Cross Site Scripting Prevention Cheat Sheet

Introduction

This cheat sheet helps developers prevent XSS vulnerabilities.

Cross-Site Scripting (XSS) is a misnomer. Originally this term was derived from early versions of the attack that were primarily focused on stealing data cross-site. Since then, the term has widened to include injection of basically any content. XSS attacks are serious and can lead to account impersonation, observing user behaviour, loading external content, stealing sensitive data, and more.

This cheatsheet contains techniques to prevent or limit the impact of XSS. Since no single technique will solve XSS, using the right combination of defensive techniques will be necessary to prevent XSS.

Framework Security

Fortunately, applications built with modern web frameworks have fewer XSS bugs, because these frameworks steer developers towards good security practices and help mitigate XSS by using templating, auto-escaping, and more. However, developers need to know that problems can occur if frameworks are used insecurely, such as:

- *escape hatches* that frameworks use to directly manipulate the DOM
- React's `dangerouslySetInnerHTML` without sanitising the HTML
- React cannot handle `javascript:` or `data:` URLs without specialized validation
- Angular's `bypassSecurityTrustAs*` functions
- Lit's `unsafeHTML` function
- Polymer's `inner-h-t-m-l` attribute and `htmlLiteral` function
- Template injection
- Out of date framework plugins or components
- and more

When you use a modern web framework, you need to know how your framework prevents XSS and where it has gaps. There will be times where you need to do something outside the protection provided by your framework, which means that Output Encoding and HTML Sanitization can be critical. OWASP will be producing framework specific cheatsheets for React, Vue, and Angular.

XSS Defense Philosophy

In order for an XSS attack to be successful, an attacker must be able to insert and execute malicious content in a webpage. Thus, all variables in a web application needs to be protected. Ensuring that **all variables** go through validation and are then escaped or sanitized is known as **perfect injection resistance**. Any variable that does not go through this process is a potential weakness. Frameworks make it easy to ensure variables are correctly validated and escaped or sanitised.

However, no framework is perfect and security gaps still exist in popular frameworks like React and Angular. Output encoding and HTML sanitization help address those gaps.

Output Encoding

When you need to safely display data exactly as a user types it in, output encoding is recommended. Variables should not be interpreted as code instead of text. This section covers each form of output encoding, where to use it, and when you should not use dynamic variables at all.

First, when you wish to display data as the user typed it in, start with your framework's default output encoding protection. Automatic encoding and escaping functions are built into most frameworks.

If you're not using a framework or need to cover gaps in the framework then you should use an output encoding library. Each variable used in the user interface should be passed through an output encoding function. A list of output encoding libraries is included in the appendix.

There are many different output encoding methods because browsers parse HTML, JS, URLs, and CSS differently. Using the wrong encoding method may introduce weaknesses or harm the functionality of your application.

Output Encoding for "HTML Contexts"

"HTML Context" refers to inserting a variable between two basic HTML tags like a `<div>` or `<b>`. For example:

```
<div> $varUnsafe </div>
```

An attacker could modify data that is rendered as `$varUnsafe`. This could lead to an attack being added to a webpage. For example:

```
<div> <script>alert`1`</script> </div> // Example Attack
```

In order to add a variable to a HTML context safely to a web template, use HTML entity encoding for that variable.

Here are some examples of encoded values for specific characters:

If you're using JavaScript for writing to HTML, look at the `.textContent` attribute. It is a **Safe Sink** and will automatically HTML Entity Encode.

```
&    &amp;
<    &lt;
>    &gt;
"    &quot;
'    &#x27;
```

Output Encoding for “HTML Attribute Contexts”

“HTML Attribute Contexts” occur when a variable is placed in an HTML attribute value. You may want to do this to change a hyperlink, hide an element, add alt-text for an image, or change inline CSS styles. You should apply HTML attribute encoding to variables being placed in most HTML attributes. A list of safe HTML attributes is provided in the **Safe Sinks** section.

```
<div attr="$varUnsafe">
<div attr=”*x” onblur=”alert(1)*”> // Example Attack
```

It's critical to use quotation marks like `"` or `'` to surround your variables. Quoting makes it difficult to change the context a variable operates in, which helps prevent XSS. Quoting also significantly reduces the characterset that you need to encode, making your application more reliable and the encoding easier to implement.

If you're writing to a HTML Attribute with JavaScript, look at the `.setAttribute` and `[attribute]` methods because they will automatically HTML Attribute Encode. Those are **Safe Sinks** as long as the attribute name is hardcoded and innocuous, like `id` or `class`. Generally, attributes that accept JavaScript, such as `onClick`, are **NOT safe** to use with untrusted attribute values.

Output Encoding for “JavaScript Contexts”

“JavaScript Contexts” refers to the situation where variables are placed into inline JavaScript and then embedded in an HTML document. This situation commonly occurs in programs that heavily use custom JavaScript that is embedded in their web pages.

However, the only ‘safe’ location for placing variables in JavaScript is inside a “quoted data value”. All other contexts are unsafe and you should not place variable data in them.

Examples of “Quoted Data Values”

```
<script>alert('$varUnsafe')</script>
<script>x='$varUnsafe'</script>
<div onmouseover="'$varUnsafe'"</div>
```

Encode all characters using the `\xHH` format. Encoding libraries often have a `EncodeForJavaScript` or similar to support this function.

Please look at the OWASP Java Encoder JavaScript encoding examples for examples of proper JavaScript use that requires minimal encoding.

For JSON, verify that the `Content-Type` header is `application/json` and not `text/html` to prevent XSS.

Output Encoding for “CSS Contexts”

“CSS Contexts” refer to variables placed into inline CSS, which is common when developers want their users to customize the look and feel of their webpages. Since CSS is surprisingly powerful, it has been used for many types of attacks. **Variables should only be placed in a CSS property value. Other “CSS Contexts” are unsafe and you should not place variable data in them.**

```
<style> selector { property : $varUnsafe; } </style>
<style> selector { property : "$varUnsafe"; } </style>
<span style="property : $varUnsafe">Oh no</span>
```

If you're using JavaScript to change a CSS property, look into using `style.property = x`. This is a **Safe Sink** and will automatically CSS encode data in it.

When inserting variables into CSS properties, ensure the data is properly encoded and sanitized to prevent injection attacks. Avoid placing variables directly into selectors or other CSS contexts.

Output Encoding for “URL Contexts”

“URL Contexts” refer to variables placed into a URL. Most commonly, a developer will add a parameter or URL fragment to a URL base that is then displayed or used in some operation. Use URL Encoding for these scenarios.

```
<a href="http://www.owasp.org?test=$varUnsafe">link</a >
```

Encode all characters with the `%HH` encoding format. Make sure any attributes are fully quoted, same as JS and CSS.

Common Mistake

There will be situations where you use a URL in different contexts. The most common one would be adding it to an `href` or `src` attribute of an `<a>` tag. In these scenarios, you should do URL encoding, followed by HTML attribute encoding.

```
url = "https://site.com?data=" + urlencode(parameter)
<a href='attributeEncode(url)'>link</a>
```

If you're using JavaScript to construct a URL Query Value, look into using `window.encodeURIComponent(x)`. This is a **Safe Sink** and will automatically URL encode data in it.

Dangerous Contexts

Output encoding is not perfect. It will not always prevent XSS. These locations are known as **dangerous contexts**. Dangerous contexts include:

```
<script>Directly in a script</script>
<!-- Inside an HTML comment -->
<style>Directly in CSS</style>
<div ToDefineAnAttribute=test />
<ToDefineATag href="/test" />
```

Other areas to be careful with include:

- Callback functions
- Where URLs are handled in code such as this CSS { background-url : “javascript:alert(xss)”; }
- All JavaScript event handlers (`onclick()` , `onerror()` , `onmouseover()`).
- Unsafe JS functions like `eval()` , `setInterval()` , `setTimeout()`

Don't place variables into dangerous contexts as even with output encoding, it will not prevent an XSS attack fully.

HTML Sanitization

When users need to author HTML, developers may let users change the styling or structure of content inside a WYSIWYG editor. Output encoding in this case will prevent XSS, but it will break the intended functionality of the application. The styling will not be rendered. In these cases, HTML Sanitization should be used.

HTML Sanitization will strip dangerous HTML from a variable and return a safe string of HTML. OWASP recommends DOMPurify for HTML Sanitization.

```
let clean = DOMPurify.sanitize(dirty);
```

There are some further things to consider:

- If you sanitize content and then modify it afterwards, you can easily void your security efforts.
- If you sanitize content and then send it to a library for use, check that it doesn't mutate that string somehow. Otherwise, again, your security efforts are void.
- You must regularly patch DOMPurify or other HTML Sanitization libraries that you use. Browsers change functionality and bypasses are being discovered regularly.

Safe Sinks

Security professionals often talk in terms of sources and sinks. If you pollute a river, it'll flow downstream somewhere. It's the same with computer security. XSS sinks are places where variables are placed into your webpage.

Thankfully, many sinks where variables can be placed are safe. This is because these sinks treat the variable as text and will never execute it. Try to refactor your code to remove references to unsafe sinks like innerHTML, and instead use textContent or value.

```
elem.textContent = dangerVariable;
elem.insertAdjacentText(dangerVariable);
elem.className = dangerVariable;
elem.setAttribute(safeName, dangerVariable);
formfield.value = dangerVariable;
document.createTextNode(dangerVariable);
document.createElement(dangerVariable);
elem.innerHTML = DOMPurify.sanitize(dangerVar);
```

Safe HTML Attributes include: `align`, `alink`, `alt`, `bgcolor`, `border`, `cellpadding`, `cellspacing`, `class`, `color`, `cols`, `colspan`, `coords`, `dir`, `face`, `height`, `hspace`, `ismap`, `lang`, `marginheight`, `marginwidth`, `multiple`, `nohref`, `noresize`, `noshade`, `nowrap`, `ref`,

`rel`, `rev`, `rows`, `rowspan`, `scrolling`, `shape`, `span`, `summary`, `tabindex`, `title`, `usemap`, `valign`, `value`, `vlink`, `vspace`, `width`.

For attributes not reported above, ensure that if JavaScript code is provided as a value, it cannot be executed.

Other Controls

Framework Security Protections, Output Encoding, and HTML Sanitization will provide the best protection for your application. OWASP recommends these in all circumstances.

Consider adopting the following controls in addition to the above.

- Cookie Attributes - These change how JavaScript and browsers can interact with cookies. Cookie attributes try to limit the impact of an XSS attack but don't prevent the execution of malicious content or address the root cause of the vulnerability.
- Content Security Policy - An allowlist that prevents content being loaded. It's easy to make mistakes with the implementation so it should not be your primary defense mechanism. Use a CSP as an additional layer of defense and have a look at the cheatsheet here.
- Web Application Firewalls - These look for known attack strings and block them. WAF's are unreliable and new bypass techniques are being discovered regularly. WAFs also don't address the root cause of an XSS vulnerability. In addition, WAFs also miss a class of XSS vulnerabilities that operate exclusively client-side. WAFs are not recommended for preventing XSS, especially DOM-Based XSS.

XSS Prevention Rules Summary

These snippets of HTML demonstrate how to render untrusted data safely in a variety of different contexts.

Data Type: String Context: HTML Body Code: `<span>UNTRUSTED DATA </span>` Sample Defense: HTML Entity Encoding (rule #1)

Data Type: String Context: Safe HTML Attributes Code: `<input type="text" name="fname" value="UNTRUSTED DATA ">` Sample Defense: Aggressive HTML Entity Encoding (rule #2), Only place untrusted data into a list of safe attributes (listed below), Strictly validate unsafe attributes such as background, ID and name.

Data Type: String Context: GET Parameter Code: `<a href="/site/search?value=UNTRUSTED DATA ">clickme</a>` Sample Defense: URL Encoding (rule #5).

Data Type: String Context: Untrusted URL in a SRC or HREF attribute Code: `<a href="UNTRUSTED URL ">clickme</a> <iframe src="UNTRUSTED URL " />` Sample Defense: Canonicalize input, URL Validation, Safe URL verification, Allow-list http and HTTPS URLs only (Avoid the JavaScript Protocol to Open a new Window), Attribute encoder.

Data Type: String Context: CSS Value Code: `HTML <div style="width: UNTRUSTED DATA ;">Selection</div>` Sample Defense: Strict structural validation (rule #4), CSS hex encoding, Good design of CSS features.

Data Type: String Context: JavaScript Variable Code: `<script>var currentValue='UNTRUSTED DATA ';</script> <script>someFunction('UNTRUSTED DATA ');</script>` Sample Defense: Ensure JavaScript variables are quoted, JavaScript hex encoding, JavaScript Unicode encoding, avoid backslash encoding (`\"` or `\'` or `\\`).

Data Type: HTML Context: HTML Body Code: `<div>UNTRUSTED HTML</div>` Sample Defense: HTML validation (JSoup, AntiSamy, HTML Sanitizer...).

Data Type: String Context: DOM XSS Code: `<script>document.write("UNTRUSTED INPUT: " + document.location.hash );<script/>` Sample Defense: DOM based XSS Prevention Cheat Sheet |

Output Encoding Rules Summary

The purpose of output encoding (as it relates to Cross Site Scripting) is 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. The following charts provides a list of critical output encoding methods needed to stop Cross Site Scripting.

Encoding Type: HTML Entity Encoding Mechanism: Convert `&` to `&`, Convert `<` to `<`, Convert `>` to `>`, Convert `"` to `&quot;`, Convert `'` to `&#x27;`

Encoding Type: HTML Attribute Encoding Encoding Mechanism: Encode all characters with the HTML Entity `&#xHH;` format, including spaces, where **HH** represents the hexadecimal value of the character in Unicode. For example, `A` becomes `&#x41;`. All alphanumeric characters (letters A to Z, a to z, and digits 0 to 9) remain unencoded.

Encoding Type: URL Encoding Encoding Mechanism: Use standard percent encoding, as specified in the W3C specification, to encode parameter values. Be cautious and only encode parameter values, not the entire URL or path fragments of a URL.

Encoding Type: JavaScript Encoding Encoding Mechanism: Encode all characters using the Unicode `\uXXXX` encoding format, where **XXXX** represents the hexadecimal Unicode code point.

For example, `A` becomes `\u0041`. All alphanumeric characters (letters A to Z, a to z, and digits 0 to 9) remain unencoded.

Encoding Type: CSS Hex Encoding Encoding Mechanism: CSS encoding supports both `\XX` and `\XXXXXX` formats. To ensure proper encoding, consider these options: (a) Add a space after the CSS encode (which will be ignored by the CSS parser), or (b) use the full six-character CSS encoding format by zero-padding the value. For example, `A` becomes `\41` (short format) or `\000041` (full format). Alphanumeric characters (letters A to Z, a to z, and digits 0 to 9) remain unencoded.

Common Anti-patterns: Ineffective Approaches to Avoid

Defending against XSS is hard. For that reason, some have sought shortcuts to preventing XSS.

We're going to examine two common anti-patterns that frequently show up in ancient posts, but are still commonly cited as solutions in modern posts about XSS defense on programmer forums such as Stack Overflow and other developer hangouts.

Sole Reliance on Content-Security-Policy (CSP) Headers

First, let us be clear, we are a strong proponent of CSP when it is used properly. In the context of XSS defense, CSP works best when it it is:

- Used as a defense-in-depth technique.
- Customized for each individual application rather than being deployed as a one-size-fits-all enterprise solution.

What we are against is a blanket CSP policy for the entire enterprise. Problems with that approach are:

Problem 1 - Assumption Browser Versions Support CSP Equally

There usually is an implicit assumption that all the customer browsers support all the CSP constructs that your blanket CSP policy is using. Furthermore, this assumption often is done without testing the explicitly the `User-Agent` request header to see if it indeed is a supported browser type and rejecting the use of the site if it is not. Why? Because most businesses don't want to turn away customers if they are using an outdated browser that doesn't support some CSP Level 2 or Level 3 construct that they are relying on for XSS prevention. (Statistically, almost all browsers support CSP Level 1 directives, so unless you are worried about Grandpa pulling out his old Windows 98 laptop and using some ancient version of Internet Explorer to access your site, CSP Level 1 support can probably be assumed.)

Problem 2 - Issues Supporting Legacy Applications

Mandatory universal enterprise-wide CSP response headers are inevitably going to break some web applications, especially legacy ones. This causes the business to push-back against AppSec guidelines and inevitably results in AppSec issuing waivers and/or security exceptions until the application code can be patched up. But these security exceptions allow cracks in your XSS armor, and even if the cracks are temporary they still can impact your business, at least on a reputational basis.

Reliance on HTTP Interceptors

The other common anti-pattern that we have observed is the attempt to deal with validation and/or output encoding in some sort of interceptor such as a Spring Interceptor that generally implements `org.springframework.web.servlet.HandlerInterceptor` or as a JavaEE servlet filter that implements `javax.servlet.Filter`. While this can be successful for very specific applications (for instance, if you validate that all the input requests that are ever rendered are only alphanumeric data), it violates the major tenet of XSS defense where perform output encoding as close to where the data is rendered is possible. Generally, the HTTP request is examined for query and POST parameters but other things HTTP request headers that might be rendered such as cookie data, are not examined. The common approach that we've seen is someone will call either `ESAPI.validator().getValidSafeHTML()` or `ESAPI.encoder.canonicalize()` and depending on the results will redirect to an error page or call something like `ESAPI.encoder().encodeForHTML()`. Aside from the fact that this approach often misses tainted input such as request headers or "extra path information" in a URI, the approach completely ignores the fact that the output encoding is completely non-contextual. For example, how does a servlet filter know that an input query parameter is going to be rendered in an HTML context (i.e., between HTML tags) rather than in a JavaScript context such as within a `<script>` tag or used with a JavaScript event handler attribute? It doesn't. And because JavaScript and HTML encoding are not interchangeable, you leave yourself still open to XSS attacks.

Unless your filter or interceptor has full knowledge of your application and specifically an awareness of how your application uses each parameter for a given request, it can't succeed for all the possible edge cases. And we would contend that it never will be able to using this approach because providing that additional required context is way too complex of a design and accidentally introducing some other vulnerability (possibly one whose impact is far worse than XSS) is almost inevitable if you attempt it.

This naive approach usually has at least one of these four problems.

Problem 1 - Encoding for specific context not satisfactory for all URI paths

One problem is the improper encoding that can still allow exploitable XSS in some URI paths of your application. An example might be a 'lastname' form parameter from a POST that normally is displayed between HTML tags so that HTML encoding is sufficient, but there may be an edge case or two where lastname is actually rendered as part of a JavaScript block where the HTML encoding is not sufficient and thus it is vulnerable to XSS attacks.

Problem 2 - Interceptor approach can lead to broken rendering caused by improper or double encoding

A second problem with this approach can be the application can result in incorrect or double encoding. E.g., suppose in the previous example, a developer has done proper output encoding for the JavaScript rendering of lastname. But if it is already been HTML output encoded too, when it is rendered, a legitimate last name like "O'Hara" might come out rendered like "O\'Hara".

While this second case is not strictly a security problem, if it happens often enough, it can result in business push-back against the use of the filter and thus the business may decide on disabling the filter or a way to specify exceptions for certain pages or parameters being filtered, which in turn will weaken any XSS defense that it was providing.

Problem 3 - Interceptors not effective against DOM-based XSS

The third problem with this is that it is not effective against DOM-based XSS. To do that, one would have to have an interceptor or filter scan all the JavaScript content going as part of an HTTP response, try to figure out the tainted output and see if it it is susceptible to DOM-based XSS. That simply is not practical.

Problem 4 - Interceptors not effective where data from responses originates outside your application

The last problem with interceptors is that they generally are oblivious to data in your application's responses that originate from other internal sources such as an internal REST-based web service or even an internal database. The problem is that unless your application is strictly validating that data *at the point that it is retrieved* (which generally is the only point your application has enough context to do a strict data validation using an allow-list approach), that data should always be considered tainted. But if you are attempting to do output encoding or strict data validation all of tainted data on the HTTP response side of an interceptor (such as a Java servlet filter), at that point, your application's interceptor will have no idea of there is tainted data present from those REST web services or other databases that you used. The approach that generally is used on response-side interceptors attempting to provide XSS defense has been to only consider the matching "input parameters" as tainted and do output encoding or HTML sanitization on them and everything else is considered safe. But sometimes it's not? While it frequently is assumed that all internal web services and all internal databases can be "trusted" and used as it, this is a very bad

assumption to make unless you have included that in some deep threat modeling for your application.

For example, suppose you are working on an application to show a customer their detailed monthly bill. Let's assume that your application is either querying a foreign (as in not part of your specific application) internal database or REST web service that your application uses to obtain the user's full name, address, etc. But that data originates from another application which you are assuming is "trusted" but actually has an unreported persistent XSS vulnerability on the various customer address-related fields. Furthermore, let's assume that you company's customer support staff can examine a customer's detailed bill to assist them when customers have questions about their bills. So nefarious customer decides to plant an XSS bomb in the address field and then calls customer service for assistance with the bill. Should a scenario like that ever play out, an interceptor attempting to prevent XSS is going to miss that completely and the result is going to be something much worse than just popping an alert box to display "1" or "XSS" or "pwn'd".

Summary

One final note: If deploying interceptors / filters as an XSS defense was a useful approach against XSS attacks, don't you think that it would be incorporated into all commercial Web Application Firewalls (WAFs) and be an approach that OWASP recommends in this cheat sheet?

Related Articles

XSS Attack Cheat Sheet:

The following article describes how attackers can exploit different kinds of XSS vulnerabilities (and this article was created to help you avoid them):

- OWASP: XSS Filter Evasion Cheat Sheet.

Description of XSS Vulnerabilities:

- OWASP article on XSS Vulnerabilities.

Discussion about the Types of XSS Vulnerabilities:

- Types of Cross-Site Scripting.

How to Review Code for Cross-Site Scripting Vulnerabilities:

- OWASP Code Review Guide article on Reviewing Code for Cross-site scripting Vulnerabilities.

How to Test for Cross-Site Scripting Vulnerabilities:

- OWASP Testing Guide article on testing for Cross-Site Scripting vulnerabilities.
- XSS Experimental Minimal Encoding Rules Provides examples and guidelines for experimental minimal encoding strategies to prevent Cross-Site Scripting (XSS) attacks.