Lab #4

Matthew Langlois - 7731813

Nov. 27

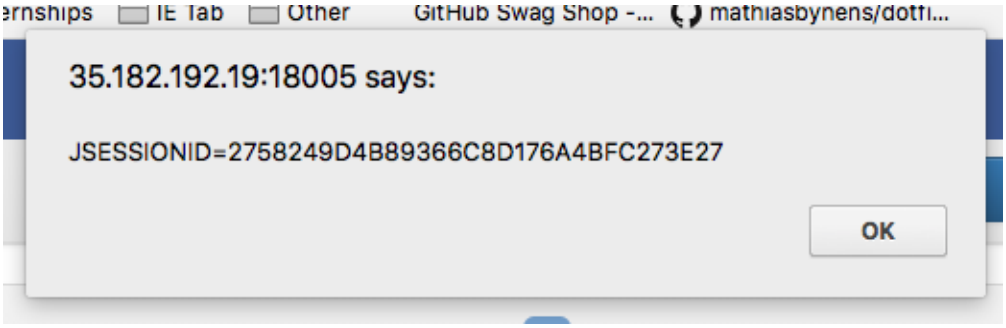
---

1. The first vulnerability was very straight forward. There was a reflected XSS in the search parameter. To exploit this vulnerability the XSS injection just needed to be placed in the query parameter of the URL.

   The code used was: `<script>alert(document.cookie)</script>`

   The vulnerability could have been prevented by sanitizing the user's input and ensuring that they are not injecting Javascript

   Here's an example of the vulnerability being exploited:

2. The second vulnerability was also quite straight forward. There was a stored XSS in the post section of Facebroke. To exploit this vulnerability the XSS injection just needed to be placed in the contents of the post.

   The code used was: `<script>alert('I know xss')</script>`

   The vulnerability could have been prevented by sanitizing the user's input and ensuring that they are not injecting Javascript.

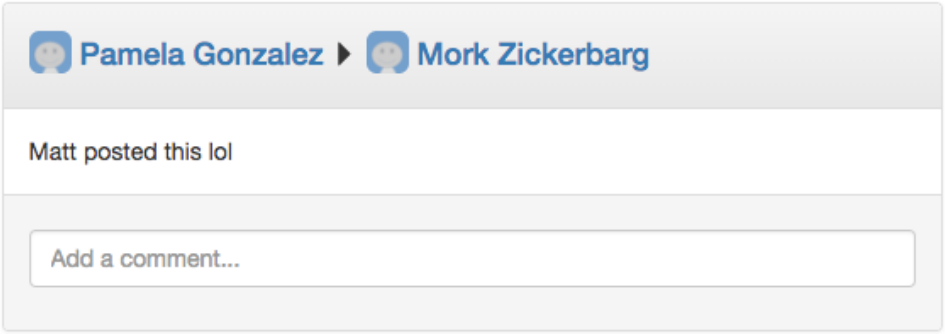
   Here's an example of the vulnerability being exploited:

3. The third vulnerability made use of modifying a POST request as it was in transit. To complete this task I setup a simple proxy using Burp to intercept the post message request. Inside the post request there was a variable representing the user's ID. This could be changed to be another user thus posting it on their behalf.

   No code required modifications for this vulnerability. Instead the POST request required modification.

   The vulnerability could have been prevented by using proper access control at the endpoints. In this case the user posting the comment should have been checked against the user which is currently logged in and sending the request. To do this the ID of the user should be stored in their session which can be identified by the cookie. Then the session can be compared against the id in the request.

   Here's an example of the vulnerability being exploited:

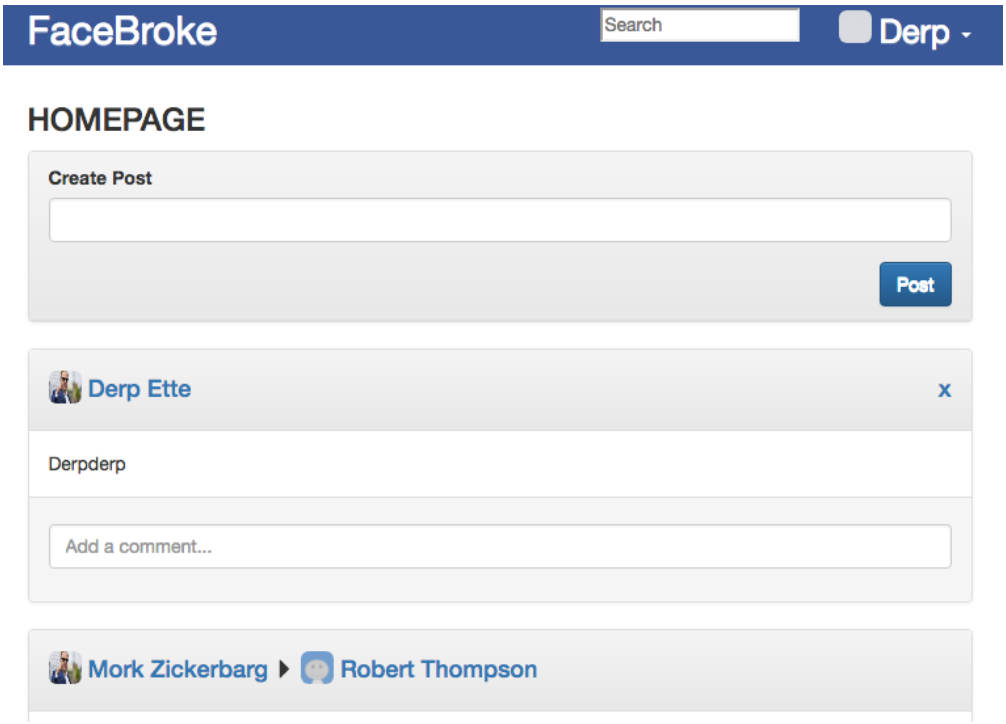


4. The 4th vulnerability made use of spoofing the user's ID again. When uploading an image the superuser ID 103 could be set thus allowing the image to be uploaded on behalf of another user.

   Similar to the last vulnerability, the exploit required no code but rather modifying a POST request.

Much like the last exploit this one could have been prevented with proper access control and authorization checks on the endpoint.

Here's an example of the vulnerability being exploited:

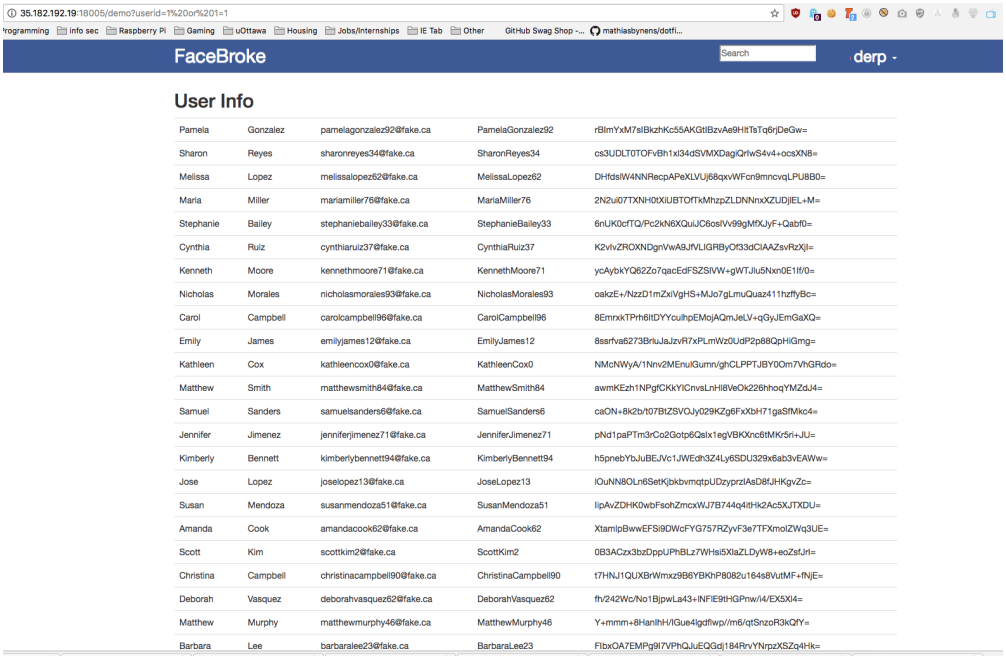


5. The 5th vulnerability made use of a very basic SQL injection attack in the query parameter of the get request.

   To exploit this vulnerability all that was required was setting the userid parameter to `1 or 1=1--`.

   The vulnerability could have been prevented by sanitizing the user's input. Furthermore prepared statements could have been used to ensure that SQL injection isn't possible.

Here's an example of the vulnerability being exploited:



6. The last vulnerability made use of an XSS within an SVG file. The SVG spec allows the use of script tags. When the browser renders the SVG the vm notices the Javascript and executes it. Thus, in this case, popping up the alert box.

   The code used was:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
 ↪  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg version="1.1" baseProfile="full"
 ↪  xmlns="http://www.w3.org/2000/svg">
  <polygon id="triangle" points="0,0 0,50 50,0" fill="#009900"
   ↪  stroke="#004400"/>
  <script type="text/javascript">
    alert('This app is probably vulnerable to XSS attacks!');
  </script>
</svg>
```

   The vulnerability could have been prevented by disallowing SVG files from being uploaded for display pictures. However if SVG is absolutely required then the uploaded file should be sanitized to ensure that there are no script tags.

Here's an example of the vulnerability being exploited: