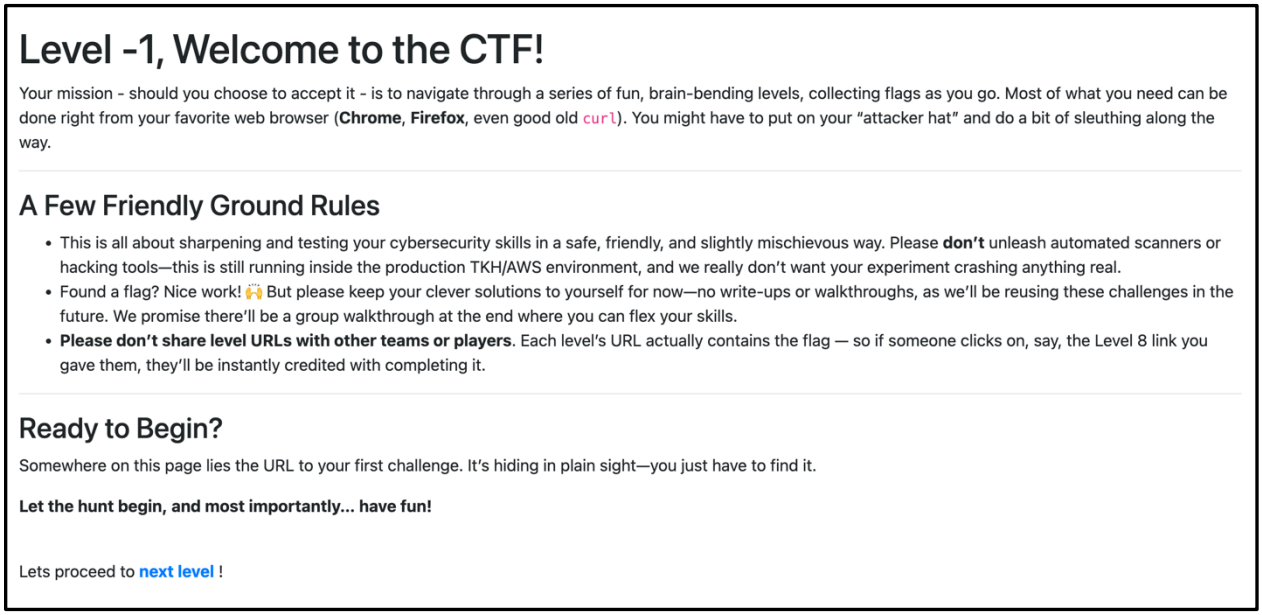
Team B - CTF Documentation

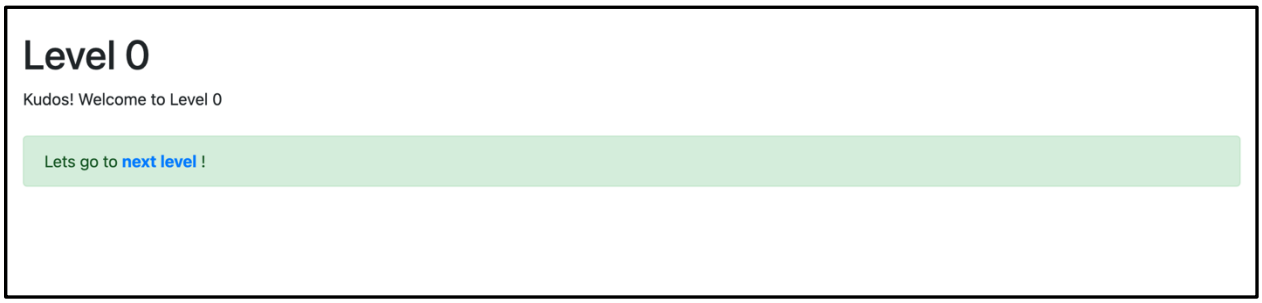


Objective: Locate the URL to begin the challenge.

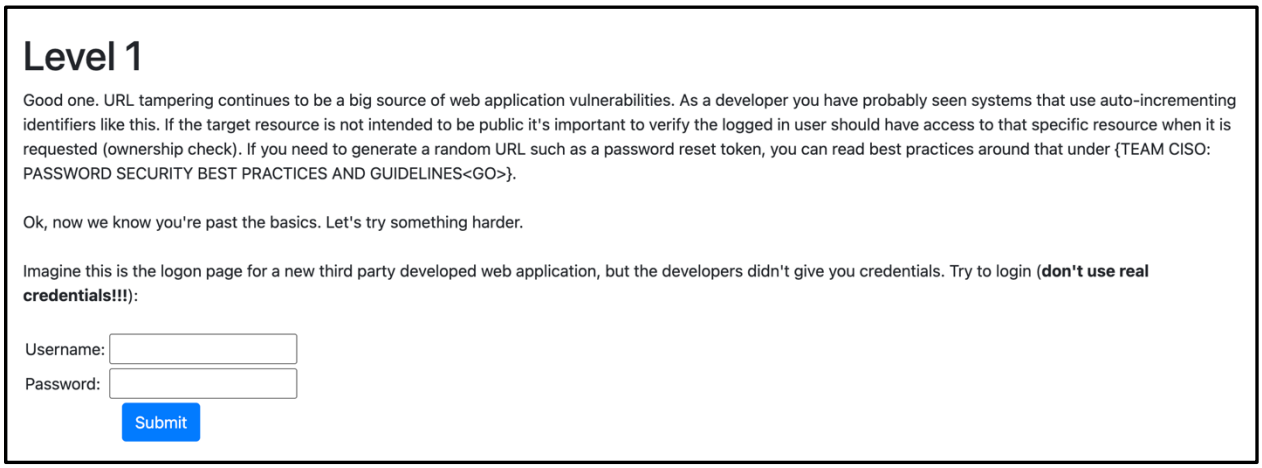
- **Approach:** Right-clicked the page → **View Page Source**.
- **Discovery:** Found a comment hidden in the HTML:

php-template
CopyEdit
-->
<!-- Always read the source, Luke! The next level is at /levels/level0

- **Action:** Manually visited /levels/level0 to proceed.

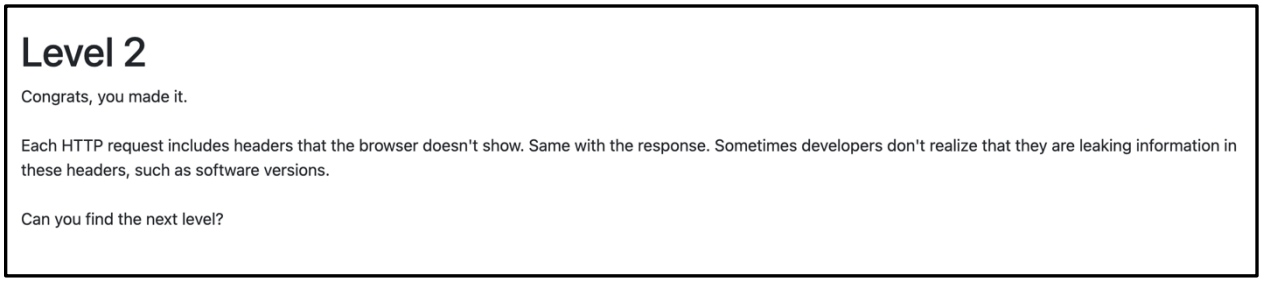


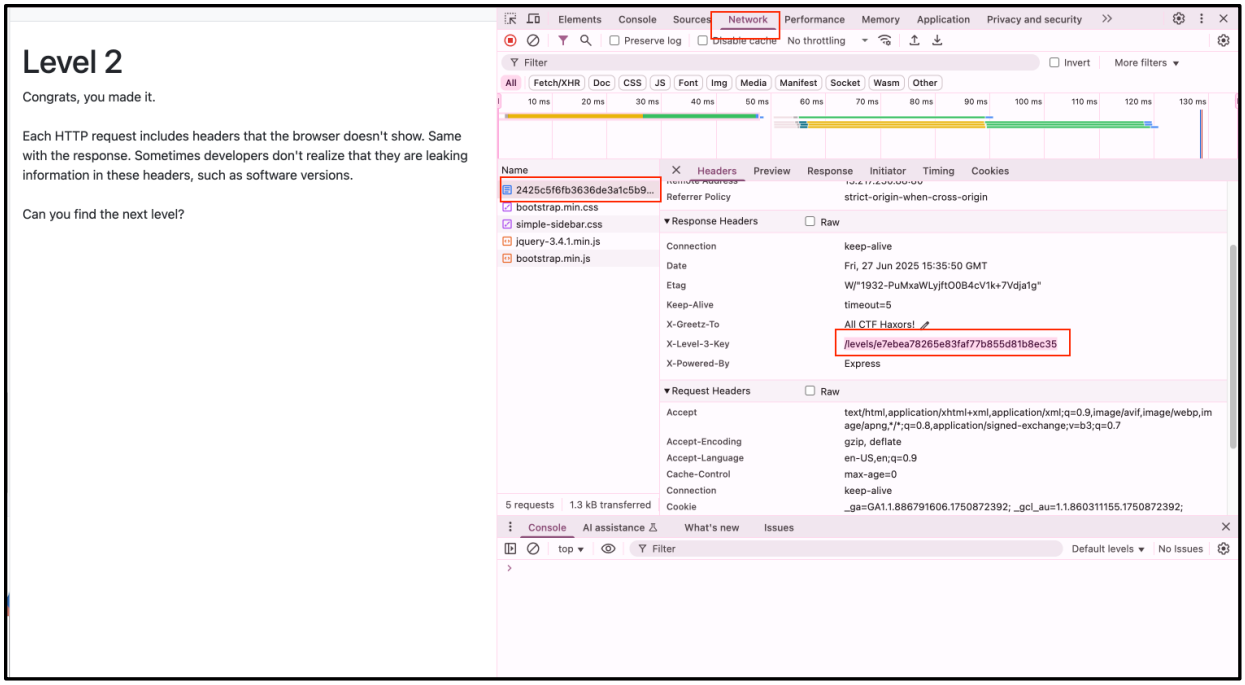
To reach to level 1 change the URL from 0 to 1



I have guessed common default credentials and admin worked

- Username: admin
- Password: admin





- **Right-click >Inspect**
- Go to > **Network tab**
- **Reload the page** while the Network tab is open.
- Click on the **first request**
- click > **Headers** tab.
- Scroll down to > **Response Header**

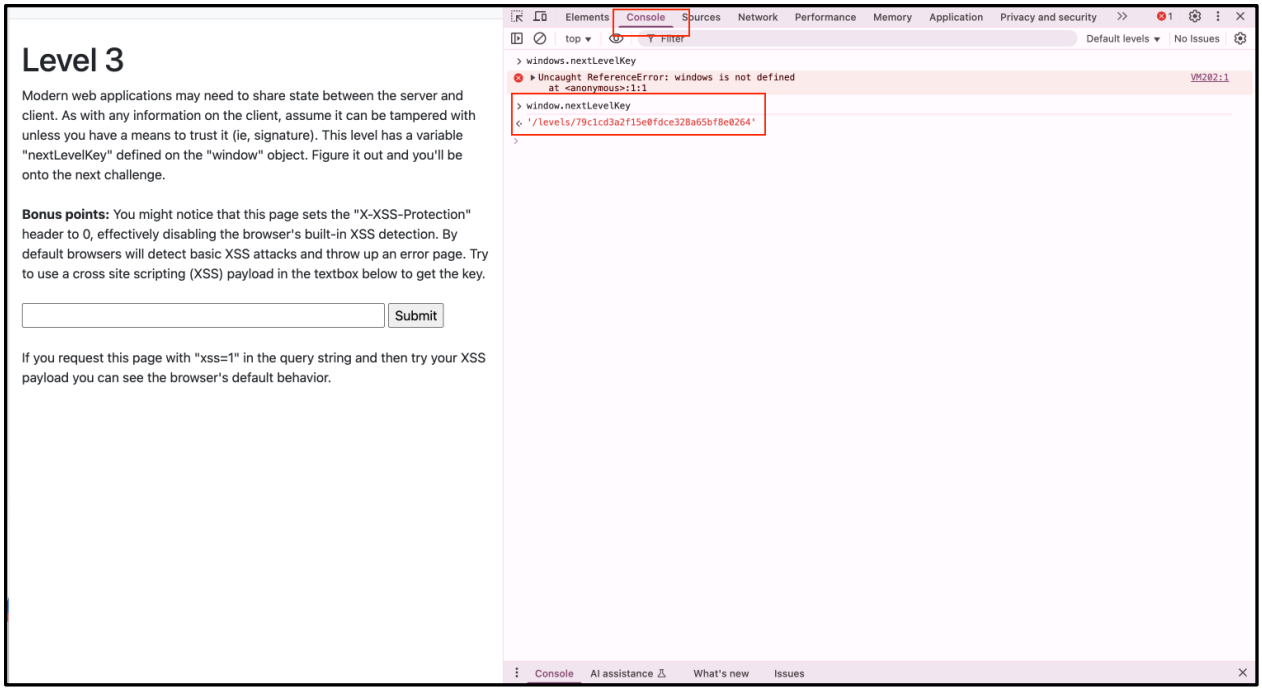
Level 3

Modern web applications may need to share state between the server and client. As with any information on the client, assume it can be tampered with unless you have a means to trust it (ie, signature). This level has a variable "nextLevelKey" defined on the "window" object. Figure it out and you'll be onto the next challenge.

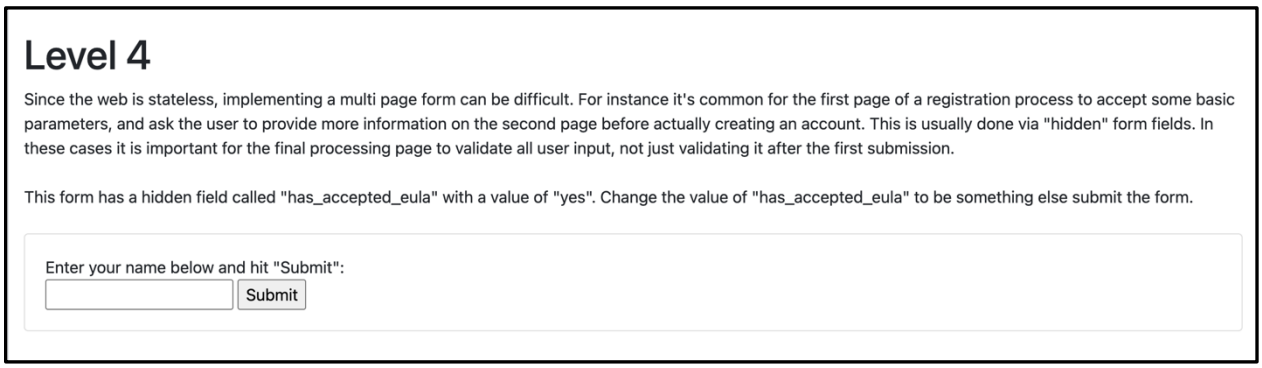
Bonus points: You might notice that this page sets the "X-XSS-Protection" header to 0, effectively disabling the browser's built-in XSS detection. By default browsers will detect basic XSS attacks and throw up an error page. Try to use a cross site scripting (XSS) payload in the textbox below to get the key.

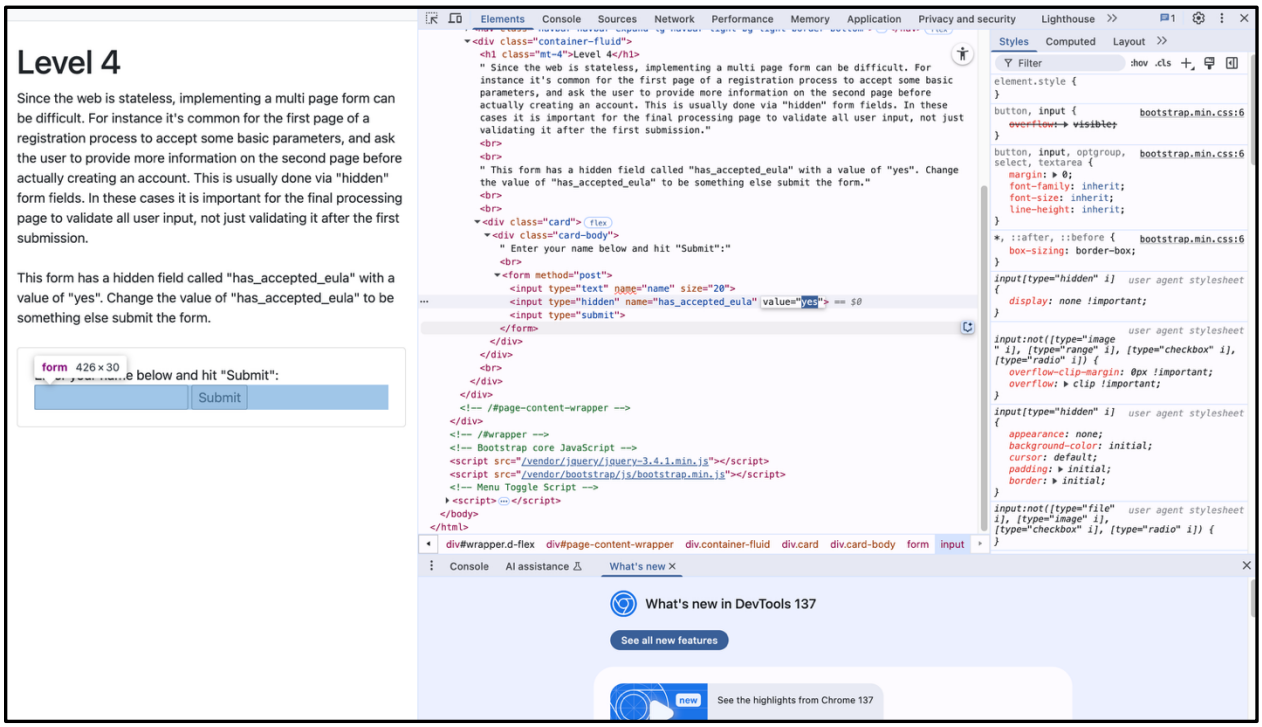
Submit

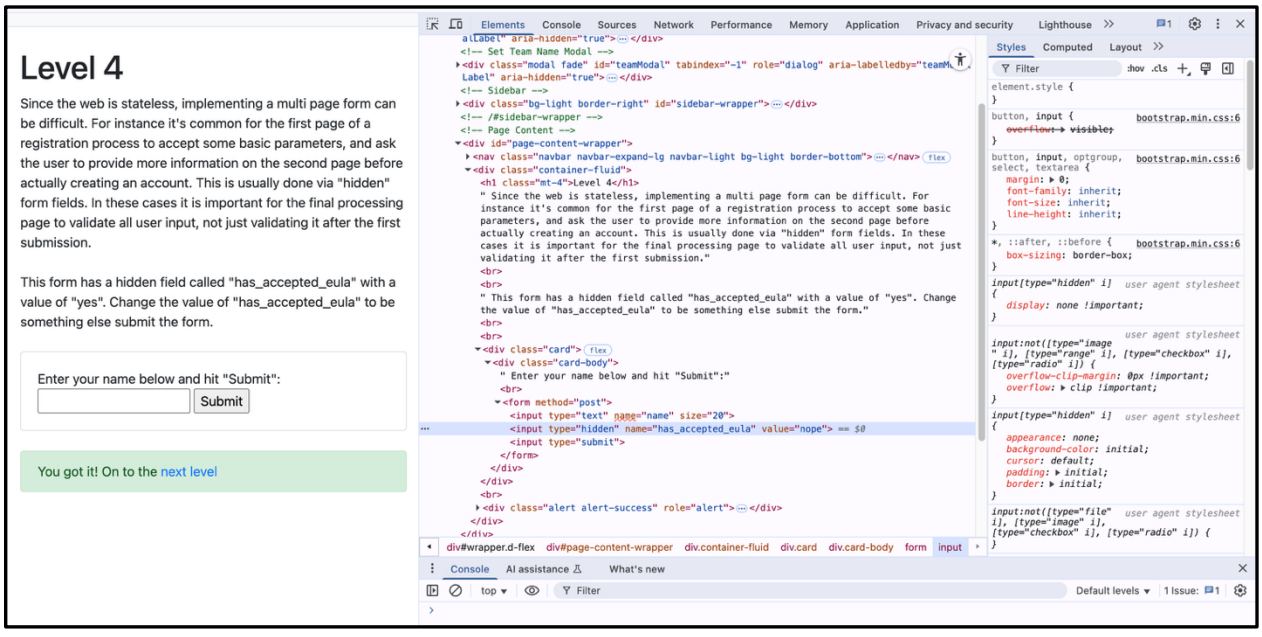
If you request this page with "xss=1" in the query string and then try your XSS payload you can see the browser's default behavior.



- Right-click > **Inspect**
- Go to the **Console** tab
- Type: window . nextLevelKey







- **Inspect** > **Elements** tab
- Find the hidden input in the form:

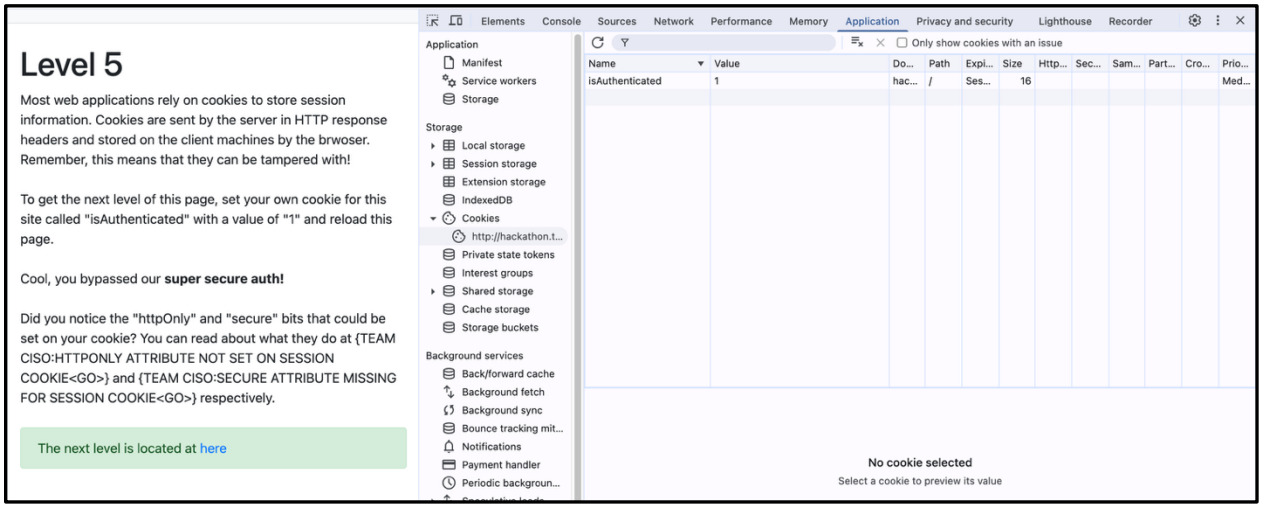
    - `<input type="hidden" name="has_accepted_eula" value="yes">`
- Double-click on the `value="yes"` and change it to something like:
    - `value="nope"`
- Click hit **Submit/ Enter**

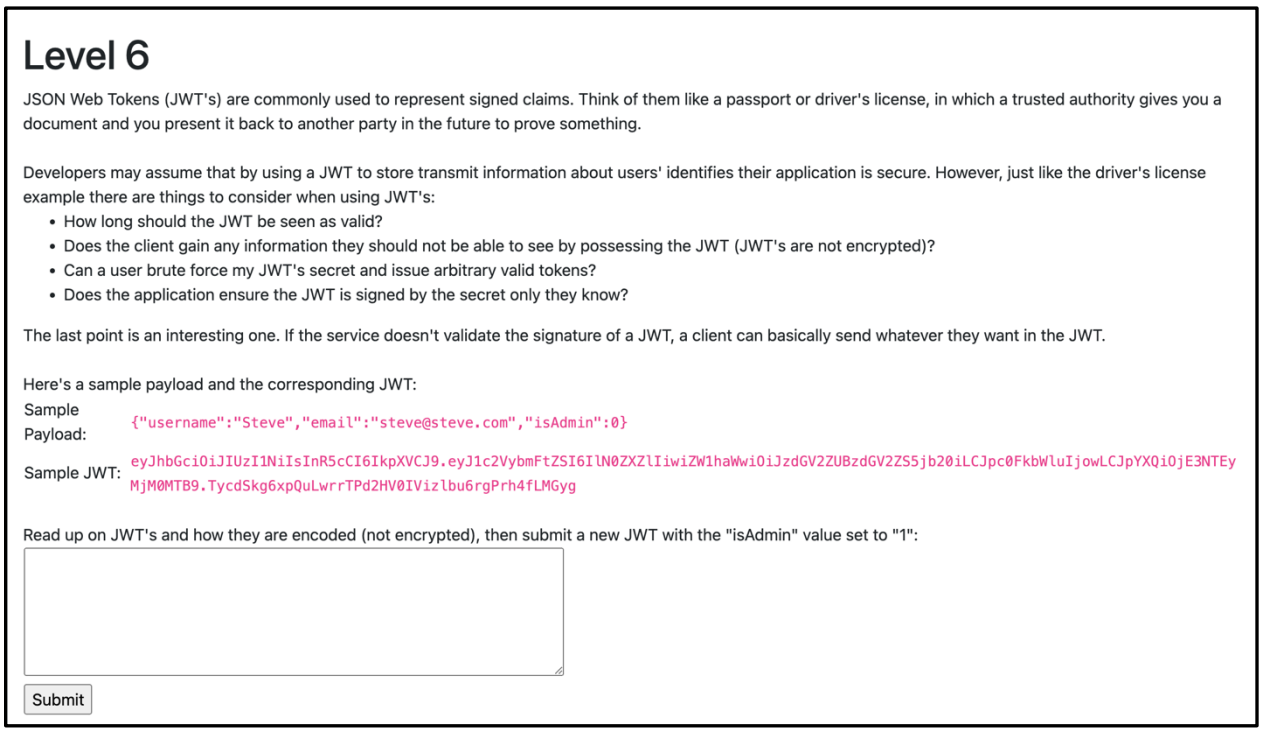
Level 5

Most web applications rely on cookies to store session information. Cookies are sent by the server in HTTP response headers and stored on the client machines by the brwoser. Remember, this means that they can be tampered with!

To get the next level of this page, set your own cookie for this site called "isAuthenticated" with a value of "1" and reload this page.

- Right-click > **Inspect**
- Go to > **Application** tab
- Click > **Cookies"**
- Add a new cookie: (double click to add)
    - **Name:** `isAuthenticated`
    - **Value:** `1`
- **Reload the page**





Objective: Elevate privileges via JWT token.

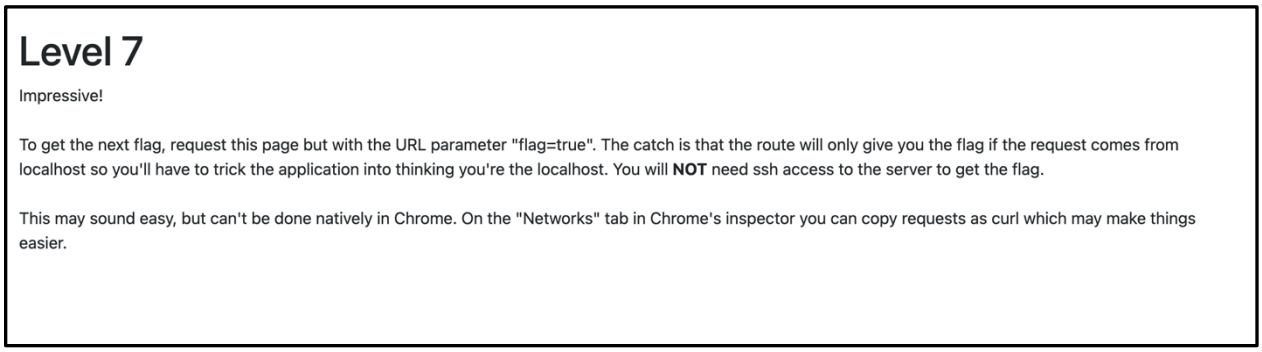
Action:

- Used jwt.io to decode original JWT.
- Discovered the secret: a-string-secret-at-least-256-bits-long
  Recreated JWT with:

  {
  "username": "Steve"
  ,
  "email": "steve@steve.com"
  ,
  "isAdmin": 1
  }

- Signed with correct secret.

Result: Authenticated as admin → Reached Level 7.



Objective: Trick the server into thinking the request came from localhost.

Approach:

- Appended ?flag=true to the URL.

  Used curl to spoof the IP:

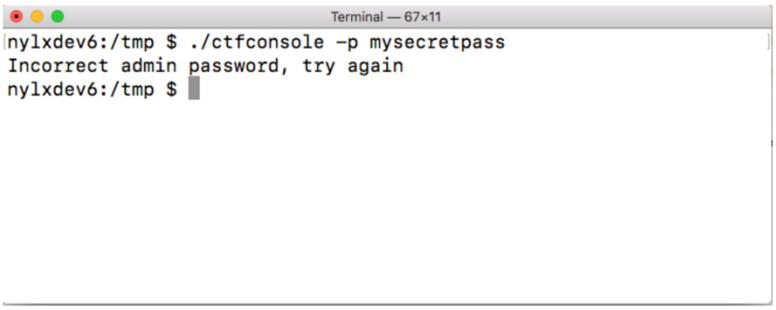
  curl -H "X-Forwarded-For: 127.0.0.1" "http://hackathon.theknowledgehouse.org/levels/015c8d5e17455064e50c407cf8ec7e1e?flag=true"

- **Outcome:** Server believed the request came from 127.0.0.1 and responded with the flag/next level.

Level 8

Lets try some very elementary binary reverse engineering for a change now!

We've got a 64-bit linux executable binary for you to download. This binary works on 64-bit linux (ubuntu/debian) machines. This program requires a password to access to the 'admin' functionality. Your mission is to reverse engineer the binary and find the password.



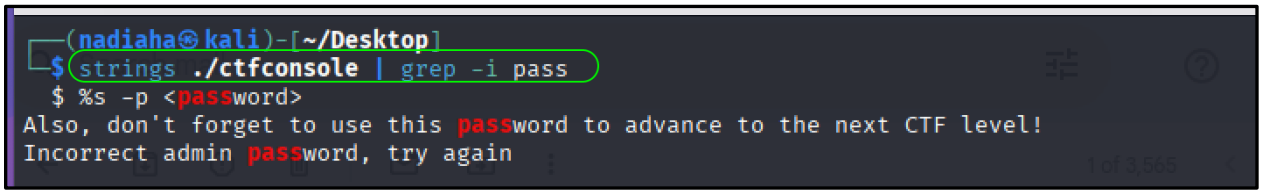
A few notes:

- Please do not brute force password on TKH's shared infrastructure. This challenge doesn't require brute forcing passwords.

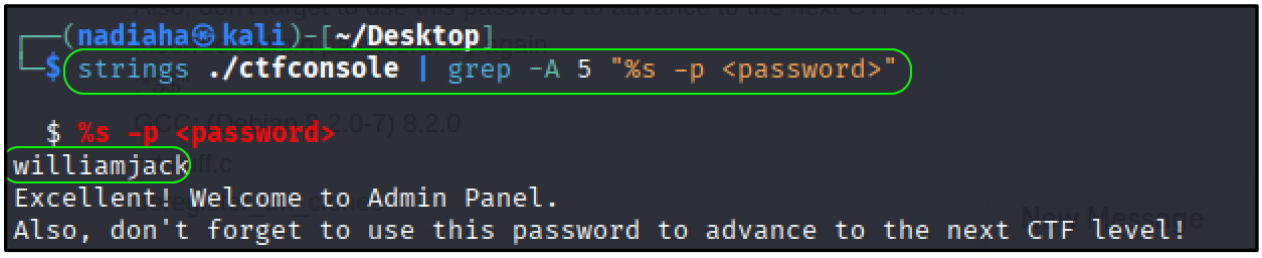
Enter the password that works for the binary above:

Submit

- Cd to where the file is downloaded and execute the following command:



- Output: **“ %s -p <password>”**
- Then execute the following command:



- Password : williamjack

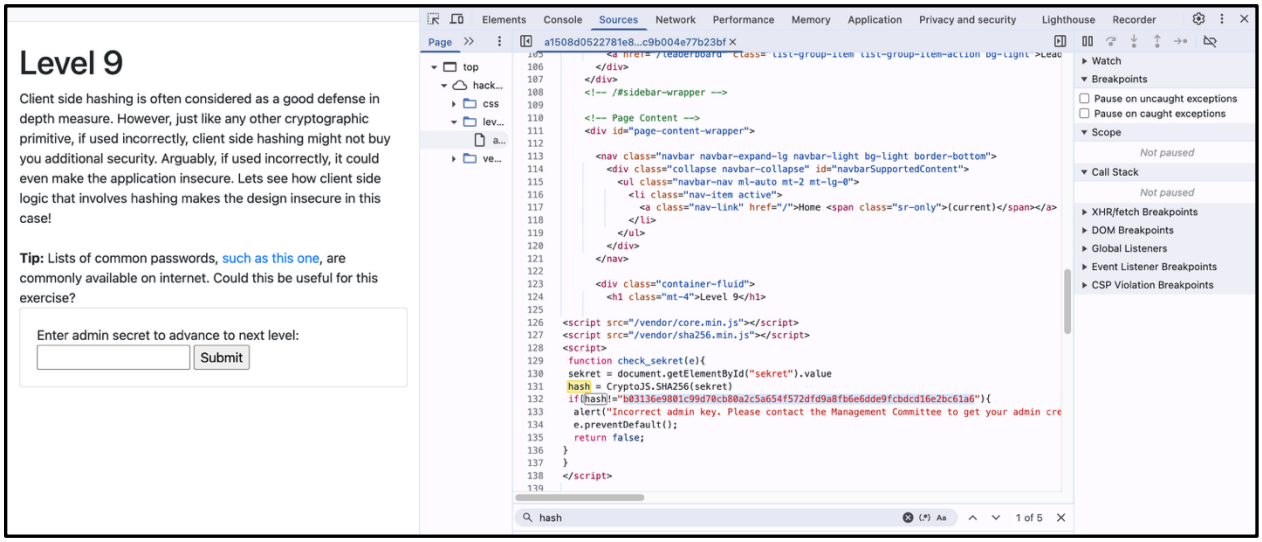
Level 9

Client side hashing is often considered as a good defense in depth measure. However, just like any other cryptographic primitive, if used incorrectly, client side hashing might not buy you additional security. Arguably, if used incorrectly, it could even make the application insecure. Lets see how client side logic that involves hashing makes the design insecure in this case!

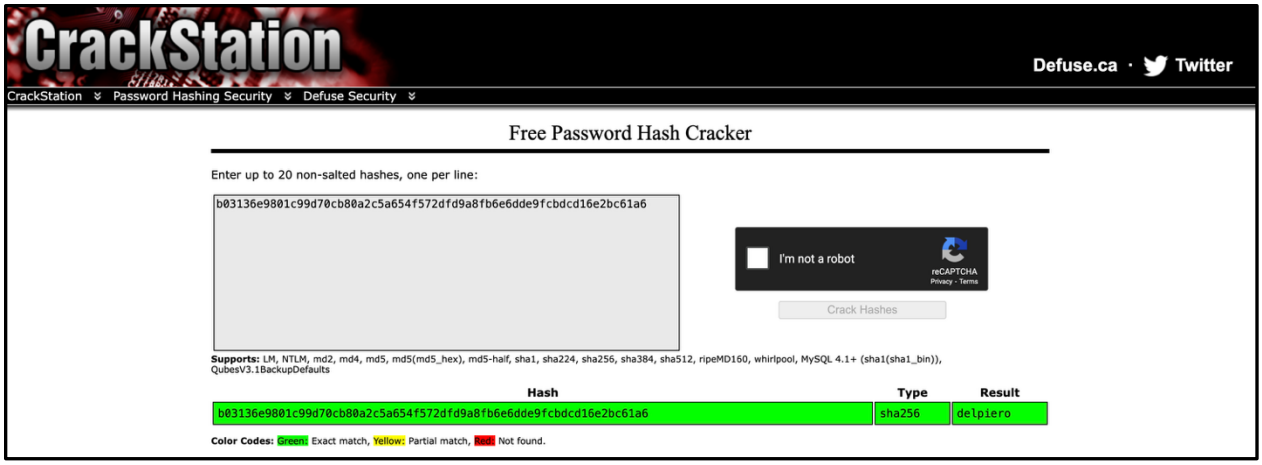
Tip: Lists of common passwords, such as this one, are commonly available on internet. Could this be useful for this exercise?

Enter admin secret to advance to next level:

Submit



- Right-click > **Inspect**
- Go to > **source** tab
- Inspect HTML/JS > find the Hash
- Take the hash Value and **Extract it**
  “b03136e9801c99d70cb80a2c5a654f572dfd9a8fb6e6dde9fcbdcd16e2bc61a”
- Use tool to reverse the hash: https://crackstation.net/



- Hash result: delpiero

Level 10

Open redirect vulnerabilities usually appear when an application tries to return the user to the previous page they were on. You have probably seen this type of pattern in URL's before:

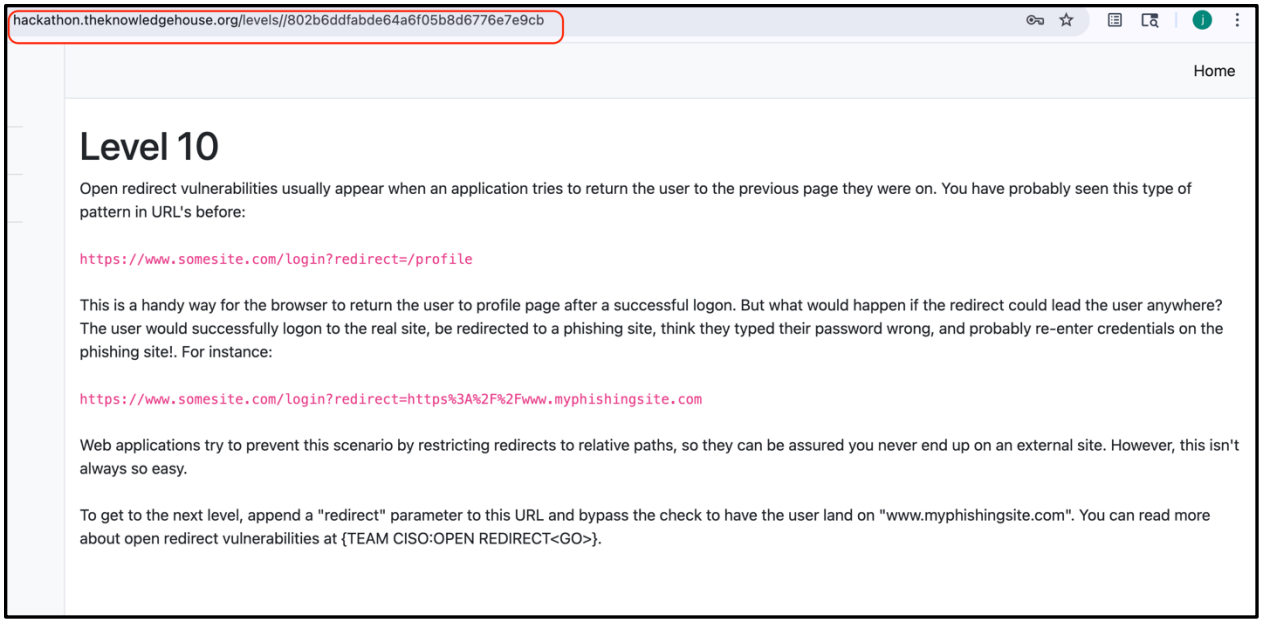
`https://www.somesite.com/login?redirect=/profile`

This is a handy way for the browser to return the user to profile page after a successful logon. But what would happen if the redirect could lead the user anywhere? The user would successfully logon to the real site, be redirected to a phishing site, think they typed their password wrong, and probably re-enter credentials on the phishing site!. For instance:

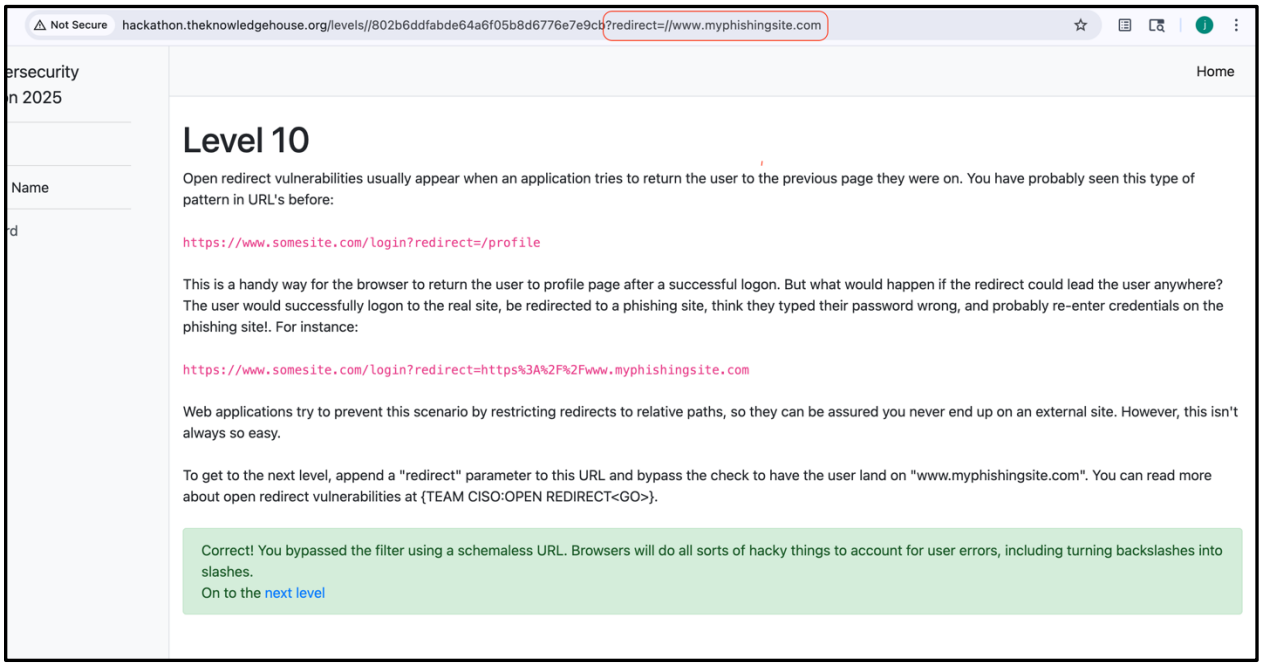
`https://www.somesite.com/login?redirect=https%3A%2F%2Fwww.myphishingsite.com`

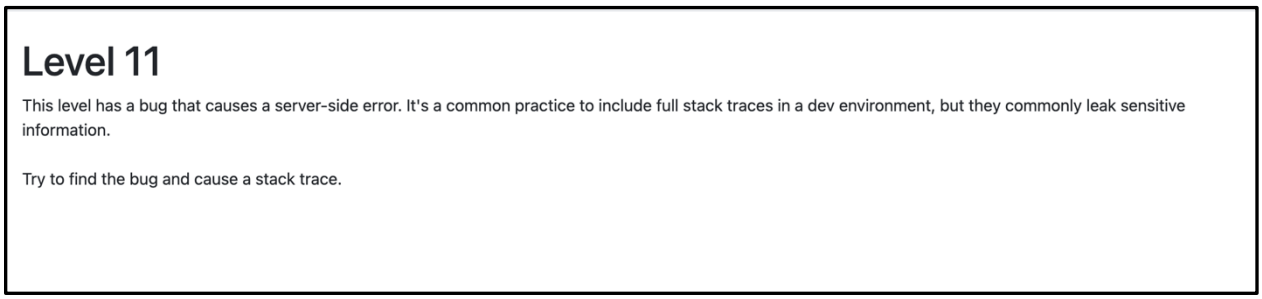
Web applications try to prevent this scenario by restricting redirects to relative paths, so they can be assured you never end up on an external site. However, this isn't always so easy.

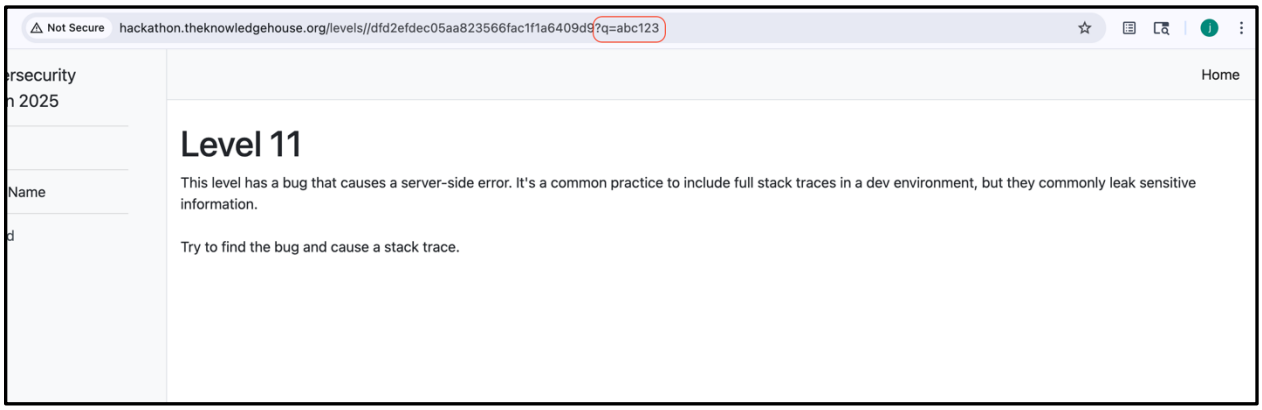
To get to the next level, append a "redirect" parameter to this URL and bypass the check to have the user land on "www.myphishingsite.com". You can read more about open redirect vulnerabilities at {TEAM CISO:OPEN REDIRECT<GO>}.



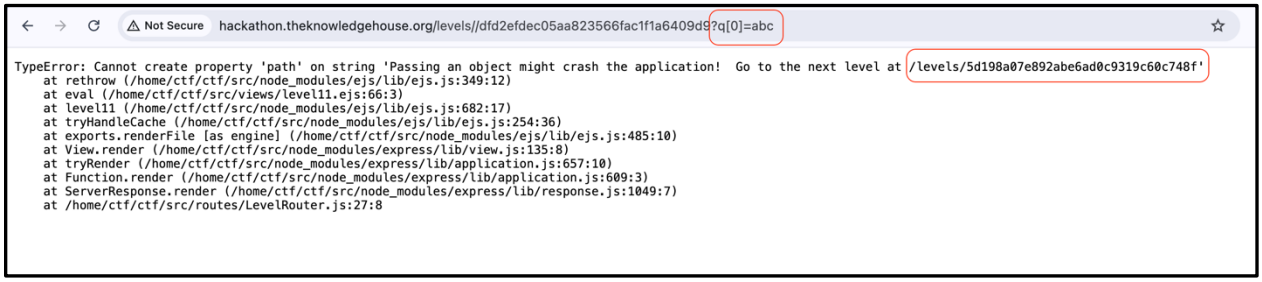
- Take the URL
- “http://hackathon.theknowledgehouse.org/levels/802b6ddfabde64a6f05b8d6776e7e9cb”
- and add “?redirect=//www.myphishingsite.com” at the end







- Change the end of the URL and add "?q[0]=abc" at the end
http://hackathon.theknowledgehouse.org/levels//dfd2efdec05aa823566fac1f1a6409d9?q[0]=abc

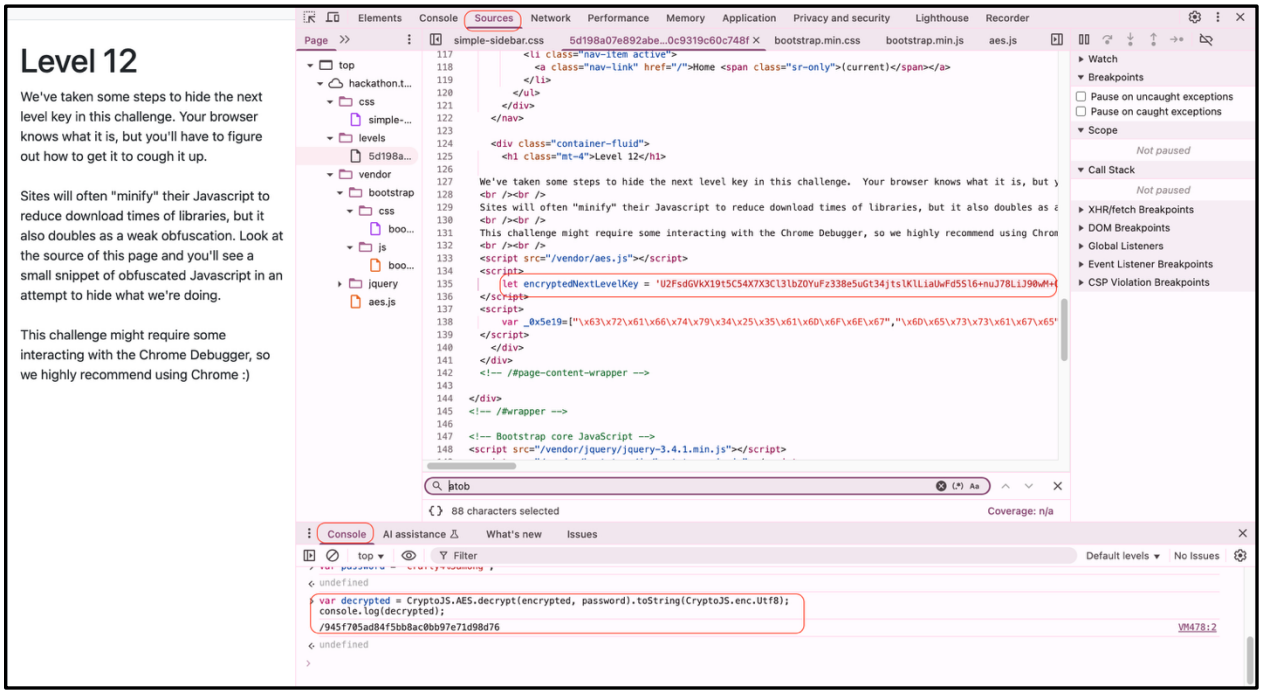


Level 12

We've taken some steps to hide the next level key in this challenge. Your browser knows what it is, but you'll have to figure out how to get it to cough it up.

Sites will often "minify" their Javascript to reduce download times of libraries, but it also doubles as a weak obfuscation. Look at the source of this page and you'll see a small snippet of obfuscated Javascript in an attempt to hide what we're doing.

This challenge might require some interacting with the Chrome Debugger, so we highly recommend using Chrome :)



Goal is to Decrypt the encrypted AES string to reveal the URL for the next level

- Right-click > **Inspect**.
- Go to > Sources Tab > Look for Js files
- Type in the console:

var decrypted = CryptoJS.AES.decrypt(encrypted, password).toString(CryptoJS.enc.Utf8);

console.log(decrypted);

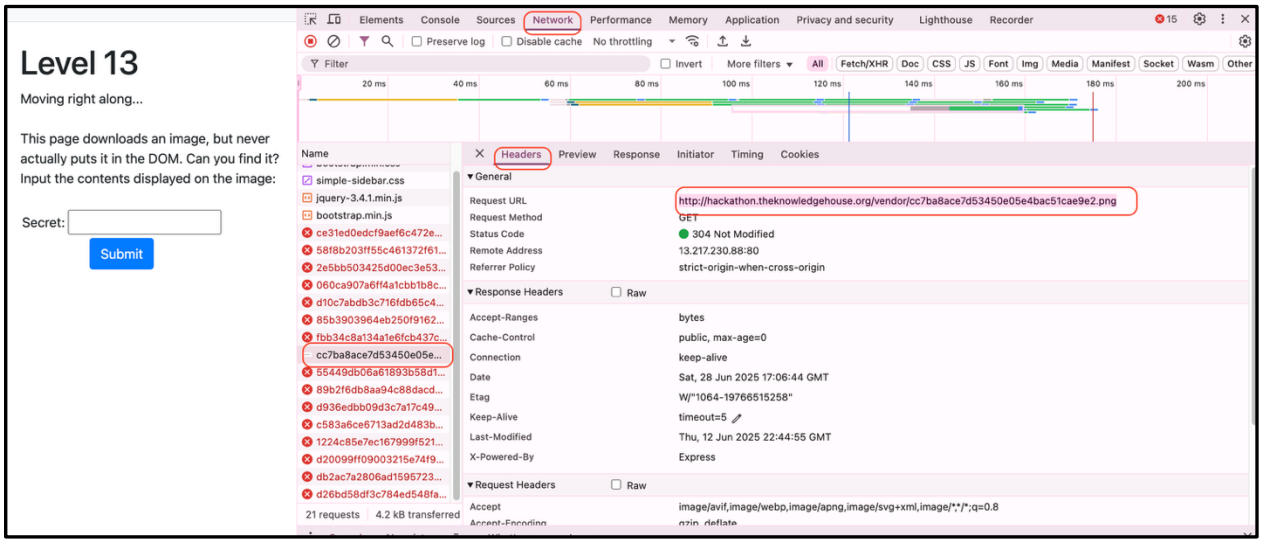
Level 13

Moving right along...

This page downloads an image, but never actually puts it in the DOM. Can you find it? Input the contents displayed on the image:

Secret:

Submit



- **Right-click** > **Inspect**.
- Go to > **Network** tab & reload the page
- Look for an **image file (.png)**
- Right click > Copy URL > paste it into a new browser tab
- The secret is “MEME 149”