code is typical:

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

[3] For details of the implementation of protection domain, and a discussion on performance and optimization techniques, please refer to [7].

Under the new architecture, the check typically should be invoked whether or not there is a classloader associated with a calling class. It should be simply:

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

Note that there are legacy cases (for example, in some browsers) where whether there is an instance of the `SecurityManager` class installed signifies one or the other security state that may result in different actions being taken. We currently do not change this aspect of the `SecurityManager` usage, but would encourage application developers to use the techniques introduced in this new version of the JDK in their future programming.

Moreover, we have not revised system code to always call `AccessController` (and not checking for the existence of a classloader), because of the potential of existing software subclassing the `SecurityManager` and customizing these check methods.

To use the privilege primitive, the following code sample should be followed:

```
try {
    AccessController.beginPrivileged();
    (some sensitive code)
} finally {
    AccessController.endPrivileged();
}
```

Some important points about being privileged. Firstly, this concept only exists within a single thread. That is, a protection domain being so privileged is scoped by the thread within which the call to become privileged is made. Other threads are not affected.

Secondly, in this example, the body of code within try-finally is privileged. However, it will lose its privilege if it calls (from within the privileged block) code that is less privileged.

Moreover, although it is a good idea to use `beginPrivileged` and `endPrivileged` in pairs as this clearly scopes the privileged code, we have to deal with the case when `endPrivileged` is not called, because forgetting to disable a privilege can be very dangerous. To reduce or eliminate the risk, we have put in additional mechanism to safe guard this primitive.

2.6 Secure Class Loading

The class `java.security.SecureClassLoader` is a concrete implementation of the abstract class `java.lang.ClassLoader` that loads classes and records the protection domains they belong to. It also provides methods to load a class from bytecode stored in a byte array, an URL, and an `InputStream`. This class can be extended to include new methods, but most existing methods are final, as this class is significant for security.

All applets and applications (except for system classes) are loaded by a SecureClassLoader either directly or indirectly (in which case, it is probably loaded by another classloader that itself is loaded by a SecureClassLoader).

`SecureClassLoader`'s `loadClass` methods enforce the following search algorithm where, if the desired class (by the given name) is not found, the next step is taken. If the class is still not found after the last step, a `ClassNotFoundException` is thrown.

1. See if the class is already loaded and resolved
2. See if the class requested is a system class. if so, load the class with the null system classloader.
3. Attempt to find the class in a customizable way, using a non-final method `findAppClass`, which by default will try to find the class in a second local search path that is defined by a property named `java.app.class.path`.

Note that in step 2, all classes on the search path CLASSPATH are treated as system classes, whereas in step 3, all classes on the search path `java.app.class.path` are considered non-system classes.[4]

Programmers who must write class loaders should, whenever feasible, subclass from the concrete `SecureClassLoader` class, and not directly from the abstract class `java.lang.ClassLoader`.

A subclass of `SecureClassLoader` may choose to overwrite the `findAppClass` method in order to customize class searching and loading. For example, the `AppletClassLoader` caches all raw class materials found inside a JAR file. Thus, it is reasonable for the `AppletClassLoader`, which is a subclass of the `SecureClassLoader`, to use `findAppClass` to look into its own cache. A class introduced in such a fashion is guaranteed not to be a system class, and is subjected to the same security policy as its loading class.

[4] The path `java.app.class.path` is currently specified in a platform dependent format. There might be a future need to develop a generic `Path` class that not only provides platform independent path names but also makes dynamical path manipulation easier.

Often a class may refer to an another class and thus cause the second class belonging to another domain to be loaded. Typically the second class is loaded by the same classloader that loaded the first class, except when either class is a system class, in which case the system class is loaded with a null classloader.

2.7 Extending Security to Applications

To apply the same security policy to applications found on the local file system, we provide a new class `java.security.Main`, which can be used in the following fashion in place of the traditional command `java application` to invoke a local application:

```
java java.security.Main application
```

This usage makes sure that any local application on the `java.app.class.path` is loaded with a `SecureClassLoader` and therefore is subjected to the security policy that is being enforced. Clearly, non-system classes that are stored on the local file system should all be on this path, not on the `CLASSPATH`.

3 Discussion

In this section, we discuss a number of open questions and possible improvement to the current architecture. But we start by discussing how a developer or user is impacted by the new architecture.

3.1 Utilizing the New Architecture

For a user of the built-in `appletviewer` or a new version of a browser that deploys this new security architecture, the user can continue to do things the same way as before, which means that the same policy in JDK1.1.x will apply.

On the other hand, a "power user" can use the `PolicyTool` built-in for JDK1.2 (or an equivalent one shipped with the browser) to customize the security policy, thus utilizing the full benefit of the new security architecture. Such customization may involve setting up a certificate store, which can be done via the `KeyTool`.

The typical application developer, in general, needs to do nothing special because, when the application is run on top of JDK1.2, the security features are invoked automatically. Except that the developer might want to use the built-in tools to package the resulting application into JAR files, and may choose to digitally sign them.

For a software library developer whose code controls certain resources, the developer may need to extend the existing permission class hierarchy to create application-specific permissions. The developer may also need to learn to use features provided by the `AccessController` class, such as the privilege primitive.

3.2 Handling Non-Class Content

When running applets or applications with signed content, the JAR and Manifest specifications on code signing allow a very flexible format. Recall that classes within the same archive can be unsigned, signed with one key, or signed with multiple keys. Other resources within the archive, such as audio clips and graphic images, can also be signed or unsigned.

This flexibility brings about the issue of interpretation. The following questions need to be answered, especially when not all signatures are granted the same privileges. Should images and audio clips be required to be signed with the same key if any class in the archive is signed? If images and audio files are signed with different keys, can they be placed in the same `appletviewer` (or browser page), or should they be sent to different viewers?

These questions are not easy to answer, and require consistency across platforms and products to be most effective. Our intermediate approach is to provide a simple answer – all images and audio clips are forwarded to be processed whether they are signed or not. This temporary solution will be improved once a consensus is reached.

3.3 Enabling Fine-Grained Privileges

The privileged primitive discussed earlier in a sense "enables" all permissions granted to a domain. We can contemplate to enrich the construct so that a protection domain can request to enable privilege for only some of its granted permissions. This should further reduce the security impact of making a programming mistake. For example, the code segment below illustrates how to turn on the privilege of only reading everything in the "/tmp" directory.

```
FilePermission p =
    new FilePermission("/tmp/*", "read");
try {
    AccessController.beginPrivileged(p);
    some sensitive code
} finally {
    AccessController.endPrivileged(p);
}
```

3.4 Extending Protection Domains

The first possibility is to subdivide the system domain. For convenience, we can think of the system domain as a single, big collection of all system code. For better protection, though, system code should be run in multiple system domains, where each domain protects a particular type of resource and is given a special set of rights. For example, if file system code and network system code run in separate domains, where the former has no rights to the networking resources and the latter has no rights to the file system resources, the risks and consequence of an error or security flaw in one system domain is more likely to be confined within its boundary.

Moreover, protection domains currently are created transparently as a result of class loading. It might be desirable to provide explicit primitives to create a new domain. Often, a domain supports inheritance in that a sub-domain automatically inherits the parent domain's security attributes, except in certain cases where the parent further restricts or expands the sub-domain explicitly.

Finally, each domain (system or application) may also implement additional protection of its internal resources within its own domain boundary. Because the semantics of such protection is unlikely to be predictable by the JDK, the protection system at this level is best left to the application developers. Nevertheless, JDK1.2 provides `SignedObject`, `Guard`, and `GuardedObject` classes that simplify a developer's task.

4 Summary and Future Work

This paper gives an overview of the motivation and the new security architecture implemented in JDK1.2. Although we do not break new theoretical ground in computer security, we attempt to distill the best practices from research in the past four decades, such as clear separation between security policy and implementation, and engineer them into a widely deployed programming platform. Our implementation has a number of novel aspects that demonstrate beyond the doubt the efficiency of language-based protection mechanisms. The success of this development effort also highlights the excellent extensibility of the Java platform.

In future releases, we are investigating user authentication techniques, an explicit principal concept, a general mechanism for cross-protection-domain authorization, and the "running-on-behalf" style delegation. We are also working towards additional features such as arbitrary grouping of permissions, the composition of security policies, and resource consumption management, which is relatively easy to implement in some cases, e.g., when limiting the number of windows any application can pop up at any one time, but more difficult in other cases, e.g., when limiting memory or file system usage.

Acknowledgments

Additional members of the JavaSoft security group, including Gigi Ankeny, Charlie Lai, Jan Luehe, and Jeff Nisewanger, made significant contributions during the course of the design and implementation of new security features in JDK1.2. Other members of the JavaSoft community, notably Josh Bloch, Sheng Liang, Roger Riggs, Nakul Saraiya, and Bill Shannon, provided invaluable insight, detailed reviews, and much needed technical assistance.

We are grateful for support from Dick Neiss, Jon Kannegaard, and Alan Baratz, for technical guidance from James Gosling, Graham Hamilton, and Jim Mitchell, and for indispensable collaboration from the testing and documentation groups. We received numerous suggestions from our corporate partners and licensees, whom we could not fully list here.

References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuchynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety, and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Colorado, December 1995. Published as ACM Operating System Review 29(5):251–266, 1995.

[2] J.S. Chase, H.M. Levy, M.J. Feeley, and E.D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[3] D. Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 18–27, Zurich, Switzerland, April 1997.

[4] D. Dean, E.W. Felten, and D.S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 190–200, Oakland, California, May 1996.

[5] M. Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold Co., New York, 1988.

[6] L. Gong. Java Security: Present and Near Future. *IEEE Micro*, 17(3):14–19, May/June 1997.

[7] L. Gong and R. Schemers. Implementing Protection Domains in the Java™ Development Kit 1.2. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, California, March 1998.

[8] J. Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Menlo Park, California, August 1996.

[9] A.K. Jones. *Protection in Programmed Systems*. Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA 15213, June 1973.

[10] B.W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971. Reprinted in ACM Operating Systems Review, 8(1):18–24, January, 1974.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Menlo Park, California, 1997.

[12] P.G. Neumann. *Computer-Related Risks*. Addison-Wesley, Menlo Park, California, 1995.

[13] U.S. General Accounting Office. Information Security: Computer Attacks at Department of Defense Pose Increasing Risks. Technical Report GAO/AIMD-96-84, Washington, D.C. 20548, May 1996.

[14] S. Ritchie. Systems Programming in Java. *IEEE Micro*, 17(3):30–35, May/June 1997.

[15] J.H. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[16] J.H. Saltzer and M.D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[17] M.D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1972.

[18] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, October 1996. Published as *ACM Operating Systems Review*, 30, special winter issue, 1996.

[19] T. Thorn. Programming Languages for Mobile Code. *ACM Cumpting Surveys*, 29(3):213–239, September 1997.

[20] W.A. Wulf, R. Levin, and S.P. Harbison. *HYDRA/C.mmp – An Experimental Computer System*. McGraw-Hill, 1981.

[21] F. Yellin. Low Level Security in Java. In *Proceedings of the 4th International World Wide Web Conference*, Boston, Massachusetts, December 1995.