

Error - CSRF

Last updated by | Vitor Tomaz | Aug 5, 2020 at 12:40 PM PDT

Contents

- [Error - CSRF](#)
 - [Issue](#)
 - [Cause](#)
 - [How to Prevent](#)
 - [Public Doc Reference](#)
 - [Classification](#)

Error - CSRF

Issue

Error Connecting to SQL DB From Query Editor /Web Browser.

'The X-CSRF-Signature header could not be validated.'

Cause

This is created and validated to prevent a certain type of attack against your Azure SQL Servers. Specifically, some web browsers can save your passwords which might then allow an attacker who doesn't know the password to issue queries using the remembered password. In order to prevent this type of attack, known as Cross Site Request Forgery (CSRF), we attach this little bit of extra data, called the "CSRF Signature". This signature proves that the credentials were known at the time of the request, not just remembered by the browser.

How to Prevent

Cross-Site Request Forgery (XSRF/CSRF) attacks

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

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



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](https://docs.microsoft.com/en-us/aspnet/core/middleware/session).

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.

Public Doc Reference

<https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-3.1>

Classification

Root Cause: Azure SQL DB v2\Connectivity>Login errors\Others

How good have you found this content?

