WEB SECURITY HW2

Minzhi He, Yuqing Wang

## 1)XSS exploit generation

### Level 1

Vulnerable source code

message = "Sorry, no results were found for <b>” + query + "</b>."

How to trigger

Input <script>alert("payload")</script>

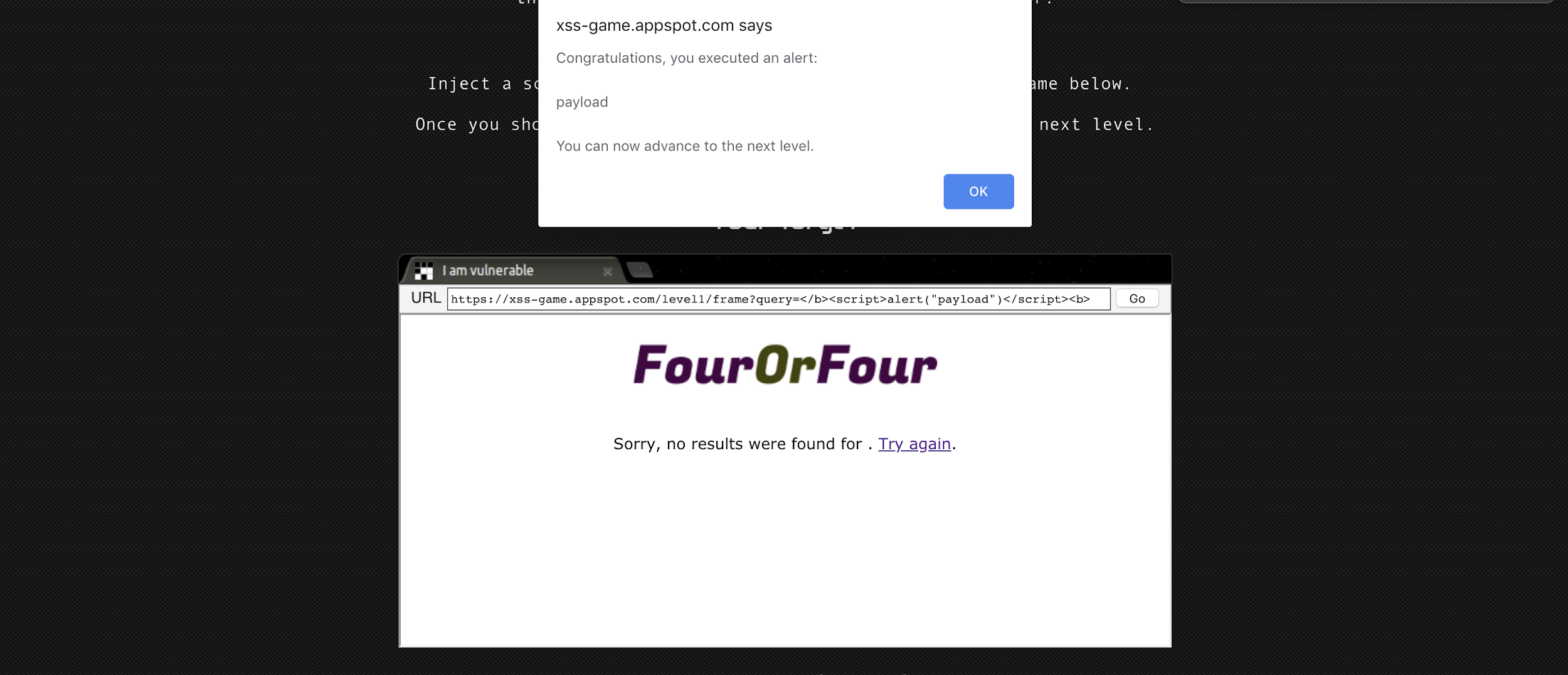


Figure 1.1

### Level 2

Vulnerable source code

html += '<b>You</b>';

html += '<span class="date">' + new Date(posts[i].date) + '</span>';

html += "<blockquote>" + posts[i].message + "</blockquote";

html += "</td></tr></table>"

containerEl.innerHTML += html;

The innerHTML method won’t execute a <script> tag, but a js attribute will work.

How to trigger

<img src="nothing" onerror=alert('payload')>

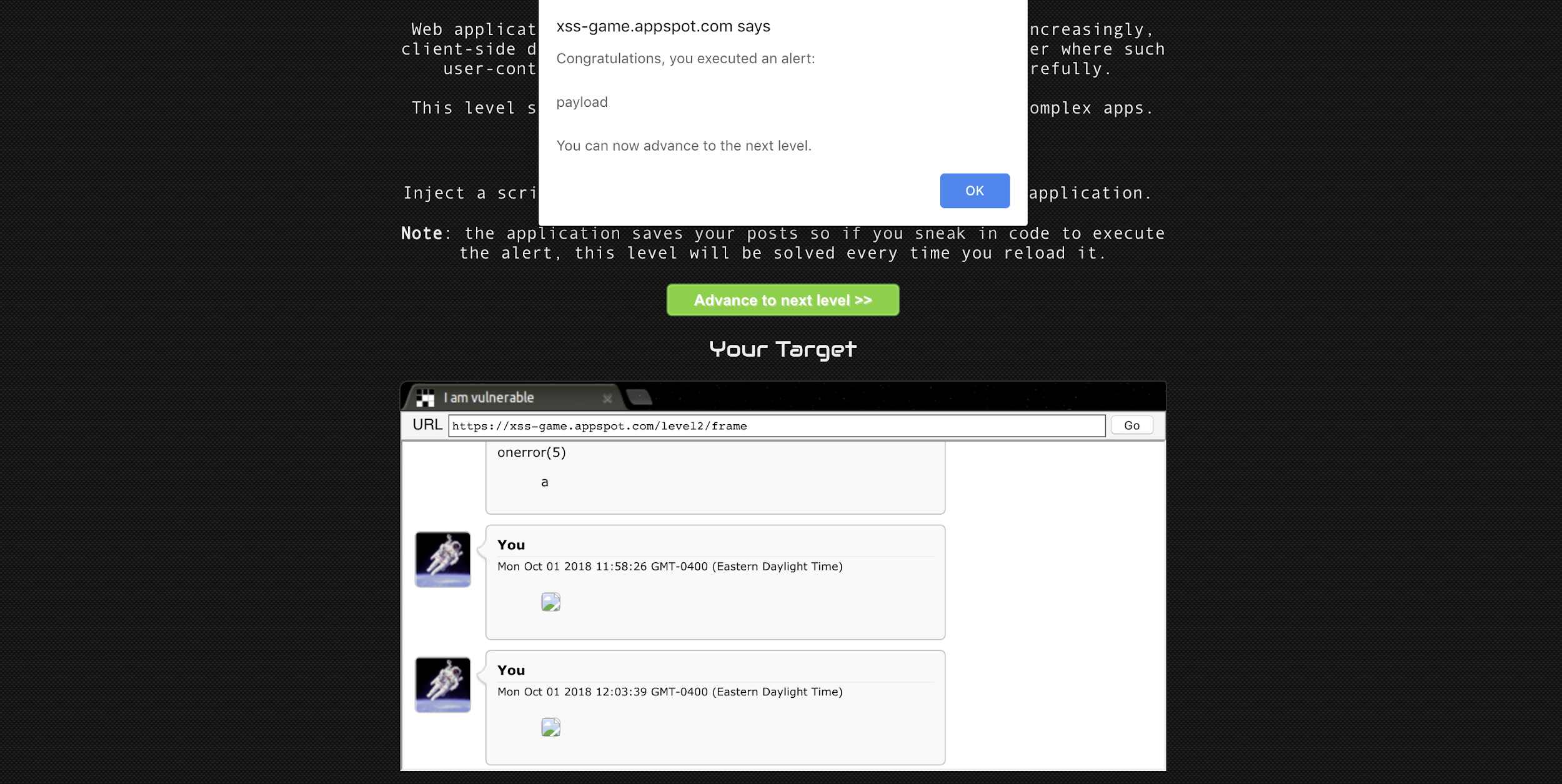


Figure 1.2

### Level 3

Vulnerability Source Code

html += "<img src='/static/level3/cloud" + num + ".jpg’ />";

How to trigger

https://xss-game.appspot.com/level3/frame#4.jpg'onerror=alert('payload')>

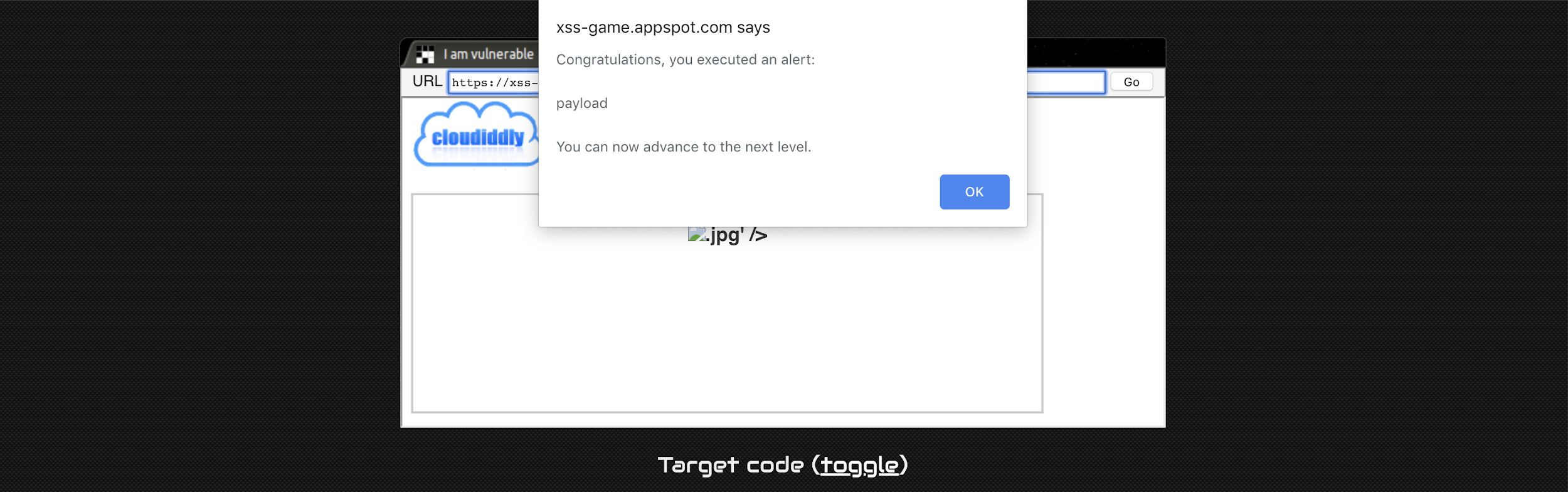


Figure 1.3

### Level 4

Vulnerability Source Code

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

We can use” ’) ” to close the parentheses ，and inject alert afterwards.

How to trigger

Input ‘);alert(‘payload

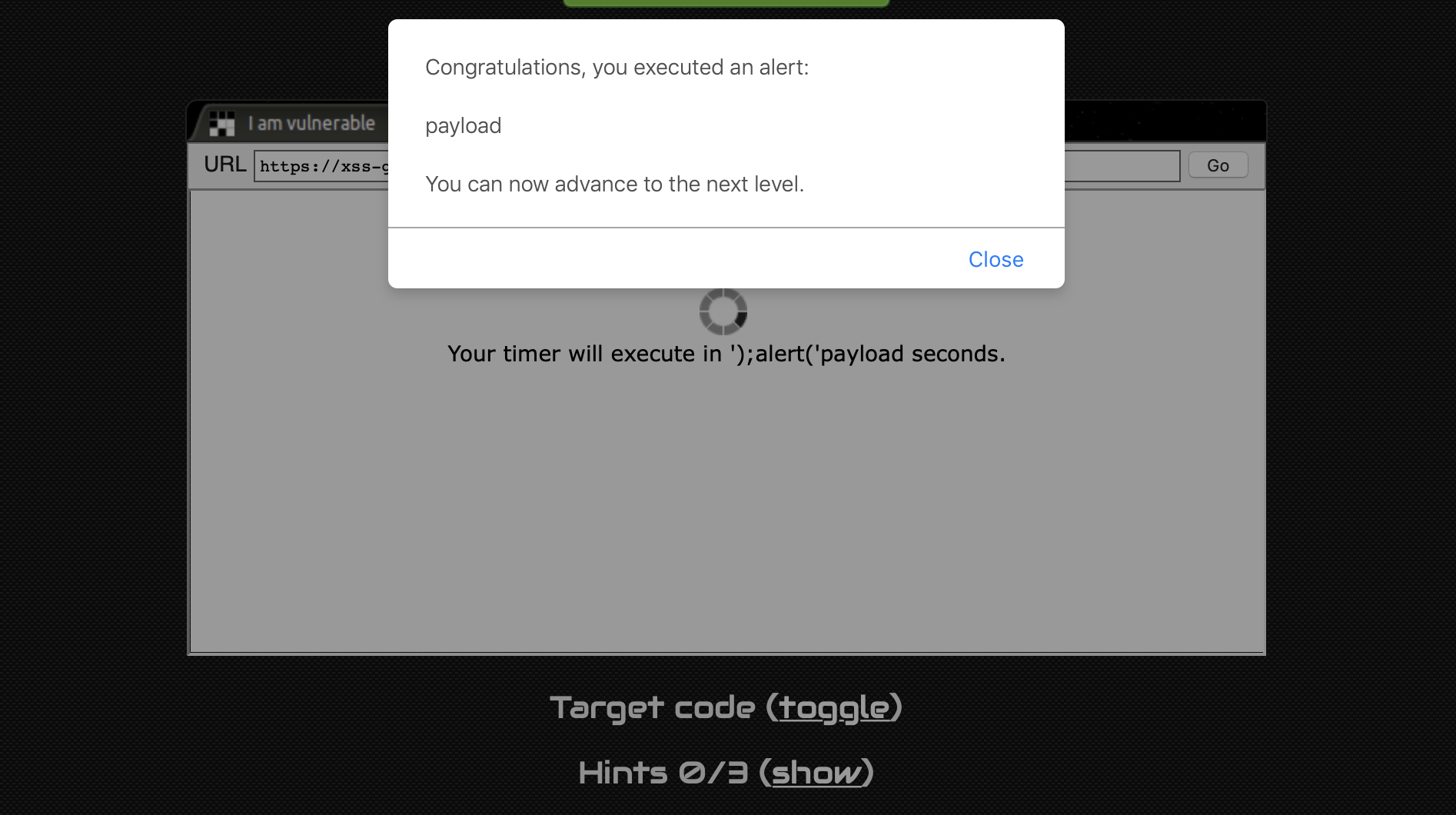


Figure 1.4

### Level 5

Vulnerability Source Code

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



Figure 1.5

How to trigger:

Insert a javascript in the url: https://xss-game.appspot.com/level5/frame/signup?next=javascript:alert(“payload”)

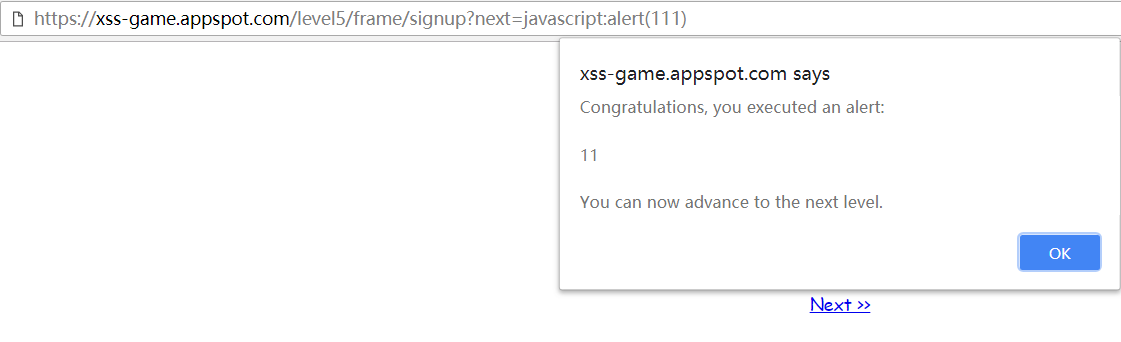


Figure 1.6

### Level 6

Vulnerability Source Code

// Take the value after # and use it as the gadget filename.

function getGadgetName() {

return window.location.hash.substr(1) || "/static/gadget.js";

}

includeGadget(getGadgetName());

Get the value after # and use it as a filename.

function includeGadget(url) {

var scriptEl = document.createElement('script');

// 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;

}

// Load this awesome gadget

scriptEl.src = url;

There is a function to sanitize the input url, which not allows a url begin with http/https.

How to trigger:

The sanitize function can’t filter upper letter such as HTTPS, however the src in the element script can deal with the upper letter link. So we can bypass the function by using upper letter.

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

https://www.google.com/jsapi?callback= is a web page which can create a function with the value of callback in the url. For example, if callback=abc, it will create an abc().



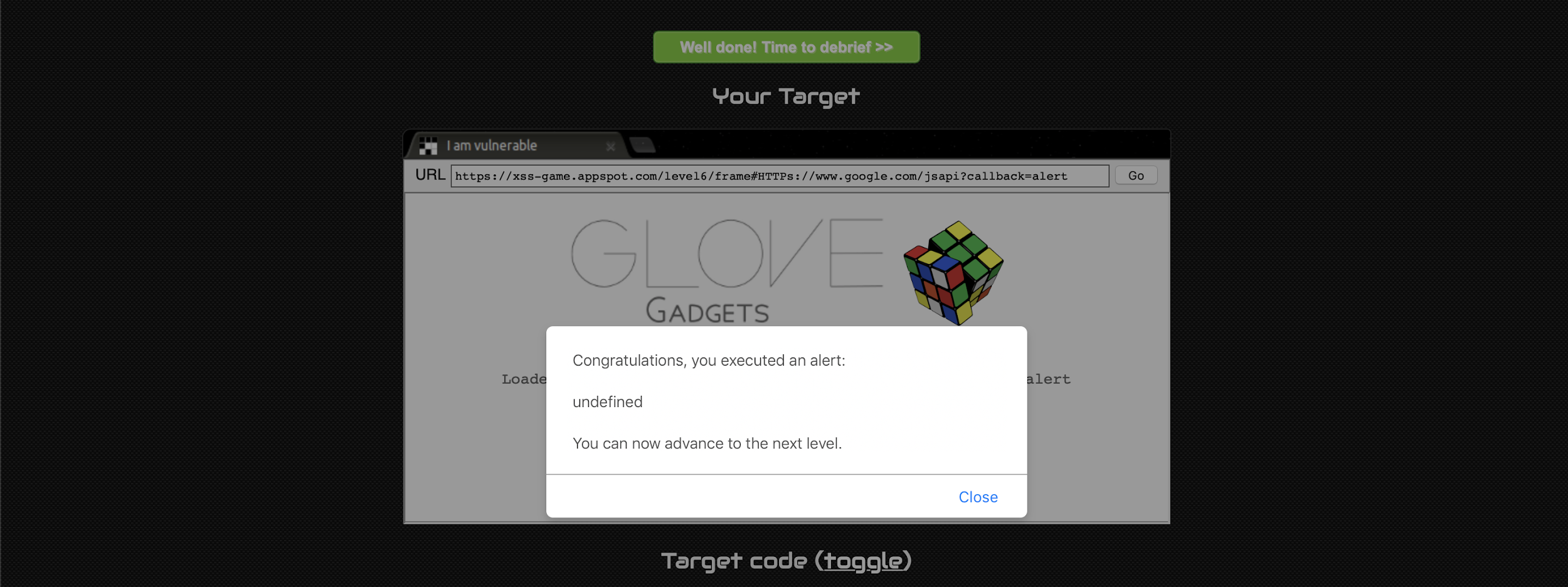
Figure 1.7

Figure 1.8

## 2)Vulnerability Patching

In this section, we rebuild the above website using google app engine. We use google app engine to build our environment since the XSS game is based on google framework. To run our websites, following steps need to be performed:

1. You must have Python version 2.7 on your computer.
2. Download Google Cloud SDK.

NOTE: The latest version Google Cloud SDK have bugs which make you can’t run the website locally. Better use the stable version 218.0.0.

1. Install the gcloud component that includes the App Engine extension for Python.
2. Add directory google-cloud-sdk/bin to your $PATH.
3. Add /usr/local/google\_appengine to your $PYTHONPATH. (export PYTHONPATH=”$PYTHONPATH: /usr/local/google\_appengine”
4. Use command “dev\_appserver.py ./” in the folder which includes the python file and app.yaml to set up the environment and run the websites.

If you can’t run our website, it is a good choice to follow the google official documents and set up the environment for google cloud.

Here is the link: <https://cloud.google.com/appengine/docs/standard/python/download> or you can just contact us :)

We patched all of the six websites, which we will demonstrate below.

To run the website, first we have to enter the directory which has app.yaml. Next, type dev\_appserver.py ./ to start the server. Then type localhost:8080 in chrome.

### Level 1

This website is a search engine, which allows user to input some contents. We patch this website by using a function similar to encodeURI() in python.



In package urllib2, there is a function called quote(), which is the same as encodeURI() in js. We encode ‘<’ and ‘>’, to prevent XSS scripting.



Figure 2.1 The result of patched version

### Level 2

This one is a board which allows user to input and post texts. We patch this website by writing a js function which is called escapeHTML.

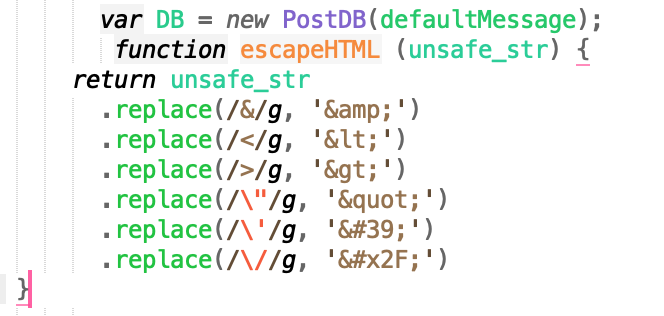


Figure 2.2

This function will encode the illegal input and prevent some malicious injections.

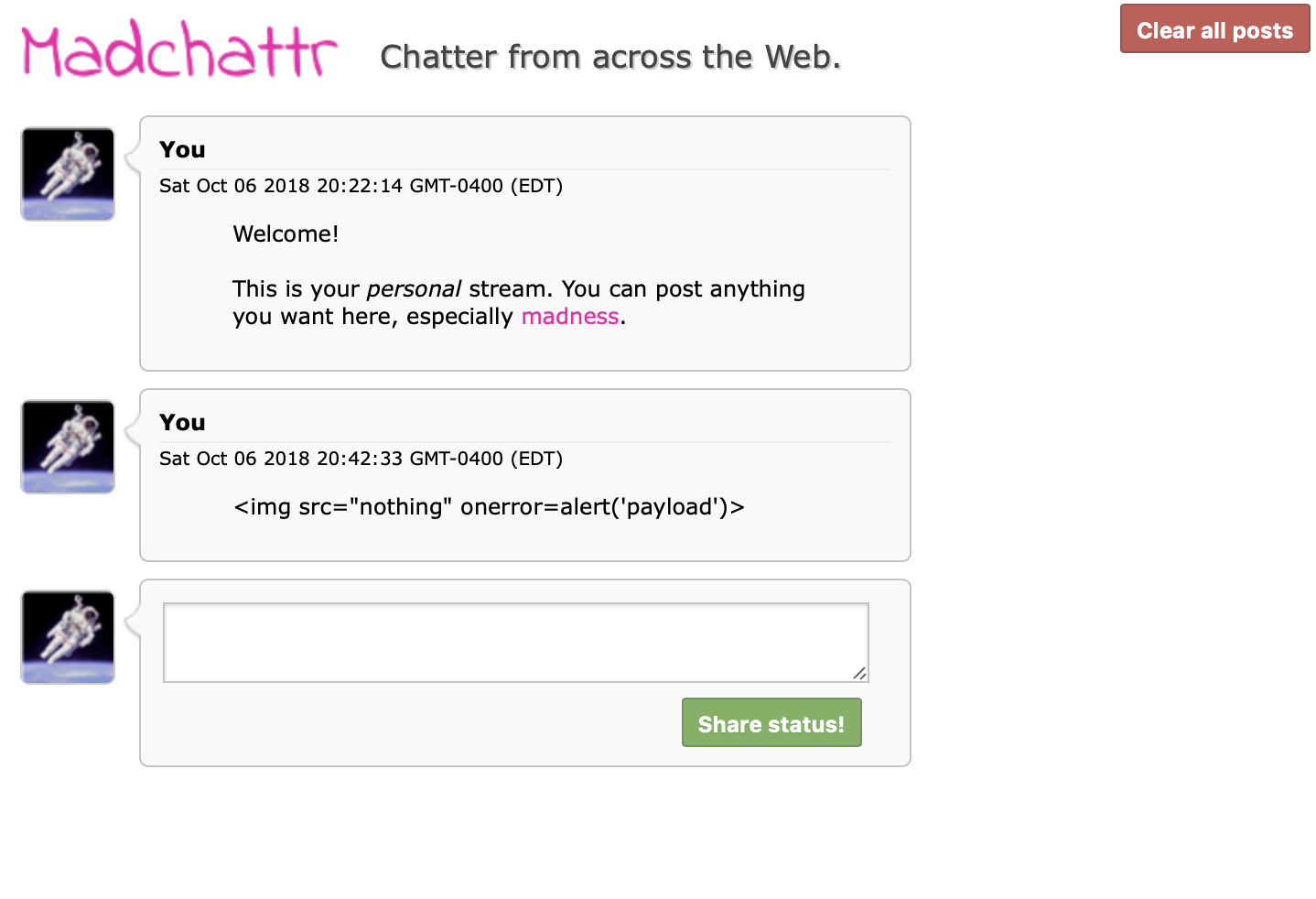


Figure 2.3

### Level 3

This website is an image data center which can switch to different images. We use a parseInt() function to filter all the input.



Figure 2.4

We found that the webpage has some bugs. The webpage will call the pictures many times in one second and never stop. The webpage will flash a lot. We didn’t change any code so we think it is not our problems.

### Level 4

This website is a timer. The original website does not check user input, so malicious code can be injected. We patch this website by adding a function on server side, which change the type of user input to float. If encounters a value error, we set the timer to three.

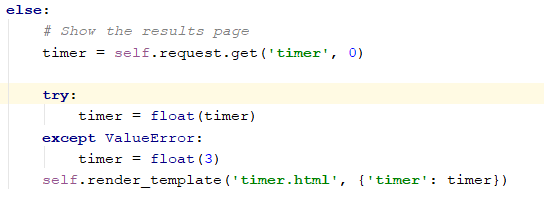


Figure 2.5

### Level 5

In this level, the mainpage is a welcome page, and when click sign up, it jumps to signup.html, and in sign up, when clicked next, the browser jumps to confirm.html, and after five seconds, the browser jumps to welcome page again.

In the original website, it uses template to get the value of next, which is completely unnecessary, as we already aware of which page we will jump to. So we remove the template, and replace it with the parameters that represents each page.

To run this website, we need to type <http://localhost:8080/level5/frame/welcome> instead of localhost:8080, which is a little different from others.

### Level 6

This one allow users to dynamically load JavaScript libraries based on the value of their URL parameters.

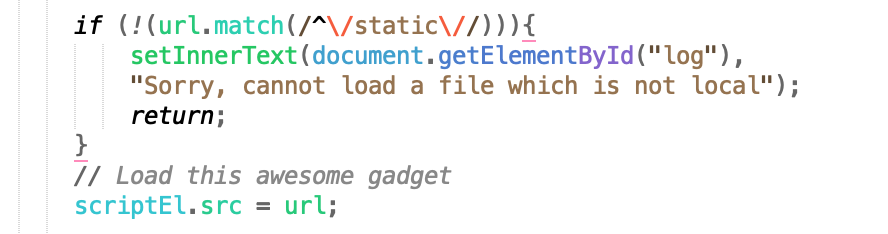


Figure 2.6

Since the blacklist doesn’t work well, we patch this one by writing a whitelist which only allows users to load the local files.



Figure 2.7

## 3)Defense via Content Security Policy

### CSP 1.0

### Level 1

In Level 1, we add

<meta http-equiv=”Content-Security-Policy” content=”default-src ‘self’>

to load resources only from its own origin.

### Level 2

In Level 2, we add

<meta http-equiv=”Content-Security-Policy” content=”default-src ‘self’>

According to CSP 1.0 documents, the web page can’t execute an inline script either from a script element or from an inline event handler. Whenever the user agent execute script contained in javascript URI, instead it must not execute the script.

Whenever the user agent would execute an inline script (either from a script element or from an inline event handler), instead the user agent must not execute script.

Whenever the user agent would execute script contained in a javascript URI, instead the user agent must not execute the script. (The user agent should execute script contained in "bookmarklets" even when enforcing this restriction.)

### Level 3

In Level 3, we add

<meta http-equiv="Content-Security-Policy" content="img-src 'self';

object-src 'self';

child-src 'self';

style-src 'self'">

In this one, the web will call the JQuery function from “//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js” which is a safe source in this situation. Since the CSP 1.0 will disable the outside script by default, we allow the website to use JQuery function in this one.

### Level 4

In level 4, the image has an onload event that is rejected by CSP 1.0.According to CSP 1.0, If 'unsafe-inline' is not in allowed script sources:

Whenever the user agent would execute an inline script (either from a script element or from an inline event handler), instead the user agent must not execute script.

Whenever the user agent would execute script contained in a javascript URI, instead the user agent must not execute the script. (The user agent should execute script contained in "bookmarklets" even when enforcing this restriction.).

Function onload is an inline function, so we have to remove this function.



Figure 3.1

And we add an EventListener in our scripts to fulfill the requirements.

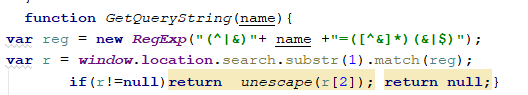


Figure 3.2

### Level 5 & 6

Level 5&6 are safe ones. We just add the strict rules of CSP 1.0

<meta http-equiv=”Content-Security-Policy” content=”default-src ‘self’>

And move the js script to our static folder, and they work well.

## CSP 2.0

### Level 1

Since Level 1 doesn’t use any script, it is not necessary to change the code.

### Level 2

Since “CSP 2.0 offers backward compatibility for inline scripts by allowing you to whitelist specific inline scripts using either a cryptographic nonce (number used once) or a hash”, we use timestamp to generate a dynamic nonce which allow specific inline scripts.

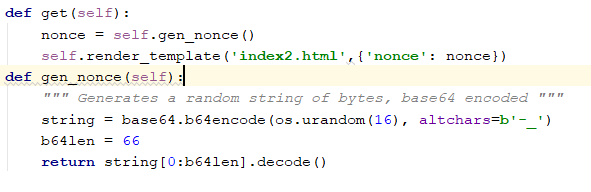


Figure 3.3

We claim the use of nonce in the header.



Figure 3.4

When we want to use the dynamic nonce for our scripts, we can just add the value nonce in the script element.

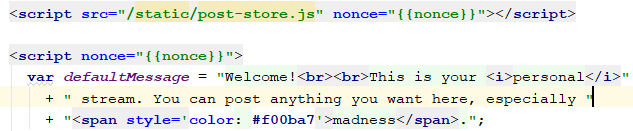


Figure 3.5

### Level 3

We tried to use addEventListener to avoid the inline scripts. However, we found that there will be some bugs if we use this way. The websites will call the function continuously. So we allow the unsafe inline in this one to maintain the function work.

### 

Figure 3.6

### Level 4

In this one, CSP 2.0 is similar to CSP 1.0. We still can’t use onload function by default since it is an unsafe-inline. We use addEventListener to replace the onload and add the dynamic nonce for it to secure the web.

### Level 5

In this one, we tried the same way in the Level 4, however it doesn’t work. The template can’t be sent in this one and we didn’t figure out the reason. So we use a static nonce in this one. Although it might not be as safe as the dynamic one, it can still monitor the use of scripts to some degrees.

### Level 6

In this one, similar to level 4, we add a dynamic nonce at each script label, and we add nonce to the meta label as shown below. We use template to pass the random nonce, which is also used in the original website.

