Graphical user interface

Description automatically generated

**Description**

Cross-Site Scripting (otherwise known as XSS) is a vulnerability that allows a malicious actor to manipulate a legitimate user's interactions with a vulnerable web application. Attackers exploit this to bypass the same origin policy, often allowing them to perform any actions that the target user would normally perform, including gaining access to their data. In cases where the victim user has privileged application access, the attacker may use XSS to gain control of the application.

XSS attacks typically occur in web applications when data is received, frequently in the form of a web request, and the data is reflected back in the HTTP response to the user without validation.

XSS attacks can generally be divided into the following three categories.

**Reflected XSS**

Reflected XSS attacks arise when a web server reflects injected script, such as a search result, an error message, or any other response that includes some or all of the input sent to the server as part of the request.

The attack is then delivered to the victim through another route (e.g. e-mail or alternative website), thus tricking the user into clicking on a malicious link. The injected code travels to the vulnerable website, which reflects the attack payload back to the user's browser. The browser then executes the code because it came from a "trusted" server.

**Stored XSS**

In the Stored XSS attack, the injected script is stored on the target application as legitimate content, such as a message in a forum, a comment in a blog post, etc. The injected code is stored in the database and sent to the users when it is retrieved by accessing the injected content, executing the attack payload in the victim's browser.

**Dom-based XSS**

DOM-based XSS vulnerabilities usually occur when the JavaScript in a page takes user-provided data from a source in the HTML, such as the document.location, and passes it to a JavaScript function that allows JavaScript code to be run, such as innerHTML(). The classic attack delivers the payload to the victim through another route (e.g. e-mail or alternative website) and thus tricks the user into visiting a malicious link. The exploitation is client-side, and the code is immediately executed in the user's browser.

**Impact**

XSS attacks can result in the disclosure of the user's session cookie, allowing an attacker to hijack the user's session and take over the account. Even though HTTPOnly is used to protect cookies, an attacker can still execute actions on behalf of the user in the context of the affected website.

As with all of the severe vulnerabilities that make up a part of the [OWASP Top 10](https://owasp.org/www-project-top-ten/), XSS attacks can result in full compromise of a user's system. As stated in the description, if an attacker compromises a user holding the 'keys to the kingdom' i.e. privileged access to applications / administrator rights, the results can be devastating.

**Prevention**

XSS attacks can be mitigated by performing appropriate server-side validation and escaping.  
Remediation relies on performing Output Encoding (e.g. using an escape syntax) for the type of HTML context where untrusted data is reflected into.

**Input Validation**

* Exact Match: Only accept values from a finite list of known values.
* Allow list: If a list of all the possible values can't be created, accept only known good data and reject all unexpected input.
* Deny list: If an allow-list approach is not feasible (on free form text areas, for example) reject all known bad values.

**Output Encoding**

Output Encoding is used to convert untrusted input into a safe form where the input is displayed as **data** to the user without executing as **code** in the browser. Output Encoding is performed when the data leaves the application to a downstream component. The table below lists the possible downstream contexts where the untrusted input could be used:

| **Context** | **Code** | **Encoding** |
| --- | --- | --- |
| HTML Body | <div>USER-CONTROLLED-DATA</div> | HTML Encoding |
| HTML Attribute | <input type="text" value="USER-CONTROLLED-DATA"> | HTML Attribute Encoding |
| URL Parameter | <a href="/search?value=USER-CONTROLLED-DATA">Search</a> | URL Encoding |
| CSS | <div style="width: USER-CONTROLLED-DATA;">Selection</div> | CSS Hex Encoding |
| JavaScript | <script>var lang ='USER-CONTROLLED-DATA';</script> <script>setLanguage('USER-CONTROLLED-DATA');</script> | JavaScript Encoding |

The following chart details a list of critical output encoding methods required to mitigate Cross Site Scripting:

| Encoding Type | Encoding Mechanism |
| --- | --- |
| HTML Entity Encoding | Convert &to &amp; Convert <to &lt; Convert >to &gt; Convert "to &quot; Convert 'to &#x27; Convert /to &#x2F; |
| HTML Attribute Encoding | Except for alphanumeric characters, escape all characters with the HTML Entity &#xHH; format including spaces. (**HH** = Hex Value) |
| URL Encoding | For standard percent encoding see [here](http://www.w3schools.com/tags/ref_urlencode.asp). URL encoding should only be used to encode parameter values, not the entire URL or path fragments of a URL. |
| JavaScript Encoding | Except for alphanumeric characters, escape all characters with the \uXXXX unicode escaping format (**XX** = Integer) |
| CSS Hex Encoding | CSS escaping supports \XX and \XXXXXX. Using a two-character escape can cause problems if the next character continues the escape sequence. There are two solutions: - Add a space after the CSS escape (it will be ignored by the CSS parser) - Use the full amount of CSS escaping possible by zero padding the value. |

**Defense in Depth**

**Content Security Policy (CSP)**

The Content Security Policy (CSP) is a browser mechanism that enables the creation of source allow lists for client-side resources of web applications e.g. JavaScript, CSS, images etc. CSP via a special HTTP header instructs the browser to only execute or render resources from those sources.

For example:

Content-Security-Policy: default-src: 'self'; script-src: 'self' static.domain.tld

The above CSP will instruct the web browser to load all resources only from the page's origin, and JavaScript source code files additionally from static.domain.tld. For more details on Content Security Policy, including what it does and how to use it, see [this](https://content-security-policy.com/) article.

**X-XSS-Protection Header**

This HTTP response header enables the Cross-Site Scripting (XSS) filter built into some modern web browsers. The header is usually enabled by default anyway, so its role is to re-enable the filter for a particular website if it was disabled by the user.

**Use a Modern Framework with an Auto-Escaping Template System**

Modern JavaScript frameworks (e.g. AngularJS, ReactJS) or server-side templating systems (e.g. Go Templates) have robust built-in protections against Reflected Cross-Site Scripting.

**Testing**

Verify that context-aware, preferably automated - or at worst, manual - output escaping protects against reflected, stored, and DOM based XSS.

* **OWASP ASVS**: [5.3.1, 5.3.3](https://github.com/OWASP/ASVS/releases/download/v4.0.2_release/OWASP.Application.Security.Verification.Standard.4.0.2-en.pdf)
* **OWASP Testing Guide**: [Testing for Reflected Cross Site Scripting](https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/01-Testing_for_Reflected_Cross_Site_Scripting.html), [Testing for Stored Cross Site Scripting](https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/02-Testing_for_Stored_Cross_Site_Scripting.html), [Testing for DOM-Based Cross Site Scripting](https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/11-Client-side_Testing/01-Testing_for_DOM-based_Cross_Site_Scripting.html)

**.NET**

**Prevention**

Apply the recommended security controls depending on the framework and template engine of choice.

**ASP.NET Core**

The Razor engine used in ASP.NET Core MVC automatically encodes all output sourced from variables for the HTML context. It uses HTML attribute encoding rules whenever you use the @ directive to render code on the page. You must ensure that you only use @ in an HTML context, not when attempting to insert untrusted input directly into JavaScript.

| **Context** | **Code** | ASP.NET Encoding mechanisms |
| --- | --- | --- |
| HTML Code and Attribute | <div>@user-controlled-variable</div> | [Automatically encoded](https://docs.microsoft.com/en-us/aspnet/core/security/cross-site-scripting?view=aspnetcore-2.2#html-encoding-using-razor) Encode data for use in HTML using HTML entity encoding |
| JavaScript | <script>var id = "@user\_controlled-variable";</script> | [encoder.Encode(string)](https://docs.microsoft.com/en-us/aspnet/core/security/cross-site-scripting?view=aspnetcore-2.2#javascript-encoding-using-razor) Encode data for insertion inside a data value or function argument in JavaScript. |

**References**

[Microsoft - Prevent Cross-Site Scripting (XSS) in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/security/cross-site-scripting)  
[OWASP - Cross-Site Scripting (XSS)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)#Stored_and_Reflected_XSS_Attacks)  
[OWASP - Code Review Guide](https://www.owasp.org/images/5/53/OWASP_Code_Review_Guide_v2.pdf)  
[OWASP - Cross-Site Scripting Prevention Cheat Sheet](https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.md)