HTML Injection

What Is HTML?

HTML stands for Hypertext Markup Language. It is a standard markup language for web pages. A collection of web pages makes a website. HTML elements are represented by <> tags. Where each tag has a different working. A resource to learn HTML: click here

What Is HTML Injection Attack?

HTML Injection is a vulnerability that occurs in web applications that allow users to insert HTML code via a specific parameter or an entry point. HTML Injection is an attack that is similar to Cross-site Scripting (XSS). While in the XSS vulnerability the attacker can inject and execute Javascript code, the HTML injection attack only allows the injection of certain HTML tags. When an application does not properly handle user-supplied data, an attacker can supply valid HTML code, typically via a parameter value, and inject their content into the page. It is generally exploited using social engineering to trick valid users of the application into opening malicious websites or to insert the credentials in a fake login form that will redirect the users to a page that captures cookies or credentials.

Severity

The severity of HTML Injection can be categorized as a P4 bug with a CVSS score of 0.1-3.9 which is Low. In case of an account takeover, it can be categorized as P3.

Logging in {felt this was the most asked question in a while… :D}

As instructed to access labs we have to follow the following path, Search => Hacktify.in > Resources > labs > login

To log in as instructed we need to enter our registered email ID in both the fields email-id and password field, For example:

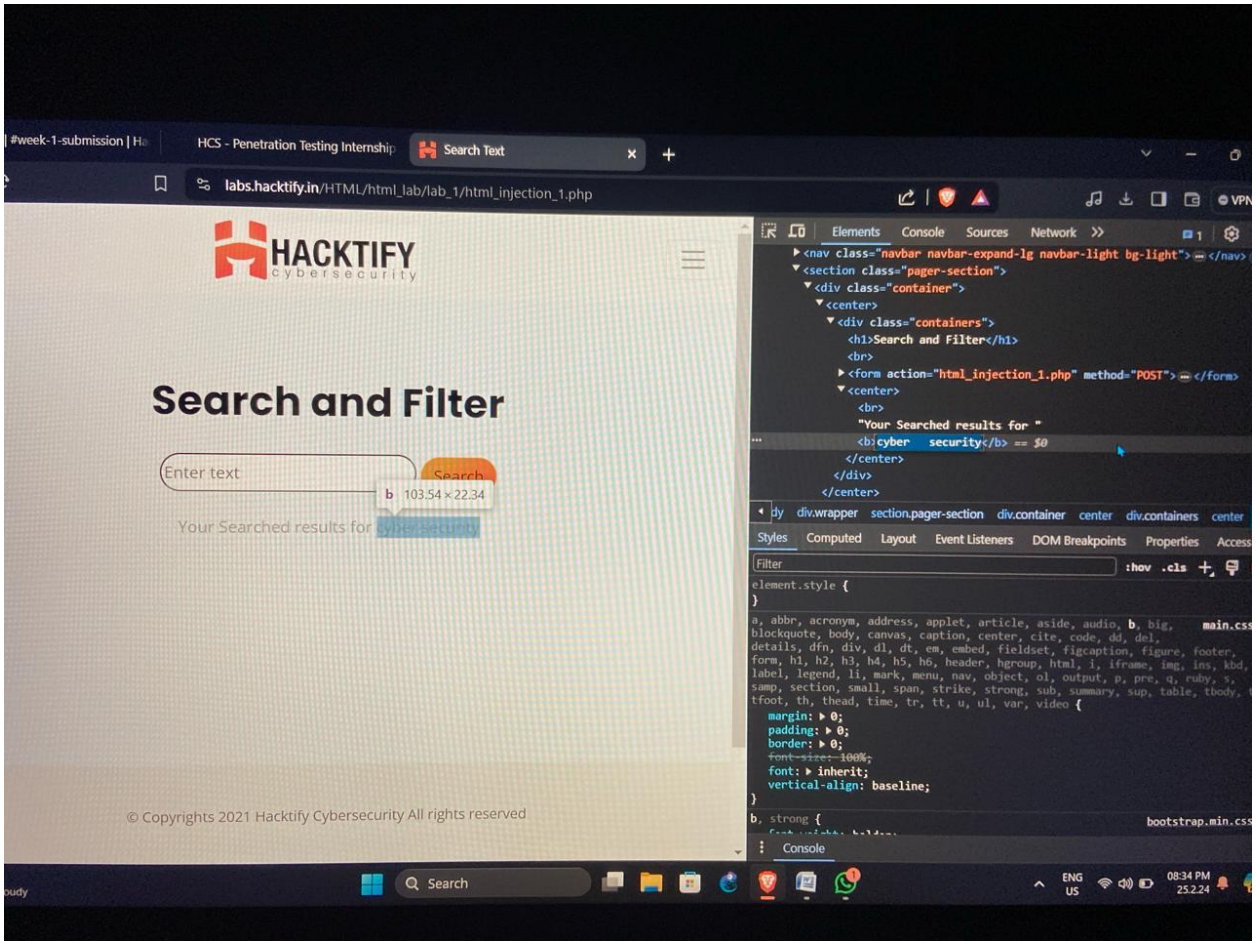
email-id: xyz@mail.com

password: xyz@mail.com

LAB 1: HTML's are easy!

Observation: On accessing the lab we can observe the following UI. Solution: The first thing to test for an HTML injection is we need to check whether or not our input field value is being reflected somewhere on the page/source code or is being stored(we'll talk about this in an upcoming lab). So, let's enter my name into our search bar/box. We can see the value is reflected on the UI and also over the source code. The next step we can do to test for HTML injection is to try a payload "< h1> cyber se urity< /h1>" in the search bar to check whether the user input HTML is being executed or not. On using the payload we can

observe the following response, This confirms that the search parameter is vulnerable to HTML injection.



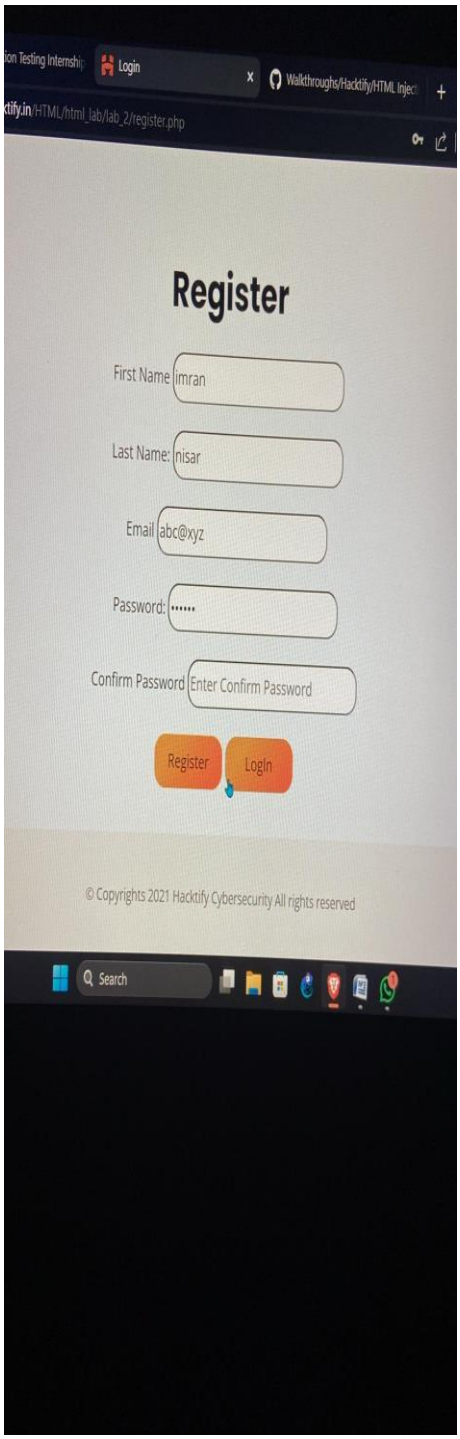
LAB 2: Let me Store them!

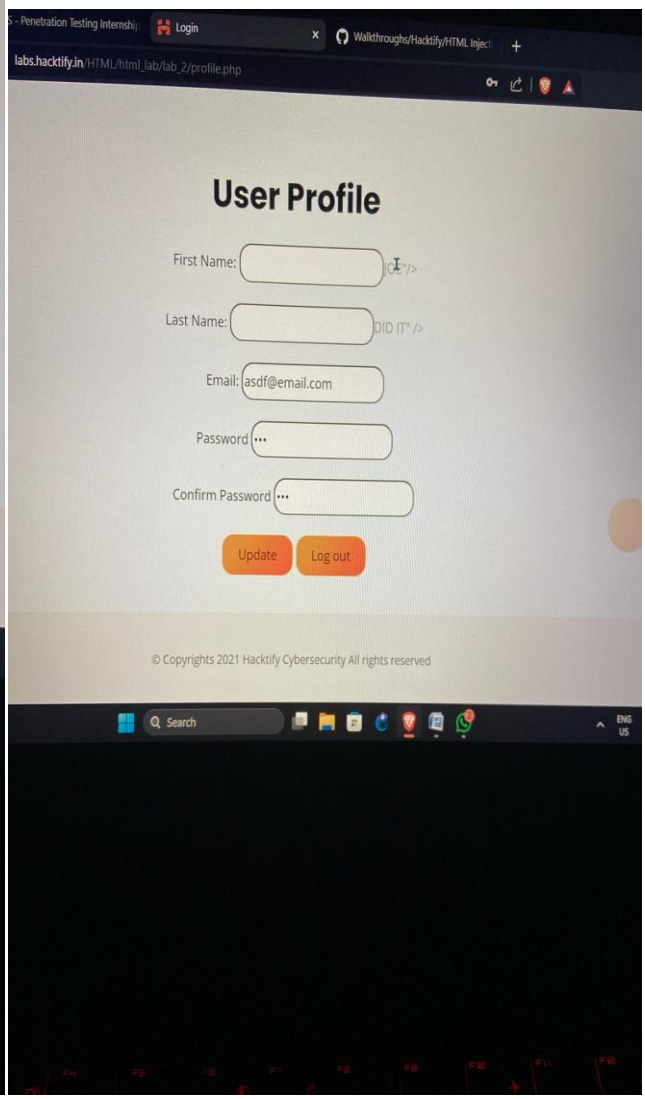
Observation: On accessing the lab we land on a login page where we are required to enter our email ID and password to get access to the user account also provides a 'Register' button which takes us to a page asking us for basic details to create a user account. The pages look like these, Solution:

Firstly let's check by entering a random email ID and password on the landing page to check whether any of the values(mostly email ID would be reflected such as "xyz@mail.com is not a valid email ID"…or something like that) NO LUCK!! :([p.s. shortcuts rarely work.] Let's go to the register page and create a user to log in, Now let's log in using the same email ID and password we used to create our user i.e. email ID - xyz@mail.com and Password - 123 Upon logging in, we are directed to a page resembling the registration form. However, in this instance, the fields are pre-filled with the information we provided when initially creating our dummy user.

Now that we have a clear idea of what should we be doing, let's log off this user and create a new user using the following values for each field, First Name: " "><h1> imran < /h1> " Last Name:""><h1> nisar < /h1> " Email-id:"abcd@xyz" Password: "123" As we log in using this new user (i.e. abc@mail.com) we observe,

The fields with our crafted payloads show the first name and last name of the user in the heading format which concludes that our fields were vulnerable to stored HTML injection.





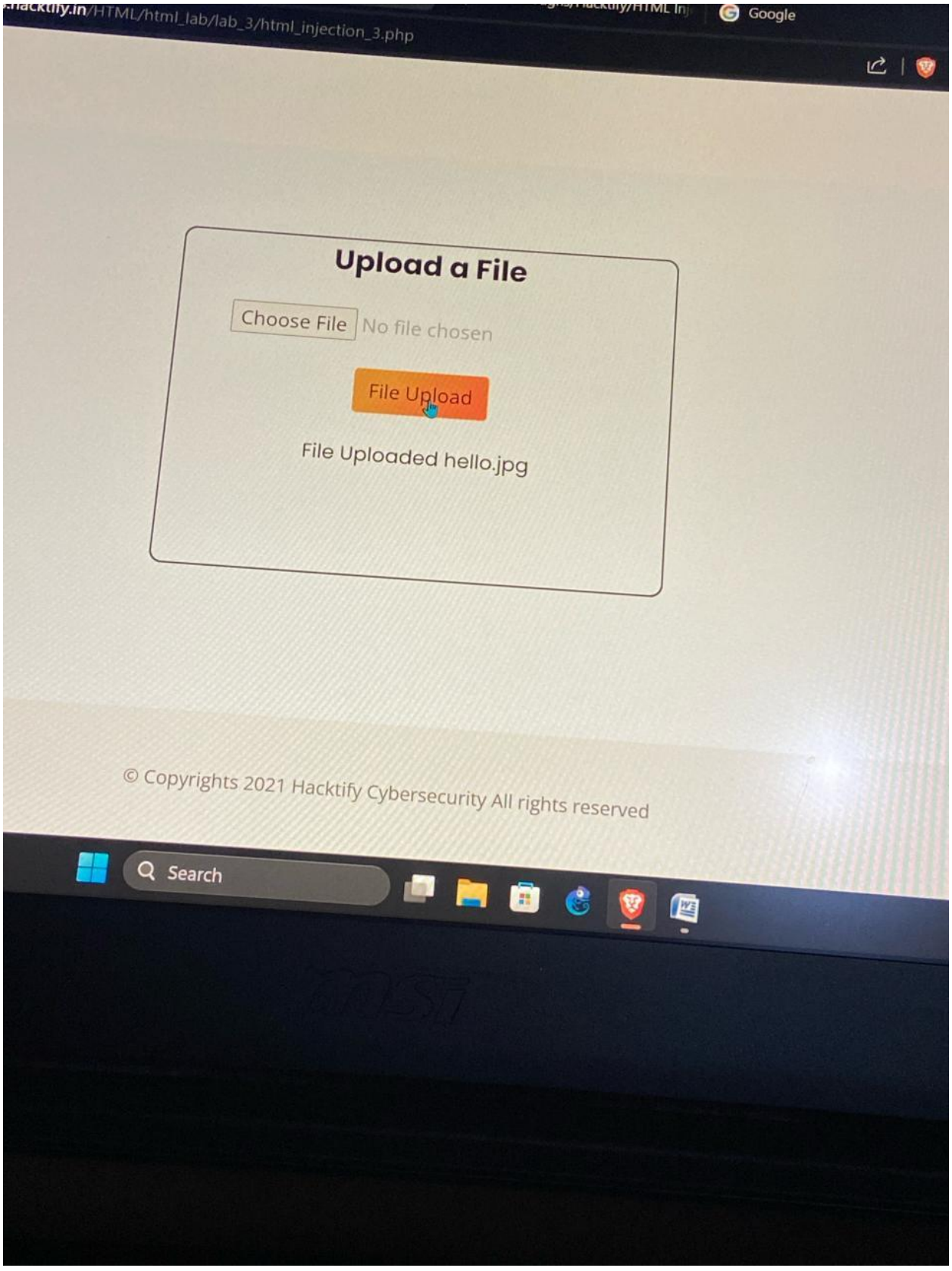
Reflected HTML injection vs Stored HTML injection:

The HTML injection we performed in lab 1 is called reflected HTML injection as the payload we entered in the search box persists only in our current browser session whereas a stored HTML injection would persist even if you log out of your account. (The best way to test it is to make your account in a browser take Firefox and then close the labs there log in to labs in Chrome go to lab 2 and enter the same email ID pass “abcd@xyz” and “123” You can observe that payload persists as stored HTML injection let’s attacker store the payload at server side and hence is more impactful than reflected.)

LAB 3: File Names are also vulnerable!

Observation: On accessing this lab we can notice a file upload feature with a browse file button and an upload button, nothing fancy!! :) Solution: The best way to know how something works or find a flaw is to use the feature and notice the changes, following this thought on uploading a file we can notice this in our lab, Now I don’t know how many of you are familiar with Burp Suite so we’ll keep the complexity of using it to as minimum as possible. To be clear Burp Suite is a Swiss army knife for hackers, and one of its main features is that it acts as a proxy between the source and the destination, and the most common usage is intercepting the requests from the source machine/browser/tab(s). Let’s intercept the request when we hit the upload button and analyze it in Burp Suite, the request looks something like this, We can see the filename in the request now the next thing is to add a payload so one way could be to add a “< h1>” tag before the filename such as, Then forward this crafted request and we get the following response, This observation suggests that the “filename” parameter, utilized to display the name of the uploaded file, lacks proper sanitization. Consequently, this enables the insertion of client-side scripts into the code.

Note: It’s always a game of trial and error, in these walkthroughs we are just using a single tag to test the vulnerability but of course, in real-world scenarios, these general tags are mostly blacklisted so we need to try with multiple HTML tags. Also if we notice here only an opening tag is being utilized and the closing tag is displayed with the file name if you use the following payload “< h1> file_name < /h1>”. So, that’s what we say “Hacking or penetration testing is the game of trial and error.”


.hacktify.in/HTML/html_lab/lab_3/html_injection_3.php
Google
Upload a File
Choose File
No file chosen
File Upload
File Uploaded hello.jpg
© Copyrights 2021 Hacktify Cybersecurity All rights reserved
Search

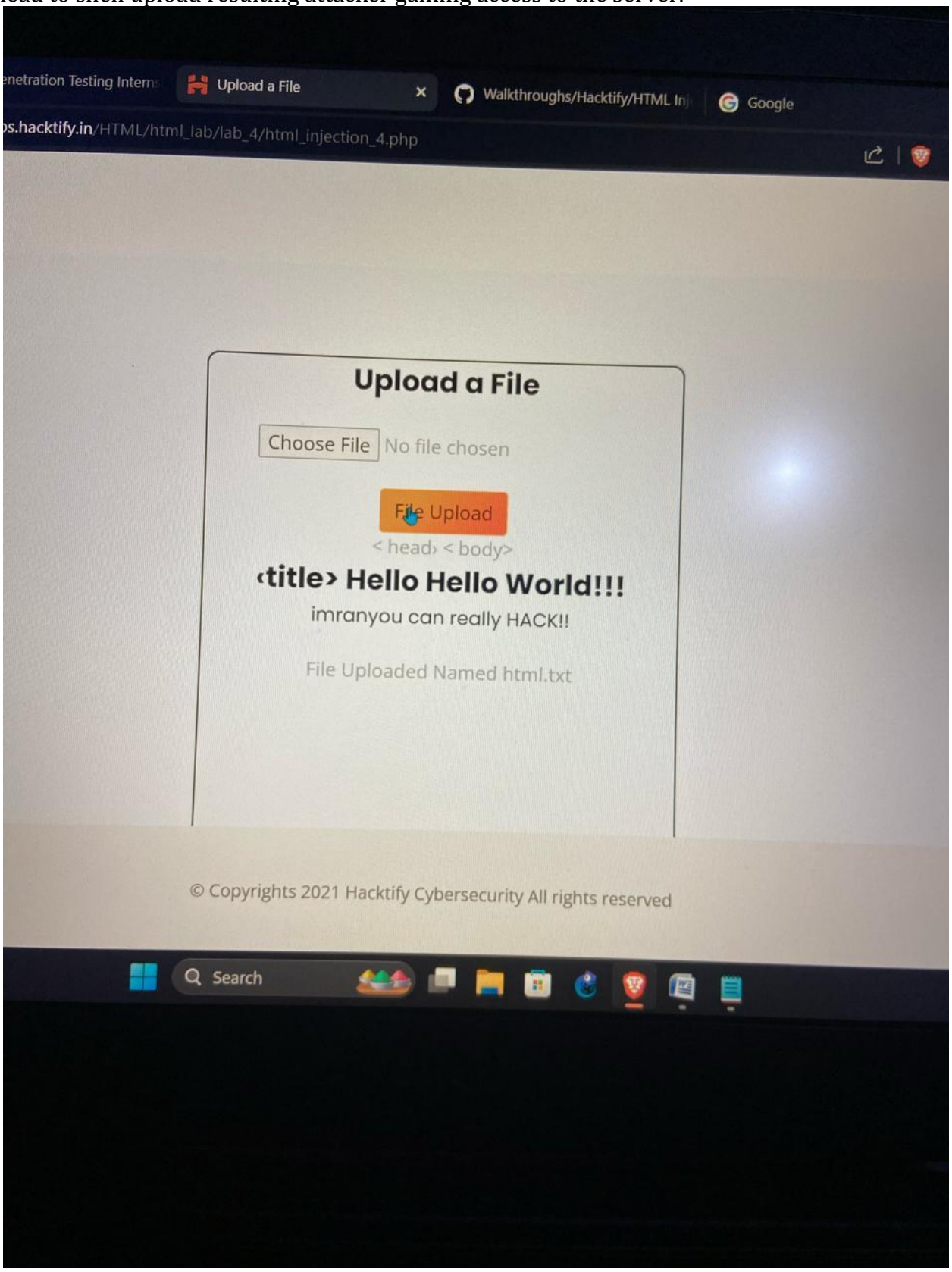
LAB 4: File Content and HTML Injection a perfect pair!

Observation: As soon as we land on the webpage for this lab the frontend looks similar to the previous lab, but the person who styled the div might be still learning the CSS as he left it too large :D. Solution: The best human trait is we learn, we gain experience and we already have the knowledge that our ancestors worked thousands of years and we start from the point they stopped. The best example would be "We don't invent the wheel, that's already done! we study about it and utilize it to make cars."

The reason to state this was, to try to do what we have previously done, so we won't discuss the testing we already did in the previous lab and directly jump to what else can we do with this lab. Reading the name of the lab gave a big hint! So without a delay, we can jump to our code editors or notepad and start typing a basic HTML code and save it into a file, let's say hello.html, and upload it. On uploading this file we might see the following output,

This concludes that our backend does not check the type of file being uploaded and it also lets us parse and execute the file as it is… in most similar cases in the real world this could

lead to shell upload resulting attacker gaining access to the server.

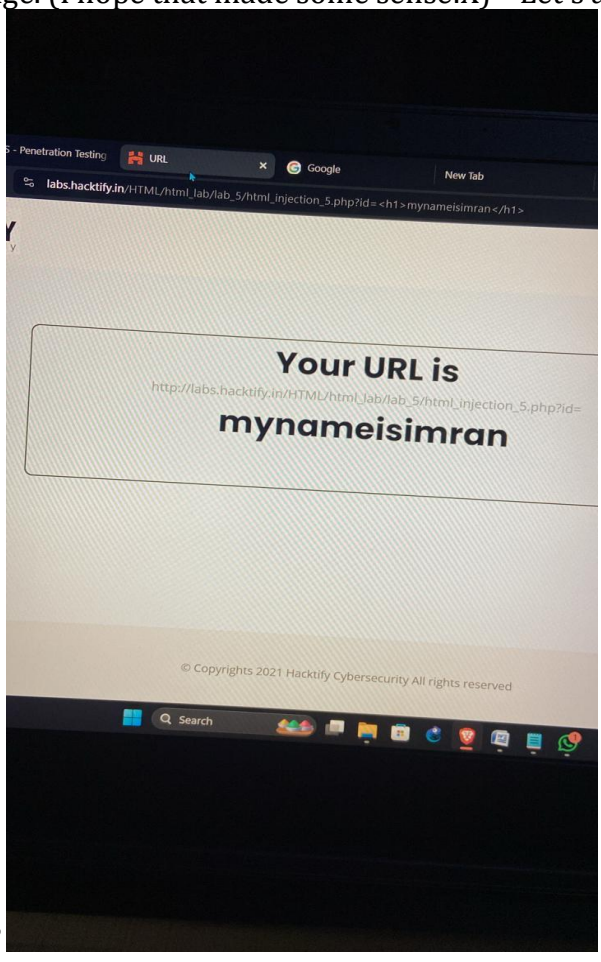


LAB 5: Injecting HTML using URL

We are reaching the end of our lab this will be our second last and by far the toughest lab for me, so stay put, and let's hack it. Observations: On accessing the lab we can observe that there is no injectable parameter like the labs we have done until this point but we can observe that the URL is displayed on the landing page of the lab as follows, Solution: After wrecking my brain and almost trying everything, I had an idea… The question we need to answer is, "Is the URL printed on the frontend hardcoded, or is getting fetched?"

To check this We need to check the source code for the lab.

We all will get something like this, but this does not answer our question, so to get the answer let's play with the the URL for our lab. Note: If the URL marked in the above image is hardcoded it won't get altered but if it's the copy or is getting fetched from the server same as the URL of our lab it will change. (I hope that made some sense:X) Let's add a parameter to our URL, Let's say "?id="



Now on hitting enter to this change, we get, That gives us the most important information that our UI is getting fetched from the server-side script to the URL we search for, Now let's use the following payload and observe what happens, "?id=&lt h1>imran /h1>" On noticing the outcome we can conclude that the queries in the URL are not being filtered/sanitized which is the cause for our URL to be susceptible to HTML injection attacks.