

Develop JAAS Security on JBoss application server

By Nima Goodarzi – November 2008

Introduction

A few days ago I was proposed to develop an airline ticketing system using JavaEE platform. For this system I decided to use EJB3 and JSF running on JBoss application server.

As long as security is a vital concern in such applications, I decided to use JAAS (Java Authentication and Authorization Service) to implement authentication and authorization.

After searching for the required configurations to implement a JAAS based security on JBoss, I couldn't find anything useful, even in the JBoss documents! (JavaEE developers are not very unfamiliar with this).

It took a while for me to find all the required settings and run my project under JAAS technology on the JBoss application server, so I decided to share my knowledge and document it, hope to be useful for somebody.

For this project I used EJB3.0, JSF 1.2, JBoss AS 4.2.3 GA

As long as we keep our users information in database, we need to setup required tables to store users information.

For this example we need a table named **user** with two columns, **username** and **password** to keep users authentication information and a table named **user_role** with two columns, **user** and **role** to keep users authorization information.

Step-By-Step Guide

First Step: Define Application Policy

As the first step we need to define a security domain for our project, security domain is the JNDI name of the security manager interface implementation that JBoss uses for the EJB and web containers. This is an object that implements both of the AuthenticationManager and RealmMapping interfaces.

To define our own security domain in JBoss, we need to define a new application policy. For this reason we modify the login-config.xml file under <JBOSS_HOME>/server/<PROFILE>/conf directory. For example if you have installed your JBoss application server in the C:\ partition and you are using the default profile, the path of the login-config.xml file should be:

```
C:\jboss-4.2.3.GA\server\default\conf\login-config.xml
```

To define a new application policy, you need to add a new <application-policy> element to the login-config.xml.

This is my <application-policy> element:

```
<application-policy name="airbus">
  <authentication>
    <login-module code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
      flag="required">
      <module-option name="dsJndiName">java:/airbusDS</module-option>
      <module-option name="principalsQuery">
        select password from user where username=?
      </module-option>
      <module-option name="rolesQuery">
        select role,'Roles' from user_role where user=?
      </module-option>
      <module-option name="hashAlgorithm">MD5</module-option>
      <module-option name="hashEncoding">base64</module-option>
    </login-module>
  </authentication>
</application-policy>
```

Now we describe each property in the above <application-policy> element:

name (airbus): This is the name of this application-policy (airbus is my project name) which will be used later while defining your security domain.

dsJndiName (java:/airbusDS): As long as we store our user's information in database, we need to check their credentials against database to authenticate/authorize them. The **java:/airbusDS** here is the JNDI name of the datasource pointing to my database.

principalsQuery: This is the SQL Query that is used to get the principals of the user. Here, we get the password of the user based on the provided username.

rolesQuery: This query is used to get roles of the provided username.

hashAlgorithm (MD5): This is the used encryption algorithm for the user's password in database. In this example we encrypt our user passwords using MD5 encryption algorithm, so the JBoss authenticator must encrypt the provided password using the same algorithm before comparing it with the actual password.

hashEncoding (base64): This is the encoding used to transform the user password data into a 64 bit string.

Second Step: Create Security Domain

After defining the application policy, you need to define the security domain, which has been introduced in the first step.

To define a security domain, you need to create a file named `jboss-web.xml` in the `WEB-INF` directory of your web application.

Example of `jboss-web.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
  <security-domain>java:/jaas/airbus</security-domain>
</jboss-web>
```

airbus in the above example is the name of the defined application policy in the first step.

Third Step: Secure The Application

In this step we secure the web application. For this reason we need to modify the `web.xml` file in the `WEB-INF` directory.

These are changes need to apply to the `web.xml` file:

1- Authentication:

We should tell JBoss to authenticate users before allowing them to enter the application. This is done by adding `<login-config>` element to the `web.xml`.

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.jsp</form-login-page>
    <form-error-page>/loginfail.jsp</form-error-page>
  </form-login-config>
</login-config>
```

In this example we tell JBoss that we need a form-based authentication (redirects users to our own login form). `login.jsp` is the designed login page and if the authentication fails, users are redirected to `loginfail.jsp`.

2- Create Login Page

Login page is a very simple JSP page with a form where the action of the form is set to `j_security_check` and a text box, `j_username` for username and a password box, `j_password` for Password.

3- Secure Web Resources:

Now we define our secured resources and required roles to access them. This is done by adding `<security-constraint>` element to `web.xml`.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>AdminPages</web-resource-name>
    <url-pattern>/faces/admin/*</url-pattern>
  </web-resource-collection>

  <auth-constraint>
    <role-name>administrator</role-name>
    <role-name>supervisor</role-name>
  </auth-constraint>
</security-constraint>
```

In this example, we define all the resources under `/admin` directory as secured resources and only users with the administrator or supervisor roles are allowed to access these resources. We can define as many resources as we need in the same way.

Note that when you define a set of resources as secured resources, none of these resources are available for users out of the allowed roles. For example in the example above, even images and stylesheets in the admin directory are blocked for the users without administrator or supervisor role.

You can use `HttpServletRequest isCallerInRole(String roleName)` method to see whether the logged in user has the specified role or not.

4- Secure EJB Methods:

The next step is to secure EJB methods; means that only allowed users can call a secured EJB method.

In EJB3 we can use annotations to secure methods.

First, we should annotate the EJB class with `@SecurityDomain("<Security Domain Name>")` annotation.

In our example we use `@SecurityDomain("airbus")` annotation which, airbus is the name of our security domain.

Then we annotate methods with `@RolesAllowed({"<Role Name>"})` which, Role Name is the allowed roles to call this method. We can also use `@PermitAll` to allow every body to access the method or `@DenyAll` to deny any access to the method.

You can use EJB Context `isCallerInRole(String roleName)` method to see whether the logged in user has the specified role or not.

This is an example of a secured EJB Session Bean:

```
@Stateless
@SecurityDomain("airbus")
public class BaseServiceImpl implements BaseService {
    static Logger logger = Logger
        .getLogger("my.com.airbus.service.impl.BaseServiceImpl");

    @PersistenceContext
    private EntityManager em;

    @PermitAll
    public Object findById(Class clazz, Long id) {
        logger.info("Find class: " + clazz.toString() + " By ID: " + id);
        return em.find(clazz, id);
    }

    @RolesAllowed({"administrator", "supervisor"})
    public void remove(Object obj){
        logger.info("Delete class: " + obj.getClass().getName());
        obj = em.merge(obj);
        em.remove(obj);
    }
}
```

In this example everybody is allowed to call the `findById` method, but only administrators and supervisors can call the `remove` method.