Using SAML security tokens with Microsoft Web Services Enhancements

A standards-based approach enabled by Tivoli Federated Identity Manager

Craig Forster
Neil Readshaw
April 29, 2008

Microsoft® Web Services Enhancements (WSE) is a framework for developing secure, interoperable Web services for the Microsoft .NET platform. WSE supports standard security token types such as Username, Kerberos and X.509 certificate tokens. One widely used security token type not supported by WSE is the Security Assertion Markup Language (SAML). This article will demonstrate an architecture and implementation capable of integrating WSE and SAML using Tivoli® Federated Identity Manager (TFIM).

Introduction

Applications based on Service-Oriented Architecture (SOA) are becoming widespread as organizations look to maximize reuse of existing IT assets. This reuse is commonly achieved by constructing loosely coupled, composite applications from application logic exposed as services. Ensuring that each service component is aware of the identity of the original service requester is crucial to correct operation of many applications based on SOA. Without identity propagation, authorization decisions and audit data are often meaningless.

Standards for Web services security are required to ensure interoperability in heterogeneous environments. The WS-Security standard and its dependents, such as a WS-Trust and WS-SecurityPolicy, are important standards in this area.

Identity propagation scenarios using the Security Assertion Markup Language (SAML) security token type are commonplace. SAML security tokens provide a representation of identity that is:

- A standard (through OASIS)
- Platform neutral
- Widely implemented
- XML format
- Capable of including custom, extended attributes
These capabilities make SAML security tokens an ideal candidate for representing identity in Web services, particularly those that cross organizational or functional boundaries. In this article, Tivoli Federated Identity Manager (TFIM) will be used to enable applications built using the Microsoft ASP.NET application server and Web Services Enhancements (WSE), a capability that ASP.NET does not natively possess.

The remainder of this introductory section provides a brief introduction to the key technologies described in this article.

**Microsoft Web Services Enhancements**

**Microsoft Web Services Enhancements** is a supported extension to the Microsoft .NET framework to provide advanced capabilities in the area of Web services. While the .NET framework provides some Web services capabilities, including basic security, WSE contains the Microsoft implementation of key industry standards in Web services such as WS-Security and WS-Trust. WSE provides integration with the Microsoft Visual Studio development environment and is wire compatible with the newer Microsoft Windows Communications Foundation (WCF).

Web services are secured in WSE through policy assertions. A number of built-in assertions (turnkey assertions) are provided. Examples of turnkey assertions are:

- usernameOverTransportSecurity - Authenticate using a UsernameToken, but don't encrypt or sign the messages
- usernameForCertificateSecurity - Client is authenticated using a UsernameToken, but server authentication is provided using an X.509 certificate. Message level security is performed
- kerberosSecurity - Both client and server are authenticated using Kerberos tokens, and message-level security is performed

Policy is commonly configured declaratively using policy configuration files that contain references to the assemblies containing the policy assertion class as well as any configuration for each assertion. The policy configuration file is referenced in the application's configuration file (that is, Web.config or app.config), and the application itself specifies the name of the policy configuration to be used. Listing 2 contains an example of such a policy configuration file.

**Listing 2. An example of a WSE 3.0 policy file**

```xml
<extensions>
</extensions>
<policy name="ServicePolicy">
  <usernameOverTransportSecurity />
  <requireActionHeader />
</policy>
</policies>
```
SAML

**SAML** is the common name for the OASIS organization's Security Services effort. The latest version of the SAML specifications is 2.0. Previous versions were 1.0 and 1.1. Originally for Security Assertion Markup Language, the specifications have come to represent much more. SAML specifications describe:

- **Assertions**, which specify how identities are represented.
- **Protocols**, which represent a sequence of XML messages designed to achieve a single goal.
- **Bindings**, which describe how protocol messages are transported over a lower-level protocol such as HTTP.
- **Profiles**, which combine a number of bindings to describe a solution for a use case.

In the SOA identity propagation context, the SAML assertion is most important. The anatomy of a SAML assertion is shown in Figure 1.

**Figure 1. Anatomy of a SAML assertion**

As mentioned above, a SAML assertion is an XML document. An example SAML 2.0 assertion is shown in Listing 1.

**Listing 1. SAML 2.0 assertion example**

```xml
<saml:Assertion ID="Assertion-uuid4cb533c2-0114-158f-a641-a2d10441b2a0"
    IssueInstant="2007-08-09T22:18:37Z"
    Version="2.0"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
    xmlns:wsse="http://oasis-open.org/wss/2004/01/oasis-wss-wssecurity-secext-1.0.xsd"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <saml:Issuer Format="urn:oasis:names:tc:SAML:2.0:nameid-format:entity">
    Acme
  </saml:Issuer>
  <ds:Signature Id="uuid4cb538fe-0114-17bf-9ca4-a2d10441b2a0">
    <ds:SignedInfo>
      <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
      <ds:SignatureValue/>
      <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
      <ds:KeyInfo>
        <ds:SubjectKeyInfo>
          <ds:KeyFormatInfo Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <ds:KeyValue>
            <ds:RSAPublicKey>
              <ds:Modulus>...
              <ds:Exponent>...
            </ds:RSAPublicKey>
          </ds:KeyValue>
        </ds:SubjectKeyInfo>
      </ds:KeyInfo>
    </ds:SignedInfo>
  </ds:Signature>
</saml:Assertion>
```
Using SAML security tokens with Microsoft Web Services Enhancements

<ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
<ds:Reference URI="#Assertion-uuid4cb533c2-8114-158f-a641-a2d10441b2a0">
<ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
<ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
<xc14n:InclusiveNamespaces PrefixList="ds saml" xmlns:xc14n="http://www.w3.org/2001/10/xml-exc-c14n#"/>
</ds:Transform>
<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
<ds:DigestValue>1m+7Q3VXjgP7D0QJo6g+CQ8QYI=</ds:DigestValue>
</ds:Reference>
<ds:SignatureValue>...</ds:SignatureValue>
<ds:KeyInfo>
<ds:X509Data>
<ds:X509Certificate>...</ds:X509Certificate>
</ds:X509Data>
</ds:KeyInfo>
<ds:Signature>
saml:Subject>
<saml:NameID Format="urn:oasis:names:tc:SAML:2.0:nameid-format:unspecified">
testuser
</saml:NameID>
</saml:Subject>
<saml:AudienceRestriction>
<saml:Audience>https://fw.acme.com/services/CreditCheck</saml:Audience>
</saml:AudienceRestriction>
</saml:Conditions>
<saml:AuthnContext>
<saml:AuthnContextClassRef>
urn:oasis:names:tc:SAML:2.0:ac:classes:Password
</saml:AuthnContextClassRef>
</saml:AuthnContext>
</saml:AuthnStatement>
<saml:AttributeStatement>
<saml:Attribute Name="FirstName">
<saml:AttributeValue xsi:type="xs:string">
http://www.w3.org/2001/XMLSchema
http://www.w3.org/2001/XMLSchema-instance">
John
</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute Name="Surname">
<saml:AttributeValue xsi:type="xs:string">
http://www.w3.org/2001/XMLSchema
http://www.w3.org/2001/XMLSchema-instance">
Citizen
</saml:AttributeValue>
</saml:Attribute>
</saml:AttributeStatement>
</saml:Assertion>

Tivoli Federated Identity Manager

Tivoli Federated Identity Manager (TFIM) is the IBM solution for identity propagation in a Service-Oriented Architecture (SOA). The core component of TFIM that enables identity propagation in SOA is the security token service (STS). The STS supports the WS-Trust specification (see Related topics), which provides a standards-based mechanism for SOA infrastructure to access an identity service to validate, transform and issue security tokens representing identities. The interaction between a SOA component and the TFIM STS is shown in Figure 2.
In the TFIM STS, different identity propagation configurations can be specified. These configurations are called *module chains*. Incoming requests to the STS are matched to trust module chains in TFIM based on the values of the AppliesTo, Issuer and TokenType parameters. In this way, the same logical STS instance can support a variety of different identity requirements.

The TFIM STS can operate with a number of different security token types, including:

- SAML 1.0
- SAML 1.1
- SAML 2.0
- Username
- Kerberos
- X.509 certificate
- Lightweight Third Party Authentication (LTPA)
- SAP

Additionally, new security token modules can be developed for custom security tokens if required.

**Tivoli Access Manager integration with Microsoft .NET**

The Tivoli Access Manager integration with Microsoft .NET (AMNET) is a supported solution available from the IBM Support Site. In a Microsoft .NET environment, AMNET provides:

- Access to the Tivoli Access Manager (TAM) Authorization API
- Access to the TAM Administration API
- Evaluation of role-based access control decisions by TAM
- A Web single sign-on solution for ASP.NET environments

**Patterns for using SAML security tokens with WSE**

In this section, conceptual architectures are shown for using SAML security tokens with Microsoft WSE. These are general patterns, not specific to use of SAML security tokens or Microsoft WSE.
They are intended to illustrate repeatable approaches to solving identity propagation challenges in SOA, and provide the motivation for the next section, which describes the prototype.

**Service requester pattern**

The *service requester* pattern recognizes the need for consumers of a service to send a service "what it expects". This pattern represents the authenticated identity in the service component in an identity token and uses an STS to transform the authenticated identity to an identity token suitable for sending in the service request (Figure 3). The most common use of this pattern is to prepare an identity token that contains an identity in the domain of the receiving service component and in the format expected by the receiver of the service request, whether it is an intermediary such as an Enterprise Service Bus or a service implementation itself.

For example, an ASP.NET application can act as a Web service client to an application published on the WebSphere® ESB. The WebSphere ESB might be expecting identities in the form of a SAML assertion as the "standard intranet token". The ASP.NET application will be required to supply the identity of the service requester in the form of a SAML assertion.

**Figure 3. Service requester identity propagation pattern**

![Service requester identity propagation pattern](image)

**Service provider pattern**

The *service provider* pattern is shown in Figure 4. An incoming identity token is sent to the STS for validation and mapping to a local identity. This pattern is used in cases where the receiving service component is expected to accept an identity token that it is not able to natively support. The motivation for this might be that an enterprise-wide token standard has been employed or that the validation capabilities of the service component are insufficient for the requirements of the particular SOA environment.

For example, an ASP.NET application acting as a Web service can receive requests from a different business unit or organization. The open, platform neutral SAML assertion token type can be chosen for maximum flexibility and interoperability in this scenario.
Figure 4. Service provider identity propagation pattern

Solution design

The key to this solution is to exploit the capability to implement a custom security policy assertion in Microsoft WSE. When implementing custom assertions, four scenarios must be considered and optionally implemented:

1. Client sending a request
2. Service receiving a request
3. Service sending a response
4. Client receiving a response

This solution customizes the behaviour of outbound requests from the client and requests received by the service. The key components are shown in Figure 5.

Figure 5. TFIM security policy assertion

TFIM Trust Client

Both the client output and service input filters will need to communicate to the TFIM STS via WS-Trust protocol. This is achieved with the help of a trust client that establishes a connection to the TFIM Trust Service, and handles all aspects of the WS-Trust token validation and exchange. During normal runtime operation, consumers of this trust client will call SendToken(), which takes a security token as a parameter and returns a security token.
Microsoft WSE provides classes such as Microsoft.Web.Services3.Security.RequestSecurityToken and Microsoft.Web.Services3.Security.RequestSecurityTokenResponse, which can be used as the basis of communicating with an STS. A small amount of additional code has been written around these classes to simplify their use. The TFIM Trust Client used in this article was first introduced in an earlier developerWorks article.

Token Manager

An important aspect of this solution is that there is no understanding of the SAML tokens required by either the policy assertion or the input and output filters. In WSE 3.0, a token that does not have a token manager is parsed as Microsoft.Web.Services3.Security.Tokens.IIssuedToken. Objects of this type can be treated like any other security token, except the token is opaque.

Implementation

This section describes in detail the important aspects of the design introduced in the preceding section. Complete source code is available in the Downloadable resources section at the end of this article.

Creating a custom security policy assertion

The custom policy assertion shown in Listing 2 must sub-class Microsoft.Web.Services3.Design.SecurityPolicy Assertion, and implement the following methods:

- public override void ReadXml(XmlReader reader, IDictionary<string, Type> extensions)

The functionality of the four "Create" methods modify output messages and validate input messages. The other two methods deserve explanation. The GetExtensions() method returns a map of the different configuration names defined for this assertion to the implementing type. These names are then declared in the WSE policy configuration file. For example, the UsernameOverTransportAssertion class above would return a map from "usernameOverTransportSecurity" to the type of its class.

The ReadXml() method is for reading configuration parameters set in the WSE policy configuration file. For this solution there is at least one configuration parameter: the URL of the STS used to validate and issue security tokens.
Listing 2. The Policy Assertion Implementation

using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Web.Services3;
using Microsoft.Web.Services3.Design;
using System.Diagnostics;
using System.Xml;

namespace IBM.Tivoli.SOA.WSSM
{
    public class TFIMSecurityAssertion : SecurityPolicyAssertion
    {
        private Uri _stsUrl = null;
        private string _issuer = null;
        private string _tokenType = null;

        #region Fields
        public Uri fimStsUrl
        {
            get { return _stsUrl; }
        }
        public string Issuer
        {
            get { return _issuer; }
        }
        public string TokenType
        {
            get { return _tokenType; }
        }
        #endregion

        public TFIMSecurityAssertion() : base()
        {
            Trace.WriteLine("TFIMSecurityAssertion()");
        }

        public override Microsoft.Web.Services3.SoapFilter CreateClientInputFilter(FilterCreationContext context)
        {
            return null;
        }

        public override Microsoft.Web.Services3.SoapFilter CreateClientOutputFilter(FilterCreationContext context)
        {
            if (context == null)
            {
                throw new ArgumentNullException("context");
            }

            return new TFIMClientOutputFilter(this);
        }

        {
            if (context == null)
            {
                throw new ArgumentNullException("context");
            }

            return new TFIMServerInputFilter(this);
        }
    }
}
public override Microsoft.WebServices3.SoapFilter CreateServiceOutputFilter(
    FilterCreationContext context)
{
    return null;
}

public override IEnumerable<KeyValuePair<string, Type>> GetExtensions()
{
    // Gets the collection of policy extensions that are registered in the policy file.
    return new KeyValuePair<string, Type>[] {
        new KeyValuePair<string, Type>("TFIMSecurityAssertion", this.GetType())
    };
}

public override void ReadXml(XmlReader reader, IDictionary<string, Type> extensions)
{
    if (reader == null)
        throw new ArgumentNullException("reader");
    if (extensions == null)
        throw new ArgumentNullException("extensions");
    bool isEmpty = reader.IsEmptyElement;
    base.ReadAttributes(reader);
    reader.ReadStartElement("TFIMSecurityAssertion");
    if (!isEmpty)
    {
        // We need the following:
        // - STS endpoint
        // - Issuer
        if (reader.MoveToContent() == XmlNodeType.Element && reader.Name == "fimStsUrl")
        {
            reader.ReadStartElement();
            string urlString = reader.ReadContentAsString();
            _stsUrl = new Uri(urlString);
            reader.ReadEndElement();
        }
        if (reader.MoveToContent() == XmlNodeType.Element && reader.Name == "fimIssuer")
        {
            reader.ReadStartElement();
            _issuer = reader.ReadContentAsString();
            reader.ReadEndElement();
        }
        if (reader.MoveToContent() == XmlNodeType.Element && reader.Name == "TokenType")
        {
            reader.ReadStartElement();
            _tokenType = reader.ReadContentAsString();
            reader.ReadEndElement();
        }
    }
    reader.ReadEndElement();
}

public override void WriteXml(XmlWriter writer)
{
    writer.WriteStartElement("TFIMSecurityAssertion");
    writer.WriteStartElement("fimStsUrl");
    writer.WriteString(_stsUrl.AbsoluteUri);
    writer.WriteEndElement();
}
The client output filter

The client output filter is responsible for requesting a SAML assertion from TFIM and inserting it into an outgoing message. The token that is sent to TFIM is the "Client Token", and is extracted from the credentials in the current message context.

To achieve this functionality, the WSE 3.0 default Microsoft.Web.Services3.Security.SendSecurityFilter class is extended. In Listing 3, configuration information from the parent Policy Assertion class is extracted. Such configuration information includes the TFIM STS endpoint URL, and optionally an override for the Issuer parameter used in the WS-Trust requests to the TFIM STS.

The method in Listing 3 that does all the work is SecureMessage(). In this method, the following steps are performed:

1. Examine the destination of the current message to construct a value of the "AppliesTo" parameter for use in WS-Trust requests to the TFIM STS.
4. Send the UsernameToken to TFIM using the TFIM Trust Client.
5. Add the token returned from TFIM to the list of tokens in the outgoing message.

Listing 3. The client output filter implementation

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using IBM.Tivoli.FIM.TrustClient;
using System.Collections;
using Microsoft.Web.Services3;
using Microsoft.Web.Services3.Addressing;
using System.Security.Principal;

namespace IBM.Tivoli.SOASWSSM
{
    /*
     * secureMessage() method is called on outgoing messages. It
     * requests a SAML assertion from TFIM and inserts it into the
     * outgoing message. The token that is sent to TFIM is the "Client Token".
     */
    public class SecureMessage
    {
        private AuditItem _auditItem;
        private string _issuer;
        private string _tokenType;
        
        // Restore the original issuer and tokenType
        public void ReturnPrincipals()
        {
            _auditItem.Issuer = _issuer;
            _auditItem.TokenType = _tokenType;
        }
        
        // Add the token to the message
        public void AddFimSecurityToken
```
public class TFIMClientOutputFilter : SecureConversationClientSendSecurityFilter
{
  private static string DEFAULT_ISSUER = "urn:tivoli:soa:token-generator";

  FIMTrustClient trustClient = null;

  public TFIMClientOutputFilter(TFIMSecurityAssertion assertion)
  : base(assertion)
  {
    //Get the issuer / appliesTo from the security assertion class
    string issuer = assertion.Issuer;
    if (assertion.Issuer != null)
      issuer = assertion.Issuer;

    trustClient = FIMTrustClientFactory.createTrustClient(
      assertion.fimStsUrl, issuer, null);
    if (assertion.TokenType != null)
      trustClient.TokenType = assertion.TokenType;
  }

  public override void SecureMessage(SoapEnvelope envelope, Security security,
  {
    EndpointReference serviceEPR = null;
    AddressingHeaders headers = envelope.Context.Addressing;
    if (headers != null && headers.Destination != null)
    {
      Uri addressingTo = headers.Destination.Address.Value;
      System.Diagnostics.Trace.WriteLine("AppliesTo: " + addressingTo.ToString());
      serviceEPR = new EndpointReference(addressingTo);
    }
    else
    {
      serviceEPR = new EndpointReference(envelope.Context.WebRequest.RequestUri);
    }
    System.Diagnostics.Trace.WriteLine("Service URL: " + serviceEPR.Address.Value);

    //Get the current user's identity
    if (currentToken == null)
    {
      //No token has been set yet, so we need to construct one based on the
      //current user's principal
      if (currentPrin != null
          && currentPrin.Identity.Name != null
          && !"".Equals(currentPrin.Identity.Name))
      {
        String userName = currentPrin.Identity.Name;
        currentToken = new UsernameToken(userName, "dummy");
        envelope.Context.Credentials.UltimateReceiver.SetClientToken(currentToken);
      }
    }
    System.Diagnostics.Trace.WriteLine("Current IdentityToken: " + currentToken);
    if (currentToken != null)
    {
      SecurityToken responseToken = trustClient.SendToken(serviceEPR, currentToken);
    }
}
The service input filter

The service input filter is responsible for sending the SAML assertion to TFIM, and setting the token returned from TFIM as the "Client Token" in the message's context.

The method in Listing 4 that does all the work is ValidateMessageSecurity(). In this method, the following steps are performed:

1. Examine the destination of the current message to construct a value for the "AppliesTo" parameter to be used in WS-Trust requests to the TFIM STS.
2. Extract the first security token in the SOAP header.
3. Send this security token to TFIM using the TFIM Trust Client.
4. Set the security token returned from TFIM as the client's identity token using envelope.Context.Credentials.UltimateReceiver.SetClientToken(responseToken);

Listing 4. The service input filter implementation

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using IBM.Tivoli.FIM.TrustClient;
using System.Collections;
using Microsoft.Web.Services3;
using Microsoft.Web.Services3.Addressing;
namespace IBM.Tivoli.SOA.WSSM
{
    public class TFIMServerInputFilter : SecureConversationServiceReceiveSecurityFilter
    {
        private static string DEFAULT_ISSUER = "urn:tivoli:soa:token-consumer";

        FIMTrustClient trustClient = null;
        public TFIMServerInputFilter(TFIMSecurityAssertion assertion)
        : base(assertion)
        {
            //Get the issuer / appliesTo from the security assertion class
            Uri stsEndpoint = assertion.fimStsUrl;

            string issuer = DEFAULT_ISSUER;
            if (assertion.Issuer != null)
                issuer = assertion.Issuer;

            trustClient = FIMTrustClientFactory.createTrustClient(stsEndpoint, issuer, null);

            if (assertion.TokenType != null)
                trustClient.TokenType = assertion.TokenType;
        }
    }
}```
public override void ValidateMessageSecurity(
    Microsoft.Web.Services3.SoapEnvelope envelope,
    Security security,
{
    AddressingHeaders headers = envelope.Context.Addressing;
    Uri addressingTo = headers.To.Value;

    System.Diagnostics.Trace.WriteLine("AppliesTo: " + addressingTo.ToString());

    EndpointReference serviceEPR = new EndpointReference(addressingTo);

    //Extract the token(s) from the message
    SecurityToken token = null;
    if (security.Tokens.Count > 0)
    {
        //At a later date, we could add functionality to only send certain types
        //of security tokens to TFIM. For the moment, we'll just send the first
        //token we find.

        IEnumerator e = security.Tokens.GetEnumerator();
        e.MoveNext();
        token = (SecurityToken)e.Current;
    }

    //Send the one we want off to TFIM
    if (token != null)
    {
        System.Diagnostics.Trace.WriteLine("Sending token: " + token);

        SecurityToken responseToken = trustClient.SendToken(serviceEPR, token);
        System.Diagnostics.Trace.WriteLine("Response token: " + responseToken);

        //The value returned from TFIM will be set as the identity
        envelope.Context.Credentials.UltimateReceiver.SetClientToken(responseToken);
        envelope.Context.IdentityToken = responseToken;
    }
    else
    {
        throw new SecurityFault("No valid token found in message");
    }
}

Using the solution

In this section, the solution described in the preceding sections is applied to a sample Web service application to demonstrate the configuration required and runtime expected behaviour. The code for the sample client and service can be obtained from the Downloadable resources section of this article. For convenience, the service interface for the Echo application from the TFIM Web Services Security Management (WSSM) component is used.

Figure 6 shows the environment that will be constructed in this section. SAML enabling a client and Web service in the same real world environment is perhaps unlikely (though convenient for testing the code attached to this article). A more practical application of the technology in this article would be to use each half of the solution separately. For example:

- **SAML-enabled Web service client**
A possible scenario could be where a presentation layer (for example, a user interface) developed in Microsoft .NET needs to communicate with a service hosted by an external partner. Such services are often configured to accept SAML security tokens because of their platform-neutral, widely implemented nature.

- **SAML-enabled Web service**
  A possible scenario could be where a Web service hosted in Microsoft ASP.NET is required to provide service to a wide variety of clients, and a plain Username token is not considered rich enough for asserting the identity of the caller. A token type such as SAML is perhaps preferred because it can contain identity information beyond the username.

In both cases, integration with TFIM provides the additional benefit of being able to perform identity mapping between the local administrative domain and the administrative domain of the partner.

**Figure 6. Prototype scenario**

At runtime, the sequence of activity shown by Figure 6 is:

1. The user attempts to access a (protected) resource running on the front-end ASP.NET application server and authenticates to WebSEAL.
2. WebSEAL proxies the request to the front-end ASP.NET application server, sending the user's serialized TAM credential in the HTTP header.
3. Tivoli Access Manager for Microsoft .NET (AMNET) uses the TAM credential from the incoming iv-creds HTTP header to set the identity of the requester in the front-end ASP.NET application server.
4. The resource being accessed on the front-end ASP.NET application server needs to make a Web service call to the back-end ASP.NET application server. The policy on the Web service requires a SAML 2.0 assertion.
5. A custom client security filter in the front-end ASP.NET application server is invoked. This client security filter uses the username from the TAM credential in a Username token in a WS-Trust RequestSecurityToken message to the TFIM STS.
6. A SAML 2.0 assertion is returned to the output security filter.
7. The Web service request (now containing the SAML 2.0 security token) is sent to the back-end ASP.NET application server.
8. A custom service security filter in the back-end ASP.NET application server is invoked to consume the SAML 2.0 security token.
9. The service security filter sends the SAML 2.0 security token in a WS-Trust RequestSecurityToken message to the TFIM STS.
10. A Username token without password is returned to the service security filter.
11. The service security filter establishes the user's identity in the back-end ASP.NET application server.

TAM configuration
As Figure 6 shows, Tivoli Access Manager (TAM) is used to protect the Echo client application. This involves configuring two different TAM components:

- **WebSEAL**
  The WebSEAL reverse proxy needs to be configured with a junction to the IIS server hosting the Echo client. WebSEAL also needs to be configured to send the credential of the authenticated user across the junction. This is done by using the "-c iv_creds" option when creating the junction.

- **AMNET**
  The AMNET authentication plug-in establishes the user identity in the ASP.NET environment from the TAM credential sent from the WebSEAL server. For more information on how to configure AMNET, consult the documentation recommended in the Related topics section of this article.

WSE policy configuration
This section describes the changes to the standard application configuration files for Microsoft WSE applications. It is assumed that the reader has some familiarity with constructing Web service applications in Microsoft .NET already, as the configuration is not explained in detail.

Echo client
The changes to the application configuration file shown in Listing 5 enable trace and specify the location of the WSE policy configuration file. Listing 5 is abridged to show the changes specific to constructing this solution.

Listing 5. Web.config (abridged)
```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
... 
<microsoft.web.services3>
  <diagnostics>
    <trace enabled="true"
      input="C:\temp\client-in.trace"
      output="C:\temp\client-out.trace" />
    <detailedErrors enabled="true" />
  </diagnostics>
  <policy fileName="wse3policyCache.config" />
</microsoft.web.services3>
</configuration>
```

The WSE policy configuration shown in Listing 6 specifies use of the TFIM security assertion, which includes the location of the TFIM STS.
Listing 6. wse3policyCache.config

```xml
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <extensions>
    <extension name="TFIMSecurityAssertion"
      type="IBM.Tivoli.SOAWSSM.TFIMSecurityAssertion, IBM.Tivoli.SOAWSSM"/>
  </extensions>
  <policy name="WSTrustClientSend">
    <TFIMSecurityAssertion>
      <fimStsUrl>http://[host]:[port]/TrustServer/SecurityTokenService</fimStsUrl>
    </TFIMSecurityAssertion>
  </policy>
</policies>
```

Echo service

The changes to the application configuration file shown in Listing 7 enable trace and specify the location of the WSE policy configuration file. Listing 7 is abridged to show the changes specific to constructing this solution. The TrustedUsernameTokenManager has been configured to disable the default .NET functionality of authenticating users in the UsernameToken against the local user store.

Listing 7. Web.config (abridged)

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    ...
    <webServices>
      ...
      <soapExtensionImporterTypes>
        <add
      </soapExtensionImporterTypes>
      <soapServerProtocolFactory
    </soapServerProtocolFactory>
  </soapExtensionImporterTypes>
</webServices>
</system.web>
<microsoft.web.services3>
  <security>
    <securityTokenManager>
      <add namespace="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
        localName="UsernameToken" type="IBM.Tivoli.FIM.TrustClient.Tokens.TrustedUsernameTokenManager, FIMTrustClient" />
    </securityTokenManager>
  </security>
  <diagnostics>
    <trace enabled="true"
      input="C:\temp\service-in.trace"
      output="C:\temp\service-out.trace" />
    <detailedErrors enabled="true" />
  </diagnostics>
  <policy fileName="wse3policyCache.config" />
</microsoft.web.services3>
</configuration>
```

The WSE policy configuration shown in Listing 8 specifies use of the TFIM security assertion, which includes the location of the TFIM STS.
Listing 8. wse3policyCache.config

```xml
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <extensions>
    <extension name="TFIMSecurityAssertion"
      type="IBM.Tivoli.SOA.WSSM.TFIMSecurityAssertion, IBM.Tivoli.SOA.WSSM"/>
  </extensions>

  <policy name="WSTrustServiceReceive">
    <TFIMSecurityAssertion>
      <fimStsUrl>http://[host]:[port]/TrustServer/SecurityTokenService</fimStsUrl>
    </TFIMSecurityAssertion>
  </policy>
</policies>
```

**TFIM configuration**

The solution utilizes the functionality of TFIM STS to provide token-agnostic authentication to .NET Web services. This section describes how to configure TFIM STS to issue SAML v2.0 assertions for the client and to consume SAML v2.0 assertions for the server. In both cases, the token being exchanged for SAML assertion is a UsernameToken.

When either the client or server filters in the example construct the RequestSecurityToken (RST) WS-Trust message to send to TFIM, the AppliesTo value is set to the URL of the receiving service. If desired, individual trust chain pairs can be configured for each service in the environment. For the example, we will create a single pair of trust chains that can be applied to different services.

Table 1 shows the trust chain configuration for the client output filter. Notice the Issuer is the value urn:tivoli:soa:token-generator. This is the default value set by the client output filter for all WS-Trust requests.

**Table 1. TFIM trust chain for client output**

<table>
<thead>
<tr>
<th>Configuration parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppliesTo</td>
<td>*</td>
</tr>
<tr>
<td>Issuer</td>
<td>urn:tivoli:soa:token-generator</td>
</tr>
<tr>
<td>Request type</td>
<td>Validate</td>
</tr>
<tr>
<td>Token type (optional)</td>
<td>Standard Token Type: SAML v2.0</td>
</tr>
<tr>
<td>Chain</td>
<td>Default UsernameToken; mode= Validate</td>
</tr>
<tr>
<td></td>
<td>Default SAML v2.0 token; mode = Issue</td>
</tr>
</tbody>
</table>

Table 2 shows the trust chain configuration for the service input filter. The Issuer in this case is urn:tivoli:soa:token-consumer. This is the default value set by the service input filter for all WS-Trust requests.

**Table 2. TFIM trust chain for service input**

<table>
<thead>
<tr>
<th>Configuration parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppliesTo</td>
<td>*</td>
</tr>
</tbody>
</table>
**Issuer**

<table>
<thead>
<tr>
<th>Request type</th>
<th>Token type (optional)</th>
<th>Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validate</td>
<td>Standard Token Type: Username</td>
<td>Default SAML v2.0 token; mode = Validate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default UsernameToken; mode= Issue</td>
</tr>
</tbody>
</table>

**Interoperability**

The client and service in this section were successfully tested against the J2EE™-based client and service provided with TFIM WSSM, version 6.1.1.

**Challenges**

An advantage of this solution is that the token type used can be changed at any time without modification to the application or its deployment. For example, future company security policy could require that X.509 certificates be used over-the-wire instead of SAML v2.0 assertions. This approach is not without a downside. As the token is opaque, attribute data in the SAML assertion cannot be used at runtime. The SAML assertion might contain extended attributes, and as the solution uses TFIM to extract just the name, these additional attributes are lost.

**Future work**

The solution could be extended in a number of ways:

1. On the service consumer (client) side, the client output filter represents the currently authenticated identity in a Username token when sending this to TFIM. This means that only the username is sent, limiting what information can be included in the security token returned by TFIM. The solution could be enhanced to extract additional information from the currently authenticated identity and represent this in a richer token format such as SAML 1.0 assertion for sending to the TFIM STS. One example would be if the client application uses AMNET, which could make the contents of the TAM credential available to the trust module chain in TFIM.

2. On the service side, the service input filter expects a Username token back from the TFIM STS. The user name from the Username token is used to establish the identity in Microsoft .NET. One possible enhancement would be to allow the service input filter to receive a more complex token type containing other attributes, and use AMNET to set the user identity in TAM credential format.

3. The entire solution could be ported to Windows Communication Foundation(WCF). WCF is the replacement for WSE 3.0 and is part of the .NET Framework 3.0

**Conclusion**

This article has described how to integrate additional security token types in Microsoft .NET by using Tivoli Federated Identity Manager and its Security Token Service. The advantage of this approach over others is that the implementation described above will work with almost any token
type without further code modification - just different trust module chain configuration in TFIM. This makes it a flexible solution that can future-proof a SOA environment.
**Downloadable resources**

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFIM Security Policy Assertion¹</td>
<td>tfim-security-assertion.zip</td>
<td>218 KB</td>
</tr>
<tr>
<td>Echo client</td>
<td>echo-client.zip</td>
<td>130 KB</td>
</tr>
<tr>
<td>Echo service</td>
<td>echo-service.zip</td>
<td>37 KB</td>
</tr>
</tbody>
</table>

**Note**

1. Includes the TFIM trust client
Related topics

- The Tivoli Federated Identity Manager Version 6.1.1 Information Center.
- Specification: Security Assertion Markup Language
- Microsoft Web Services Enhancements 3.0
- Specification: WS-Trust 1.3
- IBM Redbooks: Understanding SOA Security
- IBM Redpaper: Propagating Identity in SOA with IBM Tivoli Federated Identity Manager
- Previously on developerWorks: ASP.NET Authentication using LTPA and Tivoli Federated Identity Manager (TFIM)
- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

© Copyright IBM Corporation 2008

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)