***Windows Communication Foundation (WCF) is a secure, reliable, and scalable messaging platform for the .NET Framework 3.0. With WCF, SOAP messages can be transmitted over a variety of supported protocols including IPC (named pipes), TCP, HTTP and MSMQ. Like any distributed messaging platform, you must establish security policies for protecting messages and for authenticating and authorizing calls. This article will discuss how WCF accomplishes this.***

A consistent set of fundamental security concepts apply in any distributed messaging system. Consider a message from sender (the calling application) to receiver (the target service receiving the message for processing):

* **Authentication**. We typically think about authentication as identifying the message sender. Mutual authentication involves authenticating both the sender and the message receiver, to prevent possible man-in-the-middle attacks.
* **Authorization**. After authenticating the message sender, authorization determines what system features and functionality they are entitled to execute.
* **Integrity**. Messages should be digitally signed to ensure they have not been altered between sender and receiver.
* **Confidentiality**. Sensitive messages or specific message parts should be encrypted to ensure they cannot be openly viewed on the wire.

WCF provides a rich and configurable environment for creating security policies and setting runtime behaviors to control these security features. A variety of mutual authentication mechanisms are supported using token formats such as Windows tokens, username and password, certificates and issued tokens (in a federated environment). Authorization can be based on Windows roles, ASP.NET roles or you can provide custom authorization policies. Message protection (integrity and confidentiality) can be based on symmetric session keys, or asymmetric keys for single-hop protection.

|  |  |  |
| --- | --- | --- |
| **"** | **“A consistent set of fundamental security concepts apply in any distributed messaging system.”** | **"** |

In the following sections, I’ll show you how to configure WCF security and then take you through some common WCF deployment scenarios and their specific security configurations that employ these fundamental security concepts.

**Security, WCF Style**

The first step to securing a WCF service is defining the security policy. Once you have established requirements for authentication, authorization, and message protection it is a matter of service configuration to enforce it.

Your binding selection will influence the available configuration options for the service security policy. When you expose a service endpoint you select a binding that represents the appropriate communication protocol and message encoding format. For example, for intranet communications or systems behind the firewall, TCP protocol with binary message encoding is usually preferred. For Internet access, HTTP protocol is a typical choice using text or MTOM encoding (depending on the message size).

There are a standard set of bindings that can satisfy these protocol and encoding choices. **NetTcpBinding** is the right choice for binary TCP communications that cross machine boundaries, **BasicHttpBinding** is the right choice for HTTP communications that must support legacy Web service protocols, and **WSHttpBinding** or **WSFederationHttpBinding** are the right choice for Web services that can leverage a richer set of standards including those for secure communications (the latter is used for federated security scenarios).

Beyond bindings, behaviors also provide information about client and service credentials, and affect how authorization is handled.

You can configure bindings and behaviors declaratively or through the runtime object model-but in the following sections I’ll focus on how you declaratively configure core security settings.

***Default Security Settings***

Each binding has a default set of security settings. Consider the following service endpoint that supports **NetTcpBinding**.

<system.serviceModel>  
  <services>  
    <service   
name="HelloIndigo.HelloIndigoService" >  
      <endpoint   
contract="HelloIndigo.IHelloIndigoService"   
binding="netTcpBinding" />  
    </service>  
  </services>  
</system.serviceModel>

**NetTcpBinding** is secure by default. Specifically, callers must provide Windows credentials for authentication and all message packets are signed and encrypted over TCP protocol. Look at the expanded binding configuration illustrating these default settings.

<netTcpBinding>  
  <binding name="netTcp">  
    <security mode="Transport">  
      <transport clientCredentialType="Windows" />  
    </security>  
  </binding>  
</netTcpBinding>

When the security *mode* is set to message security, you can customize the default security settings for **NetTcpBinding** by configuring different values for *clientCredentialType* or *algorithm suite*. Other bindings such as **WSHttpBinding** also allow you to determine if a secure session will be established and control how service credentials are negotiated. Each of the standard WCF bindings supports only relevant security options for their typical usage. In the next sections, I’ll review some of the security-specific binding options available, and how you configure them.

***Security Mode***

Across all service bindings there are five possible security modes:

* **None**. Turns security off.
* **Transport**. Uses transport security for mutual authentication and message protection.
* **Message**. Uses message security for mutual authentication and message protection.
* **Both**. Allows you to supply settings for transport and message-level security (only MSMQ supports this).
* **TransportWithMessageCredential**. Credentials are passed with the message and message protection and server authentication are provided by the transport layer.
* **TransportCredentialOnly.** Client credentials are passed with the transport layer and no message protection is applied.

You can turn off security completely, or allocate authentication and message protection between transport and message-level security. Each transport protocol (TCP, IPC, MSMQ, or HTTP) has their own mechanism for passing credentials and handling message protection. Message security supports passing credentials as part of the SOAP message using interoperable standards, and also makes it possible to protect the message independent of transport all the way through to the ultimate message receiver. Transport message protection is only good from point to point.

|  |  |  |
| --- | --- | --- |
| **"** | **“NetTcpBinding is secure by default. Specifically, callers must provide Windows credentials for authentication and all message packets are signed and encrypted over TCP protocol.”** | **"** |

To control the security mode used for a particular binding, set the *mode* property for the binding’s **<security>** section. For *Transport* or modes that use transport security, the **<transport>** section should be expanded. You can see this illustrated in the **<netTcpBinding>** section shown earlier. For *Message* mode, settings are supplied in the expanded **<message>** section. For example, this **<wsHttpBinding>** snippet illustrates how to require *UserName* credentials be passed with the message.

<wsHttpBinding>  
  <binding name="wsHttp">  
    <security mode="Message">  
      <message clientCredentialType="UserName" />  
    </security>  
  </binding>  
</wsHttpBinding>

**NOTE:** Not all bindings support all security modes.

***Client Credential Type***

The choice of client credential type depends on the security mode in place. For transport security you can require a Windows credential or certificate-and there are variations on how Windows credentials are passed via TCP, HTTP and MSMQ. Message security supports any of the following settings for *clientCredentialType*:

* None
* Windows
* UserName
* Certificate
* IssuedToken

**BasicHttpBinding** only supports *UserName* and *Certificate* credentials since it is intended to be interoperable with *Basic Security Profile* per WS-I.

This code snippet illustrates how to select a *clientCredentialType* for message security.

<basicHttpBinding>  
  <binding name="basicHttp">  
    <security   
mode="TransportWithMessageCredential">  
      <message   
clientCredentialType="Certificate"/>  
    </security>  
  </binding>  
</basicHttpBinding>

Your choice of credential type may affect other configuration settings for the service. For example, *UserName* credentials require either transport message protection or a service certificate be used to protect the exchange.

***Protection Level***

By default, all secure WCF bindings will encrypt and sign messages. You cannot disable this for transport security, however, for message security you may wish to disable this for debugging purposes, or when an alternate method of protection is used such as IPSec.

Protection-level settings are controlled by the contract. You can specify protection level for all operations in the service contract using the **ServiceContractAttribute**. The following example illustrates disabling encryption.

[ServiceContract(Name="HelloIndigoContract",   
Namespace=  
"<http://www.thatindigogirl.com/2006/06/Samples>",   
ProtectionLevel=ProtectionLevel.Sign)]  
public interface IHelloIndigoService  
{  
  string HelloIndigo(string inputString);  
}

For more granular control, you can also indicate message protection per operation using the **OperationContractAttribute**.

[ServiceContract(Name="HelloIndigoContract",   
Namespace=  
<http://www.thatindigogirl.com/2006/06/Samples>]  
public interface IHelloIndigoService  
{  
  [OperationContract(ProtectionLevel=  
  ProtectionLevel.Sign)]  
  string HelloIndigo(string inputString);  
}

You can also control protection level on message contracts providing granular control over specific headers or body elements.

ProtectionLevel options are: *None, Sign,* and *EncryptAndSign*. *None* disables message protection. *EncryptAndSign* provides full message protection and is the default behavior. *Sign* indicates the message should be signed but not encrypted.

The ProtectionLevel property serves a dual role. First, it specifies the minimum requirements for message protection. If the property is not specified, the default behavior is to encrypt and sign, but not enforce those settings on the binding. By specifying a value for the property using one of the attributes I mentioned, messages that don’t meet the requirement will be rejected for not satisfying the security policy. The second role is to control how message-level protection is applied (since it has no direct affect on transport protection).

***Algorithm Suite***

Choice of algorithm suite can be particularly important for interoperability. Each binding uses *Basic256* as the default algorithm suite for message-level security. This suite defines the algorithms and key lengths for cryptographic operations like key signatures, encryption, and key wrap. Algorithm suites are described in the **WS-SecurityPolicy** specification, and can be applied by setting the *algorithm* attribute of the **<message>** section.

<wsHttpBinding>  
  <binding name="wsHttp">  
    <security mode="Message">  
      <message clientCredentialType="UserName"  
algorithmSuite="TripleDes"  />  
    </security>  
  </binding>  
</wsHttpBinding>

***Service Credentials and Negotiation***

To support mutual authentication and message protection, services must provide credentials to the caller. When transport security is used, service credentials are negotiated through the transport protocol. Service credentials for message security can also be negotiated when Windows credentials are used, otherwise a service certificate must be specified in the **<behaviors>** section under **<serviceCredentials>.**

<behaviors>  
  <serviceBehaviors>  
    <behavior name="serviceBehavior" >  
      <serviceCredentials>  
        <serviceCertificate findValue="RPKey"   
storeLocation="LocalMachine" storeName="My"   
x509FindType="FindBySubjectName" />  
      </serviceCredentials>  
    </behavior>  
  </serviceBehaviors>  
</behaviors>

In this case, the caller must have access to the public key portion of the service certificate to encrypt messages sent to the service. This can be specified out of band, or negotiated with an initial handshake.

The default behavior for message security supports negotiation. That means that the service is dynamically asked for the correct token before any messages are exchanged. For Windows client credentials **SPNego** protocol is used, and for UserName, Certificate or Anonymous credentials, **TLSNego** protocol is used. Today these are not interoperable protocols, so it may be desirable to disable this negotiation.

You can set *negotiateServiceCredential* to false in the **<message>** section to accomplish this.

<wsHttpBinding>  
  <binding name="wsHttp">  
    <security mode="Message">  
      <message clientCredentialType="UserName"  
negotiateServiceCredential="false"   />  
    </security>  
  </binding>  
</wsHttpBinding>

When negotiation is disabled for Windows client credentials, a Kerberos domain must exist. For other credential types the client must have access to the service public key to encrypt messages.

When you generate a service proxy with configuration settings for the client (using svcutil.exe) an encoded version of the public certificate is supplied in the **<identity>** section to handle this case.

<client>  
  <endpoint   
address="http://localhost:8000/HelloIndigo"   
binding="wsHttpBinding"  
bindingConfiguration="wsHttp"   
contract="Client.localhost.HelloIndigoContract"  
name="WSHttpBinding\_HelloIndigoContract">  
    <identity>  
      <certificate encodedValue="   
AwAAAAEAAAAUAAAAreiGqilku9hngWEQL1g+  
…  
oBd0vDwZaqjy47g0jFV9pF0VHhoVbTtOA=="/>  
    </identity>  
  </endpoint>  
</client>

It is also possible to install the public key of the service in the client certificate store and retrieve it from there at run time.

***Secure Session***

Another feature of message security is the ability to establish a secure session to reduce the overhead of one-off key exchange and validation. By default, secure sessions are enabled for message security. A security context token (SCT) is generated through an initial exchange between caller and service. This token is used to authorize and secure subsequent message exchanges.

If the caller plans to make several calls to a service, secure sessions are more efficient. For a single call, however, you can disable this feature by setting *establishSecurityContext* to false.

## <wsHttpBinding>   <binding name="wsHttp">     <security mode="Message">       <message clientCredentialType="UserName"   establishSecurityContext="false" />     </security>   </binding> </wsHttpBinding>

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Authentication, Authorization, and Identities**  From the discussion so far, you should gather that messages are secured according to service security policy. Mutual authentication is performed based on the supplied client and service credentials. Message protection is applied according to transport or message security configuration-which normally means that messages are both signed and encrypted. Token authentication, runtime identities, security principals, and authorization policies also play an important role in the WCF security story.   |  |  |  | | --- | --- | --- | | **"** | **“Token authentication, runtime identities, security principals and authorization policies also play an important role in the WCF security story.”** | **"** |   Access to resources during a service operation is influenced by three key elements:   * **Process Identity**. Service operations are executed under the process identity of the service host. For ASP.NET hosts this is usually the ASP.NET account, and for self-hosting it may be a different service account. This process identity is the Windows account that governs what the service code can do at run time when attempting to access protected resources such as the database, registry or file system. * **Security Principal**. If you are familiar with traditional .NET role-based security, you know that there is a security principal attached to each executing thread. That security principal holds the caller’s identity, which may or may not be tied to a Windows account and its roles. Roles govern which operations can be executed by the authenticated user when traditional .NET role-based security is applied. * **ServiceSecurityContext**. Provides run time access to other relevant information about the security context for a service operation. The ServiceSecurityContext is a run time type that includes identities, a set of claims, and authorization policies. This information can be used to apply a more fine-grained security strategy specifically for services.   **Figure 1** illustrates the relationship of these security elements. While the process identity is constant, each operation is executed on a request thread that contains a unique security principal and security context. In the following sections I’ll elaborate on the authorization process and the role of these and other security elements.  [Click for a larger version of this image.](http://www.code-magazine.com/ShowLargeArticleImage.aspx?QuickID=0611051&Image=Fig001.TIF)  **Figure 1:** Operations are executed within a security context that includes a set of claims and an identity. The identity of the security principal attached to the executing thread is equivalent to the primary identity of the security context.  ***Claims-Based Identity Model***  The identity model in WCF supports a rich, claims-based approach to authorization. A *claim* describes an individual right or action applicable to a particular resource. For example, an identity claim could represent a Windows token, a non-Windows user account, an X509 certificate or some other identity type. Claims can also be proof of possession of other information such as an e-mail address, birth date, or first and last name. Custom claims can be created to indicate the ability to access specific business entities or their storage location.  Claims are generated from security tokens. *Security tokens* are abstractions of credentials that are passed in the security headers of a message and validated against the security policy. When security tokens are validated and processed at the service, claims are extracted and placed into the security context for the operation being executed. Each credential type will result in a different set of claims (called a *claimset*) to evaluate-but you could unify the claimset by mapping the default claims to custom claims that will be evaluated by your code at run time.  Ultimately, a claimset is attached to the **ServiceSecurityContext** and available for any custom authorization code involved in the execution of the operation. Although traditional role-based security is still supported, a customized claims-based authorization model can add a welcome layer of granularity.  ***ServiceSecurityContext***  ServiceSecurityContext provides access to claims at run time. Some key properties of the ServiceSecurityContext type are:  **AuthorizationContext**. Contains one or more claimsets for authorization. Use this information to perform custom authorization.  **AuthorizationPolicies**. Contains the policies used to grant claims.  **PrimaryIdentity.** Contains the identity claim from the claim set, as a traditional IIdentity reference.  **WindowsIdentity**. Contains the identity claim from the claim set if it is a WindowsIdentity.  You can access the ServiceSecurityContext through the OperationContext.  ServiceSecurityContext security =  OperationContext.Current.ServiceSecurityContext;  With that reference, you could implement a custom authorization check that is based on claims. For example, you could check to see that the user was authenticated and that an e-mail claim was also provided.  string user = security.PrimaryIdentity.Name; string email = null;  IEnumerable<Claim> claims = security. AuthorizationContext.ClaimSets[0].FindClaims (ClaimTypes.Email,Rights.PossessProperty);  foreach (Claim c in claims) {   email = c.Resource as string;  } if (string.IsNullOrEmpty(user) || email == null)   throw new SecurityException   ("Unauthorized access. Email claim not found.");  This code illustrates using the ServiceSecurityContext for custom authorization inside an operation-but it can also be wrapped into a custom authorization policy to decouple from operation code and provide a unit of reuse.  The ServiceSecurityContext is also used during role-based authorization, to be discussed shortly.  ***Security Token Authentication***  Claims are added to the security context while tokens are authenticated. One or more security tokens can be present in a message. Each token is authenticated by its own SecurityTokenAuthenticator type. **Table 1** provides a list of commonly used token authenticators with a short description.  Windows tokens are authenticated against the Windows domain. Certificate credentials are authenticated against the certificate store based on authentication rules specified in the **<clientCertificate>** section of the service behavior. For example, the following configuration does not map certificates to Windows accounts, and will trust certificates placed in the TrustedPeople store-otherwise it will try to validate the chain of trust online.  <behavior name="serviceBehavior">   <serviceCredentials>     <clientCertificate>       <authentication   certificateValidationMode="PeerOrChainTrust"  trustedStoreLocation="LocalMachine"  revocationMode="Online"  mapClientCertificateToWindowsAccount="false" />     </clientCertificate>   </serviceCredentials> </behavior>  By default, UserName tokens authenticate against the Windows domain, but you can use the membership provider model of ASP.NET by changing the *userNamePasswordValidationMode* from “Windows” to “MembershipProvider” (see **Listing 1**). This setting engages the default ASP.NET provider model unless you specify an alternate membership provider, or a custom password validator in the **<userNameAuthentication>** section.  SAML tokens are not authenticated against a provider model, but the token itself is validated and its claims presented for authorization.  Clearly, token authentication is specific to the token type, and each has a specific set of behavioral settings appropriate to the token that control how authentication is carried out. The end result for successful authentication is a claimset that can later be used for authorization, and an identity attached to the security context and the thread. | & |  | |  | | --- | |  | |
| |  | | --- | |  | | **Table 1: A subset of commonly used SecurityTokenAuthenticator types for authenticating security tokens.**   |  |  | | --- | --- | | ***SecurityTokenAuthenticator Type*** | ***Description*** | | WindowsSecurityTokenAuthenticator | Ensures a valid Windows token. Generates a WindowsClaimSet for authorization. | | KerberosSecurityTokenAuthenticator | Ensures a valid Kerberos token. Generates a WindowsClaimsSet for authorization. | | X509SecurityTokenAuthenticator | Validates the certificate and maps it to a Windows identity if applicable. Generates an X509ClaimSet and possibly a WindowsClaimSet. | | WindowsUserNameSecurityTokenAuthenticator | Creates a Windows token for the username and password provided. Generates a WindowsClaimSet for authorization. | | CustomUserNameSecurityTokenAuthenticator | Validates the username and password against the configured membership provider or password validator. Generates a UserNameClaimSet for authorization. | | SamlSecurityTokenAuthenticator | Validates the SAML token (timestamp, signature, etc.). Resolves claims directly from the token for authorization. | |  |  | | --- | |  | | [[http://www.code-magazine.com/Images/CodeSnippet.jpg](http://www.code-magazine.com/CodeListing.aspx?QuickID=0611051&id=0)](http://www.code-magazine.com/CodeListing.aspx?QuickID=0611051&id=0)**Listing 1: Possible service model configuration to support both BasicHttpBinding and WSHttpBinding endpoints for the same service** | | <configuration>   <connectionStrings>     <remove name="LocalSqlServer"/>     <add name="LocalSqlServer" connectionString="data  source=localhost;Initial Catalog=aspnetdb; Integrated Security=True; " providerName="System.Data.SqlClient"/>   </connectionStrings>   <system.web>     <roleManager enabled="true"/>     …    </system.web>   <system.serviceModel>     <services>       <service name="HelloIndigo.HelloIndigoService"  behaviorConfiguration="serviceBehavior">         <endpoint contract="HelloIndigo.IHelloIndigoService"  binding="basicHttpBinding" bindingConfiguration="basicHttp"/>         <endpoint contract="HelloIndigo.IHelloIndigoService"  binding="wsHttpBinding" bindingConfiguration="wsHttp"/>       </service>     </services>     <bindings>       <wsHttpBinding>         <binding name="wsHttp">           <security mode="Message">             <message clientCredentialType="UserName"  negotiateServiceCredential="false"  establishSecurityContext="false"/>           </security>         </binding>       </wsHttpBinding>       <basicHttpBinding>         <binding name="basicHttp">           <security mode="TransportWithMessageCredential">             <transport/>             <message clientCredentialType="UserName"/>           </security>         </binding>       </basicHttpBinding>     </bindings>     <behaviors>       <serviceBehaviors>         <behavior name="serviceBehavior">           <serviceMetadata httpGetEnabled="true"/>           <serviceAuthorization  principalPermissionMode="UseAspNetRoles"/>           <serviceCredentials>             <userNameAuthentication  userNamePasswordValidationMode="MembershipProvider"/>             <serviceCertificate findValue="RPKey"  storeLocation="LocalMachine" storeName="My"  x509FindType="FindBySubjectName"/>           </serviceCredentials>         </behavior>       </serviceBehaviors>     </behaviors>   </system.serviceModel> </configuration> | | | | |

***Role-Based Authorization***

Despite the importance and granularity of the claims-based authorization model in WCF, role-based security is still alive and well, and is useful for controlling access to service operations and business classes used downstream. This type of authorization is based on the security principal for the request.

|  |  |  |
| --- | --- | --- |
| **"** | **“The identity model in WCF supports a rich, claims-based approach to authorization.”** | **"** |

The identity of the caller is attached to the executing request thread in the form of a security principal, accessible through the **CurrentPrincipal** property.

System.Threading.Thread.CurrentPrincipal

The security principal is a wrapper for an identity-its type directly related to the token type received. For example, it could be a **WindowsIdentity**, **X509Identity**, **GenericIdentity**, or a custom type that implements **System.Security.Principal.IIdentity**. The identity is created during authentication as I discussed.

The actual security principal is a type that implements **System.Security.Principal.IPrincipal**. This interface has two members:

* A read-only **Identity** property that returns a reference to the IIdentity for the request.
* An **IsInRole()** method that returns a true or false result after checking to see if the identity is in a particular role.

The choice of role provider for a request influences the type of security principal attached to the thread. Options for role provider include:

**None**. No role provider.

**Windows**. Use Windows roles and add a WindowsPrincipal to the security context.

**UseAspNetProvider**. Use the configured RoleProvider type, which defaults to the ASP.NET role provider. This adds a RoleProviderPrincipal to the security context.

**Custom**. Relies on a custom authorization policy to add a security principal to the security context.

The default role provider is “Windows” therefore a **WindowsPrincipal** is the default type. For Windows, UserName or Certificate credentials (when Certificates are mapped to Windows accounts) this will contain an authenticated WindowsPrincipal, otherwise the principal is unauthenticated and has no runtime use for role-based security.

If you aren’t expecting Windows credentials, you can change the role provider by setting the *principalPermissionMode* value of the **<serviceAuthorization>** behavior (**Listing 1**). If you are using the ASP.NET credentials database, you can set it to “UseAspNetProvider”. This causes a **RoleProviderPrincipal** to be attached to the thread instead of a WindowsPrincipal. This IPrincipal type is new to WCF, and holds a reference to the ServiceSecurityContext. When the identity is requested from the principal, it actually returns a reference to the ServiceSecurityContext’s **PrimaryIdentity** property (discussed earlier). When IsInRole() is invoked, it uses the configured **RoleProvider** (in this case, the default ASP.NET role provider) to check if this identity is in the specified role.

You can also customize this behavior with a custom ASP.NET RoleProvider or with a custom authorization policy.

In any case, .NET role-based security relies on the IPrincipal object attached to the thread to perform authorization checks. So, even with WCF you can use the **PrincipalPermission** type to demand things like:

* Is the user authenticated?
* Is the user in a particular role?
* Is a particular user calling?

At runtime, this can be done with an imperative permission demand within the WCF operation or any business component. Just create a PrincipalPermission object, initialize the values you want to enforce, and issue the Demand().

public string AdminsOnly()  
{  
  // unprotected code  
  
  PrincipalPermission p = new   
PrincipalPermission(null, "Administrators");  
  p.Demand();  
    
  // protected code  
}

In this example, an exception will be thrown if the user is not in the Administrators group.

You can also place a declarative **PrincipalPermissionAttribute** on any WCF operation or business component method to apply the demand before the operation or method is invoked:

[PrincipalPermission(SecurityAction.Demand, Role =  
"Administrators")]  
public string AdminsOnly()  
{  
  // protected code  
}

This approach is preferable since it decouples the security requirements from the actual code within the operation.

In both scenarios, the IsAuthenticated property of the identity is verified, and the IsInRole() method is invoked to check membership using the IPrincipal object attached to the thread.

***Custom Authorization Policies***

Credential authentication, default claimset generation, and access to the security principal for role-based demands are all features that are configurable as I’ve discussed so far in this section. These are features that you get “for free”, with little extra coding effort.

I also mentioned earlier that you can create custom authorization policies for your WCF services. These are types that implement the **IAuthorizationPolicy** interface from the **System.IdentityModel.Policy** namespace. Here are a few useful reasons to create a custom authorization policy:

* When the service requires SAML tokens as the client credential type the claims are not authenticated against any existing role provider. A custom authorization policy can inspect these claims and initialize the security context accordingly.
* A service may replace traditional role-based security with claims-based security. An authorization policy can be used to normalize the set of claims received from different tokens into a common set of claims used for claims-based security.
* Services that use a custom role provider must provide an authorization policy to create an IPrincipal for the security context. Without it, authorization will fail.

The code sample for this article includes an advanced sample that illustrates an IAuthorizationPolicy implementation.

***Impersonation***

With all this talk about authentication and authorization, impersonation is worth discussing. When Windows credentials are used, the service can be configured to impersonate callers so that the request thread operates under the impersonated Windows token. This makes it possible for services to access protected Windows resources under the identity of the caller, instead of the process identity of the service-for that request.

Using the **OperationBehaviorAttribute** you can apply impersonation rules per operation by setting the Impersonation property to one of the following:

* **ImpersonationOption.NotAllowed**. The caller will not be impersonated.
* **ImpersonationOption.Allowed**. The caller will be impersonated if a Windows credential is provided.
* **ImpersonationOption.Required**. The caller will be impersonated and a Windows credential *must* be provided to support this.

This behavior is applied to service operations.

[OperationBehavior(Impersonation =   
ImpersonationOption.Allowed)]  
public string DoSomething()  
{  
  ...  
}

You can also set this for all operations by declaratively configuring the *impersonateCallerForAllOperations* attribute for the service authorization behavior.

<behaviors>  
  <serviceBehaviors>  
    <behavior name="serviceBehavior">  
      <serviceAuthorization   
impersonateCallerForAllOperations="false"/>  
    </behavior>  
  </serviceBehaviors>  
</behaviors>

Clients can also control impersonation, to prevent services from using their identity to access resources. Windows credentials have an *AllowedImpersonationLevel* property that can be set to one of the following:

* TokenImpersonationLevel.None
* TokenImpersonationLevel.Anonymous
* TokenImpersonationLevel.Identification
* TokenImpersonationLevel.Impersonate
* TokenImpersonationLevel.Delegate

*None* and *Anonymous* protect the caller’s identity but aren’t useful for authentication. *Identify* is the default and preferred setting since it allows services to identify the caller but disallows impersonation. *Impersonate* and *Delegate* will allow impersonation across one machine, or delegation with a Kerberos ticket, respectively.

You set the value on the proxy as follows:

localhost.HelloIndigoServiceClient proxy = new   
Client.localhost.HelloIndigoServiceClient();  
…                  
proxy.ClientCredentials.Windows.  
AllowedImpersonationLevel =   
TokenImpersonationLevel.Identification;

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Applied WCF Security**  At this point you should have a pretty good idea of the security-related settings that affect how authentication, authorization and message protection are applied. You should also have a clearer picture of the authentication and authorization process, and the relevance of identities to and role-based security. Still, it helps to look at concrete examples that apply these security features.   |  |  |  | | --- | --- | --- | | **"** | **“In a federated scenario, the service can require that client applications send a security token generated and signed by a specific, trusted identity provider.”** | **"** |   If you take into consideration interoperability and the sheer number of custom situations disparate applications call for-it would be impossible to elaborate on every scenario. That said, in this section I’ll walk you through some common situations that require different bindings, security settings, and authorization techniques, to give you an idea of how that maps to an implementation in the sample code for this article.  ***Intranet Applications***  Internal applications that run on the intranet (and share the same Windows domain as the service) can generally take advantage of a faster transfer protocol like TCP. In terms of security settings, the following may apply:   * Windows client credentials are used for client authentication. * Authentication and authorization use default Windows membership and role providers. * Messages are encrypted and signed by the transport layer. * The service will be self-hosted or hosted in the Windows Activation Service (WAS) for TCP access. * The service implements role-based permission demands on protected operations. * The service rejects impersonation of Windows accounts, and uses a trusted subsystem model.   **Figure 2** illustrates this scenario. The **NetTcpBinding** service model configuration for this scenario is shown in **Listing 2**.  [Click for a larger version of this image.](http://www.code-magazine.com/ShowLargeArticleImage.aspx?QuickID=0611051&Image=Fig002.TIF)  **Figure 2:** Intranet clients on the same Active Directory domain can rely on TCP protocol and use Windows credentials for mutual authentication and message protection.  The client binding configuration to consume this service is equivalent but the behaviors are local to the service. Windows credentials of the logged in user will automatically be passed by the client channel. If you use a login dialog to collect credentials, set the Windows credentials on the proxy this way:  HelloIndigoServiceClient proxy = new  HelloIndigoServiceClient();  NetworkCredential creds = new NetworkCredential ("username", "password", "domain");  proxy.ClientCredentials.Windows. AllowedImpersonationLevel =  TokenImpersonationLevel.Identification;  proxy.ClientCredentials.Windows.AllowNtlm = false; proxy.ClientCredentials.Windows.ClientCredential  = creds;  string s = proxy.HelloIndigo("some string");  Setting *AllowNtlm* to false prevents workgroup authentication from being supported, but you will want this enabled for testing outside of a domain environment.  ***Internet Applications***  When remote clients are not part of the Windows domain, they use Internet protocols to access services (HTTP or HTTPS). The configuration settings in this case will vary based on many factors, including the type of protocols client applications can support.  To expose an endpoint that supports earlier Web service standards, *WS-I Basic Profile* is the typical baseline. Usually the same endpoint will support *WS-I Basic Security Profile* for security-related settings. So, the requirements of the service might be:   * An SSL connection is used to identify the service and to protect message transfer over HTTPS. * UserName credentials are used for client authentication. * Authentication and authorization use the built-in ASP.NET membership and provider model. * Services are hosted in IIS with ASP.NET integration.   **Figure 3** illustrates this scenario. The **BasicHttpBinding** service configuration that implements these requirements is shown in **Listing 1**.  [Click for a larger version of this image.](http://www.code-magazine.com/ShowLargeArticleImage.aspx?QuickID=0611051&Image=Fig003.TIF)  **Figure 3:** A service exposing two endpoints to support BasicHttpBinding and WSHttpBinding. Both endpoints require UserName credentials and authenticate using ASP.NET providers.  This endpoint can only be accessed over HTTPS since transport transfer security is specified. Service authorization and authentication settings are specified in the <serviceBehaviors> section, but for this binding, the <serviceCertificate> is unused since HTTPS is used.  Using svcutil.exe a proxy is created for the service, and configuration settings for both bindings in **Listing 1**. Here is the code to consume the Basic Profile binding and specify the UserName credential in code.  localhost.HelloIndigoServiceClient soap11proxy =  new BasicClient.localhost.HelloIndigoServiceClient ("BasicHttpBinding\_IHelloIndigoService");  soap11proxy.ClientCredentials.UserName.UserName =  "Admin"; soap11proxy.ClientCredentials.UserName.Password =  "[p@ssw0rd](mailto://p@ssw0rd)";  string s = soap11proxy.HelloIndigo("Soap 1.1  endpoint using BasicHttpBinding.");  For clients that can support advanced Web services standards (WS\*) a whole range of possible options exist. Still, a typical scenario that is interoperable might have these requirements:   * A service certificate is supplied to identify the service and protect messages during transfer. * UserName credentials are used for client authentication. * Secure sessions and credential negotiation are disabled. * Authentication and authorization use the built-in ASP.NET membership and provider model. * Services are hosted in IIS with ASP.NET integration.   **Figure 3** illustrates the same service exposing a **WSHttpBinding** alongside a BasicHttpBinding endpoint. **Listing 1** illustrates the binding configuration for WSHttpBinding. This endpoint shares <serviceBehavior> settings with the BasicHttpBinding endpoint, however in this case, since message transfer security is used, the <serviceCertificate> element is required.  Since the client configuration generated by svcutil.exe includes an encoded service certificate (discussed earlier) no changes are required to successfully invoke the service. The following code instantiates the proxy for the WSHttpBinding endpoint, and invokes the service with WS\* protocols:  localhost.HelloIndigoServiceClient soap12proxy =  new BasicClient.localhost.HelloIndigoServiceClient ("WSHttpBinding\_IHelloIndigoService");  soap12proxy.ClientCredentials.UserName.UserName =  "Admin"; soap12proxy.ClientCredentials.UserName.Password =  "[p@ssw0rd](mailto://p@ssw0rd)";  string s = soap12proxy.HelloIndigo("Soap 1.2  endpoint using WSHttpBinding."); | | & | |  | | |  | | --- | |  | |
| |  | | --- | |  |  |  | | --- | |  | | [[http://www.code-magazine.com/Images/CodeSnippet.jpg](http://www.code-magazine.com/CodeListing.aspx?QuickID=0611051&id=59)](http://www.code-magazine.com/CodeListing.aspx?QuickID=0611051&id=59)**Listing 2: Possible service model configuration for intranet scenarios** | | <system.serviceModel>   <services>     <service name="HelloIndigo.HelloIndigoService"  behaviorConfiguration="serviceBehavior">       <endpoint contract="HelloIndigo.IHelloIndigoService"  binding="netTcpBinding" />     </service>   </services>   <behaviors>     <serviceBehaviors>       <behavior name="serviceBehavior">         <serviceAuthorization  impersonateCallerForAllOperations="false"/>         <serviceCredentials>           <windowsAuthentication allowAnonymousLogons="false"  includeWindowsGroups="true" />         </serviceCredentials>       </behavior>     </serviceBehaviors>   </behaviors> </system.serviceModel> | | | | | | | |
| ***Business Partner Applications***  A business partner accessing your services over the Internet may require a different approach to authentication and authorization. Consider these requirements:   * A service certificate is supplied to identify the service and protect messages during transfer. * Certificates are used to uniquely identify partners. * Certificates are authenticated using the default certificate validation process. * Certificates are authorized by placing the corresponding public key of each partner in the TrustedPeople folder for the LocalMachine certificate store.   **Figure 4** illustrates this scenario and **Listing 3** shows the required configuration settings for WSHttpBinding. Certificates can also be used with other bindings as a form of authentication.  [Click for a larger version of this image.](http://www.code-magazine.com/ShowLargeArticleImage.aspx?QuickID=0611051&Image=Fig004.TIF)  **Figure 4:** A service requiring Certificate credentials over WSHttpBinding. Authorized public key certificates should be installed in the LocalMachine certificate store under TrustedPeople.  The client configuration generated by svcutil.exe doesn’t include information about the location of the client certificate. You must make a modification to the configuration on the client to add the following behavior and associate it to the client endpoint:  <behaviors>   <endpointBehaviors>     <behavior name="clientBehavior">       <clientCredentials>         <clientCertificate findValue="SubjectKey"  storeLocation="CurrentUser" storeName="My"  x509FindType="FindBySubjectName"/>       </clientCredentials>     </behavior>   </endpointBehaviors> </behaviors>  Certificate credentials can also be useful for services authenticating across tiers behind the firewall.  ***Federated Security***  It is impossible to include an in-depth discussion on federated security scenarios in this article, but it at least warrants an introduction to whet your appetite. In a federated scenario the service can require that client applications send a security token generated and signed by a specific, trusted identity provider (**Figure 5**).  [Click for a larger version of this image.](http://www.code-magazine.com/ShowLargeArticleImage.aspx?QuickID=0611051&Image=Fig005.TIF)  **Figure 5:** Communication between client application, target service (relying party) and identity provider (token issuer) when the target service relies on SAML tokens issued by the trusted identity provider.  In some cases the identity provider is the same company as the target service-but they have decoupled token issuance from business services. The client application authenticates to the identity provider and requests a token for a particular subject (usually a user). The identity provider returns a signed and encrypted token that can be used for subsequent communications with the target service. It carries the claims requested if the subject is authorized, and the service can use this for a rich, claims-based authorization model.  For this situation, the target service may have the following requirements:   * Supply a service certificate to identify the service and protect messages during transfer. * Require an IssuedToken for client credentials, identifying the authorized identity provider (another service known as a Security Token Service or STS). * Use a custom authorization policy to validate that claims received in the token satisfy service requirements.   The service configuration for this scenario will use **WSFederationHttpBinding**-which allows you to specify which identity provider you require claims from, and which claims you expect the issued token to contain. An abbreviated version of this binding configuration is illustrated in **Listing 4**.  There are many details that warrant discussion in a federated security scenario including the detailed implementation of the identity provider, authentication requirements to the provider, SAML token generation, authorization of claims and the use of custom IIdentity and IPrincipal objects to streamline authorization. You can gain insight into these details from the federated security code sample provided with this article.  **Conclusion and Next Steps**  You’ve probably gathered from the details in this article that there are many intricacies in WCF security. I have covered most of the security-related binding properties, and many behavioral settings for specific scenarios-and this will definitely get you on the right track. Beyond these topics, you will still want to explore security scenarios for some of the bindings that I have left out of this discussion for simplicity. These have overlapping uses for the concepts I’ve discussed here but also introduce more possibilities. In addition, federated security scenarios and Windows CardSpace each warrant a focused discussion since they are quite advanced and relatively new concepts to most. You can find many resources on these subjects on the .NET 3.0 community portal here: <http://www.netfx3.com/>. Enjoy!  *Michele Leroux Bustamante* | & | |  | | |  | | --- | |  | | |
| |  | | --- | |  |  |  | | --- | |  | | [[http://www.code-magazine.com/Images/CodeSnippet.jpg](http://www.code-magazine.com/CodeListing.aspx?QuickID=0611051&id=80)](http://www.code-magazine.com/CodeListing.aspx?QuickID=0611051&id=80)**Listing 3: Service model configuration to require certificate credentials and control authentication and authorization settings** | | <system.serviceModel>   <services>     <service name="HelloIndigo.HelloIndigoService"  behaviorConfiguration="serviceBehavior">       <endpoint contract="HelloIndigo.IHelloIndigoService"  binding="wsHttpBinding" bindingConfiguration="wsHttp"/>     </service>   </services>   <bindings>     <wsHttpBinding>       <binding name="wsHttp">         <security mode="Message">           <message clientCredentialType="Certificate"/>         </security>       </binding>     </wsHttpBinding>   </bindings>   <behaviors>     <serviceBehaviors>       <behavior name="serviceBehavior">         <serviceCredentials>           <clientCertificate>             <authentication  certificateValidationMode="PeerOrChainTrust"  trustedStoreLocation="LocalMachine"  />           </clientCertificate>           <serviceCertificate findValue="RPKey"  storeLocation="LocalMachine" storeName="My"  x509FindType="FindBySubjectName"/>         </serviceCredentials>       </behavior>     </serviceBehaviors>   </behaviors> </system.serviceModel> | | | | | | | |

## WCF Certificates without Certificate Store

Security has an important role in any distributed application and Windows Communication Foundation (known as WCF or Indigo), the new Microsoft communication framework, implements many security standards and has a wide range of features available.

One of the most important aspects of security is authentication. WCF can be configured to use many authentication methods:

Anonymous caller

User name and password

Certificate

Windows

CardSpace

### The problem

The use of certificates for authentication is not new, but is still one of the most common way to authenticate a subject. WCF has a built-in support for certificates that conform to the Web Services Security (WS-Security) standards.

The problem with the default configurations and examples available is that all the certificates must be installed in the Certificate Store, which basically is a central location where Windows saves all the certificates (used also for other applications: Internet Explorer, ...).

Why this solution causes some problems? The easy answer is because it is not easy to correctly configure all the certificates. For more details:

When you deploy your service to the server you must install in the Certificate Store all the certificates used (in different locations based on the use of the certificate).   
This operation must be executed using an installation program, a script file or a batch file. For this reason, it is difficult to deploy the application using an xcopy/ClickOnce installation.

Each client must also install the certificate used to authenticate itself always in the Certificate Store. This is easy if you have a small number of clients but very difficult if you must manually configure each computer (in addition, for the client, you can't use an xcopy/ClickOnce installation).

You must give to the running process (like ASP.NET) the permissions to read the certificate private key. This step usually requires changing the file system permission. This again requires a script file or an installation which is not always easy.

If you are using a shared hosting probably you can't install certificates or change certificate permissions.

As a developer I like to have each project isolated from the others. I want to be able to easy test different configurations or applications, I like to simply download the latest version from the code repository and run it, without any special configuration. Using the Certificate Store I must always remember to install or uninstall the certificates each time.

At the following MSDN page you can see an example of a configuration using certificates and a description of how to install certificates using the classic solution: [MSDN: Message Security Certificate](http://msdn2.microsoft.com/en-us/library/ms751516.aspx).

These are the reasons behind my decision to try a different approach that I will describe in the following sections.

### The solution

My goal is to find an easy way to use certificates without using Certificate Store. WCF can be easily extended; in this article I will show you how to extend WCF to load the certificates from files.

I known that storing certificates on the file system is less secure, but I think that with some attention this can be a useful alternative. See the [Disadvantages](http://www.codeproject.com/KB/WCF/wcfcertificates.aspx#Disadvantages) section for a discussion of the possible problems of my approach.

Consider that with my solution, I simply change how the certificates are loaded, all the advantages of using WCF (standards, proved code, ...) are still valid. Most important you must still use most of the settings required to use certificates. For a complete and working example, I suggest to look at the sample project in the zip file or follow the [Quick start tutorial](http://www.codeproject.com/KB/WCF/wcfcertificates.aspx#Quick_start_tutorial) section to implement this solution on your own project.

To understand how this solution works, continue reading the next section.

## Implementation details

Loading a certificate from a file is quite easy, you must simply use the System.Security.Cryptography.X509Certificates.X509Certificate2 class:

//Load the certificate from a file

X509Certificate2 certificate =

new X509Certificate2(fullpath, password);

The first parameter is the path of the certificate file, the second parameter is the password used to encrypt the private key (if present).

With X509Certificate2 class you can load 2 kinds of files:

.cer file - Used to store a public key

.pfx file - Used to store a public+private key (optional encrypted with a password)

You can obtain these files from a public certification authority or create your self-signed certificates using makecert.exe and pvk2pfx.exe, both available as Visual Studio Tools. Here an example on how to create a certificate:

makecert -r -pe -n "CN=CompanyXYZ Server" -b 01/01/2007 -e 01/01/2010

-sky exchange Server.cer -sv Server.pvk

pvk2pfx.exe -pvk Server.pvk -spc Server.cer -pfx Server.pfx

The first command (makecert) generates a public key (in this case Server.cer) and a private key (in this case Server.pvk). The second command (pvk2pfx) merges the 2 files on a single .pfx file (in this case Server.pfx).

When executing makecert and pvk2pfx, you can insert a password used to encrypt the private key.

### Service authentication

This section describes how to configure the server with the certificate used to authenticate the service.

First you must add a reference to the DevAge.ServiceModel.dll. This assembly (included in the sample project with the full code) contains some classes used to load the certificate from a file that I will soon describe.

Then you must generate the service certificate and put it in a secure directory. For ASP.NET web site, I think that a good solution is to use the App\_Data directory, which is automatically configured to not allow public access. I have generated in the App\_Data directory 2 files: Server.cer and Server.pfx.

Usually you can specify the certificate for the service inside the system.serviceModel section of the .config file like this:

<behaviors>

<serviceBehaviors>

<behavior name="serviceCredentialBehavior">

<serviceCredentials>

<serviceCertificate findValue="Contoso.com"

storeLocation="LocalMachine"

storeName="My"

x509FindType="FindBySubjectName" />

</serviceCredentials>

</behavior>

</serviceBehaviors>

</behaviors>

This configuration basically tells WCF where to find the certificate inside the Certificate Store.

To load the certificate from file, unfortunately you can't simply change the configuration, but you must set the ServiceHost.Credentials.ServiceCertificate.Certificate property.

I have created a new configuration section where you can specify where to find the server certificate:

<devage.serviceModel>

<services>

<add name="MathService"

serverCertificate="App\_Data\Server.pfx"

/>

</services>

</devage.serviceModel>

Then I have created a new ServiceHost derived class, DevAge.ComponentModel.CertificateServiceHost, which automatically reads the new configuration section and sets the ServiceHost.Credentials.ServiceCertificate.Certificate property.

If you directly create the ServiceHost class you can simply change your code to create the DevAge.ComponentModel.CertificateServiceHost instead.  
If you use ASP.NET (that automatically creates the ServiceHost class) you must configure the .svc file in this way:

<% @ServiceHost Language="C#" Debug="true"

Service="MathService"

CodeBehind="~/App\_Code/MathService.cs"

Factory="DevAge.ServiceModel.CertificateServiceHostFactory" %>

Note the Factory property that uses a specific factory class to create my DevAge.ComponentModel.CertificateServiceHost.  
This configuration replaces the standard serviceCertificate section.

NOTE: When you will create the proxy client with svcutil you will see an identity section inside the endpoint section:

<identity>

<certificate encodedValue="...." />

</identity>

This section must be recreated each time you change the service certificate, because it contains the public identification of the certificate generated at design time. This value is used by the client to be sure to speak with the expected service.

### Client authentication

This section describes how to use certificates to authenticate each client.

You must create a certificate for each client (or share the same certificate for more than one client). You can use the same command to generate self-signed certificates or obtain it from a certification authority.

In my example I have generated in the client directory 2 files: Client.cer and Client.pfx.

Normally each client can configure the application with the certificate using the configuration below:

<behaviors>

<endpointBehaviors>

<behavior name="ClientCredentialsBehavior">

<clientCredentials>

<clientCertificate findValue="Cohowinery.com"

storeLocation="CurrentUser"

storeName="My"

x509FindType="FindBySubjectName" />

</clientCredentials>

</behavior>

</endpointBehaviors>

</behaviors>

This configuration always uses the Certificate Store to locate the right certificate to use and like in the previous example, there isn't a way to directly use a certificate file. So I have again created a new configuration section to specify the certificate file name:

<devage.serviceModel>

<endPoints>

<add contract="Client.MathService.IMathService"

clientCertificate="Client.pfx" />

</endPoints>

</devage.serviceModel>

To manually set the certificate for the client you can set the ClientProxy.ClientCredentials.ClientCertificate.Certificate property.

Unfortunately in this case, there isn't a way to automatically set this property by reading the new section (or at least I cannot find a way...) so you must configure each proxy with code like this:

//Create the client proxy as usual

MathService.MathServiceClient service =

new Client.MathService.MathServiceClient();

//Configure the client reading the devage.serviceModel configuration section

DevAge.ServiceModel.Proxy<MathService.IMathService>.Configure(service);

The DevAge.ServiceModel.Proxy class reads the configuration section and sets the ClientProxy.ClientCredentials.ClientCertificate.Certificate property.

If the certificate used by the client is trusted by the server (usually must be added in the Trusted people of the Certificate Store) your configuration is complete. If otherwise the certificate is self-signed or anyway is not trusted by the server you must configure the server to accept the certificate.

To solve this issue, I have implemented a custom X509CertificateValidator class, DevAge.ServiceModel.CustomCertificateValidator, which can be configured for each service. You must extend the devage.serviceModel section of the .config file of the service (on the server) like this:

<devage.serviceModel>

<services>

<add name="MathService"

serverCertificate="App\_Data\Server.pfx"

clientCertificates="App\_Data\Client1.cer,App\_Data\Client2.cer"

/>

</services>

</devage.serviceModel>

Note the new clientCertificates property which contains a list of client certificates to consider trusted.

You must copy inside the App\_Data server directory all the client public key files (in this case Client1.cer and Client2.cer files).

The CertificateServiceHost class automatically reads this new section and sets the X509CertificateValidator using code like this:

X509ClientCertificateAuthentication authentication =

serviceHost.Credentials.ClientCertificate.Authentication;

authentication.CertificateValidationMode =

System.ServiceModel.Security.X509CertificateValidationMode.Custom;

authentication.CustomCertificateValidator =

new CustomCertificateValidator(clientCertificates);

If the certificate file has a password, you can specify it within the file name using this format: filename|password. For example you can specify the serverCertificate attribute of the web.config with this value: App\_Data\Server.pfx|yourpassword.   
Remember also that you can specify 2 different password for the \*.pvk and for the \*.pfx files, the password to use is the password specified for the \*.pfx file. See the makecert and pvk2pfx documentation for more details on how to set passwords.

### Disadvantages

Storing the certificates on files is easier but is less secure because a malicious user can steal the private key. The public key can be shared without problems but if someone steals the private key your security is compromised. With the private key a malicious user can decrypt the messages or impersonate your service or a client.

I think anyway that usually if someone can access your file system you probably have many other security problems. Consider also that if someone has access to your file system, he or she can probably (depends on the type of the attack) also access the Certificate Store.

My suggestion is to correctly evaluate your application and your security requirements. In many cases, I think that saving certificate on a file is quite secure and can be useful.

Finally consider that you can use a mixed configuration, for example reading the server certificate inside the Certificate Store and the client certificates on file system.

## Message based correlation with WF/WCF in .NET 3.5

When .NET 3.0 was released we got Windows Communication Foundation (WCF) and Windows Workflow Foundation (WF). These were great technologies, but they would have been even greater if they were used together. Unfortunately, Microsoft decided not to include this for .NET 3.0. However, with the release of .NET 3.5 we got the WCF Send and Receive activities that you can use in your workflows in order to implement a WCF service contract by using a workflow. This works great when you want to start a new workflow instance for every call made to your service. But when you want your existing workflow instance to receive the input of another service operation, you'll need to do some correlation.

Microsoft solved this by involving the client. When the first operation that created your workflow instance succeeded, a header or a cookie is attached to the outgoing message containing the workflow instance ID and it is the clients responsibility so store this ID and again attach it to the message when it performs the second operation. This can get tricky if the client is stateless, or when the client that creates the workflow instance is not the same as the client who invokes the second operation. It also requires your client to have knowledge of implementation details of the service. If the client doesn't know that it has to retrieve the instance ID from the reply it gets from the service and then use that same ID again for the second call, it's all going to break apart. This isn't much of a deal if you're developing both the client and the server, but if the client is being developed by a third party it becomes a fair bit more complex.

Ideally you would want the message that is going into the second service operation to have a key value with which you can find the workflow instance back. It would be even better if that value is something that has business value as well, such as an OrderID, or CustomerID. This means you can talk about functional keys outside the service, but you can do the mapping of those functional keys to workflow instances within the implementation of your service.

Unfortunately Microsoft hasn't provided this as an option. It is coming with .NET 4.0, but we're not quite there yet and the programming model for WF 4.0 is going to be quite different when compared to .NET 3.5. We needed a solution for this problem now, rather than having to wait for .NET 4.0 to come out. So I set out to see if we could somehow do message based correlation with the WCF Send and Receive activities in .NET 3.5. I quickly learned that that wasn't an easy task...

The implementation of the WCF Send and Receive activities relies heavily on WCF's extensibility. In fact, Microsoft has made a whole new assembly named System.WorkflowServices which contains a whole bunch of classes that live in the existing System.ServiceModel and System.Workflow namespaces. One of the things they've added is called a context binding. The context binding basically does the whole cookie/header approach to solving the correlation problem. So I thought I'd write a WCF behavior that implements [IDispatchMessageInspector](http://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.idispatchmessageinspector.aspx) that can inspect the messages that are being received at the service, fetch the instance ID from it and then add the required header to the message so that the existing infrastructure from System.WorkflowServices can pick it up and make the call to the right instance. However, it turns out that the message inspectors are almost at the bottom of the foodchain and the whole System.WorkflowServices infrastructure already decided to create a new instance for the second operation before my message inspector was invoked. So that wasn't going to work...

After some more digging around using .NET Reflector I found an interesting class called WorkflowInstanceContextProvider. It was in this class that the message header (or rather a message property that was being fed with information from the header) was being read and the appropriate workflow instance was being attached to the operation context. Unfortunately the class was made internal, so I couldn't inherit from it. It did however implement an interface, which is the standard WCF interface [IInstanceContextProvider](http://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.iinstancecontextprovider.aspx). So I decided to create my own IInstanceContextProvider implementation that would wrap an existing IInstanceContextProvider implementation, which in this case would be the WorkflowInstanceContextProvider. And to my surprise, this worked perfectly. I made a small test with a workflow that had two Receive activites. When the first one was invoked a CodeActivity made sure that the instance ID of that instance was assigned to a static member in my IInstanceContextProvider implementation. When the second call came in I would inject the value of that static member into the message by using the [ContextMessageProperty](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.contextmessageproperty.aspx) class. To make my test better I used the input of the first operation as the output of the second operation and that test worked.

All I had to do now was to actually get the value to correlate on from the message, how hard could it be? Well, a bit harder than you would think. The IInstanceContextProvider interface provides to method that both receive the Message instance that is being processed. When you have a [Message](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.message.aspx) object you can read it using either the [GetBody](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.message.getbody.aspx) or the [GetReaderAtBodyContents](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.message.getreaderatbodycontents.aspx) methods. This allows you to read the contents of the message body and do whatever parsing you need on them. The problem is that once you read a message, you can't read the message again. This is done on purpose so you can use streaming messages. If you want to read the message twice you would use the [CreateBufferedCopy](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.message.createbufferedcopy.aspx) method and work from there.

This is all good and well, but the interface I was implementing gave me only a reference to the message, but it did not give me a reference to the reference. What this means is that although you can change the message you receive, you can't change the reference to the message so can't create a new message and then assign it to your reference. Well, you can, but whoever called your method still has a reference to the original message. So if I was reading the message in my IInstanceContextProvider implementation it could not be read again further up the WCF stack. Eventually an exception would be raised saying that the message was already read and could not be read again.

So there I was. I had a solution for one problem, but now I had another problem which I couldn't easily fix. That was until my colleague [Marcel](http://blogs.infosupport.com/blogs/marcelv/) came up with a simple solution. The Message class I mentioned earlier also has a ToString method. This actually serializes the whole message into XML, regardless of whether it is a streamed or buffered message. When the message is streamed though, you will not get the body of the message, as explained [here](http://blogs.thinktecture.com/cweyer/archive/2008/06/05/415161.aspx). When you call ToString you can still read the message later on. So now we have a solution that works, but only if you're using buffered messages. If you use streamed messages things will break. We'll probably implement a check in our implementation that verifies that you are using buffered messages when the behavior is applied, but that's another problem...

## Message Correlation Sample

This sample demonstrates how a Message Queuing (MSMQ) application can send an MSMQ message to a Windows Communication Foundation (WCF) service and how messages can be correlated between sender and receiver applications in a request/response scenario. This sample uses the msmqIntegrationBinding binding. The service in this case is a self-hosted console application to allow you to observe the service that receives queued messages. k

The service processes the message received from the sender and sends a response message back to the sender. The sender correlates the response it received to the request it sent originally. The **MessageID** and **CorrelationID** properties of the message are used to correlate the request and response messages.

The IOrderProcessor service contract defines a one-way service operation that is suitable for use with queuing. An MSMQ message does not have an Action header, so it is not possible to map different MSMQ messages to operation contracts automatically. Therefore, there can be only one operation contract in this case. If you want to define more operation contracts in the service, the application must provide information as to which header in the MSMQ message (for example, the label, or correlationID) can be used to decide which operation contract to dispatch. This is demonstrated in the [Custom Demux](http://msdn.microsoft.com/en-us/library/ms752265.aspx).

The MSMQ message also does not contain information as to which headers are mapped to the different parameters of the operation contract. Therefore, there can be only one parameter in the operation contract. The parameter is of type [MsmqMessage](http://msdn.microsoft.com/en-us/library/ms600458.aspx) **MsmqMessage<T>**, which contains the underlying MSMQ message. The type "T" in the **MsmqMessage<T>** class represents the data that is serialized into the MSMQ message body. In this sample, the PurchaseOrder type is serialized into the MSMQ message body.

## Additional Notes

## 1

You can secure the service using Message level security, but streamed requests are not supported using transport security for the performance reasons.

You can also secure the transport by choosing SecurityMode = Transport, and Authentication = Anonymous.  The underlying connection WILL be secured using SSL, you just won't have credential based authn.

If you want to add custom authentication you will need to create a custom security binding

streamed requests are not supported when using authentication over transport security (Basic, Digest, NTLM, etc), you can still use transport security with Anonymous authn, this just gives you straight SSL

Depending on the type of service certificate, you may need to specify a DNS identity for your client endpoints. This is used when the client authenticates the service to ensure that it is talking to the expected service and not any other hoax service intercepting your calls to the genuine service. The exception below is an indication of this:

“Identity check failed for outgoing message. The expected DNS identity of the remote endpoint was 'X' but the remote endpoint provided DNS claim 'Y'. If this is a legitimate remote endpoint, you can fix the problem by explicitly specifying DNS identity 'Y' as the Identity property of EndpointAddress when creating channel proxy.”

## 2 netsh http add urlacl url

netsh http add urlacl url=https://+:8731/Design\_Time\_Addresses user=”user”

netsh http add sslcert ipport=10.116.1.194:8731 certhash=71e507f4022e9b47ddf997cd26a0a80aae72dddb appid={444F5B4C-8F8A-451D-B589-7E3104AA7A32}

Using HTTPNamespaceMapper to add port

<http://+:8731/Design_Time_Address>

**How to use makecert.exe to create a self-signed test certificate that can be used with IIS for SSL**

**Problem:** Special options must be specified with makecert.exe, to create a self-signed certificate that can be used with IIS (Microsoft Internet Information Server).

**Note**: Microsoft recommends to install and use the "Certificate Server" to generate an SSL test certificate ([Q216907](http://support.microsoft.com/default.aspx?scid=kb%3ben-us%3b216907)), instead of using makecert.exe. But using makecert is simpler.

**Solution:**

The following command can be used to create and import a self-signed SSL test certificate:

makecert -r -pe -n "CN=www.yourserver.com" -b 01/01/2000 -e 01/01/2036 -eku 1.3.6.1.5.5.7.3.1 -ss my -sr localMachine -sky exchange -sp "Microsoft RSA SChannel Cryptographic Provider" -sy 12

To install this certificate in IIS 5.0, open the IIS "Web Site Properties", "Directory Security", "Server Certificate...", "Assign an existing certificate" and select the new certificate from the list.

**Note:** Older versions of makecert.exe do not support the "-pe" option, which makes the private key exportable. If you have an old version of makecert.exe, you can omit the "-pe" option, but then the certificate cannot be exported including the private key.

(The October 2002 version of the [Platform SDK](http://www.microsoft.com/msdownload/platformsdk/sdkupdate/psdk-full.htm) (build 3718.1) contains a [new version of makecert.exe](http://www.inventec.ch/chdh/notes/makecert_5_131_3790_0.zip) (5.131) that supports the "-pe" option. The .NET Framework SDK 1.0 of 2002-03-19 contains an old version of makecert.exe that does not support the "-pe" option).

If the private key is exportable, you can export the certificate together with the private key into a PFX (PKCS #12) file as described in [Q232136](http://support.microsoft.com/default.aspx?scid=kb;EN-US;232136).

**Note:** SSL server certificates for IIS are stored in the "Personal" ("My") certificate store of the "computer account" ("localMachine"). The "Certificates" snap-in of the Microsoft Management Console (mmc.exe) must be used to manage these certificates. The normal certificate management window (accessible via "Internet Properties" / "Content" / "Certificates" or via "Control Panel" / "Users and Passwords" / "Advanced" / "Certificates") cannot be used.

**Note:** To create a key with more than 512 bits, use the "-len" parameter of makecert.exe.

Author: [Christian d'Heureuse](http://www.inventec.ch/chdh) ([chdh@inventec.ch](mailto:chdh@inventec.ch), [www.source-code.biz](http://www.source-code.biz))  
[Index](http://www.inventec.ch/chdh/notes/swhw_notes.htm)