Claims-based Identity & Access Control

Estimated time for completion: 60 minutes

Overview:

In this lab, you will use the Windows Identity Foundation to modify an existing WCF service so that it uses claims-based authorization. Claims will first be created manually and later, claims will be generated from a custom Security Token Service.

Goals:

- Modify a service to use the Windows Identity Foundation
- Implement a custom username/password validation logic via WIF
- Provide custom claims
- Implement claims-based authorization
- Modify the service configuration to use an STS

Part 1 – Enabling WIF in a WCF Service

In the simplest case, turning on Geneva for an existing service requires only a single line of code. However, depending on your needs, further steps may be necessary. In this part of the lab, enabling WIF in the PetShopService requires you to provide an implementation of your user name validation code.

Steps:

1. Make sure you have installed WIF and the SDK.
2. Use the makecert.exe tool to create a service certificate. However, make sure you don't install the certificate more than once. You can inspect the installed certificates by starting the Microsoft Management Console (mmc.exe) and adding the *Certificates* Snap in for the current user. For this lab, the certificate must exist in the trusted people store. Here is the command line to create the service certificate in case you need it.

```
makecert.exe -a sha1 -n CN=PetShopService -sr CurrentUser
       -ss TrustedPeople -sky exchange -sk PetShopService
```

3. Open the starter projects located at before/part1. It contains a basic pet shop service and a client. Take a quick look at the project. Notice that the service implements a simple

© 2007 Education Experiences, Inc. All rights reserved

operation PlaceOrder. Inspect the services configuration file. In the service behavior configuration you can see that the service uses a certificate for service authentication and username/password for client authentication. Also notice that the client validation is done via a custom class called PetShopUserNamePasswordValidator. Take a look at the class to see the trivial validation logic.

4. Start the service and the client to make sure the service can be called successfully. If you have difficulties starting the service or running the client, recheck your service certificate installation.
5. In the service project, add a reference to the Geneva Framework assembly Microsoft.IdentityModel.dll. You can find it the %program files% \Reference Assemblies\Microsoft\Windows Identity Foundation\v3.5 directory.
6. To enable WIF, call FederatedServiceCredentials.ConfigureServiceHost after creating the ServiceHost instance:

```
using (ServiceHost host = new ServiceHost(typeof(PetShop)))
{
    FederatedServiceCredentials.ConfigureServiceHost(host);
    ...
}
```

7. Start the Service and the Client. Notice that this time the client fails, because the client credentials cannot be authenticated any more. The PetShopUserNamePasswordValidator is no longer used to authenticate the client. Modify the client code temporarily so that a valid Windows username and the matching password are use as client credentials. Try again.
8. This time, the client should be able to call the service; however we do not want to authenticate the client with windows accounts here. Change the client to use the old username and password again. Instead of the UserNamePasswordValidator we have to implement another class that provides the validation logic. Add a new class to the project and call it PetShopUserNameSecurityTokenHandler. (Leave the default namespace DM). Mark it as public and derive it from UserNameSecurityTokenHandler, which is defined in the namespace Microsoft.IdentityModel.Tokens.
9. In your token handler, override the property CanValidateToken to return true and the method ValidateToken to perform the validation as follows. First, check if the token passed as an argument is null and throw an ArgumentNullException in that case. After that, use the as operator to cast the token to UserNameSecurityToken. If the cast fails throw an InvalidArgumentException. Perform the same trivial validation logic as in PetShopUserNamePasswordValidator. If the validation fails, throw an InvalidOperationException. If the validation succeeds, return a new ClaimsIdentityCollection containing a new ClaimsIdentity object that has a single claim. Use the claim's value use the user name. For the claim's type use the property Microsoft.IdentityModel.Protocols.WSIdentityConstants.ClaimTypes.Name.

```
public class PetShopUserNameSecurityTokenHandler :
UserNameSecurityTokenHandler
```

© 2007 Education Experiences, Inc. All rights reserved

```
{
    public override bool CanValidateToken
    {
        get
        {
            return true;
        }
    }

    public override ClaimsIdentityCollection ValidateToken(SecurityToken
token)
    {
        if (token == null)
        {
            throw new ArgumentNullException("token");
        }

        UserNameSecurityToken userNameToken = token as UserNameSecurityToken;
        if (userNameToken == null)
        {
            throw new ArgumentException("The security token is not a valid
username security token.", "token");
        }

        string username = userNameToken.UserName;
        string password = userNameToken.Password;

        if (StringComparer.Ordinal.Equals(username, password))
        {
            return new ClaimsIdentityCollection(new IClaimsIdentity[]
                        {
                            new ClaimsIdentity(new Claim[]
                            {
                                 new Claim(
                                    WSIdentityConstants.ClaimTypes.Name,
                                    username)
                            })
                        });
        }

        throw new InvalidOperationException("Wrong username or password.");
    }
}
```

1. Furthermore, you have to register your validation implementation. As we have seen before, there is a validation based on Windows accounts already registered. You have to unregister that one before you can register your own. Simply paste the XML element below as a child element of <configuration> into the Service's app.config file.

© 2007 Education Experiences, Inc. All rights reserved

```
<microsoft.identityModel>
  <service>
    <securityTokenHandlers>
      <remove type=
"Microsoft.IdentityModel.Tokens.WindowsUserNameSecurityTokenHandler,
Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"/>
      <add type=
"DM.PetShopUserNameSecurityTokenHandler, PetShopService"/>
    </securityTokenHandlers>
  </service>
</microsoft.identityModel>
```

2. Since the config section microsoft.identityModel is not registered by default, you have to register it explicitly. To do this, add the following element as the first child element of the root element <configuration>.

```
<configSections>
  <section name="microsoft.identityModel" type=
"Microsoft.IdentityModel.Configuration.MicrosoftIdentityModelSection,
Microsoft.IdentityModel, Version=3.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"/>
</configSections>
```

3. Build and test the service as well as the client.

Solution

The solution for this part of the lab can be found at after/part1.

Part 2 – Adding Claims and Authorization Logic

In this part of the lab we will implement a ClaimsAuthenticationManager that adds a claim to express the customer's experience in points (0: no experience, 100: very experienced). After that we will implement authorization logic. Depending on the ordered animal's required experience level and the customer's experience points, a customer may be allowed to buy certain animals or not. For example, a hamster does not require any experience points, and an Anaconda requires 100 points.

Steps:

1. Continue with the project where you left off in Part 1.
2. To identify the claim type for experience points, add a new static class named PetShopClaimTypes to the PetShopService project. Define a public const string field named `ExperiencePoints` in this class and set its value to `"urn:dm:petshop:claims/experiencepoints"`.

© 2007 Education Experiences, Inc. All rights reserved

3. Add a new class named PetShopClaimsAuthenticationManager to the service project. Mark the class as public and derive it from ClaimsAuthenticationManager (which is defined in the namespace Mircosoft.IdentityModel.Tokens).
4. To simplify the sample, we use a hard coded dictionary of experience points. Simply paste the following property to the ClaimsAuthenticationManager.

```
static Dictionary<string, int> experiencePointsMap =
                      new Dictionary<string, int>()
{
  { "Alice", 50 },
  { "Bob", 100 },
};
```

5. ClaimsAuthenticationManager is a very simple abstract class; it has only one abstract method called Authenticate. Implement this method in your derived class as follows. Throw an ArgumentNullException if the argument incomingPrincipal is null. Throw an ArgumentExeption if the incomingPrincipal does not have exactly one identity. Store the one and only identity in a local variable named `identity` of type IClaimsIdentity. Store the Name property of the identity into a local variable `name`. Define a local integer variable named experiencePoints and initialize it with 0; Call TryGetValue on the experiencePointsMap to get the customer's experience points from the authenticated name. Add a new claim with the claimtype PetShopClaimTypes.ExperiencePoints and the stringified experience points value to the identity and return the incomingPrincipal argument.

```
public class PetShopClaimsAuthenticationManager : ClaimsAuthenticationManager
{
    static Dictionary<string, int> experiencePointsMap =
                    new Dictionary<string, int>()
                    {
                      { "Alice", 50 },
                      { "Bob", 100 },
                    };

    public override IClaimsPrincipal Authenticate(string endpointUri,
IClaimsPrincipal incomingPrincipal)
    {
        if (incomingPrincipal == null)
            throw new ArgumentNullException("incomingPrincipal");

        if (incomingPrincipal.Identities.Count != 1)
            throw new ArgumentException("incomingPrincipal must have exactly
one identity");

        IClaimsIdentity identity = incomingPrincipal.Identities[0];
        string name = identity.Name;

        int experiencePoints = 0;
```

© 2007 Education Experiences, Inc. All rights reserved

```
        experiencePointsMap.TryGetValue(name, out experiencePoints);

        identity.Claims.Add(new Claim(PetShopClaimTypes.ExperiencePoints,
experiencePoints.ToString()));

        return incomingPrincipal;
    }
}
```

6. Furthermore, you have to register your claims authentication manager. Simply paste the XML element below as a child element of <microsoft.identityModel> into the Service's app.config file.

```
<claimsAuthenticationManager
    type="DM.PetShopClaimsAuthenticationManager,PetShopService"/>
```

7. Now that we have the claims added to the solution, we should make authorization decisions based on them. This will be done in the implementation of the service class. First, paste the following helper functions to the service class. Inspect them and notice that they throw appropriate FaultExceptions.

```
void ReportSenderFault(string message, string faultCodeName)
{
  throw new FaultException(
              new FaultReason(new FaultReasonText(message, "en")),
                     FaultCode.CreateSenderFaultCode(faultCodeName,
                                        Constants.SERVICE_NAMESPACE));
}

void ReportReceiverFault(string message, string faultCodeName)
{
  throw new FaultException(
              new FaultReason(new FaultReasonText(message, "en")),
                    FaultCode.CreateReceiverFaultCode(faultCodeName,
                                        Constants.SERVICE_NAMESPACE));
}
```

8. Like the required points, the customers experience points will be determined based on a simple dictionary. Add the following static field to the service class.

```
static Dictionary<string, int> requiredPointsMap =
        new Dictionary<string, int>()
        {
            { "Guinea pig", 0 },
            { "Hamster", 0 },
            { "Boa constrictor", 50 },
            { "Anaconda", 100 },
        };
```

© 2007 Education Experiences, Inc. All rights reserved

9. In PlaceOrder, after the diagnostic dumps to the console, define a local integer variable requiredPoints and call TryGetValue on the requiredPointsMap to determine the required points. If TryGetValue returns false, the requested product is unknown. In this case call ReportSenderFault with a faultCodeName "UnknownProduct" and an appropriate message.
10. Now that the required information is determined, check if the customer has enough experience points to buy the requested product. Use WIF's authorization plumbing by calling ClaimsPrincipalPermission.CheckAccess. CheckAccess requires two parameters – an action and a resource. Choose 'BuyAnimal' as the action, and the required experience points as the resource. CheckAccess will throw a SecurityException when authorization fails – if this is the case call ReportSenderFault with a faultCodeName "InsufficientExperiencePoints" and an appropriate message containing the the required points.
11. WIF routes calls to CheckAccess to a ClaimsAuthorizationManager. The next step is to implement such an authorization manager that models our authorization policy. For this add a new class called PetShopClaimsAuthorizationManager to the service project that derives from ClaimsAuthorizationManager. Implement the CheckAccess method by inspecting the AuthorizationContext that gets passed in. You'll find the action and resource encoded as claims on there. Now look out for an action of 'BuyAnimal' – this resource access translates to a check for the experience point claim. The last step is to query the claims collection (you'll find the principal on the AuthorizationContext) for this required claim and the appropriate value. Then return true or false.

```
public class PetShopClaimsAuthorizationManager : ClaimsAuthorizationManager
{
    public override bool CheckAccess(AuthorizationContext context)
    {
        var action = context.Action.First();
        var resource = context.Resource.First();
        var identity = context.Principal.Identities[0];

        if (action.Value.Equals("BuyAnimal",
StringComparison.OrdinalIgnoreCase))
        {
            var xp = (from claim in identity.Claims
                      where claim.ClaimType ==
PetShopClaimTypes.ExperiencePoints
                      select claim)
                     .First();

            if (int.Parse(resource.Value) > int.Parse(xp.Value))
            {
                return false;
            }
        }

        return base.CheckAccess(context);
```

© 2007 Education Experiences, Inc. All rights reserved

```
    }
}
```

12. Register the authorization manager in the WIF configuration section and test.

```
<claimsAuthorizationManager type="DM.PetShopClaimsAuthorizationManager,
PetShopService"/>
```

Solution

The solution for this part of the lab can be found at after/part2.

Part 3 – Externalizing authentication

The PetShop company has built a number of services and applications in the meantime. They realized that they have to implement the authentication and authorization logic over and over again. To streamline their architecture, the move all authorization and claims generation to a central security token service.

One caveat is that for new applications Petshop Company wants to abandon the experience point system, but rather use classifications like "Beginner", "Intermediate" and "Expert". The STS will emit those claims only. We'll see how the claims abstraction can make this change as painless as possible.

In this part of the lab, you will inspect an implementation of an STS and change your authorization policy and configure your service as well as your client to use this STS.

Steps:

1. Continue with the project where you left off in Part 2.
2. The STS needs a service certificate as well. Create it with from an elevated command prompt with the following command line:

```
makecert.exe -a sha1 -n CN=PetShopSTS -sr CurrentUser
       -ss TrustedPeople -sky exchange -sk PetShopSTS
```

3. For this lab, an STS is already implemented. Simply add the PetShopSTS project to your solution. Build and start the STS. If it fails to start, check the certificate for the STS.
4. The STS is mainly implemented in the two classes PetShopSTSConfiguration and PetShopSTS. PetShopSTS is derived from a framework-provided class SecurityTokenService and contains the custom logic for generating the claims as well as for determining the certificate used to encrypt the issued token. First, take a look at the class PetShopSTSConfiguration; it provides information needed by the STS implementation at runtime. Notice that it is derived from the framework-provided class SecurityTokenServiceConfiguration. In the constructor a few properties of this base class

© 2007 Education Experiences, Inc. All rights reserved

are set; we specify the type of the SecurityTokenService-derived class, as well as the certificate used to sign the issued token.

5. Now take a look at the class PetShopSTS, which overrides two methods of its base class SecurityTokenService. These methods are GetScope and GetOutputClaimsIdentity. Each of these methods is called once to issue a token. First, GetScope is called to determine if a token can be issued and how the issued token should be encrypted. The encryption token depends on the relying party – the service that accepts the token issued by the STS. Real life STS implementations usually must be able to support issuing tokens for more than one relying parties. In our simple case, the STS can issue tokens only for the PetShopService. Therefore, GetScope uses the AppliesTo property of the request parameter to check if the relying party for the current request is the PetShopService. If this is the case, a Scope object with the signing credentials of the STS and the encrypting credentials for the PetShopService is returned.
6. The implementation of GetOutputClaimsIdentity is very similar to the implementation of the ClaimsAuthenticationManager's Authenticate method in part 2 of this lab. It is based on a dictionary object that maps user names to experience points. In contrast to ClaimsAuthenticationManager.Authenticate, this method does not extend the incoming identity, but creates a new one. If the incoming identity was extended, the STS would blindly sign claims it has received from its client.
7. In GetOutputClaimsIdentity you can see, that the experience class claims like "Expert" get issued.
8. To understand how the STS is hosted take a look at the configuration file of the PetShopSTS project. The STS is hosted with the usual configuration elements in system.serviceModel. The service host has a baseaddress and an endpoint. In our case, the endpoint uses WS2007HttpBinding with UserName/Password credentials; however, other bindings could have been used as well. It would even be possible to have two WS2007HttpBinding: one with UserName/Password credentials and one with Windows credentials. This would allow you to get issued tokens in different ways.
9. Now it is time to reconfigure the PetShopService so that the claims come from our STS. First of all, you will not need local claim generation any more. Therefore open the App.config file of the PetShopService project and comment out the registration of the claims authentication manager in <microsoft.identityModel>.
10. Next, you have to change your endpoint. To accept issued tokens from an STS, you have to use WS2007FederationHttpBinding instead of WS2007HttpBinding. Comment out the existing endpoint and add a the following one:

```
<endpoint binding="ws2007FederationHttpBinding"
          bindingConfiguration="PetShopViaWS2007Federation"
          bindingName="PetShopViaWS2007Federation"
          bindingNamespace="urn:dm:petshop:services"
          contract="DM.IPetShop" />
```

© 2007 Education Experiences, Inc. All rights reserved

11. Notice that the endpoint element contains a bindingConfiguration attribute. In the configuration of the ws2007FederationBinding, the communication to the STS can be specified by pointing the binding to the STS metadata exchange endpoint:

```
<ws2007FederationHttpBinding>
  <binding name="PetShopViaWS2007Federation">
    <security mode="Message">
     <message>
       <issuerMetadata address="http://localhost:9000/PetShopSTS/mex"
       />
     </message>
   </security>
 </binding>
</ws2007FederationHttpBinding>
```

12. Furthermore, you have to tell the WIF that you want to accept issued tokens signed with the certificate of the STS. This requires an implementation of an issuer name registry. Add a new class called PetShopIssuerNameRegistry to your project. Mark it as public and derive it from IssuerNameRegistry (which is defined in Microsoft.IdentityModel.Tokens). Override GetIssuerName as follows:

```
public override string GetIssuerName(SecurityToken securityToken)
{
  X509SecurityToken x509Token = securityToken as X509SecurityToken;
  if (x509Token != null)
  {
    if (StringComparer.Ordinal.Equals(
            x509Token.Certificate.SubjectName.Name, "CN=PetShopSTS"))
    {
      return x509Token.Certificate.SubjectName.Name;
    }
  }

  throw new SecurityTokenException("Untrusted issuer.");
}
```

13. This code performs a simple and unsecure validation of the STS certificate. In production code, you have to perform a more secure alternative. This could include checking the thumbprint and chain validation.
14. To register the issuer name registry, put it in the <microsoft.identityModel> element of the config file:

```
<issuerNameRegistry type=
     "DM.PetShopIssuerNameRegistry,PetShopService" />
```

15. Next you have to register the allowed audience URIs in the SAML token that the service will accept (again inside of microsoft.identityModel):

© 2007 Education Experiences, Inc. All rights reserved

```
<audienceUris>
  <add value="http://localhost:9000/Services" />
</audienceUris>
```

16. The last configuration step is special to this lab. Since we don't use proper certificates (in the sense that they don't chain up to trusted roots nor have revocation lists), we have to turn off certificate validation.

```
<certificateValidation certificateValidationMode="PeerOrChainTrust"
                       revocationMode="NoCheck" />
```

17. Build and run your service. It should be able to start now. Keep it running so that you can update the service reference in the client project. To avoid problems with multiple endpoint configurations on the client side delete the client's app.config before you update the service reference.
18. Given that the certificates for the PetShopService and the STS are not signed by trusted CA's the client has to switch to PeerTrust validation for the certificates. This requires an endpoint behavior:

```
<endpointBehaviors>
  <behavior name="PetShopServiceViaWS2007Federation">
    <clientCredentials>
      <serviceCertificate>
        <authentication certificateValidationMode="PeerTrust"/>
      </serviceCertificate>
    </clientCredentials>
  </behavior>
</endpointBehaviors>
```

19. To apply this behavior, add the behaviorConfiguration attribute to the endpoint:

```
<endpoint address="http://localhost:9000/Services"
          binding="ws2007FederationHttpBinding"
          bindingConfiguration="PetShop"
          contract="PetShopSvcClient.PetShopService"
          name="PetShop"
          behaviorConfiguration="PetShopServiceViaWS2007Federation">
  <identity>…</identity>
</endpoint>
```

20. Make sure the STS and the PetShopService are running and start the client. You should now be able to call the service.
21. Since the new STS does not emit the needed claims directly – we have to find a way to update the service. Since claims and WIF provide several levels of abstraction there are two options how to adapt to the new environment without having to update all the service code:

© 2007 Education Experiences, Inc. All rights reserved

a. Use a ClaimsAuthenticationManager to transform the incoming experience class claims to experience points claims.

b. Modify the ClaimsAuthorizationManager to map the experience points to the experience classes.

22. We'll use option b here – feel free to implement option a on your own. You simply have to exchange the existing authorization manager with this new one and re-run the client.

```
public class PetShopClaimsAuthorizationManager : ClaimsAuthorizationManager
{
    public override bool CheckAccess(AuthorizationContext context)
    {
        var action = context.Action.First();
        var resource = context.Resource.First();
        var identity = context.Principal.Identities[0];

        if (action.Value.Equals("BuyAnimal",
StringComparison.OrdinalIgnoreCase))
        {
            var xp = (from claim in identity.Claims
                      where claim.ClaimType ==
PetShopClaimTypes.ExperienceClass
                      select claim.Value)
                     .First();

            if (xp == "Intermediate")
            {
                return int.Parse(resource.Value) <= 50;
            }
            else if (xp == "Expert")
            {
                return int.Parse(resource.Value) <= 100;
            }
            else
            {
                return int.Parse(resource.Value) <= 10;
            }
        }

        return base.CheckAccess(context);
    }
}
```

Solution

The solution for this part of the lab can be found at after/part3.

© 2007 Education Experiences, Inc. All rights reserved