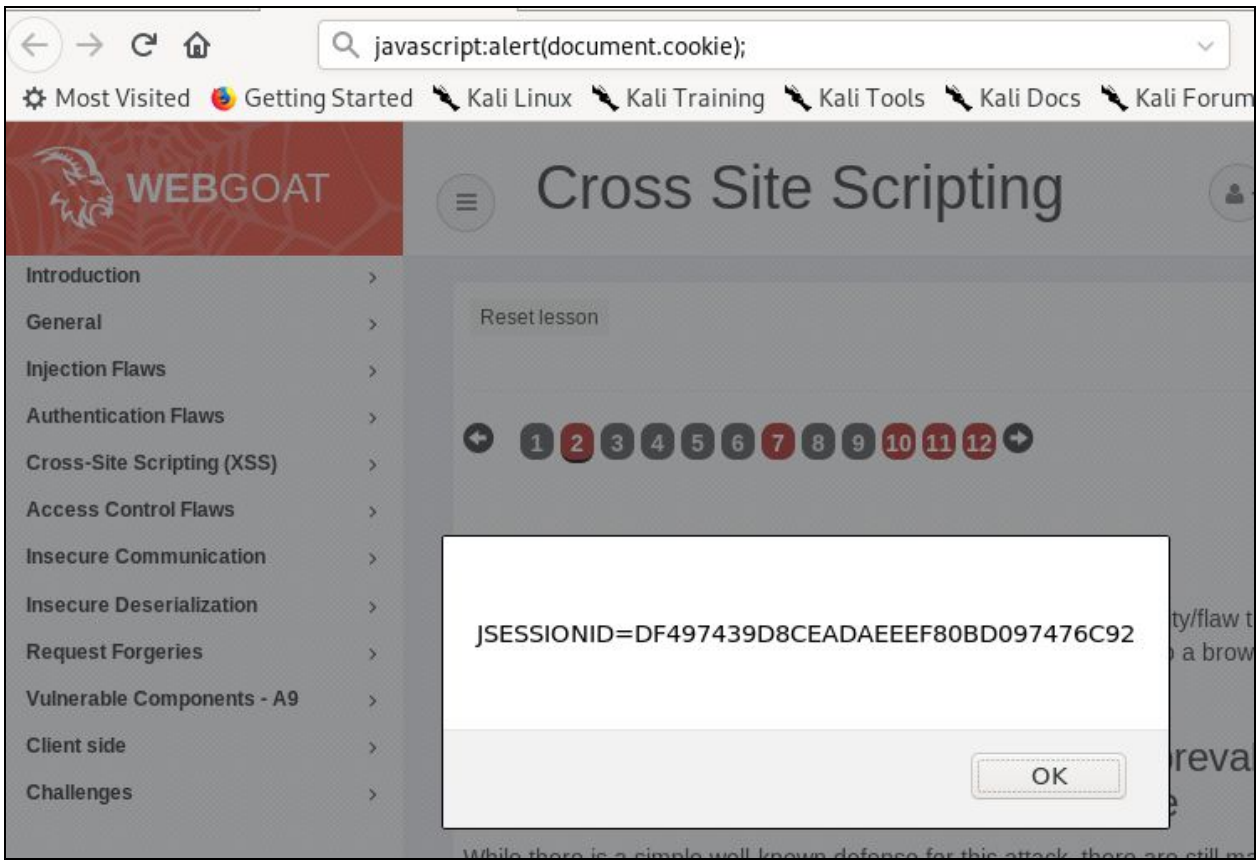
WebGoat Cross Site Scripting

Twitter: @BlackSheepSpicy
Twitch: https://twitch.tv/BlackSheepSpicy

2. This challenge is just a basic demonstration of Javascript being executable in the browser, to do that OWASP has up insert some javascript code into our URL input:

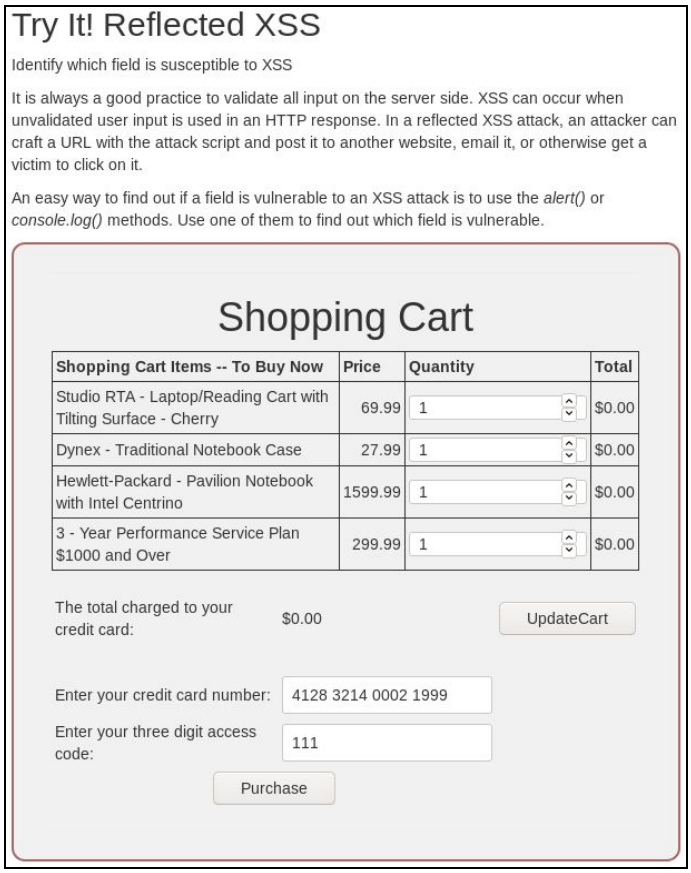
So when we do that we just get this action:



Likewise if we do the same on our other tab we get the same result:

So to answer the question OWASP asked: indeed the sessions are the same on each tab:

7. Onto the next challenge then:



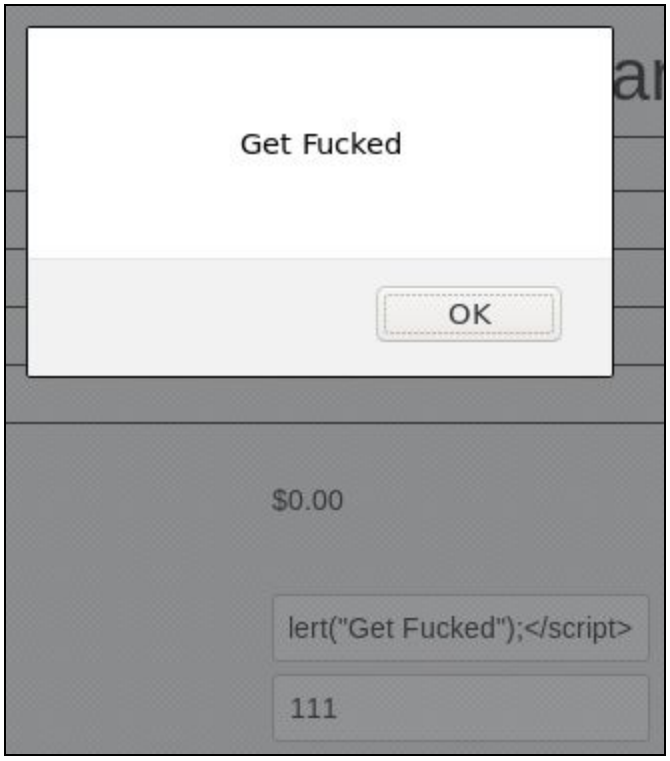
I love me a good fuzz sesh

To test each of these inputs I used this string:
<script>javascript:alert("Get Fucked");</script>

Note: the script tags here are crucial, found that out after one of my viewers suggested using them after shit didn't work (Vox_Light on twitch, rad dude.)

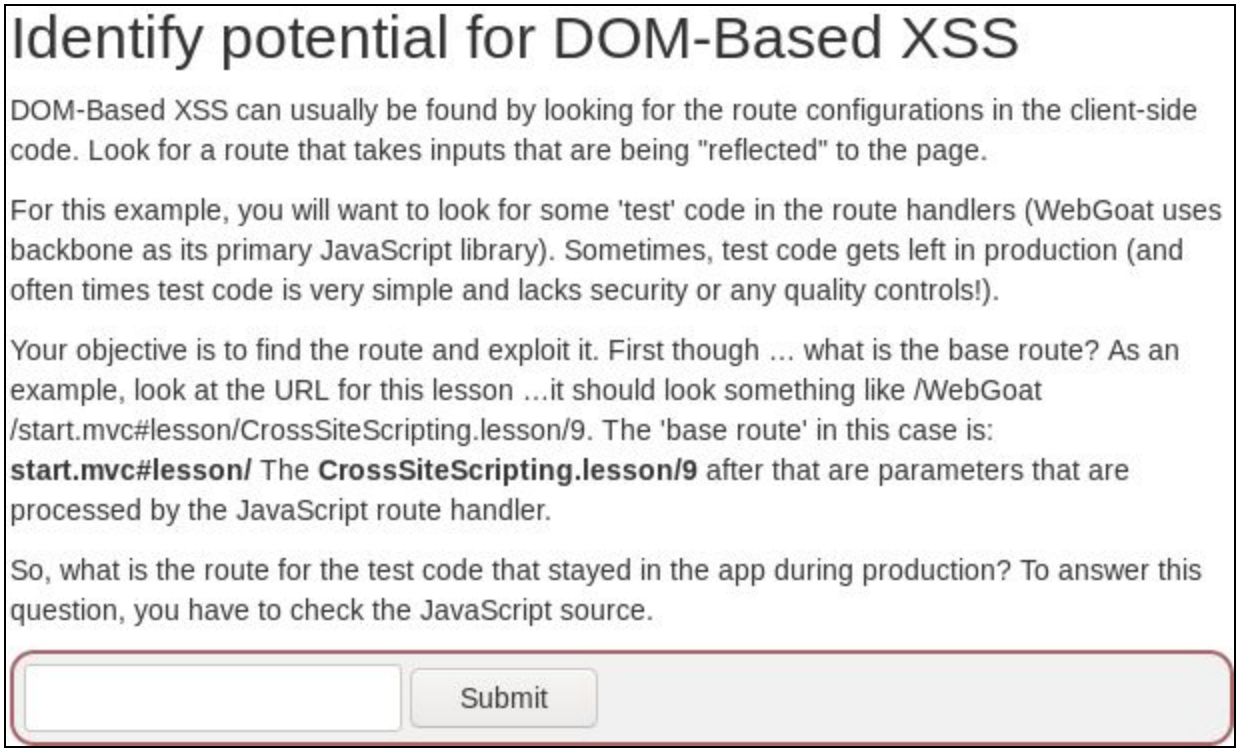
You'll find out quick that all the quantity inputs are checked after WebGoat yell at you with "InPuT mUsT bE a NuMbEr" or something along those lines. Which narrows it down to two inputs: The card number and the wacky 3 digit input.

So now we carefully fuzz each input such as to not break anything… because we're professionals…



So professional

10. Let's see what other hoops OWASP is going to make us jump through:



So they want us to dig through directories and code… Luckily i've played a fuck ton of Minecraft in my day so I guess you could say I'm quite fond of digging.

They mention finding routes so naturally the best place to look would be a script that contains the name "route" somewhere in it:

Well shit…

Now we dig through stuff we don't understand until:

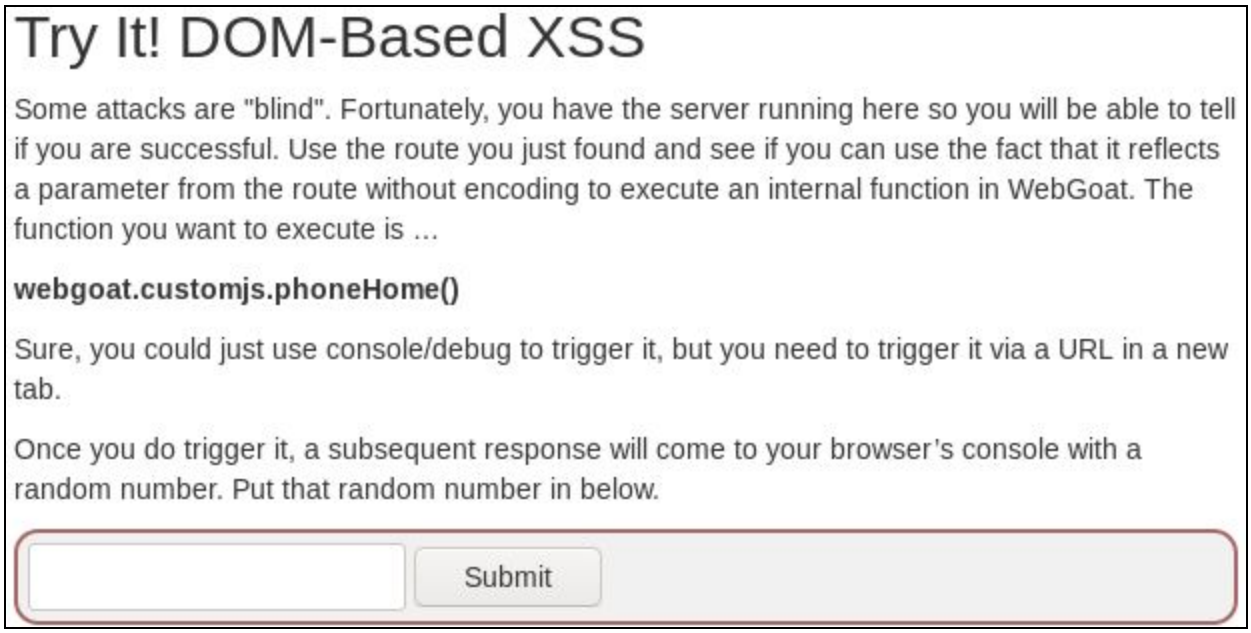
```
var GoatAppRouter = Backbone.Router.extend({

    routes: {
        'welcome': 'welcomeRoute',
        'lesson/:name': 'lessonRoute',
        'lesson/:name/:pageNum': 'lessonPageRoute',
        'test/:param': 'testRoute',
        'reportCard': 'reportCard'
    },
```

This is where shit gets a little bit weird, because instead of using the value "**testRoute**" we instead use the actual route instead: "**test/**" (note the lack of param here, we're only looking for the route)

With that said we can plug this into WebGoat and give them what they want:

Big thing to note here: **MVC**(Model, View, and Controller) is the framework WebGoat uses to create the user interface (it's some sort of ASP thing), so in order to actually make the route work correctly we have to instantiate it with **start.mvc#**.

11. Building on what we learned in the last challenge:



So pretty much the same thing as we did in the first challenge, except now we're building onto it:

```
localhost:8080/WebGoat/start.mvc#test/%3Cscript%3Ewebgoat.customjs.phoneHome()%3C%2Fscript%3E
```

There's quite a bit going on in this URL, but It's nothing we haven't seen before. The only curve ball here is the **percent encoding**. In short, percent encoding is the method used in URL's to not cause the server to think it's looking at code when parsing, it's a bit like escape characters If you have ever written or looked at code before. Without percent encoding this is our URL:

```
localhost:8080/WebGoat/start.mvc#test/<script>webgoat.customjs.phoneHome()</script>
```

Note: Because i'm lazy I used this site to percent encode my script tags: https://www.url-encode-decode.com/

So with our test route we found in the previous challenge, all we do is tell the server that this is javascript with our script tags and call that function they told us about in the example.

Now once we run that URL and we check our console:

```
test handler
phoneHome invoked
phone home said {"lessonCompleted":true,"feedback":"Congratulations. You have successfully completed the assignment.","output":"phoneHome Response is -1788503414"}
```

Sweet sweet payday

Hopefully this gave some good insight to how XSS vulnerabilities are exploited and how URL's work. If you want to see me do these challenges live be sure to drop by my Twitch when I'm live and also follow my Twitter for some fresh memes!