Table of Contents

Windows Identity Foundation

5/3/2017 • 1 min to read • Edit Online

- WIF Extensions
- WIF API Reference
- WIF Configuration Reference
  - WIF Configuration Schema Conventions

What's New in Windows Identity Foundation 4.5

5/3/2017 • 4 min to read • Edit Online

The first version of Windows Identity Foundation (WIF) shipped as a standalone download and is known as WIF 3.5 because it was introduced in the .NET 3.5 SP1 timeframe. Starting with .NET 4.5, WIF is part of the .NET framework. Having the WIF classes directly available in the framework itself allows for a much deeper integration of claims-based identity in the .NET platform, making it easier to use claims. Applications written for WIF 3.5 will need to be modified in order to take advantage of the new model; for information, see Guidelines for Migrating an Application Built Using WIF 3.5 to WIF 4.5.

Below you can find some highlights of the main changes.

WIF Is Now Part of the .NET Framework

WIF classes are now spread across several assemblies, the main ones being `mscorlib`, `System.IdentityModel`, `System.IdentityModel.Services`, and `System.ServiceModel`. Likewise, the WIF classes are spread across several namespaces: System.Security.Claims, several System.IdentityModel namespaces, and System.ServiceModel.Security. The System.Security.Claims namespace contains the new ClaimsPrincipal and ClaimsIdentity classes (see below). All principals in .NET now derive from ClaimsPrincipal. For more detailed information about the WIF namespaces and the kinds of classes that they contain, see WIF API Reference. For information about how namespaces map between WIF 3.5 and WIF 4.5, see Namespace Mapping between WIF 3.5 and WIF 4.5.

New Claims Model and Principal Object

Claims are at the very core of the .NET Framework 4.5. The base claim classes (Claim, ClaimsIdentity, ClaimsPrincipal, ClaimTypes, and ClaimValueTypes) all live directly in `mscorlib` in the System.Security.Claims namespace. It is no longer necessary to use interfaces in order to plug claims into the .NET identity system: WindowsPrincipal, GenericPrincipal, and RolePrincipal now inherit from ClaimsPrincipal; and WindowsIdentity, GenericIdentity, and FormsIdentity now inherit from ClaimsIdentity. In short, every principal class will now serve claims. The WIF 3.5 integration classes and interfaces (`WindowsClaimsIdentity`, `WindowsClaimsPrincipal`, `IClaimsPrincipal`, `IClaimsIdentity`) have thus been removed. In addition, the ClaimsIdentity class now exposes methods which make it easier to query the identity’s claims collection.

Changes to the WIF 4.5 Visual Studio Experience

- The **Add STS Reference …** Visual Studio functionality (cmdline utility FedUtil) no longer exists; instead you can use the new Visual Studio extension **Identity and Access Tool for Visual Studio 2012**. This allows you to federate with an existing STS or use LocalSTS to test your solutions. After installing the extension you can right-click on your project and look for **Identity and Access** in the context menu.

- The ASP.NET and STS templates are no longer provided as claims can be used directly in existing project templates for ASP.Net, web sites, and WCF.

- The controls in the `Microsoft.IdentityModel.Web.Controls` namespace (`SignInControl`, `FederatedPassiveSignInControl`, and `FederatedPassiveSignInStatus`) are not carried over into WIF 4.5.

Changes to the WIF 4.5 API

- In general, claims related classes are in the System.Security.Claims namespace; WCF related classes –-

service contracts, channels, channel factories, and service hosts that are used for WS-Trust scenarios -- are in System.ServiceModel.Security; and all other WIF classes are spread across various System.IdentityModel namespaces – these include, for example, classes that represent WS-* and SAML artifacts, token handlers and related classes, and classes used in WS-Federation scenarios. For more information, see Namespace Mapping between WIF 3.5 and WIF 4.5 and WIF API Reference.

- Machine key has been enabled for use in session cookies for web farm scenarios. For more information, see WIF and Web Farms.

- You now declaratively configure WIF under the <system.identityModel> and <system.identityModel.services> elements. For more information about WIF configuration, see WIF Configuration Reference.

Other notable .NET changes or features that are caused by the integration of WIF into .NET

- The potential for performing claims-based authorization (CBAC) is now integral to the .NET framework. You can configure a ClaimsAuthorizationManager object and then use the ClaimsPrincipalPermission and ClaimsPrincipalPermissionAttribute classes to perform imperative and declarative access checks in your code. CBAC provides more flexibility and greater granularity than traditional role-based access checks (RBAC). It also allows greater separation of authorization policy from business logic because the business logic can base the access check on a specific claim or set of claims and the authorization policy for those claims can be configured declaratively under the <claimsAuthorizationManager> element.

WCF changes as a result of WIF integration:

- The WCF claims-based identity model is superseded by WIF. This means that classes in the System.IdentityModel.Claims, System.IdentityModel.Policy, and System.IdentityModel.Selectors namespaces should be dropped in favor of using WIF classes.

- WIF is now enabled on a WCF service by specifying the `useIdentityConfiguration` attribute on the `<system.serviceModel>` / `<behaviors>` / `<serviceBehaviors>` / `<serviceCredentials>` element as in the following XML:

```
<serviceCredentials useIdentityConfiguration="true">
    <!--Certificate added by Identity And Access VS Package.  Subject='CN=localhost',
Issuer='CN=localhost'. Make sure you have this certificate installed. The Identity and Access tool does
not install this certificate.-->
    <serviceCertificate findValue="CN=localhost" storeLocation="LocalMachine" storeName="My"
x509FindType="FindBySubjectDistinguishedName" />
</serviceCredentials>
```

When you use the **Identity and Access Tool for Visual Studio 2012** (see **Changes to the Visual Studio Experience** above), the tool adds a `<serviceCredentials>` element with the `useIdentityConfiguration` attribute set to the configuration file for you. It also adds a corresponding <system.identityModel> element that contains the WIF configuration settings and adds a binding and other settings necessary to outsource authentication to your chosen STS.

See Also

Guidelines for Migrating an Application Built Using WIF 3.5 to WIF 4.5
Namespace Mapping between WIF 3.5 and WIF 4.5
WIF API Reference
WIF Configuration Reference

Windows Identity Foundation 4.5 Overview

5/16/2017 • 2 min to read • Edit Online

Windows Identity Foundation 4.5 is a set of .NET Framework classes for implementing claims-based identity in your applications. By using it, you'll more easily reap the benefits of claims-aware applications and services. WIF 4.5 can be used in any Web application or Web service that uses the .NET Framework version 4.5 or later. WIF is just one part of Microsoft's Federated Identity software family that implements the shared industry vision based on open standards. Federated Identity comprises three components: Active Directory® Federation Services (AD FS) 2.0, Windows Azure Access Control Services (ACS), and WIF. Together, these three components form the core of Microsoft's new claims-based cloud identity and access platform.

For more information about WIF, see the Windows Identity Foundation Web site (http://go.microsoft.com/fwlink/?LinkId=149009) at the Security Developer Center on MSDN. For an introduction to creating applications using WIF, see Programming Windows Identity Foundation (http://go.microsoft.com/fwlink/?LinkId=210158) by Vittorio Bertocci (published by Microsoft Press).

WIF 4.5 Features

WIF 4.5 is a framework for building identity-aware applications. The framework abstracts the WS-Trust and WS-Federation protocols and presents developers with APIs for building claims-aware applications and, if needed, security token services (STS)s. Applications can use WIF to process tokens issued from STSs, such as AD FS 2.0 and ACS, and make identity-based decisions at the web application or web service.

WIF 4.5 has the following major features:

- Build claims-aware applications (relying party applications). WIF helps developers build claims-aware applications. In addition to providing a claims model, it provides application developers with a rich set of APIs to help making user access decisions based on claims. WIF also provides developers with a consistent programming experience whether they choose to build their applications in ASP.NET or in WCF environments.

- Build identity delegation support into claims-aware applications. WIF offers the capability of maintaining the identities of original requestors across the multiple service boundaries. This capability can be achieved by either using the "ActAs" or the "OnBehalfOf" functionality in the framework and it offers developers the ability to add identity delegation support into their claims-aware applications.

- Build custom STSs. WIF makes it substantially easier to build a custom STS that supports the WS-Trust protocol. Such an STS is also referred to as an Active STS.

  In addition, the framework also provides support for building an STS that supports WS-Federation to enable Web browser clients. Such an STS is also referred to as a Passive STS.

- New identity and access tool for Visual Studio 11 that enables you to secure your application with claims based identity and accept users from multiple identity providers. You can download this WIF tool from the following URL: http://go.microsoft.com/fwlink/?LinkID=245849 or directly from within Visual Studio 11 by searching for "identity" directly in the Extensions Manager. For more information, see Identity and Access Tool for Visual Studio 2012.

WIF supports the following major scenarios:

- Federation. WIF makes it possible to enable federation between two or more partners. Its support for building claims-aware applications (RPs) and custom STSs helps developers achieve this scenario.

- Identity Delegation. WIF makes it easy to maintain the identities across the service boundaries so that developers can achieve an identity delegation scenario.
- Step-up Authentication. Authentication requirements for different resources within an application may vary. WIF provides developers the ability to build applications that can require incremental authentication requirements (for example: initial login with Username/Password authentication and then step-up to Smart Card authentication).

By using WIF, you'll more easily reap the benefits of the claims-based identity model. For more information, see Windows Identity Foundation White Paper for Developers.

Claims-Based Identity Model

5/16/2017 • 6 min to read • Edit Online

When you build claims-aware applications, the user identity is represented in your application as a set of claims. One claim could be the user's name, another might be an e-mail address. The idea is that an external identity system is configured to give your application everything it needs to know about the user with each request she makes, along with cryptographic assurance that the identity data you receive comes from a trusted source.

Under this model, single sign-on is much easier to achieve, and your application is no longer responsible for the following:

- Authenticating users.
- Storing user accounts and passwords.
- Calling to enterprise directories to look up user identity details.
- Integrating with identity systems from other platforms or companies.

Under this model, your application makes identity-related decisions based on claims supplied by the system that authenticated your user. This could be anything from simple application personalization with the user's first name, to authorizing the user to access higher valued features and resources in your application.

This topic provides the following information:

- Introduction to Claims-Based Identity
- Basic Scenario for a Claims-Based Identity Model

Introduction to Claims-Based Identity

The following terminology and concepts can help you understand this new architecture for identity.

Identity

For the purposes of describing the programming model in Windows Identity Foundation (WIF), we will use the term "identity" to represent a set of attributes that describe a user or some other entity in a system that you want to secure.

Claim

Think of a claim as a piece of identity information such as name, e-mail address, age, membership in the Sales role. The more claims your application receives, the more you'll know about your user. You may be wondering why these are called "claims," rather than "attributes," as is commonly used in describing enterprise directories. The reason has to do with the delivery method. In this model, your application doesn't look up user attributes in a directory. Instead, the user delivers claims to your application, and your application examines them. Each claim is made by an issuer, and you trust the claim only as much as you trust the issuer. For example, you trust a claim made by your company's domain controller more than you trust a claim made by the user herself. WIF represents claims with a Claim type, which has an Issuer property that allows you to find out who issued the claim.

Security Token

The user delivers a set of claims to your application along with a request. In a Web service, these claims are carried in the security header of the SOAP envelope. In a browser-based Web application, the claims arrive through an HTTP POST from the user's browser, and may later be cached in a cookie if a session is desired. Regardless of how these claims arrive, they must be serialized, which is where security tokens come in. A security token is a serialized set of claims that is digitally signed by the issuing authority. The signature is important: it gives you assurance that the user didn’t just make up a bunch of claims and send them to you. In low security situations where cryptography isn’t necessary or desired, you can use unsigned tokens, but that scenario is not described in this topic.

One of the core features in WIF is the ability to create and read security tokens. WIF and the .NET Framework handle all of the cryptographic work, and present your application with a set of claims that you can read.

Issuing Authority

There are lots of different types of issuing authorities, from domain controllers that issue Kerberos tickets, to certification authorities that issue X.509 certificates, but the specific type of authority discussed in this topic issues security tokens that contain claims. This issuing authority is a Web application or Web service that knows how to issue security tokens. It must have enough knowledge to be able to issue the proper claims given the target relying party and the user making the request, and might be responsible for interacting with user stores to look up claims and authenticate the users themselves.

Whatever issuing authority you choose, it plays a central role in your identity solution. When you factor authentication out of your application by relying on claims, you’re passing responsibility to that authority and asking it to authenticate users on your behalf.

Security Token Service (STS)

A security token service (STS) is the service component that builds, signs, and issues security tokens according to the WS-Trust and WS-Federation protocols. There’s a lot of work that goes into implementing these protocols, but WIF does all of this work for you, making it feasible for someone who isn’t an expert in the protocols to get an STS up and running with very little effort. You can use a pre-built STS such as Active Directory® Federation Services (AD FS) 2.0, a cloud STS such as a Windows Azure Access Control Service (ACS), or, if you want to issue custom tokens or provide custom authentication or authorization, you can build your own custom STS using WIF. WIF makes it easy to build your own STS.

Relying Party Application

When you build an application that relies on claims, you are building a relying party (RP) application. Synonyms for an RP include "claims-aware application" and "claims-based application". Web applications and Web services can both be RPs. A RP application consumes the tokens issued by a STS and extracts the claims from tokens to use them for identity related tasks. WIF offers functionalities to help you build RP applications.

Standards

In order to make all of this interoperable, several WS-* standards are used in the previous scenario. Policy is retrieved using WS-MetadataExchange, and the policy itself is structured according to the WS-Policy specification. The STS exposes endpoints that implement the WS-Trust specification, which describes how to request and receive security tokens. Most STSs today issue tokens formatted with Security Assertion Markup Langauge (SAML). SAML is an industry-recognized XML vocabulary that can be used to represent claims in an interoperable way. Or, in a multi-platform situation, this allows you to communicate with an STS on an entirely different platform and achieve single sign-on across all of your applications, regardless of platform.

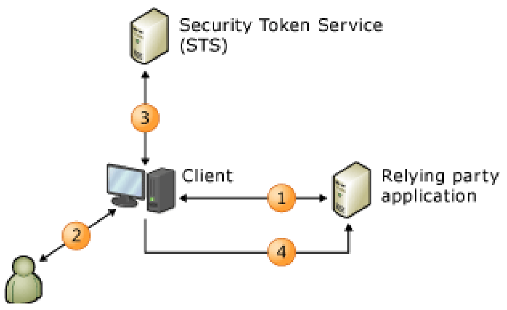
Browser-Based Applications

Smart clients aren’t the only ones who can use the claims-based identity model. Browser-based applications (also referred to as passive clients) can use it as well. The following scenario describes how this works.

First, the user points a browser at a claims-aware Web application (the relying party application). The Web application redirects the browser to the STS so the user can be authenticated. The STS is hosted in a simple web application that reads the incoming request, authenticates the user using standard HTTP mechanisms, and then creates a SAML token and replies with a piece of JavaScript code that causes the browser to initiate an HTTP POST that sends the SAML token back to the RP. The body of this POST contains the claims that the RP requested. At this point, it is common for the RP to package the claims into a cookie so that the user doesn’t have to be redirected for each request.

Basic Scenario for a Claims-Based Identity Model

The following is an example of a claims-based system.



This diagram shows a Web site (the relying party application, RP) that has been configured to use WIF for authentication and a client, a web browser, that wants to use that site.

1. When an unauthenticated user requests a page their browser is redirected to the identity provider (IP) pages.
2. The IP requires the user to present their credentials, e.g. username/password, Kerberos, etc.
3. The IP issues a token back to that is returned to the browser.
4. The browser is now redirected back to the originally requested page where WIF determines if the token satisfies the requirements to access the page. If so a cookie is issued to establish a session so the authentication only needs to occur once, and control is passed to the application.

Claims Based Authorization Using WIF

6/13/2017 • 5 min to read • Edit Online

In a relying party application, authorization determines what resources an authenticated identity is allowed to access and what operations it is allowed to perform on those resources. Improper or weak authorization leads to information disclosure and data tampering. This topic outlines the available approaches to implementing authorization for claims-aware ASP.NET web applications and services using Windows Identity Foundation (WIF) and a Security Token Service (STS), for example, the Windows Azure Access Control Service (ACS).

Overview

Since its first version, the .NET Framework has offered a flexible mechanism for implementing authorization. This mechanism is based on two simple interfaces—**IPrincipal** and **IIdentity**. Concrete implementations of **IIdentity** represent an authenticated user. For example, the **WindowsIdentity** implementation represents a user who is authenticated by Active Directory, and **GenericIdentity** represents a user whose identity is verified via a custom authentication process. Concrete implementations of **IPrincipal** help to check permissions using roles depending on the role store. For example, **WindowsPrincipal** checks **WindowsIdentity** for membership in Active Directory groups. This check is performed by calling the **IsInRole** method on the **IPrincipal** interface. Checking access based on roles is called Role-Based Access Control (RBAC). For more information, see Role-Based Access Control. Claims can be used to carry information about roles to support familiar, role-based authorization mechanisms.

Claims can also be used to enable more complicated authorization decisions beyond roles. Claims can be based on virtually any information about the user - age, zip code, shoe size, etc. An access control mechanism that is based on arbitrary claims is called claims-based authorization. For more information, see Claims-based Authorization.

Role-Based Access Control

RBAC is an authorization approach in which user permissions are managed and enforced by an application based on user roles. If a user has a role that is required to perform an action, the access is granted; otherwise, access is denied.

IPrincipal.IsInRole Method

To implement the RBAC approach in claims-aware applications, use the **IsInRole()** method in the **IPrinicpal** interface, just as you would in non-claims-aware applications. There are several ways of using the **IsInRole()** method:

- Explicitly calling on **IPrincipal.IsInRole("Administrator")**. In this approach, the outcome is a Boolean. Use it in your conditional statements. It can be used arbitrarily any place in your code.

- Using the security demand **PrincipalPermission.Demand()**. In this approach, the outcome is an exception in case the demand is not satisfied. This should fit your exception handling strategy. Throwing exceptions is much more expensive from a performance perspective compared to returning Boolean. This can be used any place in your code.

- Using the declarative attributes **[PrincipalPermission(SecurityAction.Demand, Role = "Administrator")]**. This approach is called declarative, because it is used to decorate methods. It cannot be used in code blocks inside the method’s implementations. The outcome is an exception in case the demand is not satisfied. You should make sure that it fits your exception-handling strategy.

- Using URL authorization, using the **<authorization>** section in **web.config**. This approach is suitable when you are managing authorization on a URL level. This is the most coarse level among those previously mentioned. The advantage of this approach is that changes are made in the configuration file, which means that the code should not be compiled to take advantage of the change.

Expressing Roles as Claims

When the **IsInRole()** method is called, there is a check made to see if the current user has that role. In claims-aware applications, the role is expressed by a role claim type that should be available in the token. The role claim type is expressed using the following URI:

http://schemas.microsoft.com/ws/2008/06/identity/claims/role

There are several ways to enrich a token with a role claim type:

- **During token issuance**. When a user is authenticated the role claim can be issued by the identity provider STS or by a federation provider such as the Windows Azure Access Control Service (ACS).
- **Transforming arbitrary claims into of claims role type using ClaimsAuthenticationManager**. The ClaimsAuthenticationManager is a component that ships as part of WIF. It allows requests to be intercepted when they launch an application, inspecting tokens and transforming them by adding, changing, or removing claims. For more information about how to use ClaimsAuthenticationManager for transforming claims, see How To: Implement Role Based Access Control (RBAC) in a Claims Aware ASP.NET Application Using WIF and ACS (http://go.microsoft.com/fwlink/?LinkID=247444).
- **Mapping arbitrary claims to a role type using the samlSecurityTokenRequirement configuration section**—A declarative approach where the claims transformation is done using only the configuration and no coding is required.

Claims-based Authorization

Claims-based authorization is an approach where the authorization decision to grant or deny access is based on arbitrary logic that uses data available in claims to make the decision. Recall that in the case of RBAC, the only claim used was role type claim. A role type claim was used to check if the user belongs to specific role or not. To illustrate the process of making the authorization decisions using claims-based authorization approach, consider the following steps:

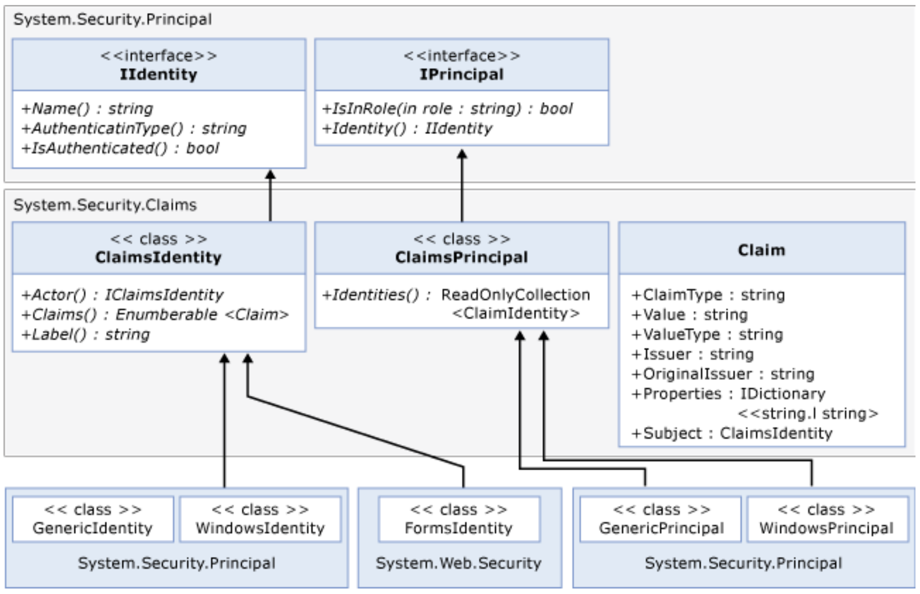
1. The application receives a request that requires the user is authenticated.
2. WIF redirects the user to their identity provider, after they are authenticated the application request is made with an associated security token representing the user containing claims about them. WIF associates those claims with the principal that represents the user.
3. The application passes the claims to the decision logic mechanism. It can be in-memory code, a call to a web service, a query to a database, a sophisticated rules engine, or using the ClaimsAuthorizationManager.
4. The decision mechanism calculates the outcome based on the claims.
5. Access is granted if the outcome is true and denied if it is false. For example, the rule might be that the user is of age 21 or above and lives in Washington State.

ClaimsAuthorizationManager is useful for externalizing the decision logic for claims-based authorization in your applications. ClaimsAuthorizationManager is a WIF component that ships as part of .NET 4.5. ClaimsAuthorizationManager allows you to intercept incoming requests and implement any logic of your choice to make authorization decisions based on the incoming claims. This becomes important when authorization logic needs to be changed. In that case, using ClaimsAuthorizationManager will not affect the application’s integrity, thereby reducing the likelihood of an application error as a result of the change. To learn more about how to use ClaimsAuthorizationManager to implement claims-based access control, see How To: Implement Claims Authorization in a Claims Aware ASP.NET Application Using WIF and ACS.

WIF Claims Programming Model

5/16/2017 • 4 min to read • Edit Online

ASP.NET and Windows Communication Foundation (WCF) developers ordinarily use the IIdentity and IPrincipal interfaces to work with the user's identity information. In .NET 4.5, Windows Identity Foundation (WIF) has been integrated such that claims are now always present for any principal as illustrated in the following diagram:


System.Security.Principal
<<interface>> IIdentity
+Name() : string
+AuthenticatinType() : string
+IsAuthenticated() : bool
<<interface>> IPrincipal
+IsInRole(in role : string) : bool
+Identity() : IIdentity
System.Security.Claims
<< class >> ClaimsIdentity
+Actor() : IClaimsIdentity
+Claims() : Enumberable <Claim>
+Label() : string
<< class >> ClaimsPrincipal
+Identities() : ReadOnlyCollection <ClaimIdentity>
Claim
+ClaimType : string
+Value : string
+ValueType : string
+Issuer : string
+OriginalIssuer : string
+Properties : IDictionary <<string.l string>
+Subject : ClaimsIdentity
<< class >> GenericIdentity
<< class >> WindowsIdentity
System.Security.Principal
<< class >> FormsIdentity
System.Web.Security
<< class >> GenericPrincipal
<< class >> WindowsPrincipal
System.Security.Principal

In .NET 4.5, System.Security.Claims contains the new ClaimsPrincipal and ClaimsIdentity classes (see diagram above). All principals in .NET now derive from ClaimsPrincipal. All built-in identity classes, like FormsIdentity for ASP.NET and WindowsIdentity now derive from ClaimsIdentity. Similarly, all built-in principal classes like GenericPrincipal and WindowsPrincipal derive from ClaimsPrincipal.

A claim is represented by Claim class. This class has the following important properties:

- Type represents the type of claim and is typically a URI. For example, the e-mail address claim is represented as `http://schemas.microsoft.com/ws/2008/06/identity/claims/email`.
- Value contains the value of the claim and is represented as a string. For example, the e-mail address can be represented as "someone@contoso.com".
- ValueType represents the type of the claim value and is typically a URI. For example, the string type is represented as `http://www.w3.org/2001/XMLSchema#string`. The value type must be a QName according to the XML schema. The value should be of the format `namespace#format` to enable WIF to output a valid QName value. If the namespace is not a well-defined namespace, the generated XML probably cannot be schema validated, because there will not be a published XSD file for that namespace. The default value type is `http://www.w3.org/2001/XMLSchema#string`. Please see http://www.w3.org/2001/XMLSchema for well-known value types that you can use safely.
- Issuer is the identifier of the security token service (STS) that issued the claim. This can be represented as URL of the STS or a name that represents the STS, such as `https://sts1.contoso.com/sts`.
- OriginalIssuer is the identifier of the STS that originally issued the claim, regardless of how many STSs are in the chain. This is represented just like Issuer.
- Subject is the subject whose identity is being examined. It contains a ClaimsIdentity.
- Properties is a dictionary that lets the developer provide application-specific data to be transferred on the wire together with the other properties, and can be used for custom validation.

Identity Delegation

An important property of ClaimsIdentity is Actor. This property enables the delegation of credentials in a multi-tier system in which a middle tier acts as the client to make requests to a back-end service.

Accessing Claims through Thread.CurrentPrincipal

To access the current user's set of claims in an RP application, use `Thread.CurrentPrincipal`.

The following code sample shows the usage of this method to get a System.Security.Claims.ClaimsIdentity:

```
ClaimsPrincipal claimsPrincipal = Thread.CurrentPrincipal as ClaimsPrincipal;
```

For more information, see System.Security.Claims.

Role Claim Type

Part of configuring your RP application is to determine what your role claim type should be. This claim type is used by System.Security.Claims.ClaimsPrincipal.IsInRole(System.String). The default claim type is `http://schemas.microsoft.com/ws/2008/06/identity/claims/role`.

Claims Extracted by Windows Identity Foundation from Different Token Types

WIF supports several combinations of authentication mechanisms out of the box. The following table lists the claims that WIF extracts from different token types.

TOKEN TYPE	CLAIM GENERATED	MAP TO WINDOWS ACCESS TOKEN
SAML 1.1	1. All claims from System.IdentityModel.SecurityTokenService.GetOutputClaimsIdentity(System.Security.Claims.ClaimsPrincipal,System.IdentityModel.Protocols.WSTrust.RequestSecurityToken,System.IdentityModel.Scope). 2. The `http://schemas.microsoft.com/ws/2008/06/identity/claims/confirmationkey` claim that contains the XML serialization of the confirmation key, if the token contains a proof token. 3. The `http://schemas.microsoft.com/ws/2008/06/identity/claims/samlissuername` claim from the Issuer element. 4. AuthenticationMethod and AuthenticationInstant claims, if the token contains an authentication statement.	In addition to the claims listed in "SAML 1.1", except claims of type `http://schemas.xmlsoap.org/ws/2005/05/identity/c` , Windows authentication related claims will be added and the identity will be represented by WindowsClaimsIdentity.
SAML 2.0	Same as "SAML 1.1".	Same as "SAML 1.1 Mapped to Windows Account".
X509	1. Claims with the X500 distinguished name, emailName, dnsName, SimpleName, UpnName, UrlName, thumbprint, RsaKey (this can be extracted using the RSACryptoServiceProvider.ExportParameters method from the X509Certificate2.PublicKey.Key property), DsaKey (this can be extracted using the DSACryptoServiceProvider.ExportParameters method from the X509Certificate2.PublicKey.Key property), SerialNumber properties from the X509 Certificate. 2. AuthenticationMethod claim with value `http://schemas.microsoft.com/ws/2008/06/identity/authenticationmethod/x509` . AuthenticationInstant claim with the value of the time when the certificate was validated in XmlSchema DateTime format.	1. It uses the Windows account fully qualified domain name as the `http://schemas.xmlsoap.org/ws/2005/05/identity/c` claim value. . 2. Claims from the X509 Certificate not mapped to Windows, and claims from the windows account obtained by mapping the certificate to Windows.
UPN	1. Claims are similar to the claims in the Windows authentication section. 2. AuthenticationMethod claim with value `http://schemas.microsoft.com/ws/2008/06/identity/authenticationmethod/password` . The AuthenticationInstant claim with the value of the time when the password was validated in XmlSchema DateTime format.	

TOKEN TYPE	CLAIM GENERATED	MAP TO WINDOWS ACCESS TOKEN
Windows (Kerberos or NTLM)	1. Claims generated from the access token such as: PrimarySID, DenyOnlyPrimarySID, PrimaryGroupSID, DenyOnlyPrimaryGroupSID, GroupSID, DenyOnlySID, and Name 2. AuthenticationMethod with the value `http://schemas.microsoft.com/ws/2008/06/identity/authenticationmethod/windows` . AuthenticationInstant with the value of the time when the Windows access token was created in the XMLSchema DateTime format.	
RSA Key Pair	1. The `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/rsa` claim with the value of RSAKeyValue. 2. AuthenticationMethod claim with the value `http://schemas.microsoft.com/ws/2008/06/identity/authenticationmethod/signature` . AuthenticationInstant claim with the value of the time when the RSA key was authenticated (that is, the signature was verified) in the XMLSchema DateTime format.	

AUTHENTICATION TYPE	URI EMITTED IN "AUTHENTICATIONMETHOD" CLAIM
Password	`urn:oasis:names:tc:SAML:1.0:am:password`
Kerberos	`urn:ietf:rfc:1510`
SecureRemotePassword	`urn:ietf:rfc:2945`
TLSClient	`urn:ietf:rfc:2246`
X509	`urn:oasis:names:tc:SAML:1.0:am:X509-PKI`
PGP	`urn:oasis:names:tc:SAML:1.0:am:PGP`
Spki	`urn:oasis:names:tc:SAML:1.0:am:SPKI`
XmlDSig	`urn:ietf:rfc:3075`
Unspecified	`urn:oasis:names:tc:SAML:1.0:am:unspecified`

Getting Started With WIF

5/3/2017 • 1 min to read • Edit Online

- Building My First Claims-Aware ASP.NET Web Application
- Building My First Claims-Aware WCF Service

See Also

Windows Identity Foundation

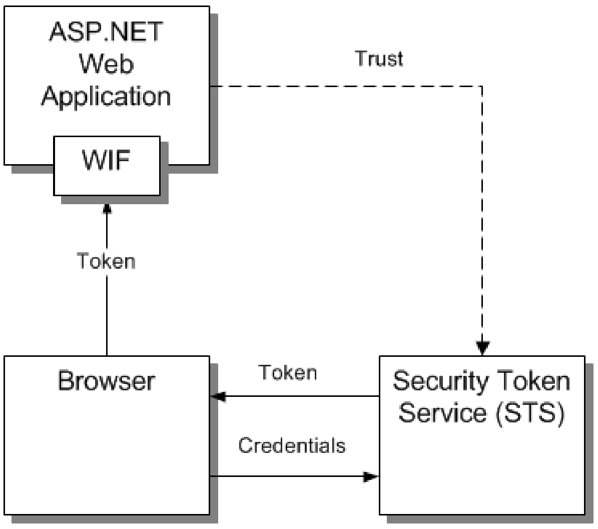
Building My First Claims-Aware ASP.NET Web Application

5/3/2017 • 1 min to read • Edit Online

Applies To

- Windows Identity Foundation (WIF)
- ASP.NET

This topic outlines the scenario of building claims-aware ASP.NET web applications using WIF. There are usually three participants in a claims-aware application scenario: the application itself, the end user, and the Security Token Service (STS). The following figure describes this scenario:



1. The claims-aware application uses WIF to identify unauthenticated requests and to redirect them to the STS.
2. The end user provides credentials to the STS and upon successful authentication the user is issued a token by the STS.
3. The user is redirected from the STS to the claims-aware application with the STS-issued token in the request.
4. The claims-aware application is configured to trust the STS and the tokens it issues. The claims-aware application uses WIF to validate the token and to parse it. Developers use the appropriate WIF API and types, for example, **ClaimsPrincpal** for the application's needs, such as implementing authorization for it.

Starting from .NET 4.5, WIF is part of the .NET framework package. Having the WIF classes directly available in the framework itself allows a much deeper integration of claims-based identity in the .NET platform, making it easier to use claims. With WIF 4.5, you do not need to install any out-of-band components in order to start developing claims-aware web applications. WIF classes are now spread across various assemblies, the main ones being System.Security.Claims, System.IdentityModel and System.IdentityModel.Services.

STS is a service that issues tokens upon successful authentication. Microsoft offers two industry standard STS's:

- Active Directory Federation Services (AD FS) 2.0 (http://go.microsoft.com/fwlink/?LinkID=247516)
- Windows Azure Access Control Service (ACS) (http://go.microsoft.com/fwlink/?LinkID=247517).

AD FS 2.0 is part of the Windows Server R2 and can be used as an STS for on-premise scenarios. ACS is a cloud service, offered as part of the Windows Azure Platform. For testing or educational purposes, you can also use other STS's in order to build your claims-aware applications. For example, you can use the Local Development STS that is part of the Identity and Access Tool for Visual Studio (http://go.microsoft.com/fwlink/?LinkID=245849) which is freely available online.

To build your first claims-aware ASP.NET application using WIF, follow the instructions in one of the following:

- How To: Build Claims-Aware ASP.NET MVC Web Application Using WIF
- How To: Build Claims-Aware ASP.NET Web Forms Application Using WIF
- How To: Build Claims-Aware ASP.NET Application Using Forms-Based Authentication

See Also

Getting Started With WIF

Building My First Claims-Aware WCF Service

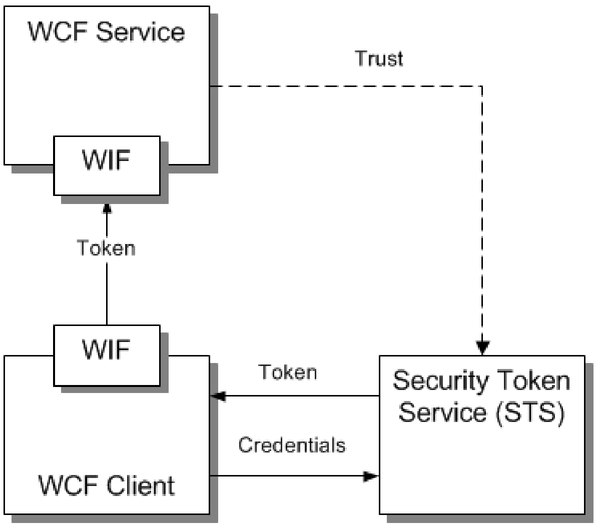
5/3/2017 • 1 min to read • Edit Online

Applies To

- Windows Identity Foundation (WIF)
- Windows Communication Foundation (WCF)

Overview

This topic outlines the scenario of building claims-aware WCF services using WIF. There are usually three participants in a claims-aware web service scenario: the web service itself, the end user, and the Security Token Service (STS). The following figure describes this scenario:



1. The WCF service client (sometimes called agent) uses WIF to send credentials to the STS and upon successful authentication, the agent is issued a token by the STS.

2. The agent sends this STS-issued token to the WCF service.

3. The claims-aware WCF service is configured to trust the STS and the tokens it issues. The claims-aware WCF service uses WIF to validate the token and to parse it. Developers use the appropriate WIF API and types, for example, **ClaimsPrincipal** for the application's needs, such as implementing authorization for it.

Starting from .NET 4.5, WIF is part of the .NET framework package. Having the WIF classes directly available in the framework itself allows a much deeper integration of claims-based identity in the .NET platform, making it easier to use claims. With WIF 4.5, you do not need to install any out-of-band components in order to start developing claims-aware web applications. WIF classes are now spread across various assemblies, the main ones being System.Security.Claims, System.IdentityModel and System.IdentityModel.Services.

STS is a service that issues tokens upon successful authentication. Microsoft offers two industry standard STS's:

- Active Directory Federation Services (AD FS) 2.0 (http://go.microsoft.com/fwlink/?LinkID=247516)
- Windows Azure Access Control Service (ACS) (http://go.microsoft.com/fwlink/?LinkID=247517).

AD FS 2.0 is part of the Windows Server R2 and can be used as an STS for on-premise scenarios. Azure Active Directory Access Control (also known as Access Control Service or ACS) is a cloud service, offered as part of Microsoft Azure. For testing or educational purposes, you can also use other STS's in order to build your claims-aware applications. For example, you can use the Local Development STS that is part of the Identity and Access Tool for Visual Studio (http://go.microsoft.com/fwlink/?LinkID=245849) which is freely available online.

To build your first claims-aware WCF service using WIF, see How To: Build Claims-Aware WCF Service Using WIF.

See Also

Getting Started With WIF

WIF Features

5/3/2017 • 1 min to read • Edit Online

- Identity and Access Tool for Visual Studio 2012
- WIF Session Management
- WIF and Web Farms
- WSFederation Authentication Module Overview
- WSTrustChannelFactory and WSTrustChannel

See Also

Windows Identity Foundation

Custom Token Handlers

6/2/2017 • 1 min to read • Edit Online

This topic discusses token handlers in WIF and how they are used to process tokens. The topic also covers what is necessary to create custom token handlers for token types that are not supported by default in WIF.

Introduction to Token Handlers in WIF

WIF relies on security token handlers to create, read, write, and validate tokens for a relying party (RP) application or a security token service (STS). Token handlers are extensibility points for you to add a custom token handler in the WIF pipeline, or to customize the way that an existing token handler manages tokens. WIF provides nine built-in security token handlers that can be modified or entirely overridden to change the functionality as necessary.

Built-In Security Token Handlers in WIF

WIF 4.5 includes nine security token handler classes that derive from the abstract base class SecurityTokenHandler:

- EncryptedSecurityTokenHandler
- KerberosSecurityTokenHandler
- RsaSecurityTokenHandler
- SamlSecurityTokenHandler
- Saml2SecurityTokenHandler
- SessionSecurityTokenHandler
- UserNameSecurityTokenHandler
- WindowsUserNameSecurityTokenHandler
- X509SecurityTokenHandler

Adding a Custom Token Handler

Some token types, such as Simple Web Tokens (SWT) and JSON Web Tokens (JWT) do not have built-in token handlers provided by WIF. For these token types and for others that do not have a built-in handler, you need to perform the following steps to create a custom token handler.

Adding a custom token handler

1. Create a new class that derives from SecurityTokenHandler.
2. Override the following methods and provide your own implementation:
   - CanReadToken
   - ReadToken
   - CanWriteToken
   - WriteToken
   - CanValidateToken
   - ValidateToken

3. Add a reference to the new custom token handler in the *Web.config* or *App.config* file, within the **<system.identityModel>** section that applies to WIF. For example, the following configuration markup specifies a new token handler named **MyCustomTokenHandler** that resides in the **CustomToken** namespace.

```
<system.identityModel>
    <identityConfiguration saveBootstrapContext="true">
        <securityTokenHandlers>
            <add type="CustomToken.MyCustomTokenHandler, CustomToken" />
        </securityTokenHandlers>
    </identityConfiguration>
</system.identityModel>
```

Note that if you are providing your own token handler to handle a token type that already has a built-in token handler, you need to add a **<remove>** element to drop the default handler and use your custom handler instead. For example, the following configuration replaces the default SamlSecurityTokenHandler with the custom token handler:

```
<system.identityModel>
    <identityConfiguration saveBootstrapContext="true">
        <securityTokenHandlers>
            <remove type="System.IdentityModel.Tokens.SamlSecurityTokenHandler, System.IdentityModel,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=abcdefg123456789">
            <add type="CustomToken.MyCustomTokenHandler, CustomToken" />
        </securityTokenHandlers>
    </identityConfiguration>
</system.identityModel>
```

Identity and Access Tool for Visual Studio 2012

5/16/2017 • 1 min to read • Edit Online

This topic describes the new Identity and Access Tool for Visual Studio 11. You can download this tool from the following URL: http://go.microsoft.com/fwlink/?LinkID=245849 or directly from within Visual Studio 11 by searching for "identity" directly in the Extensions Manager.

The Identity and Access Tool for Visual Studio 11 delivers a dramatically simplified development-time experience with the following highlights:

- Using the new tool, you can develop using web applications project types and target IIS express.
- Unlike with the blanket protection-only authentication, with the new tool, you can specify your local home realm discovery page/controller (or any other endpoint handling the authentication experience within your application) and WIF will configure all unauthenticated requests to go there, instead of redirecting them to the STS.
- The tool includes a test Security Token Service (STS) which runs on your local machine when you launch a debug session. Therefore, you no longer need to create custom STS projects and tweak them in order to get the claims you need to test your applications. The claim types and values are fully customizable.
- You can modify common settings directly via the tool's UI, without the need to edit web.config.
- You can establish federation with Active Directory Federation Services (AD FS) 2.0 (or other WS-Federation providers) in a single screen.
- The tool leverages Windows Azure Access Control Service (ACS) capabilities with a simple list of checkboxes for all the identity providers that you want to use: Facebook, Google, Live ID, Yahoo!, any OpenID provider, and any WS-Federation provider. Select your identity providers, click OK, then F5, and both your application and ACS will be automatically configured and your test application will be ACS-aware.

See Also

WIF Features

WIF Session Management

5/16/2017 • 3 min to read • Edit Online

When a client first tries to access a protected resource that is hosted by a relying party, the client must first authenticate itself to a security token service (STS) that is trusted by the relying party. The STS then issues a security token to the client. The client presents this token to the relying party, which then grants the client access to the protected resource. However, you don't want the client to have to re-authenticate to the STS for each request, especially because it might not even be on the same computer or in the same domain as the relying party. Instead, Windows Identity Foundation (WIF) has the client and relying party establish a session in which the client uses a session security token to authenticate itself to the relying party for all requests after the first request. The relying party can use this session security token, which is stored inside a cookie, to reconstruct the client's System.Security.Claims.ClaimsPrincipal.

The STS defines what authentication the client must provide. However, the client might have multiple credentials with which it can authenticate itself to the STS. For example, it might have a token from Windows Live, a user name and password, a certificate, and a smartkey. In that case, the STS grants the client several identities, with each identity corresponding to one of the credentials that the client presents. The relying party can use one or more of these identities when it decides what level of access to grant the client.

The System.IdentityModel.Tokens.SessionSecurityToken is used to reconstruct the client's System.Security.Claims.ClaimsPrincipal, which contains all of the client's identities in Identities. Each System.Security.Claims.ClaimsIdentity in the collection contains the bootstrap tokens that are associated with that identity.

If a new session token is issued with the session ID of the original session token, System.IdentityModel.Tokens.SessionSecurityTokenHandler does not update the session token in the token cache. You should always instantiate a session token with a unique session ID.

NOTE

Session.SecurityTokenHandler.ReadToken throws a XmlException exception if it receives invalid input; for example, if the cookie that contains the session token is corrupted. We recommend that you catch this exception and provide application-specific behavior.

If a protected Web page contains lots of resources (such as small graphics) that are also in the protected domain, the client must re-authenticate itself to the relying party to download each of those resources. Use of a session authentication token avoids the need to authenticate to the STS for each request, but it still means that many cookies are being sent over. You might want to set up the Web page so that the important data and resources are stored in the protected domain while minor items are stored in an unprotected domain and linked to from the main Web page. Also, set the cookie path to reference only the protected domain.

To operate in reference mode, Microsoft recommends providing a handler for the SessionSecurityTokenCreated event in the **global.asax.cs** file and setting the **IsReferenceMode** property on the token passed in the SessionToken property. These updates will ensure that the session token operates in reference mode for every request and is favored over merely setting the IsReferenceMode property on the Session Authentication Module.

Extensibility

You can extend the session management mechanism. One reason for this would be to improve the performance. For example, you could create a custom cookie handler that transforms or optimizes the session security token between its in-memory state and what goes into the cookie. To do so, you can configure the System.IdentityModel.Services.SessionAuthenticationModule.CookieHandler property of the System.IdentityModel.Services.SessionAuthenticationModule to use a custom cookie handler that derives from System.IdentityModel.Services.CookieHandler. System.IdentityModel.Services.ChunkedCookieHandler is the default cookie handler because the cookies exceed the allowable size for Hypertext Transfer Protocol (HTTP); if you use a custom cookie handler instead, you must implement chunking.

For more information, see ClaimsAwareWebFarm (http://go.microsoft.com/fwlink/?LinkID=248408) sample. This sample shows a farm ready session cache (as opposed to a tokenreplycache) so that you can use sessions by reference instead of exchanging big cookies; this sample also demonstrates an easier way of securing cookies in a farm. The session cache is WCF-based. With regard to session securing, the sample demonstrates a new capability in WIF 4.5 of a cookie transform based on MachineKey, which can be activated by simply pasting the appropriate snippet in the web.config. The sample itself is not "farmed", but it demonstrates what you need for making your app farm-ready.

WIF and Web Farms

6/2/2017 • 7 min to read • Edit Online

When you use Windows Identity Foundation (WIF) to secure the resources of a relying party (RP) application that is deployed in a web farm, you must take specific steps to ensure that WIF can process tokens from instances of the RP application running on different computers in the farm. This processing includes validating session token signatures, encrypting and decrypting session tokens, caching session tokens, and detecting replayed security tokens.

In the typical case, when WIF is used to secure resources of an RP application – whether the RP is running on a single computer or in a web farm -- a session is established with the client based on the security token that was obtained from the security token service (STS). This is to avoid forcing the client to have to authenticate at the STS for every application resource that is secured using WIF. For more information about how WIF handles sessions, see WIF Session Management.

When default settings are used, WIF does the following:

- It uses an instance of the SessionSecurityTokenHandler class to read and write a session token (an instance of the SessionSecurityToken class) that carries the claims and other information about the security token that was used for authentication as well as information about the session itself. The session token is packaged and stored in a session cookie. By default, SessionSecurityTokenHandler uses the ProtectedDataCookieTransform class, which uses the Data Protection API (DPAPI), to protect the session token. The DPAPI provides protection by using the user or machine credentials and stores the key data in the user profile.
- It uses a default, in-memory implementation of the SessionSecurityTokenCache class to store and process the session token.

These default settings work in scenarios in which the RP application is deployed on a single computer; however, when deployed in a web farm, each HTTP request may be sent to and processed by a different instance of the RP application running on a different computer. In this scenario, the default WIF settings described above will not work because both token protection and token caching are dependent on a specific computer.

To deploy an RP application in a web farm, you must ensure that the processing of session tokens (as well as of replayed tokens) is not dependent on the application running on a specific computer. One way to do this is to implement your RP application so that it uses the functionality provided by the ASP.NET `<machineKey>` configuration element and provides distributed caching for processing session tokens and replayed tokens. The `<machineKey>` element allows you to specify the keys needed to validate, encrypt, and decrypt tokens in a configuration file, which enables you to specify the same keys on different computers in the web farm. WIF provides a specialized session token handler, the MachineKeySessionSecurityTokenHandler, that protects tokens by using the keys specified in the `<machineKey>` element. To implement this strategy, you can follow these guidelines:

- Use the ASP.NET `<machineKey>` element in configuration to explicitly specify signing and encryption keys that can be used across computers in the farm. The following XML shows the specification of the `<machineKey>` element under the `<system.web>` element in a configuration file.

  ```
  <machineKey compatibilityMode="Framework45" decryptionKey="CC510D … 8925E6" validationKey="BEAC8 …
  6A4B1DE" />
  ```

- Configure the application to use the MachineKeySessionSecurityTokenHandler by adding it to the token handler collection. You must first remove the SessionSecurityTokenHandler (or any handler derived from the SessionSecurityTokenHandler class) from the token handler collection if such a handler is present. The MachineKeySessionSecurityTokenHandler uses the MachineKeyTransform class, which protects the session cookie data by using the cryptographic material specified in the `<machineKey>` element. The following XML shows how to add the MachineKeySessionSecurityTokenHandler to a token handler collection.

```
<securityTokenHandlers>
  <remove type="System.IdentityModel.Tokens.SessionSecurityTokenHandler, System.IdentityModel,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
  <add type="System.IdentityModel.Services.Tokens.MachineKeySessionSecurityTokenHandler,
System.IdentityModel.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
</securityTokenHandlers>
```

- Derive from SessionSecurityTokenCache and implement distributed caching, that is, a cache that is accessible from all computers in the farm on which the RP might run. Configure the RP to use your distributed cache by specifying the <sessionSecurityTokenCache> element in the configuration file. You can override the System.IdentityModel.Tokens.SessionSecurityTokenCache.LoadCustomConfiguration method in your derived class to implement child elements of the `<sessionSecurityTokenCache>` element if they are required.

```
<caches>
  <sessionSecurityTokenCache type="MyCacheLibrary.MySharedSessionSecurityTokenCache, MyCacheLibrary">
    <!—optional child configuration elements, if implemented by the derived class -->
  </sessionSecurityTokenCache>
</caches>
```

  One way to implement distributed caching is to provide a WCF front end for your custom cache. For more information about implementing a WCF caching service, see The WCF Caching Service. For more information about implementing a WCF client that the RP application can use to call the caching service, see The WCF Caching Client.

- If your application detects replayed tokens you must follow a similar distributed caching strategy for the token replay cache by deriving from TokenReplayCache and pointing to your token replay caching service in the <tokenReplayCache> configuration element.

IMPORTANT

All of the example XML and code in this topic is taken from the ClaimsAwareWebFarm (http://go.microsoft.com/fwlink/?LinkID=248408) sample.

IMPORTANT

The examples in this topic are provided as-is and are not intended to be used in production code without modification.

The WCF Caching Service

The following interface defines the contract between the WCF caching service and the WCF client used by the relying party application to communicate with it. It essentially exposes the methods of the SessionSecurityTokenCache class as service operations.

```
[ServiceContract()]
public interface ISessionSecurityTokenCacheService
{
    [OperationContract]
    void AddOrUpdate(string endpointId, string contextId, string keyGeneration, SessionSecurityToken value,
DateTime expiryTime);

    [OperationContract]
    IEnumerable<SessionSecurityToken> GetAll(string endpointId, string contextId);

    [OperationContract]
    SessionSecurityToken Get(string endpointId, string contextId, string keyGeneration);

    [OperationContract(Name = "RemoveAll")]
    void RemoveAll(string endpointId, string contextId);

    [OperationContract(Name = "RemoveAllByEndpointId")]
    void RemoveAll(string endpointId);

    [OperationContract]
    void Remove(string endpointId, string contextId, string keyGeneration);
}
```

The following code shows the implementation of the WCF caching service. In this example, the default, in-memory session token cache implemented by WIF is used. Alternatively, you could implement a durable cache backed by a database. `ISessionSecurityTokenCacheService` defines the interface shown above. In this example, not all of the methods required to implement the interface are shown for brevity.

```
using System;
using System.Collections.Generic;
using System.IdentityModel.Configuration;
using System.IdentityModel.Tokens;
using System.ServiceModel;
using System.Xml;

namespace WcfSessionSecurityTokenCacheService
{
    [ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
    public class SessionSecurityTokenCacheService : ISessionSecurityTokenCacheService
    {
        SessionSecurityTokenCache internalCache;

        // sets the internal cache used by the service to the default WIF in-memory cache.
        public SessionSecurityTokenCacheService()
        {
            internalCache = new IdentityModelCaches().SessionSecurityTokenCache;
        }

        ...

        public SessionSecurityToken Get(string endpointId, string contextId, string keyGeneration)
        {
            // Delegates to the default, in-memory MruSessionSecurityTokenCache used by WIF
            SessionSecurityToken token = internalCache.Get(new SessionSecurityTokenCacheKey(endpointId,
GetContextId(contextId), GetKeyGeneration(keyGeneration)));
            return token;
        }

        ...

        private static UniqueId GetContextId(string contextIdString)
        {
            return contextIdString == null ? null : new UniqueId(contextIdString);
        }

        private static UniqueId GetKeyGeneration(string keyGenerationString)
        {
            return keyGenerationString == null ? null : new UniqueId(keyGenerationString);
        }
    }
}
```

The WCF Caching Client

This section shows the implementation of a class that derives from SessionSecurityTokenCache and that delegates calls to the caching service. You configure the RP application to use this class through the <sessionSecurityTokenCache> element as in the following XML

```
<caches>
  <sessionSecurityTokenCache type="CacheLibrary.SharedSessionSecurityTokenCache, CacheLibrary">
    <!--cacheServiceAddress points to the centralized session security token cache service running in the web
farm.-->
    <cacheServiceAddress url="http://localhost:4161/SessionSecurityTokenCacheService.svc" />
  </sessionSecurityTokenCache>
</caches>
```

The class overrides the LoadCustomConfiguration method to get the service endpoint from the custom `<cacheServiceAddress>` child element of the `<sessionSecurityTokenCache>` element. It uses this endpoint to initialize an `ISessionSecurityTokenCacheService` channel over which it can communicate with the service. In this example, not all of the methods required to implement the SessionSecurityTokenCache class are shown for brevity.

```
using System;
using System.Configuration;
using System.IdentityModel.Configuration;
using System.IdentityModel.Tokens;
using System.ServiceModel;
using System.Xml;

namespace CacheLibrary
{
    /// <summary>
    /// This class acts as a proxy to the WcfSessionSecurityTokenCacheService.
    /// </summary>
    public class SharedSessionSecurityTokenCache : SessionSecurityTokenCache, ICustomIdentityConfiguration
    {
        private ISessionSecurityTokenCacheService WcfSessionSecurityTokenCacheServiceClient;

        internal SharedSessionSecurityTokenCache()
        {
        }

        /// <summary>
        /// Creates a client channel to call the service host.
        /// </summary>
        protected void Initialize(string cacheServiceAddress)
        {
            if (this.WcfSessionSecurityTokenCacheServiceClient != null)
            {
                return;
            }

            ChannelFactory<ISessionSecurityTokenCacheService> cf = new
ChannelFactory<ISessionSecurityTokenCacheService>(
                new WS2007HttpBinding(SecurityMode.None),
                new EndpointAddress(cacheServiceAddress));
            this.WcfSessionSecurityTokenCacheServiceClient = cf.CreateChannel();
        }

        #region SessionSecurityTokenCache Members
        // Delegates the following operations to the centralized session security token cache service in the
web farm.

        ...

        public override SessionSecurityToken Get(SessionSecurityTokenCacheKey key)
        {
            return this.WcfSessionSecurityTokenCacheServiceClient.Get(key.EndpointId, GetContextIdString(key),
GetKeyGenerationString(key));
        }

        ...

        #endregion

        #region ICustomIdentityConfiguration Members
        // Called from configuration infrastructure to load custom elements
        public void LoadCustomConfiguration(XmlNodeList nodeList)
        {
            // Retrieve the endpoint address of the centralized session security token cache service running
in the web farm
            if (nodeList.Count == 0)
            {
                throw new ConfigurationException("No child config element found under
<sessionSecurityTokenCache>.");
            }

            XmlElement cacheServiceAddressElement = nodeList.Item(0) as XmlElement;
            if (cacheServiceAddressElement.LocalName != "cacheServiceAddress")
            {
                throw new ConfigurationException("First child config element under <sessionSecurityTokenCache> 
is expected to be <cacheServiceAddress>.");
            }

            string cacheServiceAddress = null;
            if (cacheServiceAddressElement.Attributes["url"] != null)
            {
                cacheServiceAddress = cacheServiceAddressElement.Attributes["url"].Value;
            }
            else
            {
                throw new ConfigurationException("<cacheServiceAddress> is expected to contain a 'url' 
attribute.");
            }

            // Initialize the proxy to the WebFarmSessionSecurityTokenCacheService
            this.Initialize(cacheServiceAddress);
        }
        #endregion

        private static string GetKeyGenerationString(SessionSecurityTokenCacheKey key)
        {
            return key.KeyGeneration == null ? null : key.KeyGeneration.ToString();
        }

        private static string GetContextIdString(SessionSecurityTokenCacheKey key)
        {
            return GetContextIdString(key.ContextId);
        }

        private static string GetContextIdString(UniqueId contextId)
        {
            return contextId == null ? null : contextId.ToString();
        }
    }
}
```

See Also

SessionSecurityTokenCache
SessionSecurityTokenHandler
MachineKeySessionSecurityTokenHandler
WIF Session Management

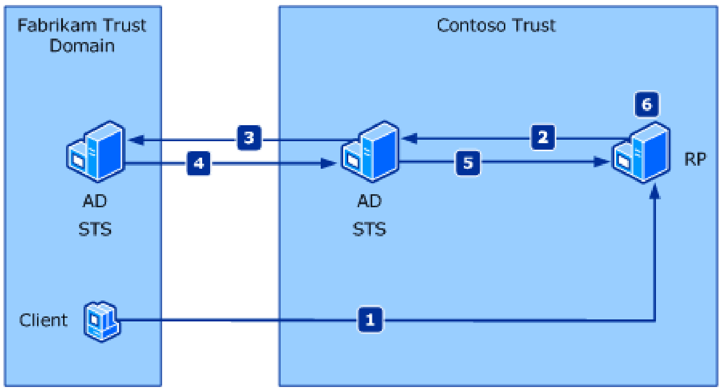
WSFederation Authentication Module Overview

6/2/2017 • 7 min to read • Edit Online

Windows Identity Foundation (WIF) includes support for federated authentication in ASP.NET applications through the WS-Federated Authentication Module (WS-FAM). This topic will help you understand how federated authentication works and how to use it.

Overview of Federated Authentication

Federated authentication allows a Security Token Service (STS) in one trust domain to provide authentication information to an STS in another trust domain when there is a trust relationship between the two domains. An example of this is shown in the following illustration.



1. A client in the Fabrikam trust domain sends a request to a Relying Party (RP) application in the Contoso trust domain.

2. The RP redirects the client to an STS in the Contoso trust domain. This STS has no knowledge of the client.

3. The Contoso STS redirects the client to an STS in the Fabrikam trust domain, with which the Contoso trust domain has a trust relationship.

4. The Fabrikam STS verifies the client's identity and issues a security token to the Contoso STS.

5. The Contoso STS uses the Fabrikam token to create its own token that can be used by the RP and sends it to the RP.

6. The RP extracts the client's claims from the security token and makes an authorization decision.

Using the Federated Authentication Module with ASP.NET

WSFederationAuthenticationModule (WS-FAM) is an HTTP module that lets you add federated authentication to an ASP.NET application. Federated authentication lets authentication logic be handled by the STS and lets you focus on writing business logic.

You configure the WS-FAM to specify the STS to which non-authenticated requests should be redirected. WIF lets you authenticate a user in two ways:

1. Passive redirect: When an unauthenticated user tries to access a protected resource, and you want to simply redirect them to an STS without requiring a login page, then this is the right approach. The STS verifies the user's identity, and issues a security token that contains the appropriate claims for that user. This option requires the WS-FAM to be added in the HTTP Modules pipeline. You can use the Identity and Access Tool for Visual Studio 2012 to modify your application's configuration file to use the WS-FAM and to federate with an STS. For more information, see Identity and Access Tool for Visual Studio 2012.

2. You can call the System.IdentityModel.Services.WSFederationAuthenticationModule.SignIn method or the RedirectToIdentityProvider method from the code-behind for a sign-in page in your RP application.

In passive redirect, all communication is performed through response/redirect from the client (typically a browser). You can add the WS-FAM to your application's HTTP pipeline, where it watches for unauthenticated user requests and redirects users to the STS you specify.

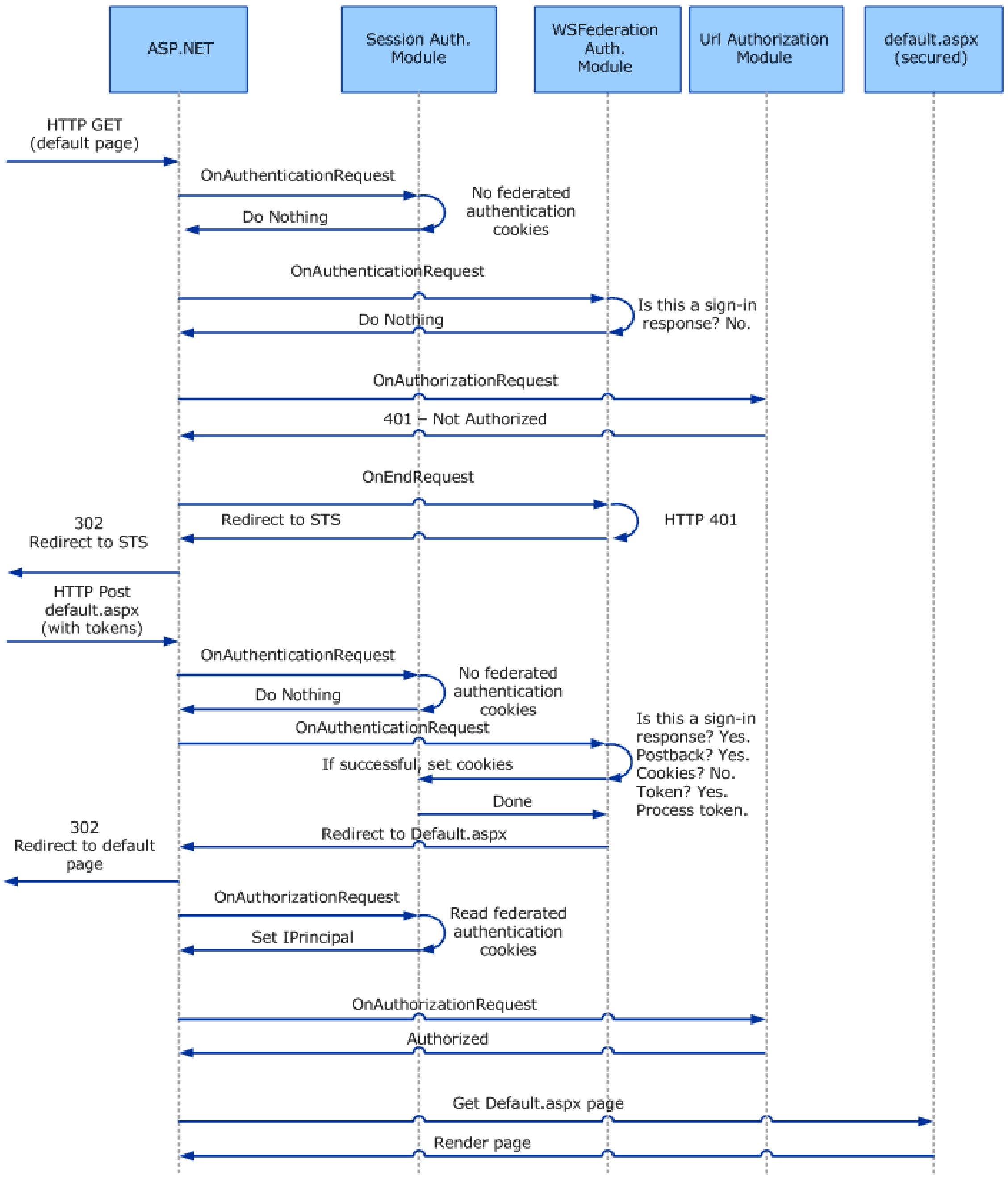
The WS-FAM also raises several events that let you customize its functionality in an ASP.NET application.

How the WS-FAM Works

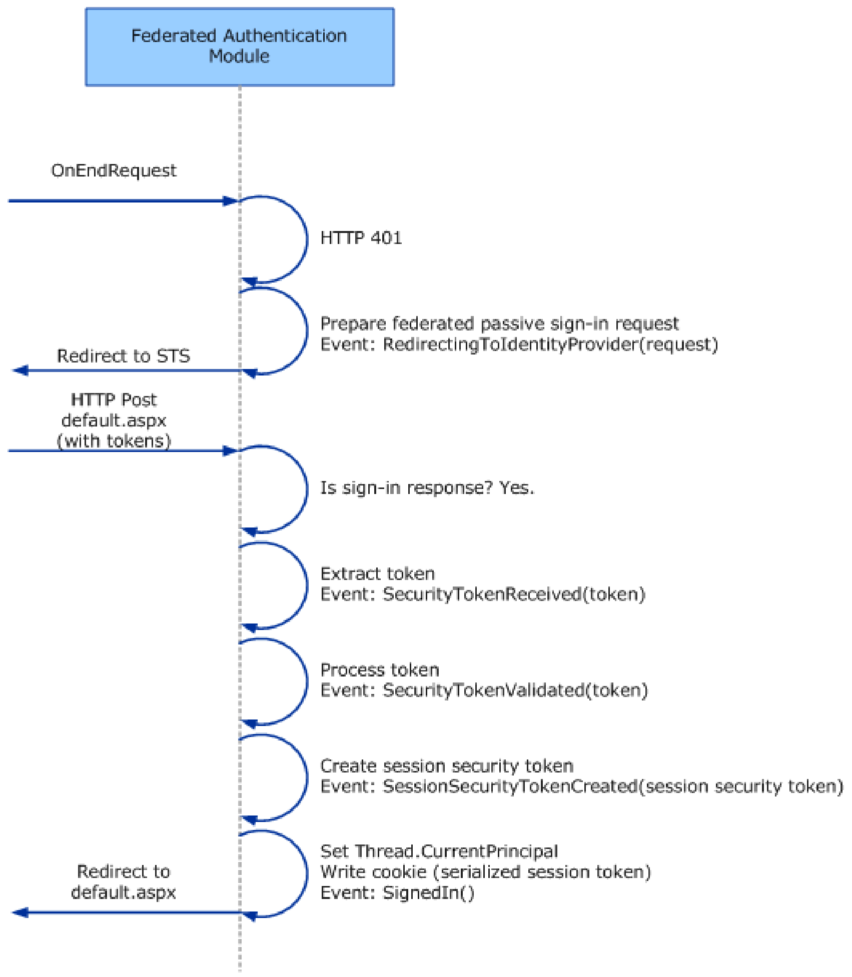
The WS-FAM is implemented in the WSFederationAuthenticationModule class. Typically, you add the WS-FAM to the HTTP pipeline of your ASP.NET RP application. When an unauthenticated user tries to access a protected resource, the RP returns a "401 authorization denied" HTTP response. The WS-FAM intercepts this response instead of allowing the client to receive it, then it redirects the user to the specified STS. The STS issues a security token, which the WS-FAM again intercepts. The WS-FAM uses the token to create an instance of ClaimsPrincipal for the authenticated user, which enables regular .NET Framework authorization mechanisms to function.

Because HTTP is stateless, we need a way to avoid repeating this whole process every time that the user tries to access another protected resource. This is where the SessionAuthenticationModule comes in. When the STS issues a security token for the user, SessionAuthenticationModule also creates a session security token for the user and puts it in a cookie. On subsequent requests, the SessionAuthenticationModule intercepts this cookie and uses it to reconstruct the user's ClaimsPrincipal.

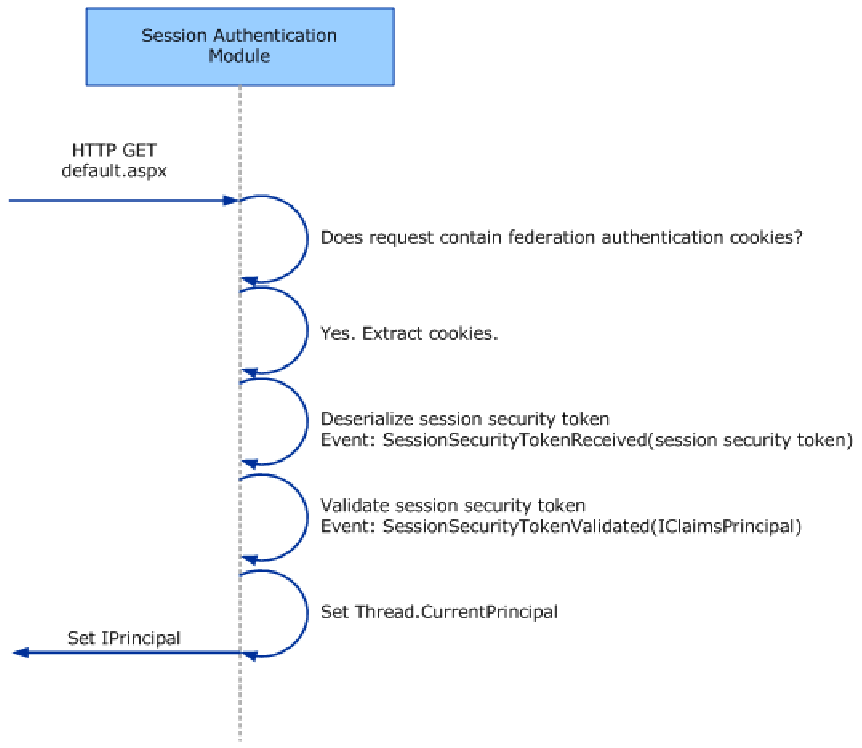
The following diagram shows the overall flow of information in the passive redirect case. The request is automatically redirected via the STS to establish credentials without a login page:



The following diagram shows more detail on what happens when the user has authenticated to the STS and their security tokens are processed by the WSFederationAuthenticationModule:


Federated Authentication Module
OnEndRequest
HTTP 401
Prepare federated passive sign-in request
Event: RedirectingToIdentityProvider(request)
Redirect to STS
HTTP Post default.aspx (with tokens)
Is sign-in response? Yes.
Extract token
Event: SecurityTokenReceived(token)
Process token
Event: SecurityTokenValidated(token)
Create session security token
Event: SessionSecurityTokenCreated(session security token)
Set Thread.CurrentPrincipal
Write cookie (serialized session token)
Event: SignedIn()
Redirect to default.aspx

The following diagram shows more detail on what happens when the user's security tokens have been serialized into cookies and are intercepted by the SessionAuthenticationModule:


Session Authentication Module
HTTP GET default.aspx
Does request contain federation authentication cookies?
Yes. Extract cookies.
Deserialize session security token
Event: SessionSecurityTokenReceived(session security token)
Validate session security token
Event: SessionSecurityTokenValidated(IClaimsPrincipal)
Set Thread.CurrentPrincipal
Set IPrincipal

Events

WSFederationAuthenticationModule, SessionAuthenticationModule, and their parent class, HttpModuleBase, raise events at various stages of processing of an HTTP request. You can handle these events in the `global.asax` file of your ASP.NET application.

- The ASP.NET infrastructure invokes the module's Init method to initialize the module.

- The System.IdentityModel.Services.FederatedAuthentication.FederationConfigurationCreated event is raised when the ASP.NET infrastructure invokes the Init method for the first time on one of the application's modules that derive from HttpModuleBase. This method accesses the static System.IdentityModel.Services.FederatedAuthentication.FederationConfiguration property, which causes configuration to be loaded from the Web.config file. This event is only raised the first time this property is accessed. The FederationConfiguration object that is initialized from configuration can be accessed through the System.IdentityModel.Services.Configuration.FederationConfigurationCreatedEventArgs.FederationConfiguration property in an event handler. You can use this event to modify the configuration before it is applied to any modules. You can add a handler for this event in the Application_Start method:

  ```
  void Application_Start(object sender, EventArgs e)
  {
      FederatedAuthentication.FederationConfigurationCreated += new
  EventHandler<FederationConfigurationCreatedEventArgs>
  (FederatedAuthentication_FederationConfigurationCreated);
  }
  ```

  Each module overrides the System.IdentityModel.Services.HttpModuleBase.InitializeModule and System.IdentityModel.Services.HttpModuleBase.InitializePropertiesFromConfiguration abstract methods. The first of these methods adds handlers for ASP.NET pipeline events that are of interest to the module. In most cases the module's default implementation will suffice. The second of these methods initializes the module's properties from its System.IdentityModel.Services.HttpModuleBase.FederationConfiguration property. (This is a copy of the configuration that was loaded previously.) You may need to override this second method if you want to support the initialization of new properties from configuration in classes that you derive from WSFederationAuthenticationModule or SessionAuthenticationModule. In such cases you would also need to derive from the appropriate configuration objects to support the added configuration properties; for example, from IdentityConfiguration, WsFederationConfiguration, or FederationConfiguration.

- The WS-FAM raises the SecurityTokenReceived event when it intercepts a security token that has been issued by the STS.

- The WS-FAM raises the SecurityTokenValidated event after it has validated the token.

- The SessionAuthenticationModule raises the SessionSecurityTokenCreated event when it creates a session security token for the user.

- The SessionAuthenticationModule raises the SessionSecurityTokenReceived event when it intercepts subsequent requests with the cookie that contains the session security token.

- Before the WS-FAM redirects the user to the issuer, it raises the RedirectingToIdentityProvider event. The WS-Federation sign-in request is available through the SignInRequestMessage property of the RedirectingToIdentityProviderEventArgs passed in the event. You may choose to modify the request before sending this out to the issuer.

- The WS-FAM raises the SignedIn event when the cookie is successfully written and the user is signed in.

- The WS-FAM raises the SigningOut event one time per session as the session is being closed down for each user. It is not raised if the session is closed down on the client-side (for example, by deleting the session cookie). In an SSO environment, the IP-STS can request each RP to sign out, too. This will also raise this event, with IsIPInitiated set to `true`.

NOTE

You should not use the System.Threading.Thread.CurrentPrincipal property during any event raised by WSFederationAuthenticationModule or SessionAuthenticationModule. This is because System.Threading.Thread.CurrentPrincipal is set after the authentication process, while events are raised during the authentication process.

Configuration of Federated Authentication

The WS-FAM and SAM are configured through the <federationConfiguration> element. The <wsFederation> child element configures default values for the WS-FAM properties; such as the Issuer property and the Realm property. (These values can be changed on a per request basis by providing handlers for some of the WS-FAM events; for example, RedirectingToIdentityProvider.) The cookie handler that is used by the SAM is configured through the <cookieHandler> child element. WIF provides a default cookie handler implemented in the ChunkedCookieHandler class that can have its chunk size set through the <chunkedCookieHandler> element. The `<federationConfiguration>` element references an IdentityConfiguration, which provides configuration for other WIF components used in the application, such as the ClaimsAuthenticationManager and the ClaimsAuthorizationManager. The identity configuration may be referenced explicitly by specifying a named <identityConfiguration> element in the `identityConfigurationName` attribute of the `<federationConfiguration>` element. If the identity configuration is not referenced explicitly, the default identity configuration will be used.

The following XML shows a configuration of an ASP.NET relying party (RP) application. The SystemIdentityModelSection and SystemIdentityModelServicesSection configuration sections are added under the `<configSections>` element. The SAM and WS-FAM are added to the HTTP Modules under the `<system.webServer>`/ `<modules>` element. Finally the WIF components are configured under the `<system.identityModel>`/ `<identityConfiguration>` and `<system.identityModel.services>`/`<federationConfiguration>` elements. This configuration specifies the chunked cookie handler as it is the default cookie handler and there is not a cookie handler type specified in the `<cookieHandler>` element.

WARNING

In the following example, both the `requireHttps` attribute of the `<wsFederation>` element and the `requireSsl` attribute of the `<cookieHandler>` element are `false`. This presents a potential security threat. In production, both these values should be set `true`.

```
<configuration>
  <configSections>
    <section name="system.identityModel" type="System.IdentityModel.Configuration.SystemIdentityModelSection,
System.IdentityModel, Version=4.0.0.0, Culture=neutral, PublicKeyToken=B77A5C561934E089" />
    <section name="system.identityModel.services"
type="System.IdentityModel.Services.Configuration.SystemIdentityModelServicesSection,
System.IdentityModel.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=B77A5C561934E089" />
  </configSections>

  ...

  <system.webServer>
    <modules>
      <add name="WSFederationAuthenticationModule"
type="System.IdentityModel.Services.WSFederationAuthenticationModule, System.IdentityModel.Services,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" preCondition="managedHandler" />
      <add name="SessionAuthenticationModule" type="System.IdentityModel.Services.SessionAuthenticationModule,
System.IdentityModel.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
preCondition="managedHandler" />
    </modules>
  </system.webServer>

  <system.identityModel>
    <identityConfiguration>
      <audienceUris>
        <add value="http://localhost:50969/" />
      </audienceUris>
      <certificateValidation certificateValidationMode="None" />
      <issuerNameRegistry type="System.IdentityModel.Tokens.ConfigurationBasedIssuerNameRegistry,
System.IdentityModel, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089">
        <trustedIssuers>
          <add thumbprint="9B74CB2F320F7AAFC156E1252270B1DC01EF40D0" name="LocalSTS" />
        </trustedIssuers>
      </issuerNameRegistry>
    </identityConfiguration>
  </system.identityModel>
  <system.identityModel.services>
    <federationConfiguration>
      <wsFederation passiveRedirectEnabled="true" issuer="http://localhost:15839/wsFederationSTS/Issue"
realm="http://localhost:50969/" reply="http://localhost:50969/" requireHttps="false" />
      <cookieHandler requireSsl="false" />
    </federationConfiguration>
  </system.identityModel.services>
</configuration>
```

See Also

SessionAuthenticationModule
WSFederationAuthenticationModule
<federationConfiguration>

WSTrustChannelFactory and WSTrustChannel

5/16/2017 • 6 min to read • Edit Online

If you are already familiar with Windows Communication Foundation (WCF), you know that a WCF client is already federation aware. By configuring a WCF client with a WSFederationHttpBinding or similar custom binding, you can enable federated authentication to a service.

WCF obtains the token that is issued by the security token service (STS) behind the scenes and uses this token to authenticate to the service. The main limitation to this approach is that there is no visibility into the client's communications with the server. WCF automatically generates the request security token (RST) to the STS based on the issued token parameters on the binding. This means that the client cannot vary the RST parameters per request, inspect the request security token response (RSTR) to get information such as display claims, or cache the token for future use.

Currently, the WCF client is suitable for basic federation scenarios. However, one of the major scenarios that Windows Identity Foundation (WIF) supports requires control over the RST at a level that WCF does not easily allow. Therefore, WIF adds features that give you more control over communication with the STS.

WIF supports the following federation scenarios:

- Using a WCF client without any WIF dependencies to authenticate to a federated service
- Enabling WIF on a WCF client to insert an ActAs or OnBehalfOf element into the RST to the STS
- Using WIF alone to obtain a token from the STS and then enable a WCF client to authenticate with this token. For more information, see ClaimsAwareWebService sample.

The first scenario is self-explanatory: Existing WCF clients will continue to work with WIF relying parties and STSs. This topic discusses the remaining two scenarios.

Enhancing an Existing WCF Client with ActAs / OnBehalfOf

In a typical identity delegation scenario, a client calls a middle-tier service, which then calls a back-end service. The middle-tier service acts as, or acts on behalf of, the client.

TIP

What is the difference between ActAs and OnBehalfOf?

From the WS-Trust procotol standpoint:

1. An ActAs RST element indicates that the requestor wants a token that contains claims about two distinct entities: the requestor, and an external entity represented by the token in the ActAs element.
   b. An OnBehalfOf RST element indicates that the requestor wants a token that contains claims only about one entity: the external entity represented by the token in the OnBehalfOf element.

The ActAs feature is typically used in scenarios that require composite delegation, where the final recipient of the issued token can inspect the entire delegation chain and see not just the client, but all intermediaries. This lets it perform access control, auditing and other related activities based on the entire identity delegation chain. The ActAs feature is commonly used in multi-tiered systems to authenticate and pass information about identities between the tiers without having to pass this information at the application/business logic layer.

The OnBehalfOf feature is used in scenarios where only the identity of the original client is important and is effectively the same as the identity impersonation feature available in Windows. When OnBehalfOf is used, the final recipient of the issued token can only see claims about the original client, and the information about intermediaries is not preserved. One common pattern where the OnBehalfOf feature is used is the proxy pattern where the client cannot access the STS directly but instead communicates through a proxy gateway. The proxy gateway authenticates the caller and puts information about the caller into the OnBehalfOf element of the RST message that it then sends to the real STS for processing. The resulting token contains only claims related to the client of the proxy, making the proxy completely transparent to the receiver of the issued token.Note that WIF does not support <wsse:SecurityTokenReference> or <wsa:EndpointReferences> as a child of <wst:OnBehalfOf>. The WS-Trust specification allows for three ways to identify the original requestor (on behalf of whom the proxy is acting). These are:

- Security token reference. A reference to a token, either in the message, or possibly retrieved out of band).
  - Endpoint reference. Used as a key to look up data, again out of band.
  - Security token. Identifies the original requestor directly.

WIF supports only security tokens, either encrypted or unencrypted, as a direct child element of <wst:OnBehalfOf>.

This information is conveyed to a WS-Trust issuer using the ActAs and OnBehalfOf token elements in the RST.

WCF exposes an extensibility point on the binding that allows arbitrary XML elements to be added to the RST. However, because the extensibility point is tied to the binding, scenarios that require the RST contents to vary per call must re-create the client for every call, which decreases performance. WIF uses extension methods on the `ChannelFactory` class to allow developers to attach any token that is obtained out of band to the RST. The following code example shows how to take a token that represents the client (such as an X.509, username, or Security Assertion Markup Language (SAML) token) and attach it to the RST that is sent to the issuer.

```
IHelloService serviceChannel = channelFactory.CreateChannelActingAs<IHelloService>( clientSamlToken );
serviceChannel.Hello("Hi!");
```

WIF provides the following benefits:

- The RST can be modified per channel; therefore, middle-tier services do not have to re-create the channel factory for each client, which improves performance.

- This works with existing WCF clients, which makes an easy upgrade path possible for existing WCF middle-tier services that want to enable identity delegation semantics.

However, there is still no visibility into the client's communication with the STS. We'll examine this in the third scenario.

Communicating Directly with an Issuer and Using the Issued Token to Authenticate

For some advanced scenarios, enhancing a WCF client is not enough. Developers who use only WCF typically use Message In / Message Out contracts and handle client-side parsing of the issuer response manually.

WIF introduces the WSTrustChannelFactory and WSTrustChannel classes to let the client communicate directly with a WS-Trust issuer. The WSTrustChannelFactory and WSTrustChannel classes enable strongly typed RST and RSTR objects to flow between the client and issuer, as shown in the following code example.

```
WSTrustChannelFactory trustChannelFactory = new WSTrustChannelFactory( stsBinding, stsAddress );
WSTrustChannel channel = (WSTrustChannel) trustChannelFactory.CreateChannel();
RequestSecurityToken rst = new RequestSecurityToken(RequestTypes.Issue);
rst.AppliesTo = new EndpointAddress(serviceAddress);
RequestSecurityTokenResponse rstr = null;
SecurityToken token = channel.Issue(rst, out rstr);
```

Note that the `out` parameter on the Issue method allows access to the RSTR for client-side inspection.

So far, we've only seen how to obtain a token. The token that is returned from the WSTrustChannel object is a `GenericXmlSecurityToken` that contains all of the information that is necessary for authentication to a relying party. The following code example shows how to use this token.

```
IHelloService serviceChannel = channelFactory.CreateChannelWithIssuedToken<IHelloService>( token );
serviceChannel.Hello("Hi!");
```

The CreateChannelWithIssuedToken extension method on the `ChannelFactory` object indicates to WIF that you have obtained a token out of band, and that it should stop the normal WCF call to the issuer and instead use the token that you obtained to authenticate to the relying party. This has the following benefits:

- It gives you complete control over the token issuance process.
- It supports ActAs / OnBehalfOf scenarios by directly setting these properties on the outgoing RST.
- It enables dynamic client-side trust decisions to be made based on the contents of the RSTR.
- It lets you cache and reuse the token that is returned from the Issue method.
- WSTrustChannelFactory and WSTrustChannel allow for control of channel caching, fault, and recovery semantics according to WCF best practices.

See Also

WIF Features

WIF How-To's Index

5/3/2017 • 1 min to read • Edit Online

How To: Build Claims-Aware ASP.NET MVC Web Application Using WIF

5/23/2017 • 3 min to read • Edit Online

Applies To

- Microsoft® Windows® Identity Foundation (WIF)
- ASP.NET® MVC

Summary

This How-To provides detailed step-by-step procedures for creating simple claims-aware ASP.NET MVC web application. It also provides instructions how to test the simple claims-aware ASP.NET MVC web application for successful implementation of claims-based authentication. This How-To does not have detailed instructions for creating a Security Token Service (STS), and assumes you have already configured an STS.

Contents

Objectives

- Configure ASP.NET MVC web application for claims-based authentication
- Test successful claims-aware ASP.NET MVC web application

Summary of Steps

- Step 1 – Create Simple ASP.NET MVC Application
- Step 2 – Configure ASP.NET MVC Application for Claims-Based Authentication
- Step 3 – Test Your Solution

Step 1 – Create Simple ASP.NET MVC Application

In this step, you will create a new ASP.NET MVC application.

To create simple ASP.NET MVC application

1. Start Visual Studio and click **File**, **New**, and then **Project**.
2. In the **New Project** window, click **ASP.NET MVC 3 Web Application**.

3. In **Name**, enter `TestApp` and press **OK**.

4. In the **New ASP.NET MVC 3 Project** dialog, select **Internet Application** from the available templates, ensure **View Engine** is set to **Razor**, and then click **OK**.

5. When the new project opens, right-click the **TestApp** project in **Solution Explorer** and select the **Properties** option.

6. On the project's properties page, click on the **Web** tab on the left and ensure that the **Use Local IIS Web Server** option is selected.

Step 2 – Configure ASP.NET MVC Application for Claims-Based Authentication

In this step you will add configuration entries to the *Web.config* configuration file of your ASP.NET MVC web application to make it claims-aware.

To configure ASP.NET MVC application for claims-based authentication

1. Add the following configuration section definitions to the *Web.config* configuration file. These define configuration sections required by Windows Identity Foundation. Add the definitions immediately after the **<configuration>** opening element:

```
<configSections>
    <section name="system.identityModel"
type="System.IdentityModel.Configuration.SystemIdentityModelSection, System.IdentityModel,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=B77A5C561934E089" />
    <section name="system.identityModel.services"
type="System.IdentityModel.Services.Configuration.SystemIdentityModelServicesSection,
System.IdentityModel.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=B77A5C561934E089" />
</configSections>
```

2. Add a **<location>** element that enables access to the application's federation metadata:

```
<location path="FederationMetadata">
    <system.web>
        <authorization>
            <allow users="*" />
        </authorization>
    </system.web>
</location>
```

3. Add the following configuration entries within the **<system.web>** elements to deny users, disable native authentication, and enable WIF to manage authentication.

```
<authorization>
    <deny users="?" />
</authorization>
<authentication mode="None" />
```

4. Add the following Windows Identity Foundation related configuration entries and ensure that your ASP.NET application's URL and port number match the values in the **<audienceUris>** entry, **realm** attribute of the **<wsFederation>** element, and the **reply** attribute of the **<wsFederation>** element. Also ensure that the **issuer** value fits your Security Token Service (STS) URL.

```
<system.identityModel>
    <identityConfiguration>
        <audienceUris>
            <add value="http://localhost:28503/" />
        </audienceUris>
        <issuerNameRegistry type="System.IdentityModel.Tokens.ConfigurationBasedIssuerNameRegistry,
System.IdentityModel, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089">
            <trustedIssuers>
                <add thumbprint="1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ1234" name="YourSTSName" />
            </trustedIssuers>
        </issuerNameRegistry>
        <certificateValidation certificateValidationMode="None" />
    </identityConfiguration>
</system.identityModel>
<system.identityModel.services>
    <federationConfiguration>
        <cookieHandler requireSsl="false" />
        <wsFederation passiveRedirectEnabled="true"
issuer="http://localhost:13922/wsFederationSTS/Issue" realm="http://localhost:28503/"
reply="http://localhost:28503/" requireHttps="false" />
    </federationConfiguration>
</system.identityModel.services>
```

5. Add reference to the System.IdentityModel assembly.

6. Compile the solution to make sure there are errors.

Step 3 – Test Your Solution

In this step you will test your ASP.NET MVC web application configured for claims-based authentication. To perform basic test you will add simple code that displays claims in the token issued by the Security Token Service (STS).

To test your ASP.NET MVC application for claims-based authentication

1. In the **Solution Explorer**, expand the **Controllers** folder and open *HomeController.cs* file in the editor. Add the following code to the **Index** method:

```
public ActionResult Index()
{
    ViewBag.ClaimsIdentity = Thread.CurrentPrincipal.Identity;

    return View();
}
```

2. In the **Solution Explorer** expand **Views** and then **Home** folders and open *Index.cshtml* file in the editor. Delete its contents and add the following markup:

```
@{
    ViewBag.Title = "Home Page";
}

<h2>Welcome: @ViewBag.ClaimsIdentity.Name</h2>
<h3>Values from Identity</h3>
<table>
    <tr>
        <th>
            IsAuthenticated
        </th>
        <td>
            @ViewBag.ClaimsIdentity.IsAuthenticated
        </td>
    </tr>
    <tr>
        <th>
            Name
        </th>
        <td>
            @ViewBag.ClaimsIdentity.Name
        </td>
    </tr>
</table>
<h3>Claims from ClaimsIdentity</h3>
<table>
    <tr>
        <th>
            Claim Type
        </th>
        <th>
            Claim Value
        </th>
        <th>
            Value Type
        </th>
        <th>
            Subject Name
        </th>
        <th>
            Issuer Name
        </th>
    </tr>
        @foreach (System.Security.Claims.Claim claim in ViewBag.ClaimsIdentity.Claims ) {
    <tr>
        <td>
            @claim.Type
        </td>
        <td>
            @claim.Value
        </td>
        <td>
            @claim.ValueType
        </td>
        <td>
            @claim.Subject.Name
        </td>
        <td>
            @claim.Issuer
        </td>
    </tr>
}
</table>
```

3. Run the solution by pressing the **F5** key.

4. You should be presented with the page that displays the claims in the token that was issued to you by Security Token Service.

Related Items

- How To: Build Claims-Aware ASP.NET Web Forms Application Using WIF

How To: Build Claims-Aware ASP.NET Web Forms Application Using WIF

6/15/2017 • 3 min to read • Edit Online

Applies To

- Microsoft® Windows® Identity Foundation (WIF)
- ASP.NET® Web Forms

Summary

This How-To provides detailed step-by-step procedures for creating simple claims-aware ASP.NET Web Forms application. It also provides instructions for how to test the simple claims-aware ASP.NET Web Forms application for successful implementation of federated authentication. This How-To does not have detailed instructions for creating a Security Token Service (STS), and assumes you have already configured an STS.

Contents

Objectives

- Configure ASP.NET Web Forms application for claims-based authentication
- Test successful claims-aware ASP.NET Web Forms application

Summary of Steps

- Step 1 – Create Simple ASP.NET Web Forms Application
- Step 2 – Configure ASP.NET Web Forms Application for Federated Authentication
- Step 3 – Test Your Solution

Step 1 – Create a Simple ASP.NET Web Forms Application

In this step, you will create a new ASP.NET Web Forms application.

To create a simple ASP.NET application

1. Start Visual Studio and click **File**, **New**, and then **Project**.
2. In the **New Project** window, click **ASP.NET Web Forms Application**.
3. In **Name**, enter `TestApp` and press **OK**.

Step 2 – Configure ASP.NET Web Forms Application for Claims-Based Authentication

In this step you will add configuration entries to the *Web.config* configuration file of your ASP.NET Web Forms application to make it claims-aware.

To configure ASP.NET application for claims-based authentication

1. Add the following configuration section entries to the *Web.config* configuration file immediately after the **<configuration>** opening element:

```
<configSections>
  <section name="system.identityModel"
type="System.IdentityModel.Configuration.SystemIdentityModelSection, System.IdentityModel,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=B77A5C561934E089" />
  <section name="system.identityModel.services"
type="System.IdentityModel.Services.Configuration.SystemIdentityModelServicesSection,
System.IdentityModel.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=B77A5C561934E089" />
</configSections>
```

2. Add a **<location>** element that enables access to the application's federation metadata:

```
<location path="FederationMetadata">
  <system.web>
    <authorization>
      <allow users="*" />
    </authorization>
  </system.web>
</location>
```

3. Add the following configuration entries within the **<system.web>** elements to deny users, disable native authentication, and enable WIF to manage authentication.

```
<authorization>
  <deny users="?" />
</authorization>
<authentication mode="None" />
```

4. Add a **<system.webServer>** element that defines the modules for federated authentication. Note that the *PublicKeyToken* attribute must be the same as the *PublicKeyToken* attribute for the **<configSections>** entries added earlier:

```
<system.webServer>
  <modules>
    <add name="WSFederationAuthenticationModule"
type="System.IdentityModel.Services.WSFederationAuthenticationModule, System.IdentityModel.Services,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" preCondition="managedHandler" />
    <add name="SessionAuthenticationModule"
type="System.IdentityModel.Services.SessionAuthenticationModule, System.IdentityModel.Services,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" preCondition="managedHandler" />
  </modules>
</system.webServer>
```

5. Add the following Windows Identity Foundation related configuration entries and ensure that your ASP.NET application's URL and port number match the values in the **<audienceUris>** entry, **realm** attribute of the **<wsFederation>** element, and the **reply** attribute of the **<wsFederation>** element. Also ensure that the **issuer** value fits your Security Token Service (STS) URL.

```
<system.identityModel>
    <identityConfiguration>
        <audienceUris>
            <add value="http://localhost:28503/" />
        </audienceUris>
        <issuerNameRegistry type="System.IdentityModel.Tokens.ConfigurationBasedIssuerNameRegistry,
System.IdentityModel, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089">
            <trustedIssuers>
                <add thumbprint="1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ1234" name="YourSTSName" />
            </trustedIssuers>
        </issuerNameRegistry>
        <certificateValidation certificateValidationMode="None" />
    </identityConfiguration>
</system.identityModel>
<system.identityModel.services>
    <federationConfiguration>
        <cookieHandler requireSsl="false" />
        <wsFederation passiveRedirectEnabled="true"
issuer="http://localhost:13922/wsFederationSTS/Issue" realm="http://localhost:28503/"
reply="http://localhost:28503/" requireHttps="false" />
    </federationConfiguration>
</system.identityModel.services>
```

6. Add reference to the System.IdentityModel assembly.

7. Compile the solution to make sure there are no errors.

Step 3 – Test Your Solution

In this step you will test your ASP.NET Web Forms application configured for claims-based authentication. To perform a basic test, you will add code that displays claims in the token issued by the Security Token Service (STS).

To test your ASP.NET Web Form application for claims-based authentication

1. Open the **Default.aspx** file under the **TestApp** project and replace its existing markup with the following markup:

```
%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html>

<html>
<head id="Head1" runat="server">
    <title></title>
</head>
<body>
    <h1><asp:label ID="signedIn" runat="server" /></h1>
    <asp:label ID="claimType" runat="server" />
    <asp:label ID="claimValue" runat="server" />
    <asp:label ID="claimValueType" runat="server" />
    <asp:label ID="claimSubjectName" runat="server" />
    <asp:label ID="claimIssuer" runat="server" />
</body>
</html>
```

2. Save **Default.aspx**, and then open its code behind file named **Default.aspx.cs**.

NOTE

Default.aspx.cs may be hidden beneath **Default.aspx** in Solution Explorer. If **Default.aspx.cs** is not visible, expand **Default.aspx** by clicking on the triangle next to it.

3. Replace the existing code in the **Page_Load** method of **Default.aspx.cs** with the following code:

```
using System;
using System.IdentityModel;
using System.Security.Claims;
using System.Threading;
using System.Web.UI;

namespace TestApp
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            ClaimsPrincipal claimsPrincipal = Thread.CurrentPrincipal as ClaimsPrincipal;

            if (claimsPrincipal != null)
            {
                signedIn.Text = "You are signed in.";

                foreach (Claim claim in claimsPrincipal.Claims)
                {
                    claimType.Text = claim.Type;
                    claimValue.Text = claim.Value;
                    claimValueType.Text = claim.ValueType;
                    claimSubjectName.Text = claim.Subject.Name;
                    claimIssuer.Text = claim.Issuer;
                }
            }
            else
            {
                signedIn.Text = "You are not signed in.";
            }
        }
    }
}
```

4. Save **Default.aspx.cs**, and build the solution.

5. Run the solution by pressing the **F5** key.

6. You should be presented with the page that displays the claims in the token that was issued to you by the Security Token Service.

How To: Build Claims-Aware ASP.NET Application Using Forms-Based Authentication

5/23/2017 • 3 min to read • Edit Online

Applies To

- Microsoft® Windows® Identity Foundation (WIF)
- ASP.NET® Web Forms

Summary

This How-To provides detailed step-by-step procedures for creating a simple claims-aware ASP.NET Web Forms application that uses Forms authentication. It also provides instructions for how to test the application to verify that claims are presented when a user signs in with Forms authentication.

Contents

Objectives

- Configure an ASP.NET Web Forms application for claims using Forms authentication
- Test the ASP.NET Web Forms application to see if it is working properly

Overview

In .NET 4.5, WIF and its claims-based authorization have been included as an integral part of the Framework. Previously, if you wanted claims from an ASP.NET user, you were required to install WIF, and then cast interfaces to Principal objects such as `Thread.CurrentPrincipal` or `HttpContext.Current.User`. Now, claims are served automatically by these Principal objects.

Forms authentication has benefited from WIF's inclusion in .NET 4.5 because all users authenticated by Forms automatically have claims associated with them. You can begin using these claims immediately in an ASP.NET application that uses Forms authentication, as this How-To demonstrates.

Summary of Steps

- Step 1 – Create a Simple ASP.NET Web Forms Application
- Step 2 – Configure ASP.NET Web Forms Application for Claims Using Forms Authentication

- Step 3 – Test Your Solution

Step 1 – Create a Simple ASP.NET Web Forms Application

In this step, you will create a new ASP.NET Web Forms application.

To create a simple ASP.NET application

1. Start Visual Studio and click **File**, **New**, and then **Project**.

2. In the **New Project** window, click **ASP.NET Web Forms Application**.

3. In **Name**, enter `TestApp` and press **OK**.

Step 2 – Configure ASP.NET Web Forms Application for Claims Using Forms Authentication

In this step you will add a configuration entry to the *Web.config* configuration file and edit the *Default.aspx* file to display claims information for an account.

To configure ASP.NET application for claims using Forms authentication

1. In the *Default.aspx* file, replace the existing markup with the following:

```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
CodeBehind="Default.aspx.cs" Inherits="TestApp._Default" %>

<asp:Content runat="server" ID="BodyContent" ContentPlaceHolderID="MainContent">
    <p>
        This page displays the claims associated with a Forms authenticated user.
    </p>
    <h3>Your Claims</h3>
    <p>
        <asp:GridView ID="ClaimsGridView" runat="server" CellPadding="3">
            <AlternatingRowStyle BackColor="White" />
            <HeaderStyle BackColor="#7AC0DA" ForeColor="White" />
        </asp:GridView>
    </p>
</asp:Content>
```

   This step adds a GridView control to your *Default.aspx* page that will be populated with the claims retrieved from Forms authentication.

2. Save the *Default.aspx* file, then open its code-behind file named *Default.aspx.cs*. Replace the existing code with the following:

```
using System;
using System.Web.UI;
using System.Security.Claims;

namespace TestApp
{
    public partial class _Default : Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            ClaimsPrincipal claimsPrincipal = Page.User as ClaimsPrincipal;

            if (claimsPrincipal != null)
            {
                this.ClaimsGridView.DataSource = claimsPrincipal.Claims;
                this.ClaimsGridView.DataBind();
            }
        }
    }
}
```

The above code will display claims about an authenticated user, including users identified by Forms authentication.

Step 3 – Test Your Solution

In this step you will test your ASP.NET Web Forms application, and verify that claims are presented when a user signs in with Forms authentication.

To test your ASP.NET Web Forms application for claims using Forms authentication

1. Press **F5** to build and run the application. You should be presented with *Default.aspx*, which has **Register** and **Log in** links in the top right of the page. Click **Register**.
2. On the **Register** page, create a user account, and then click **Register**. Your account will be created using Forms authentication, and you will be automatically signed in.
3. After you have been redirected to the home page, you should see a table beneath the **Your Claims** heading that includes the **Issuer**, **OriginalIssuer**, **Type**, **Value**, and **ValueType** claims information about your account.

How To: Build Claims-Aware ASP.NET Application Using Windows Authentication

6/2/2017 • 3 min to read • Edit Online

Applies To

- Microsoft® Windows® Identity Foundation (WIF)
- ASP.NET® Web Forms

Summary

This How-To provides detailed step-by-step procedures for creating a simple claims-aware ASP.NET Web Forms application that uses Windows authentication. It also provides instructions for how to test the application to verify that claims are presented when a user signs in using Windows authentication.

Contents

Objectives

- Configure an ASP.NET Web Forms application for claims using Windows authentication
- Test the ASP.NET Web Forms application to see if it is working properly

Overview

In .NET 4.5, WIF and its claims-based authorization have been included as an integral part of the Framework. Previously, if you wanted claims from an ASP.NET user, you were required to install WIF, and then cast interfaces to Principal objects such as `Thread.CurrentPrincipal` or `HttpContext.Current.User`. Now, claims are served automatically by these Principal objects.

Windows authentication has benefited from WIF's inclusion in .NET 4.5 because all users authenticated by Windows credentials automatically have claims associated with them. You can begin using these claims immediately in an ASP.NET application that uses Windows authentication, as this How-To demonstrates.

Summary of Steps

- Step 1 – Create a Simple ASP.NET Web Forms Application
- Step 2 – Configure ASP.NET Web Forms Application for Claims Using Windows Authentication

- Step 3 – Test Your Solution

Step 1 – Create a Simple ASP.NET Web Forms Application

In this step, you will create a new ASP.NET Web Forms application.

To create a simple ASP.NET application

1. Start Visual Studio, then click **File**, **New**, and then **Project**.
2. In the **New Project** window, click **ASP.NET Web Forms Application**.
3. In **Name**, enter `TestApp` and press **OK**.
4. After the **TestApp** project has been created, click on it in **Solution Explorer**. The project's properties will appear in the **Properties** pane below **Solution Explorer**. Set the **Windows Authentication** property to **Enabled**.

   WARNING

   Windows authentication is disabled by default in new ASP.NET applications, so you must manually enable it.

Step 2 – Configure ASP.NET Web Forms Application for Claims Using Windows Authentication

In this step you will add a configuration entry to the *Web.config* configuration file and modify the *Default.aspx* file to display claims information for an account.

To configure ASP.NET application for claims using Windows authentication

1. In the **TestApp** project's *Default.aspx* file, replace the existing markup with the following:

```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
    CodeBehind="Default.aspx.cs" Inherits="TestApp._Default" %>

<asp:Content runat="server" ID="BodyContent" ContentPlaceHolderID="MainContent">
    <p>
        This page displays the claims associated with a Windows authenticated user.
    </p>
    <h3>Your Claims</h3>
    <p>
        <asp:GridView ID="ClaimsGridView" runat="server" CellPadding="3">
            <AlternatingRowStyle BackColor="White" />
            <HeaderStyle BackColor="#7AC0DA" ForeColor="White" />
        </asp:GridView>
    </p>
</asp:Content>
```

   This step adds a GridView control to your *Default.aspx* page that will be populated with the claims retrieved from Windows authentication.

2. Save the *Default.aspx* file, then open its code-behind file named *Default.aspx.cs*. Replace the existing code with the following:

```
using System;
using System.Web.UI;
using System.Security.Claims;

namespace TestApp
{
    public partial class _Default : Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            ClaimsPrincipal claimsPrincipal = Page.User as ClaimsPrincipal;
            this.ClaimsGridView.DataSource = claimsPrincipal.Claims;
            this.ClaimsGridView.DataBind();
        }
    }
}
```

The above code will display claims about an authenticated user.

3. To change the application's authentication type, modify the **<authentication>** block in the **<system.web>** section of the project's root *Web.config* file so that it only includes the following configuration entry:

```
<authentication mode="Windows" />
```

4. Finally, modify the **<authorization>** block in the **<system.web>** section of the same *Web.config* file to force authentication:

```
<authorization>
    <deny users="?" />
</authorization>
```

Step 3 – Test Your Solution

In this step you will test your ASP.NET Web Forms application, and verify that claims are presented when a user signs in with Windows authentication.

To test your ASP.NET Web Forms application for claims using Windows authentication

1. Press **F5** to build and run the application. You should be presented with *Default.aspx*, and your Windows account name (including domain name) should already appear as the authenticated user in the top right of the page. The page's content should include a table filled with claims retrieved from your Windows account.

How To: Debug Claims-Aware Applications And Services Using WIF Tracing

5/3/2017 • 3 min to read • Edit Online

Applies To

- Microsoft® Windows® Identity Foundation (WIF)
- Service Trace Viewer Tool (SvcTraceViewer.exe)
- Troubleshooting and Debugging WIF Applications

Summary

This How-To describes required steps for how to configure WIF tracing, collect trace logs, and how to analyze the trace logs using Trace Viewer tool. It provides general mapping for trace entries to actions needed to troubleshoot issues related to WIF.

Contents

Objectives

- Configure WIF tracing.
- View trace logs in the Trace Viewer tool.
- Identify WIF related issues in the trace logs.
- Apply corrective actions to WIF related issues found in the trace logs.

Summary of Steps

- Step 1 – Configure WIF Tracing Using Web.config Configuration File
- Step 2 – Analyze WIF Trace Files Using Trace Viewer Tool
- Step 3 – Identify Solutions to Fix WIF Related Issues

Step 1 – Configure WIF Tracing Using Web.config Configuration File

In this step, you will add changes to configuration sections in the *Web.config* file that enable WIF to trace its events and store them in a trace log.

To configure WIF tracing using Web.config configuration file

1. Open the root **Web.config** or **App.config** configuration file in the Visual Studio editor by double clicking on it in **Solution Explorer**. If your solution does not have **Web.config** or **App.config** configuration file, add it by right clicking on the solution in the **Solution Explorer** and clicking **Add**, then clicking **New Item...**. On the **New Item** dialog, Select **Application Configuration File** for **App.config** or **Web Configuration File** for **Web.config** from the list and click **OK**.

2. Add the configuration entries similar to the following to the configuration file inside **<configuration>** node at the end of the configuration file:

```
<system.diagnostics>
    <sources>
        <source name="System.IdentityModel" switchValue="Verbose">
            <listeners>
                <add name="xml" type="System.Diagnostics.XmlWriterTraceListener"
initializeData="WIFTrace.e2e"/>
            </listeners>
        </source>
    </sources>
    <trace autoflush="true"/>
</system.diagnostics>
```

3. The above configuration instructs WIF to generate verbose trace events and log them into *WIFTrace.e2e* file. For a complete list of values for the **switchValue** switch, refer to the Trace Level table found in the following topic: Configuring Tracing.

Step 2 – Analyze WIF Trace Files Using Trace Viewer Tool

In this step, you will use the Trace Viewer Tool (SvcTraceViewer.exe) to analyze WIF trace logs.

To analyze WIF trace logs using Trace Viewer tool (SvcTraceViewer.exe)

1. Trace Viewer tool (SvcTraceViewer.exe) ships as part of the Windows SDK. If you haven't already installed the Windows SDK, you can download it here: Windows SDK.

2. Run the Trace Viewer tool (SvcTraceViewer.exe). It is typically available in the **Bin** folder of the installation path.

3. Open the WIF trace log file, for example, WIFTrace.e2e by selecting **File**, **Open...** option in the menu or using the **Ctrl+O** shortcut. The trace log file opens in the Trace Viewer tool.

4. Review entries in the **Activity** tab. Each entry should contain an activity number, the number of traces that were logged, duration of the activity and its start and end timestamps.

5. Click on the **Activity** tab. You should see detailed trace entries in the main area of the tool. Use the **Level** dropdown list on the menu to filter specific level of traces, for example: **All**, **Warning**, **Errors**, **Information**, etc.

6. Click on specific trace entries to review the details in the lower area of the tool. The details can be viewed using **Formatted** and **XML** view by choosing corresponding tabs.

Step 3 – Identify Solutions to Fix WIF Related Issues

In this step, you can identify solutions for WIF-related issues identified by using the WIF trace log and Trace Viewer tool. It outlines general mapping of WIF related exceptions to potential solutions or required actions to troubleshoot the issue.

To identify solutions to fix WIF related issues

1. Review the following table of WIF exceptions and the required actions to correct the issues.

ERROR ID	ERROR MESSAGE	ACTION NEEDED TO FIX THE ERROR
ID4175	The issuer of the security token was not recognized by the IssuerNameRegistry. To accept security tokens from this issuer, configure the IssuerNameRegistry to return a valid name for this issuer.	This error can be caused by copying a thumbprint from the MMC snap-in and pasting it into the *Web.config* file. Specifically, you can get an extra non-printable character in the text string when copying from the certificate properties window. This extra character causes the thumbprint match to fail.The procedure for correctly copying the thumbprint can be found here: http://msdn.microsoft.com/library/ff359102.aspx

Related Items

- Using Service Trace Viewer for Viewing Correlated Traces and Troubleshooting

How To: Display Signed In Status Using WIF

5/16/2017 • 4 min to read • Edit Online

Applies To

- Microsoft® Windows® Identity Foundation (WIF) 4.5
- ASP.NET® Web Forms

Summary

This topic describes how to display the sign in status in a WIF-enabled ASP.NET application. WIF provides the mechanism for making your application claims-aware, and managing authentication and authorization for application resources.

Contents

Overview

This topic demonstrates how to create a simple claims-aware application using WIF and how to easily display whether a user is signed in or not. The following steps use the Local Development STS that is included with the Identity and Access Visual Studio Extension. The Local Development STS is intended for a testing and development environment to provide a simple method of integrating claims into your application. It should never be used in a production environment, as it does not perform real authentication and credentials are not required. However, the imperative code in the following steps is the same for a production-ready application using real authentication.

Summary of Steps

- Step 1 – Install the Identity and Access Extension
- Step 2 – Create a Relying Party ASP.NET Application
- Step 3 – Enable Local Development STS to Authenticate Users
- Step 4 – Modify Your ASP.NET Application to Display Sign In Status
- Step 5 – Test the Integration Between WIF and Your ASP.NET Application

Step 1 – Install the Identity and Access Extension

This step describes how to configure the Identity and Access extension to Visual Studio 2012. This extension automates the process of configuring your application to communicate with STS endpoints.

To install the Identity and Access extension

1. Start Visual Studio in elevated mode as administrator.
2. In Visual Studio, click **Tools** and click **Extension Manager**. The **Extension Manager** window appears.
3. In **Extension Manager**, click **Online Extensions** from the left menu, then select **Visual Studio Gallery**.
4. In the top right corner of **Extension Manager**, search for *Identity and Access*.
5. The **Identity and Access** item will appear in the search results. Click it, and then click **Download**.
6. The **Download and Install** dialog appears. If you agree with the license terms, click **Install**.
7. When the **Identity and Access** extension has finished installing, restart Visual Studio in administrator mode.

Step 2 – Create a Relying Party ASP.NET Application

This step describes how to create a relying party ASP.NET Web Forms application that will integrate with WIF.

To create a simple ASP.NET application

1. Start Visual Studio and click **File**, **New**, and then **Project**.
2. In the **New Project** window, click **ASP.NET Web Forms Application**.
3. In **Name**, enter `TestApp` and press **OK**.

Step 3 – Enable Local Development STS to Authenticate Users

This step describes how to enable Local Development STS in your application. Local Development STS is enabled by using the Identity and Access extension for Visual Studio.

To enable Local Development STS in your ASP.NET application

1. In Visual Studio, right-click the **TestApp** project under **Solution Explorer**, then select **Identity and Access**.
2. The **Identity and Access** window appears. Under **Providers**, select **Test your application with the Local Development STS**, then click **Apply**.

Step 4 – Modify Your ASP.NET Application to Display Sign In Status

This step describes how to modify your ASP.NET application to dynamically display whether the current user is signed in. Once your STS provider has been configured, WIF handles the incoming claims. Now you need to configure your application’s code to display the result of the authentication.

To display sign in status

1. In Visual Studio, open the **Default.aspx** file under the **TestApp** project.
2. Replace the existing markup in the **Default.aspx** file with the following markup:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html>

<html>
<head runat="server">
    <title>Logged In Status</title>
</head>
<body>
    <asp:label ID="myLabel" runat="server" />
</body>
</html>
```

3. Save **Default.aspx**, and then open its code behind file named **Default.aspx.cs**.

> **NOTE**
>
> **Default.aspx.cs** may be hidden beneath **Default.aspx** in Solution Explorer. If **Default.aspx.cs** is not visible, expand **Default.aspx** by clicking on the triangle next to it.

4. Replace the existing code in **Default.aspx.cs** with the following code:

```
using System;
using System.Web.UI;
using System.Security.Claims;

namespace TestApp
{
    protected void Page_Load(object sender, EventArgs e)
    {
        ClaimsPrincipal claimsPrincipal = Thread.CurrentPrincipal as ClaimsPrincipal;

        if (claimsPrincipal != null)
        {
            myLabel.Text = "You are signed in.";
        }
        else
        {
            myLabel.Text = "You are not signed in.";
        }
    }
}
```

5. Save **Default.aspx.cs**, and build the application.

Step 5 – Test the Integration Between WIF and Your ASP.NET Application

This step describes how you can test the integration between WIF and your ASP.NET application.

To test the integration between WIF and ASP.NET

1. In Visual Studio, press **F5** to start debugging your application. If no errors are found, a new browser window will open.

2. You may notice that the browser silently redirects your request to the STS, and then opens the Default.aspx page. If WIF is properly configured, you should see the site display the following text: **"You are signed in"**.

How To: Enable WIF Tracing

6/2/2017 • 3 min to read • Edit Online

Applies To

- Microsoft® Windows® Identity Foundation (WIF)
- ASP.NET® Web Forms

Summary

This How-To provides detailed step-by-step procedures for enabling WIF tracing in an ASP.NET application. It also provides instructions testing the application to verify that the trace listener and log are working correctly. This How-To does not have detailed instructions for creating a Security Token Service (STS), and instead uses the Development STS that comes with the Identity and Access tool. The Development STS does not perform real authentication and is intended for testing purposes only. You will need to install the Identity and Access tool to complete this How-To. It can be downloaded from the following location: Identity and Access Tool

IMPORTANT

Enabling WIF tracing for passive applications, that is, applications that use the WS-Federation protocol, can potentially expose the application to denial of service (DoS) attacks or to information disclosure to a malicious party. This includes both passive RPs and passive STSes. For this reason, we recommend that you not enable WIF tracing for passive RPs or STSes in a production environment.

Contents

Objectives

- Create a simple ASP.NET application that uses WIF and the Development STS from the Identity and Access Tool
- Enable tracing and verify that it is working

Overview

Tracing enables you to debug and troubleshoot many types of issues with WIF, including tokens, cookies, claims, protocol messages, and more. WIF tracing is similar to WCF tracing; for example, you can choose the verbosity of traces to display everything from critical messages to all messages. WIF traces can be generated in **.xml** files or in **.svclog** files that are viewable by using the Service Trace Viewer Tool. This tool is located in the **bin** directory of the Windows SDK install path on your computer, for example: **C:\Program Files\Microsoft SDKs\Windows\v7.1\Bin\SvcTraceViewer.exe**.

Summary of Steps

- Step 1 – Create a Simple ASP.NET Web Forms Application and Enable Tracing
- Step 2 – Test Your Solution

Step 1 – Create a Simple ASP.NET Web Forms Application and Enable Tracing

In this step, you will create a new ASP.NET Web Forms application and modify the *Web.config* file to enable tracing.

To create a simple ASP.NET application

1. Start Visual Studio and click **File**, **New**, and then **Project**.
2. In the **New Project** window, click **ASP.NET Web Forms Application**.
3. In **Name**, enter `TestApp` and press **OK**.
4. Right-click the **TestApp** project under **Solution Explorer**, then select **Identity and Access**.
5. The **Identity and Access** window appears. Under **Providers**, select **Test your application with the Local Development STS**, then click **Apply**.
6. Create a new folder in named **logs** in the root of the **C:** drive, like shown: **C:\logs**
7. Add the following **<system.diagnostics>** element to the *Web.config* configuration file immediately following the closing **</configSections>** element, like shown:

```
<configuration>
    <configSections>
    …
    </configSections>
    <system.diagnostics>
        <sources>
            <source name="System.IdentityModel" switchValue="Verbose">
                <listeners>
                    <add name="xml" type="System.Diagnostics.XmlWriterTraceListener"
initializeData="C:\logs\WIF.xml" />
                </listeners>
            </source>
        </sources>
        <trace autoflush="true" />
    </system.diagnostics>
```

NOTE

The directory location specified in the **initializeData** attribute must exist before logging can begin. If the location does not exist, no logs will be created.

The configuration settings above will enable **Verbose** tracing for WIF and save the resulting log to the **C:logsWIF.xml** file.

Step 2 – Test Your Solution

In this step, you will test your WIF-enabled ASP.NET application to verify that logs are being recorded.

To test your WIF-enabled ASP.NET application for successful tracing

1. Run the solution by pressing the **F5** key. You should be presented with the default ASP.NET Home Page and automatically authenticated with the username *Terry*, which is the default user that is returned by the

Development STS.

2. Close the browser window and then navigate to the **C:\logs** folder. Open the **C:\logs\WIF.xml** file using a text editor.

3. Inspect the **WIF.xml** file and verify that it contains entries starting with **<E2ETraceEvent>**. These traces will contain **<TraceRecord>** elements with descriptions for the traced activity, such as **Validating SecurityToken**.

How To: Enable Token Replay Detection

6/2/2017 • 2 min to read • Edit Online

Applies To

- Microsoft® Windows® Identity Foundation (WIF)
- ASP.NET® Web Forms

Summary

This How-To provides detailed step-by-step procedures for enabling token replay detection in an ASP.NET application that uses WIF. It also provides instructions for how to test the application to verify that token replay detection is enabled. This How-To does not have detailed instructions for creating a Security Token Service (STS), and instead uses the Development STS that comes with the Identity and Access tool. The Development STS does not perform real authentication and is intended for testing purposes only. You will need to install the Identity and Access tool to complete this How-To. It can be downloaded from the following location: Identity and Access Tool

Contents

Objectives

- Create a simple ASP.NET application that uses WIF and the Development STS from the Identity and Access Tool
- Enable token replay detection and verify that it is working

Overview

A replay attack occurs when a client attempts to authenticate to a relying party with an STS token that the client has already used. To help prevent this attack, WIF contains a replay detection cache of previously used STS tokens. When enabled, replay detection checks the token of the incoming request and verifies whether or not the token has been previously used. If the token has been used already, the request is refused and a SecurityTokenReplayDetectedException exception is thrown.

The following steps demonstrate the configuration changes required to enable replay detection.

Summary of Steps

- Step 1 – Create a Simple ASP.NET Web Forms Application and Enable Replay Detection
- Step 2 – Test Your Solution

Step 1 – Create a Simple ASP.NET Web Forms Application and Enable Replay Detection

In this step, you will create a new ASP.NET Web Forms application and modify the *Web.config* file to enable replay detection.

To create a simple ASP.NET application

1. Start Visual Studio and click **File**, **New**, and then **Project**.
2. In the **New Project** window, click **ASP.NET Web Forms Application**.
3. In **Name**, enter `TestApp` and press **OK**.
4. Right-click the **TestApp** project under **Solution Explorer**, then select **Identity and Access**.
5. The **Identity and Access** window appears. Under **Providers**, select **Test your application with the Local Development STS**, then click **Apply**.
6. Add the following **<tokenReplayDetection>** element to the *Web.config* configuration file immediately following the **<system.identityModel>** and **<identityConfiguration>** elements, like shown:

```
<system.identityModel>
    <identityConfiguration>
        <tokenReplayDetection enabled="true"/>
```

Step 2 – Test Your Solution

In this step, you will test your WIF-enabled ASP.NET application to verify that replay detection has been enabled.

To test your WIF-enabled ASP.NET application for replay detection

1. Run the solution by pressing the **F5** key. You should be presented with the default ASP.NET Home Page and automatically authenticated with the username *Terry*, which is the default user that is returned by the Development STS.
2. Press the browser's **Back** button. You should be presented with a **Server Error in '/' Application** page with the following description: *ID1062: Replay has been detected for: Token:* '*System.IdentityModel.Tokens.SamlSecurityToken*', followed by an *AssertionId* and an *Issuer*.

   You are seeing this error page because an exception of type SecurityTokenReplayDetectedException was thrown when the token replay was detected. This error occurs because you are attempting to re-send the initial POST request when the token was first presented. The **Back** button will not cause this behavior on subsequent requests to the server.

How To: Enable WIF for a WCF Web Service Application

6/2/2017 • 6 min to read • Edit Online

Applies To

- Microsoft® Windows® Identity Foundation (WIF)
- Microsoft® Windows® Communication Foundation (WCF)

Summary

This How-To provides detailed step-by-step procedures for enabling WIF in a WCF web service. It also provides instructions for how to test the application to verify that the web service is correctly presenting claims when the application is run. This How-To does not have detailed instructions for creating a Security Token Service (STS), and instead uses the Development STS that comes with the Identity and Access tool. The Development STS does not perform real authentication and is intended for testing purposes only. You will need to install the Identity and Access tool to complete this How-To. It can be downloaded from the following location: Identity and Access Tool

Contents

Objectives

- Create a WCF service that requires issued tokens
- Create a WCF client that requests a token from an STS and passes it to the WCF service

Overview

This How-To is intended to demonstrate how a developer can use federated authentication when developing WCF services. Some of the benefits of using federation in WCF services include:

1. Factoring authentication logic out of WCF service code
2. Re-using existing identity management solutions
3. Interoperability with other identity solutions
4. Flexibility and resilience to future changes

WIF and the associated Identity and Access tool make it easier to develop and test a WCF service using federated authentication, as the following steps demonstrate.

Summary of Steps

- Step 1 – Create a Simple WCF Service
- Step 2 – Create a Client Application for the WCF Service
- Step 3 – Test Your Solution

Step 1 – Create a Simple WCF Service

In this step, you will create a new WCF service that uses the Development STS that is included with the Identity and Access tool.

To create a simple WCF service

1. Start Visual Studio in elevated mode as administrator.
2. In Visual Studio, click **File**, click **New**, and then click **Project**.
3. In the **New Project** window, click **WCF Service Application**.
4. In **Name**, enter `TestService` and press **OK**.
5. Right-click the **TestService** project under **Solution Explorer**, then select **Identity and Access**.
6. The **Identity and Access** window appears. Under **Providers**, select **Test your application with the Local Development STS**, then click **Apply**. The Identity and Access Tool configures the service to use WIF and to outsource authentication to the local development STS (**LocalSTS**) by adding configuration elements to the *Web.config* file.
7. In the *Service1.svc.cs* file, add a `using` directive for the **System.Security.Claims** namespace and replace the existing code with the following, then save the file:

```
public class Service1 : IService1
{
    public string ComputeResponse(string input)
    {
        // Get the caller's identity from ClaimsPrincipal.Current
        ClaimsPrincipal claimsPrincipal = OperationContext.Current.ClaimsPrincipal;

        // Start generating the output
        StringBuilder builder = new StringBuilder();
        builder.AppendLine("Computed by ClaimsAwareWebService");
        builder.AppendLine("Input received from client:" + input);

        if (claimsPrincipal != null)
        {
            // Display the claims from the caller. Do not use this code in a production application.
            ClaimsIdentity identity = claimsPrincipal.Identity as ClaimsIdentity;
            builder.AppendLine("Client Name:" + identity.Name);
            builder.AppendLine("IsAuthenticated:" + identity.IsAuthenticated);
            builder.AppendLine("The service received the following issued claims of the client:");

            // Iterate over the caller’s claims and append to the output
            foreach (Claim claim in claimsPrincipal.Claims)
            {
                builder.AppendLine("ClaimType :" + claim.Type + "   ClaimValue:" + claim.Value);
            }
        }

        return builder.ToString();
    }
}
```

The `ComputeResponse` method displays the properties of various claims that are issued by **LocalSTS**.

8. In the *IService1.cs* file, replace the existing code with the following, then save the file:

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    string ComputeResponse(string input);
}
```

9. Build the project.

10. Press **Ctrl-F5** to run the service without starting the debugger. A Web page should open for the service and you can verify that **LocalSTS** is running by looking in the notification area (system tray).

> **IMPORTANT**
>
> Both **TestService** and **LocalSTS** must be running when you add the service reference to the client application in the next step.

Step 2 – Create a Client Application for the WCF Service

In this step, you will create a console application that uses the Development STS to authenticate with the WCF service you created in the previous step.

To create a client application

1. In Visual Studio, right-click on the solution, click **Add**, and then click **New Project**.

2. In the **Add New Project** window, select **Console Application** from the **Visual C#** templates list, enter `Client`, and then press **OK**. The new project will be created in your solution folder.

3. Right-click on **References** under the **Client** project, and then click **Add Service Reference**.

4. In the **Add Service Reference** window, click the drop-down arrow on the **Discover** button and click **Services in Solution**. The **Address** will automatically populate with the WCF service you created earlier, and the **Namespace** will be set to **ServiceReference1**. Click **OK**.

> **IMPORTANT**
>
> Both **TestService** and **LocalSTS** must be running when you add the service reference to the client.

5. Visual Studio will generate proxy classes for the WCF service, and add all of the necessary reference information. It will also add elements to the *App.config* file to configure the client to get a token from the STS to authenticate with the service. When this process is finished, the **Program.cs** file will open. Add a `using` directive for **System.ServiceModel** and another for **Client.ServiceReference1**, replace the **Main** method with the following code, then save the file:

```
static void Main(string[] args)
{
    // Create the client for the service
    Service1Client client = new Service1Client();
    Console.WriteLine("-------------WCF Client Application--------------\n");

    while (!ShouldQuitApplication())
    {
        try
        {
            Console.WriteLine(client.ComputeResponse("Hello World"));
        }
        catch (CommunicationException e)
        {
            Console.WriteLine(e.Message);
            Console.WriteLine(e.StackTrace);
            Exception ex = e.InnerException;
            while (ex != null)
            {
                Console.WriteLine("===========================");
                Console.WriteLine(ex.Message);
                Console.WriteLine(ex.StackTrace);
                ex = ex.InnerException;
            }
        }
        catch (TimeoutException)
        {
            Console.WriteLine("Timed out...");
        }
        catch (Exception e)
        {
            Console.WriteLine("An unexpected exception occurred.");
            Console.WriteLine(e.StackTrace);
        }
    }

    client.Close();

    if (client != null)
    {
        client.Abort();
    }
}

static bool ShouldQuitApplication()
{
    Console.WriteLine("----------------------------------------------------------------------");
    Console.WriteLine("Press Enter key to invoke service, any other key to quit application:");
    Console.WriteLine("-----------------------------------------------------------------------");

    ConsoleKeyInfo keyInfo = Console.ReadKey();
    if (keyInfo.Key == ConsoleKey.Enter)
        return false;
    return true;
}
```

6. Open the *App.config* file and add the following XML as the first child element under the `<system.serviceModel>` element, then save the file:

```
<behaviors>
   <endpointBehaviors>
     <behavior>
       <clientCredentials>
         <serviceCertificate>
           <authentication certificateValidationMode="None"/>
         </serviceCertificate>
       </clientCredentials>
     </behavior>
   </endpointBehaviors>
 </behaviors>
```

This disables certificate validation.

7. Right-click the **TestService** solution and click on **Set StartUp Projects**. The **Startup Project** property page opens. In the **Startup Project** property page, select **Multiple startup projects** then click in the **Action** field for each project and select **Start** from the drop-down menu. Click **OK** to save the settings.

8. Build the solution.

Step 3 – Test Your Solution

In this step you will test your WIF-enabled WCF application and verify that claims are presented.

To test your WIF-enabled WCF application for claims

1. Press **F5** to build and run the application. You should be presented with a console window, and the following text: **Press Enter key to invoke service, any other key to quit application:**

2. Press **Enter**, and the following claims information should appear in the console:

```
Computed by Service1
Input received from client: Hello World
Client Name: Terry
IsAuthenticated: True
The service received the following issued claims of the client:
ClaimType :http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name    ClaimValue:Terry
ClaimType :http://schema.xmlsoap.org/ws/2005/05/identity/claims/surname    ClaimValue:Adams
ClaimType :http://schemas.microsoft.com/ws/2008/06/identity/claims/role    ClaimValue:developer
ClaimType :http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress
ClaimValue:terry@contoso.com
```

IMPORTANT

Both **TestService** and **LocalSTS** must be running before you press **Enter**. A Web page should open for the service and you can verify that **LocalSTS** is running by looking in the notification area (system tray).

3. If these claims appear in the console, you have successfully authenticated with the STS to display claims from the WCF service.

How To: Transform Incoming Claims

6/2/2017 • 3 min to read • Edit Online

Applies To

- Microsoft® Windows® Identity Foundation (WIF)
- ASP.NET® Web Forms

Summary

This How-To provides detailed step-by-step procedures for creating a simple claims-aware ASP.NET Web Forms application and transforming incoming claims. It also provides instructions for how to test the application to verify that transformed claims are presented when the application is run.

Contents

Objectives

- Configure an ASP.NET Web Forms application for claims-based authentication
- Transform incoming claims by adding an Administrator role claim
- Test the ASP.NET Web Forms application to see if it is working properly

Overview

WIF exposes a class named ClaimsAuthenticationManager that enables users to modify claims before they are presented to a relying party (RP) application. The ClaimsAuthenticationManager is useful for separation of concerns between authentication and the underlying application code. The example below demonstrates how to add a role to the claims in the incoming ClaimsPrincipal that may be required by the RP.

Summary of Steps

- Step 1 – Create a Simple ASP.NET Web Forms Application
- Step 2 – Implement Claims Transformation Using a Custom ClaimsAuthenticationManager
- Step 3 – Test Your Solution

Step 1 – Create a Simple ASP.NET Web Forms Application

In this step, you will create a new ASP.NET Web Forms application.

To create a simple ASP.NET application

1. Start Visual Studio in elevated mode as administrator.
2. In Visual Studio, click **File**, click **New**, and then click **Project**.
3. In the **New Project** window, click **ASP.NET Web Forms Application**.
4. In **Name**, enter `TestApp` and press **OK**.
5. Right-click the **TestApp** project under **Solution Explorer**, then select **Identity and Access**.
6. The **Identity and Access** window appears. Under **Providers**, select **Test your application with the Local Development STS**, then click **Apply**.
7. In the *Default.aspx* file, replace the existing markup with the following, then save the file:

```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.Master" AutoEventWireup="true"
    CodeBehind="Default.aspx.cs" Inherits="TestApp._Default" %>

<asp:Content runat="server" ID="BodyContent" ContentPlaceHolderID="MainContent">
      <h3>Your Claims</h3>
    <p>
        <asp:GridView ID="ClaimsGridView" runat="server" CellPadding="3">
            <AlternatingRowStyle BackColor="White" />
            <HeaderStyle BackColor="#7AC0DA" ForeColor="White" />
        </asp:GridView>
    </p>
</asp:Content>
```

8. Open the code-behind file named *Default.aspx.cs*. Replace the existing code with the following, then save the file:

```
using System;
using System.Web.UI;
using System.Security.Claims;

namespace TestApp
{
    public partial class _Default : Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            ClaimsPrincipal claimsPrincipal = Page.User as ClaimsPrincipal;
            this.ClaimsGridView.DataSource = claimsPrincipal.Claims;
            this.ClaimsGridView.DataBind();
        }
    }
}
```

Step 2 – Implement Claims Transformation Using a Custom ClaimsAuthenticationManager

In this step you will override default functionality in the ClaimsAuthenticationManager class to add an Administrator role to the incoming Principal.

To implement claims transformation using a custom ClaimsAuthenticationManager

1. In Visual Studio, right-click the on the solution, click **Add**, and then click **New Project**.
2. In the **Add New Project** window, select **Class Library** from the **Visual C#** templates list, enter

`ClaimsTransformation`, and then press **OK**. The new project will be created in your solution folder.

3. Right-click on **References** under the **ClaimsTransformation** project, and then click **Add Reference**.
4. In the **Reference Manager** window, select **System.IdentityModel**, and then click **OK**.
5. Open **Class1.cs**, or if it doesn't exist, right-click **ClaimsTransformation**, click **Add**, then click **Class...**
6. Add the following using directives to the code file:

```
using System.Security.Claims;
using System.Security.Principal;
```

7. Add the following class and method in the code file.

WARNING

The following code is for demonstration purposes only; make sure that you verify your intended permissions in production code.

```
public class ClaimsTransformationModule : ClaimsAuthenticationManager
{
    public override ClaimsPrincipal Authenticate(string resourceName, ClaimsPrincipal
incomingPrincipal)
    {
        if (incomingPrincipal != null && incomingPrincipal.Identity.IsAuthenticated == true)
        {
           ((ClaimsIdentity)incomingPrincipal.Identity).AddClaim(new Claim(ClaimTypes.Role, "Admin"));
        }

        return incomingPrincipal;
    }
}
```

8. Save the file and build the **ClaimsTransformation** project.
9. In your **TestApp** ASP.NET project, right-click on References, and then click **Add Reference**.
10. In the **Reference Manager** window, select **Solution** from the left menu, select **ClaimsTransformation** from the populated options, and then click **OK**.
11. In the root **Web.config** file, navigate to the **<system.identityModel>** entry. Within the **<identityConfiguration>** elements, add the following line and save the file:

```
<claimsAuthenticationManager type="ClaimsTransformation.ClaimsTransformationModule,
ClaimsTransformation" />
```

Step 3 – Test Your Solution

In this step you will test your ASP.NET Web Forms application, and verify that claims are presented when a user signs in with Forms authentication.

To test your ASP.NET Web Forms application for claims using Forms authentication

1. Press **F5** to build and run the application. You should be presented with *Default.aspx*.
2. On the *Default.aspx* page, you should see a table beneath the **Your Claims** heading that includes the **Issuer**, **OriginalIssuer**, **Type**, **Value**, and **ValueType** claims information about your account. The last row should

be presented in the following way:

LOCAL AUTHORITY	LOCAL AUTHORITY	http://schemas.microsoft.com/ws/2008/06/identity/claims/role	Admin	http://www.w3.org/2001/XMLSchema#string

WIF Guidelines

5/3/2017 • 1 min to read • Edit Online

- Guidelines for Migrating an Application Built Using WIF 3.5 to WIF 4.5
- Namespace Mapping between WIF 3.5 and WIF 4.5

See Also

Windows Identity Foundation

Guidelines for Migrating an Application Built Using WIF 3.5 to WIF 4.5

6/2/2017 • 10 min to read • Edit Online

Applies To

- Microsoft® Windows® Identity Foundation (WIF) 3.5 and 4.5.

Overview

Windows Identity Foundation (WIF) was originally released in the .NET 3.5 SP1 timeframe. That version of WIF is referred to as WIF 3.5. It was released as a separate runtime and SDK, which meant that every computer on which a WIF-enabled application ran had to have the WIF runtime installed and developers had to download and install the WIF SDK to get the Visual Studio templates and tooling that enabled development of WIF-enabled applications. Beginning with .NET 4.5, WIF has been fully integrated into the .NET Framework. A separate runtime is no longer needed and the WIF tooling can be installed in Visual Studio 2012 by using the Visual Studio Extensions Manager. This version of WIF is referred to as WIF 4.5.

The integration of WIF into .NET necessitated several changes in the WIF API surface. WIF 4.5 includes new namespaces, some changes to configuration elements, and new tooling for Visual Studio. This topic provides guidance that you can use to help you migrate applications built using WIF 3.5 to WIF 4.5. There may be scenarios in which you need to run legacy applications built using WIF 3.5 on computers that are running Windows 8 or Windows Server 2012. This topic also provides guidance about how to enable WIF 3.5 for these operating systems.

Changes Required for WIF 4.5

This section describes the changes that are required to migrate a WIF 3.5 application to WIF 4.5.

Assembly and Namespace Changes

In WIF 3.5, all of the WIF classes were contained in the `Microsoft.IdentityModel` assembly (microsoft.identitymicrosoft.identitymodel.dll). In WIF 4.5, the WIF classes have been split across the following assemblies: `mscorlib` (mscorlib.dll), `System.IdentityModel` (System.IdentityModel.dll), `System.IdentityModel.Services` (System.IdentityModel.Services.dll), and `System.ServiceModel` (System.ServiceModel.dll).

The WIF 3.5 classes were all contained in one of the `Microsoft.IdentityModel` namespaces; for example, `Microsoft.IdentityModel`, `Microsoft.IdentityModel.Tokens`, `Microsoft.IdentityModel.Web`, and so on. In WIF 4.5, the WIF classes are now spread across the System.IdentityModel namespaces, the System.Security.Claims namespace, and the System.ServiceModel.Security namespace. In addition to this reorganization, some WIF 3.5 classes have been dropped in WIF 4.5.

The following table shows some of the more important WIF 4.5 namespaces and the kind of classes they contain. For more detailed information about how namespaces map between WIF 3.5 and WIF 4.5 and about namespaces and classes that have been dropped in WIF 4.5, see Namespace Mapping between WIF 3.5 and WIF 4.5.

WIF 4.5 NAMESPACE	DESCRIPTION

WIF 4.5 NAMESPACE	DESCRIPTION
System.IdentityModel	Contains classes that represent cookie transforms, security token services, and specialized XML dictionary readers. Contains classes from the following WIF 3.5 namespaces: `Microsoft.IdentityModel` , `Microsoft.IdentityModel.SecurityTokenService` , and `Microsoft.IdentityModel.Threading` .
System.Security.Claims	Contains classes that represent claims, claims-based identities, claims based principals, and other claims based identity model artifacts. Contains classes from the `Microsoft.IdentityModel.Claims` namespace.
System.IdentityModel.Tokens	Contains classes that represent security tokens, security token handlers, and other security token artifacts. Contains classes from the following WIF 3.5 namespaces: `Microsoft.IdentityModel.Tokens` , `Microsoft.IdentityModel.Tokens.Saml11` , and `Microsoft.IdentityModel.Tokens.Saml2` .
System.IdentityModel.Services	Contains classes that are used in passive (WS-Federation) scenarios. Contains classes from the `Microsoft.IdentityModel.Web` namespace.
System.ServiceModel.Security	Classes that represent WCF contracts, channels, service hosts and other artifacts that are used in active (WS-Trust) scenarios are now in this namespace. In WIF 3.5 , these classes were in the `Microsoft.IdentityModel.Protocols.WSTrust` namespace.

IMPORTANT

The following `System.IdentityModel` namespaces contain classes that implement the WCF claims-based identity model: System.IdentityModel.Claims, System.IdentityModel.Policy, and System.IdentityModel.Selectors. The WCF claims-based identity model is superseded by WIF. You should not use classes in these three namespaces when building solutions based on WIF.

Changes Due to .NET Integration

WIF is now integrated into the .NET Framework. Most .NET identity and principal classes now derive from System.Security.Claims.ClaimsIdentity and System.Security.Claims.ClaimsPrincipal. The results in the following changes in WIF 4.5:

- WIF classes that represent claims, identities, and principals now exist in the System.Security.Claims namespace.

  IMPORTANT

  The System.IdentityModel.Claims namespace contains classes that represent artifacts in the WCF claims-based identity model. Many of these classes have names that are the same as WIF classes; for example, `Claims` . Do not use these classes when building solutions based on WIF.

- .NET identity and principal classes now derive directly from System.Security.Claims.ClaimsIdentity and System.Security.Claims.ClaimsPrincipal, which represent claims-based identities and principals. For this reason, the WIF 3.5 interfaces `IClaimsIdentity` and `IClaimsPrincipal` are no longer needed and are not available in WIF 4.5.

- Because.NET identity and principal classes such as System.Security.Principal.WindowsIdentity and System.Security.Principal.WindowsPrincipal now derive from ClaimsIdentity and ClaimsPrincipal, they have claims functionality built-in. For this reason, claims-specific identity and principal classes such as `WindowsClaimsIdentity` and `WindowsClaimsPrincipal` that were present in WIF 3.5 are no longer needed and do not exist in WIF 4.5.

Other Changes to WIF Functionality

In addition to the namespace changes and the changes due to integration with .NET, the following general changes to WIF functionality occur in WIF 4.5.

- The `Microsoft.IdentityModel.ExceptionMapper` class, which provided functionality that enabled you to map exceptions to specific SOAP Faults, is removed.
- The exception surface has been greatly reduced.
- Many of the classes that defined protocol or WS-* specific constants have been removed; for example, the `Microsoft.IdentityModel.WSAddressing10Constants` class, which defined constants related to WS-Addressing 1.0.
- The Claims to Windows Token Service (c2wts) and its associated classes in the `Microsoft.IdentityModel.WindowsTokenService` namespace are removed.

WIF Configuration Changes

Many of the changes in the configuration file have been caused by namespace updates in WIF 4.5. For example, consider the following WIF 3.5 entry in the `<httpModules>` section to add the WS-Federation Authentication Manager to an application:

```
<add name="WSFederationAuthenticationModule"
type="Microsoft.IdentityModel.Web.WSFederationAuthenticationModule, Microsoft.IdentityModel, Version=3.5.0.0,
Culture=neutral, PublicKeyToken=abcd … 5678" />
```

This entry has been updated in WIF 4.5 to include the new namespaces and assembly version:

```
<add name="WSFederationAuthenticationModule"
type="System.IdentityModel.Services.WSFederationAuthenticationModule, System.IdentityModel.Services,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=abcd … 5678"/>
```

The following list enumerates the major changes to the configuration file for WIF 4.5.

- The `<microsoft.identityModel>` section is now the <system.identityModel> section.
- The `<service>` element is now the <identityConfiguration> element.
- A new section, <system.identityModel.services>, has been added to specify settings that control behavior in passive (WS-Federation) scenarios.
- The <federationConfiguration> element and its child elements have been moved from the `<service>` element in WIF 3.5 to the new `<system.identityModel.services>` element .
- Several elements that could be specified at the service-level directly under the `<service>` element in WIF 3.5 have been restricted to being specified under the <securityTokenHandlerConfiguration> element. (They may still be specified under the <identityConfiguration> element in WIF 4.5 for backward compatibility.)

For a complete list of the WIF 4.5 configuration elements, see WIF Configuration Schema.

Visual Studio Tooling Changes

The WIF 3.5 SDK offered a stand-alone federation utility, FedUtil.exe (FedUtil), that you could use to outsource identity management in WIF-enabled applications to a security token service (STS). This tool added WIF settings to the application configuration file to enable the application to get security tokens from one or more STSs, and was surfaced in Visual Studio through the **Add STS Service Reference** button. FedUtil does not ship with WIF 4.5. Instead, WIF 4.5 supports a new Visual Studio extension named the Identity and Access Tool for Visual Studio 2012 that you can use to modify your application's configuration file with the WIF settings required to outsource identity management to an STS. The Identity and Access Tool also implements an STS called Local STS that you can use to test your WIF-enabled applications. In many cases, this feature obviates the need to build custom STSs that were often necessary in WIF 3.5 to test solutions under development. For this reason, the STS templates are no longer supported in Visual Studio 2012; however, the classes that support the development of STSs are still available in WIF 4.5.

You can install the Identity and Access Tool from the Extensions and Updates Manager in Visual Studio or you can download it from the following page on Code Gallery: Identity and Access Tool for Visual Studio 2012 on Code Gallery. The Visual Studio tooling changes are summarized in the following list:

- The Add STS Service Reference functionality is removed. The replacement is the Identity and Access Tool.

- The Visual Studio STS templates are removed. You can use the Local STS that is available through the Identity and Access Tool to test identity solutions that you develop with WIF. The Local STS configuration can be modified to customize the claims that it serves.

- The stand-alone Federation Utility (FedUtil) is not available in WIF 4.5. You can use the Identity and Access Tool to modify your configuration files to outsource identity management to an STS.

For more information about the Identity and Access Tool, see Identity and Access Tool for Visual Studio 2012

Passive (WS-Federation) Scenarios:

- Classes that support passive scenarios have been moved to the System.IdentityModel.Services namespace. In WIF 3.5 these classes were in the `Microsoft.IdentityModel.Web` namespace.

- The classes in the `Microsoft.IdentityModel.Web.Configuration` namespace have been moved to System.IdentityModel.Services.Configuration. These classes represent objects specific to configuration in passive scenarios.

- The `FederatedPasssiveSignInControl` is no longer supported; all classes in the `Microsoft.IdentityModel.Web.Controls` namespace have been removed from WIF 4.5.

- The WS-Federation Authentication Module (WSFederationAuthenticationModule) sign-out functionality is different than WIF 3.5. For more details about how sign-out works in WIF 4.5, see the WSFederationAuthenticationModule class topic.

Active (WCF/WS-Trust) Scenarios:

- The `Microsoft.IdentityModel.Protocols.WSTrust` namespace has been split mainly into two namespaces in WIF 4.5. Classes that represent artifacts that are specific to the WS-Trust protocol are now in System.IdentityModel.Protocols.WSTrust. This includes classes like RequestSecurityToken. Classes that represent service contracts, channels, service hosts and other artifacts that are involved in using WS-Trust in WCF applications have been moved to System.ServiceModel.Security; for example, the IWSTrust13AsyncContract interface.

- Configuring a WCF application to use WIF has been greatly simplified. Previously the `Microsoft.IdentityModel.Configuration.ConfigureServiceHostBehaviorExtensionElement` had to be added as a behavior extension and this functionality was then used to wedge WIF into the service behavior by specifying a `<federatedServiceHostConfiguration>` element. WIF 4.5 has been more tightly integrated with WCF. Now you enable WIF on a WCF service by specifying the `useIdentityConfiguration` attribute on the

`<system.serviceModel>` / `<behaviors>` / `<serviceBehaviors>` / `<serviceCredentials>` element as in the following XML:

```
<serviceCredentials useIdentityConfiguration="true">
    <!--Certificate added by Identity And Access VS Package.  Subject='CN=localhost',
Issuer='CN=localhost'. Make sure you have this certificate installed. The Identity and Access tool does
not install this certificate.-->
    <serviceCertificate findValue="CN=localhost" storeLocation="LocalMachine" storeName="My"
x509FindType="FindBySubjectDistinguishedName" />
</serviceCredentials>
```

- In WIF 4.5 the service certificate used by an active (WCF) service must be specified on the service host. In configuration, you can do this through the `<system.serviceModel>` / `<behaviors>` / `<serviceBehaviors>` / `<serviceCredentials>` / `<serviceCertificate>` element as shown in the preceding example. In WIF 3.5 the service certificate could be specified through the `<serviceCertificate>` child element of the WIF `<service>` element. This functionality does not exist in WIF 4.5; specify the `<serviceCertificate>` element under the `<serviceCredentials>` element instead.
- The classes used to wedge WIF 3.5 into WCF no longer exist. This includes the following classes in the `Microsoft.IdentityModel.Tokens` namespace: `FederatedSecurityTokenManager`, `FederatedServiceCredentials`, and `IdentityModelServiceAuthorizationManager`, as well as the `Microsoft.IdentityModel.Configuration.ConfigureServiceHostBehaviorExtensionElement` class.

Enabling WIF 3.5 in Windows 8

Because WIF 4.5 is part of .NET 4.5, it is automatically enabled on computers running Windows 8 and Windows Server 2012 and applications that are built using WIF 4.5 will run under Windows 8 or Windows Server 2012 by default. However, you may need to run applications that were built using WIF 3.5 on a computer that is running Windows 8 or Windows Server 2012. In this case, you need to enable WIF 3.5 on the target computer. On a computer running Windows 8, you can do this by using the Deployment Image Servicing and Management (DISM) tool. On a computer running Windows Server 2012, you can use the DISM tool or you can use the Windows PowerShell `Add-WindowsFeature` cmdlet. In both cases, the appropriate commands can be invoked either at the command line or from inside the Windows PowerShell environment.

The following commands show how to use the DISM tool from either the command line or from inside the Windows PowerShell environment. By default, the DISM PowerShell module is included in Windows 8 and Windows Server 2012 and does not need to be imported. For more information about using DISM with Windows PowerShell, see How to Use DISM in Windows PowerShell.

To enable WIF 3.5 using DISM:

```
dism /online /enable-feature:windows-identity-foundation
```

To disable WIF 3.5 using DISM:

```
dism /online /disable-feature:windows-identity-foundation
```

To check which features are enabled or disabled using DISM:

```
dism /online /get-features
```

Alternatively, on computers running Windows Server 2012, you can enable WIF 3.5 by using the Windows PowerShell `Add-WindowsFeature` cmdlet. To do so from the command line, you can enter the following command:

```
powershell "add-windowsfeature windows-identity-foundation"
```

From inside the Windows PowerShell environment, you can enter the command directly:

```
add-windowsfeature windows-identity-foundation
```

NOTE

Because many of the classes in WIF 3.5 and WIF 4.5 share the same names, when you are using both WIF 3.5 and WIF 4.5 together, be sure to either use fully qualified class names or use namespace aliases to distinguish between classes in WIF 3.5 and WIF 4.5.

See Also

WIF Configuration Schema
Namespace Mapping between WIF 3.5 and WIF 4.5
What's New in Windows Identity Foundation 4.5
Identity and Access Tool for Visual Studio 2012

Namespace Mapping between WIF 3.5 and WIF 4.5

5/3/2017 • 3 min to read • Edit Online

Beginning with .NET 4.5, Windows Identity Foundation (WIF) has been fully integrated into the .NET Framework. This integration engendered name changes and some consolidation of the WIF namespaces and API surface. This topic provides some guidance and a general mapping between the WIF 3.5 namespaces and the WIF 4.5 namespaces. It is not intended to be exhaustive, but rather provide some general information about where to find familiar WIF 3.5 classes in WIF 4.5. For more detailed information about the differences between WIF 3.5 and WIF 4.5, see What's New in Windows Identity Foundation 4.5. For guidance about how to migrate an applications built using WIF 3.5 to WIF 4.5, see Guidelines for Migrating an Application Built Using WIF 3.5 to WIF 4.5.

WIF 3.5 to WIF 4.5 Namespace Map

The WIF classes, which were collected under the `Microsoft.IdentityModel` namespaces in WIF 3.5, are now distributed among the following namespaces: `System.Security.Claims`, `System.ServiceModel.Security`, and the `System.IdentityModel` namespaces in WIF 4.5. In addition some WIF 3.5 namespaces were consolidated or dropped entirely in WIF 4.5.

IMPORTANT

The following `System.IdentityModel` namespaces contain classes that implement the WCF claims-based identity model: System.IdentityModel.Claims, System.IdentityModel.Policy, and System.IdentityModel.Selectors. The WCF claims-based identity model is superseded by WIF. You should not use classes in these three namespaces when building solutions based on WIF.

The following table provides information about where WIF 3.5 classes can be found in WIF 4.5.

WIF 3.5 NAMESPACE	WIF 4.5 NAMESPACE	COMMENTS
`Microsoft.IdentityModel`	System.IdentityModel	- Most of the classes that represent constants are not implemented. - The classes that are used to build security token services have been moved from `Microsoft.IdentityModel.SecurityTokenService` to System.IdentityModel. - The classes in `Microsoft.IdentityModel.Threading` have been moved to System.IdentityModel. - The `ExceptionMapper` and `MruSecurityTokenCache` classes are not implemented.

WIF 3.5 NAMESPACE	WIF 4.5 NAMESPACE	COMMENTS
`Microsoft.IdentityModel.Claims`	System.Security.Claims	- The `IClaimsPrincipal` and `IClaimsIdentity` interfaces are not implemented in WIF 4.5. Instead System.Security.Claims.ClaimsPrincipal and System.Security.Claims.ClaimsIdentity are now the base classes from which most .NET principal and identity classes derive. This means there is no need for specialized claims principal and identity classes like `Microsoft.IdentityModel.Claims.WindowsClaimsPri` and `Microsoft.IdentityModel.Claims.WindowsClaimsIde` in WIF 4.5, use System.Security.Principal.WindowsPrincipal and System.Security.Principal.WindowsIdentity instead. The same is true for other for the other specialized claims principal and identity classes that existed in WIF 3.5. - The `Microsoft.IdentityModel.Claims.ClaimsCollection` class is not implemented in WIF 4.5. Instead, collections of claims are exposed as enumerable collections of type System.Security.Claims.Claim. - System.Security.Claims.ClaimsPrincipal and System.Security.Claims.ClaimsIdentity provide methods that now fully support LINQ.
`Microsoft.IdentityModel.Configuration`	System.IdentityModel.Configuration	Some elements and classes have undergone name changes and some have been dropped in WIF 4.5; for example `Microsoft.IdentityModel.Configuraiton.ServiceCo` is now System.IdentityModel.Configuration.IdentityConfiguration.
`Microsoft.IdentityModel.Protocols`	System.IdentityModel.Services	-
`Microsoft.IdentityModel.Protocols.WSFederation`	System.IdentityModel.Services	-
`Microsoft.IdentityModel.Protocols.WSFederation.Metadata`	System.IdentityModel.Metadata	-
`Microsoft.IdentityModel.Protocols.WSIdentity`	Not Implemented in WIF 4.5	In WIF 3.5 contained classes to support CardSpace, not implemented in WIF 4.5.
`Microsoft.IdentityModel.Protocols.WSTrust`	Split between the System.IdentityModel.Protocols.WSTrust and System.ServiceModel.Security namespaces.	Classes that represents WS-Trust artifacts are in the System.IdentityModel.Protocols.WSTrust namespace; for example, the RequestSecurityToken class. Classes that represent WCF service contracts, service hosts, and channels that enable a WCF service to communicate using the WS-Trust protocol are in the System.ServiceModel.Security namespace; for example, the WSTrustServiceHost class.
`Microsoft.IdentityModel.Protocols.WSTrust.Bindings`	Not Implemented in WIF 4.5	-
`Microsoft.IdentityModel.Protocols.XmlEncryption`	Not Implemented in WIF 4.5	Contained classes that represent XML Encryption constants in WIF 3.5. These constants are not implemented in WIF 4.5.

WIF 3.5 NAMESPACE	WIF 4.5 NAMESPACE	COMMENTS
`Microsoft.IdentityModel.Protocols.XmlSignature`	System.IdentityModel	The `EnvelopingSignature` class and classes that represent constants are not implemented.
`Microsoft.IdentityModel.SecurityTokenService`	Split between the System.IdentityModel, System.IdentityModel.Protocols.WSTrust, and System.IdentityModel.Tokens namespaces.	-
`Microsoft.IdentityModel.Threading`	System.IdentityModel	-
`Microsoft.IdentityModel.Tokens`	System.IdentityModel.Tokens	-
`Microsoft.IdentityModel.Tokens.Saml11`	System.IdentityModel.Tokens	-
`Microsoft.IdentityModel.Tokens.Saml2`	System.IdentityModel.Tokens	-
`Microsoft.IdentityModel.Web`	System.IdentityModel.Services	-
`Microsoft.IdentityModel.Web.Configuration`	System.IdentityModel.Services.Configuration	Classes that provide configuration for passive (WS-Federation) scenarios have largely been moved to System.IdentityModel.Services.Configuration; however, some of these classes are in System.IdentityModel.Services.
`Microsoft.IdentityModel.Web.Controls`	Not Implemented in WIF 4.5	The classes in `Microsoft.IdentityModel.Web.Controls` implemented the Federated Passive Sign-In Control, which does not exist in WIF 4.5.
`Microsoft.IdentityModel.WindowsTokenService`	Not Implemented in WIF 4.5	-

See Also

What's New in Windows Identity Foundation 4.5

Guidelines for Migrating an Application Built Using WIF 3.5 to WIF 4.5

WIF Code Sample Index

5/3/2017 • 1 min to read • Edit Online

The following are code samples for Windows Identity Foundation 4.5:

- ClaimsAwareWebApp - this sample demonstrates basic use of authentication externalization (to the local test Security Token Service from the Identity and Access Tool for Visual Studio 11) on a classic ASP.NET application (as opposed to a web site).
- ClaimsAwareWebService - this sample demonstrates basic use of authentication externalization on a classic WCF service.
- ClaimsAwareMvcApplication - this sample demonstrates how to integrate WIF with MVC, including non-blanket protection and code which honors the forms authentication redirects out of the LogOn controller.
- ClaimsAwareWebFarm - this sample demonstrates a farm ready session cache (as opposed to a tokenreplycache) so that you can use sessions by reference instead of exchanging big cookies. It also demonstrates an easier way of securing cookies in a farm.
- ClaimsAwareFormsAuthentication - this very simple sample demonstrates that in .NET 4.5 you get claims in your principals regardless of how you authenticate your users.
- ClaimsBasedAuthorization- this samples shows how to use your CLaimsAuthorizationManager class and the ClaimsAuthorizationModule for applying your own authorization policies.
- FederationMetadata – this sample demonstrates both dynamic generation (on a custom STS) and dynamic consumption (on a relying party application) of metadata documents.
- CustomToken – this sample demonstrates how to build a custom Simple Web Token (SWT) token type.

See Also

Windows Identity Foundation

WIF Extensions

5/3/2017 • 1 min to read • Edit Online

This section describes the extensions for Windows Identity Foundation.

- JSON Web Token Handler
- Validating Issuer Name Registry

JSON Web Token Handler

5/3/2017 • 1 min to read • Edit Online

The JSON Web Token Handler extension for Windows Identity Foundation enables you to create and validate JSON Web Tokens (JWT) in your applications. The JWT Token Handler can be configured to run in the WIF pipeline like other built-in security token handlers, but it can also be used independently to perform token validation in lightweight applications. The JWT Token Handler is particularly useful when using an OAuth 2.0 bearer token scheme, such as authenticating to Windows Azure Active Directory.

The JWT Token Handler is available as a NuGet package. See Downloading the JSON Web Token Handler Package for more information.

Scenarios

The JWT Token Handler enables the following key scenarios:

- **Validate a JWT Token in a Server Application**: In this scenario, a company named Litware has a server application that uses WIF to handle web sign-on requests. Litware wants to enable their application to use JWT tokens for authentication. The application is updated with the JWT Token Handler, and then the application configuration is updated to add the JWT Token Handler in the WIF pipeline. After the updates have been made and a new request enters the WIF pipeline, the JWT token is validated using the new handler and successful authentication occurs.

- **Validate a JWT Token in a REST Web Service**: In this scenario, a company named Litware has a REST web service that is secured by Windows Azure Active Directory. Requests to the web service must be authenticated by Windows Azure AD, which issues a JWT token upon successful authentication. Litware has a client application that needs to access the web service. The client makes a request to the web service and presents its JWT token from Windows Azure AD, which is then validated by the web service using the JWT Token Handler. After the JWT Token Handler has validated the token, the desired resource is returned to the client by the web service.

Features

The JWT Token Handler offers the following features:

- **Validate a JWT Token**: JWT tokens can be easily validated by the token handler’s validation logic, either as a part of the application’s WIF pipeline or called independently of WIF

- **Create a JWT Token**: The JWT Token Handler can be used to create JWT tokens for authorization in downstream services

Downloading the JSON Web Token Handler Package

5/3/2017 • 1 min to read • Edit Online

This topic discusses how to download and use the JSON Web Token Handler in your project.

Downloading the JSON Web Token Handler

The JSON Web Token Handler extension is available as a NuGet package, which adds the necessary assemblies and references to your project. If you do not already have NuGet installed, go to nuget.org to install it. You can see the versioning history for the extension by visiting its page on NuGet: JSON Web Token Handler on NuGet

Downloading the JSON Web Token Handler by using the Package Manager GUI

1. In Visual Studio, right-click your project in **Solution Explorer**, and then select **Manage NuGet Packages**.
2. In the **Manage NuGet Packages** window, click the search box and enter `JWT Token Handler` and press **Enter**.
3. From the results pane, click the **Install** button for the first result.
4. The package will begin downloading. Before it is added to your project, the License Acceptance dialog will appear. If you agree to the license terms, click **I Accept**.
5. The latest JSON Web Token Handler assemblies will be downloaded and added to your project.

Downloading the JSON Web Token Handler by using the Package Manager Console

1. In Visual Studio, click **Tools**, **Library Package Manager**, and then **Package Manager Console**.
2. The **Package Manager Console** appears. Enter the following text and press **Enter**:

   ```
   Install-Package System.IdentityModel.Tokens.Jwt
   ```

3. The latest JSON Web Token Handler assemblies will be downloaded and added to your project.

JSON Web Token Handler API Reference

5/3/2017 • 1 min to read • Edit Online

This section contains the API Reference for the JSON Web Token Handler WIF Extension.

Validating Issuer Name Registry

5/3/2017 • 1 min to read • Edit Online

The Validating Issuer Name Registry (VINR) for Windows Identity Foundation enables multi-tenant applications to ensure that an incoming token has been issued by a trusted tenant and identity provider. This functionality is particularly useful for multi-tenant applications that use Windows Azure Active Directory because all tokens issued by Windows Azure AD are signed using the same certificate. In order to differentiate between requests from multiple tenants that use the same certificate – and consequently have the same thumbprint – your application must persist the issuer name for each tenant to perform validation logic. The VINR provides this functionality, and it also enables you to add custom validation logic or to store the issuer registry data in locations other than a configuration file. The extension can be added to your application's WIF pipeline or it can be used independently.

The VINR is available as a NuGet package. See Downloading the Validating Issuer Name Registry Package for more information.

Scenarios

The VINR enables the following key scenario:

- **Validate a Token in a Multi-Tenant Application**: In this scenario, a company named Litware has developed a multi-tenant application that uses an identity provider such as Windows Azure AD. This application serves two customers: Contoso and Fabrikam. When a user from Fabrikam authenticates to Litware's application, the resulting token from Windows Azure AD is signed using its standard certificate and the request is issued by Fabrikam. The application needs to verify that both the issuer name and the token is valid, and needs to differentiate requests from Contoso that are signed using the same certificate from Windows Azure AD. The VINR makes it easy for Litware's application to differentiate and validate requests from multiple tenants such as Contoso and Fabrikam.

Features

The VINR offers the following features:

- **Issuer Name and Token Validation for Multi-Tenant Applications**: Validates the incoming token by verifying the issuer name (tenant) and whether the token was signed using a valid certificate from the identity provider.

- **Extensibility for Custom Validation Logic and Data Stores**: Provides extensibility to inject your own validation logic and to specify a data store other than the default configuration file.

Downloading the Validating Issuer Name Registry Package

5/3/2017 • 1 min to read • Edit Online

This topic discusses how to download and use the Validating Issuer Name Registry (VINR) in your project.

Downloading the Validating Issuer Name Registry

The VINR is available as a NuGet package, which adds the necessary assemblies and references to your project. If you do not already have NuGet installed, go to nuget.org to install it. You can see the versioning history for the extension by visiting its page on NuGet: Microsoft Validating Issuer Name Registry on NuGet

Downloading the Validating Issuer Name Registry by using the Package Manager GUI

1. In Visual Studio, right-click your project in **Solution Explorer**, and then select **Manage NuGet Packages**.
2. In the **Manage NuGet Packages** window, click the search box and enter `ValidatingIssuerNameRegistry` and press **Enter**.
3. From the results pane, click the **Install** button for the first result.
4. The package will begin downloading. Before it is added to your project, the License Acceptance dialog will appear. If you agree to the license terms, click **I Accept**.
5. The latest VINR assemblies will be downloaded and added to your project.

Downloading the Validating Issuer Name Registry by using the Package Manager Console

1. In Visual Studio, click **Tools**, **Library Package Manager**, and then **Package Manager Console**.
2. The **Package Manager Console** appears. Enter the following text and press **Enter**:

   ```
   Install-Package System.IdentityModel.Tokens.ValidatingIssuerNameRegistry
   ```

3. The latest VINR assemblies will be downloaded and added to your project.

Validating Issuer Name Registry API Reference

5/3/2017 • 1 min to read • Edit Online

This section contains the API Reference for the Validating Issuer Name Registry WIF Extension.

WIF API Reference

5/3/2017 • 1 min to read • Edit Online

Windows Identity Foundation (WIF) classes are split across the following assemblies: `mscorlib` (mscorlib.dll), `System.IdentityModel` (System.IdentityModel.dll), `System.IdentityModel.Services` (System.IdentityModel.Services.dll), and `System.ServiceModel` (System.ServiceModel.dll). This topic provides links to the WIF namespaces and brief explanations of the classes that each namespace contains.

IMPORTANT

The following `System.IdentityModel` namespaces contain classes that implement the WCF claims-based identity model: System.IdentityModel.Claims, System.IdentityModel.Policy, and System.IdentityModel.Selectors. Starting with .NET Framework 4.5, the WCF claims-based identity model is superseded by WIF. You should not use classes in these three namespaces when building solutions based on WIF.

System.IdentityModel
Contains classes that represent cookie transforms, security token services, and specialized XML dictionary readers.

System.IdentityModel.Configuration
Contains classes that provide configuration for applications and services built using the Windows Identity Foundation (WIF). The classes in this namespace represent settings under the <identityConfiguration> element.

System.IdentityModel.Metadata
Contains classes that represent elements in a Federation Metadata document.

System.IdentityModel.Protocols.WSTrust
Contains classes that represent WS-Trust artifacts.

System.IdentityModel.Services
Contains classes that are used in passive (WS-Federation) scenarios. Also contains some classes that represent settings under the <system.identityModel.services> element. Settings under this element configure WS-Federation for applications. The `System.IdentityModel.Services.Configuration` namespace contains most of the classes that are used to configure WS-Federation.

System.IdentityModel.Services.Configuration
Contains classes that provide configuration for WIF applications that use the WS-Federation protocol. The classes in this namespace represent settings under the <system.identityModel.services> element. The `System.IdentityModel.Services` namespace also contains some classes that are used to configure WS-Federation.

System.IdentityModel.Services.Tokens
Contains specialized security token handlers for Web farm scenarios.

System.IdentityModel.Tokens
Contains classes that represent security tokens, security token handlers, and other security token artifacts.

System.Security.Claims
Contains classes that represent claims, claims-based identities, claims-based principals, and other claims-based identity model artifacts.

System.ServiceModel.Security
Contains classes that represent WCF contracts, channels, service hosts and other artifacts that are used in active (WS-Trust) scenarios. This namespace also contains classes that are specific to Windows Communication

Foundation (WCF) and that are not used by WIF.

See Also

WIF Configuration Reference

Namespace Mapping between WIF 3.5 and WIF 4.5

WIF Configuration Reference

5/3/2017 • 1 min to read • Edit Online

You can configure Windows Identity Foundation (WIF) in your applications by adding elements to a configuration file. This topic contains links to reference topics for the WIF configuration elements.

WIF Configuration Schema
The reference for the WIF configuration elements.

WIF Configuration Schema Conventions
Contains information about general attributes and formats used by the WIF configuration elements.

WIF Configuration Schema Conventions

5/3/2017 • 1 min to read • Edit Online

This topic discusses conventions used throughout the Windows Identity Foundation (WIF) configuration topics and describes some common features and attributes used in the <system.identityModel> and the <system.identityModel.services> sections.

Modes

Many of the WIF configuration elements have a `mode` attribute. This attribute typically controls which class is used to do a particular part of the processing and which configuration elements are allowed as child elements of the current element. A configuration error will be raised if elements that are included in the configuration file are ignored because of the mode settings.

Timespan Values

Where TimeSpan is used as the type of an attribute, see the Parse(String) method to see the allowed format. This format conforms to the following specification.

```
[ws][-]{ d | [d.]hh:mm[:ss[.ff]] }[ws]
```

For example, "30", "30.00:00", "30.00:00:00" all mean 30 days; and "00:05", "00:05:00", "0.00:05:00.00" all mean 5 minutes.

Certificate References

Several elements take references to certificates using the `<certificateReference>` element. When referencing a certificate, the following attributes are available.

`storeLocation`	A value of the StoreLocation enumeration: `CurrentUser` or `CurrentMachine`.
`storeName`	A value of the StoreName enumeration; the most useful for this context are `My` and `TrustedPeople`.
`x509FindType`	A value of the X509FindType enumeration; the most useful for this context are `FindBySubjectName` and `FindByThumbprint`. To eliminate the chance of error, it is recommended that the `FindByThumbprint` type be used in production environments.
`findValue`	The value used to find the certificate, based on the `x509FindType` attribute. To eliminate the chance of error, it is recommended that the `FindByThumbprint` type be used in production environments. When `FindByThumbPrint` is specified, this attribute takes a value that is the hexadecimal-string form of the certificate thumbprint; for example, "97249e1a5fa6bee5e515b82111ef524a4c91583f".

Custom Type References

Several elements reference custom types using the `type` attribute. This attribute should specify the name of the custom type. To reference a type from the Global Assembly Cache (GAC), a strong name should be used. To reference a type from an assembly in the Bin/ directory, a simple assembly-qualified reference may be used. Types defined in the App_Code/ directory may also be referenced by simply specifying the type name with no qualifying assembly.

Custom types must be derived from the type specified and they must provide a `public` default (0 argument) constructor.

See Also

<system.identityModel>
<system.identityModel.services>

Security Changes in the .NET Framework

4/14/2017 • 1 min to read • Edit Online

The most important change to security in the .NET Framework 4.5 is in strong naming. See Enhanced Strong Naming for a description of those changes.

The .NET Framework provides a two-tier security model for managed applications. Windows 8.x Store apps run in a Windows security container that limits access to resources. Within that container, managed applications run fully trusted. From a code access security (CAS) perspective, there is nothing a developer can do to elevate privileges. For information about the privileges granted by Windows, see App capability declarations (Windows Store apps) in the Windows Dev Center. For information about creating a Windows 8.x Store app, see Create your first Windows Store app using C# or Visual Basic.

Secure Coding Guidelines for Unmanaged Code

5/31/2017 • 4 min to read • Edit Online

Some library code needs to call into unmanaged code (for example, native code APIs, such as Win32). Because this means going outside the security perimeter for managed code, due caution is required. If your code is security-neutral, both your code and any code that calls it must have unmanaged code permission (SecurityPermission with the UnmanagedCode flag specified).

However, it is often unreasonable for your caller to have such powerful permissions. In such cases, your trusted code can be the go-between, similar to the managed wrapper or library code described in Securing Wrapper Code. If the underlying unmanaged code functionality is totally safe, it can be directly exposed; otherwise, a suitable permission check (demand) is required first.

When your code calls into unmanaged code but you do not want to require your callers to have permission to access unmanaged code, you must assert that right. An assertion blocks the stack walk at your frame. You must be careful that you do not create a security hole in this process. Usually, this means that you must demand a suitable permission of your callers and then use unmanaged code to perform only what that permission allows and no more. In some cases (for example, a get time-of-day function), unmanaged code can be directly exposed to callers without any security checks. In any case, any code that asserts must take responsibility for security.

Because any managed code that provides a code path into native code is a potential target for malicious code, determining which unmanaged code can be safely used and how it must be used requires extreme care. Generally, unmanaged code should never be directly exposed to partially trusted callers. There are two primary considerations in evaluating the safety of unmanaged code use in libraries that are callable by partially trusted code:

- **Functionality**. Does the unmanaged API provide functionality that does not allow callers to perform potentially dangerous operations? Code access security uses permissions to enforce access to resources, so consider whether the API uses files, a user interface, or threading, or whether it exposes protected information. If it does, the managed code wrapping it must demand the necessary permissions before allowing it to be entered. Additionally, while not protected by a permission, memory access must be confined to strict type safety.

- **Parameter checking**. A common attack passes unexpected parameters to exposed unmanaged code API methods in an attempt to cause them to operate out of specification. Buffer overruns using out-of-range index or offset values are one common example of this type of attack, as are any parameters that might exploit a bug in the underlying code. Thus, even if the unmanaged code API is functionally safe (after necessary demands) for partially trusted callers, managed code must also check parameter validity exhaustively to ensure that no unintended calls are possible from malicious code using the managed code wrapper layer.

Using SuppressUnmanagedCodeSecurityAttribute

There is a performance aspect to asserting and then calling unmanaged code. For every such call, the security system automatically demands unmanaged code permission, resulting in a stack walk each time. If you assert and immediately call unmanaged code, the stack walk can be meaningless: it consists of your assert and your unmanaged code call.

A custom attribute called SuppressUnmanagedCodeSecurityAttribute can be applied to unmanaged code entry points to disable the normal security check that demands SecurityPermission with the UnmanagedCode permission specified. Extreme caution must always be taken when doing this, because this action creates an open door into unmanaged code with no runtime security checks. It should be noted that even with

SuppressUnmanagedCodeSecurityAttribute applied, there is a one-time security check that happens at just-in-time (JIT) compilation to ensure that the immediate caller has permission to call unmanaged code.

If you use the **SuppressUnmanagedCodeSecurityAttribute**, check the following points:

- Make the unmanaged code entry point internal or otherwise inaccessible outside your code.
- Any call into unmanaged code is a potential security hole. Make sure your code is not a portal for malicious code to indirectly call into unmanaged code and avoid a security check. Demand permissions, if appropriate.
- Use a naming convention to explicitly identify when you are creating a dangerous path into unmanaged code, as described in the section below..

Naming convention for unmanaged code methods

A useful and highly recommended convention has been established for naming unmanaged code methods. All unmanaged code methods are separated into three categories: **safe**, **native**, and **unsafe**. These keywords can be used as class names within which the various kinds of unmanaged code entry points are defined. In source code, these keywords should be added to the class name, as in `Safe.GetTimeOfDay`, `Native.Xyz`, or `Unsafe.DangerousAPI`, for example. Each of these keywords provides useful security information for developers using that class, as described in the following table.

KEYWORD	SECURITY CONSIDERATIONS
safe	Completely harmless for any code, even malicious code, to call. Can be used just like other managed code. For example, a function that gets the time of day is typically safe.
native	Security-neutral; that is, unmanaged code that requires unmanaged code permission to call. Security is checked, which stops an unauthorized caller.
unsafe	Potentially dangerous unmanaged code entry point with security suppressed. Developers should use the greatest caution when using such unmanaged code, making sure that other protections are in place to prevent a security vulnerability. Developers must be responsible, as this keyword overrides the security system.

See Also

Secure Coding Guidelines