Cross Site Scripting Prevention Cheat Sheet

# Introduction

This cheat sheet provides guidance to prevent XSS vulnerabilities.

Cross-Site Scripting (XSS) is a misnomer. The name originated from early versions of the attack where stealing data cross-site was the primary focus. Since then, it has extended to include injection of basically any content, but we still refer to this as XSS. XSS is serious and can lead to account impersonation, observing user behaviour, loading external content, stealing sensitive data, and more.

**This cheatsheet is a list of techniques to prevent or limit the impact of XSS. No single technique will solve XSS. Using the right combination of defensive techniques is necessary to prevent XSS.**

# Framework Security

Fewer XSS bugs appear in applications built with modern web frameworks. These frameworks steer developers towards good security practices and help mitigate XSS by using templating, auto-escaping, and more. That said, developers need to be aware of problems that can occur when using frameworks insecurely such as:

*escape hatches* that frameworks use to directly manipulate the DOM

React’s without sanitising the HTML

dangerouslySetInnerHTML

React cannot handle or URLs without specialized validation

javascript:

data:

Angular’s

bypassSecurityTrustAs\*

Template injection

functions

Out of date framework plugins or components and more

Understand 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. This is where Output Encoding and HTML Sanitization are critical. OWASP are producing framework specific cheatsheets for React, Vue, and Angular.

# XSS Defense Philosophy

For XSS attacks to be successful, an attacker needs to insert and execute malicious content in a webpage. Each variable 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, frameworks aren't perfect and security gaps still exist in popular frameworks like React and Angular. Output Encoding and HTML Sanitization help address those gaps.

# Output Encoding

Output Encoding is recommended when you need to safely display data exactly as a user typed it in. Variables should not be interpreted as code instead of text. This section covers each form of output encoding, where to use it, and where to avoid using dynamic variables entirely.

Start with using your framework’s default output encoding protection when you wish to display data as the user typed it in. 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 or

<div>

<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, use HTML entity encoding for that variable as you add it to a web template.

Here are some examples of encoded values for specific characters.

If you're using JavaScript for writing to HTML, look at the

.textContent

**Sink** and will automatically HTML Entity Encode.

&

<

> "

'

&amp; &lt; &gt; &quot;

&#x27;

## Output Encoding for “HTML Attribute Contexts”

attribute as it is a **Safe**

“HTML Attribute Contexts” refer to placing a variable 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 using JavaScript for writing to a HTML Attribute, look at the and

.setAttribute

methods which will automatically HTML Attribute Encode. Those are **Safe Sinks**

[attribute]

as long as the attribute name is hardcoded and innocuous, like or class . Generally,



id

attributes that accept JavaScript, such as onClick , are **NOT safe** to use with untrusted attribute values.

## Output Encoding for “JavaScript Contexts”

“JavaScript Contexts” refer to placing variables into inline JavaScript which is then embedded in an HTML document. This is commonly seen in programs that heavily use custom JavaScript embedded in their web pages.

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 format. Encoding libraries often have a

\xHH

or similar to support this function.

EncodeForJavaScript

Please look at the [OWASP Java Encoder JavaScript encoding examples](https://owasp.org/www-project-java-encoder/) for examples of proper JavaScript use that requires minimal encoding.

application/json

text/html

For JSON, verify that the prevent XSS.

Content-Type

header is

and not to

## Output Encoding for “CSS Contexts”

“CSS Contexts” refer to variables placed into inline CSS. This is common when you want users to be able to customize the look and feel of their webpages. CSS is surprisingly powerful and 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.

// Add CSS Encoding Advice

## 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](http://www.owasp.org/?test=%24varUnsafe)">link</a >

Encode all characters with the same as JS and CSS.



%HH

### Common Mistake

encoding format. Make sure any attributes are fully quoted,

There will be situations where you use a URL in different contexts. The most common one

href

src

<a>

would be adding it to an

or attribute of an

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 of 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

Sometimes users need to author HTML. One scenario would be allow users to change the styling or structure of content inside a WYSIWYG editor. Output encoding here 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](https://github.com/cure53/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 a comprehensive list, check out the [DOMPurify allowlist](https://github.com/cure53/DOMPurify/blob/main/src/attrs.js)

# 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](https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html).

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

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Context** | **Code Sample** | **Defense** |
| String | HTML Body | <span>UNTRUSTED | HTML Entity Encoding (rule #1). |
|  |  | DATA </span> |  |
| String | Safe HTML Attributes | <input type="text" | Aggressive HTML Entity Encoding (rule #2), Only place |
|  |  | name="fname"  value="UNTRUSTED | untrusted data into a list of safe  attributes (listed below), Strictly |
|  |  | DATA "> | validate unsafe attributes such as background, ID and name. |
| String | GET  Parameter | <a href="/site/searc | URL Encoding (rule #5). |
|  |  | h?value=UNTRUSTED |  |
|  |  | DATA |  |
|  |  | ">clickme</a> |  |
| String | Untrusted URL in a | <a href="UNTRUSTED | Canonicalize input, URL Validation, Safe URL verification, |
|  | SRC or  HREF | URL ">clickme</a>  <iframe | Allow-list http and HTTPS URLs  only (Avoid the JavaScript |
|  | attribute | src="UNTRUSTED URL " /> | Protocol to Open a new Window), Attribute encoder. |
| String | CSS Value | HTML <div style="width: | Strict structural validation (rule #4), CSS Hex encoding, Good |
|  |  | UNTRUSTED DATA | design of CSS Features. |
|  |  | ;">Selection</div |  |
|  |  | > |  |



**Data Type**

**Context**

**Code Sample**

**Defense**

String

JavaScript

Variable

<script>var currentValue='UNT RUSTED DATA ';

</script>

<script>someFunct ion('UNTRUSTED

DATA ');</script>

Ensure JavaScript variables are quoted, JavaScript Hex Encoding, JavaScript Unicode Encoding, Avoid backslash encoding ( \" or

\' or \\ ).

HTML

HTML Body

<div>UNTRUSTED

HTML</div>

HTML Validation (JSoup,

AntiSamy, HTML Sanitizer...).

String

DOM XSS

<script>document.

write("UNTRUSTED

[DOM based XSS Prevention](https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html)

[Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html)

INPUT: " +

document.location

.hash );<script/>

## 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 details a list of critical output encoding methods needed to stop 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, encode all characters with the HTML

Entity &#xHH; format, including spaces. (**HH** = Hex Value)

URL

Encoding

Standard percent encoding, see [here](https://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.



CSS encoding supports \XX and \XXXXXX . Using a two character encode can cause problems if the next character continues the encode sequence. There are two solutions: (a) Add a space after the CSS encode (will be ignored by the CSS parser) (b) use the full amount of CSS encoding possible

by zero padding the value.

CSS Hex

Encoding

Except for alphanumeric characters, encode all characters with the \uXXXX

unicode encoding format (**X** = Integer).

JavaScript

Encoding

**Encoding Mechanism**

**Encoding**

**Type**

# Related Articles

### XSS Attack Cheat Sheet:

The following article describes how to exploit different kinds of XSS Vulnerabilities that this article was created to help you avoid:

OWASP: [XSS Filter Evasion Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html).

### Description of XSS Vulnerabilities:

OWASP article on [XSS](https://owasp.org/www-community/attacks/xss/) Vulnerabilities.

### Discussion on the Types of XSS Vulnerabilities:

[Types of Cross-Site Scripting](https://owasp.org/www-community/Types_of_Cross-Site_Scripting).

### How to Review Code for Cross-site scripting Vulnerabilities:

[OWASP Code Review Guide](https://owasp.org/www-project-code-review-guide/) article on [Reviewing Code for Cross-site scripting](https://wiki.owasp.org/index.php/Reviewing_Code_for_Cross-site_scripting) Vulnerabilities.

### How to Test for Cross-site scripting Vulnerabilities:

[OWASP Testing Guide](https://owasp.org/www-project-web-security-testing-guide/) article on testing for Cross-Site Scripting vulnerabilities. [XSS Experimental Minimal Encoding Rules](https://wiki.owasp.org/index.php/XSS_Experimental_Minimal_Encoding_Rules)