Global Information Assurance Certification Paper

Application Security within Java 2, Standard Edition (J2SE)

JAAS, JCE, and JSSE

By Damon Kaberna
GSEC Assignment v.1.4B
March 16, 2004

Table of Contents

Abstract

Several “tools” have been implemented with Java 2 Standard Edition (J2SE) that can be used to secure applications. With J2SE you can authenticate, authorize, encrypt data, and encrypt the transmission of data all within code. These tools include the Java Authentication and Authorization Service (JAAS), the Java Cryptography Extension (JCE), and the Java Secure Socket Extension (JSSE).

This paper will examine what the security professional needs to know to make informed decisions about how security is implemented within Java applications using J2SE tools. With the proper understanding, security can be implemented within a Java application, making the application very safe and secure.

Application Security Basics

When designing an application the focus of the security professional is often on operating system (OS) security, network security, or even physical security. It is easy to understand the need to secure a server, the physical lines that connect servers, or the room in which the server is housed. Unfortunately application security is often overlooked.

What is application security? For the purposes of this paper, application security is security that is implemented or dramatically influenced by the person or persons who build applications using Java. Most typical security elements including authentication, authorization and encryption can be implemented within application code. It is my hope to shed light on Java application security topics which are often considered beyond the responsibility of the security professional.

There are security elements which may be implemented at the application layer or at the system layer. Security which occurs at the system layer is often termed “declarative” security because the security specifics must be declared to the system in which a resource resides. File security implemented by an operating system is a good example of declarative security. If a security professional locks down a file, then the operating system protects the file by making sure that identities attempting access to the file have the appropriate permissions.

Security which occurs at the application layer is often termed “programmatic” because decisions made by the application developer within the application programming logic affect how the application is secured.

An example of programmatic security is a web page which makes decisions on which content to show a user based on the user’s attributes. For example, an employee logs on to an intranet site which is restricted to management employees. If the employee is a manager then the site is displayed. If the employee is not a manager then the employee is directed to some other site. The

web site checks to see if the employee is a manger by looking in a data store (perhaps an LDAP directory).

Java 2 Standard Edition (J2SE) and Java 2 Enterprise Edition (J2EE)

J2SE and J2EE are not the name thing. J2SE is often described as “core Java”. J2SE provides the core functionality required for a modern programming language to be useful. J2EE sits “on top” of J2SE providing functionality such as transactional support, isolation, and security to J2EE applications. Enterprise Java Beans (EJBs) are notable J2EE constructs which can be called remotely and provide this functionality.

Basic J2SE Security

Java is a mobile language. What I mean by this is that Java code is designed to run regardless of source or target platform. When you download an applet (a piece of Java code which runs in a browser) it does not matter where the code was downloaded from, it does not matter that you are downloading to a particular system, and it does not matter what your intentions are in downloading the code. If the applet contains malicious code then you could be in trouble if your security settings are not properly enabled.

“The basic principle underpinning the Java 2 platform security architecture can be summarized as follows: A system-level *security policy* defines *access permissions* (per the needs of the application under consideration) for executing code grouped into *protection domains*. The security policy is used for access control checks, which are performed by the JVM at runtime.”[1]

Java grants access to a resource through a relatively simple mechanism. Classes are used to represent the security architecture. Let’s take a look at some of these classes:

The Permission class (java.security.Permission) contains a named permission and a list of actions which are allowed for the resource in question. The FilePermission and SocketPermission classes are examples of classes which implement the Permission class to protect a particular resource. You can also create your own implementation of this class (i.e. WidgetPermission) to custom tailor permissions. The “implies” method of the Permission class is used to determine if access to the resource is implied given an instance of a permission class.

The PermissionCollection (java.security.PermissionCollection) is a collection of Permissions of the same kind. The “implies” method of the PermissionCollection

[1] Belapurkar, p. 2

class is used to aggregate the results of the “implies” methods of all of the Permission objects within the collection.

The Permissions class (java.security.Permissions) is a collection of PermissionCollection instances (that may not contain Permission objects of the same type) which represent the total access of a resource. The “implies” method of the Permissions class is used to determine the results of the appropriate PermissionCollection related to the resource in question.

The CodeSource class (java.security.CodeSource) encapsulates the URL of a piece of running code and the certificates used to sign the code into a class so that a particular piece of code can be identified.

A ProtectionDomain (java.security.ProtectionDomain) is created using an instance of a CodeSource class and a PermissionCollection. It represents a mapping between code source and permissions.

The Policy class (java.security.Policy) is used to map the systems policy (in the form of permissions) into individual ProtectionDomains. The GetPermissions method of the Policy class allows callers to retrieve the permissions for a particular CodeSource. Policy is typically represented by a file, but the provider for the policy can be changed by changing the value of the “policy.providor” property in the Java.security properties file. The java.security.policy environment variable is used by the Java Virtual Machine (JVM) to determine the location of the physical policy store.

The AccessController (java.security.AccessController) is a class which is responsible for access checking at run-time. The AccessController must be specified in the “java.security.manager” environment variable so that when the JVM spins up the AccessController can be created. The CheckPermission method throws an exception if the requested permission is not valid. AccessController is used by both the system and by the application programmer to check permissions. The SecurityManager class delegates calls to AccessController to enable the system to check access.

Now, let’s talk about how all these classes relate to one another. When any class is loaded, the system uses a class called the SecureClassLoader (java.security.SecureClassLoader) to load the class. First the class is loaded from the specified URL and then the digital signature of the class is checked to make sure that the class is correct and intact. A CodeSource object is derived using known information. Using the GetPermissions method of the Policy instance (the Policy instance is maintained by the system) and the CodeSource, the PermissionCollection for the resource is acquired. The ProtectionDomain is then created using the CodeSource and PermissionCollection. Using an instance of the AccessController class permissions can be checked within an application.

The system uses the SecurityManager class which delegates access checks to the AccessController to check permissions.

In summary, if you want to do "code centric" access checks in Java: 1) Make sure that the SecurityManager is known to the JVM. 2) The Policy Provider must be in place and known to the system. 3) A policy file (or what ever is needed by the policy provider) must be in place and known to the system. 4) The developer can make access checks in code if necessary. 5) Custom Permission and PermissionCollection classes may need to be written to handle resources and permissions that you might want to secure.

There is a lot more to the story than what I have presented here. Check out the references below for details on privileged checks and threads of execution. Also see Appendix C for a code example of how a logon might be performed.

Beyond code centric security, J2SE security has been augmented with JAAS, JCE, and JSSE. These services add authentication, authorization, encryption, and transport layer encryption (SSL) services to Java. JAAS, JSSE, and JCE are integrated into the 1.4 version of the J2SDK. In version 1.3 of the SDK, JAAS, JSSE, and JCE were optional elements. Let's talk about these security tools.

Java Authentication and Authorization Service (JAAS)

"Authentication is the process by which an entity, also called a principle, verifies that another entity is who or what it claims to be. A principle can be a user, some executable code, or a computer. "[2] There are many was to perform authentication, however, it is most often performed by applying credentials such as an ID and password.

Authorization is the process of access control. Once you prove who you are by authenticating you need to get to the resources that are required to do our work. You should not be able to get to resources that are not required to do your work. Authorization allows the system to determine what resources you should have access to and what resources you should not have access to.

The Java Authentication and Authorization Service (JAAS) is a toolset which enables J2SE applications to authenticate and authorize the caller to access specific resources. JAAS builds on J2SE (code centric) security adding authentication and implementing "user-centric" authorization. Custom authentication and authorization solutions can be created with JAAS.

You may be asking yourself, why would I want to create my own authentication or authorization scheme for Java? There are lots of reasons to do this – some good reasons, but many bad reasons. Do the research and make an informed

[2] Howard, p. 46-47

decision. If you have determined that you need a custom scheme, proceed carefully. Getting the correct elements in place can be the difference between success and failure.

Let's talk about authentication first.

JAAS Authentication

Authentication with JAAS is relatively simple. Here are the basic elements:

To login a user in code, the developer must create an instance of the LoginContext (javax.security.auth.login.LoginContext). Using the LoginContext, the Login method is called to login the user. Sounds simple, right? Well there is a little bit more that has to be done. When the LoginContext is created, a runtime login configuration file entry friendly name and a callback handler must be supplied.

The callback handler is an instance of a class (javax.security.auth.callbackhandler) which handles user interaction during authentication. User interaction might include a prompt for a user ID and password.

The runtime login configuration file is located by the system by looking in the Java.security file at the login.config.url.x property. The x is a number between 1 and n and is used to specify more than one configuration file. The login.config.url.x property actually contains a fully qualified file name which points to a runtime login configuration file. Optionally, the runtime login configuration file can be set through the java.security.auth.login.config property in code or as a parameter when the application is executed.

Within the runtime login configuration file are a series of runtime login configuration file entry names (i.e. JndiLoginModule, UnixLoginModule…) which associate a friendly name (supplied when the LoginContext is created) to a class which actually does the authentication (the LoginContext interface in the javax.security.auth.spi). Check out Appendix A for an example of the login configuration file.

In order to log in, the "login" method from the LoginContext is used. Login makes an association between the LoginModule (which manages how authentication works) and the callback handler (which gets input from the user). If authentication succeeds the commit method of the LoginModule is called and the result is an instance of a Subject class which represents the user. Within the Subject are instances of Principle classes which further identify the context of the user based on how the user authenticated.

In summary, what does this all mean for the security professional? If you want to implement authentication in Java, the developer has to: 1) create an instance of a callback handler to handle user input, 2) create or identify a class to do authentication, 3) associate the class which does authentication to a friendly name in the runtime login configuration file, 4) make sure that the runtime login configuration file is known to the system and 5) create an instance of the LoginContext and call its login method in code.

JAAS Authorization

JAAS Authorization is an extension of the process started during authentication. Authorization is not possible without some form of authentication. The primary mechanisms of JAAS authorization are 1) an authenticated Principle or a CodeBase, 2) an implementation of the "java.security.PrivilegedAction" class, 3) a file which describes the authorization policy to be followed and 4) some code to run that contains some actions needing authorization.

In order to perform JAAS authorization, you need to know three things. You need to know 1) who or what is authorizing, 2) the resource they are trying to use and 3) what they are actually trying to do with the resource (read, write, execute, delete…).

There are a couple of ways to identify who is authorizing. Actually, the question is often not "who" but "what" is authorizing. A Principle, a CodeBase or both can be used to identify who or what is attempting authorization.

We have already talked about Principles. A Principle is a representation of an authenticated user created during authentication. A CodeBase is just what it sounds like: a chunk of code. A CodeBase can be a Java class or a whole application.

The principle and/or CodeBase are specified in the authorization policy file along with their related privileges. The second grant within the Sample Authorization Policy File in Appendix B is a good example of how a CodeBase is specified to the system. The third grant shows how a principle is specified.

When known events occur such as creating a file, reading a property, or listening at a socket, a security check is triggered. If the CodeBase or Principle has the authority to perform the operation then the operation is allowed. If not then the operation is not allowed.

Let's talk about the specific mechanics of authorization. The first step to authorization is creating an implementation of the PrivilegedAction class and overriding the run method. The run method must contain all of the custom code which needs to be authorized to run.

The next step is to create an authorization policy file which contains the “rules” which determine the privileges needed to access particular resources. Appendix B contains an example of a policy file.

When a developer needs to run some code which needs to be authorized, the doAsPrivileged (or doAs) method of the Subject class must be called. The Subject is acquired through the instance of the LoginContext which was created during authentication.

Calling doAsPrivileged requires an instance of the subject, an instance of the overridden PrivilegedAction (or PrivilegedActionException) class, and an instance of an AccessControlContext. The AccessControlContext is not required. If it is included it will be used as the base context to apply the current authorization context to.

So, if you want to authorize using JAAS you have to 1) authenticate using JAAS, 2) create a policy file for authorization rules, 3) override the PrivilegedAction class, and 4) call the doAsPrivileged method of the Subject acquired during authentication or from the LoginContext.

Summary

Alright, now that we are all JAAS’d, let’s sum up. JAAS is a toolset within J2SE which offers custom authentication and authorization services. There are many authentication mechanisms which have already been written to perform authentication using JAAS to various trusted sources.

Authentication is achieved through the use of a call back handler which is used to gather credentials from the user, a LoginModule which performs the actual authentication, a Subject which contains information representing the authenticated user in the form of one or more Principles, and a login configuration file which tells JAAS what class to use to authenticate.

Policy drives authorization. Creating a good policy file can be tricky and takes time. Authorization is achieved through an authenticated Subject, an overridden PrivilegedAction class, and an authorization policy file.

Using JAAS must be a well considered choice. JAAS does not replace system authentication and authorization and it also potentially introduces administrative complexity to an application. Do your research and make sure that JAAS and J2SE security meet your objectives before you implement.

Java Cryptography Extension (JCE)

Maintaining the confidentiality and integrity of data is one of the most significant issues that security professionals face today. The internet enables us to

communicate across broad public vistas, but, because of its public nature, it also makes us more vulnerable to manipulation and attack. Even company owned intranets are more open to attacks than ever before. Company owned intranets are often semi-public intranets where consultants, vendors, and company employees mix.

Cryptography is one tool used to protect the confidentiality and integrity of data both at rest and as it crosses the wire. Cryptographic solutions that cross platforms have been few and far between in the past. Java, because it provides tools for cryptography and because its nature is portable, offers some relief from this problem.

JCE is an extension of the Java Cryptography Architecture (JCA). According to Sun Corporation in its Java Cryptography Extension (JCE) Reference Guide for the Java 2 SDK, Standard Edition, v 1.4, "the Java Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects"[3].

We will not look at all of the implementation details of JCE as we did with JAAS. Instead we take a quick look at what JCE can do.

Encryption/Decryption

In JCE, the Cipher class provides Encryption and Decryption services. AES encryption with a 256 bit key is the strongest encryption algorithm supported by JCE. DES3, DES, Blowfish and many other algorithms are also supported.

Encryption and decryption are fairly easy with JCE. To get an instance of the Cipher class, call the static "getInstance" method of the Cipher class with the encryption algorithm and, optionally, a transform and padding. The transform specifies a method to use to transform the data prior to encryption. Padding specifies how data is packed when it is returned after an encryption or decryption.

The instance of the Cipher class must then be initialized with its function. These functions include encrypt, decrypt, wrap, and unwrap. Encryption and decryption are actually performed by the update and doFinal methods. Wrap and unwrap – the process of wrapping a key into a secure form and un-wrapping the key into a usable form - are performed by the wrap and unwrap methods.

Key Generation

[3] Sun Corporation

In order to perform encryption and decryption using a symmetric algorithm, you have to have a key. Key generation is the process of generating a key for use with cryptographic functions.

JCE has the facility to generate keys in the form of the KeyGenerator class. To get an instance of the KeyGenerator class, call the static member function getInstance with the algorithm name that you want to generate a key for. Initialize the key with a seed or key size using the init method, then use generateKey to actually acquire a key from the generator. Easy!

Key Agreement

Key agreement is a process that allows two parties to come up with the same key. The process gives the users a level of surety that the party that they are coming to agreement with is who they say they are.

During key agreement, sometimes a key is exchanged, sometimes information is exchanged which allows both parties to generate the same key. Key agreement is an attempt to solve the problem of sharing keys safely. If one party is sending encrypted data to another, the receiver had better be able to decrypt the encrypted data and the sender should be sure that the receiver is the only one who can decrypt the data.

The details of key agreement can be very confusing. JCE uses the KeyAgreement class to perform key agreement functions. The SUN JCE Reference Guide contains a good example of how the Diffie-Hellman key exchange algorithm might be coded if you are interested. However, key agreement should not be taken lightly. Consider your options carefully before you code and use such algorithms. The confidentiality and integrity of your data is at stake.

Message Authentication Code

Ensuring the integrity and availability of data is a problem. Data stored at rest can be altered and replaced. Data moving across the wire can be intercepted and replaced. Message authentication codes are one way that you can attempt to ensure the integrity of data.

With JCE you can create hashed messages using message authentication codes (mac) and the Mac class. The hash is based on a secret key that 2 parties share. Key agreement and data transfer are the responsibility of the application or application user.

There are uses for message authentication codes within applications. However, most modern encryption mechanisms (such as SSL) include some sort of digital signature or hash to ensure message integrity. Security professionals should use

MACs where appropriate, but should not rely on this mechanism for an "enterprise" solution to data integrity.

In summary, JCE is a useful technology for encryption, decryption, and key generation within an application. Key agreement is available and can be used to exchange keys among parties. However, key agreement is complex and should be approached with caution. Message authentication codes can be used to validate the integrity of data, but should not be used extensively because other encryption technologies do the same thing.

Java Secure Socket Extension (JSSE)

JSSE is used primarily as a wrapper for the Secure Sockets Layer (SSL) protocol. SSL is used to protect the integrity, confidentiality, and availability of data as it crosses the wire through the use of encryption technologies.

Application layer SSL is very complicated. To use SSL, you have to know about sockets, key stores, trust stores, encryption keys, and encryption algorithms. JSSE does not eliminate the need to be well informed, but it does hide a lot of the implementation details of using SSL.

SSL

Setting up an SSL session can be tricky. Simple examples of how to enable an SSL session are available on Sun's web site so I will not describe the details. Instead, let's take a look at the major components of setting up a generic outbound SSL session.

The first thing you need to do to set up an SSL connection within a Java application is to make the system aware of the properties you want your SSL session to have. You have to set up the providers that you will need to make SSL work and make the system aware that you are using SSL and JSSE.

The next thing you need to do is set up a key manager. A key manager is used to manage credentials and authentication when an SSL connection is created. To create a key manager, you need a key store. The key store contains certificates and keys used for authentication. The structure of the key store depends on the platform that you are running on.

The third step is to create a trust manager. A trust manager uses the information in a trust store to decide who to trust. A trust store is a key store which contains certificates and keys related to trusted servers. Trusted servers are those servers that you want to connect to without having to actively make a decision as to whether it is a good idea or not.

Key stores and trust stores are really just files which contain certificates and keys. As often as not, you have to create and manage these stores for yourself. However, different systems support different forms of key and trust stores so check with you security administrators before building your own store.

Lastly, an SSL context is created using the key managers and trust managers that were just created. This context is used by Java to initiate SSL traffic to a chosen destination.

HTTPS

The method just described is a generic method for creating an SSL context which allows Java to use SSL under the covers.

There are other ways to initiate SSL using JSSE. The java.net.url class can be used to establish an HTTPS connection (or whatever other kind of protocol supported) to a resource (typically a file) on a server. URL is easy to use and should be considered if trying to establish HTTPS connections in code.

Conclusions - Using J2SE Security

Even though I have spent the last several weeks researching J2SE security and find it fascinating, I find myself asking the question: Under what circumstances would I use JAAS, JCE, and JSSE? This is a tough question that every security professional, application designer, and developer should ask themselves before diving into J2SE security.

JAAS has some very interesting features. You can authenticate users. You can authorize one application to access another through the use of a code base. You can authorize an application or user to access system resources. You can also sign code to make sure that the integrity of the code has not been violated.

Authentication with JAAS seems worthwhile. Applications should be able to make decisions based on who accesses them. Code signing might also be useful in a shared or exposed environment.

I have serious doubts about using authorization with JAAS however. Implementing authorization is not trivial. It takes work. J2SE code-centric security and JAAS do give you the ability to authorize system resources, but I cannot think of a situation where authorizing a piece of code or user to access a socket or property would, in a real-world sense, be worthwhile. If someone were to supply an authorization layer over JAAS to authorize business resources then JAAS authorization would be worthwhile, however, I have not seen such an implementation.

JCA is useful for encryption, decryption and key generation. However, I do not believe that the typical application would need to support key agreement or message authentication codes. Key agreement seems very complex and would be difficult to code correctly. Also, there seems to be few occasions when key agreement is really necessary from code. Message authentication codes can be useful to ensure that data is not tampered with, but again I cannot think of a situation where other mechanisms cannot be used more easily.

JSSE is one of the best ways to establish SSL connections. If you want to get to a web page from code via HTTPS it is relatively easy with JSSE if you can figure out how to configure your application.

If you produce or need to secure Java applications, you will use J2SE security at some point. Authentication, authorization, data encryption and transport layer encryption are all supported in one way or another. Use J2SE security to augment your other security mechanisms and your security model will be complete.

References

(1) SUN Corp. “Java Secure Socket Extension (JSSE) Reference Guide for the Java 2 SDK, Standard Edition, v 1.4.2” URL: http://Java.sun.com/j2se/1.4.2/docs/guide/security/jsse/JSSERefGuide.html

(2) SUN Corp. “Java Authentication and Authorization Service (JAAS) Reference Guide for the Java 2 SDK, Standard Edition, v 1.4”
URL: http://Java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASRefGuide.html

(3) SUN Corp. “Java Cryptography Extension (JCE) Reference Guide for the Java 2 SDK, Standard Edition, v 1.4”
URL:http://Java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html

(4) Howard, Michael and LeBlanc, David. Writing Secure Code. Microsoft Press, 2002

(5) Belapurkar, Abhijit. “Java Authorization Internals”
URL: http://www-106.ibm.com/developerworks/java/library/j-javaauth/ (04 May 2004)

(6) SUN Corp. “Java Class URL”
URL: http://java.sun.com/j2se/1.4.2/docs/api/java/net/URL.html

(7) SUN Corp. “Default Policy Implementation and Policy File Syntax”
URL:http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html

(8) SUN Corp. “Policy Tool – Policy File Creation and Management Tool”
URL:http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/policytool.html

(9) SUN Corp. “JAAS Authentication Tutorial”
URL:http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/tutorials/GeneralAcnOnly.html

(10) SUN Corp. “JAAS Authorization Tutorial”
URL:http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/tutorials/GeneralAcnAndAzn.html

(11) SUN Corp. “Class ConfigFile”
URL:http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/spec/com/sun/security/auth/login/ConfigFile.html

(12) SUN Corp. “JAAS Login Configuration File”
URL:http://java.sun.com/j2se/1.4.2/docs/guide/security/jgss/tutorials/LoginConfigFile.html

Appendix A – Sample Logon Configuration File (JAAS)

```
MLogin {
  mcompany.security.access.MLoginModule required debug=true;
};
**********************
```

“MLogin” is the friendly name that JAAS uses to associate an authentication attempt to the correct LoginModule class.

“mcompany.security.access.MLoginModule” is the fully qualified Java class actually containing custom code used to login.

Appendix B – Sample Policy File (JAAS)

```
/* The first grant gives basic permissions to all principles and CodeBases */
grant {
  permission java.lang.RuntimePermission "stopThread";
  permission java.net.SocketPermission "localhost:1024-", "listen";
  permission java.util.PropertyPermission "java.version", "read";
  permission java.util.PropertyPermission "java.vendor", "read";
  permission java.util.PropertyPermission "java.vendor.url", "read";
  permission java.util.PropertyPermission "java.class.version", "read";
  permission java.util.PropertyPermission "os.name", "read";
  permission java.util.PropertyPermission "os.version", "read";
  permission java.util.PropertyPermission "os.arch", "read";
  permission java.util.PropertyPermission "file.separator", "read";
  permission java.util.PropertyPermission "path.separator", "read";
  permission java.util.PropertyPermission "line.separator", "read";
  permission java.util.PropertyPermission "java.specification.version", "read";
  permission java.util.PropertyPermission "java.specification.vendor", "read";
  permission java.util.PropertyPermission "java.specification.name", "read";
  permission java.util.PropertyPermission "java.vm.specification.version", "read";
  permission java.util.PropertyPermission "java.vm.specification.vendor", "read";
  permission java.util.PropertyPermission "java.vm.specification.name", "read";
  permission java.util.PropertyPermission "java.vm.version", "read";
  permission java.util.PropertyPermission "java.vm.vendor", "read";
  permission java.util.PropertyPermission "java.vm.name", "read";
};

/* The second grant gives the code in the codebase "file:mJAAS.jar" the */
/* authority to do all the things it need to do to authenticate and authorize */
grant codebase "file:mJAAS.jar" {
  permission javax.security.auth.AuthPermission "createLoginContext.MLogin";
  permission javax.security.auth.AuthPermission "doAsPrivileged";
  permission javax.security.auth.AuthPermission "modifyPrincipals";
};

/* The third grant give the principle “testUser” the authority to mess with the */
/* blee.txt file. Notice that the principle class used to contain the principle */
/* information in the Subject must be supplied */
grant Principal mcompany.security.access.MPrinciple "testUser"  {
  permission java.io.FilePermission "blee.txt", "read, write, delete, execute";
};
```

Appendix C – Code Example – Logging In with JAAS

```
// This code shows how a developer might make some function calls to login

private LoginContext lc = null;

public void Login() throws Exception
{
        try
        {
                // Create a login context using the login method friendly
                // name (MLogin) from the login.config file
                // and an instance of the CallbackHandler class
                // (MCallbackHandler) that was created to handle login.
                lc = new LoginContext("SFLogin", new MCallbackHandler());

                // perform a login
                lc.login();
        }
        catch (LoginException le)
        {
                System.err.println("Cannot create LoginContext. " +
                        le.getMessage());
                System.exit(-1);
        }
        catch (SecurityException se)
        {
                System.err.println("Cannot create LoginContext. " +
                        se.getMessage());
                 System.exit(-1);
        }
}
```

Appendix D – Code Example – Authorizing with JAAS

```
static LoginContext lc = null;
static Subject lSubject = null;

// Run assumes that the login process has already been completed

public Object Run()
{
        // MAction implements the PrivilegedAction class
        MAction lMAction = new MAction();

        // Get the Subject which represents the logged on
        // user.
        lSubject = lc.getSubject();

        // JAAS actually runs the run method of the lMAction class
        // for you. It is able to do this because lMAction implements
        // PrivilegedAction.
        return Subject.doAsPrivileged(lSubject, lMAction, null);
}
```

Appendix E – Running Code (JAAS)

```
// The following “java” command is an example of how you might run the code
// within the mJAAS.jar file. The assumption is that mJAAS.jar contains all the
// files it needs to authenticate and authorize.

java -Djava.security.manager -Djava.security.policy=="java.policy" -
Djava.security.auth.login.config=="login.config" -jar mJAAS.jar
```