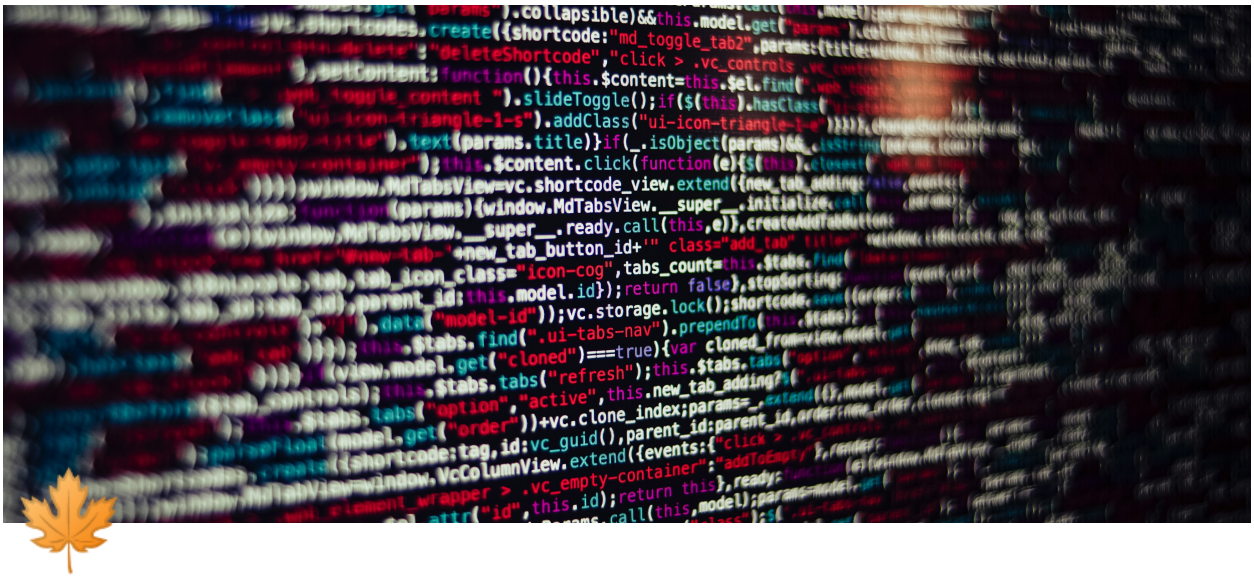


MapleCTF: XSS Guide

This is an introductory guide for XSS-based attacks that you may come across in MapleCTF. While not exhaustive, this will serve to be a baseline to introduce everyone into the wonderful world of Cross-Site Exploits!

Authors: Vie 🦢

Context

(This section covers some basics of HTML, REST APIs, and JS. Feel free to skip if you're already comfortable with this stuff!)

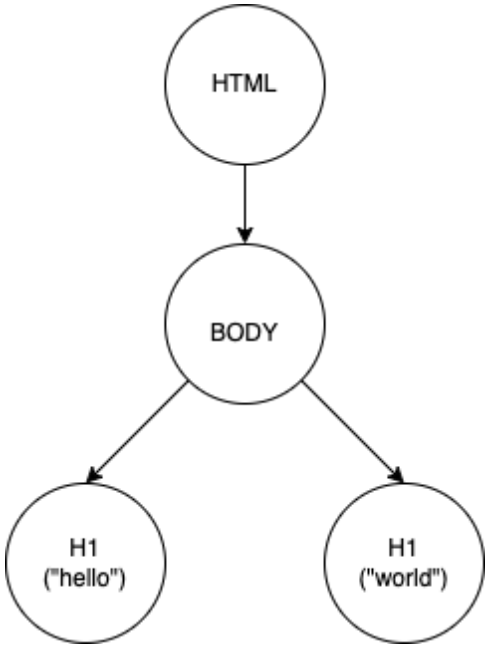
HTML

The basis of every webpage ever on the internet. This markup language is a fundamental part of the web! When dealing with HTML, some important things to note are as follows:

HTML is organized into a tree. Take a look:

```
<html>
  <body>
    <h1>hello</h1>
    <h1>world</h1>
  </body>
</html>
```

There’s 4 nodes that make up this tree. It’s better to visualize in a graph:



The head node is the `html` tag, and the `h1` tags are the leaf nodes. You can try it out - copy paste the above HTML code into a text editor and save it as an `.html` file. You can open it in a browser and it should look like this.

hello

world

JavaScript

If you want your website to have fancy animations and other cool things, you'll likely be familiarized with JavaScript. JS started as a front-end rendering language, but has quickly rose to great heights with what it can do now. Due to the nature of JS' beginning, HTML allows you to execute JavaScript in its skeleton, using the `<script>` tag. So, this bit of HTML code would execute the JS `alert` function (try it out!):

```
<html>
  <body>
    <script>alert("Hello world");</script>
  </body>
</html>
```

Now, modern day websites will probably have hundreds, if not thousands, of lines of JS running all the time. Having that much JS code is pretty tedious to keep in one HTML file. In this case, when the amount of JS becomes so large it needs to be in its own file, you can simply host it in a different `.js` file and import the code using either a script tag with the `src` attribute.

```
<script src="script.js"></script>
```

Using `script` tags with the `src` attribute will specify that the execution of the script will occur with some external resource, in this case called `script.js`.

What is Cross-Site Scripting?

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.

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 executes rogue JS code.

What's a DOM?

The Document-Object-Module
system (DOM for short) is a
programming interface for web clients, which organizes things like HTML documents into a tree-like structure.

In a CTF

CTFs will often employ challenges that require you to take advantage of this vulnerability. In the goold old days of the internet, XSS vulnerabilities were frequently used against insecure websites to steal a user’s personal information, such as their cookies.

And you may think “cookies - the thing that a website tells me that they use but I don’t really care about?” Yea, those cookies. When the XSS vulnerability reigned as king of the exploits back then, people used cookies as a way to hold

sensitive information such as a user’s password, location, and more. The probelm here is that cookies can be easily manipulated and handled by those who don’t own them, and bad shenanigans can ensue as a result. Nowadays, people are more cautious and cookies aren’t used for important stuff anymore, but that doesn’t mean the attack is obsolete!

In a CTF, you will very likely see some user with special cookies in an XSS challenge, and it is your job to try and steal those cookies. The best way to approach this is to have some sort of server or other place you can observe the traffic to, and use an XSS exploit on a vulnerable website to forward that user’s cookies to your server.

Mitigations

Sanitizers

Now you may be thinking: “Hold on. If the issues causing XSS are user-input mishaps, why don’t we just clean the user input?” And that’s an incredibly valid question - giving rise to a bunch of HTML-based XSS sanitizers that serve to clean the provided input to make it hax-free.

Now, you can use a few open-source libraries available, and the most commonly used are DOMpurify, xss, and a few others, but you can also make your own.

While technically within the realm of possibility, I would caution against creating your own sanitizer, as XSS attacks can get creative and may involve some fancy workarounds to bypass conventional filters. There are a few obvious offenders you would have to worry about: rogue script tags, potential areas to store JS, etc etc, but focusing on blocking only those kinds of inputs still leaves plenty of freedom for an attacker to get crafty!

Content Security Policies

Go to 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.

CSPs can be implemented as a meta tag in the HTML of the webpage, or as a header that the server includes in a response to a request.

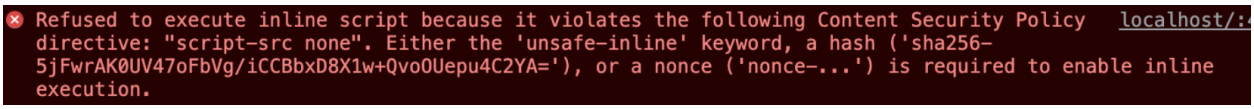
In most cases, CSPs are implemented as the latter - as a response header. This means that when the client (which is us, on our browser) interacts with the server, the server replies and includes a specific “Content-Security-Policy” header in the response. Once that is recieved by the client, the browser will adhere to whatever directives it has.

This means the browser listens to that specific header and will execute only what 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:



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.

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.