

Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core

12/05/2019 • 14 minutes to read • S R D S G +15

In this article

[Authentication fundamentals](#)

[ASP.NET Core antiforgery configuration](#)

[Antiforgery options](#)

[Configure antiforgery features with IAntiforgery](#)

[Refresh tokens after authentication](#)

[JavaScript, AJAX, and SPAs](#)

[Extend antiforgery](#)

[Additional resources](#)


By [Rick Anderson](#), [Fiyaz Hasan](#), and [Steve Smith](#)

Cross-site request forgery (also known as XSRF or CSRF) is an attack against web-hosted apps whereby a malicious web app can influence the interaction between a client browser and a web app that trusts that browser. These attacks are possible because web browsers send some types of authentication tokens automatically with every request to a website. This form of exploit is also known as a *one-click attack* or *session riding* because the attack takes advantage of the user's previously authenticated session.

An example of a CSRF attack:

1. A user signs into `www.good-banking-site.com` using forms authentication. The server authenticates the user and issues a response that includes an authentication cookie. The site is vulnerable to attack because it trusts any request that it receives with a valid authentication cookie.
2. The user visits a malicious site, `www.bad-crook-site.com`.

The malicious site, `www.bad-crook-site.com`, contains an HTML form similar to the following:

HTML	 Copy
<pre><h1>Congratulations! You're a Winner!</h1> <form action="http://good-banking-site.com/api/account"></pre>	

```
method="post">
  <input type="hidden" name="Transaction" value="withdraw">
  <input type="hidden" name="Amount" value="1000000">
  <input type="submit" value="Click to collect your prize!">
</form>
```

Notice that the form's `action` posts to the vulnerable site, not to the malicious site. This is the "cross-site" part of CSRF.

3. The user selects the submit button. The browser makes the request and automatically includes the authentication cookie for the requested domain, `www.good-banking-site.com`.
4. The request runs on the `www.good-banking-site.com` server with the user's authentication context and can perform any action that an authenticated user is allowed to perform.

In addition to the scenario where the user selects the button to submit the form, the malicious site could:

- Run a script that automatically submits the form.
- Send the form submission as an AJAX request.
- Hide the form using CSS.

These alternative scenarios don't require any action or input from the user other than initially visiting the malicious site.

Using HTTPS doesn't prevent a CSRF attack. The malicious site can send an `https://www.good-banking-site.com/` request just as easily as it can send an insecure request.

Some attacks target endpoints that respond to GET requests, in which case an image tag can be used to perform the action. This form of attack is common on forum sites that permit images but block JavaScript. Apps that change state on GET requests, where variables or resources are altered, are vulnerable to malicious attacks. **GET requests that change state are insecure. A best practice is to never change state on a GET request.**

CSRF attacks are possible against web apps that use cookies for authentication because:

- Browsers store cookies issued by a web app.
- Stored cookies include session cookies for authenticated users.
- Browsers send all of the cookies associated with a domain to the web app every request regardless of how the request to app was generated within the browser.

However, CSRF attacks aren't limited to exploiting cookies. For example, Basic and Digest authentication are also vulnerable. After a user signs in with Basic or Digest authentication, the browser automatically sends the credentials until the session[†] ends.

[†]In this context, *session* refers to the client-side session during which the user is authenticated. It's unrelated to server-side sessions or [ASP.NET Core Session Middleware](#).

Users can guard against CSRF vulnerabilities by taking precautions:

- Sign off of web apps when finished using them.
- Clear browser cookies periodically.

However, CSRF vulnerabilities are fundamentally a problem with the web app, not the end user.

Authentication fundamentals

Cookie-based authentication is a popular form of authentication. Token-based authentication systems are growing in popularity, especially for Single Page Applications (SPAs).

Cookie-based authentication

When a user authenticates using their username and password, they're issued a token, containing an authentication ticket that can be used for authentication and authorization. The token is stored as a cookie that accompanies every request the client makes. Generating and validating this cookie is performed by the Cookie Authentication Middleware. The [middleware](#) serializes a user principal into an encrypted cookie. On subsequent requests, the middleware validates the cookie, recreates the principal, and assigns the principal to the [User](#) property of [HttpContext](#).

Token-based authentication

When a user is authenticated, they're issued a token (not an antiforgery token). The token contains user information in the form of [claims](#) or a reference token that points the app to user state maintained in the app. When a user attempts to access a resource requiring authentication, the token is sent to the app with an additional authorization header in form of Bearer token. This makes the app stateless. In each subsequent request, the token is passed in the request for server-side validation. This token isn't *encrypted*; it's *encoded*. On

the server, the token is decoded to access its information. To send the token on subsequent requests, store the token in the browser's local storage. Don't be concerned about CSRF vulnerability if the token is stored in the browser's local storage. CSRF is a concern when the token is stored in a cookie. For more information, see the [GitHub issue SPA code sample adds two cookies](#).

Multiple apps hosted at one domain

Shared hosting environments are vulnerable to session hijacking, login CSRF, and other attacks.

Although `example1.contoso.net` and `example2.contoso.net` are different hosts, there's an implicit trust relationship between hosts under the `*.contoso.net` domain. This implicit trust relationship allows potentially untrusted hosts to affect each other's cookies (the same-origin policies that govern AJAX requests don't necessarily apply to HTTP cookies).

Attacks that exploit trusted cookies between apps hosted on the same domain can be prevented by not sharing domains. When each app is hosted on its own domain, there is no implicit cookie trust relationship to exploit.

ASP.NET Core antiforgery configuration


Warning

ASP.NET Core implements antiforgery using **ASP.NET Core Data Protection**. The data protection stack must be configured to work in a server farm. See **Configuring data protection** for more information.

Antiforgery middleware is added to the [Dependency injection](#) container when one of the following APIs is called in `Startup.ConfigureServices`:

- [AddMvc](#)
- [MapRazorPages](#)
- [MapControllerRoute](#)
- [MapBlazorHub](#)

In ASP.NET Core 2.0 or later, the [FormTagHelper](#) injects antiforgery tokens into HTML form elements. The following markup in a Razor file automatically generates antiforgery tokens:

CSSHTML	 Copy
<pre><form method="post"> ... </form></pre>	


Similarly, [IHtmlHelper.BeginForm](#) generates antiforgery tokens by default if the form's method isn't GET.

The automatic generation of antiforgery tokens for HTML form elements happens when the `<form>` tag contains the `method="post"` attribute and either of the following are true:


- The action attribute is empty (`action=""`).
- The action attribute isn't supplied (`<form method="post">`).

Automatic generation of antiforgery tokens for HTML form elements can be disabled:

- Explicitly disable antiforgery tokens with the `asp-antiforgery` attribute:

CSSHTML	 Copy
<pre><form method="post" asp-antiforgery="false"> ... </form></pre>	

- The form element is opted-out of Tag Helpers by using the Tag Helper [! opt-out symbol](#):

CSSHTML	 Copy
<pre><!form method="post"> ... <!/form></pre>	

- Remove the `FormTagHelper` from the view. The `FormTagHelper` can be removed from a view by adding the following directive to the Razor view:

CSSHTML	 Copy
<pre>@removeTagHelper Microsoft.AspNetCore.Mvc.TagHelpers.FormTagHelper, Microsoft.AspNetCore.Mvc.TagHelpers</pre>	


❗ Note

Razor Pages are automatically protected from XSRF/CSRF. For more information, see [XSRF/CSRF and Razor Pages](#).


The most common approach to defending against CSRF attacks is to use the *Synchronizer Token Pattern* (STP). STP is used when the user requests a page with form data:

1. The server sends a token associated with the current user's identity to the client.
2. The client sends back the token to the server for verification.
3. If the server receives a token that doesn't match the authenticated user's identity, the request is rejected.

The token is unique and unpredictable. The token can also be used to ensure proper sequencing of a series of requests (for example, ensuring the request sequence of: page 1 > page 2 > page 3). All of the forms in ASP.NET Core MVC and Razor Pages templates generate antiforgery tokens. The following pair of view examples generate antiforgery tokens:

CSHTML	 Copy
<pre><form asp-controller="Manage" asp-action="ChangePassword" method="post"> ... </form> @using (Html.BeginForm("ChangePassword", "Manage")) { ... }</pre>	

Explicitly add an antiforgery token to a `<form>` element without using Tag Helpers with the HTML helper [@Html.AntiForgeryToken](#):

CSHTML	 Copy
<pre><form action="/" method="post"> @Html.AntiForgeryToken() </form></pre>	

In each of the preceding cases, ASP.NET Core adds a hidden form field similar to the following:

CSSHTML	 Copy
<pre><input name="__RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ... s2-m9Yw"></pre>	

ASP.NET Core includes three [filters](#) for working with antiforgery tokens:

- [ValidateAntiForgeryToken](#)
- [AutoValidateAntiforgeryToken](#)
- [IgnoreAntiforgeryToken](#)

Antiforgery options

Customize [antiforgery options](#) in `Startup.ConfigureServices`:

C#	 Copy
<pre>services.AddAntiforgery(options => { // Set Cookie properties using CookieBuilder properties†. options.FormFieldName = "AntiforgeryFieldname"; options.HeaderName = "X-CSRF-TOKEN-HEADERNAME"; options.SuppressXFrameOptionsHeader = false; });</pre>	


†Set the antiforgery `Cookie` properties using the properties of the [CookieBuilder](#) class.

Option	Description
Cookie	Determines the settings used to create the antiforgery cookies.
FormFieldName	The name of the hidden form field used by the antiforgery system to render antiforgery tokens in views.
HeaderName	The name of the header used by the antiforgery system. If <code>null</code> , the system considers only form data.
SuppressXFrameOptionsHeader	Specifies whether to suppress generation of the <code>X-Frame-Options</code> header. By default, the header is generated with a value of "SAMEORIGIN". Defaults to <code>false</code> .

For more information, see [CookieAuthenticationOptions](#).


Configure antiforgery features with IAntiforgery

[IAntiforgery](#) provides the API to configure antiforgery features. [IAntiforgery](#) can be requested in the `Configure` method of the `Startup` class. The following example uses middleware from the app's home page to generate an antiforgery token and send it in the response as a cookie (using the default Angular naming convention described later in this topic):

C#	 Copy
<pre>public void Configure(IApplicationBuilder app, IAntiforgery antiforgery) { app.Use(next => context => { string path = context.Request.Path.Value; if (string.Equals(path, "/", StringComparison.OrdinalIgnoreCase) string.Equals(path, "/index.html", StringComparison.OrdinalIgnoreCase)) { // The request token can be sent as a JavaScript-readable cookie, // and Angular uses it by default. var tokens = antiforgery.GetAndStoreTokens(context); context.Response.Cookies.Append("XSRF-TOKEN", tokens.RequestToken, new CookieOptions() { HttpOnly = false }); } return next(context); }); }</pre>	

Require antiforgery validation

[ValidateAntiForgeryToken](#) is an action filter that can be applied to an individual action, a controller, or globally. Requests made to actions that have this filter applied are blocked unless the request includes a valid antiforgery token.

C#	 Copy
----	--


```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> RemoveLogin(RemoveLoginViewModel account)
{
    ManageMessageId? message = ManageMessageId.Error;
    var user = await GetCurrentUserAsync();

    if (user != null)
    {
        var result =
            await _userManager.RemoveLoginAsync(
                user, account.LoginProvider, account.ProviderKey);

        if (result.Succeeded)
        {
            await _signInManager.SignInAsync(user, isPersistent:
false);
            message = ManageMessageId.RemoveLoginSuccess;
        }
    }

    return RedirectToAction(nameof(ManageLogins), new { Message = message });
}

```

The `ValidateAntiForgeryToken` attribute requires a token for requests to the action methods it marks, including HTTP GET requests. If the `ValidateAntiForgeryToken` attribute is applied across the app's controllers, it can be overridden with the `IgnoreAntiforgeryToken` attribute.

❗ Note

ASP.NET Core doesn't support adding antiforgery tokens to GET requests automatically.

Automatically validate antiforgery tokens for unsafe HTTP methods only

ASP.NET Core apps don't generate antiforgery tokens for safe HTTP methods (GET, HEAD, OPTIONS, and TRACE). Instead of broadly applying the `ValidateAntiForgeryToken` attribute and then overriding it with `IgnoreAntiforgeryToken` attributes, the


`AutoValidateAntiforgeryToken` attribute can be used. This attribute works identically to the `ValidateAntiForgeryToken` attribute, except that it doesn't require tokens for requests made using the following HTTP methods:

- GET
- HEAD
- OPTIONS
- TRACE


We recommend use of `AutoValidateAntiforgeryToken` broadly for non-API scenarios. This ensures POST actions are protected by default. The alternative is to ignore antiforgery tokens by default, unless `ValidateAntiForgeryToken` is applied to individual action methods. It's more likely in this scenario for a POST action method to be left unprotected by mistake, leaving the app vulnerable to CSRF attacks. All POSTs should send the antiforgery token.

APIs don't have an automatic mechanism for sending the non-cookie part of the token. The implementation probably depends on the client code implementation. Some examples are shown below:

Class-level example:

C#	 Copy
<pre>[Authorize] [AutoValidateAntiforgeryToken] public class ManageController : Controller {</pre>	

Global example:

C#	 Copy
<pre>services.AddControllersWithViews(options => options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute()));</pre>	

Override global or controller antiforgery attributes

The `IgnoreAntiforgeryToken` filter is used to eliminate the need for an antiforgery token for a given action (or controller). When applied, this filter overrides

`ValidateAntiForgeryToken` and `AutoValidateAntiforgeryToken` filters specified at a higher level (globally or on a controller).

C#	 Copy
<pre>[Authorize] [AutoValidateAntiforgeryToken] public class ManageController : Controller { [HttpPost] [IgnoreAntiforgeryToken] public async Task<IActionResult> DoSomethingSafe(SomeViewModel model) { // no antiforgery token required } }</pre>	

Refresh tokens after authentication

Tokens should be refreshed after the user is authenticated by redirecting the user to a view or Razor Pages page.

JavaScript, AJAX, and SPAs

In traditional HTML-based apps, antiforgery tokens are passed to the server using hidden form fields. In modern JavaScript-based apps and SPAs, many requests are made programmatically. These AJAX requests may use other techniques (such as request headers or cookies) to send the token.

If cookies are used to store authentication tokens and to authenticate API requests on the server, CSRF is a potential problem. If local storage is used to store the token, CSRF vulnerability might be mitigated because values from local storage aren't sent automatically to the server with every request. Thus, using local storage to store the antiforgery token on the client and sending the token as a request header is a recommended approach.

JavaScript

Using JavaScript with views, the token can be created using a service from within the view. Inject the `Microsoft.AspNetCore.Antiforgery.IAntiforgery` service into the view and call

GetAndStoreTokens:

CSSHTML

 Copy

```
@{
    ViewData["Title"] = "AJAX Demo";
}
@inject Microsoft.AspNetCore.Antiforgery.IAntiforgery Xsrf
@functions{
    public string GetAntiXsrfRequestToken()
    {
        return Xsrf.GetAndStoreTokens(Context).RequestToken;
    }
}

<input type="hidden" id="RequestVerificationToken"
    name="RequestVerificationToken"
    value="@GetAntiXsrfRequestToken()">

<h2>@ViewData["Title"]</h2>
<h3>@ViewData["Message"]</h3>

<div class="row">
    <p><input type="button" id="antiforgery" value="Antiforgery"></p>
    <script>
        var xmlhttp = new XMLHttpRequest();
        xmlhttp.onreadystatechange = function() {
            if (xmlhttp.readyState == XMLHttpRequest.DONE) {
                if (xmlhttp.status == 200) {
                    alert(xmlhttp.responseText);
                } else {
                    alert('There was an error processing the AJAX
request. ');
                }
            }
        };

        document.addEventListener('DOMContentLoaded', function() {
            document.getElementById("antiforgery").onclick = function
() {
                xmlhttp.open('POST', '@Url.Action("Antiforgery",
"Home")', true);
                xmlhttp.setRequestHeader("RequestVerificationToken",
document.getElementById('RequestVerificationToken').value);
                xmlhttp.send();
            }
        });
    </script>
</div>
```


This approach eliminates the need to deal directly with setting cookies from the server or reading them from the client.

The preceding example uses JavaScript to read the hidden field value for the AJAX POST header.


JavaScript can also access tokens in cookies and use the cookie's contents to create a header with the token's value.

C#	 Copy
<pre>context.Response.Cookies.Append("CSRF-TOKEN", tokens.RequestToken, new Microsoft.AspNetCore.Http.CookieOptions { HttpOnly = false });</pre>	

Assuming the script requests to send the token in a header called X-CSRF-TOKEN, configure the antiforgery service to look for the X-CSRF-TOKEN header:

C#	 Copy
<pre>services.AddAntiforgery(options => options.HeaderName = "X-CSRF-TOKEN");</pre>	

The following example uses JavaScript to make an AJAX request with the appropriate header:

JavaScript	 Copy
<pre>function getCookie(cname) { var name = cname + "="; var decodedCookie = decodeURIComponent(document.cookie); var ca = decodedCookie.split(';'); for(var i = 0; i <ca.length; i++) { var c = ca[i]; while (c.charAt(0) == ' ') { c = c.substring(1); } if (c.indexOf(name) == 0) { return c.substring(name.length, c.length); } } return " "; } var csrfToken = getCookie("CSRF-TOKEN"); var xhttp = new XMLHttpRequest();</pre>	

```

xhttp.onreadystatechange = function() {
    if (xhttp.readyState == XMLHttpRequest.DONE) {
        if (xhttp.status == 200) {
            alert(xhttp.responseText);
        } else {
            alert('There was an error processing the AJAX request.');

```

AngularJS

AngularJS uses a convention to address CSRF. If the server sends a cookie with the name `XSRF-TOKEN`, the AngularJS `$http` service adds the cookie value to a header when it sends a request to the server. This process is automatic. The header doesn't need to be set in the client explicitly. The header name is `X-XSRF-TOKEN`. The server should detect this header and validate its contents.

For ASP.NET Core API to work with this convention in your application startup:

- Configure your app to provide a token in a cookie called `XSRF-TOKEN`.
- Configure the antiforgery service to look for a header named `X-XSRF-TOKEN`.

C#

 Copy

```

public void Configure(IApplicationBuilder app, IAntiforgery
antiforgery)
{
    app.Use(next => context =>
    {
        string path = context.Request.Path.Value;

        if (
            string.Equals(path, "/",
StringComparison.OrdinalIgnoreCase) ||
            string.Equals(path, "/index.html",
StringComparison.OrdinalIgnoreCase))
        {
            // The request token can be sent as a JavaScript-readable
cookie,
            // and Angular uses it by default.

```

```

        var tokens = antiforgery.GetAndStoreTokens(context);
        context.Response.Cookies.Append("XSRF-TOKEN",
tokens.RequestToken,
            new CookieOptions() { HttpOnly = false });
    }

    return next(context);
});
}

public void ConfigureServices(IServiceCollection services)
{
    // Angular's default header name for sending the XSRF token.
    services.AddAntiforgery(options => options.HeaderName = "X-XSRF-
TOKEN");
}

```

[View or download sample code \(how to download\)](#)

Extend antiforgery

The [IAntiForgeryAdditionalDataProvider](#) type allows developers to extend the behavior of the anti-CSRF system by round-tripping additional data in each token. The [GetAdditionalData](#) method is called each time a field token is generated, and the return value is embedded within the generated token. An implementer could return a timestamp, a nonce, or any other value and then call [ValidateAdditionalData](#) to validate this data when the token is validated. The client's username is already embedded in the generated tokens, so there's no need to include this information. If a token includes supplemental data but no [IAntiForgeryAdditionalDataProvider](#) is configured, the supplemental data isn't validated.

Additional resources

- [CSRF](#) on [Open Web Application Security Project](#) (OWASP).
- [Host ASP.NET Core in a web farm](#)

Is this page helpful?

 Yes  No
