



Using jLock's Java Authentication and Authorization Service (JAAS) for Application-Level Security

Updated January 2006
www.2ab.com

Introduction

Access management is a simple concept. Every business has information that needs to be protected from unauthorized disclosure. To protect information, companies define policies that govern who can access specific classes of business and/or personal information. For example, if a manager seeks to access the salary of a subordinate, they should have authorization to do so, however, they should not be authorized to access the same information about a chief executive. That is, there is a policy that specifically governs the release of an employee's salary. Or is there? The answer is: "Probably not." What exists is a written policy related to disclosure of proprietary business information (and perhaps even a separate policy related to disclosure of employee personal information). Because human beings are skilled at generalizations, we expect someone in authority to be able to classify the request for salary information and make a decision.

Access Management software has a simple goal. It allows the human who previously acted as a guardian of sensitive information to be removed from the process without loss of access control. This sounds simple, but most businesses are struggling with the implementation of access management as they integrate and extend their Java applications. This is because machines cannot classify information or make access decisions unless they are explicitly programmed with algorithms to accomplish this. When you take the responsibility for access decisions away from human beings, it becomes necessary to insert software guards into your applications.

The Java Authentication and Authorization Service (JAAS) defines the standard programming interface for building these software guards in a Java environment. Prior to JAAS, security mechanisms in Java were strictly code-based. That is, you granted permissions based on the code that was running – there was no way to grant permissions based on the identity (or credentials) of the user of the application. For this reason, any user-based access control mechanisms had to be coded directly into the business application (typically requiring new database tables and/or directory infrastructure). When access policy or audit requirements changed, application software had to be modified, tested and redeployed. Additionally, when access policy needs to be examined, or applications audited for conformance, a code review was required.

Access management solutions, such as commercial implementations of JAAS, provide scalable alternatives to the costly embedding of access control mechanisms and access policy. They allow application software guards to leverage services that enable access policy to be modified, tested and deployed dynamically without application code changes. This enables your developers to concentrate on providing business software. Access management solutions efficiently enable high performance access controls in distributed environments while allowing centralized management of access policy. Any commercial access management solution includes application programming interfaces (APIs) and policy management tools. JAAS defines these APIs for the Java environment.

This paper describes a systematic approach to using JAAS to manage the complexity associated with software access management. It explains the JAAS service-oriented architecture (SOA), which maintains a clean separation of concerns between application functionality and access management. It discusses the types of JAAS features that a scalable JAAS implementation should support. We offer coding examples using jLock, 2AB's commercial JAAS implementation. We hope this paper will help you determine if JAAS should be part of your application-level security infrastructure.



Why use the Java Authentication and Authorization Service?

Application security must address any security-related requirements not provided by the runtime security infrastructure. In the area of access management, any requirement to restrict a) the usage of application features or b) access to business and personal information is part of “application security.” There are many excellent overviews of JAAS on Sun’s JavaSoft Web site. For that reason, we will assume the reader has some familiarity with the JAAS model, and we will focus on how the JAAS model can be used to provide the fine-grain access control requirements of “application security.”

JAAS consists of two parts: Authentication and Authorization.

- Authentication answers the question: “How do I know that you are who you say you are?” The goal of authentication is to securely determine who is executing Java code, regardless of whether or not the code is part of a standalone Java application, a servlet, an applet or an Enterprise Java Bean.
- Authorization answers the question: “Now that I know who you are, how do I know if you are allowed to access the information or application feature that you are requesting?” The goal of authorization is to protect business and personal information and sensitive application features from being used by people who legitimately have access to the application and some subset of its functionality.

There are a number of papers from various sources that focus on how to build an implementation of JAAS. These papers explain in detail the concepts, design center and classes that are used in a JAAS implementation. The classes include LoginContext, LoginModule, CallbackHandler, Subject, Principal, Permission and AccessController. We will discuss those concepts in this paper as they become visible to the Java programmer but will not discuss details of their implementation. This paper will focus on how a Java developer uses a commercial implementation of JAAS. We will also explore how user identity and access policies are managed using jLock’s Administrative Tools. All the code samples, build and run scripts shown in this paper are available for download.

JAAS Authentication Example

JAAS Authentication is based on the Pluggable Authentication Module (PAM) architecture. Leveraging an architecture that supports ‘plug-ins’ for authentication ensures that Java applications can be independent of the underlying authentication mechanism. This has the advantage that new or revised authentication mechanisms can be plugged in without modifying the application code. That is, management of User IDs and Passwords (or other methods of authentication) are removed from the application’s concern. For this example, we will leverage the dialog-based User ID and Password authenticator that is supplied with the jLock product.

The first thing you need to do is specify the JAAS implementation that you are using. This is done with a login configuration file. This may be done on the command line when you invoke your application.

```
java -Djava.security.auth.login.config=jaas_config.txt ...
```

The **jaas_config.txt** file supplied with the example is shown below. It specifies an application name (JaasDemo) and the jLock plug-in class for the LoginModule. There are a number of additional configurable options available, but for this example, we will use the defaults.

```
/** Login Configuration for the jLock JAAS Demo Applications */
JaasDemo {
    com.twoab.jaas.LoginModuleUP required instance="standarddemo";
};
```

JAAS Login Configuration File (jaas_config.txt)



This is the Java code for a class that prints “Hello iLock World” if user authentication succeeds. The two JAAS methods your application needs to invoke to use a JAAS authenticator are shown in **bold** font.

```
import com.twoab.jaas.*;
import java.util.*;
import javax.security.auth.login.*;
import javax.security.auth.*;

public class StandardJAAS {

    public StandardJAAS(String [] args) {
        LoginContext lc = null;

        /** Create a LoginContext object. */
        try {
            lc = new LoginContext("JaasDemo", new DialogCallbackHandlerUP());
        }
        catch (LoginException le) {
            System.out.println("Cannot create LoginContext. " + le.getMessage());
            System.exit(1);
        }
        catch (SecurityException se) {
            System.out.println("Cannot create LoginContext. " + se.getMessage());
            System.exit(-1);
        }

        try {
            lc.login();
        }
        catch (LoginException le) {
            System.out.println("\nAuthentication failed:");
            System.out.println(" " + le.getMessage());
            System.exit(1);
        }
        System.out.println("\nHello iLock World!\n");
        .....
    }
}
```

JAAS Authentication Code Sample (from StandardJAAS.java)

That is all the code and configuration you need! When you run the example (run.bat), at the point where the **lc.login()** is called, the following dialog will appear.





Type in a User ID and Password as shown above and click **OK**. jLock will authenticate the user.

Assuming you typed a valid User ID and Password, the example program results will, as you might expect, look like the following:

```
C:\iLock\demos\jaas\basic>run -d
Hello jLock World!
```

Authentication Succeeded

Of course, if you should fail to provide a valid User ID and/or Password, you will see this:

```
C:\iLock\demos\jaas\basic>run -d
Authentication failed:
Login Failure: all modules ignored
```

Authentication Failed

[How the JAAS Authentication Example Works](#)

The JaasDemo has obviously written no Java code to manage Users or Passwords nor to do the work required to authenticate the user (in this case verify the password). That is the great thing about the JAAS architecture; just “plug in” jLock, and it securely manages all that for you!

When the LoginContext object was constructed:

```
lc = new LoginContext("JaasDemo", new DialogCallbackHandlerUP());
```

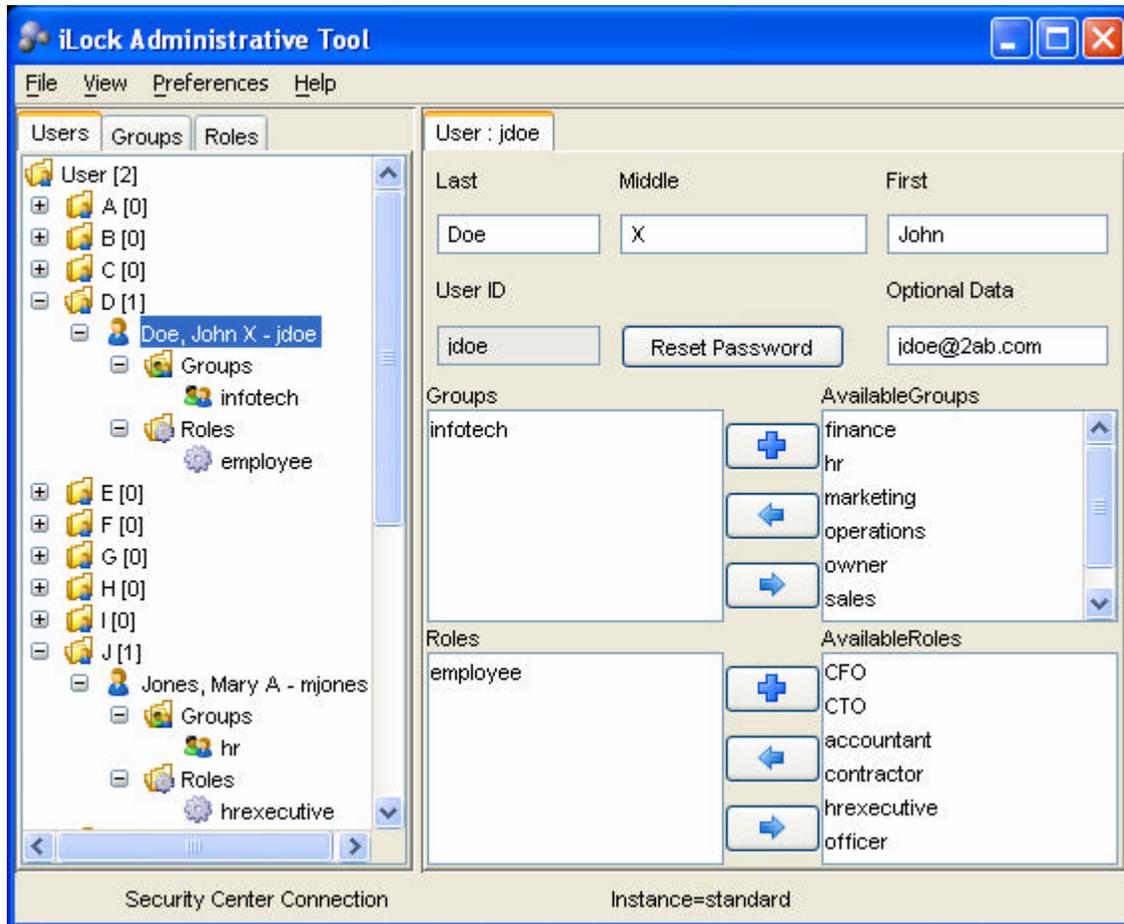
- The application name (JaasDemo) was specified.
- The associated JaasDemo entry in the config.txt file determined the LoginModule class that the Java Runtime would use (jLock’s).
- This LoginModule class understands how to use jLock to authenticate users.
- The login configuration file was specified by the java.security.auth.login.config property.
- The *DialogCallbackHandlerUP* class was provided to the LoginContext. This is a handler supplied with jLock’s JAAS implementation. It understands how to obtain the information necessary to authenticate the user (in this case via a Dialog box). *NOTE: JAAS vendors typically supply multiple callback handlers, but you can write your own custom handler if you wish.*

When you requested authentication: **lc.login()**, the DialogCallbackHandlerUP obtained the User ID and Password and supplied it to the LoginModule, which contacted jLock with a request for authentication. jLock ensures that the password is never available in clear text. jLock securely stores and transmits password information.



Managing Identity Information with jLock

Before we explore JAAS Authorization, it is important to understand how Identity is managed in jLock.



The Identity Manager View in the Security Center Administrative Tool

The Identity Manager in jLock allows you to create users and passwords. An administrator using the **Reset Password** button on the Identity Manager can reset the passwords. Password format requirements can be set using the **Preferences** menu. Additionally, users can be assigned additional security attributes such as Groups and/or Roles by selecting an available group/role and moving it to the assigned groups and roles.

AccessIds, Groups and Roles can be used to create access policies that govern what a user (using jLock) is allowed to do. We will explore that in detail when we cover the JAAS Authorization example.



JAAS Authorization Model

The JAAS Authorization model extends the code-centric, Java security architecture that uses a security policy to specify what access is granted to executing code (such as access to files, sockets or specific operations). The extension allows security access policy to be defined based on the credentials associated with the user of the code. Just as a commercial JAAS Authentication may be plugged in, the JAAS Authorization model also allows vendors to offer commercial solutions that offer scalability, management and enhanced support for sophisticated access policy.

To understand the JAAS Authorization model, you must first understand a little more about what happens when you authenticate using JAAS. When the user (*jdoe* in the example above) was authenticated, a **Subject** object was created. A Subject represents the entity that was authenticated – that is, the entity that has been able to prove their identity. A Java **Principal** is a “security attribute” or “credential” that can be associated with one or more Subjects. During the authentication process, the jLock authenticator acquired the credentials of the Subject and associated them with the subject by creating the appropriate Principal objects. A user (i.e. Subject) may always be able to prove their identity, but their credentials (i.e. Principals) may change over time. For this reason, security access policy is defined in terms of the security attributes (or in Java terminology Principals) that are associated with the Subject at the time identity was authenticated. jLock supports three types of Principals: 1) AccessIdPrincipal, 2) RolePrincipal and 3) GroupPrincipal. These map to the UserIds, Groups and Roles shown in the Identity Manager. These are the fundamental building blocks of access policy. In the section above, you can see that the user, John X. Doe, has the following jLock security attributes.

- AccessId: *jdoe*
- Roles: *employee*
- Group: *infotech*

If you add the following code to the example, you can see that the LoginContext allows navigation to a Subject that manages a set of Principals.

```
.....
    java.util.Set prin_set = lc.getSubject().getPrincipals();
    java.util.Iterator it = prin_set.iterator();
    while (it.hasNext() == true) {
        java.lang.Object obj = it.next();
        if (obj instanceof AccessIdPrincipal) {
            System.out.println("AccessId - " +
                ((AccessIdPrincipal)obj).getName();
            )
        }
        else if (obj instanceof GroupPrincipal) {
            System.out.println("Group - " +
                ((GroupPrincipal)obj).getName();
            )
        }
        else if (obj instanceof RolePrincipal) {
            System.out.println("Role - " +
                ((RolePrincipal)obj).getName();
            )
        }
        else {
            System.out.println("Unknown principal type");
        }
    }
}
```

Code to display the names of the Principals associated with the authenticated Subject



Running with this code, you will see the output below following authentication:

```
Command Prompt
C:\ilock\demos\jaas\basic>run -d
Hello jLock World!
JAAS Principals:
AccessId - AUTHENTICATED
AccessId - PUBLIC
AccessId - uid:jdoo
Group - infotech
Role - employee
```

JAAS Authorization Example

At this point, you are ready to use JAAS authorization. Sun's reference implementation requires that grant statements that define access policy be placed in policy files for each user and that the application use the Java Security Manager (in the same way that grant statements and policy files are used for code-centric security). Since it obviously is not practical (or secure) to manage user-based access policy in local, plain-text files for a large user community, JAAS providers such as 2AB offer solutions that allow identity and access policy to be managed separately from the application. Sun's reference implementation also requires that any code that requires user-based access control be placed in a separate class and executed only via Subject.doAs (or doAsPrivileged) methods. That sets the scope of the user-based software guard to the class where the sensitive code is located. jLock does not preclude the use of the do.As operations for access management but does support the insertion of software guards that use the JAAS Principal-based authorization model without the requirement to segment the code into separate classes.

After the user has authenticated, it is still necessary to determine if the user has permission to access Salary information.

```
System.out.println("Attempting SpecialAction");

try {
    ResourcePermission p = new ResourcePermission("Salary");
    AccessController.checkPermission(p);
    System.out.println("Access to Salary Info is granted");
}
catch (com.twoab.jaas.AccessControlException ace) {
    System.out.println("Sorry - Access to SalaryInfo is denied");
    ace.printStackTrace();
}
```

Code to protect access to the "Salary" code

It is that simple to add authorization to your application. Notice that while we can certainly run this application with the Java Security Manager installed (adding a few permissions to the java.policy file), this demo does not require the Security Manager to leverage the jLock JAAS features. You simply insert your sensitive code in a try block and check for the appropriate permission before running it. Remember, you are not checking whether the code has access to the resource, you are only checking whether or not the application should provide the resource to the user.

When you run the demo and authenticate using *jdoo* as the User ID, you will see the following dialog showing that John Doe is not allowed to access Salary information:



```
Command Prompt
C:\ilock\demos\jaas\basic>run -d

Hello jLock World!

JAAS Principals:
  AccessId - AUTHENTICATED
  AccessId - PUBLIC
  AccessId - uid:jdoh
  Group - infotech
  Role - employee

Attempting SpecialAction
Sorry - Access to SalaryInfo is denied
```

However, if you run the demo and authenticate using *mjones* as the User ID, you will see that Mary Jones is allowed to access Salary information:

```
Command Prompt
Hello jLock World!

JAAS Principals:
  AccessId - AUTHENTICATED
  AccessId - PUBLIC
  AccessId - uid:mjones
  Role - hrexecutive
  Group - hr

Attempting SpecialAction
Access to Salary Info is granted
```

How the JAAS Authorization Example Worked

The JaasDemo has obviously written no Java code to assign permissions to a user, manage access policy nor to determine if the user has the required permissions. The access policy is completely hidden from the application. Most people would guess that *mjones* has been granted the Salary permission and *jdoh* has not. That assumption is wrong. Neither AccessID *mjones* nor AccessID *jdoh* is part of the access policy. It is the RolePrincipal *2ab/hrexecutive* or the GroupPrincipal *2ab/hr* that is required to access (view) Salary.

When the jLock AccessController is asked to check a Permission:

```
ResourcePermission p = new ResourcePermission("Salary");  
AccessController.checkPermission(p);
```

- The AccessController uses the current Subject (e.g. User) object to extract the Principals (e.g. Security Attributes)
- It then asks jLock to evaluate the policy that protects the Permission (e.g. Resource) to determine if someone with these Principals (e.g. Security Attributes) should be granted access.
- The AccessController returns if access should be allowed and throws an exception if access should be denied.

NOTE: All of the tools for creating and managing the access policy are provided by jLock. The jLock architecture supports dynamic policy updates that take effect immediately, so applications do not need to be restarted when access policies change. jLock also supports remote policy administration.

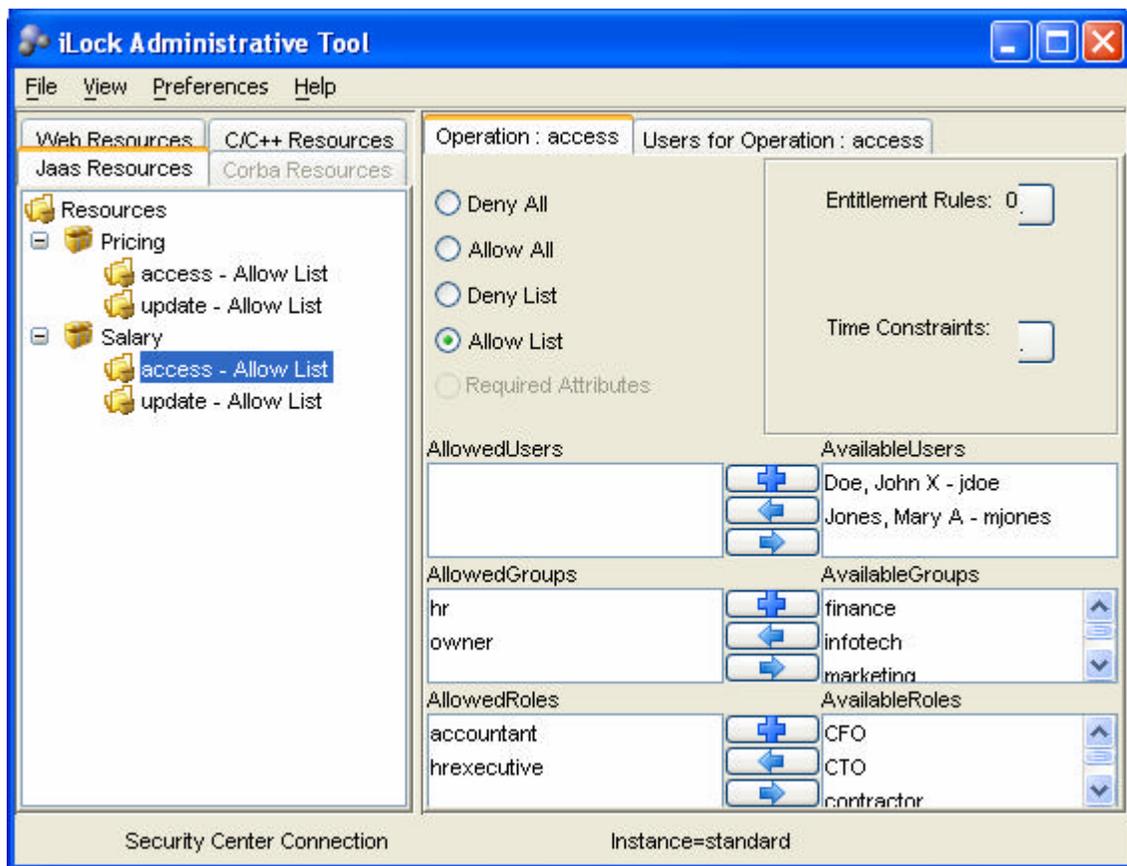


Managing Access Policy with jLock

Now let's look at how access policy is managed in jLock.

You have already seen how security attributes within the jLock Security Center are associated with a user using the administrative tools. These attributes are represented as JAAS Principals. Because jLock supports many different access policy models - including, but not limited to access control lists (ACLs) and role-based access control (RBAC), there is some terminology mapping that needs to be explained. A Permission (as defined by JAAS) is represented as a Resource to the jLock Security Center. Therefore, the jLock Security Center Resource editor is used to create a JAAS Permission.

The "Salary" Resource Permission looks like the following:



Defining Access Policy for the Salary (JAAS Permission) Resource

This access policy states that in order to have permission to "access" Salary, the user is required to be in the Group *hr*, or have the Role *hrexecutive* or the Role *owner*. By associating this policy with the "Salary" Resource (jLock permission), any person with one of these attributes is allowed to access Salary. If the Role *hr* is removed from *mjones*, Mary Jones will no longer have access. If this role is added to *jdoe*, John Doe will have access. We will leave this as an exercise for the interested reader. Note that there is another operation named "update" associated with Salary. The `AccessController.checkpermission()` also allows the user to specify an operation or action (access is the default). In this way, different rules can be created for different actions on application resources.

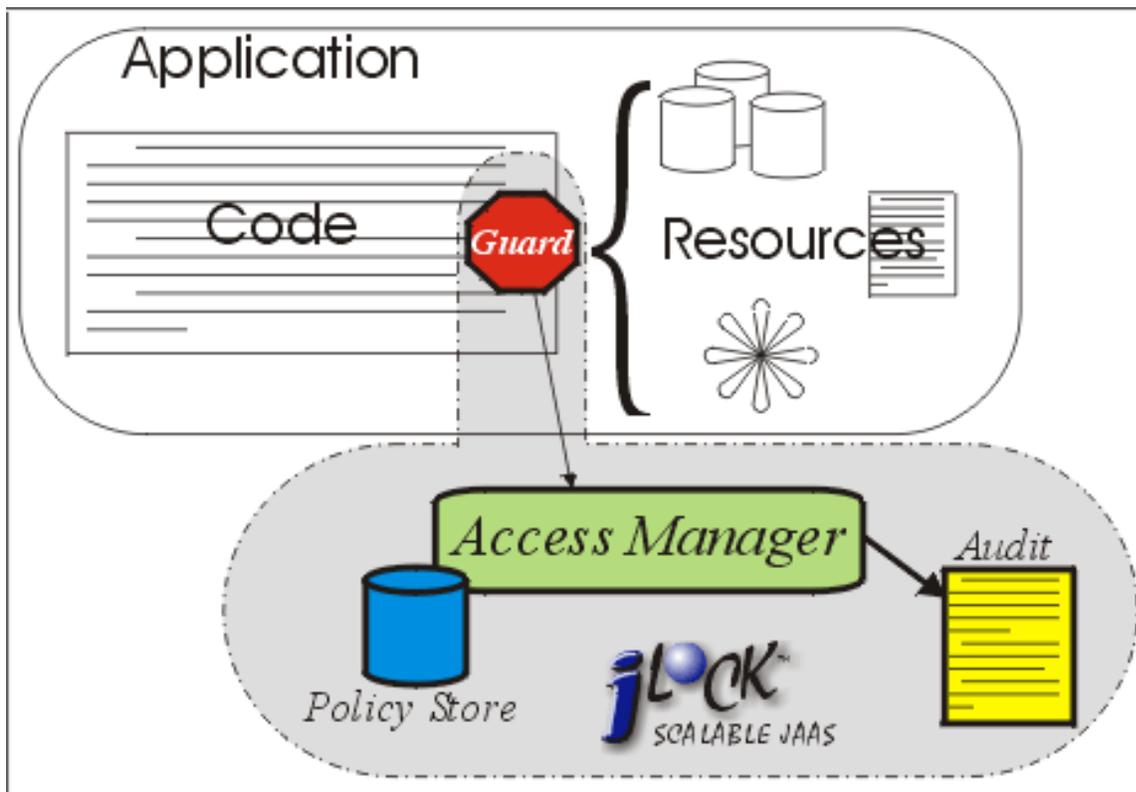
Access Policy can be very simple or quite sophisticated. Once it has been determined that applications require access management features, they typically begin with very simple access control policy based on



AccessIDs or Roles as we have outlined in this paper. There are many applications, however, that require sophisticated access policy. jLock supports access policy based on Identity, Role, Group, Context, Entitlements/Clearances and Relationships. It also supports complex rules that combine diverse policy types. To gain a better understanding of Access Management and policy models, you may review our Access Management White Paper (downloadable from www.2ab.com/pdf/AccessManagement.pdf)

Summary

jLock supports a service-oriented architectural approach to dealing with application-level security. A JAAS implementation such as jLock provides APIs that enable you to authenticate and easily integrate access control checks within your business applications. JAAS supports a pluggable architecture that allows you to select your JAAS vendor based



upon your requirements for authentication and access policy support.

Utilizing JAAS, your business developers simply insert AccessController calls (or Guards) at the points in the software where sensitive resources are exposed. This AccessController consults with the JAAS who evaluates the policy and advises the Guard on allowing access. The JAAS architecture enables many different policy models to be leveraged by a Java business application.

JAAS supports a service-oriented architecture for authentication and authorization

We hope that this paper has helped you understand how jLock can be leveraged for application-level security. If you would like to evaluate jLock in your environment, please contact sales@2ab.com