developerWorks®

Java security, Part 2: Authentication and authorization

Brad Rubin (BradRubin@BradRubin.com)
Principal
Brad Rubin & Associates Inc.

19 July 2002

The Java platform, both its base language features and library extensions, provides an excellent base for writing secure applications. In this tutorial, Part 2 of 2, Brad Rubin introduces the basic concepts of authentication and authorization and provides an architectural overview of JAAS. Through the use of a sample application he'll guide your understanding of JAAS from theory to practice. By the end of the tutorial you will have a good foundation for working with JAAS on your own.

About this tutorial

What is this tutorial about?

There is perhaps no software engineering topic of more timely importance than application security. Attacks are costly, whether the attack comes from inside or out, and some attacks can expose a software company to liability for damages. As computer (and especially Internet) technologies evolve, security attacks are becoming more sophisticated and frequent. Staying on top of the most up-to-date techniques and tools is one key to application security; the other is a solid foundation in proven technologies such as data encryption, authentication, and authorization.

In this two-part tutorial, we've been learning about the security features of the Java platform. Part 1 served as a beginner's introduction to Java cryptography. Here, in Part 2, we'll expand the discussion to encompass access control, which is managed in the Java platform by the Java Authentication and Authorization Service (JAAS).

Should I take this tutorial?

This is an intermediate-level tutorial. It is assumed that you know how to read and write basic Java applications and applets. If you're already a Java programmer, and you've been curious about authentication and authorization technologies and the Java library that supports them, then this tutorial is for you.

We'll start with an introduction to the basic concepts of authentication and authorization, as well as an architectural overview of JAAS. Next, we'll use a JAAS sample application to take your understanding of JAAS from theory to practice, both by breaking down the components of the application and by viewing the final execution result. As part of this exercise, we'll study a variety of JAAS configuration options that will help further cement the concepts you've learned.

JAAS is a complex technology, rich in function and features. We'll take it in slowly, in bite-sized chunks, and it is recommended that you go over each new concept more than once. By the end of the tutorial you will have a good foundation for working with JAAS on your own.

You need not have taken Part 1 of this tutorial to understand Part 2.

Tools, code samples, and installation requirements

JAAS started out as an extension to the Java 2 platform, Standard Edition. Recently, however, it has been added to version 1.4. To complete this tutorial you will need the following:

- JDK 1.4, Standard Edition
- The tutorial source code and classes, JavaSecurity2-source.jar, so that you can follow the examples as we go along.
- A browser that supports the Java 1.4 plug-in.

You can use JDK 1.3.x, but you must install the JCE and JSSE yourself.

Conceptual overview

Authentication and authorization

Authentication is a process by which a human user or computing device verifies his, her, or its identity. *Authorization* is a process by which a sensitive piece of software allows access and operations that depend on the identity of the requesting user. These two concepts go hand-in-hand. Without authorization, there is little need to know the user's identity. Without authentication, it is impossible to distinguish between trusted and untrusted users, which makes it impossible to safely authorize access to many parts of the system.

It isn't always necessary to identify or authenticate individual entities; in some cases you can authenticate by group, granting certain authorization to all entities within a given group. In other cases, individual authentication is essential to the security of the system.

Another interesting aspect of authentication and authorization is that a single entity can have several roles in a system. For example, a human user could be both an employee of a company, which means he would need access to corporate e-mail, and an accountant within the company, which means he would need access to the company accounting system.

Elements of authentication

Authentication is based on one or more of the following elements:

- **What you know.** This category includes information that an individual knows that is not generally known by others. Examples include PINs, passwords, and personal information such as a mother's maiden name.

- **What you have.** This category includes physical items that enable individual access to resources. Examples include ATM cards, Secure ID tokens, and credit cards.
- **Who you are.** This category includes biometrics such as fingerprints, retina profiles, and facial photographs.

Often, it isn't sufficient to use only one category for authorization. For example, an ATM card is generally used in combination with a PIN. Even if the physical card is lost, both the user and the system are presumably safe, since a thief would have to know the PIN to access any resources.

Elements of authorization

There are two fundamental ways of controlling access to sensitive code:

- **Declarative** authorization can be performed by a system administrator, who configures the system's access (that is, declares who can access which applications in the system). With declarative authorization, user access privileges can be added, changed, or revoked without affecting the underlying application code.
- **Programmatic** authorization uses Java application code to make authorization decisions. Programmatic authorization is necessary when authorization decisions require more complex logic and decisions, which are beyond the capabilities of declarative authorization. Since programmatic authorization is built into the application code, making programmatic authorization changes requires that some part of the application code be rewritten.

You'll learn about both declarative and programmatic authorization techniques in this tutorial.

Protecting users and code from each other

The Java platform allows fine-grained access control to computing resources (such as disk files and network connections) based on the degree of trust the user has in the code. Most of the base security features of the Java platform are designed to protect users from potentially malicious code. For example, digitally signed code, backed by a third-party certificate, ensures the identity of the code source. Based on his knowledge of the code source, a user can choose to grant or deny execution rights to this code. Similarly, a user can grant or deny access based on the download URL of a given code source.

Access control on Java-based systems is implemented via a policy file, which contains statements like the one below:

```
grant signedBy "Brad", codeBase "http://www.bradrubin.com" {

       permission java.io.FilePermission "/tmp/abc", "read";
};
```

This statement allows code signed by "Brad" and loaded from http://www.bradrubin.com to read the */tmp/abc* directory.

Other Java platform features, such as lack of pointers, further protect users from potentially malicious code. Working together, JAAS's authentication and authorization services provide a complementary function: they protect sensitive Java application code from potentially malicious users.

Pluggable authentication modules

JAAS implements a Java version of the Pluggable Authentication Modules (PAM) framework. Sun Microsystems created PAM for its Solaris operating system; with JAAS, PAM is now available in a platform-independent form.

The main purpose of PAM is to allow application developers to write to a standard authentication interface at development time, leaving the decision of which authentication technologies will be used (and how they will be used) up to the system administrator. Authentication technologies are implemented in login modules that can be deployed after an application has been written, and are specified in a text file called a *login configuration file* (named login.config in this tutorial). The login.config file can specify not only which modules to call, but the conditions for overall authentication success.

PAM allows new authentication techniques or technologies to be more easily added to existing applications. Likewise, an authentication policy can be changed by updating the login.config file, rather than rewriting the entire application.

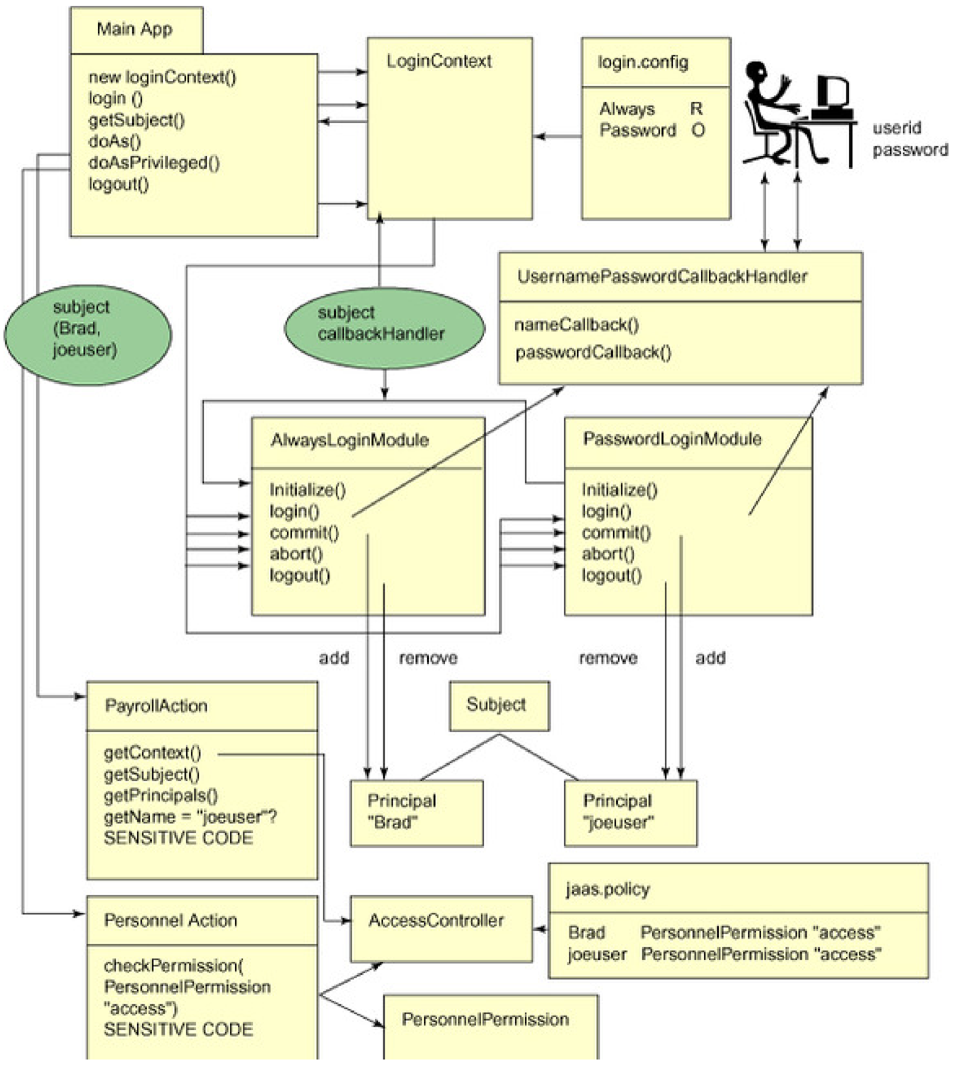
JDK 1.4 comes with the following PAM modules. We'll use one of them and also practice writing two of our own later in the tutorial:

- `com.sun.security.auth.module.NTLoginModule`
- `com.sun.security.auth.module.NTSystem`
- `com.sun.security.auth.module.JndiLoginModule`
- `com.sun.security.auth.module.KeyStoreLoginModule`
- `com.sun.security.auth.module.Krb5LoginModule`
- `com.sun.security.auth.module.SolarisSystem`
- `com.sun.security.auth.module.UnixLoginModule`
- `com.sun.security.auth.module.UnixSystem`

JAAS example and diagram

In this tutorial, we will look at code for a JAAS example application, piece by piece. To help keep track of the big picture, the figure below shows how all of these pieces fit together. The running example (main program JAASExample) first authenticates a user using two techniques, or login modules, and then allows or disallows (or authorizes) access to two pieces of sensitive code based on the results of the authentication step.

Here is a diagram of the JAASExample program. The flow of operations is described in the next panel.



JAASExample flow of operations

The following is a brief description of the overall authentication and authorization flow illustrated by the JAASExample diagram. Each of the steps below will be described in greater detail throughout the rest of the tutorial.

We begin with the first step of authentication, which is to create a login context and attempt to log in. The `LoginContext` is a Java class that uses information in the login.config file to decide which login modules to call and what criteria will be used to determine success. For this example, there are two login modules. The first, which is called `AlwaysLoginModule` does not require a password, so it always succeeds (this is unrealistic, but it's sufficient to illustrate how JAAS works). This module is tagged with the keyword `required`, meaning that it is required to succeed (which it always does). The second, which is called `PasswordLoginModule`, requires a password, but the success of this module is optional because it is tagged with the keyword `optional`. This means that the overall login could still succeed even if `PasswordLoginModule` fails.

After initialization, the selected login modules undergo a two-phase commit process controlled by the `LoginContext`. As part of this process, a `UsernamePasswordCallbackHandler` is called to get the username and password from an individual, who is represented by a `Subject` object. If the authentication is successful, a `Principal` is added to the `Subject`. A `Subject` may have many `Principal`s (in this case, "Brad" and "joeuser"), each of which authorizes the user for different levels of access to the system. This completes the authentication step.

Once authentication is complete, we use the `Subject` to try to execute some sensitive payroll-action code, using a programmatic authorization technique and the `doAs` method. JAAS checks to see if the `Subject` is authorized for access. If the `Subject` has a `Principal` that authorizes access to the payroll code, the execution is allowed to proceed. Otherwise, execution will be denied.

Next, we attempt to execute some sensitive personnel-action code using a declarative authorization technique and the `doAsPrivilaged` method. This time JAAS deploys a user-defined permission (`PersonnelPermission`), a Java policy file (jaas.policy), and the Java access controller (`AccessController`) to decide if the execution can proceed.

Authentication in JAAS

Overview

In this section, we'll concentrate on the elements of authentication in JAAS. We'll start with a description of a simple login and authentication procedure, which will give you a high-level view of the JAAS authentication architecture. Following that, we'll discuss each piece of the architecture in detail. At the end of the section, you'll have the opportunity to study the code for two login modules in detail.

If you haven't already done so, now is the time to download the source code for this tutorial, JavaSecurity2-source.jar. The source will better illustrate the steps outlined in the discussion that follows.

Subjects and Principals

A `Subject` is a Java object that represents a single entity, such as an individual. A single `Subject` can have a number of associated identities, each of which is represented by a `Principal` object. So, say a single `Subject` represents an employee who requires access to both the e-mail system and the accounting system. That `Subject` will have two `Principal`s, one associated with the employee's user ID for e-mail access and the other associated with his user ID for the accounting system.

`Principal`s are not persistent, so they must be added to the `Subject` each time the user logs in. A `Principal` is added to a `Subject` as a part of a successful authentication procedure. Likewise, a `Principal` is removed from the `Subject` if the authentication fails. Regardless of the success or failure of authentication, all `Principal`s are removed when the application performs a logout.

In addition to containing a set of `Principal`s, the `Subject` can contain two sets of credentials: one public and one private. A *credential* is a password, key, token, and so on. Access to the public and private credential sets is controlled by Java permissions, which we'll discuss later in the tutorial. A complete discussion of credentials is beyond the scope of this tutorial.

The Subject's methods

The `Subject` object has several methods, some of which are as follows:

- `subject.getPrincipals()` returns a set of `Principal` objects. Because the result is a ` Set`, the operations `remove()`, `add()`, and `contains()` apply.
- `subject.getPublicCredentials()` returns a set of the publicly accessible credentials associated with the `Subject`.
- `subject.getPrivateCredentials()` returns a set of the privately accessible credentials associated with the `Subject`.

The Principal interface

A `Principal` is a Java interface. A programmer-written `PrincipalImpl` object implements the `Principal` interface, along with the `Serializable` interface, a name string, a `getName()` method that returns this string, and other support methods such as `hashCode()`, `toString()`, and `equals()`.

`Principal`s are added to the `Subject` during the login process. As we will see later, declarative authorization is based on entries in a policy file. When an authorization request is made, the system's authorization policy is compared with the `Principal`s contained in the `Subject`. Authorization is granted if the `Subject` has a `Principal` that meets the security requirement in the policy file; otherwise it is denied.

PrincipalImpl

Here is the `PrincipalImpl` we'll use in this tutorial.

```
import java.io.Serializable;
import java.security.Principal;

//
// This class defines the principle object, which is just an encapsulated
// String name
public class PrincipalImpl implements Principal, Serializable {

     private String name;

     public PrincipalImpl(String n) {
       name = n;
     }

     public boolean equals(Object obj) {
       if (!(obj instanceof PrincipalImpl)) {
         return false;
       }
       PrincipalImpl pobj = (PrincipalImpl)obj;
       if (name.equals(pobj.getName())) {
         return true;
       }
       return false;
     }

     public String getName() {
       return name;
     }

     public int hashCode() {
       return name.hashCode();
     }

     public String toString() {
       return getName();
     }

}
```

Login configuration

JAAS allows tremendous flexibility in the kind of authentication procedures required of a `Subject`, the order in which they're performed, and the combinations of authentication success or failure required before the `Subject` is deemed authenticated.

JAAS uses the login.config file to specify the terms of authentication for each login module. The `login.config` file is specified on the Java execution command line with the property `-Djava.security.auth.login.config==login.config`. Java has a default login configuration file, so the double equals sign (`==`) replaces the system login configuration file. If a single equals sign were used the login.config file would add to, rather than replace, the system login configuration file. Because we don't know what might be in your system file, we do this to ensure reliable results across a wide range of tutorial users.

The login.config file contains a text string referenced in the `LoginContext` constructor and a list of the login procedures. Several parameters are used to specify the impact of the success or failure of a given login procedure on the overall authentication procedure. The parameters are as follows:

- `required` means that the login module must be successful. Other login modules will also be called even if it is not successful.
- `optional` means the login module can fail but the overall login may still be successful if another login module succeeds. If all the login modules are optional, at least one must be successful for the authentication to succeed.
- `requisite` means the login module must be successful, and if it fails no other login modules will be called.
- `sufficient` means that the overall login will be successful if the login module succeeds, assuming that no other required or requisite login modules fail.

Example login.config file

The login.config file we'll use in this tutorial is as follows:

```
JAASExample {
      AlwaysLoginModule required;
      PasswordLoginModule optional;
};
```

As you see, the `AlwaysLoginModule` must succeed and the `PasswordLoginModule` can either succeed or fail. This isn't a realistic scenario, but we'll modify these parameters later to see how different configurations change the code behavior.

The important thing to realize about this login configuration technique is that it leaves all major decision making (such as the types of authentication required and the specific criteria for success or failure of authentication) to be established at deployment time. A successful login will result in the addition of a new `Subject` to the `LoginContext`, with the addition of any number of successfully authenticated `Principal`s to that `Subject`.

Login context

A `LoginContext` is a Java class that is used to set up the login process, actually login, and get the `Subject` if the login is successful. It has four main methods, as follows:

- `LoginContext("JAASExample", newUsernamePasswoerdCallbackHandler())` is the constructor. It takes the string used in the login.config file as its first parameter, and a callback handler, which does the real work, as its second parameter. (We'll discuss callback handlers next.)
- `login()` actually attempts the login, subject to the rules specified in the login.config file.
- `getSubject()` returns the authenticated `Subject` if the login was an overall success.
- `logout()` logs the `Subject` off of the `LoginContext`.

The callback handler

The JAAS login uses a callback handler to get authentication information from users. The `CallbackHandler` is specified in the constructor of the `LoginContext` object. In this tutorial, the callback handler employs several prompts to get username and password information from the user. The handler's `handle()` method, which is called from a login module, takes an array of `Callback` objects as its parameter. During login, the handler iterates through the array of `Callback`s. The `handle()` method examines the type of the `Callback` object and performs the appropriate user action. `Callback` types are as follows:

- `NameCallback`
- `PasswordCallback`
- `TextInputCallback`
- `TextOutputCallback`
- `LanguageCallback`
- `ChoiceCallback`
- `ConfirmationCallback`

In some applications, no user interaction is required because JAAS will be used to interface to the operating system's authentication mechanism. In such cases, the `CallbackHandler` parameter in the `LoginContext` object will be null.

A callback handler at work

Below is the code for the `UsernamePasswordCallbackHandler` used in this tutorial. It is called once by the `AlwaysLoginModule` (with only one callback to get the userid) and once by the `PasswordLoginModule` (with two callbacks to get both a user ID and password).

```
import java.io.*;
import java.security.*;
import javax.security.auth.*;
import javax.security.auth.callback.*;
//
```

```
// This class implements a username/password callback handler that gets
// information from the user public class
UsernamePasswordCallbackHandler implements CallbackHandler {
     //
     // The handle method does all the work and iterates through the array
     // of callbacks, examines the type, and takes the appropriate user
     // interaction action.
     public void handle(Callback[] callbacks) throws
         UnsupportedCallbackException, IOException {

       for(int i=0;i&lt;callbacks.length;i++) {
         Callback cb = callbacks[i];
         //
         // Handle username aquisition
         if (cb instanceof NameCallback) {
           NameCallback nameCallback = (NameCallback)cb;
           System.out.print( nameCallback.getPrompt() + "? ");
           System.out.flush();
           String username = new BufferedReader(
               new InputStreamReader(System.in)).readLine();
           nameCallback.setName(username);
           //
           // Handle password aquisition
         } else if (cb instanceof PasswordCallback) {
           PasswordCallback passwordCallback = (PasswordCallback)cb;
           System.out.print( passwordCallback.getPrompt() + "? ");
           System.out.flush();
           String password = new BufferedReader(
               new InputStreamReader(System.in)).readLine();
           passwordCallback.setPassword(password.toCharArray());
           password = null;
           //
           // Other callback types are not handled here
         } else {
           throw new UnsupportedCallbackException(cb, "Unsupported
Callback Type");
         }
       }
     }
}
```

Login modules

A `LoginModule` is an interface for the methods necessary to participate in the JAAS authentication process. Because the success or failure of a specific login procedure may not be known until other login procedures are executed, a two-phase commit process is used to determine success. The following methods are implemented by a `LoginModule` object:

- `initialize( subject, callbackHandler, sharedState, options)` initializes the `LoginModule`. (Note that a discussion of `sharedState` and `options` is beyond the scope of this tutorial.)
- `login()` sets up any necessary callbacks, calls the `CallbackHandler` to handle them, and compares the returned information (that is, username and password) with the permitted values. If there's a match, the login module is successful, although it could still be aborted if another login module is unsuccessful, depending on the settings in the login.config file.
- `commit()` is called to determine success as part of the two-phase commit process. If all login modules are successful subject to the constraints specified in the login.config file, a new `Principal` is created along with the username, and added to the `Subject`'s principal set.
- `abort()` is called if the overall login is unsuccessful; if an abort occurs the internal `LoginModule` state must be cleaned up.

- `logout()` is called to remove the `Principal` from the `Subject`'s principal set and do other internal state cleanup.

The following two sections illustrate two login modules. The first, `AlwaysLoginModule`, is always successful. The second, `PasswordLoginModule`, is only successful if the user ID and password match certain hard-coded values. While neither of the example modules is a realistic implementation, together they demonstrate the results of a variety of JAAS options.

AlwaysLoginModule

The `AlwaysLoginModule` authentication will always be successful, so it's really only used to get a username via the `NameCallback` function. Assuming the other login modules are successful, the `AlwaysLoginModule`'s `commit()` method will create a new `PrincipalImpl` object with the username and add it to the `Subject`'s `Principal` set. The logout removes the `PrincipalImpl` from the `Subject`'s `Principal` set.

```
import java.security.*;
import javax.security.auth.*;
import javax.security.auth.spi.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import java.io.*;
import java.util.*;

// This is a JAAS Login Module that always succeeds.  While not realistic,
// it is designed to illustrate the bare bones structure of a Login Module
// and is used in examples that show the login configuration file
// operation.

public class AlwaysLoginModule implements LoginModule {

     private Subject subject;
     private Principal principal;
     private CallbackHandler callbackHandler;
     private String username;
     private boolean loginSuccess;
     //
     // Initialize sets up the login module.  sharedState and options are
     // advanced features not used here
     public void initialize(Subject sub, CallbackHandler cbh,
       Map sharedState, Map options) {

       subject = sub;
       callbackHandler = cbh;
       loginSuccess = false;
     }
     //
     // The login phase gets the userid from the user
     public boolean login() throws LoginException {
       //
       // Since we need input from a user, we need a callback handler
       if (callbackHandler == null) {
         throw new LoginException( "No CallbackHandler defined");

       }
       Callback[] callbacks = new Callback[1];
       callbacks[0] = new NameCallback("Username");
       //
       // Call the callback handler to get the username
       try {
         System.out.println( "\nAlwaysLoginModule Login" );
         callbackHandler.handle(callbacks);
```

```
        username = ((NameCallback)callbacks[0]).getName();
      } catch (IOException ioe) {
        throw new LoginException(ioe.toString());
      } catch (UnsupportedCallbackException uce) {
        throw new LoginException(uce.toString());
      }
      loginSuccess = true;
      System.out.println();
      System.out.println( "Login: AlwaysLoginModule SUCCESS" );
      return true;
    }
    //
    // The commit phase adds the principal if both the overall authentication
    // succeeds (which is why commit was called) as well as this particular
    // login module
    public boolean commit() throws LoginException {
      //
      // Check to see if this login module succeeded (which it always will

      // in this example)
      if (loginSuccess == false) {
        System.out.println( "Commit: AlwaysLoginModule FAIL" );
        return false;
      }
      //
      // If this login module succeeded too, then add the new principal
      // to the subject (if it does not already exist)
      principal = new PrincipalImpl(username);
      if (!(subject.getPrincipals().contains(principal))) {
        subject.getPrincipals().add(principal);
      }
      System.out.println( "Commit: AlwaysLoginModule SUCCESS" );
      return true;
    }
    //
    // The abort phase is called if the overall authentication fails, so
    // we have to clean up the internal state
    public boolean abort() throws LoginException {

      if (loginSuccess == false) {
        System.out.println( "Abort: AlwaysLoginModule FAIL" );
        principal = null;
        return false;
      }
      System.out.println( "Abort: AlwaysLoginModule SUCCESS" );
      logout();

      return true;
    }
    //
    // The logout phase cleans up the state
    public boolean logout() throws LoginException {

      subject.getPrincipals().remove(principal);
      loginSuccess = false;
      principal = null;
      System.out.println( "Logout: AlwaysLoginModule SUCCESS" );
      return true;
     }
}
```

PasswordLoginModule

The `PasswordLoginModule` uses a `NameCallback` to get the username and a `PasswordCallback` to get the password. If the username is "joeuser" and the password is "joe" then this authentication will be successful.

```
import java.security.*;
import javax.security.auth.*;
import javax.security.auth.spi.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import java.io.*;
import java.util.*;
//
// This is a JAAS Login Module that requires both a username and a
// password. The username must equal the hardcoded "joeuser" and
// the password must match the hardcoded "joeuserpw".
public class
PasswordLoginModule implements LoginModule {

     private Subject subject;
     private Principal principal;
     private CallbackHandler callbackHandler;
     private String username;
     private char[] password;
     private boolean loginSuccess;
     //
     // Initialize sets up the login module.  sharedState and options are
     // advanced features not used here
     public void initialize(Subject sub, CallbackHandler cbh,
       Map sharedState,Map options) {

       subject = sub;
       callbackHandler = cbh;
       loginSuccess = false;
       username = null;
       clearPassword();
     }
     //
     // The login phase gets the userid and password from the user and
     // compares them to the hardcoded values "joeuser" and "joeuserpw".
     public boolean login() throws LoginException {
       //
       // Since we need input from a user, we need a callback handler
       if (callbackHandler == null) {
          throw new LoginException("No CallbackHandler defined");
       }
       Callback[] callbacks = new Callback[2];
       callbacks[0] = new NameCallback("Username");
       callbacks[1] = new PasswordCallback("Password", false);
       //
       // Call the callback handler to get the username and password
       try {
         System.out.println( "\nPasswordLoginModule Login" );
         callbackHandler.handle(callbacks);
         username = ((NameCallback)callbacks[0]).getName();
         char[] temp = ((PasswordCallback)callbacks[1]).getPassword();
         password = new char[temp.length];
         System.arraycopy(temp, 0, password, 0, temp.length);
         ((PasswordCallback)callbacks[1]).clearPassword();
       } catch (IOException ioe) {
         throw new LoginException(ioe.toString());
       } catch (UnsupportedCallbackException uce) {
         throw new LoginException(uce.toString());
       }
       System.out.println();
```

```
    //
    // If username matches, go on to check password
    if ( "joeuser".equals(username)) {
      System.out.println
        ( "Login: PasswordLoginModule Username Matches" );
      if ( password.length == 5 &&
          password[0] == 'j' &&
          password[1] == 'o' &&
          password[2] == 'e' &&
          password[3] == 'p' &&
          password[4] == 'w' ) {
        //
        //If userid and password match, then login is a success
        System.out.println
          ( "Login: PasswordLoginModule Password Matches" );
        loginSuccess = true;
        System.out.println
          ( "Login: PasswordLoginModule SUCCESS" );
        clearPassword();
        return true;
      } else {
        System.out.println
         ( "Login: PasswordLoginModule Password Mismatch" );
      }
    } else {
      System.out.println( "Login: PasswordLoginModule Username Mismatch" );
    }
    //
    // If either mismatch, then this login module fails

    loginSuccess = false;
    username = null;
    clearPassword();
    System.out.println( "Login: PasswordLoginModule FAIL" );
    throw new FailedLoginException();
  }
  //
  // The commit phase adds the principal if both the overall
  // authentication succeeds (which is why commit was called)
  // as well as this particular login module
  public boolean commit() throws LoginException {

    //
    // Check to see if this login module succeeded
    if (loginSuccess == false) {
      System.out.println( "Commit: PasswordLoginModule FAIL" );
      return false;
    }
    // If this login module succeeded too, then add the new principal
    // to the subject (if it does not already exist)
    principal = new PrincipalImpl(username);
    if (!(subject.getPrincipals().contains(principal))) {
      subject.getPrincipals().add(principal);
    }
    username = null;
    System.out.println( "Commit: PasswordLoginModule SUCCESS" );
    return true;
  }
  //
  // The abort phase is called if the overall authentication fails, so
  // we have to cleanup the internal state
  public boolean abort() throws LoginException {

    if (loginSuccess == false) {
      System.out.println( "Abort: PasswordLoginModule FAIL" );
      principal = null;
      username = null;
```

```
          return false;
        }
        System.out.println( "Abort: PasswordLoginModule SUCCESS" );
        logout();
        return true;
      }
      //
      // The logout phase cleans up the state
      public boolean logout() throws LoginException {
        subject.getPrincipals().remove(principal);
        loginSuccess = false;
        username = null;
        principal = null;
        System.out.println( "Logout: PasswordLoginModule SUCCESS" );
        return true;
      }
      //
      // Private helper function to clear the password, a good programming
      // practice
      private void clearPassword() {
        if (password == null) {
          return;
        }
        for (int i=0;i<password.length;i++) {
          password[i] = ' ';
        }
        password = null;
      }
}
```

Authorization in JAAS

Overview

It is important to understand how the Java platform implements access control for authorization to understand the concepts we'll discuss in this section. The Java platform uses the notion of an *access control context* to determine the authority of the current thread of execution. Conceptually, this can be viewed as a token that is attached to every thread of execution. Prior to JAAS, access control was based on knowing the code source of the current Java .class file or the identity of the digital signer. Under this model, access control was based on knowing where code was coming from. With JAAS, we turn the model around. By adding the `Subject` to the access control context, we can begin to grant or deny access based on who is executing (or asking to execute) a given piece of code.

In this section, you'll learn about JAAS's mechanism for controlling access to sensitive code. We'll start with a description of how authorization works in JAAS, then proceed to a more in-depth description of each component of the authorization framework. We'll close this section with some code samples, used in the larger running example, that demonstrate both programmatic and declarative authorization techniques. At the end of this section, you should have a clear idea of how JAAS's authentication and authorization mechanisms work together to secure Java-based systems.

Access control and authority

Because a thread of execution can cross multiple modules with different context characteristics, the Java platform implements the concept of *least privilege*. In the whole stack of callers that pertain to a given thread of execution, where members of the call stack have different characteristics, the result used for determining authority is the intersection of all of these characteristics, or the least common denominator. For example, if a piece of calling code has limited authority (maybe it isn't trusted because it isn't signed), but it calls a piece of code that is more trusted (maybe this one has a signature), then the authority in the called code is *reduced* to match the lesser trust.

The authority characteristics contained in the access control context are compared against the Java permission `grant` statements in the policy file to indicate whether sensitive operations are allowed. This is done by a Java facility called the `AccessController`, which has interfaces for checking permissions programmatically and getting the current `Subject` associated with the active access control context as well. (The older Java Security Manager interfaces are becoming obsolete, so do use the `AccessController` methods.)

Binding the Subject to the access control context

Because a `Subject` can be authenticated after an application starts, there must be a way to dynamically bind the `Subject` to the access control context to create a single context that contains the code authority (where it was loaded from and who signed it) as well as the user authority (the `Subject`). For this, we use the method `Object doAs(Subject subject, PrivilegedAction action)`. This `doAs` method calls a class specially designed for authorization, which implements the `PrivilegedAction` interface.

Another call that can be used to specify an access control context, instead of using the thread's current one, is the method `Object doAsPrivileged(Subject, PrivilegedAction action, AccessControlContext acc)`. A special use of this is to set the `AccessControlContext` to null, which short-circuits the call stack at the point where the `doAsPrivileged` call occurs, allowing an *increase* in the authority while in the `PrivilegedAction` object. The authority will be later reduced when the object returns to the caller. Both techniques are illustrated later in this tutorial.

Both the `doAs` and `doAsPrivileged` methods come in forms that allow a `PrivilegedActionException` to be thrown.

Permissions

The Java platform has a number of built-in permissions that are used to control access to system resources. For example:

```
grant signedBy "Brad", codeBase "http://www.bradrubin.com" {
       permission java.io.FilePermission "/tmp/abc", "read";
};
```

allows code signed by "Brad" and loaded from "http://www.bradrubin.com" to read the `/tmp/abc` directory. See Resources for a complete listing of Java permissions.

Create your own permissions

The Java platform lets you create your own permission objects. Like regular permissions, these can be placed in the policy file and configured at deployment time. To demonstrate, take a look at the following `PersonnelPermission`. We'll use this code later on to allow access to some sensitive personnel-action code.

```
import java.security.*;
//
// Implement a user defined permission for access to the personnel //
code for this example public class PersonnelPermission extends
BasicPermission {

     public PersonnelPermission(String name) {
       super(name);
     }

     public PersonnelPermission(String name, String action) {
       super(name);
     }
}
```

Here's what you should note about the above permission: first, a constructor takes the user-defined name of the permission (in this example there is only one type, called *access*). A second constructor then takes an additional refining parameter called an *action*, although we won't use it here. For this example we'll use a `BasicPermission` class. If we need more features, we can use a `Permission` class.

Policy files

Policy files are the main mechanism to control access to system resources, including sensitive code. The policy file in this example is named jaas.policy, and is specified in the Java command line by the property `-Djava.security.policy==jaas.policy`. The double equals sign (`==`) replaces the system policy file, instead of adding to the system policy file permissions. Here's the jaas.policy file we're working with in this tutorial:

```
grant {
     permission javax.security.auth.AuthPermission "createLoginContext";
     permission javax.security.auth.AuthPermission "doAs";
     permission javax.security.auth.AuthPermission "doAsPrivileged";
     permission javax.security.auth.AuthPermission "modifyPrincipals";
     permission javax.security.auth.AuthPermission "getSubject"; };

grant      principal PrincipalImpl "Brad" {
     permission PersonnelPermission "access";
};
```

The system must have certain permissions -- that is, the first five in the example -- in order to bootstrap the JAAS mechanism. With those in place, the principal known as "Brad" is granted access to the `PersonnelPermission` user-defined permission.

JAAS main program example

Here's the main application program (the one called from the command line) for this example. It instantiates a login context, logs in, tries to execute two sensitive objects (one using programmatic authorization, the other using declarative authorization), and then logs out. Next, we'll look more closely at two elements of the main program: programmatic authorization and declarative authorization.

```
import java.security.*;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
//
// This is the main program in the JAAS Example.  It creates a Login
// Context, logs the user in based on the settings in the Login
// Configuration file,and calls two sensitive pieces of code, the
// first using programmatic authorization, and the second using
// declarative authorization.
public class JAASExample {

     static LoginContext lc = null;

     public static void main( String[] args) {
       //
       // Create a login context
       try {
         lc = new LoginContext("JAASExample",
            new UsernamePasswordCallbackHandler());
       } catch (LoginException le) {
         System.out.println( "Login Context Creation Error" );
         System.exit(1);
       }
       //

       // Login
       try {
         lc.login();
       } catch (LoginException le) {
         System.out.println( "\nOVERALL AUTHENTICATION FAILED\n" );
         System.exit(1);
       }
       System.out.println( "\nOVERALL AUTHENTICATION SUCCEEDED\n" );
       System.out.println( lc.getSubject() );
       //
       // Call the sensitive PayrollAction code, which uses programmatic
       // authorization.
       try {
         Subject.doAs( lc.getSubject(), new PayrollAction() );
       } catch (AccessControlException e) {
         System.out.println( "Payroll Access DENIED" );
       }
       //
       // Call the sensitive PersonnelAction code, which uses declarative
       // authorization.
       try {
         Subject.doAsPrivileged( lc.getSubject(), new PersonnelAction(), null );
       } catch (AccessControlException e) {
         System.out.println( "Personnel Access DENIED" );
       }
       try {
         lc.logout();
       } catch (LoginException le) {
         System.out.println( "Logout FAILED" );
         System.exit(1);
       }
       System.exit(0);

     }
}
```

Programmatic authorization example

In this example, we see how programmatic authority decisions are coded. The `PrivilegedAction` class is called by a `doAs` method from the main JAASExample program, so the authenticated `Subject` is bound to the application context on the thread when it enters the `run` method.

We retrieve the current `Subject` from the access controller, and iterate through any contained authenticated `Principal`s, looking for "joeuser". If we find him, we can do a sensitive operation and return. If not, we throw an `AccessControlException`. Obviously, in real life we would use a more administration-friendly and scalable technique rather than hard-coding user names directly into an application.

```
import java.io.*;
import java.security.*;
import javax.security.auth.*;
import javax.security.auth.login.*;
import java.util.*;
//
// This class is a sensitive Payroll function that demonstrates the
// use of programmatic authorization which only allows a subject
// that contains the principal "joeuser" in class PayrollAction
implements PrivilegedAction {
     public Object run() {
       // Get the passed in subject from the DoAs
       AccessControlContext context = AccessController.getContext();
       Subject subject = Subject.getSubject(
context );
       if (subject == null ) {
         throw new AccessControlException("Denied");
       }
       //
       // Iterate through the principal set looking for joeuser.  If
       // he is not found,
       Set principals = subject.getPrincipals();
       Iterator iterator = principals.iterator();
       while (iterator.hasNext()) {
         PrincipalImpl principal = (PrincipalImpl)iterator.next();
         if (principal.getName().equals( "joeuser" )) {
           System.out.println("joeuser has Payroll access\n");
           return new Integer(0);
         }
       }
       throw new AccessControlException("Denied");
     }
}
```

Declarative authorization example

In this example, we show how authorization checks can be declaratively controlled by a permission grant in a policy file, using the user-defined permission `PersonnelPermission`. We just ask the `AccessController` if this permission has been granted and it throws an `AccessControlException` if it hasn't, or keeps running if it has. We call this `PrivilegedAction` with a `doAsPrivileged` call and a null access control context in the main JAASExample code to short-circuit the call stack at the point of the call. This is needed because prior to the combining of the `Subject` with the context in the `doAsPrivileged` call, the `Subject` was not part of context and not authorized to the grant statement, and because of least privilege and the use of the intersection of permissions, an increase in authority would not otherwise be allowed.

```
import java.io.*;
import java.security.*;
//
// This class is a sensitive Personnel function that demonstrates
// the use of declarative authorization using the user defined
// permission PersonnelPermission, which throws an exception
// if it not granted
class PersonnelAction implements PrivilegedAction {
     public Object run() {
       AccessController.checkPermission(new PersonnelPermission("access"));
       System.out.println( "Subject has Personnel access\n");
       return new Integer(0);
     }
}
```

JAAS example

Overview

Finally, we've come to the fun part of the tutorial: learning by doing. We'll start by running the comprehensive JAASExample application, which contains all of the authentication and authorization mechanisms (and code) we've worked with throughout the tutorial. After running the example and seeing the results of our first configuration, we'll explore some different configuration options, checking the results as we go along. At the end of this final section of the tutorial, you'll have had your first taste of programming in JAAS.

Running the example

The JAASExample application is designed to show several authentication and authorization techniques and the impact of some configuration settings. Start running the example with the following statement, which you'll find in the file run.bat in the tutorial source file (see JavaSecurity2-source.jar).

```
 java -Djava.security.manager
-Djava.security.auth.login.config==login.config
-Djava.security.policy==jaas.policy JAASExample
```

The statement instructs the system's default security manager to use a login configuration file called login.config, to use a security policy file called jaas.policy, and, finally, to run the main application program JAASExample. Note that the double equals sign (`==`) indicates that the system default login configuration and policy files should *not* be added to the ones we've listed here. A single equals sign (`=`) would indicate the file should be concatenated with the system default.

Example results and notes

Here's the result of running JAASExample:

```
AlwaysLoginModule Login
Username? Brad

Login: AlwaysLoginModule SUCCESS

PasswordLoginModule Login
Username? joeuser
Password? joepw
```

```
Login: PasswordLoginModule Username Matches
Login: PasswordLoginModule Password Matches
Login: PasswordLoginModule SUCCESS
Commit: AlwaysLoginModule SUCCESS
Commit: PasswordLoginModule SUCCESS

OVERALL AUTHENTICATION SUCCEEDED

Subject:
Principal: Brad
Principal: joeuser

joeuser has Payroll access

Subject has Personnel access

Logout: AlwaysLoginModule SUCCESS
Logout: PassswordLoginModule SUCCESS
```

Here's a blow-by-blow description of the normal execution in the above result:

1. The `login.config` defines two login modules; `AlwaysLoginModule` is required. It runs first.
2. `AlwaysLoginModule` starts with the login phase, which calls the callback handler to get the username (`Brad`). Login is successful.
3. The second login module, `PasswordLoginModule`, is optional. It runs next, calling the callback handler to get both the username (`joeuser`) and password (`joepw`), both of which match. This login is also successful.
4. Because both the required and optional modules succeed, `commit` is called on both login modules, and the whole authentication is a success. As a result, the `Subject` now contains both `Principal`s, `Brad` and `joeuser`.
5. The payroll program, which uses programmatic authorization, checks to see if `joeuser` is in the `Subject`'s `Principal` set, which it is, and grants it access.
6. The personnel program, which uses declarative authorization, sees that there is a grant statement in the jaas.policy file, granting `Brad` permission to the `PersonnelPermission`, so it also succeeds.
7. Both login modules logout.

A failed authentication

Just for fun, let's see what happens when we do something wrong. In the example below, the setup is the same but we'll enter a wrong password for `joeuser`. Check the output below and see for yourself how it differs from the above results.

```
AlwaysLoginModule Login
Username? Brad

Login: AlwaysLoginModule SUCCESS

PasswordLoginModule Login
Username? joeuser
Password? wrongpw

Login: PasswordLoginModule Username Matches
Login: PasswordLoginModule Password Mismatch
Login: PasswordLoginModule FAIL
```

```
Commit: AlwaysLoginModule SUCCESS
Commit: PasswordLoginModule FAIL

OVERALL AUTHENTICATION SUCCEEDED

Subject:
Principal: Brad

Payroll Access DENIED
Subject has Personnel access

Logout: AlwaysLoginModule SUCCESS
Logout: PasswordLoginModule SUCCESS
```

As you can see, the `PasswordLoginModule` login has failed. Because this module is configured `optional` in the login.config file, however, the overall authentication was still a success. The difference is that only the `Brad Principal` has been added to the `Subject`. The payroll program could not find a `joeuser Principal`, so access was denied. The personnel program was able to match the `Brad Principal` with the `Brad` grant statement, so it was successfully added and access was granted.

In the next several sections, we'll try out a few different variations in how we configure the login.config file, then check the results for each new config.

Variation 1: Login configuration

First, let's see what happens if we change the login.config file so that *both* login modules are required in order for authentication to be a success. The new config is:

```
JAASExample {
      AlwaysLoginModule required;
      PasswordLoginModule required;
};
```

And here's the resulting output:

```
AlwaysLoginModule Login
Username? Brad

Login: AlwaysLoginModule SUCCESS

PasswordLoginModule Login
Username? joeuser
Password? wrongpw

Login: PasswordLoginModule Username Matches
Login: PasswordLoginModule Password Mismatch
Login: PasswordLoginModule FAIL
Abort: AlwaysLoginModule SUCCESS
Logout: AlwaysLoginModule SUCCESS
Abort: PasswordLoginModule FAIL

OVERALL AUTHENTICATION FAILED
```

When `joeuser` entered the wrong password, the `PasswordLoginModule` failed just like it did before. Because this module was required, however, the abort phase ran and the overall authentication failed. No sensitive code was executed.

Variation 2: The power of PAM

This variation is designed to demonstrate the utility of Pluggable Authentication Modules. We go back to the original login.config file, which says that `AlwaysLoginModule` is required and `PasswordLoginModule` is optional, and add an `NTLoginModule` (or any other module appropriate for your platform) to the file. The new module will be `required`. The modified login.config file should look like this:

```
JAASExample {
      AlwaysLoginModule required;
      PasswordLoginModule optional;
      com.sun.security.auth.module.NTLoginModule required;
};
```

Next, run the example. In the output below you'll note that a new authentication method has been added, as well as several nifty new `Principal`s (and one public credential).

```
AlwaysLoginModule Login
Username? Brad

Login: AlwaysLoginModule SUCCESS

PasswordLoginModule Login
Username? joeuser
Password? joepw

Login: PasswordLoginModule Username Matches
Login: PasswordLoginModule Password Matches
Login: PasswordLoginModule SUCCESS
Commit: AlwaysLoginModule SUCCESS
Commit: PasswordLoginModule SUCCESS

OVERALL AUTHENTICATION SUCCEEDED

Subject:
Principal: Brad
           Principal: joeuser
           Principal: NTUserPrincipal: Brad
           Principal: NTDomainPrincipal: WORKGROUP
           Principal: NTSidUserPrincipal:
S-1-5-21-2025429265-1580813891-854245398-1004
           Principal: NTSidPrimaryGroupPrincipal:
S-1-5-21-2025429265-1580418891-85 4245398-513
           Principal: NTSidGroupPrincipal:
S-1-5-21-2025429265-1580818891-854245398-513
           Principal: NTSidGroupPrincipal: S-1-1-0
           Principal: NTSidGroupPrincipal: S-1-5-32-544
           Principal: NTSidGroupPrincipal: S-1-5-32-545
           Principal: NTSidGroupPrincipal: S-1-5-5-0-49575

           Principal: NTSidGroupPrincipal: S-1-2-0
           Principal: NTSidGroupPrincipal: S-1-5-4
           Principal: NTSidGroupPrincipal: S-1-5-11
           Public Credential: NTNumericCredential: 1240

joeuser has Payroll access

Subject has Personnel access

Logout: AlwaysLoginModule SUCCESS
Logout: PasswordLoginModule SUCCESS
```

And the cool thing is, we didn't even touch our application code. All of the above changes come from the native OS authentication mechanism. This should give you an inkling of the power of PAM.

Variation 3: Policy file configuration

In this final variation, we'll see what happens when we modify the access control policy. We start by modifying the grant file in the original `login.config` so that `joeuser`, not `Brad` has `PersonnelPermission`, as shown below:

```
grant Principal PrincipalImpl "joeuser" {
     permission PersonnelPermission "access";
};
```

Next, we run the application, entering the wrong password for `joeuser`. The results are shown below:

```
AlwaysLoginModule Login
Username? Brad

Login: AlwaysLoginModule SUCCESS

PasswordLoginModule Login
Username? joeuser
Password? wrongpw


Login: PasswordLoginModule Username Matches
Login: PasswordLoginModule Password Mismatch
Login: PasswordLoginModule FAIL
Commit: AlwaysLoginModule SUCCESS
Commit: PasswordLoginModule FAIL

OVERALL AUTHENTICATION SUCCEEDED

Subject:
Principal: Brad

Payroll Access DENIED
Personnel Access DENIED
Logout: AlwaysLoginModule SUCCESS
Logout: PasswordLoginModule SUCCESS
```

As you can see, only `Brad` is in the `Subject`'s `Principal` set. Both the attempted payroll access and the attempted personnel access have failed. Why? The first failed because there is no `Principal` named `joeuser`, and the second failed because there is no grant permission statement for `Brad`.

Don't stop here

In this section, we've pulled all of the JAAS authentication and authorization pieces together to illustrate a complete running JAAS application. We've also played with several variations to get a feel for what's really going on under the hood and how flexible this architecture is for application security.

To expand on what you've learned here, you should continue playing with JAAS and see what happens when you try different login configurations. For example, if you have Kerberos running in your installation, try running the Kerberos login module.

Wrap-up

Summary

This tutorial serves as an introduction to the Java platform authentication and authorization service, known as JAAS. In addition to becoming familiar with all the essential components of JAAS, you've had a hands-on introduction to several login modules and learned the basics of working with JAAS on the command line. You've also had the opportunity to run an actual JAAS application and try out several successful and unsuccessful configurations.

At the close of this tutorial, you should find yourself well poised for continuing exploration of the JAAS programming framework. Combining JAAS's authentication and authorization techniques with the cryptographic technologies discussed in Part 1 of this tutorial will enable you to architect and implement a vast array of application security solutions. To further your learning, you should continue to play with JAAS, exploring a number of different login modules, configurations, and security scenarios.

Resources

- Read Part 1 of this tutorial series, "Crypto basics," also by Brad Rubin.
- See the Java Developer Connection for a complete listing of Java permissions.
- While not covered in this tutorial, the Java General Security Service (JGSS), new with JDK 1.4, provides a generic framework for securely exchanging messages between applications. A recently released whitepaper from Sun discusses how JAAS, JGSS, and Kerberos can be used to provide single sign-on application security.
- Sun also hosts several tutorials and user guides describing the different uses and procedures of JAAS and JGSS. One particularly good reference guide describes when to use JGSS versus JSEE.
- See Sun Microsystems's Java Security site to learn more about the most current Java security technologies.
- Joseph Sinclair offers a three-pronged solution for identifying users in the series "Securing systems" (*developerWorks*, June 2001).
- Once you've got the basics down, Carlos Fonseca will show you how to "Extend JAAS for class instance-level authorization" (*developerWorks*, April 2002).
- In "Enhance Java GSSAPI with a login interface using JAAS" Thomas Owusu provides some insight on credentials and secret keys (*developerWorks*, November 2001).
- For an overall discussion of Web security and Java technology, see *Web Security, Privacy, and Commerce, 2nd Edition*, by Simson Garfinkel and Gene Spafford, O'Reilly, 2002.
- If you want to focus more on Java security, see *Professional Java Security*, by Jess Garms and Daniel Somerfield, Wrox Press, 2001.
- Another great resource for learning about Java security is *Java Security*, by Scott Oaks, O'Reilly & Associates, 2001.
- Find out what everyone needs to know about security in order to survive and be competitive in *Secrets and Lies: Digital Security in a Networked World* (http://www.counterpane.com/sandl.html), by Bruce Schneier, 2000.
- If you want to focus on authentication technologies, see *Authentication: From passwords to public keys*, by Richard E. Smith, Addison-Wesley, 2002.
- The IBM Java Security Research page details various security projects in the works.
- Visit the Tivoli Developer domain for help in building and maintaining the security of your e-business.
- You'll find hundreds of articles about every aspect of Java programming in the *developerWorks*Java technology zone.
- Download the complete source code and classes used in this tutorial, JavaSecurity2-source.jar.
- The Java 2 platform, Standard Edition is available from Sun Microsystems.
- See the developerWorks tutorials page for a complete listing of Java technology-related free tutorials from *developerWorks*.

About the author

Brad Rubin

Brad Rubin is principal of Brad Rubin & Associates Inc., a computer-security consulting company specializing in wireless network and Java application security and education. Brad spent 14 years with IBM in Rochester, MN, working on all facets of the AS/400 hardware and software development, starting with its first release. He was a key player in IBM's move to embrace the Java platform, and was lead architect of IBM's largest Java application, a business application framework product called SanFrancisco (now part of WebSphere). He was also chief technology officer for the Data Storage Division of Imation Corp., as well as the leader of its R&D organization. Brad has degrees in Computer and Electrical Engineering, and a Doctorate in Computer Science from the University of Wisconsin, Madison. He currently teaches the Senior Design course in Electrical and Computer Engineering at the University of Minnesota, and will develop and teach the university's Computer Security course in Fall 2002.