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 us insert some javascript code into our URL input:

Try It! Using Chrome or Firefox

- Open a second tab and use the same url as this page you are currently on (or any url within this instance of WebGoat)

- Then, in the address bar on each tab, type `javascript:alert(document.cookie);` **NOTE:** If you /cut/paste you will need to add the `javascript:` back in.

Were the cookies the same on each tab? [_____] [Submit]

So when we do that we just get this action:
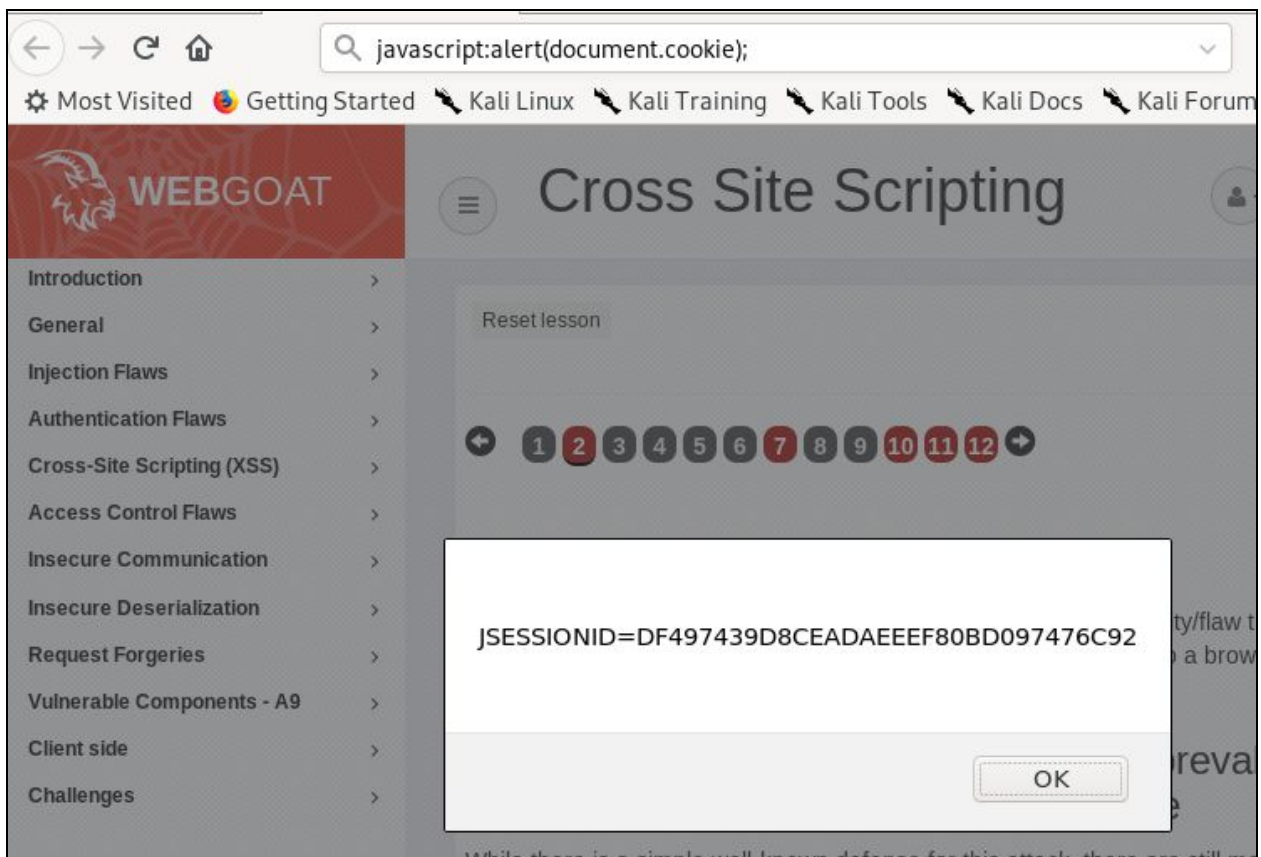
Likewise if we do the same on our other tab we get the same result:

JSESSIONID=DF497439D8CEADAEEEF80BD097476C92

OK

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

Were the cookies the same on each tab? | yes | Submit

Congratulations. You have successfully completed the assignment.

7.   Onto the next challenge then:

Try It! Reflected XSS

Identify which field is susceptible to XSS

It is always a good practice to validate all input on the server side. XSS can occur when unvalidated user input is used in an HTTP response. In a reflected XSS attack, an attacker can craft a URL with the attack script and post it to another website, email it, or otherwise get a victim to click on it.

An easy way to find out if a field is vulnerable to an XSS attack is to use the *alert()* or *console.log()* methods. Use one of them to find out which field is vulnerable.

Shopping Cart

Shopping Cart Items -- To Buy Now	Price	Quantity		Total
Studio RTA - Laptop/Reading Cart with Tilting Surface - Cherry	69.99	1		$0.00
Dynex - Traditional Notebook Case	27.99	1		$0.00
Hewlett-Packard - Pavilion Notebook with Intel Centrino	1599.99	1		$0.00
3 - Year Performance Service Plan $1000 and Over	299.99	1		$0.00

The total charged to your credit card: $0.00   UpdateCart

Enter your credit card number: 4128 3214 0002 1999

Enter your three digit access code: 111

Purchase

To test each of these inputs I used this string:

`<script>javascript:alert("Get Fucked");</script>`

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…
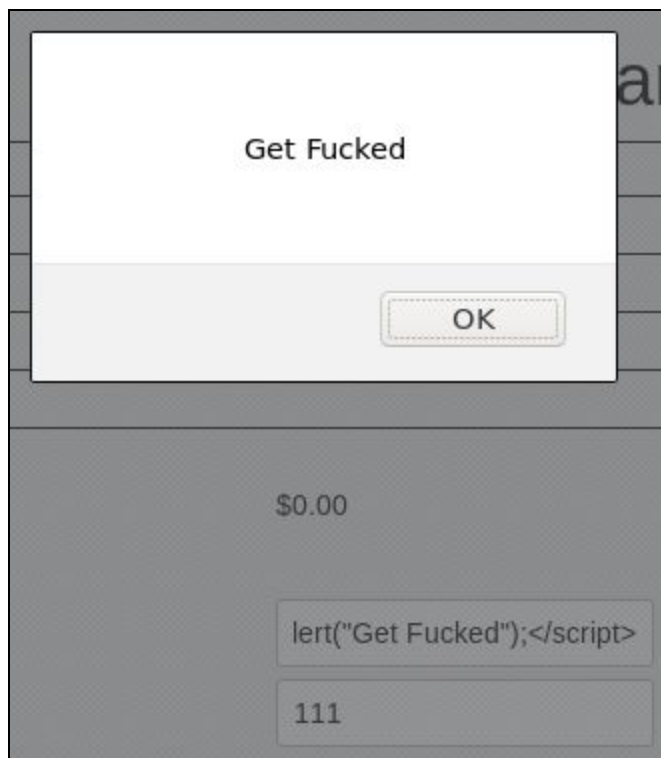
So professional

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

Identify potential for DOM-Based XSS

DOM-Based XSS can usually be found by looking for the route configurations in the client-side code. Look for a route that takes inputs that are being "reflected" to the page.

For this example, you will want to look for some 'test' code in the route handlers (WebGoat uses backbone as its primary JavaScript library). Sometimes, test code gets left in production (and often times test code is very simple and lacks security or any quality controls!).
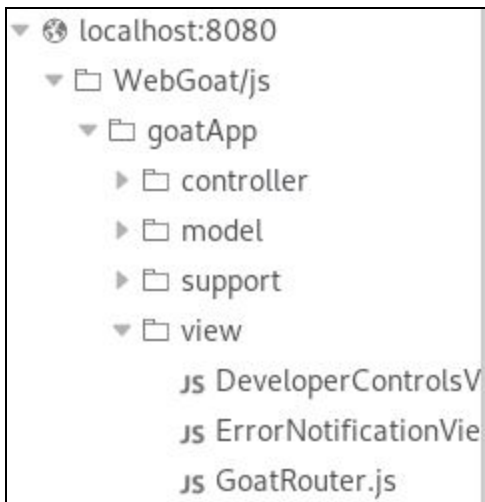
Your objective is to find the route and exploit it. First though … what is the base route? As an example, look at the URL for this lesson …it should look something like /WebGoat /start.mvc#lesson/CrossSiteScripting.lesson/9. The 'base route' in this case is: **start.mvc#lesson/** The **CrossSiteScripting.lesson/9** after that are parameters that are processed by the JavaScript route handler.

So, what is the route for the test code that stayed in the app during production? To answer this question, you have to check the JavaScript source.

[   ] [Submit]

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:

```
▼ 🌐 localhost:8080
  ▼ 🗁 WebGoat/js
    ▼ 🗁 goatApp
      ▶ 🗁 controller
      ▶ 🗁 model
      ▶ 🗁 support
      ▼ 🗁 view
          Js DeveloperControlsV
          Js ErrorNotificationVie
          Js GoatRouter.js
```

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:

✔

| start.mvc#test/ | Submit |

Correct! Now, see if you can send in an exploit to that route in the next assignment.

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:

Try It! DOM-Based XSS

Some attacks are "blind". Fortunately, you have the server running here so you will be able to tell if you are successful. Use the route you just found and see if you can use the fact that it reflects a parameter from the route without encoding to execute an internal function in WebGoat. The function you want to execute is ...

webgoat.customjs.phoneHome()

Sure, you could just use console/debug to trigger it, but you need to trigger it via a URL in a new tab.

Once you do trigger it, a subsequent response will come to your browser's console with a random number. Put that random number in below.

| | Submit |

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!