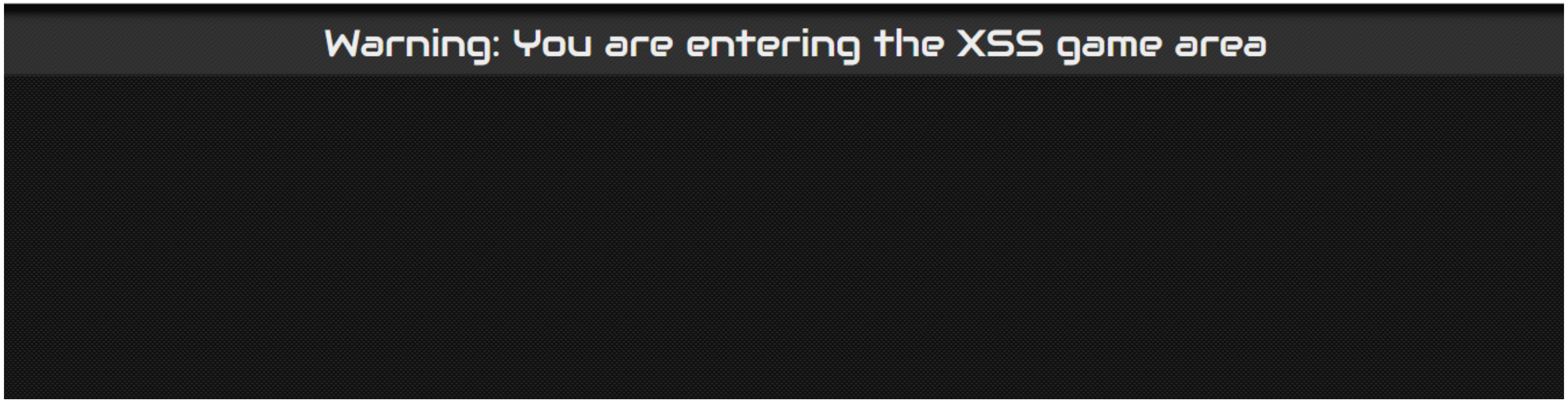
Google XSS game walkthrough

Pedro Henrique Cardoso Follow
Jul 19, 2019 · 6 min read



Google has a funny beginner like XSS game, and although it was quite easy I learned a thing or two.

So this article is for those who are stuck in the game or someone who wants to just understand why the levels' payloads works.

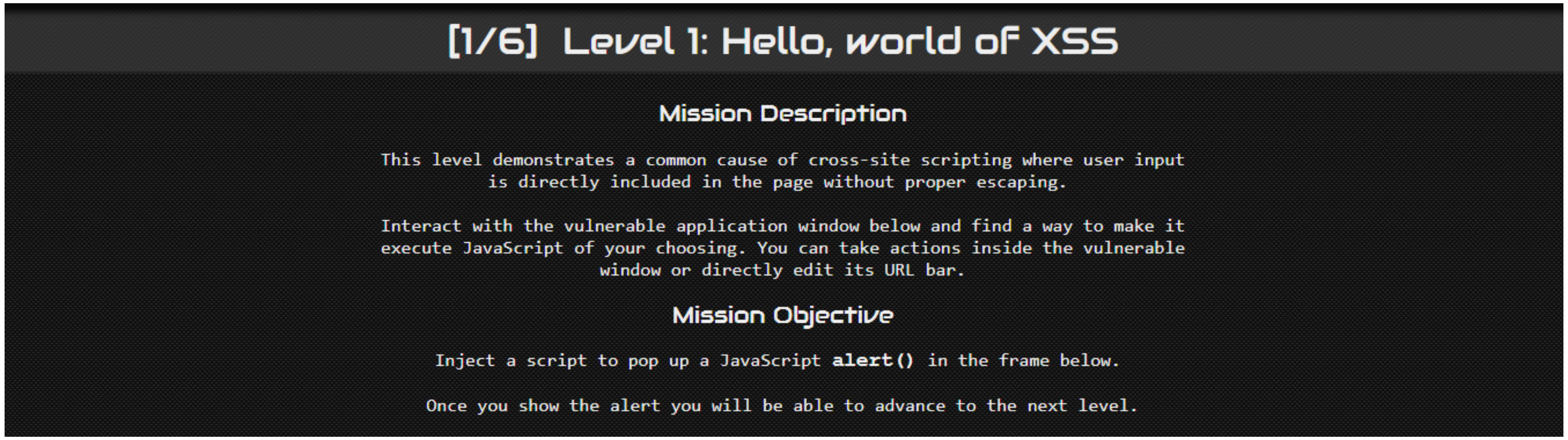
. . .

Just an advice: If you are just stuck Irecommend you to take a walk, drink some coffee, and try later with a fresher mind. Going straight to the solution very often do not teach you too much as when you do by yourself.

Well, let's go:

. . .

Level 1: Hello, world of XSS



[1/6] Level 1: Hello, world of XSS

Mission Description

This level demonstrates a common cause of cross-site scripting where user input is directly included in the page without proper escaping.

Interact with the vulnerable application window below and find a way to make it execute JavaScript of your choosing. You can take actions inside the vulnerable window or directly edit its URL bar.

Mission Objective

Inject a script to pop up a JavaScript **alert()** in the frame below.

Once you show the alert you will be able to advance to the next level.

Solution: Well this level is just an introductory challenge just to show how a XSS works, of course its a *"very basic and non realistic"* example. Anything we type get embedded into the page. Hence, just inject a script tag with an alert code.

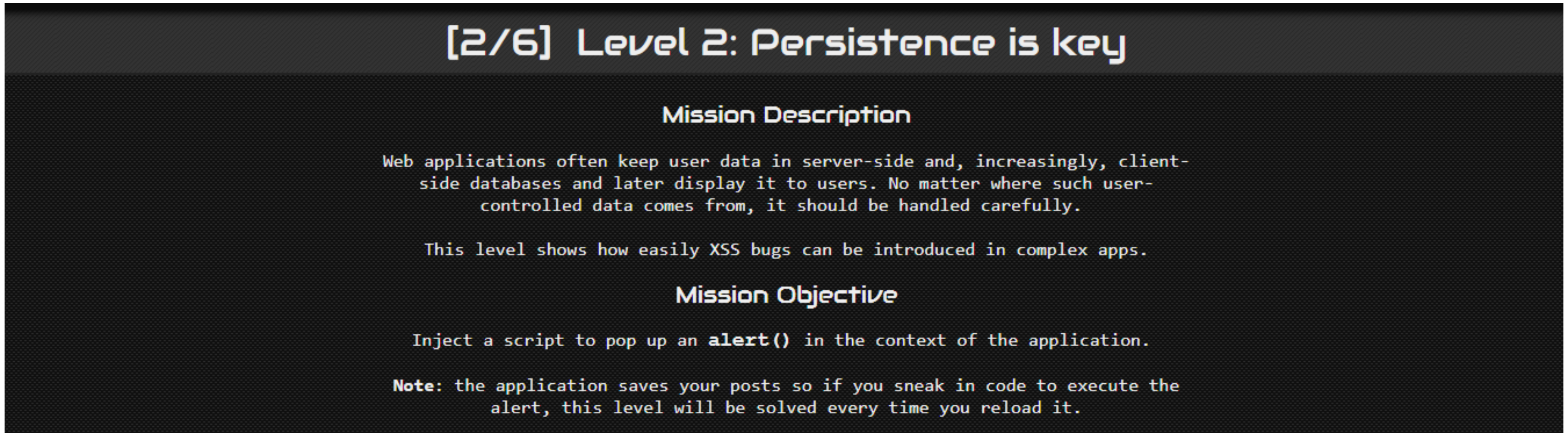
Level 1 Payload:

```
<script>alert()</script>
```

Why does it work: The code is just adding the user input into the webpage with no sanitizing, so it is understood by the browser as part of the page code and executed.

. . .

Level 2: Persistence is key



[2/6] Level 2: Persistence is key

Mission Description

Web applications often keep user data in server-side and, increasingly, client-side databases and later display it to users. No matter where such user-controlled data comes from, it should be handled carefully.

This level shows how easily XSS bugs can be introduced in complex apps.

Mission Objective

Inject a script to pop up an **alert()** in the context of the application.

Note: the application saves your posts so if you sneak in code to execute the alert, this level will be solved every time you reload it.

Solution: Like the previous level, this page's vulnerability is to include HTML straight into the page. But, this time there is a validation which prevent us from using the script tag. To bypass it we can insert a *image tag* with an invalid URL and a onerror attribute which will execute a javascript alert.

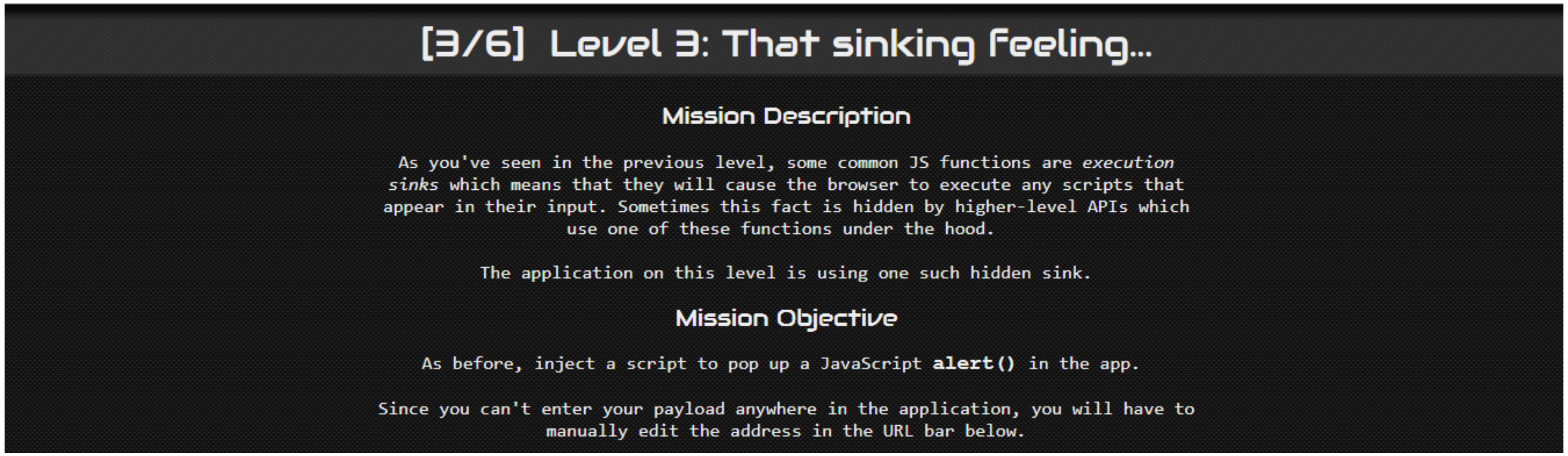
Level 2 Payload:

```
<img src='x' onerror='alert()'>
```

Why does it work: The page will try to load the image from the source '*x*' which will fail miserably and then trigger the *onerror* attribute's code.

. . .

Level 3: That sinking Feeling…



[3/6] Level 3: That sinking feeling...

Mission Description

As you've seen in the previous level, some common JS functions are *execution sinks* which means that they will cause the browser to execute any scripts that appear in their input. Sometimes this fact is hidden by higher-level APIs which use one of these functions under the hood.

The application on this level is using one such hidden sink.

Mission Objective

As before, inject a script to pop up a JavaScript **alert()** in the app.

Since you can't enter your payload anywhere in the application, you will have to manually edit the address in the URL bar below.

Solution: The application chooses the image tab based on first URL fragment, (the hash after the URL). Passing a malicious fragment that will get inserted into the page will trigger the alert.

Level 3 Payload:

```
1' onerror='alert();//
```

Why does it work: The page's source code shows that it gets the URL fragment and passes to a *choosetab* function.

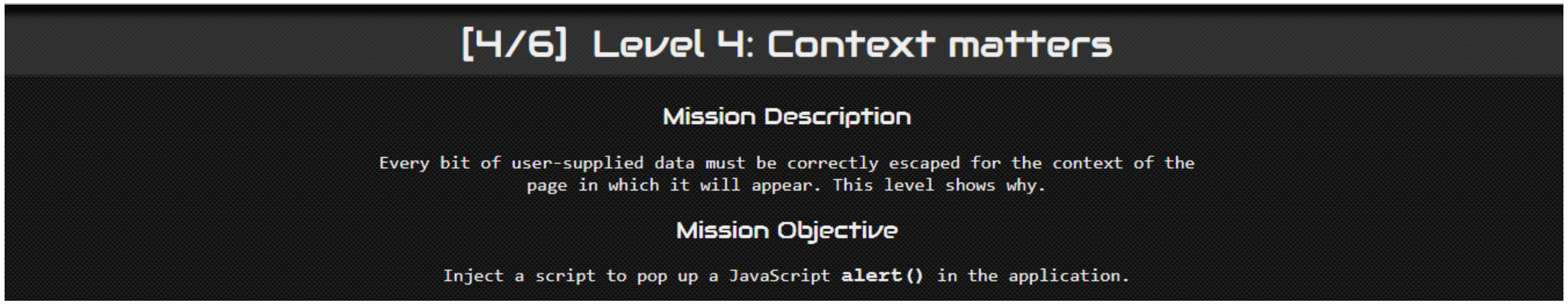
```
window.onload = function() {

chooseTab(unescape(self.location.hash.substr(1)) || "1");

}
```

. . .

Level 4: Context matters



[4/6] Level 4: Context matters

Mission Description

Every bit of user-supplied data must be correctly escaped for the context of the page in which it will appear. This level shows why.

Mission Objective

Inject a script to pop up a JavaScript **alert()** in the application.

Solution: 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 arbitrary code, since it is straightly added to the page.

Level 4 Payload:

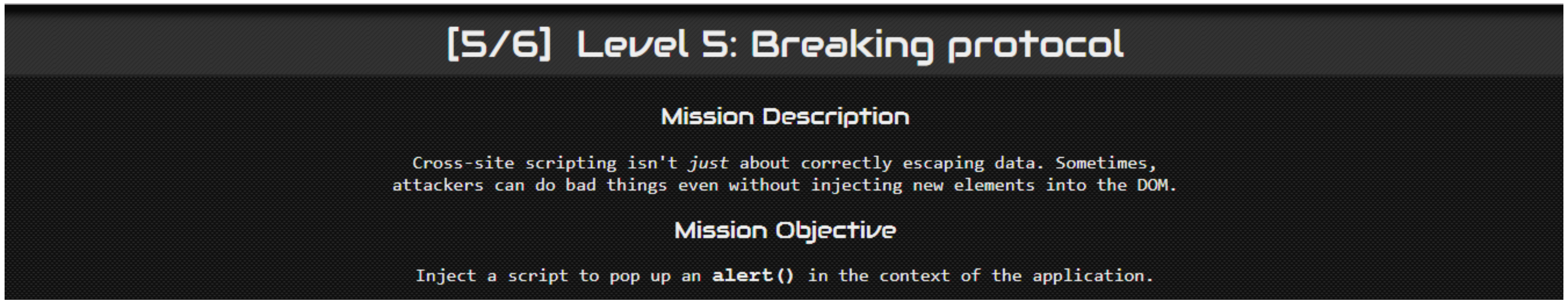
```
3'**alert());//
```

Why does it work: The value we passed by the index page is straightly added into the function parameter at the timer.

```
<img src="/static/loading.gif" onload="startTimer('{{ timer }}');" />
```

. . .

Level 5: Breaking protocol



[5/6] Level 5: Breaking protocol

Mission Description

Cross-site scripting isn't *just* about correctly escaping data. Sometimes, attackers can do bad things even without injecting new elements into the DOM.

Mission Objective

Inject a script to pop up an **alert()** in the context of the application.

Solution 1

Solution: Actually the *welcome* page does nothing, just lead us to the *sign up* page, where the magic happens. At the *sign up* page we can enter an e-mail address (or anything, its not evaluated after all), and hit the link *next* that will lead us to the page passed by the next paramete*r* (e. g. ?next=confirm). We can fool this parameter by injecting javascript code into it, executing the alert function.

Level 5 Payload:

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

Why does it work: The page code just input the parameter next’s value into the href attribute of the next link.

· · ·

Solution 2

Solution: After hitting the next link, you will go to the *confirm* page, that will redirect you to the beginning in a given time. But looking at the *confirm's* source code, we see that it also accepts the next parameter, which is used to redirect you to its given value. We can then, again, mess with this.

Level 5 Payload (need to wait the time runs out):

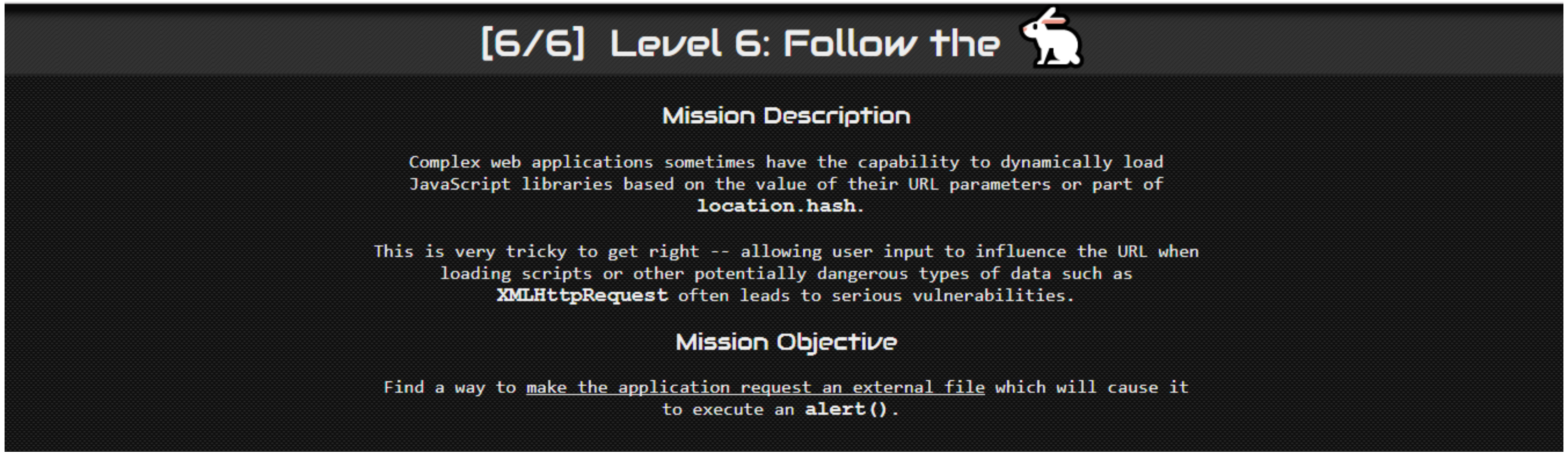
```
https://xss-game.appspot.com/level5/frame
/confirm?next=javascript:alert()
```

Why does it work: The confirm page also accepts the next parameter and uses it to redirect the user to its given value.

```
<script>
```

. . .

Level 6: Follow the 🐇



[6/6] Level 6: Follow the 🐇

Mission Description

Complex web applications sometimes have the capability to dynamically load JavaScript libraries based on the value of their URL parameters or part of **location.hash**.

This is very tricky to get right -- allowing user input to influence the URL when loading scripts or other potentially dangerous types of data such as **XMLHttpRequest** often leads to serious vulnerabilities.

Mission Objective

Find a way to make the application request an external file which will cause it to execute an **alert()**.

Solution: The page adds a script tag with the *src* attribute pointing to the first URL fragment's value. But, before doing so, it evaluates if the fragment begins with the 'http' or 'https' words, to prevent us from loading external files. But we can bypass this validation by omitting the 'http' protocol. For this solution I am using the default file provided by google in the challenge (replacing 'foo' for 'alert').

Level 6 Payload:

```
//www.google.com/jsapi?callback=alert
```

Why does it work: When you omit the protocol in the URL it will inherit the protocol from the current environment (the page). In this case it will inherit the protocol 'https'. More about it here (briefly) and here (complete).

. . .

Congratulations

Congratulations!

You have successfully completed the game!

So we have completed the challenge. Just a last tip: the main skill to exploit the whole application was being able to look for sources and sinks, so if you are not familiarized with this concept, take a look at this awesome video.

How Irated

How did you like the game overall? **A: I loved it!**

How about the difficulty? **A: Too easy.**

Hope you enjoyed the write-up! If you liked, send me some claps 👏, tell me where have you been stuck, if you solved it in a different way or how you rated this challenge in the comments.

See ya! 😁

Ctf Writeup Xss Attack Pentesting JavaScript Hacking