# **一、创建自定义AuthorizationPolicy**

我们先来演示通过自定义AuthorizationPolicy以提供当前安全主体的方式。我们通过自定义AuthorizationPolicy实现这样的授权策略：如果用户名为Foo（假设为管理员），我们创建一个包含“Administrators”角色的安全主体；而对于其他的用户，提供的安全主体的角色列表中仅仅包括“Guest”。我们为该自定义AuthorizationPolicy起名为SimpleAdministrators，SimpleAdministrators整个定义如下。

1: public class SimpleAuthorizationPolicy : IAuthorizationPolicy

2: {

3: public SimpleAuthorizationPolicy()

4: {

5: this.Id = Guid.NewGuid().ToString();

6: }

7: public bool Evaluate(EvaluationContext evaluationContext, ref object state)

8: {

9: string userName = string.Empty;

10: foreach (ClaimSet claimSet in evaluationContext.ClaimSets)

11: {

12: foreach (Claim claim in claimSet.FindClaims(ClaimTypes.Name, Rights.PossessProperty))

13: {

14: userName = (string)claim.Resource;

15: }

16: }

17:

18: if (userName.Contains('\\'))

19: {

20: userName = userName.Split('\\')[1];

21: }

22: evaluationContext.Properties["Principal"] = GetPrincipal(userName);

23: return false;

24: }

25:

26: private IPrincipal GetPrincipal(string userName)

27: {

28: GenericIdentity identity = new GenericIdentity(userName);

29: if (string.Compare("Foo", userName, true) == 0)

30: {

31: return new GenericPrincipal(identity, new string[] { "Administrators" });

32: }

33: return new GenericPrincipal(identity, new string[] {"Guest" });

34: }

35:

36: public ClaimSet Issuer

37: {

38: get { return ClaimSet.System; }

39: }

40: public string Id { get; private set; }

41: }

这个安全主体的提供实现在Evaluate方法中，而其中唯一值得一提的是当前认证用户名的获取。在客户端被成功认证之后，被认证的用户实际上也通过某个声明（Claim）保存下来。该声明的类型为“http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name”，可以通过ClaimTypes的静态属性Name得到。而该Claim对象的Resource就是用户名。在得到当前认证用户名之后，相应的GenericPrincipal对象被创建出来，并被置于EvaluationContext的属性列表中。并且该属性对应的Key为“Principal”。

# **二、创建自定义ServiceAuthorizationManager**

接下来我们来通过自定义ServiceAuthorizationManager来实现与上面完全一样的功能，而已授权策略很简单，我们照例将该自定义ServiceAuthorizationManager起名为SimpleServiceAuthorizationManager。以下是SimpleServiceAuthorizationManager的定义。

1: public class SimpleServiceAuthorizationManager : ServiceAuthorizationManager

2: {

3: protected override bool CheckAccessCore(OperationContext operationContext)

4: {

5: string userName = operationContext.ServiceSecurityContext.PrimaryIdentity.Name;

6: if (userName.Contains('\\'))

7: {

8: userName = userName.Split('\\')[1];

9: }

10: operationContext.ServiceSecurityContext.AuthorizationContext.Properties["Principal"] = GetPrincipal(userName);

11: return true;

12: }

13: private IPrincipal GetPrincipal(string userName)

14: {

15: GenericIdentity identity = new GenericIdentity(userName);

16: if (string.Compare("Foo", userName, true) == 0)

17: {

18: return new GenericPrincipal(identity, new string[] { "Administrators"});

19: }

20: return new GenericPrincipal(identity, new string[] { "Guest" });

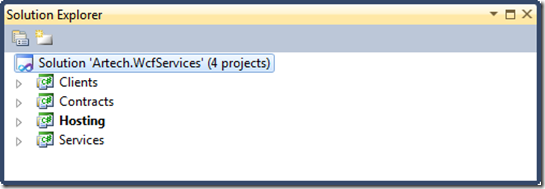
21: }

22: }

和自定义AuthorizationPolicy不同的是，认证用户的获取在这里变得更加容易，我们直接可以通过当前ServiceSecurityContext的PrimaryIdentity获取。需要提醒一下的是，如果你在自定义AuthorizationPolicy的Evaluate方法中调用该属性，会出现一个[StackOverflowException](http://msdn2.microsoft.com/xcewth9z.aspx" \o "StackOverflowException Class)异常，因为该属性的调用本身又会触发Evaluate方法的调用。最后被创建的GnericPrincipal被保存在当前AuthorizationContext的属性列表中，属性的Key依然是“Principal”。

# **三、通过自定义AuthorizationPolicy实现授权**

现在我们常见一个实例程序来应用我们创建的自定义AuthorizationPolicy，看看它是否能够起到我们期望的授权的作用。我们依然沿用我们再熟悉不过的计算服务的例子，解决方案依然按照如下图所示的结构来设计。整个解决方式包括四个项目：Contracts、Services、Hosting和Client。对于这样的结构我们已经了解得够多了，在这里没有必要再赘言叙述了。



在实例解决方案的整个结构建立之后，我们分别在Contracts和Services项目中定义服务契约接口和服务类型。下面是契约接口ICalculator和服务CalculatorService的定义。而在CalculatorService类的Add方法中应用了PrincipalPermissionAttribute特性，并将Roles属性设置成了Adminstrators，意味着该服务操作只能被管理员用户组中的用户调用。

1: [ServiceContract(Namespace = "http://www.artech.com/")]

2: public interface ICalculator

3: {

4: [OperationContract]

5: double Add(double x, double y);

6: }

7:

8: public class CalculatorService : ICalculator

9: {

10: [PrincipalPermission(SecurityAction.Demand, Role = "Administrators")]

11: public double Add(double x, double y)

12: {

13: return x + y;

14: }

15: }

现在通过Hosting这个控制台程序对上面创建的服务进行寄宿。下面给出的是整个寄宿程序的配置，从中我们可以看出：应用到CalculatorService的服务行为列表中包含了PrincipalPermissionMode为Custom的ServiceAuthorizationBehavior。而我们定义的SimpleAuthorizationPolicy类型被配置到了<authorizationPolicies>列表中。

1: <?xml version="1.0"?>

2: <configuration>

3: <system.serviceModel>

4: <services>

5: <service name="Artech.WcfServices.Services.CalculatorService" behaviorConfiguration="useCustomAuthorization">

6: <endpoint address="http://127.0.0.1/calculatorservice" binding="ws2007HttpBinding" contract="Artech.WcfServices.Contracts.ICalculator"/>

7: </service>

8: </services>

9: <behaviors>

10: <serviceBehaviors>

11: <behavior name="useCustomAuthorization">

12: <serviceAuthorization principalPermissionMode="Custom" >

13: <authorizationPolicies >

14: <add policyType="Artech.WcfServices.Hosting.SimpleAuthorizationPolicy, Artech.WcfServices.Hosting" />

15: </authorizationPolicies>

16: </serviceAuthorization>

17: <serviceDebug includeExceptionDetailInFaults="true"/>

18: </behavior>

19: </serviceBehaviors>

20: </behaviors>

21: </system.serviceModel>

22: </configuration>

由于我们使用了WSHttpBinding，而它在默认的情况下采用Windows客户端凭证，为此我们需要创建两个Windows帐号Foo和Bar，密码被设定为Password。在如下所示的客户端代码中，我们分别以Foo和Bar的名义调用了服务。最后将服务能够成功调用的结果打印出来。

1: class Program

2: {

3: static void Main(string[] args)

4: {

5: ChannelFactory<ICalculator> channelFactory = new ChannelFactory<ICalculator>("calculatorService");

6: NetworkCredential credential = channelFactory.Credentials.Windows.ClientCredential;

7: credential.UserName = "Foo";

8: credential.Password = "Password";

9: ICalculator calculator = channelFactory.CreateChannel();

10: Invoke(calculator);

11:

12: channelFactory = new ChannelFactory<ICalculator>("calculatorService");

13: credential = channelFactory.Credentials.Windows.ClientCredential;

14: credential.UserName = "Bar";

15: credential.Password = "Password";

16: calculator = channelFactory.CreateChannel();

17: Invoke(calculator);

18:

19: Console.Read();

20: }

21: static void Invoke(ICalculator calculator)

22: {

23: try

24: {

25: calculator.Add(1, 2);

26: Console.WriteLine("服务调用成功...");

27: }

28: catch (Exception ex)

29: {

30: Console.WriteLine("服务调用失败...");

31: }

32: }

33: }

从下面的结果来看，只有在用户名为Foo才能成功调用服务，而Bar由于权限不足会导致服务调用失败。这充分证明了通过自定义AuthorizationPolicy能够正确地起到授权的作用。

1: 服务调用成功...

2: 服务调用失败...

# **四、通过自定义ServiceAuthorizationManager实现授权**

在证明我们自定义的AuthorizationPolicy确实能够按照我们定义的策略进行授权之后，我们来试试我们自定义的ServiceAuthorizationManager能否同样完成授权的使命。为此我们唯一需要做的就是改变一下服务寄宿程序的配置。

1: <?xml version="1.0"?>

2: <configuration>

3: <system.serviceModel>

4: <services>

5: <service name="Artech.WcfServices.Services.CalculatorService" behaviorConfiguration="useCustomAuthorization">

6: <endpoint address="http://127.0.0.1/calculatorservice" binding="ws2007HttpBinding"

7: contract="Artech.WcfServices.Contracts.ICalculator"/>

8: </service>

9: </services>

10: <behaviors>

11: <serviceBehaviors>

12: <behavior name="useCustomAuthorization">

13: <serviceAuthorization principalPermissionMode="Custom"

14: serviceAuthorizationManagerType="Artech.WcfServices.Hosting.SimpleServiceAuthorizationManager,

15: Artech.WcfServices.Hosting" >

16: <!--<authorizationPolicies >

17: <add policyType="Artech.WcfServices.Hosting.SimpleAuthorizationPolicy, Artech.WcfServices.Hosting" />

18: </authorizationPolicies>-->

19: </serviceAuthorization>

20: <serviceDebug includeExceptionDetailInFaults="true"/>

21: </behavior>

22: </serviceBehaviors>

23: </behaviors>

24: </system.serviceModel>

25: </configuration>

上面所示的采用自定义ServiceAuthorizationManager实现授权的配置。我们将之前添加的AuthorizationPolicy注释掉，然后通过ServiceAuthorizationBehavior配置节的serviceAuthorizationManagerType属性设置成我们自定义的SimpleServiceAuthorizationManager的类型。运行程序后，你会得到和上面一样的输出结果。