# COMMON VULNERABILITIES: CROSS-SITE SCRIPTING ATTACKS, SQL INJECTION ATTACKS, CROSS-SITE REQUEST FORGERY (CSRF), OPEN REDIRECT ATTACKS

## Cross-site Scripting Attacks

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 is HTML encoded. HTML encoding takes characters such as < and changes them into a safe form like <
3. Before putting untrusted data into an HTML attribute ensure it is 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 is URL encoded.

### HTML Encoding using Razor

The Razor engine used in MVC automatically encodes all output sourced from variables, unless you work really hard to prevent it doing so. It uses HTML attribute encoding rules whenever you use the @ directive. As HTML attribute encoding is a superset of HTML encoding this means you don't have to concern yourself with whether you should use HTML encoding or HTML attribute encoding. You must ensure that you only use @ in an HTML context, not when attempting to insert untrusted input directly into JavaScript. Tag helpers will also encode input you use in tag parameters.

Take the following Razor view:

**CS HTML**

```
@{
        var untrustedInput = "<\"123\">";
```

```
}
@untrustedInput
```

This view outputs the contents of the *untrustedInput* variable. This variable includes some characters which are used in XSS attacks, namely <, " and >. Examining the source shows the rendered output encoded as:

**HTML**

```
&lt;&quot;123&quot;&gt;
```

### Warning

ASP.NET Core MVC provides an HtmlString class which isn't automatically encoded upon output. This should never be used in combination with untrusted input as this will expose an XSS vulnerability.

### JavaScript Encoding using Razor

There may be times you want to insert a value into JavaScript to process in your view. There are two ways to do this. The safest way to insert values is to place the value in a data attribute of a tag and retrieve it in your JavaScript. For example:

**CSHTML**

```
@{
        var untrustedInput = "<\"123\">";
}

<div id="injectedData" data-untrustedinput="@untrustedInput" />
<script>
     var injectedData = document.getElementById("injectedData");
     // Allclients
       var clientSideUntrustedInputOldStyle =
                          injectedData.getAttribute("data-untrustedinput");
       // HTML5clientsonly
       var clientSideUntrustedInputHtml5 =
                          injectedData.dataset.untrustedinput;
       document.write(clientSideUntrustedInputOldStyle);
       document.write("<br />")
       document.write(clientSideUntrustedInputHtml5);
</script>
```

This will produce the following HTML

**HTML**

```
<div id="injectedData" data-untrustedinput="&lt;&quot;123&quot;&gt;" />
<script>
       var injectedData = document.getElementById("injectedData");
       var clientSideUntrustedInputOldStyle =
                 injectedData.getAttribute("data-untrustedinput");
```

```
        var clientSideUntrustedInputHtml5 =
                  injectedData.dataset.untrustedinput;
        document.write(clientSideUntrustedInputOldStyle);
        document.write("<br />");
        document.write(clientSideUntrustedInputHtml5);
</script>
```

Which, when it runs, will render the following:

```
<"123">
<"123">
```

You can also call the JavaScript encoder directly:

**CSHTML**

```
@using System.Text.Encodings.Web;
@inject JavaScriptEncoder encoder;

@{
        var untrustedInput = "<\"123\">";
}
<script>
        document.write("@encoder.Encode(untrustedInput)");
</script>
```

This will render in the browser as follows:

**HTML**

```
<script>
        document.write("\u003C\u0022123\u0022\u003E");
</script>
```

### Warning

Don't concatenate untrusted input in JavaScript to create DOM elements. You should use createElement() and assign property values appropriately such as node.TextContent=, or use element.SetAttribute()/element[attribute]= otherwise you expose yourself to DOM-based XSS.

### Accessing encoders in code

The HTML, JavaScript and URL encoders are available to your code in two ways, you can inject them via dependency injection or you can use the default encoders contained in the System.Text.Encodings.Web namespace. If you use the default encoders then any you applied to character ranges to be treated as safe won't take effect - the default encoders use the safest encoding rules possible.

To use the configurable encoders via DI your constructors should take an *HtmlEncoder*, *JavaScriptEncoder* and *UrlEncoder* parameter as appropriate. For example;

**C#**

```
public class HomeController :Controller {
       HtmlEncoder _htmlEncoder;
       JavaScriptEncoder _javaScriptEncoder;
       UrlEncoder _urlEncoder;
       public HomeController(HtmlEncoderhtmlEncoder,
                 JavaScriptEncoderjavascriptEncoder, UrlEncoderurlEncoder)
       {
           _htmlEncoder = htmlEncoder;
           _javaScriptEncoder = javascriptEncoder;
           _urlEncoder = urlEncoder;
       }
   }
```

### Encoding URL Parameters

If you want to build a URL query string with untrusted input as a value use the UrlEncoder to encode the value. For example,

C#

```
var example = "\"Quoted Value with spaces and &\"";
var encodedValue = _urlEncoder.Encode(example);
```

After encoding the encodedValue variable will contain %22Quoted%20Value%20with%20spaces%20and%20%26%22. Spaces, quotes, punctuation and other unsafe characters will be percent encoded to their hexadecimal value, for example a space character will become %20.

**Warning**

Don't use untrusted input as part of a URL path. Always pass untrusted input as a query string value.

### Customizing the Encoders

By default encoders use a safe list limited to the Basic Latin Unicode range and encode all characters outside of that range as their character code equivalents. This behavior also affects Razor TagHelper and HtmlHelper rendering as it will use the encoders to output your strings.

The reasoning behind this is to protect against unknown or future browser bugs (previous browser bugs have tripped up parsing based on the processing of non-English characters). If your web site makes heavy use of non-Latin characters, such as Chinese, Cyrillic or others this is probably not the behavior you want.

You can customize the encoder safe lists to include Unicode ranges appropriate to your application during startup, in ConfigureServices().

For example, using the default configuration you might use a Razor HtmlHelper like so;

**HTML**

```
<p>This link text is in Chinese: @Html.ActionLink("汉语/漢語", "Index")</p>
```

When you view the source of the web page you will see it has been rendered as follows, with the Chinese text encoded;

**HTML**

```
<p>This link text is in Chinese: <ahref="/">&#x6C49;&#x8BED;/&#x6F22;&#x8A9E;</a></p>
```

To widen the characters treated as safe by the encoder you would insert the following line into the `ConfigureServices()` method in `startup.cs`;

**C#**

```
services.AddSingleton<HtmlEncoder>(
HtmlEncoder.Create(allowedRanges: new[] { UnicodeRanges.BasicLatin,
UnicodeRanges.CjkUnifiedIdeographs }));
```

This example widens the safe list to include the Unicode Range CjkUnifiedIdeographs. The rendered output would now become

**HTML**

```
<p>This link text is in Chinese: <a href="/">汉语/漢語</a></p>
```

Safe list ranges are specified as Unicode code charts, not languages. The Unicode standard has a list of code charts you can use to find the chart containing your characters. Each encoder, Html, JavaScript and Url, must be configured separately.

**Note**

Customization of the safe list only affects encoders sourced via DI. If you directly access an encoder via System.Text.Encodings.Web.*Encoder.Default then the default, Basic Latin only safelist will be used.

**Where should encoding take place?**

The general accepted practice is that encoding takes place at the point of output and encoded values should never be stored in a database. Encoding at the point of output allows you to change the use of data, for example, from HTML to a query string value. It also enables you to easily search your data without having to encode values before searching and allows you to take advantage of any changes or bug fixes made to encoders.

### Validation as an XSS prevention technique

Validation can be a useful tool in limiting XSS attacks. For example, a numeric string containing only the characters 0-9 won't trigger an XSS attack. Validation becomes more complicated when accepting HTML in user input. Parsing HTML input is difficult, if not impossible. Markdown, coupled with a parser that strips embedded HTML, is a saferoption for accepting rich input.Never rely on validation alone. Always encode untrusted input before output, no matter what validation or sanitization has been performed.

## SQL Injection Attacks

SQL injection, also known as SQLI, is a common attack vector that uses malicious SQL code for backend database manipulation to access information that was not intended to be displayed. This information may include any number of items, including sensitive company data, user lists or private customer details.

The impact SQL injection can have on a business is far-reaching. A successful attack may result in the unauthorized viewing of user lists, the deletion of entire tables and, in certain cases, the attacker gaining administrative rights to a database, all of which are highly detrimental to a business.

When calculating the potential cost of an SQLi, it's important to consider the loss of customer trust should personal information such as phone numbers, addresses, and credit card details be stolen.

While this vector can be used to attack any SQL database, websites are the most frequent targets.

**What are SQL queries**

SQL is a standardized language used to access and manipulate databases to build customizable data views for each user. SQL queries are used to execute commands, such as data retrieval, updates, and record removal. Different SQL elements implement these tasks, e.g., queries using the SELECT statement to retrieve data, based on user-provided parameters.

**SQL injection examples**

There are a wide variety of SQL injection vulnerabilities, attacks, and techniques, which arise in different situations. Some common SQL injection examples include:

- Retrieving hidden data, where you can modify an SQL query to return additional results.
- Subverting application logic, where you can change a query to interfere with the application's logic.
- UNION attacks, where you can retrieve data from different database tables.
- Examining the database, where you can extract information about the version and structure of the database.
- Blind SQL injection, where the results of a query you control are not returned in the application's responses.

**Retrieving hidden data**

Consider a shopping application that displays products in different categories. When the user clicks on the Gifts category, their browser requests the URL:

```
https://insecure-website.com/products?category=Gifts
```

This causes the application to make an SQL query to retrieve details of the relevant products from the database:

```
SELECT * FROM products WHERE category = 'Gifts' AND released = 1
```

This SQL query asks the database to return:

- all details (*)
- from the products table
- where the category is Gifts
- and released is 1.

The restriction `released = 1` is being used to hide products that are not released. For unreleased products, presumably `released = 0`.

The application doesn't implement any defenses against SQL injection attacks, so an attacker can construct an attack like:

```
https://insecure-website.com/products?category=Gifts'--
```

This results in the SQL query:

```
SELECT * FROM products WHERE category = 'Gifts'--' AND released = 1
```

The key thing here is that the double-dash sequence `--` is a comment indicator in SQL, and means that the rest of the query is interpreted as a comment. This effectively removes the remainder of the query, so it no longer includes `AND released = 1`. This means that all products are displayed, including unreleased products.

Going further, an attacker can cause the application to display all the products in any category, including categories that they don't know about:

```
https://insecure-website.com/products?category=Gifts'+OR+1=1--
```

This results in the SQL query:

```
SELECT * FROM products WHERE category ='Gifts' OR 1=1--' AND released = 1
```

The modified query will return all items where either the category is Gifts, or 1 is equal to 1. Since `1=1` is always true, the query will return all items.

### Subverting application logic

Consider an application that lets users log in with a username and password. If a user submits the username `wiener` and the password `bluecheese`, the application checks the credentials by performing the following SQL query:

```
SELECT * FROM users WHERE username ='wiener' AND password = 'bluecheese'
```

If the query returns the details of a user, then the login is successful. Otherwise, it is rejected.

Here, an attacker can log in as any user without a password simply by using the SQL comment sequence `--` to remove the password check from the `WHERE` clause of the query. For example, submitting the username `administrator'--` and a blank password results in the following query:

```
SELECT * FROM users WHERE username ='administrator'--' AND password =''
```

This query returns the user whose username is `administrator` and successfully logs the attacker in as that user.

## Retrieving data from other database tables

In cases where the results of an SQL query are returned within the application's responses, an attacker can leverage an SQL injection vulnerability to retrieve data from other tables within the database. This is done using the `UNION` keyword, which lets you execute an additional `SELECT` query and append the results to the original query.

For example, if an application executes the following query containing the user input "Gifts":

```
SELECT name, description FROM products WHERE category = 'Gifts'
```

then an attacker can submit the input:

```
' UNION SELECT username, password FROM users--
```

This will cause the application to return all usernames and passwords along with the names and descriptions of products.

**Examining the database**

Following initial identification of an SQL injection vulnerability, it is generally useful to obtain some information about the database itself. This information can often pave the way for further exploitation.

You can query the version details for the database. The way that this is done depends on the database type, so you can infer the database type from whichever technique works. For example, on Oracle you can execute:

```
SELECT * FROM v$version
```

You can also determine what database tables exist, and which columns they contain. For example, on most databases you can execute the following query to list the tables:

```
SELECT * FROM information_schema.tables
```

**Blind SQL injection vulnerabilities**

Many instances of SQL injection are blind vulnerabilities. This means that the application does not return the results of the SQL query or the details of any database errors within its responses. Blind vulnerabilities can still be exploited to access unauthorized data, but the techniques involved are generally more complicated and difficult to perform.

Depending on the nature of the vulnerability and the database involved, the following techniques can be used to exploit blind SQL injection vulnerabilities:

- You can change the logic of the query to trigger a detectable difference in the application's response depending on the truth of a single condition. This might involve injecting a new condition into some Boolean logic, or conditionally triggering an error such as a divide-by-zero.
- You can conditionally trigger a time delay in the processing of the query, allowing you to infer the truth of the condition based on the time that the application takes to respond.
- You can trigger an out-of-band network interaction, using OAST techniques. This technique is extremely powerful and works in situations where the other techniques do not. Often, you can directly exfiltrate data via the out-of-band channel, for example by placing the data into a DNS lookup for a domain that you control.

**How to detect SQL injection vulnerabilities**

The majority of SQL injection vulnerabilities can be found quickly and reliably using Burp Suite's web vulnerability scanner.

SQL injection can be detected manually by using a systematic set of tests against every entry point in the application. This typically involves:

- Submitting the single quote character ' and looking for errors or other anomalies.
- Submitting some SQL-specific syntax that evaluates to the base (original) value of the entry point, and to a different value, and looking for systematic differences in the resulting application responses.
- Submitting Boolean conditions such as OR 1=1 and OR 1=2, and looking for differences in the application's responses.

- Submitting payloads designed to trigger time delays when executed within an SQL query, and looking for differences in the time taken to respond.
- Submitting OAST payloads designed to trigger an out-of-band network interaction when executed within an SQL query, and monitoring for any resulting interactions.

**SQL injection in different parts of the query**

Most SQL injection vulnerabilities arise within the `WHERE` clause of a `SELECT` query. This type of SQL injection is generally well-understood by experienced testers.

But SQL injection vulnerabilities can in principle occur at any location within the query, and within different query types. The most common other locations where SQL injection arises are:

- In UPDATE statements, within the updated values or the WHERE clause.
- In INSERT statements, within the inserted values.
- In SELECT statements, within the table or column name.
- In SELECT statements, within the ORDER BY clause.

**Second-order SQL injection**

First-order SQL injection arises where the application takes user input from an HTTP request and, in the course of processing that request, incorporates the input into an SQL query in an unsafe way.

In second-order SQL injection (also known as stored SQL injection), the application takes user input from an HTTP request and stores it for future use. This is usually done by placing the input into a database, but no vulnerability arises at the point where the data is stored. Later, when handling a different HTTP request, the application retrieves the stored data and incorporates it into an SQL query in an unsafe way.

Second-order SQL injection often arises in situations where developers are aware of SQL injection vulnerabilities, and so safely handle the initial placement of the input into the database. When the data is later processed, it is deemed to be safe, since it was previously placed into the database safely. At this point, the data is handled in an unsafe way, because the developer wrongly deems it to be trusted.

**Database-specific factors**

Some core features of the SQL language are implemented in the same way across popular database platforms, and so many ways of detecting and exploiting SQL injection vulnerabilities work identically on different types of database.

However, there are also many differences between common databases. These mean that some techniques for detecting and exploiting SQL injection work differently on different platforms. For example:

- Syntax for string concatenation.
- Comments.
- Batched (or stacked) queries.
- Platform-specific APIs.
- Error messages.

**How to prevent SQL injection**

Most instances of SQL injection can be prevented by using parameterized queries (also known as prepared statements) instead of string concatenation within the query.

The following code is vulnerable to SQL injection because the user input is concatenated directly into the query:

```
String query = "SELECT * FROM products WHERE category = '"+ input + "'";
Statement statement = connection.createStatement();
ResultSetresultSet = statement.executeQuery(query);
```

This code can be easily rewritten in a way that prevents the user input from interfering with the query structure:

```
PreparedStatement statement = connection.prepareStatement("SELECT * FROM products WHERE category = ?");
statement.setString(1, input);
ResultSetresultSet = statement.executeQuery();
```

Parameterized queries can be used for any situation where untrusted input appears as data within the query, including the `WHERE` clause and values in an `INSERT` or `UPDATE` statement. They can't be used to handle untrusted input in other parts of the query, such as table or column names, or the `ORDER BY` clause. Application functionality that places untrusted data into those parts of the query will need to take a different approach, such as white-listing permitted input values, or using different logic to deliver the required behavior.

For a parameterized query to be effective in preventing SQL injection, the string that is used in the query must always be a hard-coded constant, and must never contain any variable data from any origin. Do not be tempted to decide case-by-case whether an item of data is trusted, and continue using string concatenation within the query for cases that are considered safe. It is all too easy to make mistakes about the possible origin of data, or for changes in other code to violate assumptions about what data is tainted.

## 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-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

**CSHTML**

```
<form method="post">
    ...
</form>
```

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:

  **CSHTML**

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

```
</form>
```

- The form element is opted-out of Tag Helpers by using the Tag Helper ! opt-out symbol:

  **CSHTML**

```
<!form method="post">
 ...
</!form>
```

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

  **CSHTML**

```
@removeTagHelper Microsoft.AspNetCore.Mvc.TagHelpers.FormTagHelper,
Microsoft.AspNetCore.Mvc.TagHelpers
```

**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**

```
<form asp-controller="Manage" asp-action="ChangePassword" method="post">
    ...
</form>


@using (Html.BeginForm("ChangePassword", "Manage"))
{
    ...
}
```

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

**CSHTML**

```
<form action="/"method="post">
@Html.AntiForgeryToken()
</form>
```

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

**CSHTML**

```
<input name="__RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ...
s2-m9Yw">
```

ASP.NET Core includes three filters for working with antiforgery tokens:

- ValidateAntiForgeryToken
- AutoValidateAntiforgeryToken
- IgnoreAntiforgeryToken

Antiforgery options

Customize antiforgery options in `Startup.ConfigureServices`:

**C#**

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

**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#**

```
[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 });
}
```

## Open Redirect Attack

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.

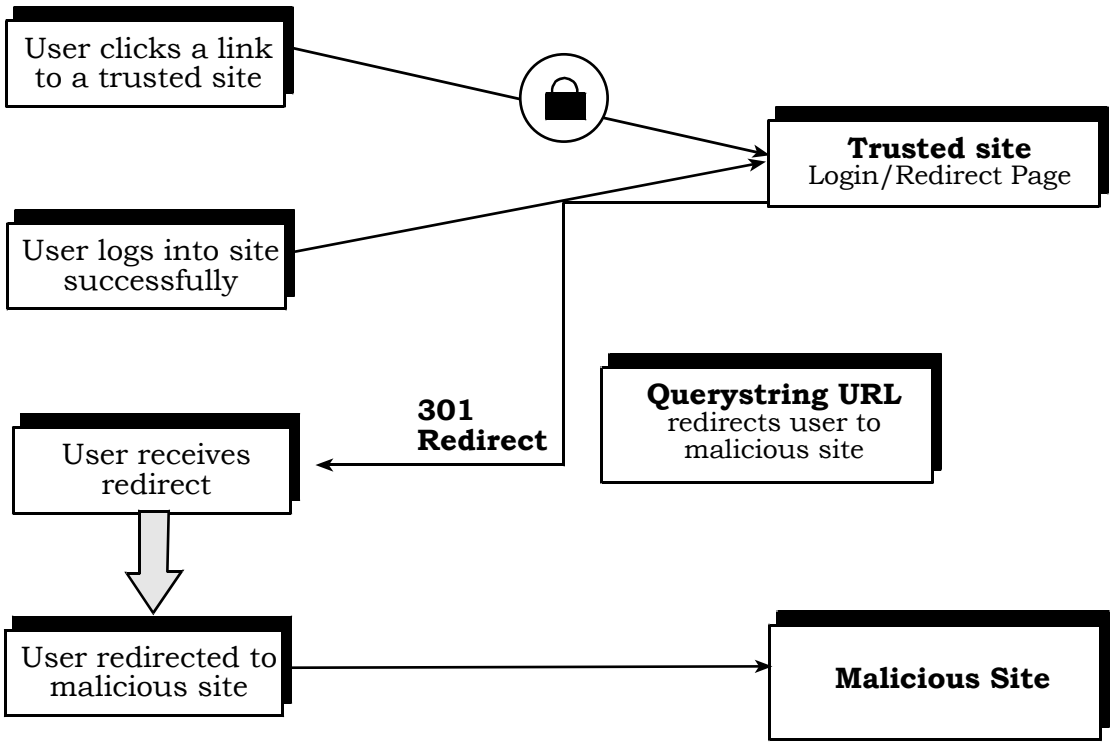


**Figure 1: Open Redirect Attack Process**

In addition to login pages, some sites provide redirect pages or endpoints. Imagine your app has a page with an open redirect, /Home/Redirect. An attacker could create, for example, a link in an email that goes to [yoursite]/Home/Redirect?url=http://phishingsite.com/Home/Login. A typical user will look at the URL and see it begins with your site name. Trusting that, they will click the link. The open redirect would then send the user to the phishing site, which looks identical to yours, and the user would likely login to what they believe is your site.

**Protecting against open redirect attacks**

When developing web applications, treat all user-provided data as untrustworthy. If your application has functionality that redirects the user based on the contents of the URL, ensure that such redirects are only done locally within your app (or to a known URL, not any URL that may be supplied in the querystring).

LocalRedirect

Use the LocalRedirect helper method from the base Controller class:

**C#**

```
public IActionResult SomeAction(stringredirectUrl)
{
        return LocalRedirect(redirectUrl);
}
```

LocalRedirect will throw an exception if a non-local URL is specified. Otherwise, it behaves just like the Redirect method.

IsLocalUrl

Use the IsLocalUrl method to test URLs before redirecting:

The following example shows how to check whether a URL is local before redirecting.

**C#**

```
private IActionResult RedirectToLocal(stringreturnUrl)
{
        if (Url.IsLocalUrl(returnUrl))
        {
           return Redirect(returnUrl);
        }
        else
        {
           return RedirectToAction(nameof(HomeController.Index), "Home");
        }
}
```

The IsLocalUrl method protects users from being inadvertently redirected to a malicious site. You can log the details of the URL that was provided when a non-local URL is supplied in a situation where you expected a local URL. Logging redirect URLs may help in diagnosing redirection attacks.

## Discussion Exercise

1. Why Authentication and Authorization is important in your Application?
2. Write about ASP.NET Core Identity. How do you add authentication to your apps?
3. What are the ways to authorize your ASP.NET Core Application?
4. How can you apply Roles, Claims and Policies for authorization features?
5. What are the common vulnerabilities that can be found in the software application?
6. Show how you can protect your application against Cross-site Scripting attacks.
7. What is SQL Injection Attack and how can we prevent such attacks?
8. Explain you Cross-Site Request Forgery(XSRF/CSRF) attacks. How can you do antiforgery validation?
9. Explain Open redirect attack with example.