



# Cross-Site Scripting (XSS)

By: Vie 🐦 and Ming 💧

XSS is a classic vulnerability that existed in the first days of the internet. Nowadays, it's pretty difficult to find some easy-to-do XSS exploits out in the wild, but there are clever ways to achieve such exploits despite the modern security measures that exist to mitigate it.

## Brief Overview

Simplified, XSS exploits occur when user-input is not properly sanitized before it's put into a webpage. In this scenario, we can achieve many things - but specifically for XSS exploits, we can achieve **arbitrary javascript execution**.

In most CTFs, where there is an XSS vulnerability, there is a way to embed your own rogue code into a webpage, and trick people into visiting that webpage to achieve different effects. The common scenario? Cookie stealing, or tricking an admin into performing authenticated requests without them knowing.

XSS vulnerabilities give way for an attacker to steal another user's cookies, which are defined as bits of data transmitted between a server (the website you visit) and a client (you, on your browser, visiting that website). This is certainly not the only situation where XSS are effective exploits, but this is a common one you'll often see employed in CTFs.

There are plenty of different XSS payloads to try:

*As a URI*

```
javascript:alert(1);
```

*As a script tag (why does this work?)*

```
<script>alert(1)</script>
```

As an image tag (why does this work?)

```
<img src=x onerror=alert(1);></img>
```

And more...

*NOTE: In these examples we're using the JS function `alert()` as a way to test for XSS exploits. Realistically, any small JS expression will work, we just like using `alert()` since its quick and easy to type.*

There are 3(ish) types of XSS exploits to know about:

## Reflected XSS:

Probably the most common form of XSS attacks out there. Reflected XSS situations arise when a webpage will "reflect" user-input back in a response. This kind of XSS attack is non-persistent, meaning it doesn't last once the browser ends the session with the website where the XSS payload resides.

## Stored XSS:

XSS exploits that are stored somewhere in memory in the server, and then execute when its rendered in a later response. Common vehicles of attack are user comments in forums, or rendering user titles from a database. This type of XSS attack is persistent, since it is stored in some way by the server, then returned in a response in the future. As long as the payload is stored in the server, it will continue to exploit people who visit the website with it.

## DOM-Based XSS:

Not super common, but they are relevant nonetheless. In here, the XSS payload results from a modified DOM environment. The modifications made to the DOM environment are what execute



What's a DOM?

The Document-Object-Module system (DOM for short) is a programming interface for web

malicious code, but the client-side code itself remains unchanged.

clients, which organizes things like HTML documents into a tree-like structure.

It's important to describe another form of XSS that isn't super relevant for us called self-XSS. Simply put, its situations where XSS was technically achievable (rogue JavaScript may have been able to be executed on a webpage), but that JS code can only affect the attacker (unless you manage to convince someone to paste unknown code into their browser console 😊).

## XSS Mitigations

### DOMPurify

DOMpurify is an open-source and rigorous library used for input sanitization. It is an incredibly powerful and effective library that sanitizes HTML elements, one of the core vehicles for XSS delivery. If you see DOMpurify used in a webpage, it could be incredibly difficult to execute XSS payloads due to most of your malicious input being detected and sanitized properly.

### Basic Input Sanitizers

Perhaps developers may code their own sanitization functions that serve the purpose of escaping/deleting any potentially dangerous characters out of your input. Look out for functions that may escape characters of interest, such as <>, :, ', or “.

### Content-Security-Policy

Go to [github.com](https://github.com) and inspect the network tab when you make a request on there. Notice a specific response header called Content-Security-Policy.

This is a list of directives that are given by the server, which tells the browser what sources can be trusted for specific scripts and HTML tags. CSPs can either be implemented as a response header or a meta-tag embedded in the website, but the

effect achieved is the same: they will restrict the origins that certain kinds of content will be loaded from.

The format of a CSP will have 2 main components: the directives, and the sources. Take this example:

```
default-src: none;
```

This is a directive and source pair. The `default-src` directive is the guidelines that the browser will follow by *default* (duh) unless some other directive says something else. In this case, the source value, "none", further tells the browser that it shouldn't trust any script loaded from any source. Altogether, you'd read this as: by default, reject any script loaded from anywhere.

An example CSP may look like:

```
<meta http-equiv="Content-Security-Policy"
      content="default-src 'self'; img-src https://*;">
```

How do we interpret this? Well, the `default-src` directive is self-explanatory, it's the course of action to take when all other given directives do not match the scenario the browser is encountering. In this case, `default-src` is set to 'self', meaning that the browser shouldn't trust (and therefore, won't render) any script that it sees on the webpage EXCEPT for scripts loaded by the webpage itself.

The `img-src` directive tells the browser which images to trust (anything with an image file extension) on the website. In this case, any image coming from an HTTPS website is allowed.

But what happens if you do something that a CSP doesn't allow?

Consider the following CSP:


```
default-src 'self'; img-src 'self' allowed-website.com; style-src 'self'; script-src 'none'; object-src 'none';
```

This is a pretty robust and strong CSP. Notably, it accounts for most all potential sources of malicious JS. If we try a payload such as `<script>alert(1);</script>` then the payload

would be embedded into the page, but it would never run. Inspecting your console may show an error, like so:

```
✖ Refused to execute inline script because it violates the following Content Security Policy localhost/:4  
directive: "script-src none". Either the 'unsafe-inline' keyword, a hash ('sha256-  
5jFwrAK0UV47oFbVg/iCCBbxD8X1w+Qvo0Uepu4C2YA='), or a nonce ('nonce-...') is required to enable inline  
execution.
```

CSPs can be as long as possible or as short as possible. With each new directive stipulated in a CSP is a set of rules that apply to whatever directive was called. There are plenty of different 'rules' or values these directives can have, and a more comprehensive list of them resides [here](#).

 There exists a certain CSP source called the 'nonce', which takes the form of a randomly generated hash that the stipulated directives MUST have, in order to be considered 'safe' and can be rendered. A good nonce will be hard to guess, and changes with each load on the webpage, meaning that someone can't reuse a nonce to inject their payload.

It is *IMPERATIVE* that a CSP follows the strict syntax conventions defined here. CSPs can only be a collection of directive:source pairs separated by a `;`, and any deviations to this may break the CSP (rendering it useless) or change its rules.

## Bypassing XSS Protections: Some Examples

**Sanitizers that only look out for `<script>` or `<img>`**

We have the following code for this example:

```
(async () => {  
  await new Promise((resolve) => {  
    window.addEventListener('load', resolve);  
  });  
  
  const content = window.location.hash.substring(1);  
  display(atob(content));  
})();
```

```

function display(input) {
    document.documentElement.innerHTML = clean(input);
}

//grabs every HTML element in the webpage, and passes it to sanitize()
function clean(input) {
    const template = document.createElement('template');
    const html = document.createElement('html');
    template.content.appendChild(html);
    html.innerHTML = input;

    sanitize(html);

    const result = html.innerHTML;
    return result;
}

//XSS input sanitizer!
function sanitize(element) {
    const attributes = element.getAttributeNames();
    for (let i = 0; i < attributes.length; i++) {
        //Only the given attributes are allowed. No attributes that allow JS!
        if (!['src', 'width', 'height', 'alt', 'class'].includes(attributes[i])) {
            element.removeAttribute(attributes[i]);
        }
    }

    const children = element.children;
    for (let i = 0; i < children.length; i++) {
        //If anything in the HTML element is a script element, remove it.
        if (children[i].nodeName === 'SCRIPT') {
            element.removeChild(children[i]);
            i--;
        } else {
            //Trust the natural recursion or whatever
            sanitize(children[i]);
        }
    }
}

```

The function `sanitize` is of importance: it will first iterate through the attributes of a provided HTML element, and if any attribute is encountered with a name that doesn't match the provided list ('src', 'width', 'height', 'alt', 'class'), then that attribute is removed. It then, recursively, explicitly searches for a tag in the children of the HTML element with the name "script", and removes it.

This code will remove `<img>`-based and `<script>`-based XSS payloads. But, this is a very basic input-sanitizer that doesn't account for *all* possible XSS payloads. What other

XSS payloads can be used that won't be detected by the `sanitize` function?

## Content-Security-Policy

If you come across a good CSP, it may be near impossible to achieve XSS.

However, some CSPs can be pretty weak due to oversights or misconfigurations.

Consider this CSP:

```
Content-Security-Policy: script-src https://facebook.com https://google.com 'unsafe-inline' https://*;
```

The source 'unsafe-inline' is self-explanatory, allowing for JS execution in the HTML code without any checking or validation. A classic XSS payload can easily bypass this CSP.

```
<!--unsafe-inline is a CSP rule that tells the browser that any JS that is found in the HTML element (anything in a script tag) is allowed to be rendered, regardless of where that JS came from-->
"/><script>alert(1337);</script>
```