Security Application Block Hands-On Lab for Enterprise Library

This walkthrough should act as your guide for learning about the Security Application Block and will allow you to practice employing its capabilities in various application contexts.

After completing this lab, you will be able to do the following:

- You will be able to use ASP.NET Membership for application authentication and authorization.
- You will be able to use the Enterprise Library Security Application Block to add rule-based authorization.

This hands-on lab includes the following two labs:

- Lab 1: Secure an Application
- Lab 2: Use Rule-Based Authorization in an Application

The estimated completion for this lab is **30 minutes**.

Authors

These Hands-On Labs were produced by the following individuals:

- Product/Program Management: Grigori Melnik (Microsoft Corporation)
- Development: Chris Tavares (Microsoft Corporation), Nicolas Botto (Digit Factory), Olaf Conijn (Olaf Conijn BV), Fernando Simonazzi (Clarius Consulting), Erik Renaud (nVentive Inc.)
- Testing: Rick Carr (DCB Software Testing, Inc) plus everyone above
- Documentation: Alex Homer and RoAnn Corbisier (Microsoft Corporation) and Dennis DeWitt (Linda Werner & Associates Inc)

All of the Hands-On Labs use a simplified approach to object generation through Unity and the Enterprise Library container. The recommended approach when developing applications is to generate instances of Enterprise Library objects using dependency injection to inject instances of the required objects into your application classes, thereby realizing all of the advantages that this technique offers.

However, to simplify the examples and make it easier to see the code that uses the features of each of the Enterprise Library Application Blocks, the examples in the Hands-On Labs use the simpler approach to resolve Enterprise Library objects from the container by using the **GetInstance** method of the container service locator. You will see this demonstrated in each of the examples.

To learn more about using dependency injection to create instances of Enterprise Library objects, see the documentation installed with the Enterprise Library, or available on MSDN® at http://msdn.microsoft.com/entlib/.

Lab 1: Secure an Application

In this lab, you will add authentication and role-based authorization to an existing application. You will configure application users and roles via the ASP.NET Membership API.

To begin this exercise, open the BugSmak.sln file located in the ex01\begin folder.

To add authentication to the application

1. Select the **Debug | Start Without Debugging** menu command to run the application. The application as yet is unable to authenticate users.

   The application will be configured to use the ASP.NET Membership API for authentication (login security).

2. Close the application.

3. Select the **Security\SecurityHelper.cs** file in the Solution Explorer. Select the **View | Code** menu command.

   Add the following namespace inclusion to the list of namespaces at the top of the file:

```
using System.Web.Security;
```

4. Add the following highlighted code to the **Authenticate** method.

```
public static bool Authenticate(string username, string password)
{
  bool authenticated = false;

  // TODO: Authenticate Credentials
  authenticated = Membership.ValidateUser(username, password);

  // TODO: Get Roles
  return authenticated;
}
```

   The **Authenticate** method is called by the **LoginForm** to validate the user credentials. The **Membership.ValidateUser** method is called to perform the validation.

   The membership system uses a provider model so that the application is not tied to a particular data store. ASP.NET ships with two membership providers: one that uses Microsoft® SQL Server® as a data source and another that uses Windows® Active Directory®.

   It is also possible to create a custom membership provider; this was done in the sample application to read the application members from an XML file.

5. Select the **Security | Providers | ReadOnlyXmlMembershipProvider.cs** file in the Solution Explorer. Select the **View | Code** menu command and review the code.

The **ReadOnlyXmlMembershipProvider** (inherits from **MembershipProvider**) is an example of a custom membership provider. This implementation, which reads an unencrypted XML file, is not meant as an example of good practice, but useful for this laboratory exercise.

6. Select the **App.config** file in the Solution Explorer, right-click, and select **Open**.

   Review the membership provider configuration. The authentication data store is identified as the **Users.xml** file.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
 <system.web>
  <membership defaultProvider="ReadOnlyXmlMembershipProvider">
   <providers>
    <add name="ReadOnlyXmlMembershipProvider"
         type="BugSmak.Security.Providers.ReadOnlyXmlMembershipProvider,
               BugSmak"
         description="Read-only XML membership provider"
         xmlFileName="Users.xml" />
   </providers>
  </membership>
  ...
 </system.web>
</configuration>
```

   Once you have a custom membership provider, you can configure your application to use that provider in the same way that you configure the application to use an ASP.NET provider. The Membership class will automatically invoke your custom provider to communicate with your authentication data source.

7. Select the **Users.xml** file in the Solution Explorer. Select the **View | Open** menu command.

   The following users have been pre-defined.

Username	Password	Role(s)
Tom	P@ssw0rd	Employee
Dick	P@ssw0rd	Developer
Harry	P@ssw0rd	Manager

8. Select the **Debug | Start Without Debugging** menu command to run the application.

   Sign in as **Tom**, **Dick**, or **Harry** to confirm the correct setup of the membership provider.

   Select the **File | Sign Out** menu command and attempt to sign in with an incorrect username or password.

The Password field has been initialized to **P@ssw0rd** (see LoginForm.cs) to save you repeatedly typing the password during the laboratory exercise.

> The application is the skeleton of a simple bug tracking system. The application has a skeleton for the following functions: raise a bug, assign a bug to a developer for resolution, and resolve a bug.
>
> The password is case sensitive, but the username is not.

9. Sign in as **Tom**. Select the **Tasks | Raise New Bug** menu command. You are confronted with a message: "Sorry, you aren't allowed to access that form." Similarly, attempt to open the other tasks (**Assign Bug**, and **Resolve Bug**).
10. Close the application.

To add role-based authorization to the application

1. Select the **TaskForms\RaiseBug.cs** file in the Solution Explorer. Select the **View | Code** menu command.

   Review the PrincipalPermissions.

   The RaiseBug form demands that the user have any of the developer, employee, or manager roles. Attempting to run the form without the necessary authorization results in a **SecurityException**, which is caught by the MainForm (see MainForm.cs).

   The following role requirements are enforced via PrincipalPermissions:

TaskForm	Role required
RaiseBug	Employee, Developer, or Manager
AssignBug	Manager
ResolveBug	Developer or Manager

   Authentication has been implemented, but the user roles have not been determined. The application will be configured to use a membership **RoleProvider** to retrieve user roles later in this section of the lab.

2. Select the **Security\SecurityHelper.cs** file in the Solution Explorer. Select the **View | Code** menu command.

Add the following highlighted code to the **Authenticate** method.

```
public static bool Authenticate(string username, string password)
{
  bool authenticated = false;

  // TODO: Authenticate Credentials
  authenticated = Membership.ValidateUser(username, password);

  // TODO: Get Roles
  if (!authenticated)
      return false;

  IIdentity identity;
  identity = new GenericIdentity(username, Membership.Provider.Name);

  string[] roles = Roles.GetRolesForUser(identity.Name);
  IPrincipal principal = new GenericPrincipal(identity, roles);

  // Place user's principal on the thread
  Thread.CurrentPrincipal = principal;

  return authenticated;
}
```

The roles are retrieved from the Users.xml file via a custom **RoleProvider** (ReadOnlyXmlRoleProvider.cs), and a new principal is created, which contains the user's identity and roles.

To use the authenticated principal in our application we place it on the thread.

3. Select the **App.config** file in the Solution Explorer. Select the **View | Open** menu command.

   Review the role manager provider configuration. The data store is identified as the **Users.xml** file.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
 <system.web>
  ...
  <roleManager enabled="true"
               defaultProvider="ReadOnlyXmlRoleProvider">
   <providers>
    <add name="ReadOnlyXmlRoleProvider"
         type="BugSmak.Security.Providers.ReadOnlyXmlRoleProvider, BugSmak"
         description="Read-only XML role provider"
         xmlFileName="Users.xml" />
   </providers>
  </roleManager>
 </system.web>
```

```
</configuration>
```

4. Select the **Debug | Start Without Debugging** menu command to run the application.

   Sign in as **Tom**, **Dick**, and **Harry**, in turn, to confirm the correct setup of the role provider.

User	Task access
Tom (Employee)	Raise new bug
Dick (Developer)	Raise new bug and resolve bug
Harry (Manager)	Raise new bug, resolve bug, and assign bug

To verify that you have completed the exercise correctly, you can use the solution provided in the ex01\end folder.

Lab 2: Use Rule-Based Authorization in an Application

In this lab, you will use an AuthorizationProvider to secure application tasks.

In the previous lab, you secured an application using role-based authorization. Role-based authorization can be difficult to configure because roles do not always map neatly to tasks or use cases.

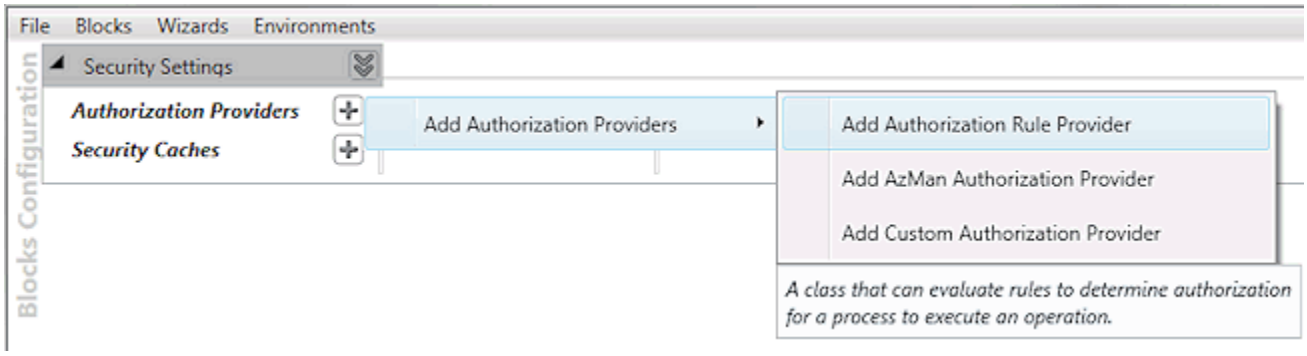
You will use the AuthorizationProvider to do the following:

- Map task-based authorization to complex combinations of roles.
- Extract authorization from the application code.

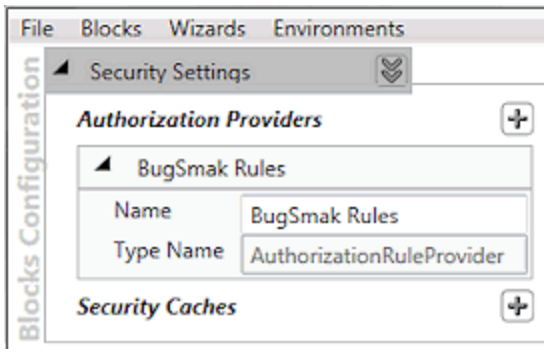
To begin this exercise, open the BugSmak.sln file located in the ex02\begin folder.

To run the Enterprise Library Configuration tool

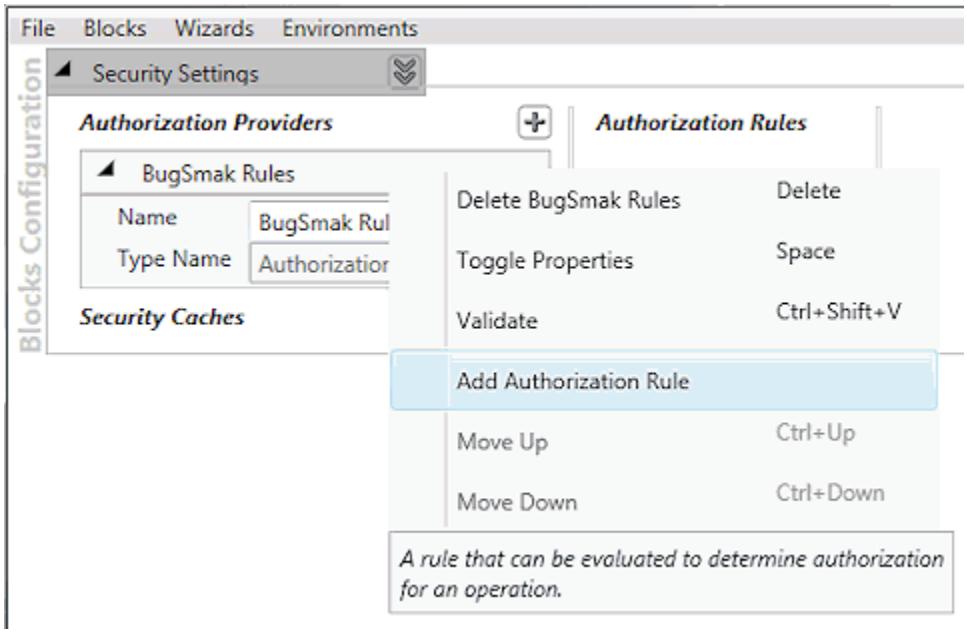
1. Open the application configuration in the configuration editor using one of these approaches:
    - Start the appropriate version of the stand-alone tool from the Windows Start menu (select **All Programs | Microsoft patterns & practices | Enterprise Library 5.0 | Enterprise Library Configuration**) and open the App.config file.
    - In Visual Studio®, right-click on the App.config file in Solution Explorer and click **Edit Enterprise Library V5 Configuration**.
2. Click the **Blocks** menu and select **Add Security Settings**. This adds a section to the tool containing two columns. The first has sections for **Authorization Providers** and **Security Caches**, and the second is where you configure **Authorization Rules**.
3. Add a new authorization rule provider. Click the plus-sign icon next to **Authorization Providers**, point to **Add Authorization Providers**, and click **Add Authorization Rule Provider**, as shown below.



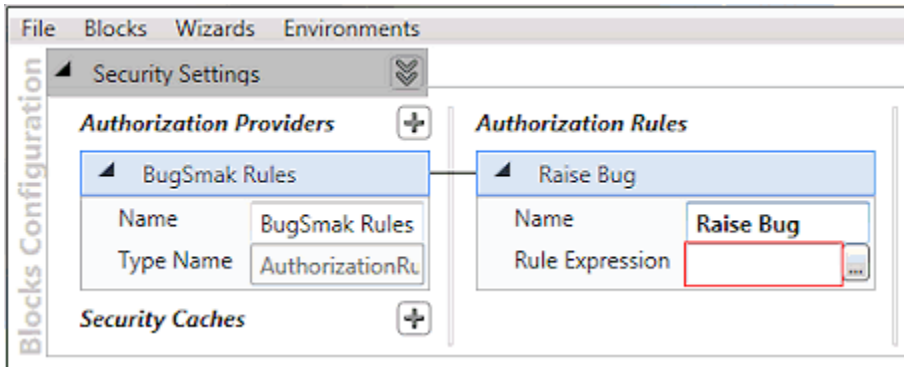
4. Change the **Name** property to **BugSmak Rules**, as shown here.



5. Right-click the title of the **BugSmak Rules** provider item and click **Add Authorization Rule**. A new **Authorization Rule** item appears in the **Authorization Rules** column, as you can see below.

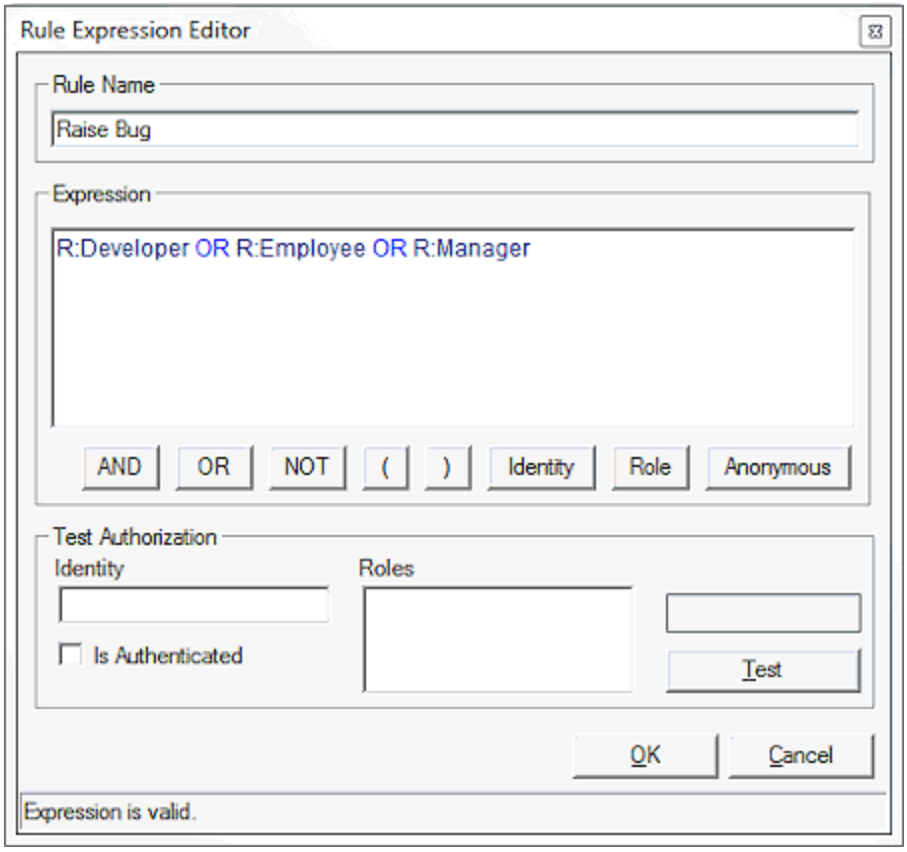


6. Change the **Name** property of the new rule to **Raise Bug**.



7. Click the ellipsis button (**…**) in the **Rule Expression** property to open the rule expression editor dialog. Type in the **Expression** text area, and use the buttons below it, to create the following rule:

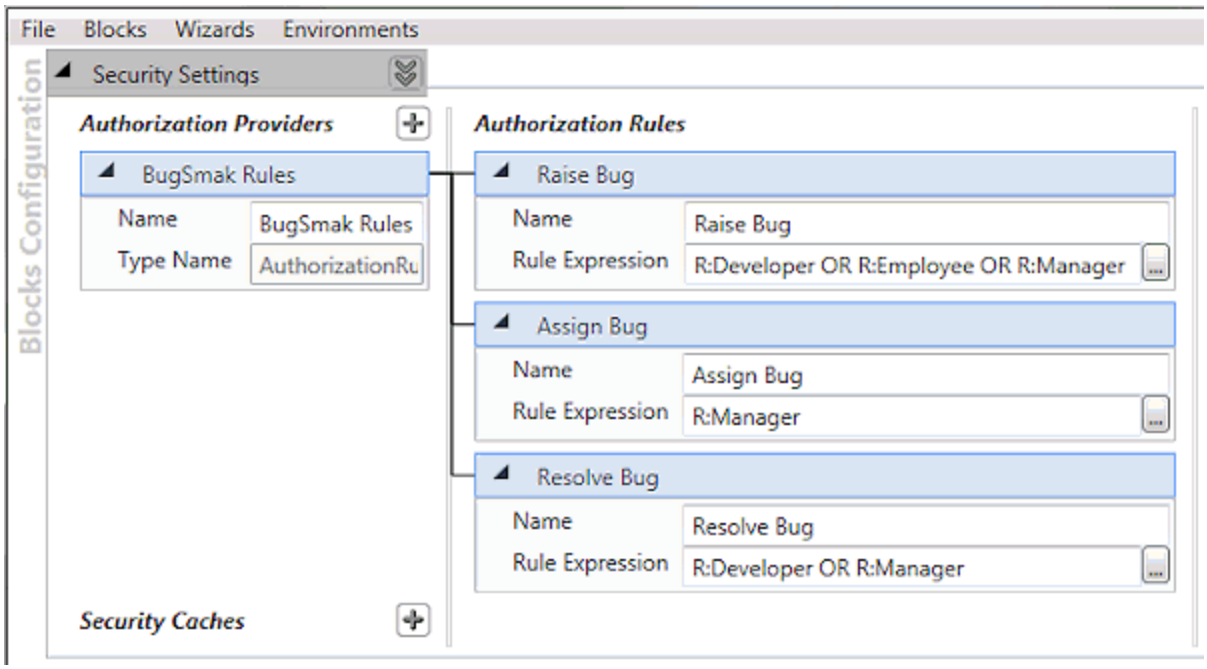
R:Developer OR R:Employee OR R:Manager



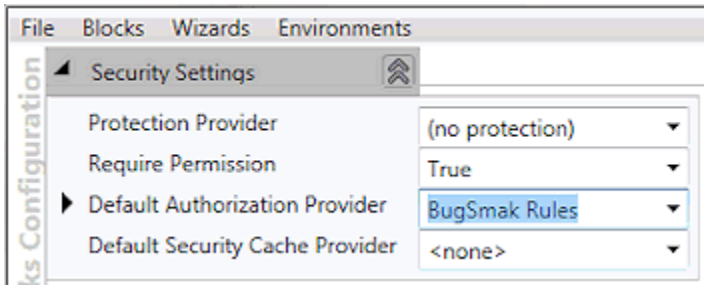
The user must be in the developer, employee, or manager role.

8. Click **OK** to close the rule expression editor dialog.
9. Now repeat steps 5 through 8 to add the following new rules and expressions. Notice that you can, if you wish, type the rule directly into the rule expression property instead of using the rule expression editor dialog.

Rule name	Expression
Assign Bug	R:Manager
Resolve Bug	R:Developer OR R:Manager



10. Click the chevron arrow to the right of the **Security Settings** title to display the properties for this configuration section. In the drop-down list for the **Default Authorization Provider** property, select the **BugSmak Rules** rule provider.



11. Select the **File | Save** menu command to save the configuration.

To add task-based authorization to the application

1. Select the **TaskForms\RaiseBug.cs** file in the Solution Explorer. Select the **View | Code** menu command.

   The form no longer uses the hard-coded PrincipalPermissions (it is commented out). Instead, the more flexible task authorization is checked before the object is constructed.

   AssignBug.cs and ResolveBug.cs have similar code.

```
//[PrincipalPermission(SecurityAction.Demand, Role = "Employee")]
//[PrincipalPermission(SecurityAction.Demand, Role = "Developer")]
//[PrincipalPermission(SecurityAction.Demand, Role = "Manager")]
public static RaiseBug Create()
{
  // TODO: Check Authorization
```

```
  if (!SecurityHelper.Authorized(AuthRule.Raise))
  {
    throw new SecurityException();
  }

  return new RaiseBug();
}
```

For compile-time checking, the rule names (such as Raise Bug) are mapped to constants in the Constants.cs file.

2. Select the **Project | Add Reference** menu command. Select the **Browse** tab and select the following assembly located in the Enterprise Library **bin** folder (typically C:\Program Files\Microsoft Enterprise Library 5.0\Bin).
    - Microsoft.Practices.EnterpriseLibrary.Security.dll
    - Microsoft.Practices.EnterpriseLibrary.Common.dll
    - Microsoft.Practices.ServiceLocation.dll
3. Select the **Security\SecurityHelper.cs** file in the Solution Explorer. Select the **View | Code** menu command.

   Add the following namespace inclusions to the list of namespaces at the top of the file:

```
using Microsoft.Practices.EnterpriseLibrary.Security;
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
```

4. Add the following code to the **Authorized** method.

```
public static bool Authorized(string rule)
{
  bool authorized = false;

  // TODO: Check rule-base authorization
  // No parameter passed to GetInstance method as
  // we'll set the Default Authorization Instance in App.config.
  IAuthorizationProvider ruleProvider =
    EnterpriseLibraryContainer.Current.GetInstance<IAuthorizationProvider>();

  authorized = ruleProvider.Authorize(Thread.CurrentPrincipal, rule);

  return authorized;
}
```

   This code retrieves the default authorization provider, and uses it to test the current principal (roles and identity) against the rule.

5. Select the **Debug | Start Without Debugging** menu command to run the application.

   Sign in as **Tom**, **Dick**, and **Harry**, in turn, to confirm the correct setup of the role provider.

User	Task access
Tom (Employee)	Raise new bug
Dick (Developer)	Raise new bug and resolve bug
Harry (Manager)	Raise new bug, resolve bug, and assign bug

6. Close the application.

To verify that you have completed the exercise correctly, you can use the solution provided in the ex02\end folder.

Copyright