Polyglots: The Ultimate XSS Payloads.

Learn to execute XSS attacks in any context with just one payload.

Examples

XSS Polyglot: embed | open

Polyglot credit: 0xSobky

video transcript

Hello, world! I'm Jesse from Chef Secure, and this is an XSS polyglot.

```
jaVasCript:/*-/*`/*\`/*'/*"/*%0D%0A%0d%0a*/(/* */oNcliCk=alert()
)//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3ciframe/<iframe/oNloAd=alert()//>\x3e
```

It's a single, XSS payload that works in multiple contexts, and, in fact, it works to attack every recipe example we've covered thus far.

Polyglots can save you lots of time when looking for vulnerabilities when you don't have access to the code, or if you have a scanner doing the work for you, while you do more important, difficult things.

If I had shown you this polyglot before you knew what you now know, it would just be a confusing mess of code that somehow magically worked.

But after everything you've learned so far, it's –

No, you still have no idea what the heck's going on, do you?

But at least you have the knowledge, given a particular context, to figure out how the polyglot works to launch an attack.

The key is just to know what the context requires to work normally, and what's required to break out and deliver an XSS payload.

Let's look a little closer, starting off with a regular 'ol HTML content injection. We know there needs to be an opening tag to launch our attack, so let's just scan the polyglot until we find one.

(gasps)

```
jaVasCript:/*-/*`/*\`/*'/*"/*%0D%0A%0d%0a*/(/* */oNcliCk=alert() )//</stYle/
```

No, wait, that's a closing tag.

(gasps louder)

```
jaVasCript:/*-/*`/*\`/*'/*"/*%0D%0A%0d%0a*/(/* */oNcliCk=alert() )//</stYle/</titLe/
```

Nope! Nah, another closing tag. Darn it!

Here...we...go.

```
jaVasCript:/*-/*`/*\`/*'/*"/*%0D%0A%0d%0a*/(/* */oNcliCk=alert()
)//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3ciframe/<iframe/>
```

We're now opening an iframe element.

As a side note, a forward slash can be used after a tag name in place of a space character, which his helpful for bypassing filters.

Moving on, next we have an event handler onload equals alert, which you're already familiar with, followed by two slashes that work as a JavaScript comment after the alert, then the tag is closed.

```
jaVasCript:/*-/*`/*\`/*'/*"/*%0D%0A%0d%0a*/(/* */oNcliCk=alert()
)//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3ciframe/<iframe/oNloAd=alert()//>
```

So, overall, you know that the onload progress event handler is how this attack will work, and everything else you don't really care about, because it's not relevant to the execution of this payload. And the browser's automatically going to insert a closing iframe tag.

Let's see this in action right now. Go to the XSS Polyglot example for this recipe below.

XSS Polyglot: embed | open

You see that the polyglot is already set as the input's value, so make sure there's no escaping, and select the HTML context, then click the button to inject the polyglot.

After the alert, right click and inspect the text at the bottom that the polyglot injected.

You see that after the text, the iframe with the onload event handler is what triggered the alert, as expected.

So, as promised, there's no magic going on here.

Now let's move on and scan the polyglot again for the URL context.

```
jaVasCript:
```

Upfront, we already see the JavaScript scheme is being used with mixed capitalization bypass filters. Also, since we know we're in a JavaScript URL, everything else is gonna be treated as JavaScript.

Next a block comment is being opened,

```
jaVasCript:/*
```

so let's just jump to where it closes, so that we can get to the actual part of the code that gets run.

```
jaVasCript:/*-/*`/*\`/*'/*"/*%0D%0A%0d%0a*/
```

After this comment, there are parenthesis surrounding code, which works as a grouping operator like in math to set precedence.

```
jaVasCript:/*-/*`/*\`/*'/*"/*%0D%0A%0d%0a*/(/* */oNcliCk=alert() )
```

In other words, everything inside parenthesis gets run first.

The grouping operator here works, but it's pretty useless because there's no code on the outside to run. The real purpose is for another context, which we'll go over in a little bit.

Next, there's another block comment, and then, finally, a variable named oNcliCk set to the alert call. And this is where the attack executes.

```
jaVasCript:/*-/*`/*\`/*'/*"/*%0D%0A%0d%0a*/(/* */oNcliCk=alert() )
```

Afterward, there are just two slashes, which just comment out the rest of the polyglot.

```
jaVasCript:/*-/*`/*\`/*'/*"/*%0D%0A%0d%0a*/(/* */oNcliCk=alert() )//
```

So let's run this in the example now, setting the escaping to HTML, the context to URL, then click the inject polyglot button.

And you'll see at the bottom, there's a link that says URL. Click that, and an alert will execute exactly as we discussed.

Feel free to pause right now if you wanna try this out without escaping. Since it's a polyglot, you're gonna see that you still get execution, but it's gonna be different, because it's escaping out of the href attribute.

Now, as promised, let's go over those mysterious parenthesis from earlier.

In the URL context, the parenthesis inside the JavaScript scheme worked as an unnecessary grouping operator. However, if you're injecting inside a JavaScript regular expression, then those parenthesis are important.

First off, regular expressions are used to find patterns within strings.

They start and end with a forward slash, and the pattern is contained inside.

So, for the polyglot, this means it needs to first break out of the regular expression using a forward slash. So let's scan the polyglot so we can find that.

```
jaVasCript:/
```

It's right here.

Then afterward there's an asterisk for multiplication, followed by a minus, which becomes a negative sign, since we're already multiplying.

```
jaVasCript:/*-
```

Then we have block comments, followed by the grouping operator with the same variable and alert call from earlier.

```
jaVasCript:/*-/*`/*\`/*'/*"/*%0D%0A%0d%0a*/(/* */oNcliCk=alert() )
```

Now I'm no mathematician, but, where I come from, multiplication and minuses they're used for numbers.

But we all know JavaScript is very special, so it created a value called NaN, or Not a Number, which allows you to do math operations on things that aren't numbers.

So this isn't gonna throw any errors.

Let's try this out now in the example using HTML escaping in the JavaScript regular expression context.

Then click the inject polyglot button. There is an alert. So if you inspect the injection at the bottom, and look at the script containing the regular expression again, it works exactly as we discussed.

I've got a question. What happens when we take out the parenthesis?

Change the payload to remove the parenthesis and let's try this again.

No alert this time. So open the console. And look at the error message.

The assignment to the alert fails and errors because there isn't a valid variable name. Instead, everything to the left of the equals sign is being used, rather than just oNcliCk.

So, those parenthesis were important after all.

Now, finally, in the JavaScript string context, those parenthesis work as a function call that actually does produce an error, but the polyglot still gets the alert to execute, because the alert as an argument gets evaluated first.

But I'll let you check that out on your own if you'd like.

This is very good practice, by the way, in case the worst-case scenario happens where your site is being actively exploited.

Attackers are not gonna make it easy on you to figure out their code, yet you need to understand what the heck's going on, so you can fix the issue as quickly as possible.

So, if you ever come across a crazy -lookin payload like this, don't worry. All you need to do is where it might be injected, then simply go through it piece by piece, and then you can figure out exactly how it works, so, like me, you can ruin the magic for everyone.

Go to challenges for this recipe ›

Courses
Cross Site Scripting (XSS)

Learn More
About Chef Secure
Chef Secure Blog

Support
Contact
Security
Terms of use
Privacy policy