Network Security (CS-473)

Homework 1

Documentation of Attacks and Vulnerability Defense Task

Name: Muhammad Raahim Khan

Roll Number: 21100157

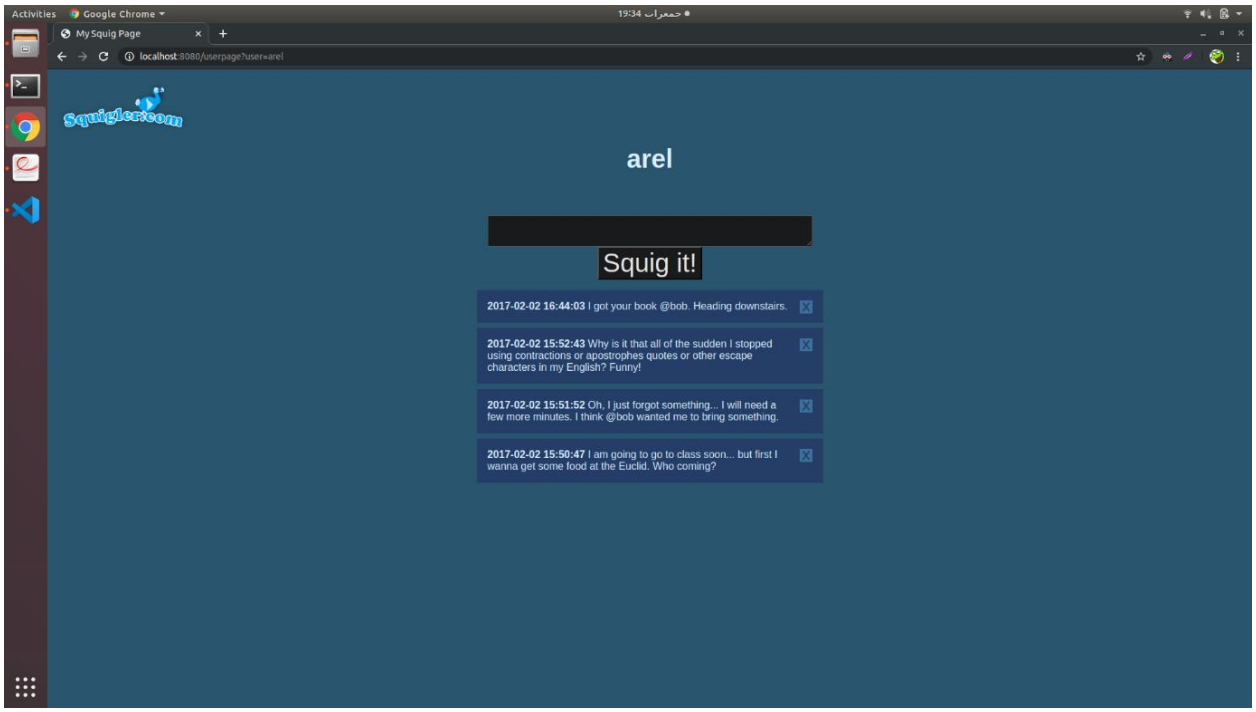
T. 1: SQL Injection

- **T. 1.1: Change a Users Password**

  - Input String Used

```
b' or 1=1; UPDATE accounts SET password='qwerty' WHERE username='arel';--
```

- Screenshot



- Explanation and Observation

On line 92 of the file webserver.py the sql query is as follows: **"SELECT password from accounts where username='%s'"**. This query is vulnerable to sql injection attack. I noticed that when I inputted test' in the username field, the server crashed. This is because the query changes to **SELECT password from accounts where username='test"**. Here the extra single quote (after test) causes the sql query to terminate early without executing properly, hence the server crashes. To utilize this scenario in my favor and to perform sql injection attack I made use of semicolons and sql comment (--) to craft my input string which as mentioned above as well is: **b' or 1=1; UPDATE accounts SET password='qwerty' WHERE username='arel';--**. This causes the sql query to change into: **SELECT password from accounts where username='b' or 1=1; UPDATE accounts SET password='qwerty' WHERE username='arel';-- '**. First query has two parts, either username = b or 1=1 (always true) hence this query executes. Second query then updates the password for 'arel' to 'qwerty' successfully. Finally, the sql comment (--) takes care of the extra single quote

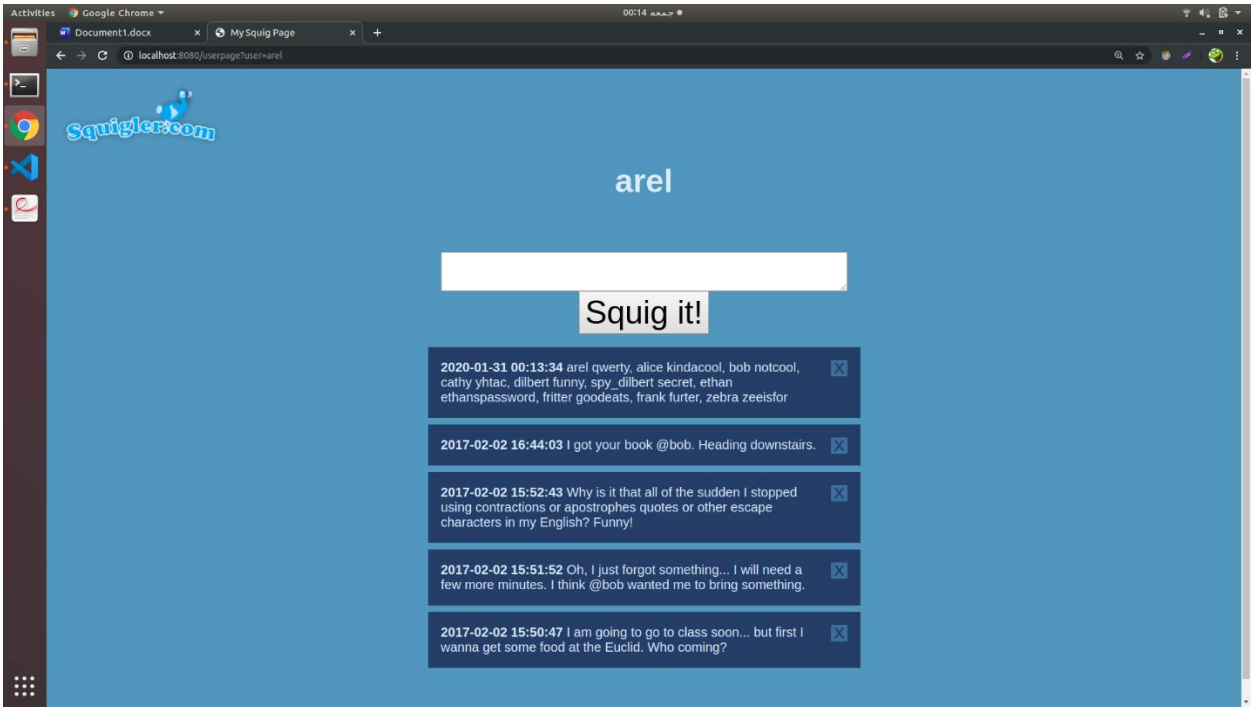
that was initially causing the server to crash. It should be noted that both the queries are separated by a semicolon.

- **T. 1.2: Password Dump through Post**

  - Input String Used:

```
' || (SELECT GROUP_CONCAT(username || ' ' || password, ',' || ' ') FROM accounts), datetime('now', 'localtime'));--
```

- Screenshot



- Explanation and Observation

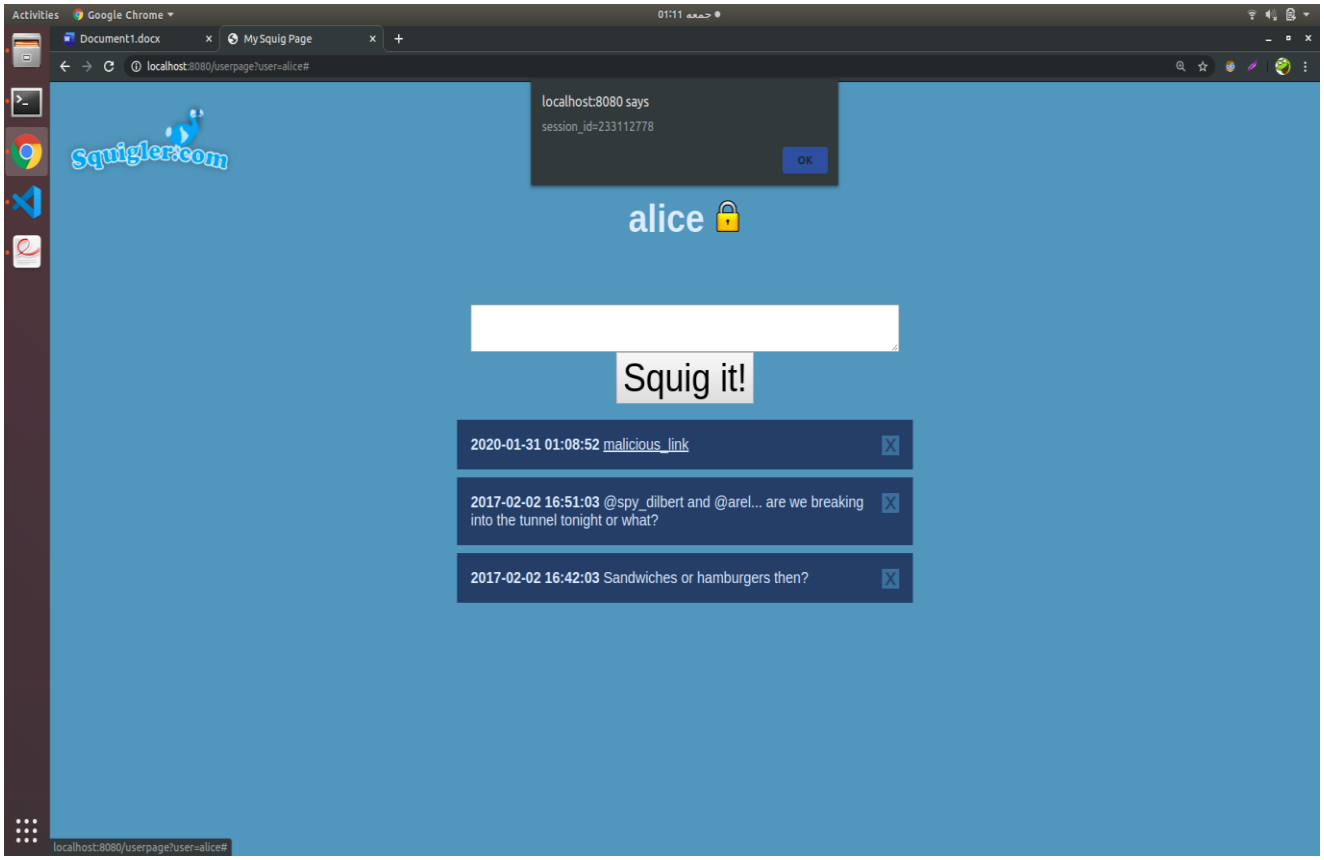
On line 126 of the file webserver.py the sql query is as follows: **"INSERT INTO squigs VALUES ('%s', '%s', datetime('now', 'localtime'))**. This sql statement has a vulnerability pretty like the one I exploited in task T.1.1. To figure out the vulnerability I again inputted single quote and tried to Squib it! As expected, the server crashed since the extra single quote cased the sql query to terminate early without executing successfully. Hence, I crafted the malicious input string as follows: **' || (SELECT GROUP_CONCAT(username || ' ' || password, ',' || ' ') FROM accounts), datetime('now', 'localtime'));--**. This causes the sql query to change into: **INSERT INTO squigs VALUES ('%s', '' || (SELECT GROUP_CONCAT(username || ' ' || password, ',' || ' ') FROM accounts), datetime('now', 'localtime'));-- ', datetime('now', 'localtime'));**. The sql comment (--) causes the extra single quote and the remaining of the sql statement to be ignored which makes my input post an entirely new sql statement. (||) is just a string concatenation operator whereas group_concat is an aggregate function that concatenates strings from a group into a single string with various options. What is happening in my input string is that empty string (‘’) is being concatenated with a sql query. Rest of the query is unchanged. As a result, a single concatenated string containing usernames and passwords is dumped through post.

T. 2: Extended CSRF

- Input String Used

```
<a href="#" onclick="alert(document.cookie);"><body
onload="alert(document.cookie);">malicious_link</a>
```

- Screenshot



- Explanation and Observation

This task was simple. As done in lab, first I made an anchor tag named as malicious_link. It should be noted that href=”#” simply scrolls back to the top of the page. Have we wanted the user to redirect to some other website, I would have inputted a link instead of #. And whenever the user clicks on the link, alert is called and session id or cookie of the user who clicked on the link is displayed on top of the page as shown in the screenshot attached. This attack could be thought of as both reflected and stored XSS in a way. For the purpose of CSRF attack, I included an addition script inside the anchor tag that is the ‘body onload’ script. Hence, every time the link is clicked, or page is visited/refreshed script will be executed and user session will be displayed in the form of an alert box. This attack is a combination of CSRF and reflected/ persistent XSS. The malicious script bounces off and executes into the victim’s browser as well as pops up every time the page is visited or refreshed. This way private information such as user cookies are stealed.

The attack was made possible due to many reasons such as unsanitized user input in the output.

T. 3: Vulnerability Defenses

- **SQL Injection**

To prevent SQL injection, I patched SQL queries and turned them into prepared SQL statements. I changed the SQL queries into prepared SQL statements for the following functions in the webserver.py code:

- get_all_public_users()
- get_user(session_id)
- logout_user(user)
- test_password(user, password)
- login_user(user)
- get_squigs(user)
- post_squig(user, squig)
- del_squig(user, time)
- search_squigs(query)

Though most of the functions did not require prepared SQL statements but I patched them anyways ensuring maximum protection from SQL injection attacks from every way possible. The problem with SQL injection is, that a user input is used as part of the SQL statement. This way an attacker exploits the user input by using combination of SQL comments, semi-colons to introduce completely new SQL queries that could do harmful actions. By using prepared statements, I forced the user input to be handled as the content of a parameter (and not as a part of the SQL command). All patches for prepared SQL queries follow similar pattern. I will write down about the one that were exploited in above tasks,

```
sql_select_query = """SELECT password from accounts where username = ?;"""
select_data = (user,)
db_password = c.execute(sql_select_query, select_data).fetchone()
```

Hence, now username and/or password fields cannot be exploited to change user passwords or other credentials to gain access to their squig accounts illegally.

Similarly following patch was introduced in the post_squig(user, squig) function

```
sql_insert_query = """INSERT INTO squigs VALUES (?, ?, datetime('now', 'localtime'));"""
insert_tuple = (user, squig)
c.execute(sql_insert_query, insert_tuple)
```

With inclusion of this patch, attacker cannot gain access to usernames and their passwords by posting malicious SQL queries in squigs.

- **XSS and CSRF**

Whenever HTML code is generated dynamically, and the user input is not sanitized and is reflected on the page an attacker could insert his own HTML code. The web browser will still show the user's code since it pertains to the website where it is injected. There are many methods to patch up XSS vulnerabilities. In some cases, it might be enough to encode the HTML special characters, such as opening and closing tags. In other cases, a correctly applied URL encoding is necessary. Links should generally be disallowed if they don't begin with a whitelisted protocol such as http:// or https://.

However, for the purpose of this assignment, as mentioned in the PDF file, we were only required to take care of <script> tags. To achieve this, I utilized the functions of BeautifulSoup4 (a python library). It sanitizes user input and filters out <script> tags from user input. Hence, hackers cannot inject JavaScript using <script> tags.

I applied sanitization to the following functions of server.py:

- post_squig(user, squig)
- search_squigs(query)

since I only found these vulnerable to XSS attacks (both reflected and stored).

Sanitization function is as follows (see next page):

```
def sanitize(value):
    VALID_TAGS = ['strong', 'em', 'p', 'ul', 'li', 'br']
    soup = BeautifulSoup(value, features='html.parser')

    for tag in soup.findAll(True):
        if tag.name not in VALID_TAGS:
            tag.hidden = True

    return soup.renderContents()
```

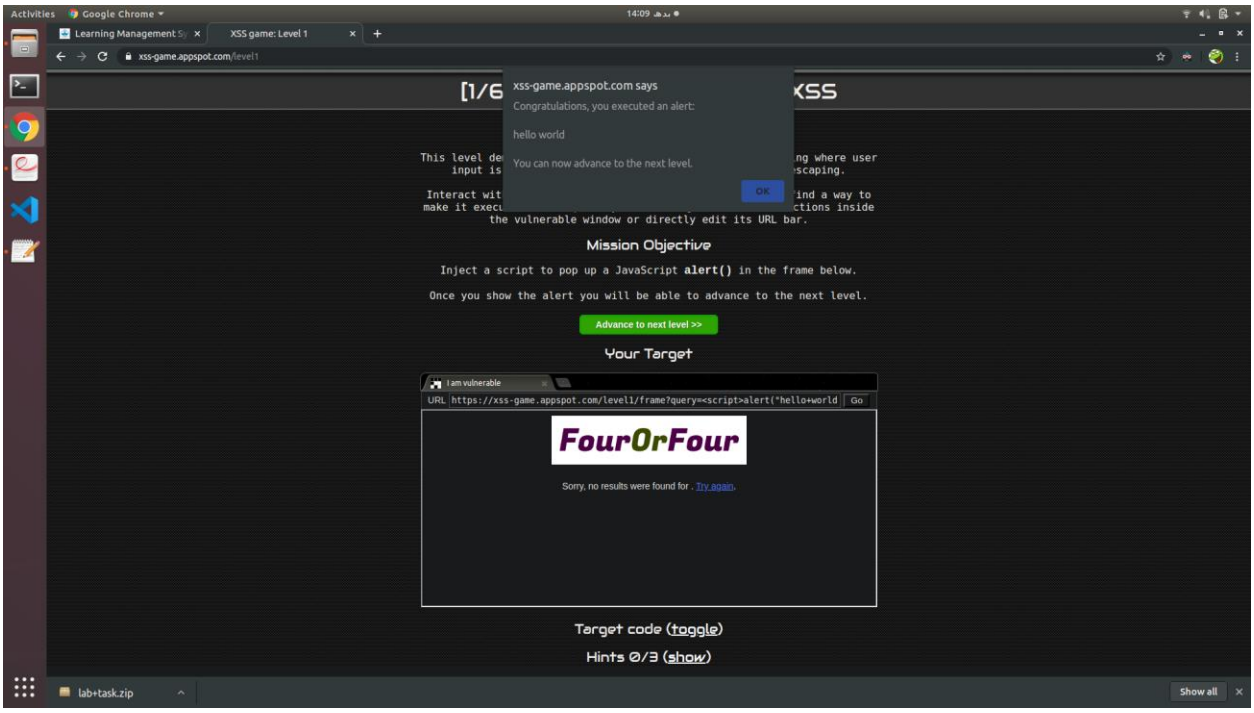
Squigs and queries from user input are now passed through this function in modified server.py before anything else is done to them. Tried and tested again for the above tasks, and vulnerabilities were successfully patched up and no XSS attacks were executed using the same scripts I wrote in task 3.

Patches introduced for XSS attacks were sufficient to prevent CSRF attacks executed in above tasks as well. Although there are multiple ways of preventing CSRF attacks such as generating CSRF tokens and then validating them upon each request received but for the scope of this assignment, escaping HTML tags and sanitizing user input was enough to prevent CSRF attacks.

After all these patches, I executed all the above attacks once again and none of them could bypass my security patches.

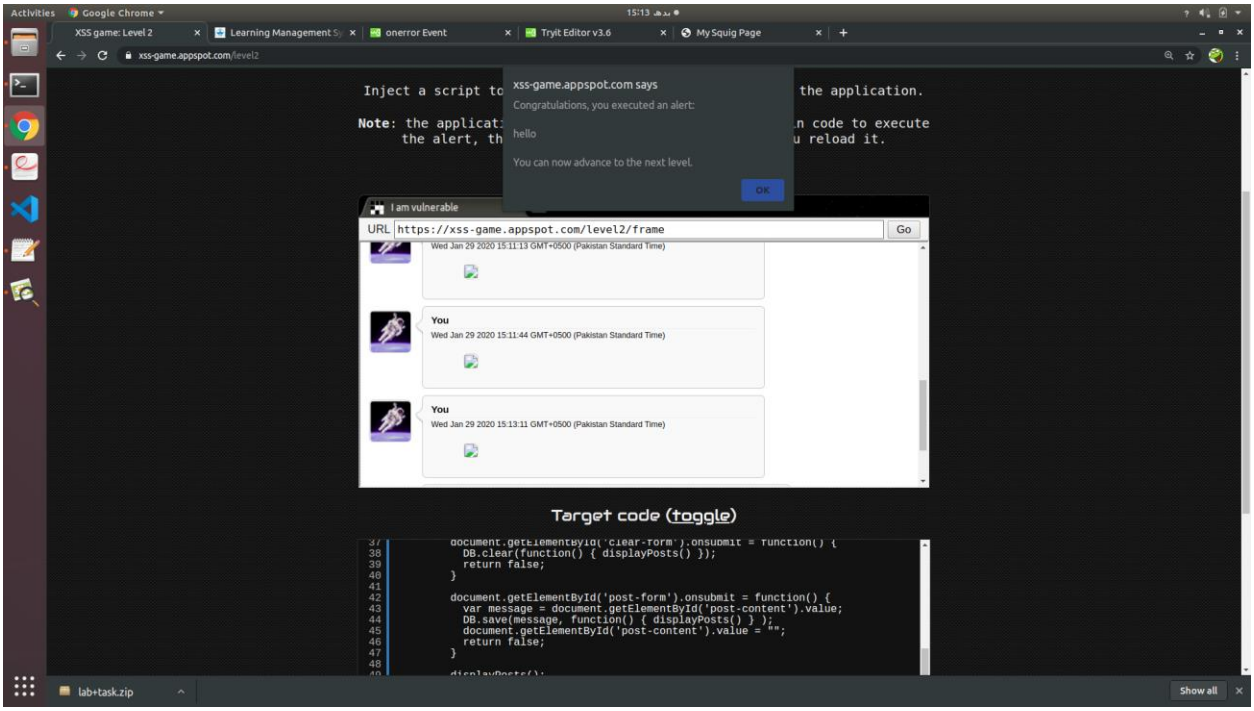
T. 4: XSS – Cross Site Scripting

- Level 1



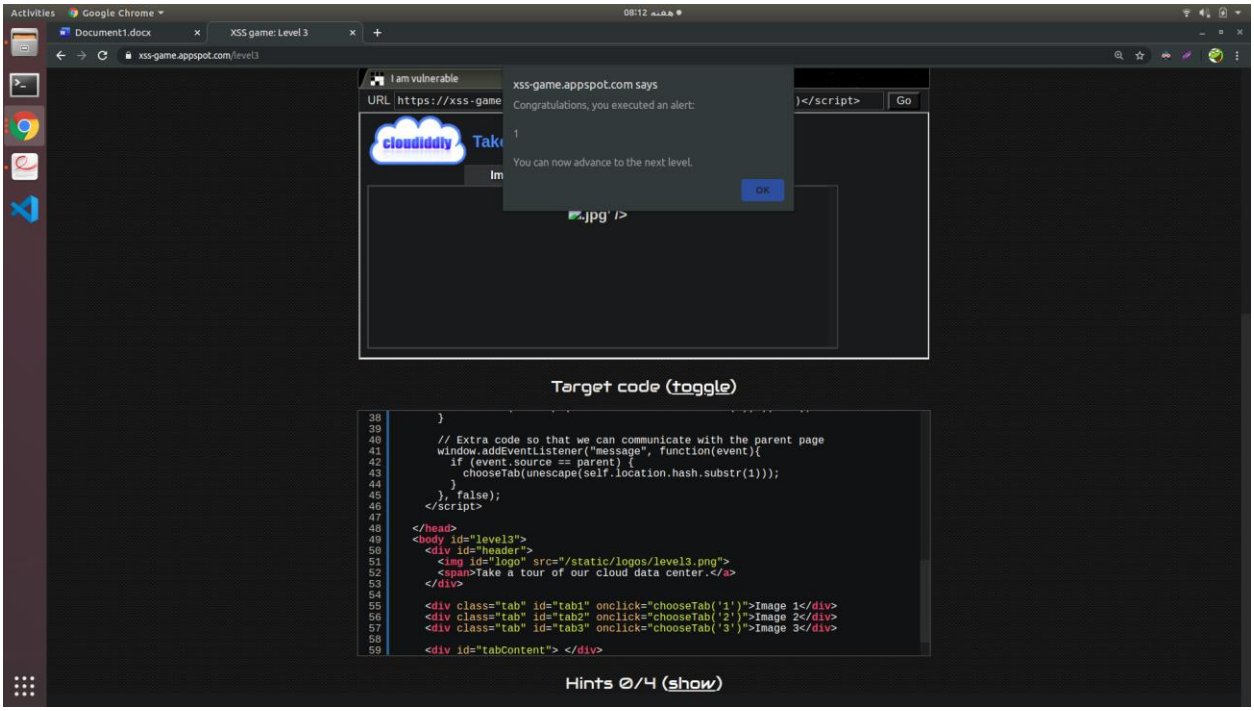
In this level, user input was directly included in the page without proper escaping and sanitization hence it was an easy attack. To call alert(), all I had to do was inject the following script: **<script>alert('hello world');</script>**.

- Level 2



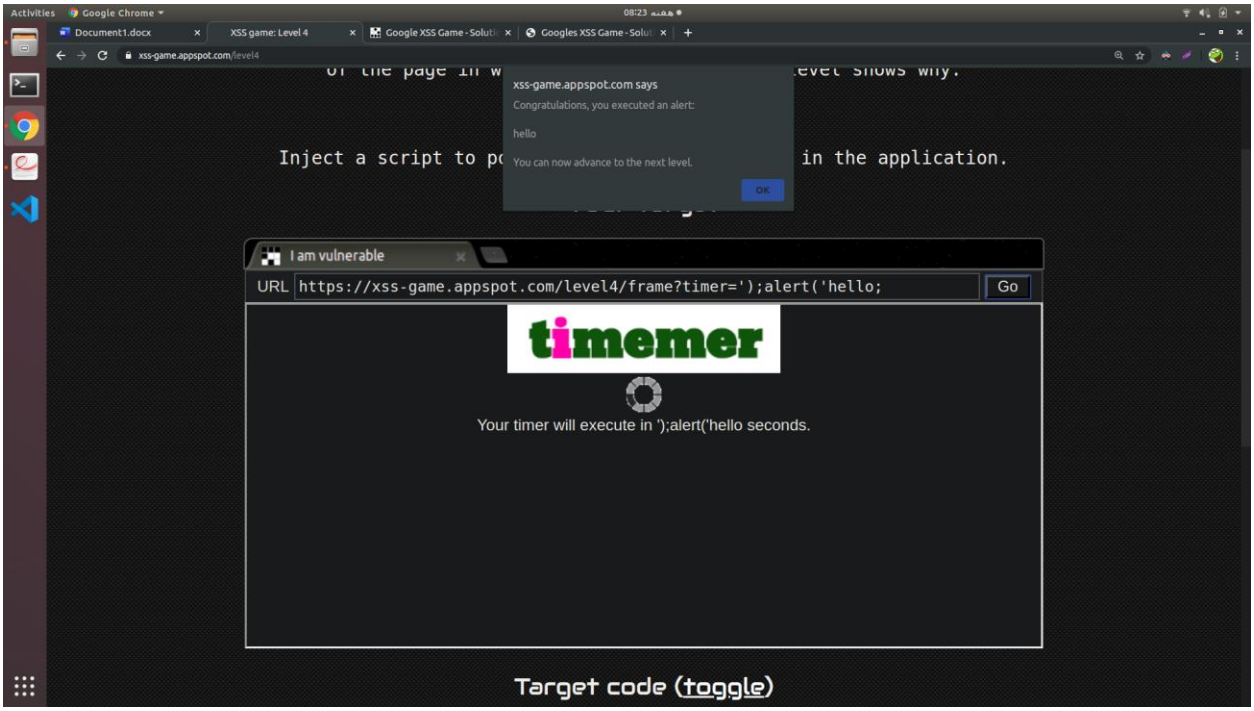
This is similar to level 1 except that we cannot use script tag because there is a validation which prevent us from using the script tag. I inspected the welcome post and found out it contains HTML, which indicates that the template does not escape the contents of the status message. Since entering script tag did not work out, I tried an element with a JavaScript attribute instead. To bypass it we can insert a image tag with an invalid URL and a onerror attribute which will execute a JavaScript alert. Hence, my injected script was: **<img src='invalid' onerror=”alert(‘hello’)”>**.

- Level 3



For this level, I had to toggle the code to look into the source code. I found a function inside index.html named chooseTab(num). Num parameter in the function was being used for generating the img tag. The code snippet which could be exploited is as follows: **html += "";**. I figured out I just had to break-out the quotes and insert my script. As in level 2, I used the onerror attribute to insert JavaScript. In this way I fooled it by closing the src attribute with a single quote. Hence, my injected script was: **' onerror='alert(1)';**. Alternatively, I could also have written: **' onerror='alert();//.jpg' />**. In this way, the .jpg part is commented out using the double slashes.

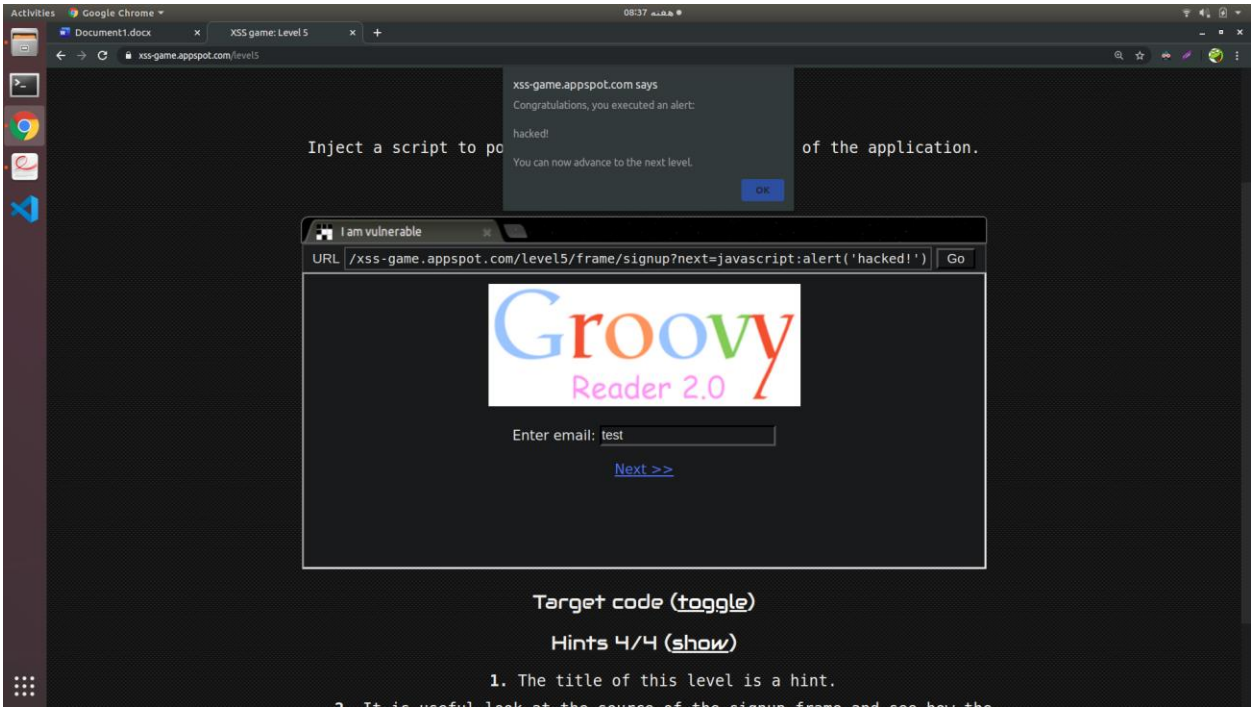
- Level 4



Similar to previous level, I toggled code for this level as well. The index page shows a form which we can pass a number, that is passed to the timer page which function is just to count the number we passed in seconds and then redirect us back to the beginning. We can fool its timer function to execute malicious code, since it is straightly added to the page. Following is the code snippet that I exploited: ****. Hence, I manipulated the JavaScript that could be executed here. Hence, my injected script was:
https://xssgame.appspot.com/level4/frame?timer=')%3Balert('hello')%3Bvar b=('.

This translates to: **startTimer(');alert('hello');var b=(');**. It should be noted that %3B represents the semi-colon in URL encoding.

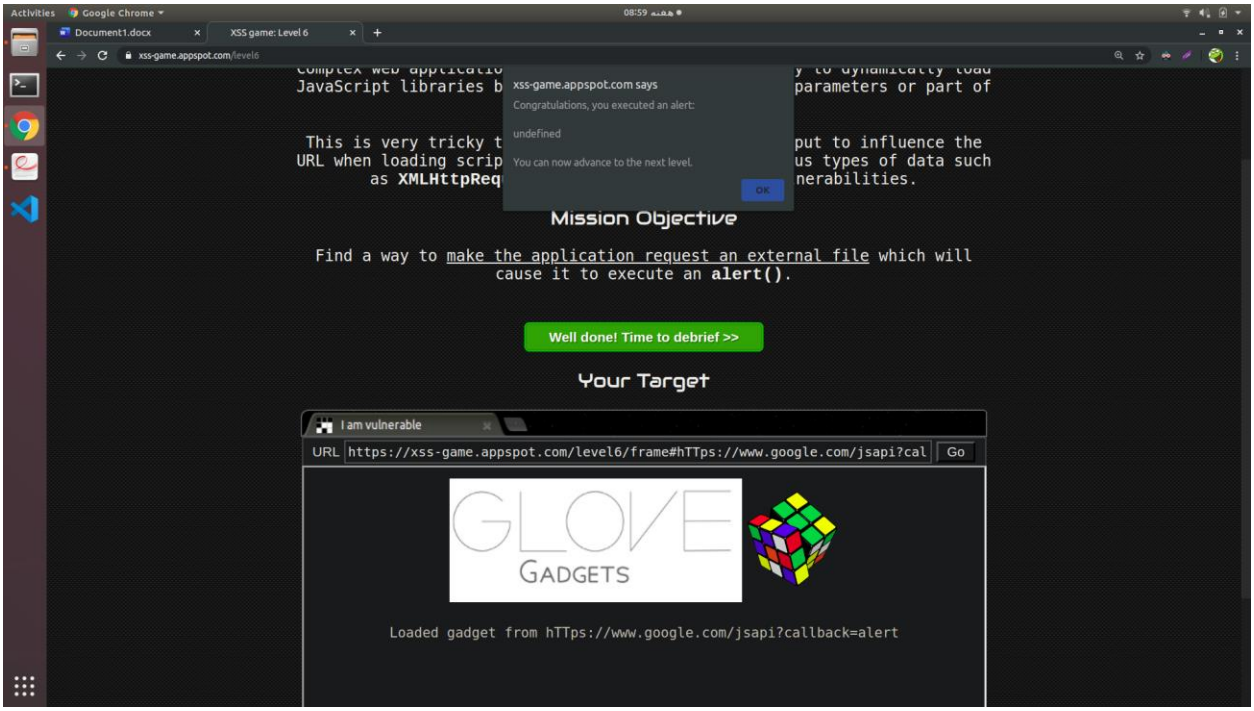
- Level 5



This level was quite easy to figure out. After clicking signup, I was redirected onto the next page that had the following URL: **https://xssgame.appspot.com/level5/frame/signup?next=confirm**.

I then looked for the ‘next’ parameter in the confirm.html source code. It had the following code snippet: **setTimeout(function() { window.location = '{{ next }}'; }, 5000);**. This is vulnerable to XSS attack since the window.location is set based on the ‘next’ parameter. Similarly, the ‘next’ parameter is being used as an <a> tag in the signup.html: **Next >>**. All I had to do was modify the URL and hence my malicious script became: **https://xss-game.appspot.com/level5/frame/signup?next=javascript:alert(‘hacked!’)**. Basically, the value of next is used to go onto the next page once we click on ‘Next’ hyperlink (reference to signup.html). So, I exploited this vulnerability. I simply redirected to above URL, wrote some dummy text in the signup field, clicked on ‘Next’ and voila! XSS attack was made.

- Level 6



On inspecting, index.html showed we are not allowed to have a URL containing 'https'. This prevents malicious URLs from executing. But the catch is the regex expression used to identify the protocol is not case-sensitive hence we can use 'HTTPS', 'hTTps', or literally any other combination to bypass this. Hence, I only had to replace **/static/gadget.js** in order to call alert(). So, my injected script became:
https://xssgame.appspot.com/level6/frame#hTTps://www.google.com/jsapi?callback=alert.

Honestly, it was a no brainer and took much less time than I had originally anticipated. Hints provided proved helpful as well.

T. 5: Briefly answer these questions

Q1: Provide an example of a simple test to determine if XSS vulnerability is present in a web application.

- XSS flaws can be difficult to identify and remove from a web application. The best way to find flaws is to perform a security review of the code and search for all places where input from an HTTP request could possibly make its way into the HTML output. Note that a variety of different HTML tags can be used to transmit a malicious JavaScript. There are many kinds of vulnerabilities that could be exploited to perform XSS attacks such as input validation vulnerability and session management vulnerability.
- As an example, I will provide test for input validation vulnerability. An application is exposed to input validation vulnerabilities if an attacker finds that the application makes untested assumptions about the type, duration, format, or scope of input data.
- As an example, take a web-app, which has a search bar. If the search field is vulnerable, when the user enters any script, then it will be executed. For instance, if the user/attacker enters a very basic **<script>alert(1)</script>**, it will be executed. More harmful scripts may be executed in this way as well. This is a simple test to determine XSS vulnerability in a web-app.

Q2: Is your example above for reflected XSS or stored XSS? And which type of XSS attack can affect wider audience? Why?

- Example is of reflected XSS attack since malicious script bounces off of another website to the victim's browser. In the above case, dialog box with an alert would be displayed on the victim’s screen.
- Stored XSS is riskier and provides more damage and hence affects a wider audience.
- In this type of attack, the malicious code or script is being saved on the web server (for example, in the database) and executed every time when the users will call the appropriate functionality. This way stored XSS attack can affect

many users. Also, as the script is being stored on the web server, it will affect the website for a longer time.

Q3: One of your colleagues has developed a mechanism to prevent XSS attack by removing any instances of word "script" in any input. The prevention is as:
1) User enters a string to input field and presses "enter"
2) Your code at server side, takes the string and runs the following code before doing anything else:
(See Attached Image in PDF file)
Is there anything fishy you can still do? Note that you are bound to only use "script" tags for injecting any scripts so figure out if this is still possible.

- Yes, looking at the mechanism developed, I instantly noticed a flaw. The snippet checking for "script" tags is not case sensitive. Meaning it will only detect script when it is lower case. Upper case or any other combination will pass on without being removed.
- So, I can execute the script in any other possible combinations of upper and lower case.
    - <SCRIPT>alert("hacked")</SCRIPT>
    - <SCrIpT>alert("hacked")</SCrIpT>
    - <sCRIPt>alert("hacked")</ScRipT>
    - Literally, any combination will work! I tried it myself.

Q4: Tell whether a white-list approach or a black-list approach would be a better solution for prevention against CSRF attacks. If your answer is white-list approach, clearly explain the disadvantages of black-list approach.

- White-list approach would be a better solution.
- White-listing basically limits the user's input to known values for us which means that we already know about all the possible inputs that the user can make and therefore, we also know that there cannot be an attack from our provided input. For blacklisting, however, we can only guard against known malicious inputs. There may be a new malicious input that is unknown by us and therefore can bypass our security measure. In this sense, white-list is a better approach since no such attacks can occur.

Q5: Write a short note explaining SQL Prepared Statements.

- A prepared statement is a feature used to execute the same (or similar) SQL statements repeatedly with high efficiency.
- A SQL statement template is created and sent to the database. Certain values are left unspecified, called parameters (labeled "?"). For example, INSERT INTO TableName VALUES(?, ?, ?).
- The database parses and compiles the SQL statement template, and stores the result without executing it
- At a later time, the application binds the values to the parameters, and the database executes the statement. The application may execute the statement as many times as it wants with different values
- Prepared statements are very useful against SQL injections, because parameter values, which are transmitted later using a different protocol, need not be correctly escaped. If the original statement template is not derived from external input, SQL injection cannot occur.
- Example
    - “””SELECT password FROM accounts WHERE username = ?;”””
    - This way SQL injection attacks are prevented since any kind of data from user would not cause the SQL query to be changed or modified.
