For more information on implementing message layer security, see the following composite patterns:

- Implementing Direct Authentication with UsernameToken in WSE 3.0
- Implementing Message Layer Security with Kerberos in WSE 3.0
- Implementing Message Layer Security with X.509 Certificates in WSE 3.0

There is already a lot of good information available on using transport layer security to secure Web services, so this information is provided in the form of the following references, which point you to appropriate guidance for implementing transport layer security. For more information on implementing transport layer security, see the following sections in References for Transport Layer Security:

- Implementing Brokered Authentication Using Windows Integrated Security on IIS
- Implementing Transport Layer Data Confidentiality Using HTTPS
- Implementing Transport Layer Security Using HTTPS Basic over HTTPS
- Implementing Transport Layer Security Using X.509 Certificates and HTTPS
- Implementing Transport Layer Security with Kerberos and IPSec on Windows Server 2003

Implementing Direct Authentication with UsernameToken in WSE 3.0

Context

You are implementing direct authentication for an online application that consumes a Web service that uses Web Service Enhancements (WSE) 3.0. You are using message layer authentication. The credentials used to prove the identity of the client are validated by an authentication service.

Objectives

The objectives of this pattern are to:

- Implement direct authentication against Active Directory, Active Directory Application Mode (ADAM), or a custom SQL Server™ database using a security token that contains a user ID and password.
- Secure the communication channel by providing data confidentiality and data integrity. You can do this either at the message layer or at the transport layer.
- Demonstrate how to develop a custom **UsernameTokenManager** to support authentication against ADAM or a custom SQL database.
- Demonstrate how to use ASP.NET 2.0 membership providers for SQL Server and a directory service.

Content

This pattern consists of the following sections:

- **Implementation Strategy**: This section provides a high-level description of the strategy used to implement the Direct Authentication pattern. The section also discusses identity stores that you can use and different approaches to ensure secure communication between the participants.
- **Implementation Approach**: This section describes the steps necessary to implement this pattern:
  - General setup
  - Configure the client
  - Configure the service
- **Resulting Context**: This section outlines the benefits, liabilities, and security considerations related to this pattern.
- **Variants**: This section describes alternate choices to using Active Directory as an identity store, demonstrating how to implement both a database and a directory service as an identity store.

Note: The code examples in this pattern are also available as executable QuickStarts on the Web Service Security community workspace.

Implementation Strategy

The WSE 3.0 implementation of **UsernameToken** is used to implement direct authentication at the message layer. The client passes the credentials to the Web service as part of a secure message exchange. A password is sent in the message as plaintext, which is data in its unencrypted or decrypted form. The Web service decrypts the message, validates the credentials, verifies the message signature, and then sends an encrypted response back to the client.

Identity Store Options

There are three options for this pattern to implement different types of identity stores that the service can use to validate the credentials presented in a **UsernameToken**:

- Active Directory
- Database
- Directory service

Note: Direct authentication using Active Directory is described in the base pattern. The other two options are described at the end of this pattern as variants.

Active Directory

The ability to validate credentials presented in a **UsernameToken** to an Active Directory domain is provided with the **UsernameTokenManager** in WSE 3.0. Using Active Directory as an identity store has the following advantages:

- Unlike validating credentials using a database or a Lightweight Directory Access Protocol (LDAP)-enabled directory service, credential validation using Active Directory does not require a custom **UsernameTokenManager** class or an ASP.NET 2.0 membership provider.
- Of the three approaches for this pattern, Active Directory is the simplest option to implement in WSE 3.0.
- While Active Directory does require that users and their roles are maintained in an Active Directory infrastructure so that the service can use them to validate credentials, it does allow you to authenticate users without using Windows Integrated Security.

Database

You can use a database to store credentials that the service can then validate. Using a database as an identity store has the following advantages:

- It provides the capability to integrate with an existing database that is being used as an identity store. If you use a custom database schema, it may require you to implement a custom ASP.NET membership and possibly a role provider. For more information about how to create a custom identity provider, see the "Variant 2 — Using an LDAP Directory Service as the Identity Store" section later in this pattern.
- It supports transactional and concurrent updates to user credentials. For example, concurrent updates to security claims (such as role information for a single user) could occur if the maintenance of user credentials in the database is delegated to several different individuals. If concurrent updates are a concern, you should use either a directory service that supports transactional updates or a database to store user credentials and roles.

Using a database as an identity store does have the disadvantage that it is more difficult to maintain if the database is not shared across multiples services that authenticate the same users. This may cause data ownership and synchronization issues when changes are made to one identity store that must be propagated to the others.

For more information about using a database as an identity store, see the "Variant 1 — Using a Database as the Identity Store" section later in this pattern.

Directory Service

You also can use an LDAP-enabled directory service to store credentials for validation by the service. Using a directory service has the following advantages:

- It provides a viable alternative when you have an LDAP-enabled directory service in place of an Active Directory infrastructure.
- It can be used when you need to authenticate users using ADAM or Active Directory through LDAP ports due to firewall restrictions.

For more information about using a directory service as an identity store, see the "Variant 2 — Using an LDAP Directory Service as the Identity Store" section later in this pattern.

Providing Secure Communication

This implementation provides examples that show how to secure the communication channel between the client and the service, using both the **usernameForCertificateSecurity** and the **usernameOverTransportSecurity** WSE 3.0 turnkey assertions. The communication channel is secured by providing data confidentiality to prevent eavesdropping. Data origin authentication is also provided to prevent tampering or message spoofing. For more information, see Data Confidentiality and Data Origin Authentication in Chapter 2, "Message Protection Patterns."

The **usernameForCertificateSecurity** turnkey assertion secures the communication channel between the client and the service at the message layer using the service's X.509 certificate. But it is not compatible with client computers that have implemented WS-Security 1.0. This is because the **usernameForCertificateSecurity** turnkey assertion depends on the ability to reference **<EncryptedKey>** elements as security tokens, and enables the option for signature confirmation to correlate a response message with the request that prompted it. Both of these features are only available in WS-Security 1.1.

The **usernameOverTransportSecurity** turnkey assertion assumes that communication between the client and service will be secured at the transport layer. This approach is WS-Security 1.0 compatible, but it does not provide security features at the message layer. It also does not ensure that the channel is secured at the transport layer.

If you need to secure the communication channel between the client and service at the message layer with a solution that is compatible with WS-Security 1.0, you will need to create a custom policy assertion.

Note: At the time this pattern was published, most vendors supported WS-Security 1.0 implementations. WSE 3.0 supports features in WS-Security 1.1 and WS-Security 1.0. If you need to interoperate with platforms that do not support WS-Security 1.1 features, choose an option that best supports your interoperability requirements.

Participants

- **Client**. The client accesses the Web service. The client provides the credentials for authentication during the request to the Web service.
- **Service**. The service is the Web service that requires authentication of a client prior to making access control decisions.
- **Identity store**. The entity that stores a client's credentials for a particular identity domain.

Process

The process section of Direct Authentication in Chapter 1, "Authentication Patterns," describes how identity and proof-of-possession are used for authentication. This pattern provides a more refined description of that process within the context of the implementation.

Figure 3.5 illustrates the direct authentication process.

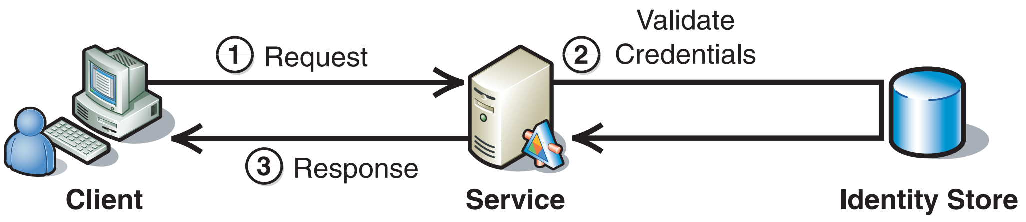

Figure 3.5
The direct authentication process

The steps for this implementation are divided into two parts, based on what happens with the client and what happens with the service:

- The client generates a Web service request.
- The service authenticates a client and returns a response.

The Client Generates a Web Service Request

This part of the process includes three steps:

1. Initialize the **UsernameToken**.
2. Establish message integrity.
3. Encrypt sensitive data in the message.

Step One: Initialize the UsernameToken

This pattern implements a **UsernameToken** with the **SendPlainText** password option to send the password over the network as plaintext. The plaintext value is the actual password because Active Directory requires plaintext passwords for credential validation. This option, which the default implementation of **UsernameTokenManager** uses, is similar to basic authentication over HTTP. You should always secure the communication between the client and server, either at the transport layer using Secure Sockets Layer (SSL) or at the message layer with WSE 3.0.

Step Two: Establish Message Integrity

Data origin authentication is established between the client and the service, either implicitly or explicitly, depending upon one of the two following methods that you can choose to secure messages between the client and the service:

- The **usernameOverTransportSecurity** turnkey assertion with HTTPS.
- The **usernameForCertificateSecurity** turnkey assertion.

HTTPS using the **usernameOverTransportSecurity** turnkey assertion provides data confidentiality and data integrity when you use server certificates. If you require data origin authentication from the client, you need to install and use a certificate for the client. For more information, see the reference, Implementing Transport Layer Security Using X.509 Certificates and HTTPS in Chapter 3, "Implementing Transport and Message Layer Security."

WSE 3.0 policy provides data confidentiality and data origin authentication when the **usernameForCertificateSecurity** assertion is used. The client includes a derived key token in the request message that is encrypted with a wrapped symmetric encryption key. The wrapped symmetric key is encrypted with the service's X.509 certificate public key. This key is referred to as an encrypted key. Accompanied by a valid **UsernameToken**, data origin authentication is provided when the client uses the derived key token to sign the message. For more information about derived key tokens, see Web Services Secure Conversation Language (WS-SecureConversation).

Step Three: Encrypt Sensitive Data in the Message

You should encrypt the message from the client to the service to ensure that only the service, as the intended recipient of the message, can process it. The method that you choose to secure the communication channel between the client and the service should also provide data confidentiality.

The Service Authenticates the Client and Returns a Response

This part of the process has five steps:

1. Decrypt the request message.
2. Verify message integrity.
3. Validate the password.
4. Establish the response integrity
5. Encrypt the response.

Step One: Decrypt the Request Message

The option you choose to secure communication between the client and the service determines how the request message is decrypted. The **usernameOverTransportSecurity** assertion relies on SSL to decrypt the message at the transport layer. WSE 3.0 policy using the **usernameForCertificateSecurity** assertion decrypts the derived key token encrypted with the wrapped symmetric key, and then uses the derived key token to decrypt the message signature, **UsernameToken**, and any other message parts that the client encrypted.

Step Two: Verify Message Integrity

The option you chose to secure communication between the client and the service determines how the message integrity is established and verified. The **usernameOverTransportSecurity** assertion relies on SSL to verify message integrity. If a client certificate is used for the client, the client also provides data origin authentication. WSE 3.0 using the **usernameForCertificateSecurity** assertion verifies the message integrity using the derived key token sent by the client that was decrypted in Step One.

Step Three: Validate the Password

After the service receives the message, the information in **UsernameToken** is verified by WSE 3.0 using the **UsernameTokenManager** class. WSE 3.0 uses the **AuthenticateToken** method of the **UsernameTokenManager** class to validate the information in the **UsernameToken**.

The **UsernameTokenManager** released with WSE 3.0 validates credentials against an Active Directory domain controller. If either a directory service or a database is used to store credentials for validation, then you will need to implement a custom **UsernameTokenManager** class. For more information, see the “Variants” section later in this pattern.

The **UsernameTokenManager** validates the username and password that was sent in the message with Active Directory through the **AuthenticateToken** method. The default **UsernameTokenManager** also establishes a **WindowsPrincipal** instance for the authenticated client and attaches it to the token’s **Principal** property.

Step Four: Establish the Response Integrity

The method used to establish the response message's integrity depends upon whether communication is secured at the message layer using WSE 3.0 or at the transport layer using SSL. If communication is secured at the transport layer, message integrity is provided through SSL. If communication is secured at the message layer, the response message is signed with a key derived from the encrypted key that was sent in the request message.

Step Five: Encrypt the Response

The method used to encrypt the response message depends upon whether communication is secured at the message layer through WSE 3.0 or at the transport layer using SSL. If communication is secured at the transport layer, the response message is encrypted through SSL. If communication is secured at the message layer, the response signature and message parts are encrypted with a key derived from the encrypted key sent in the request message.

Implementation Approach

This section describes how to implement the pattern. This section is divided into three major tasks:

1. **General setup**. This task provides the required steps for both the client and the service.
2. **Configure the client**. This task provides the required steps to configure policy and code on the client.
3. **Configure the service**. This task provides the required steps to configure policy and code on the service.

Note: For the code examples included in this pattern, an ellipsis (...) is used where segments of code, such as class declarations and designer-generated code, have been omitted. You must name variables, methods, and return values and ensure that they are of the appropriate type for the client application.

General Setup

You must install WSE 3.0 on computers that you use to develop WSE-enabled applications. After WSE 3.0 is installed, you must enable the client and the service to support WSE 3.0. You can achieve this by performing the following steps.

► **To enable a Visual Studio 2005 project to support WSE 3.0**

1. In Visual Studio 2005, right-click the application project and select **WSE Settings 3.0**.
2. On the **General** tab, select the **Enable this project for Web Services Enhancements** check box, and then click **OK**.

If you are using the **usernameForCertificateSecurity** assertion to secure communication at the message layer between the client and service, you must configure the X.509 settings for WSE 3.0. For more information about setting up X.509 in WSE 3.0, see General Setup in the Implementing Message Layer Security with X.509 Certificates in WSE 3.0 in Chapter 3, "Implementing Transport and Message Layer Security."

Note: WSE 3.0 offers four different protection levels that determine how messages are secured using SOAP message security. Generally, you should use the **Sign, Encrypt, and Encrypt Signature** setting for best message protection. This setting encrypts the message body and the XML signature, which reduces the likelihood of a successful cryptographic guessing attack against the signature. For this reason, all the composite implementation patterns use this value as default. If you want to use this setting in new Web services you should change the **messageProtectionOrder** attribute to the following value in your security policy:

```
messageProtectionOrder="SignBeforeEncryptAndEncryptSignature"
```

Configure the Client

After enabling the client application to support WSE 3.0 during General Setup, you must enable policy support for it. If your application does not currently have a policy cache file, you can add one for this purpose, and enable policy support by performing the following steps.

► **To add policy support to a WSE 3.0-enabled Visual Studio 2005 project**

1. In Visual Studio 2005, right-click the application project and select WSE Settings 3.0.
2. On the **Policy** tab, select the **Enable Policy** checkbox. Selecting this setting adds a policy cache file with the default name *wse3policyCache.config*.
3. Under **Edit Application Policy**, click **Add**, and then type a policy friendly name for the new application policy, such as "usernameTokenSecurity."
4. Click **OK** to start the WSE Security Settings Wizard, and then click **Next**.
5. On the **Authentication Settings** page, the wizard provides a choice to secure a service or a client. Select **secure a client application** to configure the client.
6. The wizard also provides a choice of authentication methods in the same step. Select **Username**, and then click **Next**.
7. On the **Optionally Provide Username and Password** page, the wizard provides you with options to define a user name and password. Ensure that the **Specify Username Token in code** checkbox is selected and click **Next**.

8. On the **Message Protection** page, you configure options for message protection. For transport layer security, select **None (rely on transport protection)** for the **Protection Order** to use the **usernameOverTransportSecurity** assertion. If you select any other protection option, the policy assertion will be **usernameForCertificateSecurity**.

   You should select the option for **Sign, Encrypt, Encrypt Signature**. By default, the **Enable WS-Security 1.1 Extensions** check box is enabled. This setting must be enabled if you are using message layer security. For more information about these settings, see the "Implementation Strategy" section earlier in this pattern.
9. Click **Next**.
10. If you selected **None (rely on transport protection)** to use transport security in Step 8, skip this step. If you selected any other option, the wizard will prompt you to select a server X.509 certificate for the service on the **Server Certificate** page. Change the Store Location to **LocalMachine** instead of using the default value of **CurrentUser**. Select the certificate for the service to use, and then click **Next**.
11. On the **Create Security Settings** page, review your settings, and then click **Finish**.

After you complete these tasks, your client security policy should look similar to the following code example. Examples for both the **usernameForCertificateSecurity** and **usernameOverTransportSecurity** assertions are included.

```
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
 <extensions>
 </extensions>

 <!--Uncomment this policy to use the UsernameForCertificateSecurity scenario-->
 <policy name="usernameTokenSecurity">
  <usernameForCertificateSecurity establishSecurityContext="true"
renewExpiredSecurityContext="true" requireSignatureConfirmation="false"
messageProtectionOrder="SignBeforeEncryptAndEncryptSignature"
requireDerivedKeys="true" ttlInSeconds="60">
   <serviceToken>
    <!-- WSE2 QuickStart Server Certificate -->
    <x509 storeLocation="LocalMachine" storeName="My"
findValue="CN=WSE2QuickStartServer" findType="FindBySubjectDistinguishedName" />
   </serviceToken>
   <protection>
    <request signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
    <response signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
    <fault signatureOptions="IncludeAddressing, IncludeTimestamp, IncludeSoapBody"
encryptBody="false" />
   </protection>
  </usernameForCertificateSecurity>
  <requireActionHeader />
 </policy>
```

(continued)

(continued)

```
 <!--Uncomment this policy to use the UsernameOverTransportSecurity scenario-->
 <!--<policy name="usernameTokenSecurity">
  <usernameOverTransportSecurity />
  <requireActionHeader />
 </policy>-->
</policies>
```

When you add a Web reference to the service from the client application, two proxies are generated for the Web service — one is a non-WSE 3.0 proxy and the other is WSE 3.0–enabled. In this guidance, Microsoft uses the WSE 3.0–enabled proxy class, which is defined as name + “Wse.” For example, if your Web service is named “MyService,” your WSE 3.0–enabled Web service proxy class name would be “MyServiceWse.”

The following code example provides an example of how to initialize an instance of a **UsernameToken** and to bind the appropriate policy defined in the preceding policy file to the Web service proxy. You can copy and insert this code into a new code module.

```
...
using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Security.Tokens;
...
try
{
  Service.ServiceWse proxy = new Service.ServiceWse();
  string userName = null;
  if (txtDomain.Text.Trim().Length > 0)
  {
   userName = String.Format(@"{0}\{1}", txtDomain.Text, txtUsername.Text);
  }
  else
  {
   userName = txtUsername.Text;
  }

  UsernameToken token = new UsernameToken(userName, txtPassword.Text,
PasswordOption.SendPlainText);

  proxy.SetClientCredential(token);

  proxy.SetPolicy("usernameTokenSecurity");

  Service.Product product = proxy.GetProductInformation(txtProduct.Text);

  lblResults.Text = String.Format(CultureInfo.InvariantCulture,
            "Product: {0}, Quantity {1}, Unit price {2}",
            product.Name, product.Quantity, product.UnitPrice);
```

(continued)

continued)

```
}
catch (Exception ex)
{
  lblResults.Text = ex.ToString();
}
...
```

As appropriate, replace the **Product** class and code that processes the response returned from the service used in the preceding code example for the object type returned by your service.

Configure the Service

You must perform the following steps to configure the service to enable WSE 3.0 extensions.

► **To enable a Visual Studio 2005 project to support WSE 3.0 SOAP extensions**

1. In Visual Studio 2005, right-click the application project and select **WSE Settings 3.0**.
2. On the **General** tab, select the **Enable Microsoft Web Services Enhancement SOAP Protocol Factory** check box, and click **OK**.

After you enable the service application to support WSE 3.0 SOAP extensions, you must enable policy support. If your application does not currently have a policy cache file, you can add one and enable policy support by performing the following steps.

► **To add policy support to a WSE 3.0-enabled Visual Studio 2005 project**

1. In Visual Studio 2005, right-click the application project and select **WSE Settings 3.0**.
2. On the **Policy** tab, select the **Enable Policy** check box. Selecting this check box adds the wse3policyCache.config file as the default name for the policy cache file.
3. Under **Edit Application Policy**, click **Add** and then type a policy friendly name for the new application policy, such as “usernameTokenSecurity.”
4. Click **OK** to start the WSE Security Settings Wizard, and then click **Next**.
5. On the **Authentication Settings** page, the wizard provides you with options to secure a service or a client. Select the **secure a service application** option button to configure the service.
6. The wizard also provides you with authentication method choices on the same page. Select **Username** and click **Next**.

7. On the **Users and Roles** page, you configure authorization based on the user name or roles associated with the user represented in the **UsernameToken**. by default, the **perform authorization** check box is cleared. If you want to perform authorization through the policy assertion, select the **perform authorization** check box, add users and roles as appropriate, and then click **Next**.
8. On the **Message Protection** page, you configure options for message protection. For transport layer security, select **None (rely on transport protection)** for the **Protection Order** to use the **usernameOverTransportSecurity** assertion.

   If you select any other protection option, the policy assertion will use **usernameForCertificateSecurity**. If you select any option under **Protection Order** other than **None (rely on transport protection)**, select the option for **Sign, Encrypt, Encrypt Signature**.

   By default, the **Enable WS-Security 1.1 Extensions** check box is selected. You must enable this option if you are using certificate security. For more information about these settings, see the "Implementation Strategy" section earlier in this pattern.
9. Click **Next**.
10. If you opted to use transport security by selecting the **None (rely on transport protection)** setting in step 8, skip this step. If you selected any other option, the wizard will prompt you to select a server X.509 certificate for the service on the **Server Certificate** page. Select the certificate that you want to use for the service, click **Next**.
11. On the **Create Security Settings** page, review your settings, and then click **Finish**.

After you complete these tasks, your client security policy should look similar to the following code example. Examples for both the **usernameForCertificateSecurity** and **usernameOverTransportSecurity** assertions are included.

```
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
 <!--Uncomment this policy to use the UsernameForCertificateSecurity scenario-->
 <policy name="usernameTokenSecurity">
  <authorization>
   <allow role="Users" />
   <deny role="*" />
  </authorization>
  <usernameForCertificateSecurity establishSecurityContext="true"
renewExpiredSecurityContext="true" requireSignatureConfirmation="false"
messageProtectionOrder="SignBeforeEncryptAndEncryptSignature"
requireDerivedKeys="true" ttlInSeconds="60">
   <serviceToken>
    <!-- WSE2 QuickStart Server Certificate -->
    <x509 storeLocation="LocalMachine" storeName="My"
findValue="CN=WSE2QuickStartServer" findType="FindBySubjectDistinguishedName" />
   </serviceToken>
   <protection>
    <request signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
```

(continued)

(continued)

```
    <response signatureOptions="IncludeAddressing, IncludeTimestamp,
IncludeSoapBody" encryptBody="true" />
    <fault signatureOptions="IncludeAddressing, IncludeTimestamp, IncludeSoapBody"
encryptBody="false" />
   </protection>
  </usernameForCertificateSecurity>
  <requireActionHeader />
 </policy>

 <!--Uncomment this policy to use the UsernameOverTransportSecurity scenario-->
 <!--<policy name="usernameTokenSecurity">
  <authorization>
   <allow role="Administrators" />
   <deny role="*" />
  </authorization>
  <usernameOverTransportSecurity />
  <requireActionHeader />
 </policy>-->
</policies>
```

The service's policy configuration is identical to the client's, except that the policy assertions for the service can contain an **<authorization>** assertion. This assertion allows users who belong to the Users group to call the service, and denies access to all other users. The roles that this policy assertion evaluates are obtained when the user is authenticated. The default **UsernameTokenManager** populates a security principal containing the user's roles in the Active Directory domain.

Note: WSE 3.0 uses the default **UsernameTokenManager** class to validate credentials presented in a **UsernameToken** by calling the Win32 **LogonUser** function. In Windows XP and Windows 2000, the service account, under which the Web application validating the credentials runs, can only call the **LogonUser** function if it has **Log on locally** permissions to the server hosting the service.

The following code example demonstrates how to apply the policy provided earlier when the service processes a request. You can copy and insert this code into a new code module.

```
using System;
using System.Web.Services;

using Microsoft.Web.Services3;
using Microsoft.Web.Services3.Security.Tokens;

using Microsoft.Practices.WSSP.WSE3.QuickStart.Common;
```

(continued)

(continued)

```
namespace
Microsoft.Practices.WSSP.WSE3.QuickStart.UsernameTokenWithWindows.Service
{
  /// <summary>
  /// This class represents a web service used to query products catalog, secured
with a UsernameToken
  /// </summary>
  [WebService(Namespace =
"http://schemas.microsoft.com/WSSP/WSE3/QuickStart/DirectAuthentication/2005-
10/UsernameTokenWithWindows.wsdl")]
  [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
  [Policy("usernameTokenSecurity")]
  public class Service : System.Web.Services.WebService
  {
    const string AdmistratorsRole = "Administrators";

    public Service()
    {
    }

    /// <summary>
    /// Returns some information about the specified product
    /// </summary>
    /// <param name="productName"></param>
    /// <returns></returns>
    [WebMethod]
    public Product GetProductInformation(string productName)
    {
      CheckPrincipalRoles();

      Product product = new Product();
      product.Name = productName;
      product.Quantity = 10;
      product.UnitPrice = 2.5M;
      return product;
    }

    /// <summary>
    /// Verifies if the user has permissions to execute this service
    /// </summary>
    private void CheckPrincipalRoles()
    {
      SecurityToken token = RequestSoapContext.Current.IdentityToken;
      bool isInRole = token.Principal.IsInRole(AdmistratorsRole);

      if (!isInRole)
      {
        throw new
UnauthorizedAccessException(string.Format(Resources.Messages.AuthorizationExceptio
n, AdmistratorsRole));
      }
    }
  }
}
```

In the preceding code example, the Web service applies the appropriate policy through the **Policy** attribute in the class declaration. Ensure that the value specified in the **Policy** attribute matches the name of your policy assertion that you want to use.

The **UnauthorizedAccessException** class uses a string from a resource file to provide a message for the exception. Alternatively, a simple string could be provided instead of accessing a resource file.

If you secure communication at the transport layer using the **usernameForCertificateSecurity** assertion, you must also install an X.509 certificate into the local machine certificate store where the service is hosted. Also, you must ensure that the service account under which the service is configured to run has read permissions to the certificate private key. You can do this by using the Certificates tool released with WSE 3.0. If you are running the service under the default service account for ASP.NET, you need to grant read permissions to that account. On Windows 2000 and Windows XP, the default account is ASPNET. On Windows Server 2003, the default account is the NETWORK SERVICE account.

When securing direct authentication using X.509 certificates either at the message layer or the transport layer, ensure that anonymous access is enabled for the virtual directory where the service is hosted in Internet Information Services (IIS) 6.0. Otherwise, the service may expect the client to authenticate at the transport layer and reject the client's attempts to authenticate at the message layer with a **UsernameToken**.

► **To enable a Anonymous Access on a virtual directory in IIS 6.0**

1. In IIS 6.0, right-click the virtual directory where the service is hosted, and then select **Properties**.
2. Click the **Directory Security** tab.
3. Under **Authentication and access control**, click **Edit**.
4. Ensure that the **Enable anonymous access** checkbox is selected, click **OK**, and then click **OK** again.

Resulting Context

This section describes some of the more significant benefits, liabilities, and security considerations of using this implementation pattern.

Note: The information in this section is not intended to be comprehensive. However, it does discuss many of the issues that are most commonly encountered for this pattern.

Benefits

The benefits of using the Implementing Direct Authentication with UsernameToken in WSE 3.0 pattern include the following:

- The pattern provides interoperable password-based authentication at the message layer.
- The pattern allows for flexibility to secure communication at either the message layer or the transport layer.
- The pattern enables flexible configuration for using different authentication services/identity stores to validate credentials presented in a **UsernameToken**.

Liabilities

The liabilities associated with the Implementing Direct Authentication with UsernameToken in WSE 3.0 pattern include the following:

- When using **UsernameTokens**, you can configure WSE 3.0 to prevent replay attacks by using a nonce and timestamp with a replay cache on the server through configuring the **<replayDetection>** element. For more information about this topic, see <replayDetection> Element. However, the replay cache is not shared across a server farm. One solution you can use to mitigate this issue is to create a replay cache that is shared across the server farm. If you are using the **usernameOverTransportSecurity** assertion, the method used to secure communication at the message layer (such as SSL) must provide message replay detection because the message is not signed. For more information about message replay detection, see Message Replay Detection and Implementing Message Replay Detection in WSE 3.0 in Chapter 5, "Service Boundary Protection Patterns."
- The **usernameForCertificateSecurity** assertion uses features that are introduced in WS-Security 1.1, which makes it incompatible with Web services implementing the WS-Security 1.0 specification.
- Implementing message layer security is likely to reduce the throughput and increase the latency of Web services, due to the overhead of the cryptographic operations that support canonicalization, XML signatures, and encryption. As part of your development process, you should identify performance objectives for your application and test the application against those objectives. For more information, see *Improving .NET Performance and Scalability*.

Security Considerations

Security considerations associated with the Implementing Direct Authentication with UsernameToken in WSE 3.0 pattern include the following:

- The password in a **UsernameToken** should always be encrypted, using either message layer security or transport layer security, such as SSL. This mitigates the threat of an eavesdropper obtaining credentials from the **UsernameToken**.
- If SSL is implemented between several intermediaries providing point-to-point security, the environment is vulnerable to man-in-the-middle and XML attacks.

Passwords are considered one of the weakest forms of identity used for authentication, but they are also the most common. As a result, it is important to understand threats and vulnerabilities associated with passwords. Passwords are often based on words and phrases that users can remember. This makes it easier to discover passwords through using brute force attacks that try thousands of common passwords and word combinations. You can mitigate this vulnerability by using complex passwords or password phrases, although if user passwords become too difficult to remember, users are likely to write them down.

Variants

The following variants describe alternate choices to Active Directory as an identity store, as discussed in the “Identity Store Options” section under the Implementation Strategy section earlier in this pattern. Both the database and directory service identity stores require a custom **UsernameTokenManager** class and an ASP.NET 2.0 membership provider that is configured for them.

Variant 1 — Using a Database as the Identity Store

Instead of validating credentials with an Active Directory domain controller as described in the base pattern, this variant describes how to configure the implementation to use a database as the identity store.

As previously stated in this pattern, whenever you use something other than Active Directory to manage user credentials, WSE 3.0 requires you to use a custom **UsernameTokenManager** class and an ASP.NET 2.0 membership provider that is configured for the service. For instructions and examples about how to create and configure a custom **UsernameTokenManager** class, see “Create a Custom UsernameTokenManager” at the end of this section.

To use a database as an ASP.NET 2.0 membership provider, you must configure the service to use a **SqlMembershipProvider**. For more details about how to configure a **SqlMembershipProvider**, see “Using the SQLMemberShipProvider” in How To: Use Membership in ASP.NET 2.0. After following these steps to configure the **SqlMembershipProvider** for your service, the configuration for your membership provider should look similar to the following service’s Web.config file.

```
...
<connectionStrings>
 <add name="MySqlConnection" connectionString="Data Source=MySqlServer;Initial
Catalog=aspnetdb;Integrated Security=SSPI;" />
</connectionStrings>
<system.web>
...
 <membership defaultProvider="SqlProvider" userIsOnlineTimeWindow="15">
 <providers>
  <clear />
  <add
  name="SqlProvider"
  type="System.Web.Security.SqlMembershipProvider"
  connectionStringName="MySqlConnection"
  applicationName="MyApplication"
  enablePasswordRetrieval="false"
  enablePasswordReset="true"
  requiresQuestionAndAnswer="true"
  requiresUniqueEmail="true"
  passwordFormat="Hashed" />
 </providers>
 </membership>
...
```

Variant 2 — Using an LDAP Directory Service as the Identity Store

Instead of validating credentials with an Active Directory domain controller as described in the base pattern, this variant describes how to configure the implementation to use a an LDAP-enabled directory service as an identity store.

As previously stated in this pattern, whenever you use something other than Active Directory to manage user credentials, WSE 3.0 requires you to use a custom **UsernameTokenManager** class and an ASP.NET 2.0 membership provider that is configured for the service. For instructions and examples about how to create and configure a custom **UsernameTokenManager**, see the end of this section.

To use Active Directory through LDAP or ADAM joined to an Active Directory instance, you must configure the service to use an **ActiveDirectoryMembershipProvider**. For more details about how to configure an ASP.NET 2.0 membership provider, see How To: Use Membership in ASP.NET 2.0.

After following these steps to configure the **ActiveDirectoryMembershipProvider** for your service, the configuration for your membership provider should look similar to the following service's Web.config file. The connection string in this code example has been substituted for the one that is required to connect your directory service. An ellipsis (...) represents sections of the configuration file that have been omitted for brevity.

```
<connectionStrings>
 <add name="ADConnectionString"
 connectionString=
 "LDAP://domain.testing.com/CN=Users,DC=domain,DC=testing,DC=com" />
</connectionStrings>
...
<system.web>
 ...
 <membership defaultProvider="MembershipADProvider">
 <providers>
 <add
  name="MembershipADProvider"
  type="System.Web.Security.ActiveDirectoryMembershipProvider, System.Web,
   Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
    connectionStringName="ADConnectionString"
    connectionUsername="<domainName>\directoryservice"
    connectionPassword="password"/>
 </providers>
 </membership>
 ...
</system.web>
...
```

Different directory services may require different formatting of a user name when credentials are validated. For example, ADAM requires a format of *username@domain*. The client can do this when it creates a **UsernameToken** instance. In which case, the service should check the formatting in the **CustomUsernameTokenManager** before the credentials are validated against the directory service. The formatting can also be done directly in the **CustomUsernameTokenManager** before the credentials are validated against the directory service, with the expectation that the client will send the user name without a specified domain, and that the **CustomUsernameTokenManager** will add the domain name with proper formatting.

If you use an LDAP-enabled directory service other than Active Directory or ADAM to validate credentials, you may need to create a custom membership provider. For more details on how to build custom ASP.NET 2.0 providers, see Building Custom Providers for ASP.NET 2.0 Membership. Also, depending how you store and retrieve account roles in your directory service, you may need to implement a custom **RoleProvider**. For example, if you use an LDAP schema for user roles that is not supported through **ActiveDirectoryMembershipProvider**, you will need to implement a custom **RoleProvider** to retrieve roles for your users.

In a custom **RoleProvider** class, you need to retrieve the user roles from the directory service by overriding the **GetRolesForUser()** method. The code to retrieve user roles from the directory service would look like the following example.

```
public override string[] GetRolesForUser(string username)
    {
      using (DirectoryEntry rootEntry = new DirectoryEntry(this.connectionString))
      {
        rootEntry.Username = this.username;
        rootEntry.Password = this.password;

        rootEntry.AuthenticationType = AuthenticationTypes.None;
        rootEntry.RefreshCache();

        //Search the user in the directory service
        using (DirectorySearcher searcher = new DirectorySearcher(rootEntry))
        {
          searcher.PropertiesToLoad.Add("memberOf");
          searcher.PropertiesToLoad.Add(this.usernameAttribute);

          searcher.Filter = String.Format("(&(objectClass=user)({0}={1}))",
this.usernameAttribute, username);
          SearchResult result = searcher.FindOne();
          DirectoryEntry userEntry = result.GetDirectoryEntry();

          string[] roles = null;

          PropertyValueCollection property = userEntry.Properties["memberOf"];
          if (property.Value is Array)
          {
            Array values = (Array)property.Value;
            roles = new string[values.Length];
            values.CopyTo(roles, 0);
          }
          else if (property.Value is string)
          {
            roles = new string[1];
            roles[0] = (string)property.Value;
          }
          return roles;
        }
      }
    }
```

Create a Custom UsernameTokenManager

When validating credentials against a database or an LDAP-enabled directory service, you need to create and implement a custom **UsernameTokenManager** class. This is not necessary if you are validating credentials against an Active Directory domain.

To implement a custom **UsernameTokenManager** for either a database or a directory service, you must derive a custom class from the **UsernameTokenManager** and configure the service to use the custom class in its Web.config file.

The easiest way to add an entry for a custom **UsernameTokenManager** in the service's Web.config file is by using the WSE 3.0 Settings tool. To add a custom **UsernameTokenManager** entry, right-click the service project, select **WSE Settings 3.0**, and then on the **Security** tab, type the security token manager's information.

The following configuration example provides an example of what a custom **UsernameTokenManager** in the service's Web.config file might look like after you have added it through the WSE 3.0 Settings tool. An ellipsis (...) indicates configuration sections that have been omitted for brevity.

```
<configuration>
...
  <microsoft.web.services3>
   ...
   <securityTokenManager>
     <add localName="UsernameToken"
type="Microsoft.Practices.WSSP.WSE3.QuickStart.UsernameTokenWithDatabase.Service.C
ustomUsernameTokenManager" namespace="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"/>        ...
    </securityTokenManager>
      <Microsoft.web.services3>
   ...
</configuration>
```

In the previous example, the **type** attribute represents the fully qualified name of the custom **UsernameTokenManager** class. Set this attribute based on the namespace and class name that you chose for your custom **UsernameTokenManager** class.

The following code example provides an example of a custom **UsernameTokenManager** class.

```
using System;
using System.Xml;
using System.Security.Permissions;
using System.Web.Security;
using System.Security.Principal;

using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Security.Tokens;
```

(continued)

(continued)

```
namespace
Microsoft.Practices.WSSP.WSE3.QuickStart.UsernameTokenWithDatabase.Service
{
    /// <summary>
    /// By implementing UsernameTokenManager we can verify the signature
    /// on messages received.
    /// </summary>
    [SecurityPermissionAttribute(SecurityAction.Demand,
Flags=SecurityPermissionFlag.UnmanagedCode)]
    public class CustomUsernameTokenManager : UsernameTokenManager
    {
        /// <summary>
        /// Constructs an instance of this security token manager.
        /// </summary>
        public CustomUsernameTokenManager()
        {
        }

        /// <summary>
        /// Constructs an instance of this security token manager.
        /// </summary>
        /// <param name="nodes">An XmlNodeList containing XML elements from a
configuration file.</param>
        public CustomUsernameTokenManager(XmlNodeList nodes)
            : base(nodes)
        {
        }

        /// <summary>
        /// Returns the password or password equivalent for the username provided.
    /// Adds a principal to the token with user's roles.
        /// </summary>
        /// <param name="token">The username token</param>
        /// <returns>The password (or password equivalent) for the
username</returns>
        protected override string AuthenticateToken( UsernameToken token )
        {
      bool validCredentials = Membership.ValidateUser(token.Username,
token.Password);
      if (!validCredentials)
      {
        throw new ApplicationException(Resources.Messages.AuthenticationError);
      }

      GenericIdentity identity = new GenericIdentity(token.Username);
      GenericPrincipal principal = new GenericPrincipal(identity,
Roles.GetRolesForUser(token.Username));
      token.Principal = principal;

      return token.Password;
        }

    }
```