Custom WSE 3.0 Policy Assertions for Signing and Encrypting SOAP Messages with Username Tokens

**[Feodor Fitsner](http://www.codeproject.com/script/Membership/View.aspx?mid=2432832)**, 8 Nov 2005 http://www.codeproject.com/App_Themes/CodeProject/Img/read32.png 307.4Khttp://www.codeproject.com/App_Themes/CodeProject/Img/download32.png 1.8Khttp://www.codeproject.com/App_Themes/CodeProject/Img/bookmark32.png 83

|  |  |
| --- | --- |
| http://codeproject.cachefly.net/script/Ratings/Images/stars-fill-lg.png  http://codeproject.cachefly.net/script/Ratings/Images/stars-empty-lg.png | 4.88 (39 votes) |
|  | |  |  |  | | --- | --- | --- | | Rate: | [vote 1](http://www.codeproject.com/Articles/12189/Custom-WSE-Policy-Assertions-for-Signing-and-E" \o "vote 1)[vote 2](http://www.codeproject.com/Articles/12189/Custom-WSE-Policy-Assertions-for-Signing-and-E" \o "vote 2)[vote 3](http://www.codeproject.com/Articles/12189/Custom-WSE-Policy-Assertions-for-Signing-and-E" \o "vote 3)[vote 4](http://www.codeproject.com/Articles/12189/Custom-WSE-Policy-Assertions-for-Signing-and-E" \o "vote 4)[vote 5](http://www.codeproject.com/Articles/12189/Custom-WSE-Policy-Assertions-for-Signing-and-E) |  | |

An article describing the implementation of custom WSE 3.0 policy assertions for signing and encrypting SOAP messages with Username tokens.

* [**Download source - 24.3 Kb**](http://www.codeproject.com/KB/XML/WSE30UsernameAssertion/WSE30UsernameAssertion_src.zip)

Introduction

This article is an attempt to share my WSE 3.0 investigations and give a working real-world example of securing web services.

The WSE 3.0 library provides a number of “Turnkey” security mechanisms based on X.509 certificates, Username tokens over SSL, or Kerberos tickets. All that stuff like X.509, SSL, Kerberos tickets is good in laboratory conditions or for enterprise level applications where the hosting environment (or deployment target) is under full control. Relatively simple applications require a much simpler implementation (but not less secure Wink | ;-) ), so we used to implement a security based on Username tokens derived from username and password. This technique doesn’t demand any specific requirements to the hosting environment.

This article gives answers to the following questions:

* How to implement a custom WSE 3.0 policy assertion class
* How to read assertion configuration settings from the policy file
* How to implement output client and input service SOAP filters for use within a custom policy assertion
* How to sign a SOAP message with Username token
* How to encrypt the message body with Username token
* How to encrypt custom SOAP headers with Username token
* How to implement Username token manager for verifying and decrypting client SOAP requests
* How to configure both client and server applications to enable security

Background

Of course, this article requires a good understanding of SOAP web services and some previous experience with the WSE 2.0 library. However, the code from this article is self-containing and can be used in your applications with minor or no changes.

Before including this code to your production application, take a look at the [WSE development center](http://msdn.microsoft.com/webservices/webservices/building/wse/default.aspx) to grab more information about what is this and how it can be used.

The WSE 3.0 October CTP library (the latest WSE release) that is required to run the samples below can be downloaded from [here](http://www.microsoft.com/downloads/details.aspx?FamilyId=6BAFE8A6-CDC9-4AE6-9625-E6260CCDEF24&displaylang=en). Please note, the library is still in beta, but it works perfectly with the .NET 2.0 RTM. Microsoft promises that WSE will be released closer to the official VS.NET 2005 go-live date which is currently November 7, 2005.

So, don’t waste your time and start learning this exciting new technology right now and migrating your almost legacy Wink | ;-) WSE 2.0 code!

Using the code

Before describing the code snippets I’d like to say some words about the basic idea of the overall WSE 3.0 architecture. The first sight on WSE 3.0 leaves a feeling that it will be so hard and long to get it work for you and there is not any desire for implementing it. But after spending some time studying and experimenting with the code you start realizing that WSE 3.0 is a huge step forward in comparison to the previous WSE 2.0 allowing writing less code and giving more flexibility in configuration.

WSE 3.0 is based on policies. Each individual policy can be applied to either a web service class or a web service proxy (client) class. A policy is a set of assertions. I think “assertion” is quite an unsuitable word here, but it occurs that assertion is just a factory class for creating input/output SOAP filters that will be injected in the SOAP processing pipeline. All policies can be declared in a separate “policy cache” file and then can be assigned to the web service class with the help of an attribute or to the proxy class programmatically. A policy cache file is an XML document including:

1. the set of so-named “extensions” that are simply the list of named assertions with their corresponding type names and
2. policies themselves each of which is a list of extensions (assertions) with corresponding optional configuration elements inside.

Well, let’s start writing the code! The first step is to implement a custom assertion which will be applied to the web service proxy. The primary function of assertion is constructing four SOAP filters:

1. Client output filter (working with the SOAP request issued by the client)
2. Client input filter (working with the SOAP response received from the service)
3. Service input filter (working with the SOAP request received from the client)
4. Service output filter (working with the SOAP response issued by the web service)

The second optional function of assertion is parsing the assertion configuration elements from the policy cache file. This step is required if the assertion is applied declaratively through the policy cache file.

The code for our client Username assertion:

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

public class UsernameClientAssertion : SecurityPolicyAssertion

{

private string username;

private string password;

public UsernameClientAssertion(string username, string password)

{

this.username = username;

this.password = password;

}

public override SoapFilter

CreateClientOutputFilter(FilterCreationContext context)

{

return new ClientOutputFilter(this, context);

}

public override SoapFilter

CreateClientInputFilter(FilterCreationContext context)

{

*// we don't provide ClientInputFilter*

return null;

}

public override SoapFilter

CreateServiceInputFilter(FilterCreationContext context)

{

*// we don't provide any processing for web service side*

return null;

}

public override SoapFilter

CreateServiceOutputFilter(FilterCreationContext context)

{

*// we don't provide any processing for web service side*

return null;

}

...

}

As our client assertion requires username and password, it will be assigned to the proxy imperatively in the code and thus, the assertion has a constructor for passing credentials. The credentials are just stored in the private fields and then can be accessed in output filter.

Our client assertion provides only one “ClientOutputFilter”. The primary purposes of the client output filter are:

1. Create username token.
2. Sign SOAP message with username token.
3. Encrypt SOAP body.
4. Encrypt SOAP custom headers marked with a special namespace.

The code of the client output filter:

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

class ClientOutputFilter : SendSecurityFilter

{

UsernameClientAssertion parentAssertion;

FilterCreationContext filterContext;

public ClientOutputFilter(UsernameClientAssertion parentAssertion,

FilterCreationContext filterContext)

: base(parentAssertion.ServiceActor, false, parentAssertion.ClientActor)

{

this.parentAssertion = parentAssertion;

this.filterContext = filterContext;

}

public override void SecureMessage(SoapEnvelope envelope, Security security)

{

UsernameToken userToken = new UsernameToken(

parentAssertion.username,

parentAssertion.password,

PasswordOption.SendNone);

*// we don't send password over network*

*// but we just use username/password to sign/encrypt message*

*// Add the token to the SOAP header.*

security.Tokens.Add(userToken);

*// Sign the SOAP message by using the UsernameToken.*

MessageSignature sig = new MessageSignature(userToken);

security.Elements.Add(sig);

*// encrypt BODY*

EncryptedData data = new EncryptedData(userToken);

*// encrypt custom headers*

for (int index = 0; index <

envelope.Header.ChildNodes.Count; index++)

{

XmlElement child =

envelope.Header.ChildNodes[index] as XmlElement;

*// find all SecureSoapHeader headers*

*// marked with a special attribute*

if (child != null && child.NamespaceURI ==

"http://company.com/samples/wse/")

{

*// create ID attribute for referencing purposes*

string id = Guid.NewGuid().ToString();

child.SetAttribute("Id", "http://docs.oasis-" +

"open.org/wss/2004/01/oasis-200401-" +

"wss-wssecurity-utility-1.0.xsd", id);

*// Create an encryption reference for the custom SOAP header.*

data.AddReference(new EncryptionReference("#" + id));

}

}

*// add ancrypted data to the security context*

security.Elements.Add(data);

}

}

It is pretty suitable to define assertion filters as inner classes, because they would have simple names and they can see parent assertion fields.

The base SOAP header class that all your headers should be derived from to be secured, looks like the following:

Hide   Copy Code

/// *<summary>*

/// *This is base class for all custom SOAP headers*

/// *that should be encrypted in the response.*

/// *</summary>*

public class SecureSoapHeader : SoapHeader

{

/// *<summary>*

/// *This property is just a flag telling us*

/// *that this SOAP header should be encrypted.*

/// *</summary>*

[XmlAttribute("SecureHeader",

Namespace="http://company.com/samples/wse/")]

public bool SecureHeader;

}

The code encrypting custom SOAP headers is quite unique here. I was Googling the internet trying to find information about this subject and had no success. Then I decided to write my own solution. I spent a certain time to realize how it can be implemented using standard functionality. The code is based on the sample: how to sign custom SOAP headers, given in the WSE 3.0 documentation. When playing with signing custom headers, I noticed at least two inconsistencies in the provided sample:

1. Within the custom SOAP header you can’t declare a custom “Id” property mapped to the XML “Id” attribute with [this](http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd) namespace. Certainly, you can do this, but it doesn’t make sense, because when the initial SOAP message is serialized, this system namespace is re-declared with a random prefix, not the “wsu” prefix as expected. The answer to this problem is that if you want to add a service “wsu:id” attribute to your custom header, you should do this after the “wsu” namespace has been declared.
2. The second inconsistency is that when you are creating SignatureReference orEncryptionReference, to add it to the respective collection you should specify the reference URI in the form “#[id]” (where [id] is the unique identifier of the node – GUID), not “#Id:[id]” like the documentation suggests.

That’s all the required code for the client side. The code for how to specify this assertion for the client will be given below. For now let’s turn to the web service side and consider the service-side assertion. The code forUsernameServiceAssertion is:

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

public class UsernameServiceAssertion : SecurityPolicyAssertion

{

public UsernameServiceAssertion()

{

}

public override SoapFilter

CreateClientOutputFilter(FilterCreationContext context)

{

*// we don't provide any processing for client side*

return null;

}

public override SoapFilter

CreateClientInputFilter(FilterCreationContext context)

{

*// we don't provide any processing for client side*

return null;

}

public override SoapFilter

CreateServiceInputFilter(FilterCreationContext context)

{

return new ServiceInputFilter(this, context);

}

public override SoapFilter

CreateServiceOutputFilter(FilterCreationContext context)

{

*// we don't provide ServiceOutputFilter*

return null;

}

public override void ReadXml(XmlReader reader,

IDictionary<string, Type> extensions)

{

if (reader == null)

throw new ArgumentNullException("reader");

if (extensions == null)

throw new ArgumentNullException("extensions");

*// determine the name of the extension*

string tagName = null;

foreach (string extName in extensions.Keys)

{

if (extensions[extName] ==

typeof(UsernameServiceAssertion))

{

tagName = extName;

break;

}

}

*// read the first element (maybe empty)*

reader.ReadStartElement(tagName);

}

public override void WriteXml(XmlWriter writer)

{

*// Typically this is not needed for custom policies*

}

...

}

Note the difference from the client assertion: we return only the service input filter and add two additional methods for working with the assertion configuration elements. The ReadXml method in our example is just reading an empty XML element declaring an assertion in the policy. Also, the documentation doesn’t mention these two methods, but when you apply the assertion declaratively in the policy cache file, it will fail if theReadXml method is not overridden.

The main purpose of the service input filter is checking which SOAP elements are signed and encrypted (I’ve removed the code checking encrypted elements to make the example simpler).

The code for the ServiceInputFilter class:

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

public class ServiceInputFilter : ReceiveSecurityFilter

{

UsernameServiceAssertion parentAssertion;

FilterCreationContext filterContext;

public ServiceInputFilter(UsernameServiceAssertion parentAssertion,

FilterCreationContext filterContext)

: base(parentAssertion.ServiceActor, false,

parentAssertion.ClientActor)

{

this.parentAssertion = parentAssertion;

this.filterContext = filterContext;

}

public override void ValidateMessageSecurity(SoapEnvelope envelope,

Security security)

{

bool IsSigned = false;

if (security != null)

{

foreach (ISecurityElement element in security.Elements)

{

if (element is MessageSignature)

{

*// The given context contains a Signature element.*

MessageSignature sign = element as MessageSignature;

if (CheckSignature(envelope, security, sign))

{

*// The SOAP message is signed.*

if (sign.SigningToken is UsernameToken)

{

*// The SOAP message is signed*

*// with a UsernameToken.*

IsSigned = true;

}

}

}

}

}

if (!IsSigned)

throw new SecurityFault("Message did" +

" not meet security requirements.");

}

private bool CheckSignature(SoapEnvelope envelope,

Security security, MessageSignature signature)

{

*//*

*// Now verify which parts of the message were actually signed.*

*//*

SignatureOptions actualOptions = signature.SignatureOptions;

SignatureOptions expectedOptions = SignatureOptions.IncludeSoapBody;

if (security != null && security.Timestamp != null)

expectedOptions |= SignatureOptions.IncludeTimestamp;

*//*

*// The <Action> and <To> are required addressing elements.*

*//*

expectedOptions |= SignatureOptions.IncludeAction;

expectedOptions |= SignatureOptions.IncludeTo;

if (envelope.Context.Addressing.FaultTo != null &&

envelope.Context.Addressing.FaultTo.TargetElement != null)

expectedOptions |= SignatureOptions.IncludeFaultTo;

if (envelope.Context.Addressing.From != null &&

envelope.Context.Addressing.From.TargetElement != null)

expectedOptions |= SignatureOptions.IncludeFrom;

if (envelope.Context.Addressing.MessageID != null &&

envelope.Context.Addressing.MessageID.TargetElement != null)

expectedOptions |= SignatureOptions.IncludeMessageId;

if (envelope.Context.Addressing.RelatesTo != null &&

envelope.Context.Addressing.RelatesTo.TargetElement != null)

expectedOptions |= SignatureOptions.IncludeRelatesTo;

if (envelope.Context.Addressing.ReplyTo != null &&

envelope.Context.Addressing.ReplyTo.TargetElement != null)

expectedOptions |= SignatureOptions.IncludeReplyTo;

*//*

*// Check if the all the expected options are the present.*

*//*

return ((expectedOptions & actualOptions) == expectedOptions);

}

}

The filter class is also implemented as an inner class for UsernameServiceAssertion.

In order to verify username tokens on the service side, you should provide a custom implementation of the Username token manager class. For our example, it may look as follows:

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

public class ServiceUsernameTokenManager : UsernameTokenManager

{

/// *<summary>*

/// *Constructs an instance of this security token manager.*

/// *</summary>*

public ServiceUsernameTokenManager()

{

}

/// *<summary>*

/// *Constructs an instance of this security token manager.*

/// *</summary>*

/// *<param name="nodes">An XmlNodeList containing*

/// *XML elements from a configuration file.</param>*

public ServiceUsernameTokenManager(XmlNodeList nodes)

: base(nodes)

{

}

/// *<summary>*

/// *Returns the password or password equivalent for the username provided.*

/// *</summary>*

/// *<param name="token">The username token</param>*

/// *<returns>The password (or password equivalent) for the username</returns>*

protected override string AuthenticateToken(UsernameToken token)

{

string username = token.Username;

*// it's up to you where you will get a password for some user*

*// you may:*

*// 1) get the password hash from web.config or system registry*

*// if you are implementing per-server security*

*// 2) get the password from the database or XML file for the given user name*

*// for example purposes we just return a reversed value of username*

char[] ch = username.ToCharArray();

Array.Reverse(ch);

return new String(ch);

}

}

The goal of the AuthenticateToken method is to return passwords corresponding to usernames.

Now let’s see how to use all the logic above to secure the communication between the client and the service.

The first step is enabling WSE 3.0 support for both the client and the service in the *app.config* and the *web.config*files respectively:

1. Add WSE configuration section handler.
2. Add the <soapServerProtocolFactory> element for WSE in the <webServices> element (service only).
3. Add the WSE configuration section.
4. Specify the custom security token manager class (service only).
5. Specify the name of the policy cache file (service only and possibly client, but not in our example).

For more details on how to enable WSE support, you may inspect the *app.config* and the *web.config* in the attached article sample.

The second step is creating a policy cache file for use within the web service. For our example, it may look as the follows:

Hide   Copy Code

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

<extensions>

<extension name="usernameAssertion"

type="UsernameAssertionLibrary.UsernameServiceAssertion,

UsernameAssertionLibrary" />

</extensions>

<policy name="ServerPolicy">

<usernameAssertion />

</policy>

</policies>

The third step is applying a policy to the web service with the “Policy” attribute as follows:

Hide   Copy Code

...

using Microsoft.Web.Services3;

[WebService(Namespace = "http://company.com/samples/wse/")]

[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1\_1)]

[Policy("ServerPolicy")]

*// we define policy on service level,*

*// so each service method will be covered by it*

public class Service : System.Web.Services.WebService

*// look, you don't need to inherit*

*// your service from some custom class*

{

...

The forth step is applying a policy to the web service proxy:

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

*// create web service proxy*

*// NOTE!!! When updating web reference in Visual Studio,*

*// don't forget to change its base class*

*// to Microsoft.Web.Services3.WebServicesClientProtocol then*

WseSample.Service srv = new WseSample.Service();

*// create custom SOAP header and assign it to web service*

WseSample.BankAccountSettings settings =

new WseSample.BankAccountSettings();

settings.PinCode = "1111";

srv.BankAccountSettingsValue = settings;

*// create custom policy assertion and assign it to proxy*

*// for password we just use reversed username*

*// it's important, because UsernameTokenManager*

*// on the service side applies the same logic*

*// when looking for user password*

UsernameClientAssertion assert =

new UsernameClientAssertion("admin", "nimda");

*// create policy*

Policy policy = new Policy();

policy.Assertions.Add(assert);

*// and set it to web service*

srv.SetPolicy(policy);

*// invoke web service method*

bool valid = srv.CheckAccountStatus("123456");

That’s all! Now all calls to your web service are authenticated and all sensitive request information is encrypted! You may verify this by checking the *InputTrace.webinfo* and *OutputTrace.webinfo* trace files for both the service and the client.

Conclusion

I’d like to give some ideas for how the sample from the article can be improved:

1. You may add authorization logic to your web service. Just add the code for creating custom IPrincipalin UsernameTokenManager and role checks inside web methods.
2. Signing and encrypting the SOAP response. You may implement additional service output and client input filters to sign/encrypt SOAP responses and verify them on the client side.
3. Compression assertion. You may write a custom assertion using System.IO.Compression classes to compress/decompress SOAP requests/responses.
4. You may implement assertions to make authentication/authorization work with SOAP clients written in other languages as Perl, PHP, and VBScript (SOAP Toolkit).

History

* 11/06/2005 – The initial version of the article.

License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found [here](http://www.codeproject.com/info/Licenses.aspx)

Share