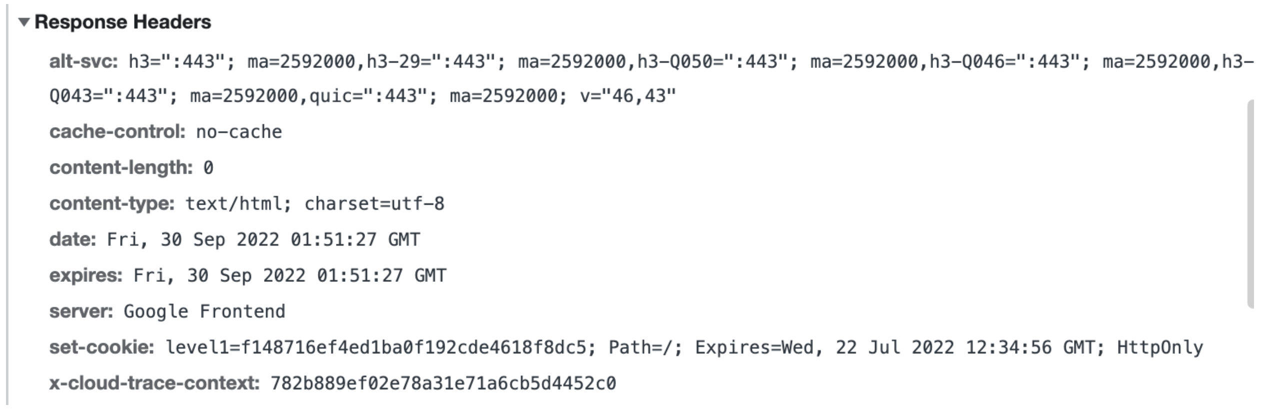
Homework 2 XSS Scripting Report

Tianjian Li
Johns Hopkins University

Part 1 Bypassing Cookie Restriction and Solving the Game

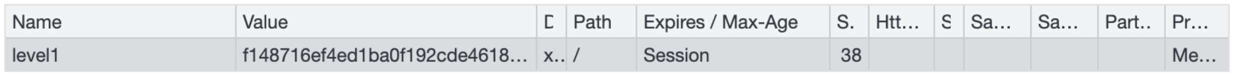
The HTTP response header is shown below:



```
▼ Response Headers
   alt-svc: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000,h3-Q050=":443"; ma=2592000,h3-Q046=":443"; ma=2592000,h3-
   Q043=":443"; ma=2592000,quic=":443"; ma=2592000; v="46,43"
   cache-control: no-cache
   content-length: 0
   content-type: text/html; charset=utf-8
   date: Fri, 30 Sep 2022 01:51:27 GMT
   expires: Fri, 30 Sep 2022 01:51:27 GMT
   server: Google Frontend
   set-cookie: level1=f148716ef4ed1ba0f192cde4618f8dc5; Path=/; Expires=Wed, 22 Jul 2022 12:34:56 GMT; HttpOnly
   x-cloud-trace-context: 782b889ef02e78a31e71a6cb5d4452c0
```

The set-cookie sets the cookie to be expired on 22 July 2022, and the cookie ‘HttpOnly’ flag is set, which forbids the client(in this case, me) from editing the cookie using javascript.

However, you can still use the cookie-editor extension in chrome to modify your cookie by simply adding a new cookie with the same name and value as the expired cookie with a much later expiration date, which will advance you to the next level of the game.



Name	Value	D	Path	Expires / Max-Age	S.	Htt...	S	Sa...	Sa...	Part..	Pr...
level1	f148716ef4ed1ba0f192cde4618...	x..	/	Session	38						Me...

Here we can see that the expires/max-age becomes a session. Therefore, it does not expires. However, we need to manually add cookies at each level of the game.

Level 1 Trigger:
<script>alert(‘hello’)</script>
Vulnerable Code:
message = “Sorry, no results were found for <b>” + query + “</b>.”
message += " <a href='?'>Try again</a>."

Explanation:
Here this code just naturally assumes that “query” is a string and put into the message string. However here if the query is executable javascript code wrapped with <script></script>, the compiler will execute it and will result in an alert.

Level 2 Trigger
<img src='x' onerror='alert()'>
Code
 html += "<blockquote>" + posts[i].message + "</blockquote">;
Explanation
Html does not escape status messages, which we can insert an onerror=alert() to trigger the alert box.

Level 3 Trigger:
12313' onerror='alert();//
Code:
html += "<img src='/static/level3/cloud" + num + ".jpg' />";
Explanation:
The code here does not check if num contains js functions, therefore if we end the string and enter a js alert function would successfully trigger the alert.

Level 4 Trigger
1212'+alert());//
Code:
<img src="/static/loading.gif" onload="startTimer('{{ timer }}');" />
Explanation:
The code does not check if timer is a valid entry. Therefore we can end the string with a ‘ and inject the alert() function here.
This code evaluates the time to be the concatenation of ‘3’ and the result of the function alert(), which results in an alert box showing up.

Level 5 Trigger
Entering
javascript:alert() at the entering email page
next=javascript:alert()
Vulnerable Code:
<a href="{{ next }}">Next >></a>

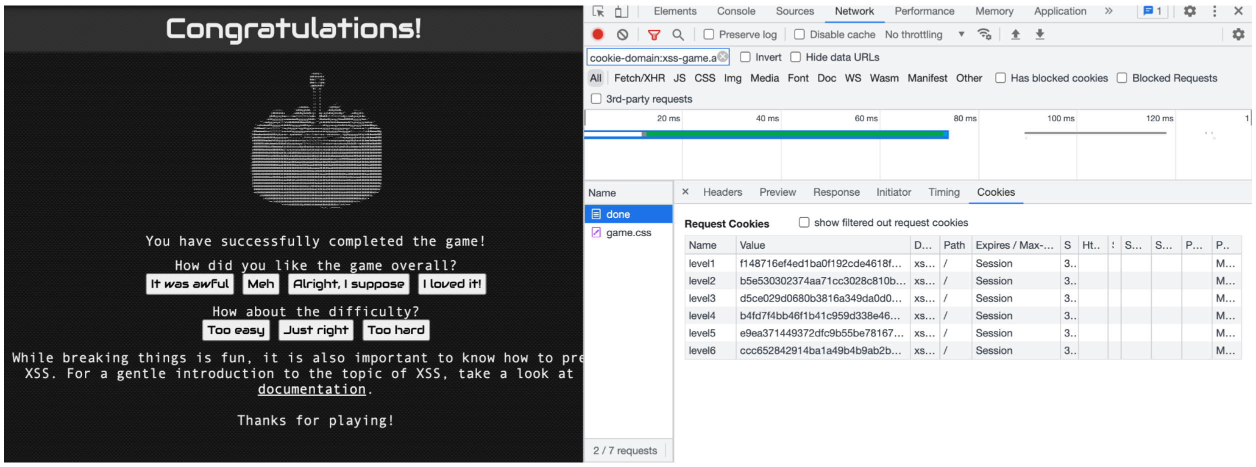
Explanation:
Here if javascript:alert() is passed to the “next” parameter, the code becomes
<a href='javascript:alert()'>...</a>, which results in the browser executing javascript code.

Level 6 Trigger
//google.com/jsapi?callback=alert
Vulnerable Code:

```
    if (url.match(/^https?:\/\//)) {
      setInnerText(document.getElementById("log"),
        "Sorry, cannot load a URL containing \"http\".");
      return;   }
```

Explanation:
Here if we simply omit the protocol, it will automatically inherit the protocol from the current environment, which results an call of the alert page.

Showing of the ending page and all the request cookies in session.


Congratulations!

You have successfully completed the game!
How did you like the game overall?
It was awful | Meh | Alright, I suppose | I loved it!
How about the difficulty?
Too easy | Just right | Too hard
While breaking things is fun, it is also important to know how to pre
XSS. For a gentle introduction to the topic of XSS, take a look at
documentation.

Thanks for playing!

Request Cookies

Name	Value	D...	Path	Expires / Max-...	S	Ht..	!	S...	S...	P...	P..
level1	f148716ef4ed1ba0f192cde4618f...	xs...	/	Session	3..						M...
level2	b5e530302374aa71cc3028c810b...	xs...	/	Session	3..						M...
level3	d5ce029d0680b3816a349da0d0...	xs...	/	Session	3..						M...
level4	b4fd7f4bb46f1b41c959d338e46...	xs...	/	Session	3..						M...
level5	e9ea371449372dfc9b55be78167...	xs...	/	Session	3..						M...
level6	ccc652842914ba1a49b4b9ab2b...	xs...	/	Session	3..						M...

Part 2 Patching Vulnerable Code: directory at /hw2

The code is based on python3. In order to run the code, one must install flask

- pip3 install flask

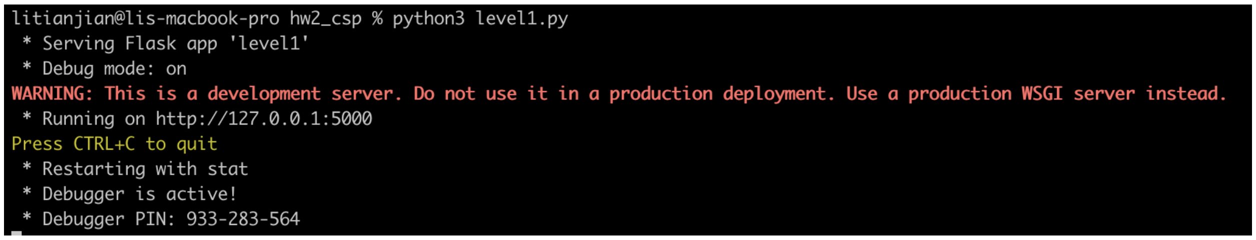
In order to run the code using Content Security Policy, one must install flask_csp

- pip3 install flask_csp

To run level x, simply type in the console,

- python3 level{x}.py

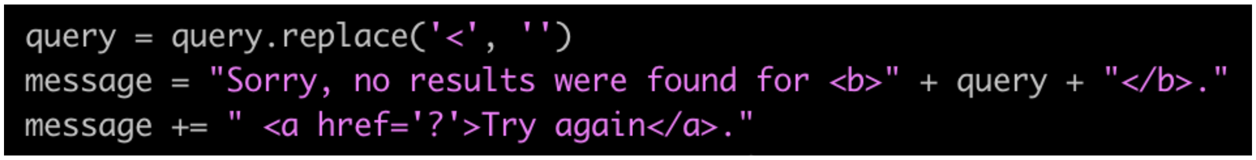
and the console would output an URL like this:



```
litianjian@lis-macbook-pro hw2_csp % python3 level1.py
 * Serving Flask app 'level1'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 933-283-564
```

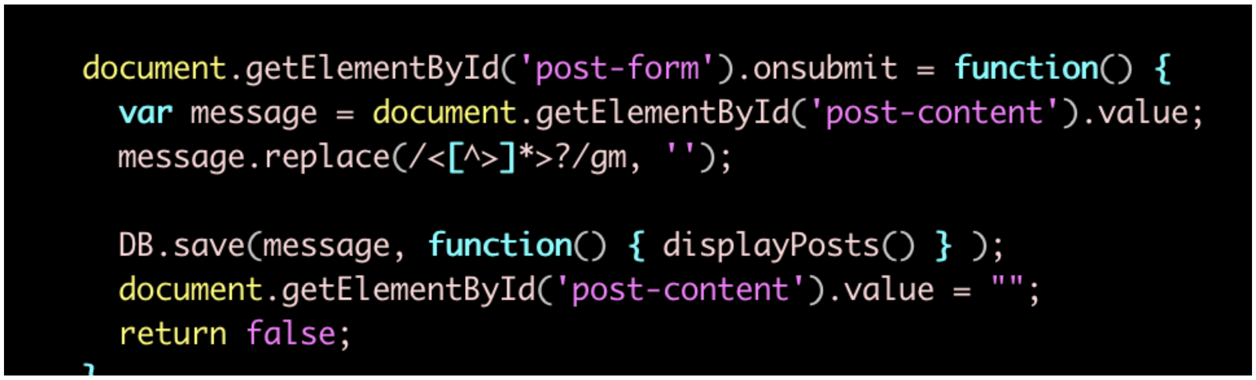
Entering the URL in Chrome(note that other browsers might not support CSP 3) would start the application.

Level 1: Used a filter that filters through html tags



```
query = query.replace('<', '')
message = "Sorry, no results were found for <b>" + query + "</b>."
message += " <a href='?'>Try again</a>."
```

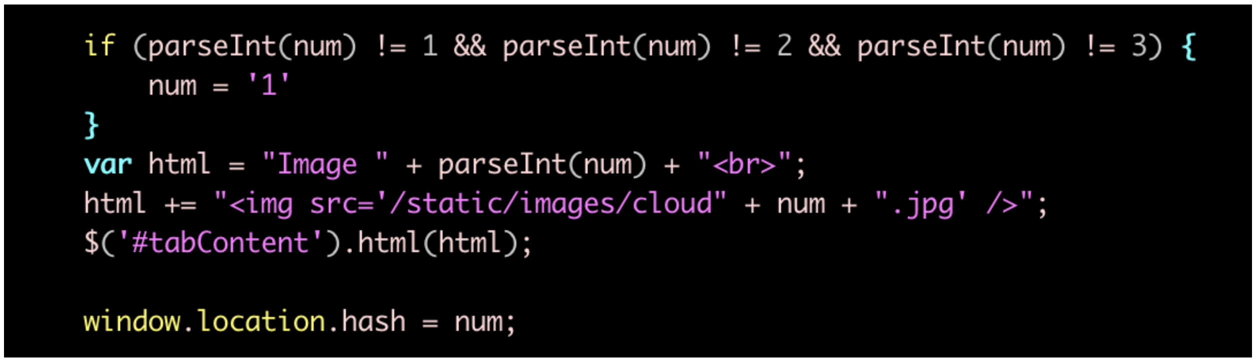
Level 2: Used a javascript filter to filer out all the html tags, so that the client would not be able to use something like <img = ‘nonexist’, onerror = alert()> to trigger the alert.



```
document.getElementById('post-form').onsubmit = function() {
  var message = document.getElementById('post-content').value;
  message.replace(/<[^>]*>?/gm, '');

  DB.save(message, function() { displayPosts() } );
  document.getElementById('post-content').value = "";
  return false;
}
```

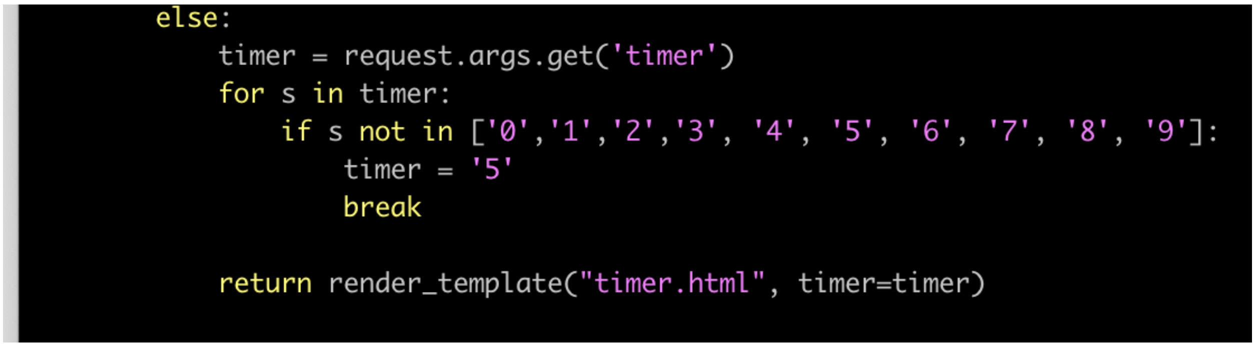
Level 3: Restricted the only possible numeric values to be 1, 2 and 3 since we only have 3 pictures



```
if (parseInt(num) != 1 && parseInt(num) != 2 && parseInt(num) != 3) {
    num = '1'
}
var html = "Image " + parseInt(num) + "<br>";
html += "<img src='/static/images/cloud" + num + ".jpg' />";
$('#tabContent').html(html);

window.location.hash = num;
```

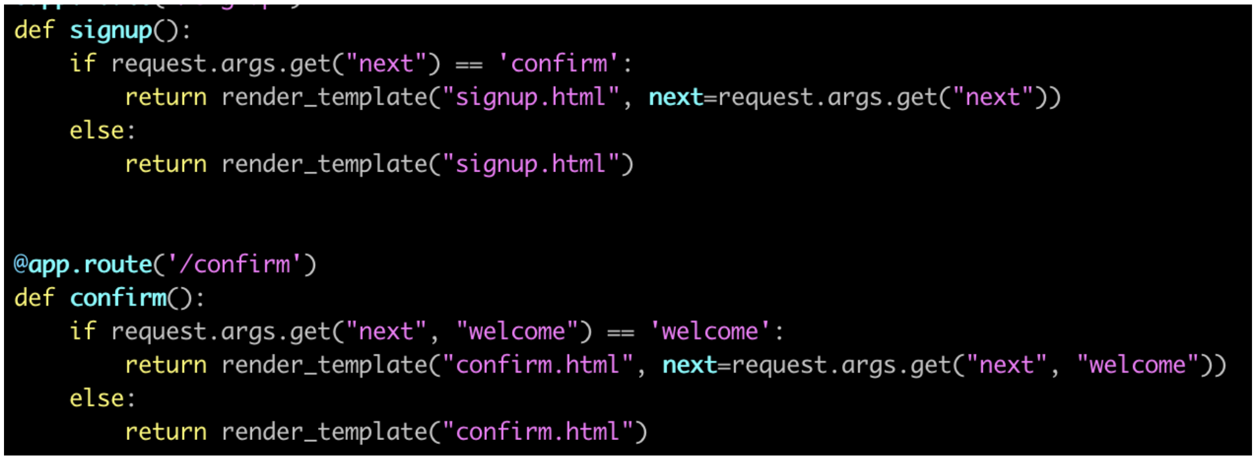
Level 4: Used a filterer that filters out all the non-numeric values since the only possible value to enter is a number.



```
        else:
            timer = request.args.get('timer')
            for s in timer:
                if s not in ['0','1','2','3', '4', '5', '6', '7', '8', '9']:
                    timer = '5'
                    break

            return render_template("timer.html", timer=timer)
```

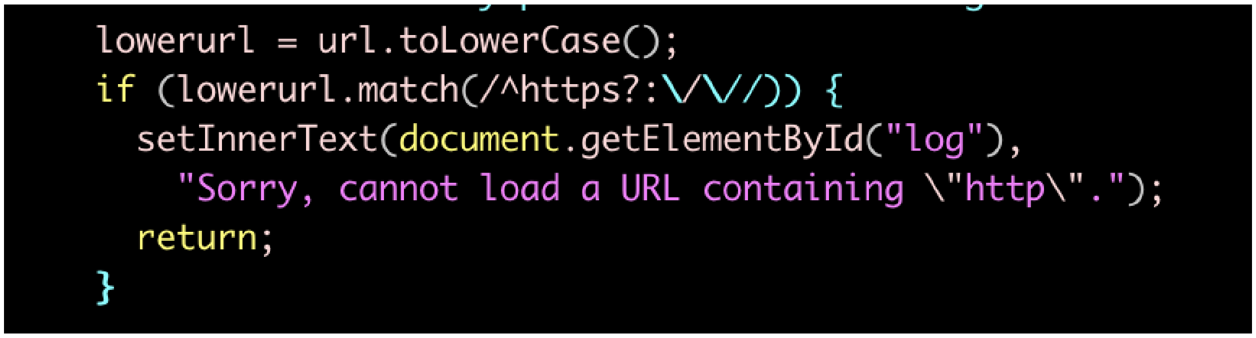
Level 5:Since the only options are “confirm” and “welcome”, we hardcode to make the application only recognize these two inputs.



```
def signup():
    if request.args.get("next") == 'confirm':
        return render_template("signup.html", next=request.args.get("next"))
    else:
        return render_template("signup.html")


@app.route('/confirm')
def confirm():
    if request.args.get("next", "welcome") == 'welcome':
        return render_template("confirm.html", next=request.args.get("next", "welcome"))
    else:
        return render_template("confirm.html")
```

Level 6: Make the url to lower case then apply the default filterer that filters out “http”



```
    lowerurl = url.toLowerCase();
    if (lowerurl.match(/^https?:\/\//)) {
      setInnerText(document.getElementById("log"),
        "Sorry, cannot load a URL containing \"http\".");
      return;
    }
```

Part 3 CSP Policy

(i) We make all the inline script <script> external by creating a new .js document and including them by <script src = “document.js”> </script>. The Python implementation is at the directory /hw2_csp

Level 1:

Content-Security-Policy: script-src 'self'

Level 2:
Content-Security-Policy: script-src 'self', img-src 'self' ssl.gstatic.com

Level 3
I move the inline script to /static/level3.js
Content-Security-Policy: script-src ‘self’ ajax.googleapis.com + hashes of the onclick functions

Level 4: Moved inline script to /static/level4.js
Content-Security-Policy: script-src ‘self’

Level 5: Moved inline script to /static/level5_confirm.js
Content-Security-Policy: script-src ‘self’

Level 6: Move inline script to /static/level6.js
Content-Security-Policy: script-src ‘self’

(ii) CSP - Nonce: Directory at hw2_nonce
Instead of using the hashes, we can use a nonce to “whitelist” the script we would like to execute.
An example would be:
content-Security-Policy: script-src 'nonce-anynameyoulike'
And at the script you want to execute
<script nonce=“anynameyoulike”> </script>

The code implementation can be found at csp_nonce

Level 1:
Content-Security-Policy: script-src 'self'

Level 2: Used <script nonce=‘pizza’> js code </script> to whitelist the script
Content-Security-Policy: script-src 'self' ‘nonce-pizza’, img-src 'self' ssl.gstatic.com

Level 3: Used <script nonce=‘pizza’> js code </script> to whitelist the script
Content-Security-Policy: 'script-src': 'self' 'nonce-pizza' ajax.googleapis.com + the hashes of the onclick divs"

Level 4: Used <script nonce='pizza'> js code </script> to whitelist the script
Content-Security-Policy: 'script-src': 'self' 'nonce-pizza' + the hash of the onload start timer call at /tempates/timer.html

Level 5: Used <script nonce='pizza'> js code </script> to whitelist the script
Content-Security-Policy: 'script-src': 'self' 'nonce-pizza'

Level 6: Used <script nonce='pizza'> js code </script> to whitelist the script
Content-Security-Policy: 'script-src': 'self' 'nonce-pizza'