Web Security

1. [What’s OWASP?](#_OWASP)
2. [What is “Vulnerability”?](#_Vulnerability)
3. [What is “SQL Injection”?](#_SQL_Injection)
4. What is cross-site scripting (XSS)?
5. Cross-Site Request [Forgery](#_Cross-Site_Request_Forgery) (XSRF/CSRF) attacks in ASP.NET Core
6. What is an open redirect attack?

# OWASP

<https://cheatsheetseries.owasp.org/cheatsheets/DotNet_Security_Cheat_Sheet.html>

The **Open Web Application Security Project®** (OWASP) is a nonprofit foundation that works to improve the security of software. Through community-led open-source software projects, hundreds of local chapters worldwide, tens of thousands of members, and leading educational and training conferences, the OWASP Foundation is the source for developers and technologists to secure the web.

## OWASP Top 10 of 2021

* The Injection
* The Broken Authentication
* The Sensitive Data Exposure
* The XML External Entities (XXE)
* The Broken Access Control
* The Security Misconfigurations
* The Cross-Site Scripting (XSS)
* The Insecure Deserialization
* The Using Components with Known Vulnerabilities
* The Insufficient Logging and Monitoring

# Vulnerability

Vulnerability can be defined as the weakness of any system through which intruders or bugs can attack the system.  
If security testing has not been performed rigorously on the system then chances of vulnerabilities get increased. Time to time patches or fixes is required to prevent a system from the vulnerabilities.

# SQL Injection

SQL Injection is one of the common attacking techniques used by hackers to get critical data.

Hackers check for any loophole in the system through which they can pass SQL queries, bypass the security checks, and return back the critical data. This is known as SQL injection. It can allow hackers to steal critical data or even crash a system.

SQL injections are very critical and need to be avoided. Periodic security testing can prevent this kind of attack. SQL database security needs to be defined correctly and input boxes and special characters should be handled properly.

**Solutions:**

* Use stored procedure (SP)
* Re-validate data in stored procedures.
* Use parameterized query
* Use ORM tools (NHybernet, EF)
* Use regular expression to discard input string

# Cross Site Scripting (XSS)

<https://docs.microsoft.com/en-us/aspnet/core/security/cross-site-scripting?view=aspnetcore-3.1>

<https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html>

Cross-Site Scripting (XSS) is a security vulnerability which enables an attacker to place client side scripts (usually JavaScript) into web pages. When other users load affected pages the attacker's scripts will run, enabling the attacker to steal cookies and session tokens, change the contents of the web page through DOM manipulation or redirect the browser to another page. XSS vulnerabilities generally occur when an application takes user input and outputs it to a page without validating, encoding or escaping it.

## **Protecting your application against XSS**

At a basic level XSS works by tricking your application into inserting a <script> tag into your rendered page, or by inserting an On\* event into an element. Developers should use the following prevention steps to avoid introducing XSS into their application.

1. Never put untrusted data into your HTML input, unless you follow the rest of the steps below. Untrusted data is any data that may be controlled by an attacker, HTML form inputs, query strings, HTTP headers, even data sourced from a database as an attacker may be able to breach your database even if they cannot breach your application.
2. Before putting untrusted data inside an HTML element ensure it's HTML encoded. HTML encoding takes characters such as < and changes them into a safe form like &lt;
3. Before putting untrusted data into an HTML attribute ensure it's HTML encoded. HTML attribute encoding is a superset of HTML encoding and encodes additional characters such as " and '.
4. Before putting untrusted data into JavaScript place the data in an HTML element whose contents you retrieve at runtime. If this isn't possible, then ensure the data is JavaScript encoded. JavaScript encoding takes dangerous characters for JavaScript and replaces them with their hex, for example < would be encoded as \u003C.
5. Before putting untrusted data into a URL query string ensure it's URL encoded.

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

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

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:

HTMLCopy

<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-site" part of CSRF.

1. 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.
2. 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.

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.

**GET requests that change state are insecure. A best practice is to never change state on a GET request.**

# ASP.NET Core antiforgery configuration

Antiforgery middleware is added to the [Dependency injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1) container when one of the following APIs is called in Startup.ConfigureServices:

* [AddMvc](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.mvcservicecollectionextensions.addmvc)
* [MapRazorPages](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.razorpagesendpointroutebuilderextensions.maprazorpages)
* [MapControllerRoute](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.controllerendpointroutebuilderextensions.mapcontrollerroute)
* [MapBlazorHub](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.componentendpointroutebuilderextensions.mapblazorhub)

In ASP.NET Core 2.0 or later, the [FormTagHelper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-3.1#the-form-tag-helper) injects antiforgery tokens into HTML form elements. The following markup in a Razor file automatically generates antiforgery tokens:

CSHTMLCopy:

<form method="post">

...

</form>

Similarly, [IHtmlHelper.BeginForm](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.rendering.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" attributes 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:

## CSHTMLCopy:

<form method="post" asp-antiforgery="false">

...

</form>

# What is an open redirect attack?

<https://docs.microsoft.com/en-us/aspnet/core/security/preventing-open-redirects?view=aspnetcore-3.1>

Web applications frequently redirect users to a login page when they access resources that require authentication. The redirection typically includes a returnUrl querystring parameter so that the user can be returned to the originally requested URL after they have successfully logged in. After the user authenticates, they're redirected to the URL they had originally requested.

Because the destination URL is specified in the querystring of the request, a malicious user could tamper with the querystring. A tampered querystring could allow the site to redirect the user to an external, malicious site. This technique is called an open redirect (or redirection) attack.

# **An example attack**

A malicious user can develop an attack intended to allow the malicious user access to a user's credentials or sensitive information. To begin the attack, the malicious user convinces the user to click a link to your site's login page with a returnUrl querystring value added to the URL. For example, consider an app at contoso.com that includes a login page at http://contoso.com/Account/LogOn?returnUrl=/Home/About. The attack follows these steps:

1. The user clicks a malicious link to http://contoso.com/Account/LogOn?returnUrl=http://contoso1.com/Account/LogOn (the second URL is "contoso**1**.com", not "contoso.com").
2. The user logs in successfully.
3. The user is redirected (by the site) to http://contoso1.com/Account/LogOn (a malicious site that looks exactly like real site).
4. The user logs in again (giving malicious site their credentials) and is redirected back to the real site.

The user likely believes that their first attempt to log in failed and that their second attempt is successful. The user most likely remains unaware that their credentials are compromised.