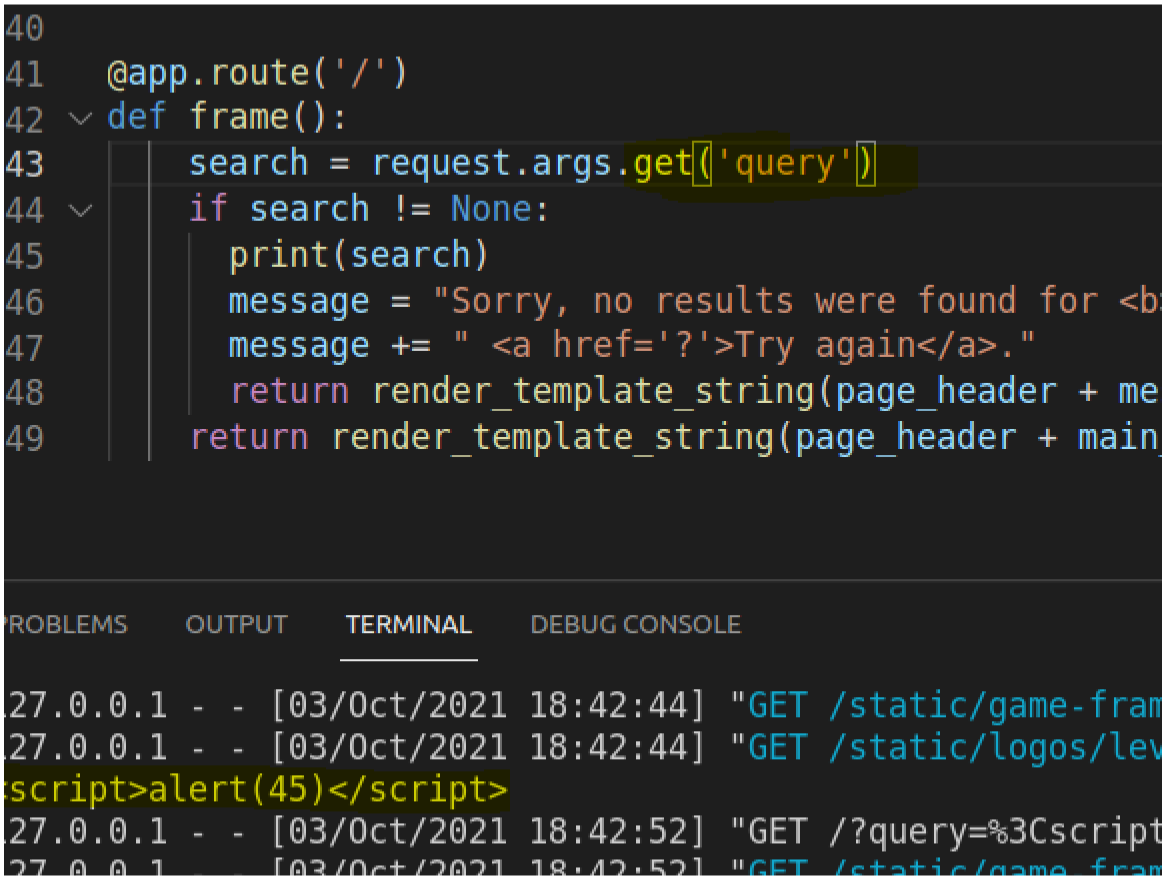
Part 2: Vulnerability Patching

Level 1: Hello, world of XSS

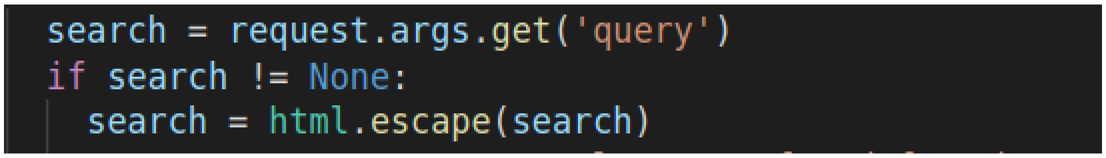
We can infer from the below screenshot that no sanitization takes place.



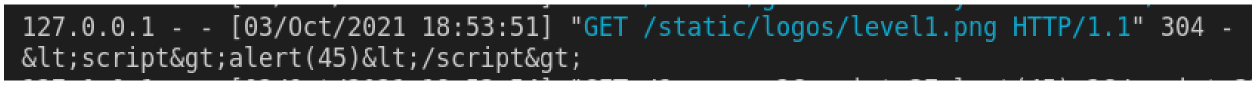
```
@app.route('/')
def frame():
    search = request.args.get('query')
    if search != None:
      print(search)
      message = "Sorry, no results were found for <b
      message += " <a href='?'>Try again</a>."
      return render_template_string(page_header + me
    return render_template_string(page_header + main
```

```
27.0.0.1 - - [03/Oct/2021 18:42:44] "GET /static/game-fram
27.0.0.1 - - [03/Oct/2021 18:42:44] "GET /static/logos/lev
script>alert(45)</script>
27.0.0.1 - - [03/Oct/2021 18:42:52] "GET /?query=%3Cscript
```

So, I will be sanitizing the query with the help of html library and using its escape function to sanitize the input before rendering it back on the screen.



```
search = request.args.get('query')
if search != None:
  search = html.escape(search)
```



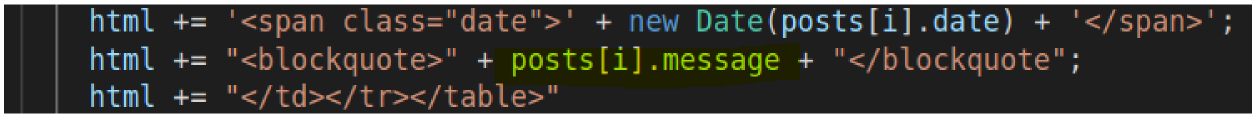
```
127.0.0.1 - - [03/Oct/2021 18:53:51] "GET /static/logos/level1.png HTTP/1.1" 304 -
&lt;script&gt;alert(45)&lt;/script&gt;
```

Instruction to run the website:- a) export FLASK_APP=level1.py

a) flask run

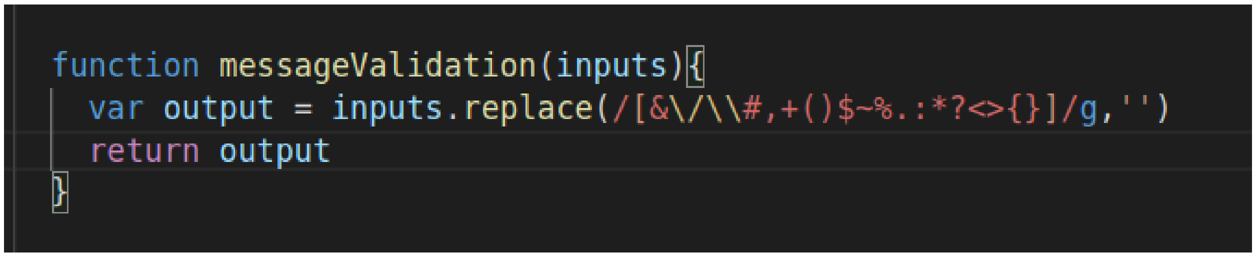
Level 2: Persistence is key

We have to sanitize **posts[i].message** before it is parsed by the browser.

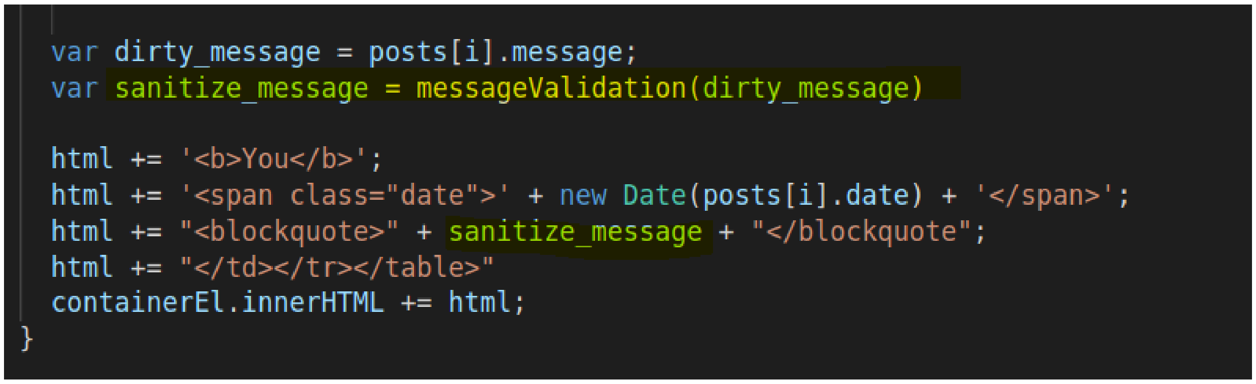


```
html += '<span class="date">' + new Date(posts[i].date) + '</span>';
html += "<blockquote>" + posts[i].message + "</blockquote";
html += "</td></tr></table>"
```

Created message validation function which will sanitize the posts before they are rendered on the screen.



```
function messageValidation(inputs){
  var output = inputs.replace(/[&\/\\#,+()$~%.:*?<>{}]/g,'')
  return output
}
```



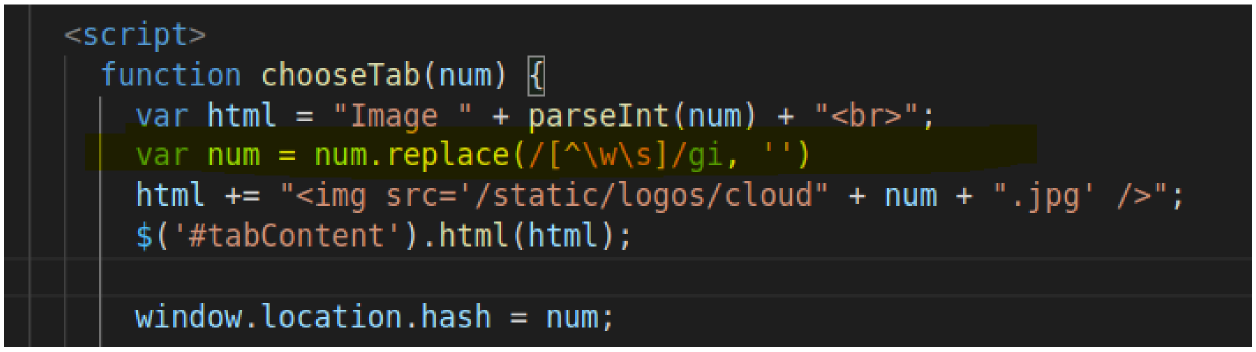
```
  var dirty_message = posts[i].message;
  var sanitize_message = messageValidation(dirty_message)

  html += '<b>You</b>';
  html += '<span class="date">' + new Date(posts[i].date) + '</span>';
  html += "<blockquote>" + sanitize_message + "</blockquote";
  html += "</td></tr></table>"
  containerEl.innerHTML += html;
}
```

Now while parsing the special characters will be ignored by the browser thereby effectively patching our Xss vulnerability.

Instruction to Run the Website:

a) export FLASK_APP=level2.py
b) flask run

Level 3: That sinking feeling...



```
<script>
  function chooseTab(num) {
    var html = "Image " + parseInt(num) + "<br>";
    var num = num.replace(/[^\w\s]/gi, '')
    html += "<img src='/static/logos/cloud" + num + ".jpg' />";
    $('#tabContent').html(html);

    window.location.hash = num;
```

The xss in this level was exploited by controlling the URL with the help of **"num"** and using to trigger an alert via event.

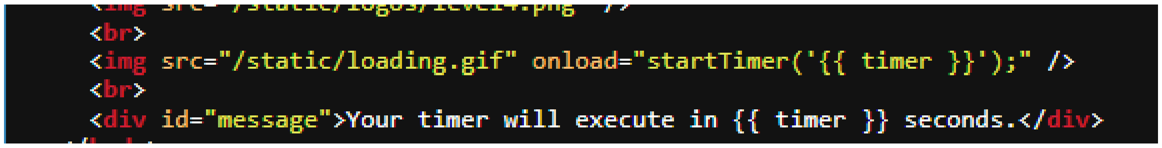
So, to patch this vulnerability, we will be validating/sanitizing **num** for special characters with the help of replace() function.

Instruction to Run the Website:

a) export FLASK_APP=level3.py
b) flask run

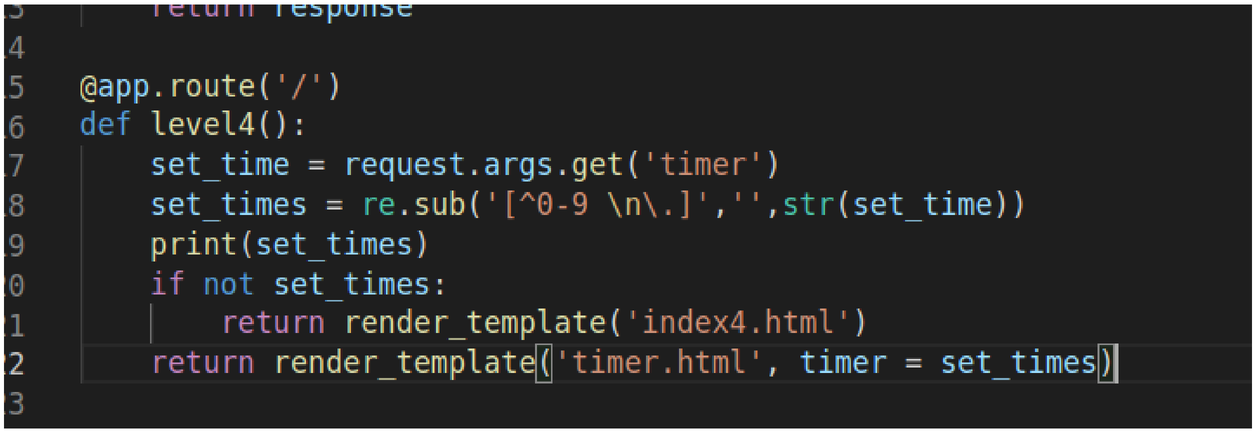
Level 4: Context matters

We can observe that the value of the timer is passed without any sanitation.



```
<br>
<img src="/static/loading.gif" onload="startTimer('{{ timer }}');" />
<br>
<div id="message">Your timer will execute in {{ timer }} seconds.</div>
```

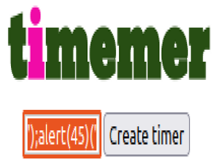
So, I will be using re.sub for replacing any that is not a number with ' '. Please find the patch implemented below:-

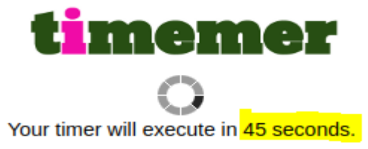


```
    return response

@app.route('/')
def level4():
    set_time = request.args.get('timer')
    set_times = re.sub('[^0-9 \n\.]','',str(set_time))
    print(set_times)
    if not set_times:
        return render_template('index4.html')
    return render_template('timer.html', timer = set_times)
```

The exploit payload for this task was :- **');alert(45)('.**

Now if we enter this again, we will get only 45, remaining characters will be stripped away.





Your timer will execute in 45 seconds.

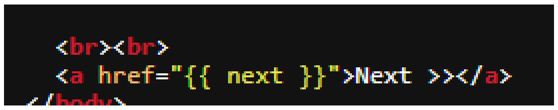
Instruction to Run the Website:

a) export FLASK_APP=level4.py
b) flask run

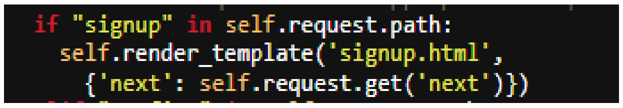
Level 5: Breaking protocol

Exploit Payload: javascript:alert("45")

In the sign-up page, the parameter of the url **next=**confirm influences the behavior of the web application.

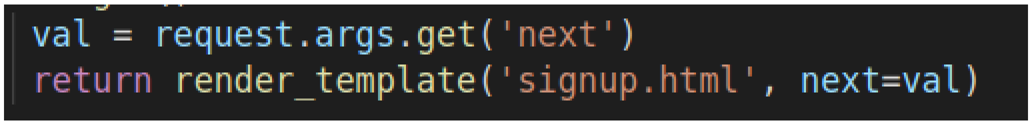


```
<br><br>
<a href="{{ next }}">Next >></a>
```



```
if "signup" in self.request.path:
  self.render_template('signup.html',
    {'next': self.request.get('next')})
```

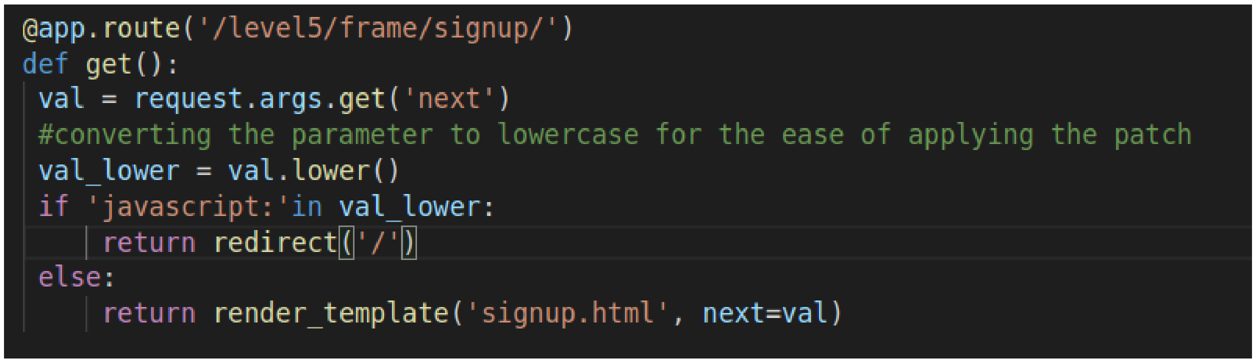
So, to patch this, we need to make sure { { next }} query is handled properly. Since the user will be entering an email id, we need to blacklist the keyword “javascript” which cannot be allowed while creating this patch We will be searching for these keywords in our request.args.get(‘**next’**) parameter both at the sign up page and confirm page.



```
val = request.args.get('next')
return render_template('signup.html', next=val)
```

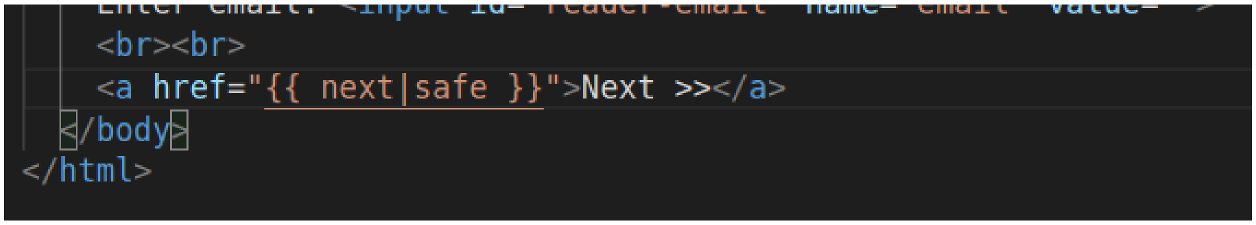
Patch :-

Val holds our request.args.get parameter. It is the converted to lowercase. After this we check for the keyword **javascript:** in our query. If it is found, the attacker will be redirected to welcome.html page.



```
@app.route('/level5/frame/signup/')
def get():
 val = request.args.get('next')
 #converting the parameter to lowercase for the ease of applying the patch
 val_lower = val.lower()
 if 'javascript:'in val_lower:
     return redirect('/')
 else:
     return render_template('signup.html', next=val)
```

In addition to above sanitization, we will also be employing |safe which won't translate "dangerous" symbols into html entities.



```
    <br><br>
    <a href="{{ next|safe }}">Next >></a>
  </body>
</html>
```

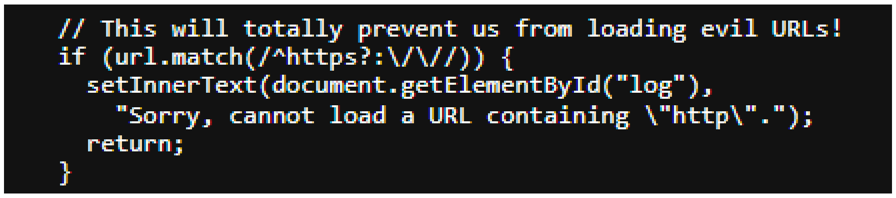
Instruction to Run the Website:

a) export FLASK_APP=level5.py
b) flask run

Level 6: Follow the Rabbit

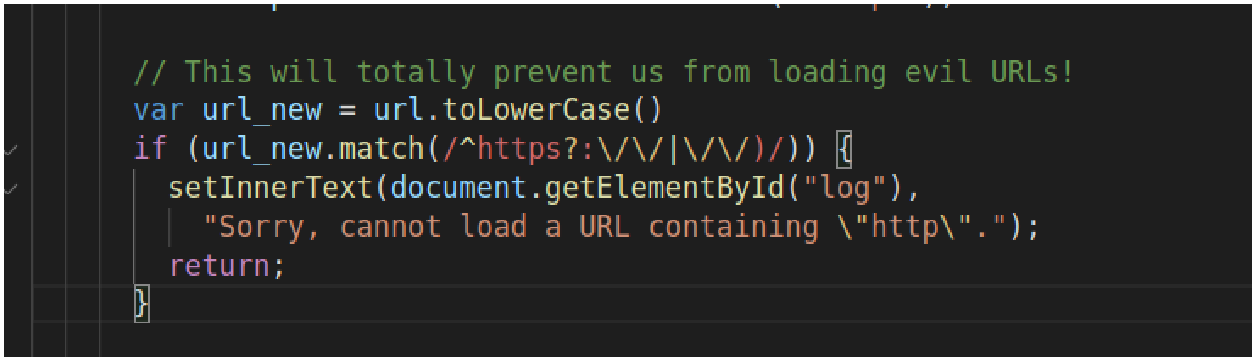
Payload Exploit: hTTps://www.google.com/jsapi?callback=alert

In this level we were able to trigger the alert by bypassing the url.match function. So I will be modifying this in order to patch this web application.



```
  // This will totally prevent us from loading evil URLs!
  if (url.match(/^https?:\/\//)) {
    setInnerText(document.getElementById("log"),
      "Sorry, cannot load a URL containing \"http\".");
    return;
  }
```

Patch :-



```
    // This will totally prevent us from loading evil URLs!
    var url_new = url.toLowerCase()
    if (url_new.match(/^https?:\/\/|\/\/)/)) {
      setInnerText(document.getElementById("log"),
        "Sorry, cannot load a URL containing \"http\".");
      return;
    }
```

Instruction to Run the Website:

a) export FLASK_APP=level6.py
b) flask run