

WAPH-Web Application Programming and Hacking

Instructor: Dr. Phu Phung

Student

Name: Ruthvik Suvarnakanti

Email: suvarnrk@mail.uc.edu

Hackathon 1: Cross-Site Scripting Attacks and Defenses

Overview: The Hackathon centers around understanding and addressing XSS (Cross-Site Scripting) attacks, with participants engaging in two distinct tasks. In the Attack Phase (Task 1), participants are tasked with uncovering and exploiting XSS vulnerabilities within a designated website (<http://waph-hackathon.eastus.cloudapp.azure.com/xss/>). This site is structured with six levels of vulnerabilities, designed to provide insight into the mechanics of XSS attacks. Following this, in the Defense Phase (Task 2), participants focus on implementing secure coding practices to mitigate XSS threats. Guided by OWASP guidelines, they learn techniques such as input validation and output sanitization to fortify their applications against XSS vulnerabilities effectively. Upon completing both phases, participants document their discoveries and solutions using Markdown format. Utilizing tools like Pandoc, they generate a comprehensive PDF report detailing their experiences, including the vulnerabilities identified, exploitation methods employed, and the strategies adopted to mitigate risks. Overall, the Hackathon serves as a practical learning platform, offering participants hands-on experience in identifying, exploiting, and safeguarding against XSS vulnerabilities, thus fostering a deeper understanding of web security protocols.

Link to the repository: <https://github.com/suvarnrk/waph-suvarnrk/blob/main/labs/hackathon1/README.md>



Figure 1: Ruthvik Suvarnakanti

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 Ruthvik Suvarnakanti")</script>
```

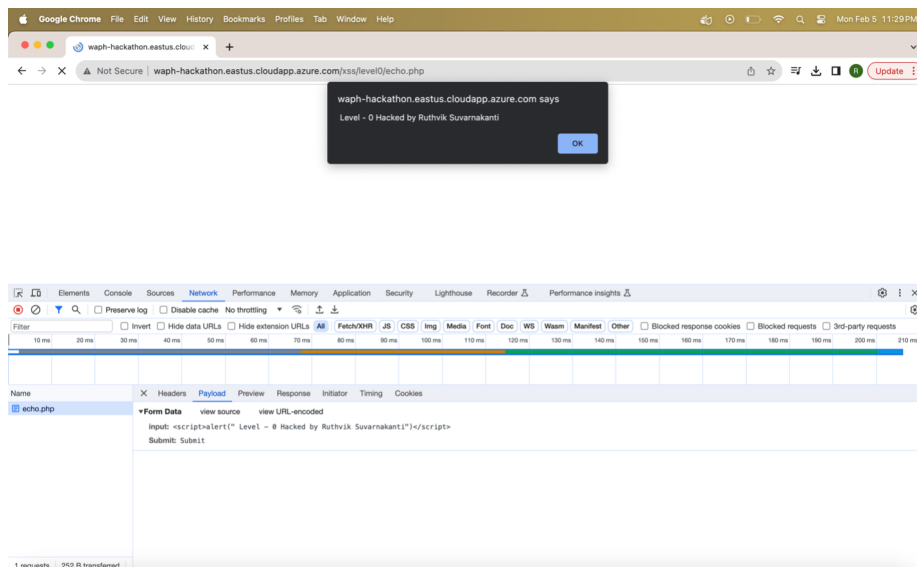


Figure 2: Level 0

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 Ruthvik Suvarnakanti")</script>

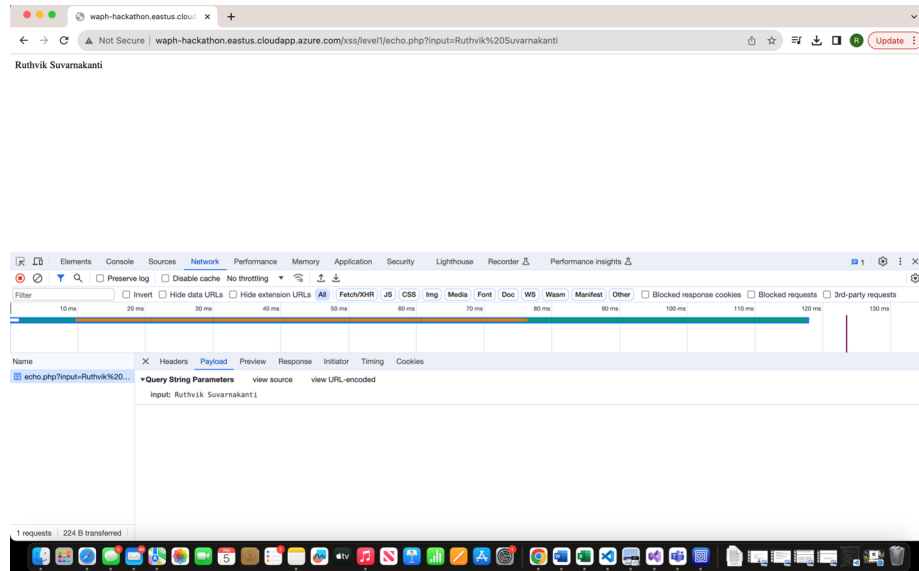


Figure 3: Level 1

Level 2

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

Since the HTTP request for this URL doesn't provide an input field or accept path variables, a workaround was needed. The URL was linked to a simple HTML `<form>`. Through this form, the attacking script is transmitted. This method allows for a more structured and controlled injection of malicious scripts, aiding in the exploration and exploitation of XSS vulnerabilities within the web application. By embedding the attacking script directly into the form submission, participants can interact with the website and observe how their injected code affects it. This approach ensures that XSS attacks are carried out within the web application's environment, giving participants a clearer understanding of the vulnerabilities and their potential consequences for the application's security.

```
<script>alert("Level 2: Hacked by Ruthvik Suvarnakanti")</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'];
```

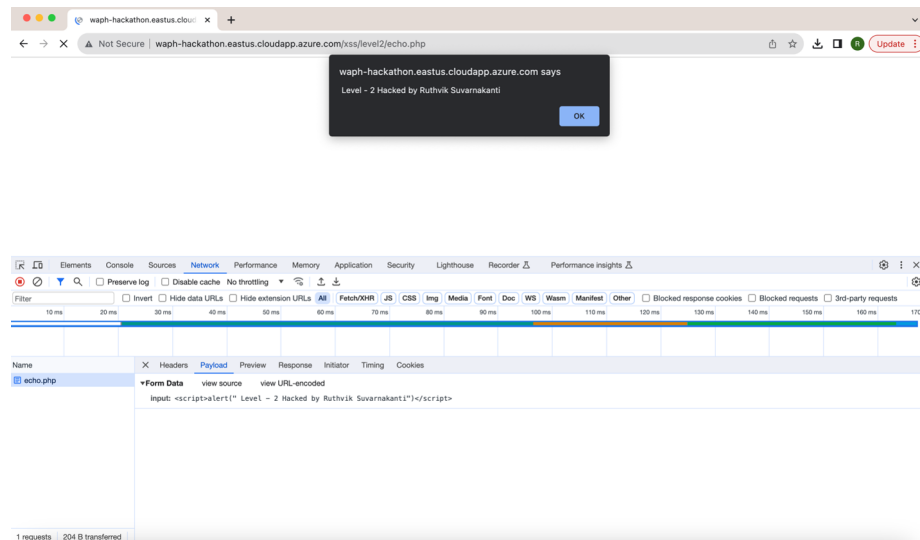


Figure 4: Level 2

Level 3

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

In this level, if the `<script>` tag is directly passed in the input variable, it is filtered out. Therefore, to carry out an attack on this URL, the malicious code had to be divided into multiple parts and then appended together to trigger an alert on the webpage. This approach bypasses the filtering mechanism and allows the attacker to execute their payload successfully.

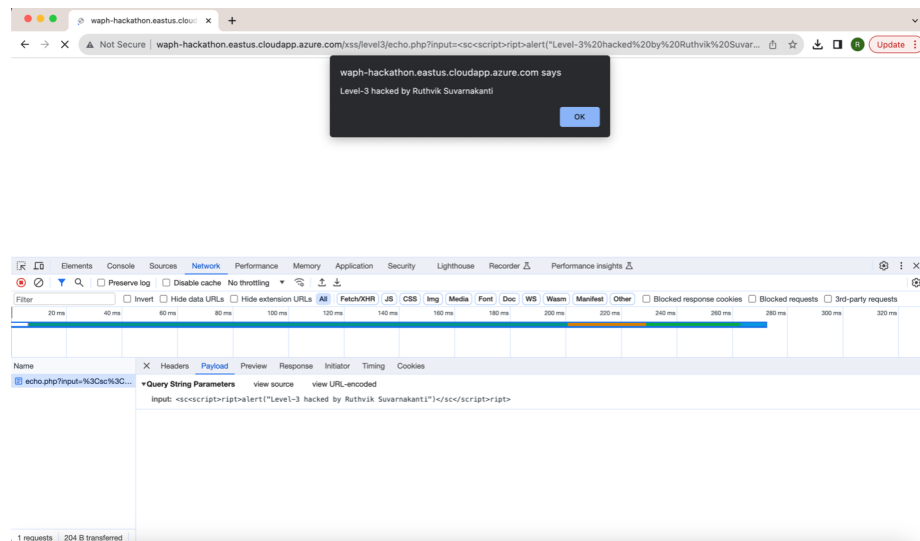


Figure 5: Level 3

?input=<script<script>>alert("Hacked by Ruthvik Suvarnakanti")</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>

In this level, the filtering mechanism completely blocks the `<script>` tag, even if it's attempted by breaking the string and concatenating it. To inject the XSS script, I utilized the `onerror()` attribute of the `` tag. By leveraging this attribute, I triggered an alert to be raised on the webpage. This approach circumvents the filtering mechanism, allowing the XSS script to execute successfully.

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

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);
```

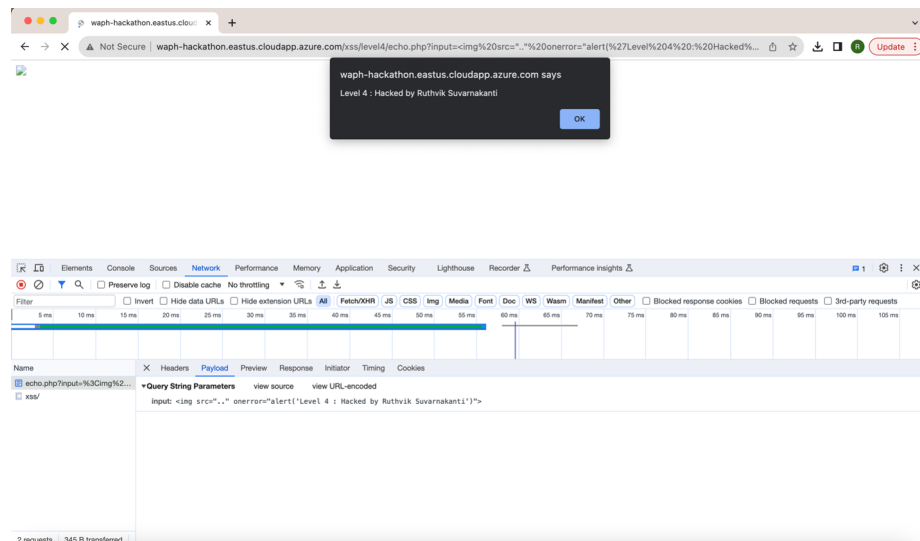


Figure 6: Level 4

Level 5

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

In this level, both the `<script>` tag and the `alert()` method are filtered out. To trigger a popup alert, I employed a combination of Unicode encoding and the `onerror()` method of the `` tag. By utilizing these techniques together, I successfully raised a popup alert on the webpage. This method effectively bypasses the filtering mechanisms in place, allowing the XSS payload to execute.

```
?input=
```

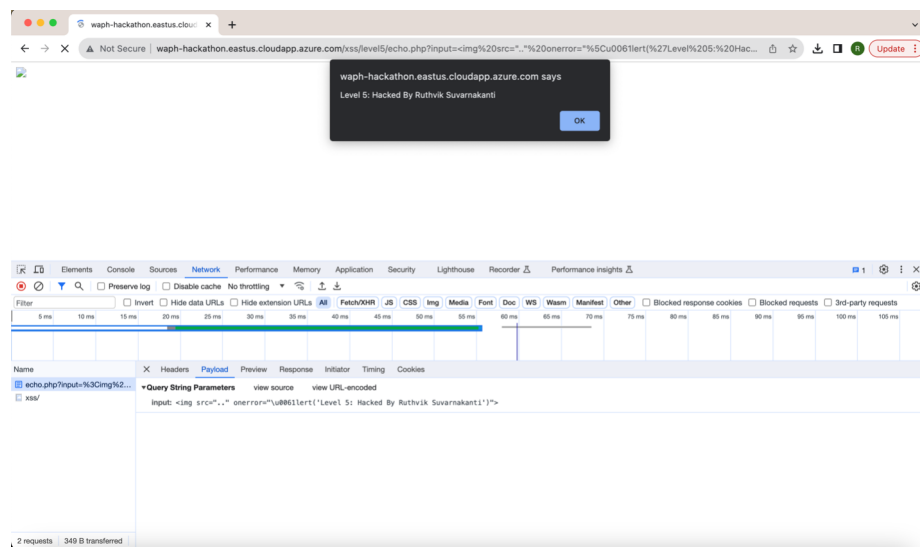


Figure 7: Level 5

Level 6

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

In this level, user input is accepted, but it appears that the source code utilizes the `htmlspecialchars()` method to convert all relevant characters into their corresponding HTML entities. This ensures that the user input is rendered strictly as text on the webpage. To trigger an alert on the webpage in this scenario, I utilized JavaScript event listeners such as `onmouseover()`, `onclick()`, and `onkeyup()`. Specifically, I opted for the `onkeyup()` event listener, which generates the alert on the webpage whenever a key is pressed within the input field. This approach allowed me to bypass the HTML entity conversion and successfully execute the desired action.

```
/" onkeyup="alert('Level 6 : Hacked by Ruthvik Suvarnakanti')"
```

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 Ruthvik Suvarnakanti')" method="POST">
  Input:<input type="text" name="input" />
  <input type="submit" name="Submit"/>
```

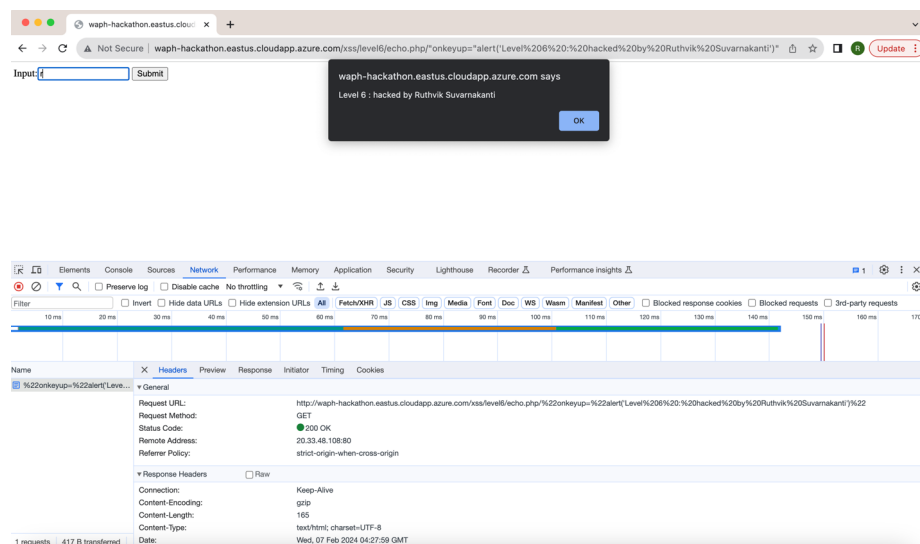


Figure 8: Level 6

source code guess of echo.php:

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

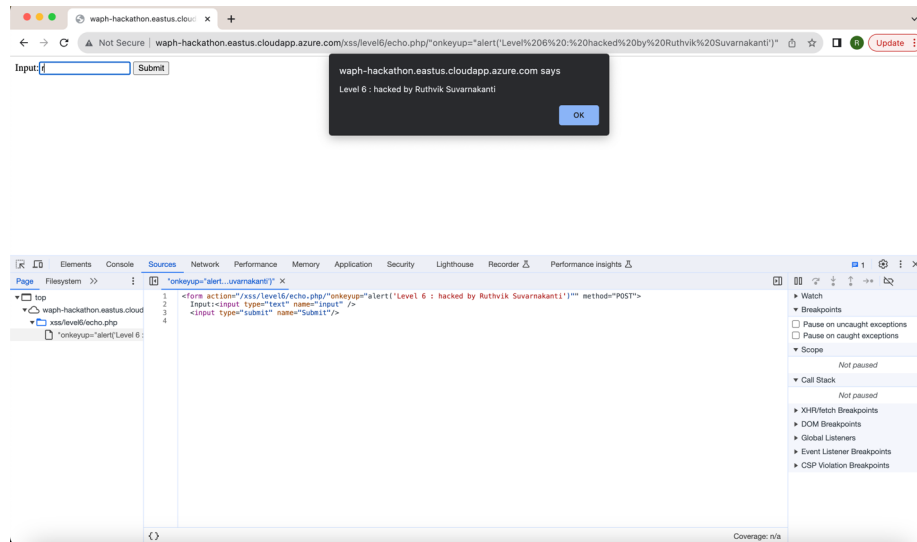


Figure 9: Level 6 after injecting XSS code

TASK 2 : DEFENSE

A . echo.php

The `echo.php` file in Lab 1 has been updated to include input validation and XSS defense measures. Initially, it checks whether the input is empty. If it is, the PHP execution stops. If the input is valid, the `htmlspecialchars()` function is utilized to sanitize the input data. This function converts special characters into their HTML entities, ensuring that the text is displayed as plain text on the webpage. This action serves to protect against XSS attacks by neutralizing any potentially harmful scripts embedded within the input.

```
if(empty($_REQUEST["data"])){
    exit("please enter the input field 'data'");
}
$input=htmlspecialchars($_REQUEST["data"]);
echo ("The input from the request is <strong>". $input. "</strong>.<br>");
```

```
1 <?php
2 - $data = $_REQUEST["input"];
3 - echo $data;
4 +
5 + $inputData = $_REQUEST["data"];
6 + echo "The requested input is <strong>". $inputData . "</strong>.<br>";
7 +
8 +
9 + $input = htmlspecialchars($_REQUEST["data"]);
10 + echo "The sanitized input from the request is <strong>". $input . "</strong>.<br>";
11 +
12 + echo "The input from the request is: ". $input;
```

Figure 10: Defense echo.php

B . Lab 2 front-end part

The `waph-suvarnrk.html` file underwent a thorough review and update, focusing on identifying and securing areas where external input is accepted. Validation procedures were implemented to ensure the integrity of input data, while measures were taken to sanitize output text to enhance security. i) In particular, when dealing with HTTP GET and POST request forms, input data is meticulously validated. A new function named `validateInput()` was introduced for this purpose. This function mandates that users provide input text before executing their request. This precautionary measure significantly reduces the risk of receiving invalid or potentially malicious input, thereby fortifying the overall security posture of the application.

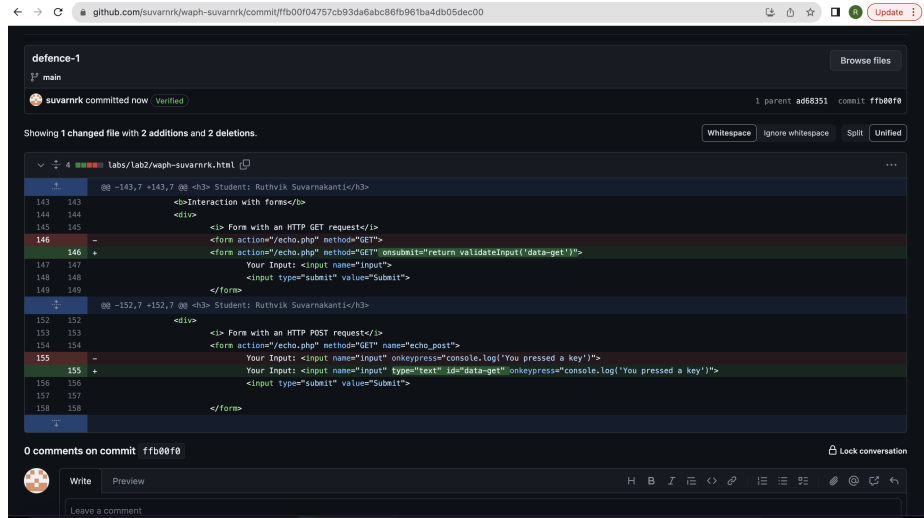


Figure 11: Defense waph-suvarnrk.html

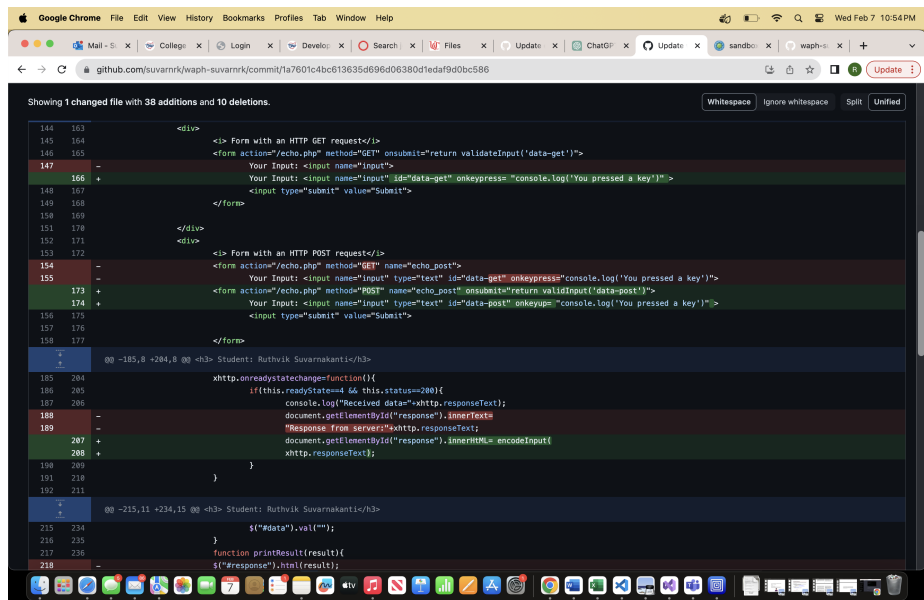
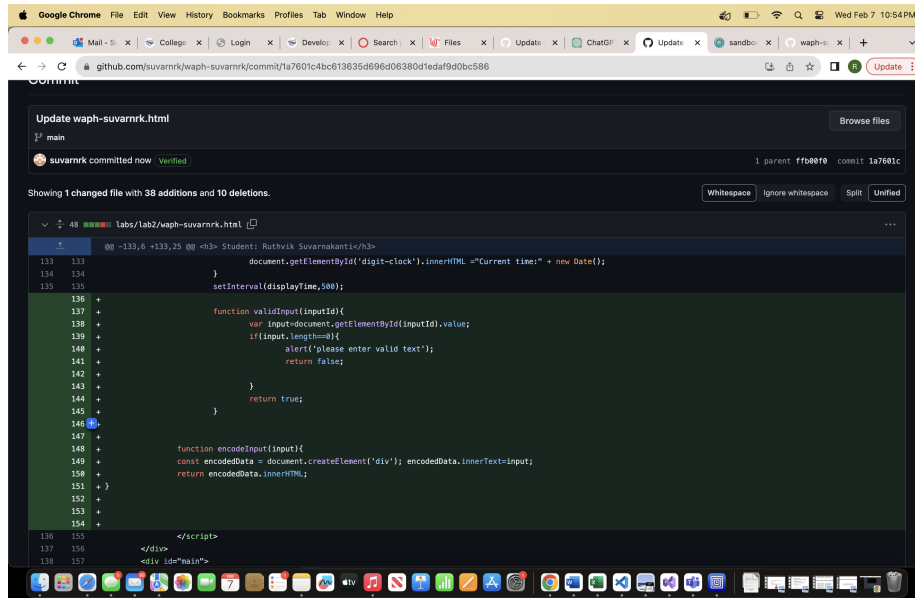


Figure 12: Validating HTTP requests input

ii) In the code, we made a change from using `.innerHTML` to `.innerText` in situations where HTML rendering wasn't necessary, and only plain text needed to be shown. This adjustment ensures that any text content is treated strictly as text, without any interpretation of HTML tags or rendering of elements. By using `.innerText`, we reduce the risk of unintended HTML injection or XSS vulnerabilities, making the display of text content safer within the application.



The screenshot shows a web browser window displaying a GitHub commit diff for the file `waph-suvarnrk.html`. The commit is by `suvarnrk` and is titled "Update waph-suvarnrk.html". The diff shows a single file change with 38 additions and 10 deletions. The code is shown in a dark-themed editor. The change is in the `encodeInput` function, where `encodedData.innerHTML` is replaced with `encodedData.innerText`. The diff is as follows:

```
133 | document.getElementById('digit-clock').innerHTML += new Date();
134 | }
135 | setInterval(displayTime, 500);
136 |
137 | function validateInput(inputId) {
138 |     var input = document.getElementById(inputId).value;
139 |     if (input.length === 0) {
140 |         alert('Please enter valid text');
141 |         return false;
142 |     }
143 |     return true;
144 | }
145 |
146 | function encodeInput(input) {
147 |     const encodedData = document.createElement('div');
148 |     encodedData.innerText = input;
149 |     return encodedData.innerHTML;
150 | }
151 |
152 |
153 |
154 |
155 |
156 |
157 |
158 |
159 |
160 |
161 |
162 |
163 |
164 |
165 |
166 |
167 |
168 |
169 |
170 |
171 |
172 |
173 |
174 |
175 |
176 |
177 |
178 |
179 |
180 |
181 |
182 |
183 |
184 |
185 |
186 |
187 |
188 |
189 |
190 |
191 |
192 |
193 |
194 |
195 |
196 |
197 |
198 |
199 |
200 |
201 |
202 |
203 |
204 |
205 |
206 |
207 |
208 |
209 |
210 |
211 |
212 |
213 |
214 |
215 |
216 |
217 |
218 |
219 |
220 |
221 |
222 |
223 |
224 |
225 |
226 |
227 |
228 |
229 |
230 |
231 |
232 |
233 |
234 |
235 |
236 |
237 |
238 |
239 |
240 |
241 |
242 |
243 |
244 |
245 |
246 |
247 |
248 |
249 |
250 |
251 |
252 |
253 |
254 |
255 |
256 |
257 |
258 |
259 |
260 |
261 |
262 |
263 |
264 |
265 |
266 |
267 |
268 |
269 |
270 |
271 |
272 |
273 |
274 |
275 |
276 |
277 |
278 |
279 |
280 |
281 |
282 |
283 |
284 |
285 |
286 |
287 |
288 |
289 |
290 |
291 |
292 |
293 |
294 |
295 |
296 |
297 |
298 |
299 |
300 |
301 |
302 |
303 |
304 |
305 |
306 |
307 |
308 |
309 |
310 |
311 |
312 |
313 |
314 |
315 |
316 |
317 |
318 |
319 |
320 |
321 |
322 |
323 |
324 |
325 |
326 |
327 |
328 |
329 |
330 |
331 |
332 |
333 |
334 |
335 |
336 |
337 |
338 |
339 |
340 |
341 |
342 |
343 |
344 |
345 |
346 |
347 |
348 |
349 |
350 |
351 |
352 |
353 |
354 |
355 |
356 |
357 |
358 |
359 |
360 |
361 |
362 |
363 |
364 |
365 |
366 |
367 |
368 |
369 |
370 |
371 |
372 |
373 |
374 |
375 |
376 |
377 |
378 |
379 |
380 |
381 |
382 |
383 |
384 |
385 |
386 |
387 |
388 |
389 |
390 |
391 |
392 |
393 |
394 |
395 |
396 |
397 |
398 |
399 |
400 |
401 |
402 |
403 |
404 |
405 |
406 |
407 |
408 |
409 |
410 |
411 |
412 |
413 |
414 |
415 |
416 |
417 |
418 |
419 |
420 |
421 |
422 |
423 |
424 |
425 |
426 |
427 |
428 |
429 |
430 |
431 |
432 |
433 |
434 |
435 |
436 |
437 |
438 |
439 |
440 |
441 |
442 |
443 |
444 |
445 |
446 |
447 |
448 |
449 |
450 |
451 |
452 |
453 |
454 |
455 |
456 |
457 |
458 |
459 |
460 |
461 |
462 |
463 |
464 |
465 |
466 |
467 |
468 |
469 |
470 |
471 |
472 |
473 |
474 |
475 |
476 |
477 |
478 |
479 |
480 |
481 |
482 |
483 |
484 |
485 |
486 |
487 |
488 |
489 |
490 |
491 |
492 |
493 |
494 |
495 |
496 |
497 |
498 |
499 |
500 |
501 |
502 |
503 |
504 |
505 |
506 |
507 |
508 |
509 |
510 |
511 |
512 |
513 |
514 |
515 |
516 |
517 |
518 |
519 |
520 |
521 |
522 |
523 |
524 |
525 |
526 |
527 |
528 |
529 |
530 |
531 |
532 |
533 |
534 |
535 |
536 |
537 |
538 |
539 |
540 |
541 |
542 |
543 |
544 |
545 |
546 |
547 |
548 |
549 |
550 |
551 |
552 |
553 |
554 |
555 |
556 |
557 |
558 |
559 |
560 |
561 |
562 |
563 |
564 |
565 |
566 |
567 |
568 |
569 |
570 |
571 |
572 |
573 |
574 |
575 |
576 |
577 |
578 |
579 |
580 |
581 |
582 |
583 |
584 |
585 |
586 |
587 |
588 |
589 |
590 |
591 |
592 |
593 |
594 |
595 |
596 |
597 |
598 |
599 |
600 |
601 |
602 |
603 |
604 |
605 |
606 |
607 |
608 |
609 |
610 |
611 |
612 |
613 |
614 |
615 |
616 |
617 |
618 |
619 |
620 |
621 |
622 |
623 |
624 |
625 |
626 |
627 |
628 |
629 |
630 |
631 |
632 |
633 |
634 |
635 |
636 |
637 |
638 |
639 |
640 |
641 |
642 |
643 |
644 |
645 |
646 |
647 |
648 |
649 |
650 |
651 |
652 |
653 |
654 |
655 |
656 |
657 |
658 |
659 |
660 |
661 |
662 |
663 |
664 |
665 |
666 |
667 |
668 |
669 |
670 |
671 |
672 |
673 |
674 |
675 |
676 |
677 |
678 |
679 |
680 |
681 |
682 |
683 |
684 |
685 |
686 |
687 |
688 |
689 |
690 |
691 |
692 |
693 |
694 |
695 |
696 |
697 |
698 |
699 |
700 |
701 |
702 |
703 |
704 |
705 |
706 |
707 |
708 |
709 |
710 |
711 |
712 |
713 |
714 |
715 |
716 |
717 |
718 |
719 |
720 |
721 |
722 |
723 |
724 |
725 |
726 |
727 |
728 |
729 |
730 |
731 |
732 |
733 |
734 |
735 |
736 |
737 |
738 |
739 |
740 |
741 |
742 |
743 |
744 |
745 |
746 |
747 |
748 |
749 |
750 |
751 |
752 |
753 |
754 |
755 |
756 |
757 |
758 |
759 |
760 |
761 |
762 |
763 |
764 |
765 |
766 |
767 |
768 |
769 |
770 |
771 |
772 |
773 |
774 |
775 |
776 |
777 |
778 |
779 |
780 |
781 |
782 |
783 |
784 |
785 |
786 |
787 |
788 |
789 |
790 |
791 |
792 |
793 |
794 |
795 |
796 |
797 |
798 |
799 |
800 |
801 |
802 |
803 |
804 |
805 |
806 |
807 |
808 |
809 |
810 |
811 |
812 |
813 |
814 |
815 |
816 |
817 |
818 |
819 |
820 |
821 |
822 |
823 |
824 |
825 |
826 |
827 |
828 |
829 |
830 |
831 |
832 |
833 |
834 |
835 |
836 |
837 |
838 |
839 |
840 |
841 |
842 |
843 |
844 |
845 |
846 |
847 |
848 |
849 |
850 |
851 |
852 |
853 |
854 |
855 |
856 |
857 |
858 |
859 |
860 |
861 |
862 |
863 |
864 |
865 |
866 |
867 |
868 |
869 |
870 |
871 |
872 |
873 |
874 |
875 |
876 |
877 |
878 |
879 |
880 |
881 |
882 |
883 |
884 |
885 |
886 |
887 |
888 |
889 |
890 |
891 |
892 |
893 |
894 |
895 |
896 |
897 |
898 |
899 |
900 |
901 |
902 |
903 |
904 |
905 |
906 |
907 |
908 |
909 |
910 |
911 |
912 |
913 |
914 |
915 |
916 |
917 |
918 |
919 |
920 |
921 |
922 |
923 |
924 |
925 |
926 |
927 |
928 |
929 |
930 |
931 |
932 |
933 |
934 |
935 |
936 |
937 |
938 |
939 |
940 |
941 |
942 |
943 |
944 |
945 |
946 |
947 |
948 |
949 |
950 |
951 |
952 |
953 |
954 |
955 |
956 |
957 |
958 |
959 |
960 |
961 |
962 |
963 |
964 |
965 |
966 |
967 |
968 |
969 |
970 |
971 |
972 |
973 |
974 |
975 |
976 |
977 |
978 |
979 |
980 |
981 |
982 |
983 |
984 |
985 |
986 |
987 |
988 |
989 |
990 |
991 |
992 |
993 |
994 |
995 |
996 |
997 |
998 |
999 |
1000 |
```

Figure 13: modifying innerHTML to innerText

iii) A new function called `encodeInput()` was introduced to sanitize the response. This function converts special characters into their respective HTML entities before inserting them into the HTML document, thereby preventing cross-site scripting attacks. This ensures that the content is displayed purely as text and cannot be executed. Moreover, the code creates a new `<div>` element and inserts the content as `innerText` into this newly created element. Subsequently, this content is returned as the HTML content. This method guarantees that any potentially harmful content is properly encoded and displayed securely within the HTML document, thus reducing the risk of XSS vulnerabilities. Utilizing `innerText` ensures that the content is treated strictly as text, preventing any unintended HTML rendering or script execution.

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

iv) Additional validations have been introduced for the API `https://v2.jokeapi.dev/joke/Programming?type=single`, used to fetch jokes. These validations now verify if the received result and the `result.joke` in the JSON response are not empty. If either of these values turns out to be null or empty, an error message is generated. This enhancement ensures that only valid and non-empty joke data is handled and presented, thereby enhancing the application's reliability and user experience.

```
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.");
}
```

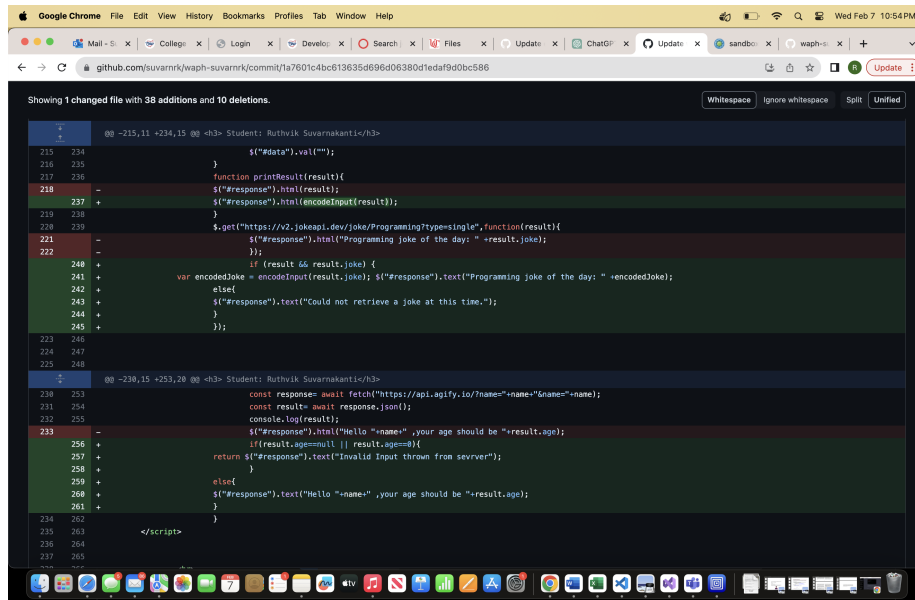


Figure 14: handling Joke API and Guess age API

v) In the asynchronous function `guessAge()`, new validations have been added. These validations ensure that both the received result and the user-entered input are not empty, null, or equal to 0. If either the result or the input is found to be in such a state, an error message is thrown. These measures are put in place to uphold data integrity and guarantee the proper functioning of the function across different scenarios, thereby improving its reliability and usability.

```
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);
```

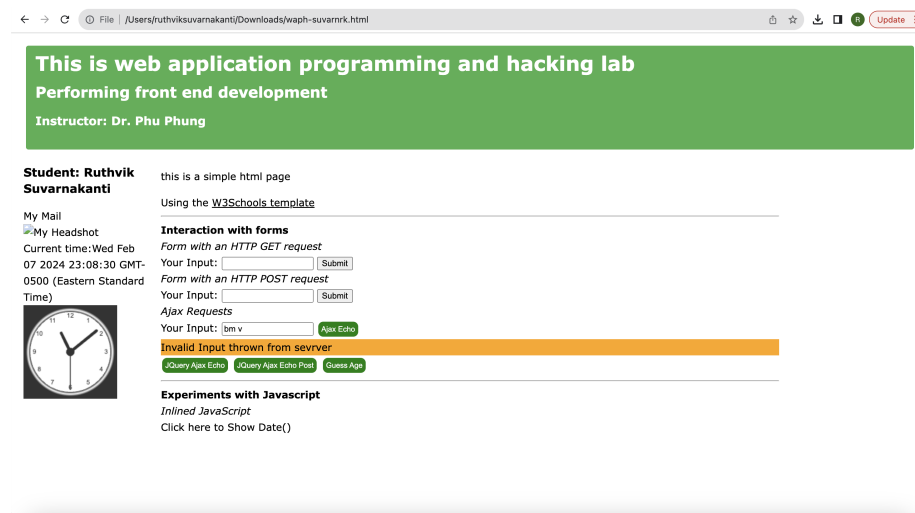


Figure 15: Guess age function in case error is thrown