Protected web API: Verify scopes and app roles

Article • 04/17/2023

This article describes how you can add authorization to your web API. This protection ensures that the API is called only by:

- Applications on behalf of users who have the right scopes and roles.
- Daemon apps that have the right application roles.

The code snippets in this article are extracted from the following code samples on GitHub:

- ASP.NET Core web API incremental tutorial
- ASP.NET web API sample

To protect an ASP.NET or ASP.NET Core web API, you must add the `[Authorize]` attribute to one of the following items:

- The controller itself if you want all controller actions to be protected
- The individual controller action for your API

C#

```csharp
[Authorize]
public class TodoListController : Controller
{
 // ...
}
```

But this protection isn't enough. It guarantees only that ASP.NET and ASP.NET Core validate the token. Your API needs to verify that the token used to call the API is requested with the expected claims. These claims in particular need verification:

- The *scopes* if the API is called on behalf of a user.
- The *app roles* if the API can be called from a daemon app.

Verify scopes in APIs called on behalf of users

If a client app calls your API on behalf of a user, the API needs to request a bearer token that has specific scopes for the API. For more information, see Code configuration | Bearer token.

ASP.NET Core

In ASP.NET Core, you can use Microsoft.Identity.Web to verify scopes in each controller action. You can also verify them at the level of the controller or for the whole application.

Verify the scopes on each controller action

You can verify the scopes in the controller action by using the `[RequiredScope]` attribute. This attribute has several overrides. One that takes the required scopes directly, and one that takes a key to the configuration.

Verify the scopes on a controller action with hardcoded scopes

The following code snippet shows the usage of the `[RequiredScope]` attribute with hardcoded scopes.

C#

```csharp
using Microsoft.Identity.Web

[Authorize]
public class TodoListController : Controller
{
    /// <summary>
    /// The web API will accept only tokens that have the `ac-
cess_as_user` scope for
    /// this API.
    /// </summary>
    const string scopeRequiredByApi = "access_as_user";

    // GET: api/values
    [HttpGet]
    [RequiredScope(scopeRequiredByApi)]
    public IEnumerable<TodoItem> Get()
    {
        // Do the work and return the result.
        // ...
    }
```

```
 // ...
}
```

Verify the scopes on a controller action with scopes defined in configuration

You can also declare these required scopes in the configuration, and reference the configuration key:

For instance if, in the appsettings.json you have the following configuration:

JSon

```
{
 "AzureAd" : {
   // more settings
   "Scopes" : "access_as_user access_as_admin"
  }
}
```

Then, reference it in the `[RequiredScope]` attribute:

C#

```
using Microsoft.Identity.Web

[Authorize]
public class TodoListController : Controller
{
    // GET: api/values
    [HttpGet]
    [RequiredScope(RequiredScopesConfigurationKey = "AzureAd:Scopes")]
    public IEnumerable<TodoItem> Get()
    {
        // Do the work and return the result.
        // ...
    }
 // ...
}
```

Verify scopes conditionally

There are cases where you want to verify scopes conditionally. You can do this using the `VerifyUserHasAnyAcceptedScope` extension method on the `HttpContext`.

C#

```csharp
using Microsoft.Identity.Web

[Authorize]
public class TodoListController : Controller
{
    /// <summary>
    /// The web API will accept only tokens 1) for users, 2) that have
the `access_as_user` scope for
    /// this API.
    /// </summary>
    static readonly string[] scopeRequiredByApi = new string[] { "ac-
cess_as_user" };

    // GET: api/values
    [HttpGet]
    public IEnumerable<TodoItem> Get()
    {
         HttpContext.VerifyUserHasAnyAcceptedScope(scopeRequiredByApi);
        // Do the work and return the result.
        // ...
    }
 // ...
}
```

Verify the scopes at the level of the controller

You can also verify the scopes for the whole controller

Verify the scopes on a controller with hardcoded scopes

The following code snippet shows the usage of the `[RequiredScope]` attribute with hardcoded scopes on the controller. To use the RequiredScopeAttribute, you'll need to either:

- Use `AddMicrosoftIdentitWebApi` in the Startup.cs, as seen in Code configuration
- or otherwise add the `ScopeAuthorizationRequirement` to the authorization policies as explained in authorization policies .

C#

```
using Microsoft.Identity.Web

[Authorize]
[RequiredScope(scopeRequiredByApi)]
public class TodoListController : Controller
{
    /// <summary>
    /// The web API will accept only tokens 1) for users, 2) that have
the `access_as_user` scope for
    /// this API.
    /// </summary>
    static readonly string[] scopeRequiredByApi = new string[] { "ac-
cess_as_user" };

    // GET: api/values
    [HttpGet]
    public IEnumerable<TodoItem> Get()
    {
        // Do the work and return the result.
        // ...
    }
 // ...
}
```

Verify the scopes on a controller with scopes defined in configuration

Like on action, you can also declare these required scopes in the configuration, and reference the configuration key:

C#

```
using Microsoft.Identity.Web

[Authorize]
[RequiredScope(RequiredScopesConfigurationKey = "AzureAd:Scopes")]
public class TodoListController : Controller
{
    // GET: api/values
    [HttpGet]
    public IEnumerable<TodoItem> Get()
    {
        // Do the work and return the result.
        // ...
```

```
      }
   // ...
  }
```

Verify the scopes more globally

Defining granular scopes for your web API and verifying the scopes in each controller action is the recommended approach. However it's also possible to verify the scopes at the level of the application or a controller. For details, see Claim-based authorization in the ASP.NET core documentation.

What is verified?

The `[RequiredScope]` attribute and `VerifyUserHasAnyAcceptedScope` method, does something like the following steps:

- Verify there's a claim named `http://schemas.microsoft.com/identity/claims/scope` or `scp`.
- Verify the claim has a value that contains the scope expected by the API.

Verify app roles in APIs called by daemon apps

If your web API is called by a daemon app, that app should require an application permission to your web API. As shown in Exposing application permissions (app roles), your API exposes such permissions. One example is the `access_as_application` app role.

You now need to have your API verify that the token it receives contains the `roles` claim and that this claim has the expected value. The verification code is similar to the code that verifies delegated permissions, except that your controller action tests for roles instead of scopes:

ASP.NET Core

The following code snippet shows how to verify the application role

C#

```csharp
using Microsoft.Identity.Web

[Authorize]
public class TodoListController : ApiController
{
    public IEnumerable<TodoItem> Get()
    {
        HttpContext.ValidateAppRole("access_as_application");
        // ...
    }
```

Instead, you can use the [Authorize(Roles = "access_as_application")] attributes on the controller or an action (or a razor page).

CSharp

```csharp
[Authorize(Roles = "access_as_application")]
MyController : ApiController
{
    // ...
}
```

Role-based authorization in ASP.NET Core lists several approaches to implement role based authorization. Developers can choose one among them which suits to their respective scenarios.

For working samples, see the web app incremental tutorial on authorization by roles and groups .

Verify app roles in APIs called on behalf of users

Users can also use roles claims in user assignment patterns, as shown in How to add app roles in your application and receive them in the token. If the roles are assignable to both, checking roles will let apps sign in as users and users sign in as apps. We recommend that you declare different roles for users and apps to prevent this confusion.

If you have defined app roles with user/group, then roles claim can also be verified in the API along with scopes. The verification logic of the app roles in this scenario remains same as if API is called by the daemon apps since there is no differentiation in the role claim for user/group and application.

Accepting app-only tokens if the web API should be called only by daemon apps

If you want only daemon apps to call your web API, add the condition that the token is an app-only token when you validate the app role.

C#

```
string oid = ClaimsPrincipal.Current.FindFirst("oid")?.Value;
string sub = ClaimsPrincipal.Current.FindFirst("sub")?.Value;
bool isAppOnly = oid != null && sub != null && oid == sub;
```

Checking the inverse condition allows only apps that sign in a user to call your API.

Using ACL-based authorization

Alternatively to app-roles based authorization, you can protect your web API with an Access Control List (ACL) based authorization pattern to control tokens without the roles claim.

If you are using Microsoft.Identity.Web on ASP.NET core, you'll need to declare that you are using ACL-based authorization, otherwise Microsoft Identity Web will throw an exception when neither roles nor scopes are in the Claims provided:

Text

```
System.UnauthorizedAccessException: IDW10201: Neither scope or roles claim
was found in the bearer token.
```

To avoid this exception, set the `AllowWebApiToBeAuthorizedByACL` configuration property to true, in the appsettings.json or programmatically.

Json

```
{
 "AzureAD"
 {
  // other properties
```

```
    "AllowWebApiToBeAuthorizedByACL" : true,
    // other properties
   }
  }
```

If you set `AllowWebApiToBeAuthorizedByACL` to true, this is **your responsibility** to ensure the ACL mechanism.

Next steps

Move on to the next article in this scenario, Move to production.