

1. *Introducing JAAS*

JAAS, the Java Authentication and Authorization Service, has been a standard part of the Java security framework since version 1.4 version and was available as an optional package in J2SE 1.3. Before that time, the previous security framework provided a way to allow or disallow access to resources based on what code was executing. For example, a class loaded from another location on the Internet would have been considered less trustworthy and disallowed access to the local file system. By changing the security settings, this restriction could be relaxed and a downloaded applet could modify local files. Viewed mainly as a tool for writing clients, early Java didn't seem to need much more than this.

As the language matured and became popular for server-side applications as well, the existing security framework was deemed too inflexible, so an additional restriction criterion was added: *who was running the code*. If User A was logged in to the application, file access may be allowed, but not so for User B. Accomplishing this requires authentication and authorization. *Authentication* is the process of verifying the identity of users. *Authorization* is the process of enforcing access restrictions upon those authenticated users. JAAS is the subsection of the Java security framework that defines how this takes place.

Like most Java APIs, JAAS is exceptionally extensible. Most of the sub-systems in the framework allow substitution of default implementations so that almost any situation can be handled. For example, an application that once stored user ids and passwords in the database can be changed to use Windows OS credentials. Java's default, file-based mechanism for configuration of access rights can be swapped out with a system that uses a database to store that information. The incredible flexibility of JAAS and the rest of the security architecture, however, produces some complexity. The fact that almost any piece of the entire infrastructure can be overridden or replaced has major implications for coding and configuration. For example, every application server's JAAS customizations have a different file format for configuring JAAS, all of which are different from the default one provided by Java¹.

1.1 *User Access Control*

Suppose you're tasked with writing a web application that allows users to log in with an id and password and then allows the users to view their employment information. Because of the sensitivity of the data, it is important that employees not have access to each other's data. At this point, the protection logic is not very complex: only let the user that is currently logged in see the information that is mapped to himself. But now add the idea of a manager who may be able to see some of the other employees' items, such as salary or hiring date. Then add the idea of a human resources administrator. Then an accountant. Or an auditor.

¹ But, we're Java programmers, we live for that kind of stuff, right?



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

The CEO. All these users need access to different information and should not be allowed anything more than necessary.

Complex security domains like these are where JAAS comes in handy. Additionally, most application servers and servlet containers use JAAS to provide a way to pass login information into the application. The application can even take advantage of login methods provided by the application server and never deal with the interaction with the user.

By the end of this book, you will both understand and use the functionality in JAAS, and also be able to replace many of the pieces provided by the JDK or whatever application server you may be using with your own custom classes. The rest of this chapter covers high levels security concepts, narrowing down to JAAS at the end.

1.2 The Java 2 Security Architecture

The main functionality of the Java 2 security architecture is protecting resources. “Resources” can be anything, but are usually some chunk of data: employee records, databases, or more abstract pieces of data such as class files. The classes in the `java.security` package do that work directly by defining the process for testing access permission, and associating permissions with code based on the source from which it was loaded. There are additional subsections and utilities also included in the architecture:

- The Java Cryptography Architecture (JCA) defines interfaces and classes for encrypting and decrypting information.
- The Java Secure Socket Extension (JSSE) uses the JCA classes to make secure network connections.
- JAAS, the topic of this book, which performs authentication and authorization.





1.3 JAAS

This book is concerned primarily with the lower right box in the diagram above: JAAS. JAAS is a mix of classes specific to JAAS, and classes “borrowed” from other parts of the Java security framework. The primary goal of JAAS is to manage the granting of permissions and performing security checks for those permissions. As such, JAAS is not concerned with other aspects of the Java security framework, such as encryption, digital signatures, or secure connections.

There are several concepts and components that make up JAAS, but all of them revolve around one part: permissions.

1.3.1 Permissions

A *permission* is a named right to perform some action on a target. For example, one permission might be “all classes in the package `com.myapp` can open sockets to the Internet address `www.myapp.com`.”



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

Who is Granted a Permission?

As the example implies, some “entity” is granted a permission. In Java, this entity is usually either a user or a “code base.” Users are a familiar concept, and generally map to a person or process executing code in your application. Users as entities are discussed at length below in the sections on Subjects and Principals. On the other hand, a code base is a bit more vexing in it’s meaning. A code base is a group of code, usually delineated by a JAR or the URL from which the code was physically loaded. For example, all the classes downloaded as an applet from a remote server could be put into a single code base. Then, because permissions can be applied to code bases, your application could disallow code from that applet from accessing the local file system. You would want to deny access to the local file system to, for example, prevent applets from installing spy- or ad-ware on your machine, or installing viruses. This ability to “sandbox” code, keeping remote, un-trusted code from performing malicious actions, was one of the prime selling points of Java early on.

In Java, sub-classes of the abstract `java.security.Permission` class are used to represent all permissions. There are several types of permissions shipped in the SDK, such as `java.io.FilePermission` for file access, or `java.net.SocketPermission` for network access. The special permission `java.security.AllPermission` serves as a stand-in for any permission. Additionally, you can create any number of custom permissions by extending the class yourself.

A `Permission` is composed of three things, only the first two of which are required:

1. The type of `Permission`, implicit in its class-type.
2. The `Permission`’s name, generally the target(s) the `Permission` controls.
3. Actions that may be performed on the target.

Conceptually, the type and the name of the `Permission` specify what is being accessed. The actions are generally a comma-delimited set of allowed actions. A `FilePermission` named “pristinebadger.doc” with actions “read, write” would let the possessor read from and write to the file “pristinebadger.doc”. The table below illustrates a 4 more examples:

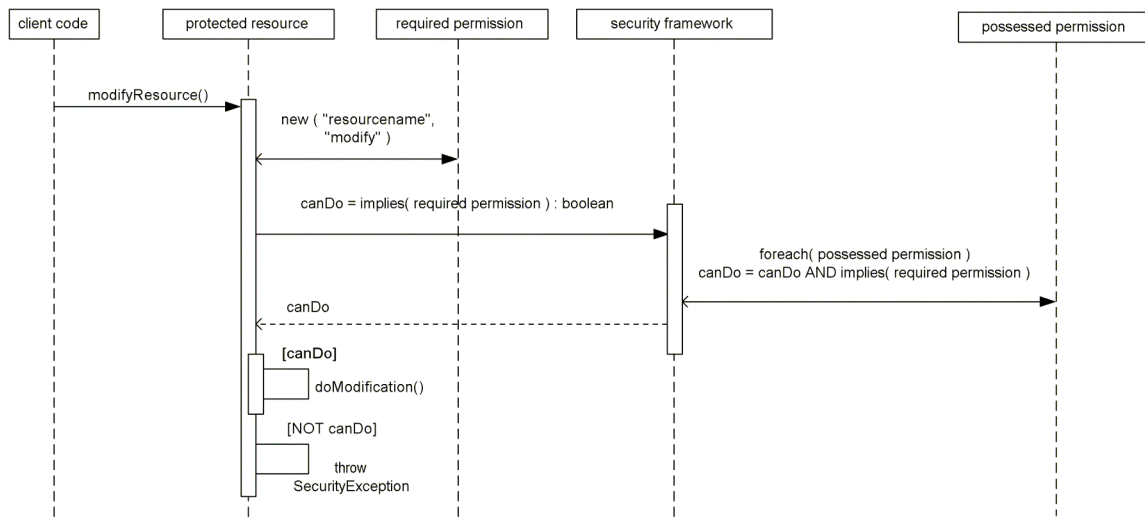
Permission Type	Target	Action
<code>ProfilePermission</code>	All	read
<code>ProfilePermission</code>	All	write
<code>DocumentPermission</code>	“Project Avocado”	read
<code>DocumentPermission</code>	All “programming group” documents	write

The actual “allowing” takes place in the boolean `implies(Permission)` method in `Permission`. When resource access occurs, the resource constructs an instance of the corresponding `Permission` class and passed it to the security framework. The framework then tests if the current security context² has been granted the right(s) described by the `Permission` instance. The security framework searches for a permission with the correct type and name. If it finds one, it calls `implies` on it, passing in the newly constructed

² The “security context” includes the rights granted to the currently executing code and, if available, the currently logged in user.



Permission instance. If `true` is returned, access is allowed. The fact that the instance of `Permission` performs the check means that custom permissions can be created. The diagram below illustrates this process:



The general flow of permission checking

Permissions are positive

Permissions express what *can* be done, not what *cannot* be done. First, this design decision helps avoid any conflict in permissions. For example, if a user has a permission that gives it access to the C: drive, and another permission that expressly forbids access to the C: drive, which one applies? Some method of resolving problems such as this would need to be part of the security model. Instead of devising a conflict resolution process, Java defines only positive permissions. When permissions can only express the positive any chance of conflict is removed.

Alternatively, the creators of JAAS could have chosen to express only negative permissions. A permission would be granted unless it was expressly forbidden. This opens up a security hole in that you must know about everything you want to restrict ahead of time. For example, if a new file enters the system, users will by default have access to that file until the system expressly forbids access. In the mean time, a malicious user could access the file. With a positive permission model, JAAS avoids this need to express everything that is forbidden and the problems that can arise in such a system.

1.3.2 The SecurityManager and AccessController

The pre-JAAS security model employed the concept of a security manager, a singleton of `java.lang.SecurityManager` through which all the types of permission were expressed and checked. This class is still used in current versions of Java as the preferred entry point for security checks, but it has traces of the pre-JAAS security model. A quick inspection of the



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

`java.lang.SecurityManager` reveals methods with names like `checkDelete` and `checkExec`. These correspond to each task that needs explicit permission to be performed, such as deleting a file or creating an exec process. Each of these methods will throw a `SecurityException` if the permission in question has not been granted. If the permission has been granted, the method simply returns. The code guarding the resource would call the specific check method either granting access, or throwing a `SecurityException` if permission is not granted³.

For example, the code in `java.io.File` delete may have included something like:

```
public void delete( String filename ) {  
    ...  
    SecurityManager.getInstance().checkDelete(filename)  
    // no SecurityException thrown, so delete file  
    ...  
}
```

If permission has been granted to delete `filename`, the call to `checkDelete` will pass, and the file will be deleted by additional code. If permission has not been granted, `checkDelete` will throw a `SecurityException`, meaning that the code calling `delete` will need to catch that exception and respond accordingly (perhaps by displaying an error message to the user).

A developer wanting to protect a new resource would have to extend the `SecurityManager`, create new `checkXXX` methods, and tell the VM to use this new `SecurityManager` using the runtime property, `java.security.manager`. This unwieldy option spurred the creation of the `Permission` class in Java 2. Also, a new method, `checkPermission`, could now be used to check any `Permission`, enabling the creation of new permissions without having to create a new `SecurityManager`.

To back this new model, a new permission checking service class was introduced `java.security.AccessController`. For backwards compatibility, the existing `SecurityManager` was preserved and, as mentioned above, is still the preferred entry point for permission checking. Like `SecurityManager`, `AccessController` also has a `checkPermission` method. In fact, the default implementation of `SecurityManager` delegates its calls to `AccessController`. This class knows how to access the current thread's security-related context information, which includes all the permissions allowed for the code-stack that is currently executing. A call to `AccessController.checkPermission()` thus extracts those permission and checks the supplied permission against each piece of code executing on the stack.

Enabling the SecurityManager

By default, the `SecurityManager`, and thus security as a whole, is disabled when you run Java. The `SecurityManager` can be enabled with the VM argument `-Djava.security.manager`, or by setting the `SecurityManager` to use programmatically

³ This model of security checking is why so many methods in the JDK throw `SecurityException`.



at runtime.

1.3.3. Policies

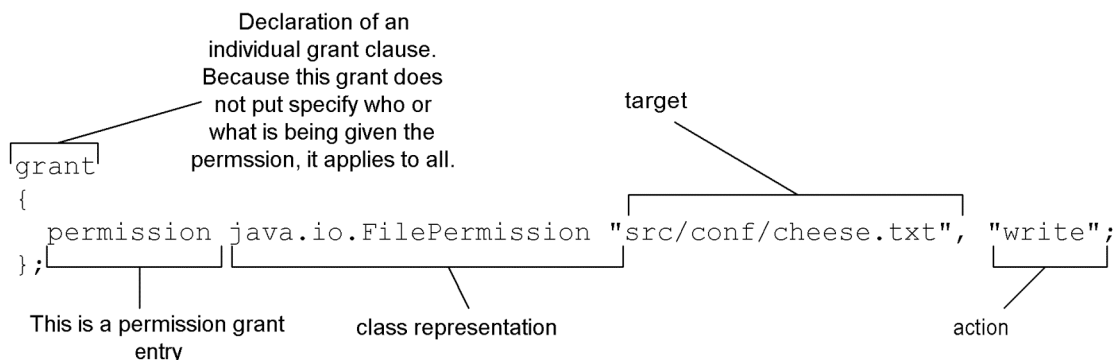
Backing the `SecurityManager` and `AccessController` is a policy that expresses which permission are granted to a given security context. Policies map the entities that might attempt to access the resources to their `Permissions`. For example, the code found on the `D:` drive may be disallowed from modifying any files on the `C:` drive. As we'll see later in this chapter, this can also be applied to users in addition to code. The main purpose of separating out this concept is that the policy is the logical place for deployment-time configuration. Keeping this responsibility in one place makes managing your security system easier and less error prone.

In the Java security framework, a policy is represented by sub-classes of the abstract class `java.security.Policy`. This class is always a singleton: there is only ever one instance in the VM. Because of this, a `Policy` can be thought of as a service for resolving `Permission` checks. The `Policy` in use can be specified as a VM argument, or can be changed at runtime by calling the static method `Policy.setPolicy(policy)`. Because the `Policy` being used can be swapped out as needed, custom `Policies` can be used in your Java applications. As we'll see, coupled with the generic nature of JAAS classes, this allows you to use the Java security framework as a rich user based permission system.

The default policy implementation

By default, there is no security manager in effect. This effectively allows all permissions for all resources. If the argument `-Dsecurity.manager=classname` is passed to the VM at runtime, an instance of `classname` is constructed and used. If the property is supplied with no value, a default implementation is chosen.

If the default security manager is specified, the default policy implementation receives its permissions via a flat-file that can be specified at run-time via the VM argument `-Dsecurity.policy.file=policyfilename`. If the path is not specified, the file `$JAVA_HOME/lib/security/java.policy` is used.



Sample grant clause from the configuration file for Java's default policy implementation



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

This policy gives all code the ability to write to the file named “src/conf/cheese.txt”. Without this permission an attempt to modify the file will result in a `SecurityException`.

The Default Policy File

This default policy file is located at `JAVA_HOME/lib/security/java.policy`. This location can be changed with the VM argument `java.security.policy`, which points to the file path of the policy file to use. Also, modifying the `policy.url` properties in the file `JAVA_HOME/lib/security/java.security` will change the location that default policy file is read from.

As we’ll see in later chapters, you can also change the policy being used at runtime, even specifying a class to use to resolve permission checks instead of a flat file.

1.4. JAAS

When Java code is executing, it needs to figure out which `Permissions` to apply to the current thread of execution. That is, when JAAS is doing a `Permission` check, it must answer the question “which `Permissions` are currently granted?” As mentioned above, JAAS associates `Permissions` with a “`Subject`.”⁴ In most cases, a `Subject` can be thought of as a “user,” but put more broadly, a `Subject` is any entity that Java code is being executed on behalf of. That is, a `Subject` need not always be a person who’s, for example, logged into a system with their username and password.

For example, when a person logs into a JAAS-enabled online banking system, the system creates a `Subject` that represents that user. JAAS resolves which `Permissions` the `Subject` has been granted, and associates those `Permissions` with the `Subject`. A “non-human” `Subject` could be another program that is accessing the application. For example, when the nightly batch-processing agent authenticates, or logs into, the system, a `Subject` that represents the agent is created, and the appropriate `Permissions` are associated with that `Subject`.

1.4.1. Authenticating Subjects

So, the first step in JAAS taking effect is logging a user into the system or “authenticating” the users. When a system authenticates a user, it establishes that the user is who they claim to be. As a real world example, when a bank asks one of its clients for their driver’s license and compares the picture to them, they’re authenticating the client’s identity. Similarly, when a user logs in to their bank’s online banking application, they’re prompted to provide a username and password. Because the user knows both of these items, called “credentials,” the online banking system trusts [believes?] the claim that they’re Joe User, and allows them to

⁴ Actually, in the Java security model, `Permissions` can be associated directly with code as well, further specified by the code’s origin. For example, you could specify that all classes loaded from the JAR `somecode.jar` be given a specific set of `Permissions`. This will be covered in later chapters. For our purposes here, we’ll skip this detail.



see their account balances, transfer money, and pay bills.

1.4.2. Principals: Multiple Identities

JAAS doesn't directly associate a user's identities with a Subject. Instead, each Subject holds onto any number of Principals. In the simplest sense, a Principal is an identity. Thus, a Subject can be thought of as a container for all of Subject's identities, similar to how your wallet contains all of your id cards: driver's license, social security, insurance card, or pet store club card. For example, a Principal could be:

- The user "jsmith," which is John Smith's login for the server.
- Employee number #4592 which is John Smith's employee number.
- John's Social Security number which is used by the HR department.

Each of these identity Principals is associated with John Smith and, thus, once John authenticates with the JAAS-enabled system, each Principal is associated to his Subject.

Breaking out a Subject's identities into Principals creates clear lines of responsibility between the two: Principals tell JAAS who a user is, while Subjects aggregate the user's multiple identities. Also, this scheme allows for easier integration with non-JAAS authentication systems, such as single-signon services. For example, when a Subject is authenticated with a single-signon service, all of the different users are converted to Principals, and bundled into the Subject. In this scheme, the Subject is an umbrella for all the different identities the user, as represented by a Subject, can take on.

1.4.3. Subjects and Principals: Roles

In addition to Principals representing different identities of a Subject, they can also represent different roles the Subject is authorized to perform. For example, John Smith can perform the following of roles:

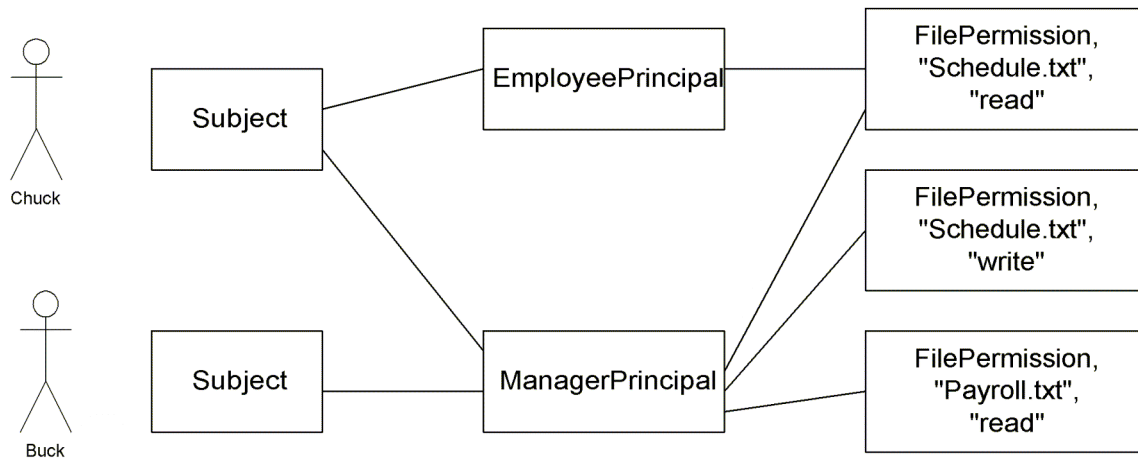
- Administer users, including approving new user requests, deleting users, or resetting their passwords.
- Set system-wide configuration properties, like which mail server to use, or the name of the company.

Each of these roles can be encapsulated in JAAS as a Principal. Two roles could be created for the above items: User Administrator, System Administrator. As with identities, rather than directly associate each role's abilities, or permissions, with a Subject, JAAS associates the role's abilities with Principals. John Smith's Subject, then, would have Principals that represent both of these Administrator roles.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>



A mapping of Employee and Manager roles to permissions. Note that Chuck is an employee AND a manager.

Separating roles into their own Principals has the same dividing-up-responsibility advantages as separating identities. The primary benefit, however, is with managing and maintaining the user's permissions. The management of who can do what in these systems can become extremely cumbersome and time consuming if a user's abilities are directly associated with each Subject instead of a role-based Principal, which is associated with n-users.

For example, suppose we have a system with 5 User Administrator. If we followed a model where permissions were associated directly with Subjects instead of Principals, each Subject would be assigned the Permission to approve user requests, delete users, and reset their password. During the lifetime of the system, the permissions one of these User Administrators have will change. For example, suppose we decide that resetting a password for a user should only be done by the user: if a User Administrator reset their password, someone other than the user would know the password and might do evil things with that knowledge, or let it slip into the wrong hands. So, we have to modify the Permission of each of those 5 User Administrators, and remove the reset password permission. Similarly, we must visit each of the 5 User Administrators if we add a Permission.

While this may not seem too onerous a process for 5 users, doing it for more than 5 users can start to be tedious. For example, suppose we had a user base of 10,000 users, 4,032 of which you want to add the new permission "can delete documents." If Permissions were directly associated with Subjects, you'd have to visit all 4,032 of those users! Instead, if the 4,032 users are all in the "Document Editor" role, as expressed by a Principal in JAAS, you need only edit that one role.

1.4.3. Credentials

In addition to Principals, Subjects also have Credentials. The most common types of credential are a username and password pair. When you log in to your email account, for example, you're prompted to enter your username and password.

Credentials can take many forms other than a username and password:



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

- A single credential, like your password for a voice mail system.
- A physical credential, like a garage door opener to open your garage.
- A digital certificate.
- A mix of physical and keyed-in credentials, like your ATM card and your PIN number.
- Biometric credentials, like your thumbprint or retinal scan.

Put simply, anything you use to prove your identity is a credential.

In JAAS, Subjects hold onto two types of credentials: public and private credentials. A username is, in most cases, a public credential: anyone can see your username. A password is, in most cases, a private credential: only the user should know their password. JAAS doesn't specify an interface, or type, for credentials, as any `Object` can be a credential. Thus, determining the semantics of a credential—what that credential “means”—are left up to the code that uses the credential.

1.4.4. Principals and the policy

Once a Subject has been authenticated, having all of its Principals associated with it, JAAS uses the `Policy` service to resolve which Permissions the Principals are granted. A `Policy` is simply a singleton that extends the abstract class `java.security.Policy`. JAAS uses the `Policy`'s `implies()` and `getPermissions()` methods to resolve which Permissions a Subject has been granted.

The default `Policy` implementation is driven by a flat-file, allowing for declaratively configuring the Permissions pre-runtime. Because this implementation is file-driven, however, it's effectively static: once your application starts, you can't cleanly change the file's contents, thus changing Permissions.

To provide a more dynamic, runtime Permission configuration, you'll need to provide your own `Policy` implementation. You can swap out the `Policy` in effect through a VM argument, or at runtime. As with many other JAAS functions, the currently executing code must have permission to swap out the `Policy`.

1.4.5. Access Control: Checking for Permissions

Once a Subject is logged in, and has its Principals associated with it, JAAS can begin to enforce access control. Access control is simply the process of verifying that the Subject has been granted any rights required to executing the code. JAAS implements access control by wrapping Permission checks around blocks of code. The block of code could be an entire method, or a single line of code. Indeed, for best performance and the finest grain of security control, wrapping the smallest chunk of code possible is the best option.

Because of its nature, then, access control is often done in JAAS in a “try/catch” fashion: attempting to execute the protected code, and then dealing with security exceptions that are thrown due to failed Permission checks. Permission checking can also be done in a more query-related fashion: before executing a block of code, you can first see if the Subject has the appropriate permissions.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

Once the `Subject` is authenticated, and the appropriate `Permissions` are loaded, JAAS-enabled code executes securely. Before the code executes, JAAS verifies that the `Subject` has the appropriate `Permissions`, throwing a security exception if the `Subject` does not. This is what is meant by JAAS's claim of being "code centric": the actual code being protected by `Permissions` often does the checking itself.

In most cases, the JAAS model of access control requires the code that is performing the sensitive action to do the permission checking itself. For example, instead of blocking access to a calling `java.io.File`'s `delete()` method, the method itself does the security check. This is usually the safest and quickest approach, as finding all the places that call `delete()` is much more difficult than simply putting access checks in the method itself. In this code-centric approach, the code must include `Permission` checks, and, thus, be knowable of which `Permissions` to check.

In some situations, such a model may be too cumbersome to maintain. Each time a new type of `Permission` is added that's relevant to deleting a file, you must modify the `delete()` method to check for this `Permission`. Strategies that use Dynamic Proxies, declarative meta-information (such as `XDocket` or `Annotations` in J2SE 1.5) or Aspect Oriented Programming, can be used to more easily solve problems like this. Indeed, those and similar strategies can often be used as a cleaner alternative to embedding access control code yourself.

1.4.6. Pluggability

As the above more code-level talk implies, JAAS is a highly pluggable system. What this means is that you can provide functionality to JAAS that wasn't originally shipped with the SDK. It also means that you can change the way in which parts of JAAS work. For example, if the default flat-file based `Policy` doesn't fit your requirements, you can implement and use your own, as later sections in this book will detail.

Unfortunately, as with other high-level frameworks, being pluggable also means there's a bit of code that you'll have to write to customize JAAS to your needs. The good news, however, is that you *can* customize it to your needs.

For example, out of the box, JAAS has no idea how to identity users against your HR system. But, you can write a small amount of code that will do just this. Better, instead of having to conform the HR system to how JAAS works, you can customize JAAS to conform to how the HR system works. This is what "pluggable" means in relation to JAAS: you can add in new functionality that wasn't previously there, and you're not restricted to the original intent, design, and functionality of the framework.

The authentication system is pluggable primarily through providing custom `LoginModule` implementations, while the authorization system is pluggable by providing both custom `java.security.Permission` implementations, and `java.security.Policy` implementations. Chapters X and XX cover creating these custom implementations in great detail.

1.5 Looking ahead

The first part of the book covers the different components of JAAS, going into the above



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

major components and “supporting classes” is much more detail. Included in this part will be an example of creating a database-backed `Policy` and custom `Permissions`.

While the first part has many examples of using these JAAS classes, the second part will provide more in-depth examples of common uses of JAAS, such as logging users in, managing user groups, and creating data-centric `Permissions`.

Finally, the appendixes will go over changes in JAAS in J2SE 5.0, standard J2SE `Permissions`, and a concise reference for configuring JAAS.

Summary

Our first encounter with security in Java began with the need to provide a secure web application for accessing employee information. With that problem at hand, we started exploring the broad topic of Java security, and narrowed down to the Java Authentication and Authorization Service, or JAAS. We introduced JAAS's primary concepts and classes: permissions, policies, and the service layers needed to enforce the granting of permissions. While we dipped our toe into the code-waters of JAAS, our discussion remained fairly high-level so that we could establish the domain needed to dive into the code.

