Penetration Testing Report

Name: RU

Program : HCPT

Mar 31, 2025

Table of Content

Introduction

This report document hereby describes the proceedings and results of a Black Box security assessment conducted against the **Hacktify HCPT Internship Program**. The report hereby lists the findings and corresponding best practice mitigation actions and recommendations.

I. Objective

The objective of the assessment was to uncover vulnerabilities in the **Hacktify HCPT Internship Program** and provide a final security assessment report comprising vulnerabilities, remediation strategy and recommendation guidelines to help mitigate the identified vulnerabilities and risks during the activity.

II. Scope

The scope of the penetration testing project by **Hacktify Cyber Security** includes identifying vulnerabilities in the web application's frontend. Testing will focus on detecting HTML injection points will be assessed to identify potential avenues for malicious code insertion and Cross-Site Scripting (XSS) vulnerabilities that could lead to unauthorized actions by users. The boundaries of the project exclude testing of backend systems and network infrastructure. Results will be provided with recommendations for mitigation to enhance the application's security posture.

Application Name	**Lab 1 – HTML Injection** **Lab 2 - Cross-Site Scripting** **Lab 3 - Insecure Direct Object References (IDOR)** **Lab 4 - SQL Injection (SQLi)** **Lab 5 - Cross-Site Request Forgery (CSRF)** **Lab 6 - Cross-Origin Resource Sharing (CORS)** **Lab 7 - Week 4 CTF Challenges**

III. Summary

Outlined is a Black Box Application Security assessment for the **Week 1-4 Hacktify Cyber Security Penetration Testing (HCPT) Internship Labs**. **Total number of Sub-labs: 61**

High	Medium	Low
20	20	21

High - 20 Sub-lab with high difficulty level

Medium -20 Sub-labs with medium difficulty level

Low - 21 Sub-labs with low difficulty level

Week 1

1. HTML Injection

1.1. HTML's are easy!

Reference	Risk Rating
Sub-lab-1: HTML's are easy!	Low

Tools Used

Browser "*View Page Sources*" is used to find the vulnerability.

Vulnerability Description

User Input in search Field: The form submits user input to html_injection_1.php via POST. If html_injection_1.php does not properly sanitize or escape user input, an attacker could inject malicious HTML or JavaScript.

How It Was Discovered

1. Injecting HTML elements such as <h1 style="color:red;">Hacked!</h1> could alter the page structure.
2. Injecting a Malicious Link Such as: Click here for free money! which could be used for phishing.

Vulnerable URLs

https://labs.hacktify.in/HTML/html_lab/lab_1/html_injection_1.php

Consequences of not Fixing the Issue

If the vulnerability is not patched, an attacker can be able to write his own code to get the cookie information of the victim user, this HTML Injection attack leads to XSS attack.

Suggested Countermeasures

2. Implement strict input validation and sanitize user-supplied data. This will only allow **alphanumeric** characters and reject special HTML tags

```
if (!preg_match("/^[a-zA-Z0-9 ]*$/", $search)) {
   die("Invalid input detected!");}
```

2. Use contextual output encoding to prevent script execution.
3. Deploy Content Security Policy (CSP) to restrict script sources. Such as:
header("Content-Security-Policy: default-src 'self'; script-src 'self'");
4. Educate developers on secure coding practices.
5. Regularly audit and test for vulnerabilities.

References

https://owasp.org/www-community/Injection_Information
https://portswigger.net/web-security/cross-site-scripting/html-injection

https://en.wikipedia.org/wiki/HTML_injection

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab. Injecting HTML elements such as <h1 style="color:red;">Hacked!</h1> could alter the page structure.

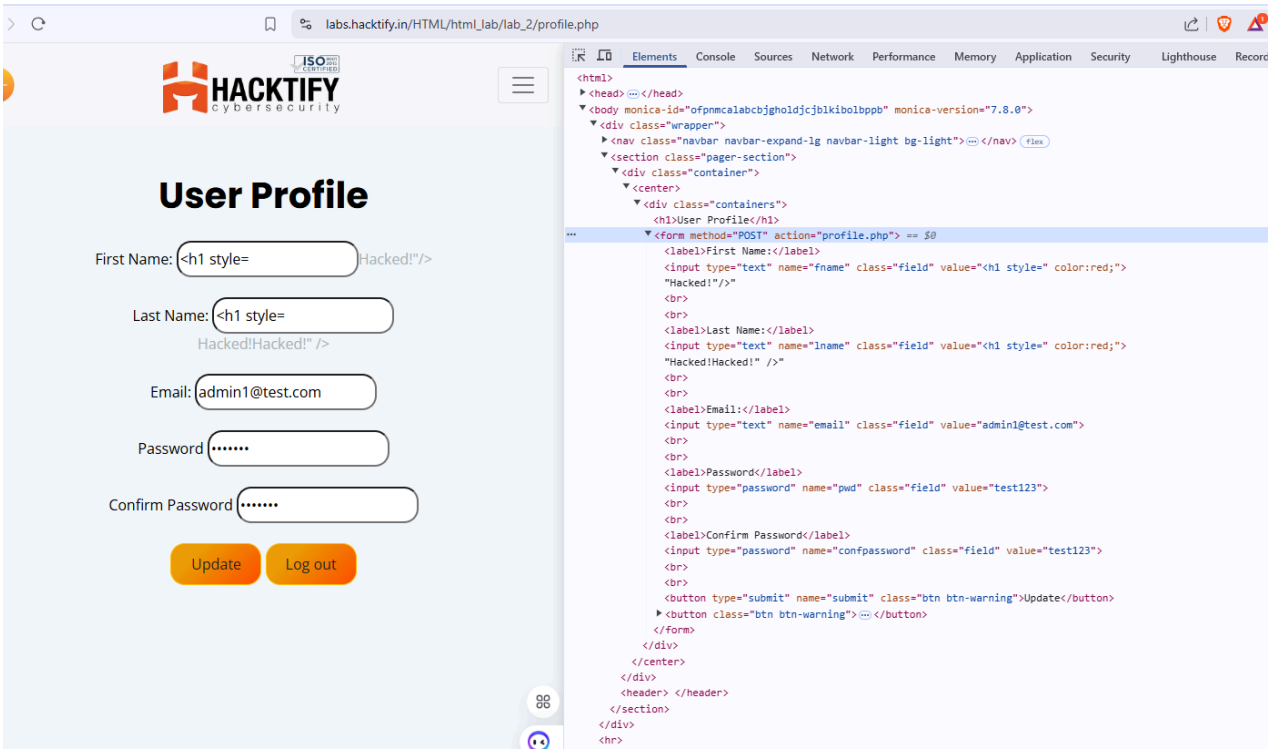
1.2: Let me Store them!

Reference	Risk Rating
Sub-lab- 2: Let me Store them!	Low
Tools Used	
Browser Developer Tools (Inspect Element), Burp Suite, Manual Input Testing	
Vulnerability Description	
Stored HTML Injection vulnerability allows attackers to inject and store malicious HTML content in user profile fields, which later gets executed when the profile page is viewed. This could lead to UI defacement, phishing, or stored XSS attacks	
How It Was Discovered	
The vulnerability was identified by modifying the First Name field with the payload Hacked! , which was successfully stored and rendered on the profile page without sanitization.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_2/profile.php https://labs.hacktify.in/HTML/html_lab/lab_2/html_injection_2.php	
Consequences of not Fixing the Issue	
- Attackers can manipulate profile data to inject harmful scripts. - Possibility of stored XSS, allowing session hijacking and cookie theft.	

- Defacement of the website. - Phishing attempts that trick users into entering credentials on malicious pages. - Attackers gaining persistent access to the application.
Suggested Countermeasures
- **Sanitize Input Before Storage:** Use htmlspecialchars($_POST['fname'], ENT_QUOTES, 'UTF-8');. - **Sanitize Output Before Display:** Use htmlentities($fname, ENT_QUOTES, 'UTF-8');. - **Enforce Content Security Policy (CSP):** Restrict inline scripts to prevent XSS execution. - **Implement Input Validation:** Reject inputs containing <script>, <h1>, or other HTML tags. - **Use Parameterized Queries:** Prevent malicious input execution via database interactions.
References
- OWASP HTML Injection Guide: OWASP - Stored XSS Prevention: OWASP XSS Prevention Cheat Sheet

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab.



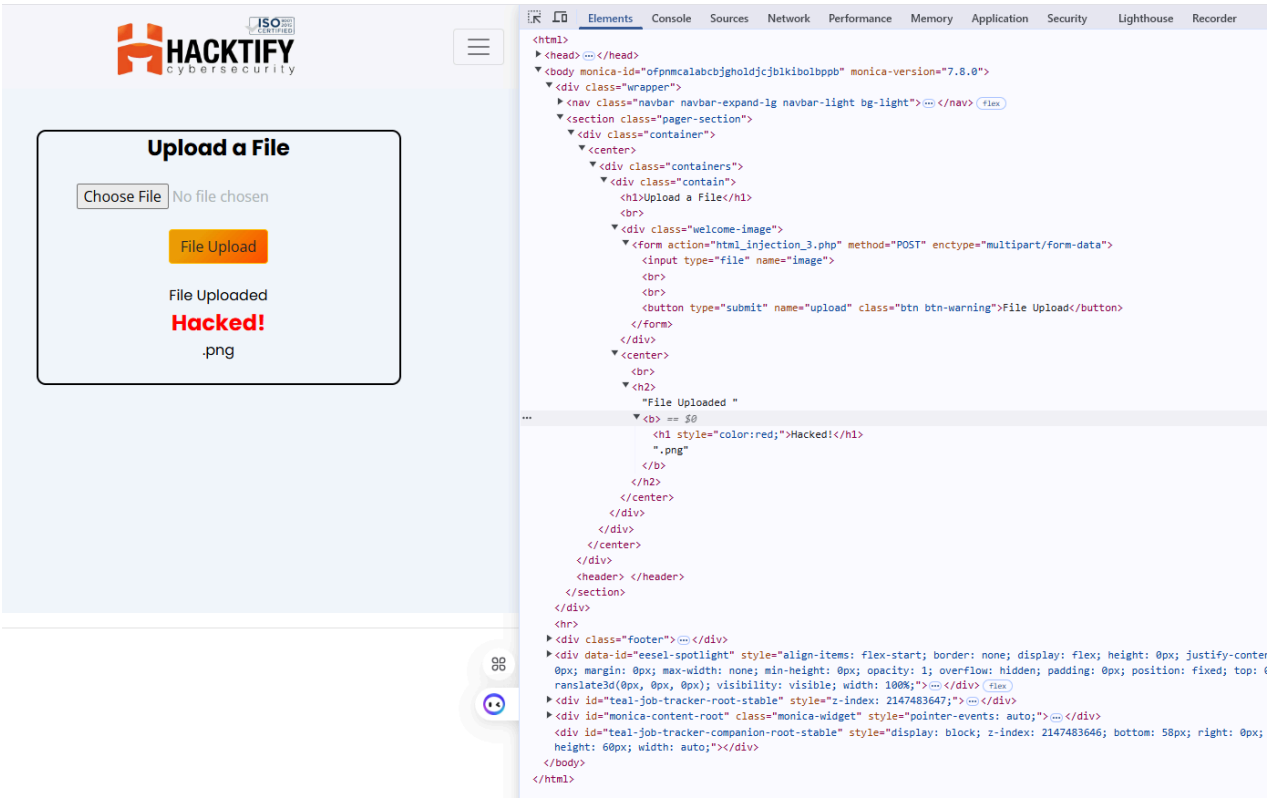
1.3: File Names are also vulnerable!

Reference	Risk Rating
Sub-lab- 3:File Names are also vulnerable!	Low

Tools Used
Burp Suite, Browser Developer Tools (Inspect Element), Manual Payload Testing, OWASP ZAP
Vulnerability Description
The application does not properly sanitize user-provided file names before storing or displaying them. If a malicious file name contains HTML or JavaScript, it gets executed when displayed on a web page. This can lead to Stored HTML Injection or Stored XSS, affecting all users who view the infected file name.
How It Was Discovered
The vulnerability was tested by uploading a file with a modified filename containing an HTML tag (Hacked! .png) and a JavaScript payload (<script>alert('XSS!')</script>.png). After uploading, the file name was reflected on the page without sanitization, proving that arbitrary HTML/JS could be executed.
Vulnerable URLs
https://labs.hacktify.in/HTML/html_lab/lab_3/html_injection_3.php
Consequences of not Fixing the Issue
This vulnerability could lead to Stored XSS, allowing attackers to execute scripts in the browser of an administrator or user. If exploited, it could lead to: - Stored XSS attacks: Malicious scripts get stored and executed for every user viewing the filename. - Session hijacking: Attackers can steal cookies via JavaScript (document.cookie). - Website defacement: Attackers can inject rogue HTML that alters page appearance. - Phishing attacks: Fake login forms can be injected via <iframe> tags. - Malicious redirects: Users could be redirected to an attacker-controlled site.
Suggested Countermeasures
- Sanitize file names before storing: Remove or encode special characters using preg_replace("/[^a-zA-Z0-9.\-_]/", "", $filename);. - Escape output before displaying file names: Use htmlspecialchars($filename, ENT_QUOTES, 'UTF-8');. - Restrict allowed file extensions: Allow only png, jpg, pdf, etc. - Store files outside web-accessible directories: Serve them via an authenticated script. - Set strict Content Security Policy (CSP): Prevent inline script execution.
References
- OWASP XSS Prevention Cheat Sheet: OWASP XSS Prevention - OWASP Unrestricted File Upload Guide: OWASP File Upload

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



1.4: File Content and HTML Injection a perfect pair!

Reference	Risk Rating
Sub-lab-4: File Content and HTML Injection a perfect pair!	Medium

Tools Used

Browser Developer Tools (Inspect Element), Burp Suite, Text editor

Vulnerability Description

The application allows users to upload files without properly sanitizing the file names. This oversight permits attackers to inject malicious HTML or JavaScript code through crafted file names, leading to potential HTML injection or cross-site scripting (XSS) attacks.

How It Was Discovered

By uploading a file with a name containing HTML tags or JavaScript code and observing the application's response, it was noted that the malicious code was executed when the file name was rendered in the browser.

Vulnerable URLs

https://labs.hacktify.in/HTML/html_lab/lab_4/html_injection_4.php

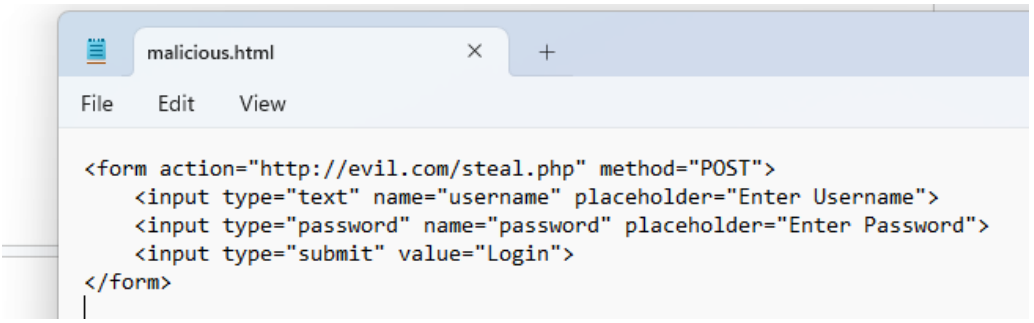
Consequences of not Fixing the Issue

If unaddressed, attackers can execute arbitrary HTML or JavaScript in the context of the application's

users. This can lead to session hijacking, defacement, redirection to malicious sites, or unauthorized actions on behalf of users.
Suggested Countermeasures
Implement strict input validation to ensure file names do not contain any HTML or script tags. Sanitize file names by removing or encoding special characters. Restrict allowed file types and enforce safe file naming conventions. Store uploaded files with unique identifiers rather than user-provided names to prevent malicious code execution.
References
HTML Injection in filename leads to XSS HTML Injection | Pentest Vulnerability Wiki File uploads | Web Security Academy

Proof of Concept

1. Create a file named malicious.html.
2. Upload the file using the application's file upload feature.
3. Navigate to the page where the uploaded file is listed or displayed.
4. Observe that the JavaScript code in the file name executes, triggering an alert box with the message 'XSS'.



1.5: Injecting HTML using URL

Reference	Risk Rating
Sub-lab-2.5: Injecting HTML using URL	Medium

Tools Used

Browser Developer Tools (Inspect Element), Burp Suite, Manual Input Testing

Vulnerability Description

The application is vulnerable to HTML injection through URL parameters. This occurs when user-supplied data in the URL is not properly sanitized, allowing attackers to inject malicious HTML code

How It Was Discovered

By analyzing the URL structure and testing with various inputs, it was observed that the application reflects user-supplied data without proper sanitization, leading to HTML injection.

Vulnerable URLs

http://labs.hacktify.in/HTML/html_lab/lab_5/html_injection_5.php

Consequences of not Fixing the Issue

- Unauthorized content display
- Phishing attacks
- Potential for further exploitation

Suggested Countermeasures

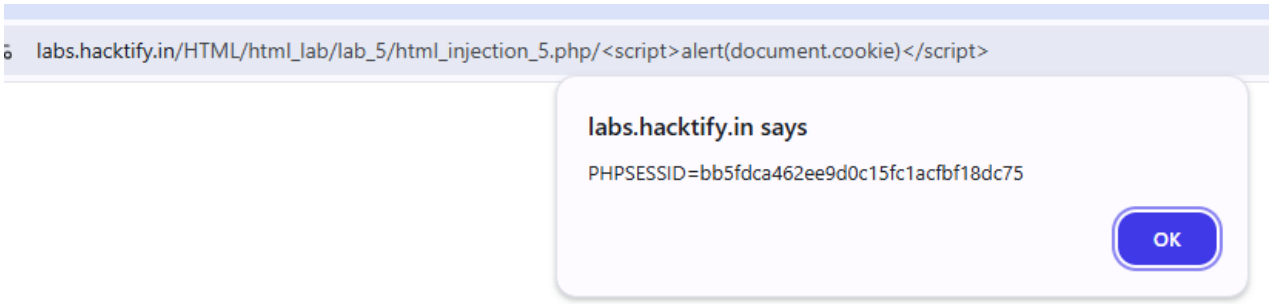
- Implement input validation and sanitization to ensure that user inputs do not contain malicious HTML code.
- Use a Content Security Policy (CSP) to restrict the sources from which content can be loaded.
- Regularly conduct security audits to identify and remediate such vulnerabilities.

References

What Is HTML Injection | Types, Risks & Mitigation Techniques
HTML Injection - Invicti

Proof of Concept

By appending a parameter to the URL, such as ?name=<h1>Injected</h1>, and <script>alert(document.cookie)</script>, the application displays the injected HTML content and user session cookie respectively, thus demonstrating the vulnerability.



1.6: Encode IT!

Reference	Risk Rating
Sub-lab-2.6: Encode IT!	High

Tools Used

Browser Developer Tools (Inspect Element), Burp Suite or OWASP ZAP

Vulnerability Description

The application fails to properly encode user input in the search functionality, leading to HTML Injection vulnerabilities. Attackers can inject malicious HTML or JavaScript code, which gets executed in the context of the user's browser.

How It Was Discovered

During testing, it was observed that entering encoded HTML tags in the search field resulted in the execution of the injected code, indicating improper input handling.

Vulnerable URLs

https://labs.hacktify.in/HTML/html_lab/lab_6/html_injection_6.php

Consequences of not Fixing the Issue

Exploitation can lead to session hijacking, defacement, or redirection to malicious sites, compromising user data and trust.

Suggested Countermeasures

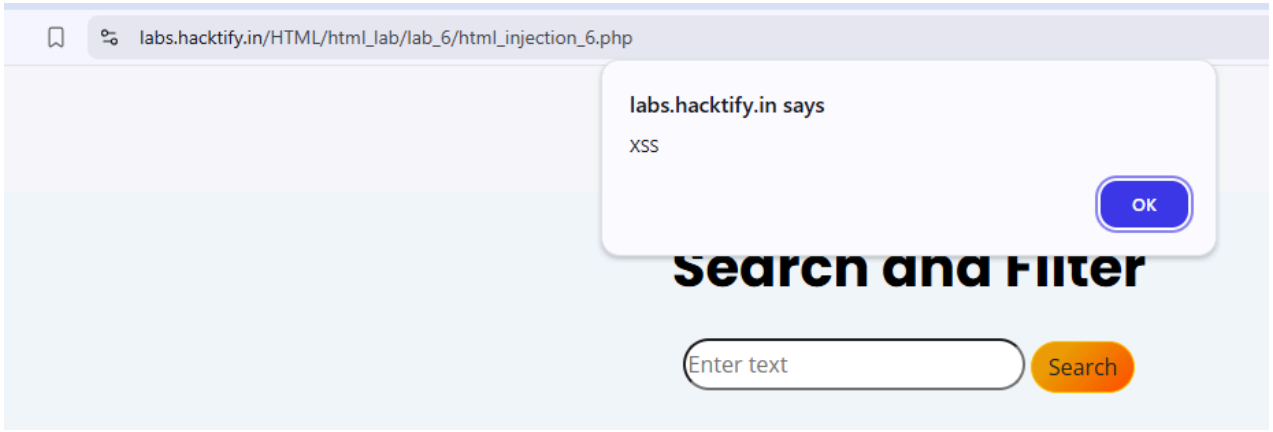
Implement proper input validation and output encoding. Utilize security libraries or frameworks that automatically handle encoding. Regularly update and patch the application to address known vulnerabilities.

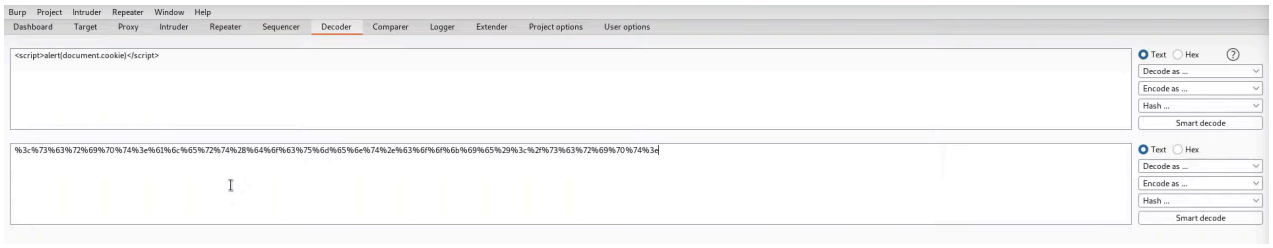
References

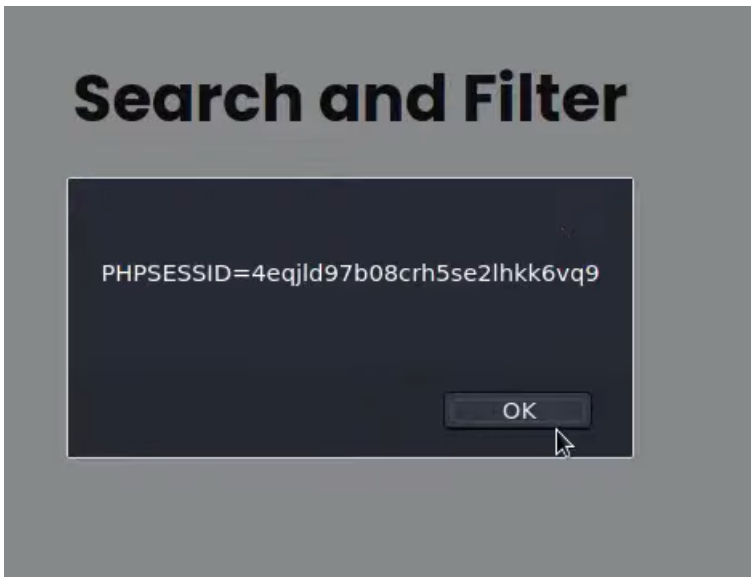
HTML Injection - Penetration Testing Lab
What Is HTML Injection | Types, Risks & Mitigation Techniques

Proof of Concept

Injecting the encoded payloads %3Cscript%3Ealert%28%27XSS%27%29%3B%3C%2Fscript%3E and %3Cscript%3Ealert(document.cookie)%3C/script%3E into the search field results in the execution of <script>alert('XSS');</script> and alert(document.cookie) script, demonstrating the vulnerability.







2. Cross-Site Scripting Labs

2.1: Let's Do IT!

Reference	Risk Rating
Sub-lab-2.1: Let's Do IT!	Low

Tools Used

Browser Developer Tools (Inspect Element)

Vulnerability Description

The application fails to sanitize user input in the email field, allowing attackers to inject and execute malicious scripts. This vulnerability arises from improper handling of user-supplied data, leading attackers to execute arbitrary scripts in the context of the user's browser, leading to potential data theft, session hijacking, and other malicious activities.

How It Was Discovered

By analyzing the form's input handling and testing with a simple script injection, it was observed that the application executed the injected script, indicating an XSS vulnerability. The form accepts user input through the email field and submits it to lab_1.php using the GET method. If lab_1.php processes this input without proper validation or sanitization, it becomes vulnerable to XSS attacks.

Vulnerable URLs

https://labs.hacktify.in/HTML/xss_lab/lab_1/lab_1.php

Consequences of not Fixing the Issue

Data theft
Session hijacking
Defacement of web pages
Phishing attacks
Loss of user trust

Suggested Countermeasures

- Input Validation: Ensure that all user inputs are validated against expected formats.
- Output Encoding: Encode user inputs before rendering them in the browser to prevent script execution.
- Content Security Policy (CSP): Implement CSP headers to restrict the execution of unauthorized scripts.
- Use Security Libraries: Utilize security libraries or frameworks that automatically handle input sanitization and output encoding.

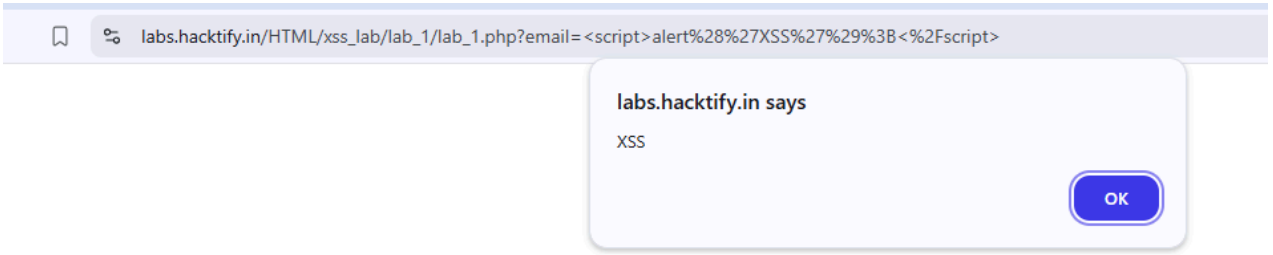
References

PortSwigger: Cross-site scripting (XSS)

OWASP: Cross-site Scripting (XSS)

Proof of Concept

By submitting the payload <script>alert('XSS');</script> through the email input field, the application processes and reflects this input without sanitization, leading to the execution of the script. This confirms the presence of an XSS vulnerability.



2.2: Balancing is Important in Life!

Reference	Risk Rating
Sub-lab-2: Balancing is Important in Life!	Low
Tools Used	
Web Browser Developer Tools (Inspect Element)	
Vulnerability Description	
The application is vulnerable to Reflected Cross-Site Scripting (XSS). This occurs when user-supplied input is included in the application's response without proper validation or encoding, allowing attackers to inject malicious scripts.	
How It Was Discovered	
I entered the payload <script>alert('XSS')</script> in the email subscription field. The response displayed: "You'll receive email on <script>alert('XSS')</script>." The script tags were not executed, indicating partial sanitization. I then inputted the payload ">Hacked!<script>alert('Hacked:)</script>. The application executed the script, displaying an alert with the message "Hacked:)." This suggests that the application fails to properly handle input containing both quotation marks and script tags, leading to successful script execution.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_2/lab_2.php	
Consequences of not Fixing the Issue	
• Session Hijacking: Attackers can steal user session cookies, leading to unauthorized account access.	

- Phishing: Malicious scripts can redirect users to fraudulent sites, compromising sensitive information.
- Data Theft: Attackers can capture user inputs, such as login credentials or personal details.
- Reputation Damage: Exploitation of this vulnerability can erode user trust and harm the organization's reputation.

Suggested Countermeasures

To prevent such Cross-Site Scripting (XSS) vulnerabilities, it's essential to:

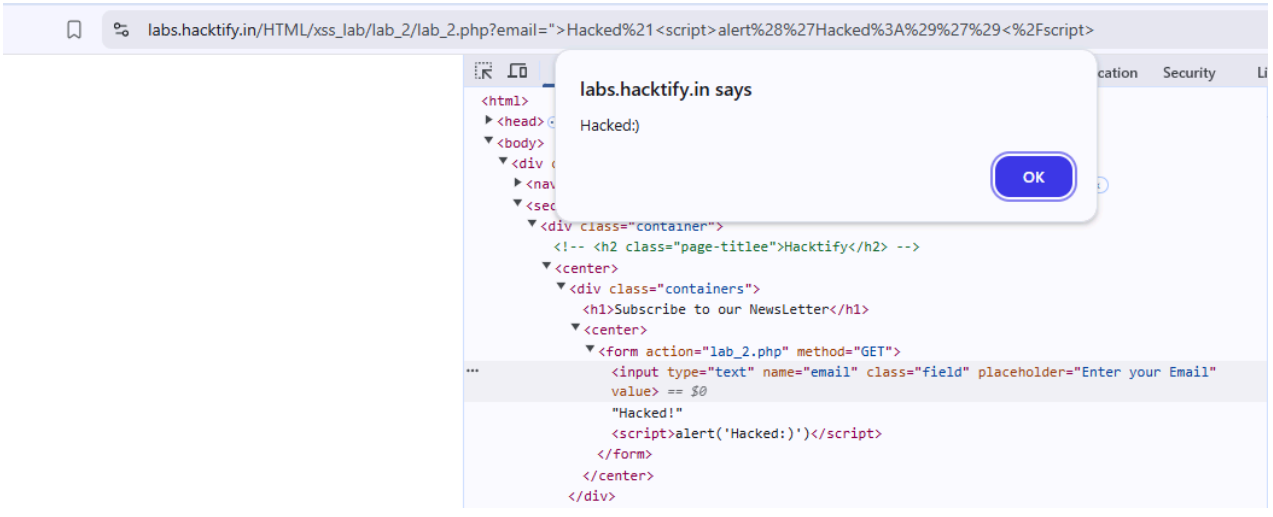
- Sanitize User Inputs: Ensure that all user inputs are properly sanitized and encoded before being incorporated into HTML content. This includes escaping characters like <, >, &, and " to their respective HTML entities.
- Use Security Libraries: Utilize security libraries or frameworks that automatically handle input sanitization and encoding.
- Implement Content Security Policy (CSP): A CSP can help mitigate the impact of XSS attacks by restricting the sources from which scripts can be loaded.

References

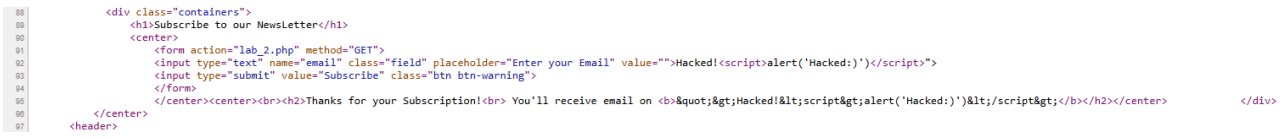
OWASP Cross-Site Scripting (XSS) Overview: owasp.org
Reflected Cross-Site Scripting (RXSS) Report Template: the-red.team

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



`



2.3: XSS is everywhere!

Reference	Risk Rating
Sub-lab-3: XSS is everywhere!	Low

Tools Used

Web Browser Developer Tools (Inspect Element)

Vulnerability Description

The application is vulnerable to Reflected Cross-Site Scripting (XSS). It fails to properly validate and sanitize user input in the 'email' parameter of the subscription form. This allows attackers to inject malicious scripts, which are then executed in the context of the user's browser.

How It Was Discovered

During testing, the following steps were performed: I Entered a standard string (test) in the email input field and submitted the form. The application responded with "Please Enter Valid Email address," indicating some level of input validation.
I then entered a string containing a script tag (<script>alert('Hacked!')</script>@test.com) in the email input field and submitted the form. The script executed, displaying an alert with the message "Hacked!". This demonstrated that the application does not adequately sanitize input containing script tags, leading to the execution of injected scripts.

Vulnerable URLs

https://labs.hacktify.in/HTML/xss_lab/lab_3/lab_3.php

Consequences of not Fixing the Issue

If this vulnerability remains unaddressed, attackers can:

- Execute arbitrary JavaScript in the context of users' browsers.
- Steal sensitive information such as session cookies, leading to account hijacking.
- Deface the website or redirect users to malicious sites.
- Potentially spread malware to users.

Suggested Countermeasures

- **Input Validation**: Implement strict input validation on the server side to ensure that only properly formatted email addresses are accepted.
- **Output Encoding**: Encode user inputs before displaying them in the browser to prevent the execution of malicious scripts.
- **Content Security Policy (CSP)**: Implement a strong CSP to restrict the execution of unauthorized scripts.
- **Regular Security Testing**: Conduct regular security assessments, including automated and manual testing, to identify and remediate XSS vulnerabilities.

References

OWASP Cross-Site Scripting (XSS)

OWASP XSS Prevention Cheat Sheet

Proof of Concept

This confirms the presence of a Reflected XSS vulnerability in the application.

2.4: Alternatives are must!

Reference	Risk Rating
Sub-lab-4: Alternatives are must!	Medium

Tools Used

Web Browser, Developer Tools (Inspect Element, Console), Burp Suite

Vulnerability Description

The application allows arbitrary JavaScript execution i.e it does not validate or sanitize user input in the email field before displaying it back on the webpage. This allows attackers to inject malicious JavaScript payloads, leading to Reflected XSS attacks.

How It Was Discovered

1. Entered a normal string (test) → Accepted without validation, showing lack of input filtering.
2. Injected <script>alert('Hacked!')</script>@test.com → Displayed as raw text, showing partial sanitization.
3. Injected ><script>alert('Hacked!')</script>@test.com → Script executed successfully, confirming XSS.
4. Injected ><script>prompt(1)</script>@test.com → A prompt box appeared, proving JavaScript execution.

Vulnerable URLs

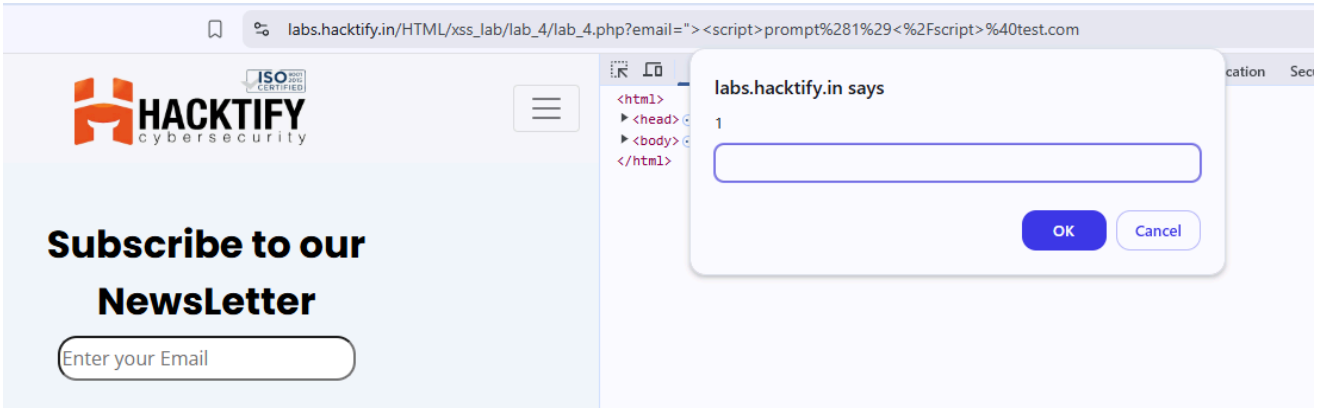
https://labs.hacktify.in/HTML/xss_lab/lab_4/lab_4.php

Consequences of not Fixing the Issue

This can lead to **session hijacking, phishing, Malware Injection and website defacement**.
Suggested Countermeasures
1. Input Validation: Ensure the email field only accepts valid email addresses using regex: Example: ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ 2. Output Encoding: Convert special characters (<, >, " etc.) to HTML entities: Example (PHP): htmlspecialchars($_GET['email'], ENT_QUOTES, 'UTF-8'); 3. Use HTTP Headers: Implement Content Security Policy (CSP) to prevent inline scripts: Example: Content-Security-Policy: default-src 'self'; script-src 'none' 4. Sanitize Input Data: Use built-in functions to sanitize input before processing. Example (PHP): filter_var($_GET['email'], FILTER_SANITIZE_EMAIL);
References
- OWASP XSS Prevention Cheat Sheet: OWASP Guide - PortSwigger Web Security Academy: XSS Labs

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



2.5: Developer hates scripts!

Reference	Risk Rating
Sub-lab-5: Developer hates scripts!	High
Tools Used	
Web Browser Developer Tools (Inspect Element)	
Vulnerability Description	
The application is vulnerable to Reflected Cross-Site Scripting (XSS). This occurs when user input is improperly validated and then reflected back in the application's response, allowing attackers to inject malicious scripts.	
How It Was Discovered	
During testing, the following steps were taken: 1. Entered a simple string like "test" in the email input field, which resulted in a prompt stating, "Please Enter Valid Email address." The Input ">hello<IMG SRC=javascript:alert(1)>@test.com confirmed the vulnerability, as the scripts executed upon submission.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_5/lab_5.php	
Consequences of not Fixing the Issue	
If unaddressed, attackers can execute arbitrary scripts in the context of the user's session. This can lead to unauthorized actions, data theft, session hijacking, and a loss of user trust.	
Suggested Countermeasures	
To mitigate this vulnerability: 1. Input Validation: Implement strict input validation to ensure only valid email formats are accepted. 2. Output Encoding: Encode user inputs before reflecting them back in the response to prevent script execution. 3. Content Security Policy (CSP): Deploy a robust CSP to restrict the execution of untrusted scripts.	
References	
1. PortSwigger: Reflected XSS 2. OWASP: XSS Prevention Cheat Sheet	

Proof of Concept

A screenshot demonstrating the execution of the injected script using the payload ">hello<IMG SRC=javascript:alert(1)>@test.com"

2.6: Change the Variation!

Reference	Risk Rating
Sub-lab-6: Change the Variation!	High

Tools Used

Web Browser Developer Tools (Inspect Element), Manual Testing

Vulnerability Description

The successful execution of the second payload suggests that the application is vulnerable to XSS attacks within HTML attributes. Specifically, by injecting a " character, it's possible to break out of an attribute context and introduce a new element with an event handler that executes JavaScript.

How It Was Discovered

Initially, the payload <script>alert(document.cookie)</script> did not trigger a popup, indicating that the application might be filtering or sanitizing certain inputs. However, by using the payload "><img src="x" onerror="alert('XSS')">, a popup was successfully triggered.

Vulnerable URLs

https://labs.hacktify.in/HTML/xss_lab/lab_6/lab_6.php

Consequences of not Fixing the Issue

Risk of XSS attacks stemming from improper handling of user inputs within HTML attributes

Suggested Countermeasures

To mitigate this vulnerability:

1. **Input Validation**: Implement strict validation to ensure that inputs conform to expected formats. For email fields, use regular expressions to validate the structure of the email address.
2. **Output Encoding**: Encode user inputs before rendering them in the HTML response. This ensures that special characters are treated as literals rather than executable code.
3. **Content Security Policy (CSP)**: Define a CSP that restricts the execution of unauthorized scripts. For instance, disallow inline scripts and only permit scripts from trusted sources.
4. **Attribute Context Handling**: When inserting user input into attribute values, ensure that quotes and other special characters are properly encoded to prevent breaking out of the attribute context.

References

https://portswigger.net/

Proof of Concept

A popup displaying "XSS" should appear, confirming the execution of the injected script "><img src="x" onerror="alert('XSS')">


labs.hacktify.in/HTML/xss_lab/lab_6/lab_6.php?email="><img+src%3D"x"+onerror%3D"alert%28%27XSS%27%29">
labs.hacktify.in says
XSS
OK
Subscribe to our
NewsLetter
Enter your Email
Subscribe
Thanks for your Subscription!
You'll receive email on "><img src="x"
onerror="alert('XSS')">

2.7: Encoding is the key?

Reference	Risk Rating
Sub-lab-7: Encoding is the key?	Medium

Tools Used

Web browser, developer tools, Burp Suite, URL Encode and Decode tools

Vulnerability Description

The application fails to properly sanitize or encode user input in the email field before reflecting it back on the page. This permits attackers to inject encoded Cross-Site Scripting (XSS) payloads that, when decoded by the browser, execute arbitrary JavaScript. In this lab, encoding the payload was critical to bypass any naive filtering mechanisms.

How It Was Discovered

An unencoded payload "><script>alert('Hacked!')</script> was submitted, but no popup appeared—suggesting the input was being filtered or not interpreted as active code.
I then submitted an encoded version of the payload, using URL encoding: %22%3E%3Cscript%3Ealert%28%27Hacked%21%27%29%3C%2Fscript%3E. The encoded payload successfully broke out of the attribute context, and the injected script executed, displaying a popup with the alert "Hacked!".

Vulnerable URLs

https://labs.hacktify.in/HTML/xss_lab/lab_7/lab_7.php

Consequences of not Fixing the Issue

- **Session Hijacking:** Malicious scripts could steal user cookies.
- **Data Theft & Phishing:** Attackers can inject code to redirect users to fraudulent sites or harvest sensitive data.
- **Website Defacement:** Injected scripts could alter the appearance of the page.
- **Loss of User Trust:** Repeated attacks erode confidence in the application's security.

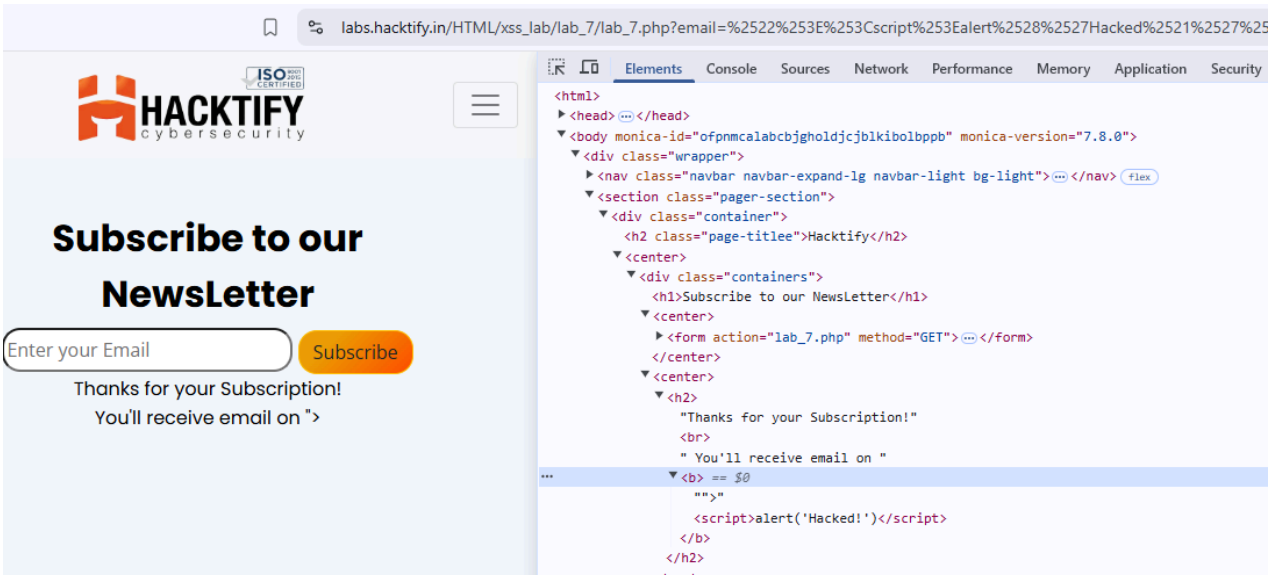
Suggested Countermeasures

1. **Input Validation:** Strictly validate the email field using a robust regex pattern to allow only valid email formats.
Example (PHP):
php
if (!preg_match("/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/", $_GET['email'])) {
die("Invalid email address!");
}
2. **Output Encoding:** Ensure that any user-provided data is properly encoded before rendering it to the browser using functions such as htmlspecialchars().
Example (PHP):
php
echo htmlspecialchars($_GET['email'], ENT_QUOTES, 'UTF-8');
3. **Content Security Policy (CSP):** Implement CSP headers to restrict the execution of inline scripts.
Example:
php
header("Content-Security-Policy: default-src 'self'; script-src 'self'");

4. **Sanitization Libraries:** Use libraries that automatically sanitize and validate inputs
References
- OWASP XSS Prevention Cheat Sheet - [PortSwigger: Cross-site scripting (XSS)]

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



2.8: XSS with File Upload (file name)

Reference	Risk Rating
Sub-lab-XSS with File Upload (file name)	Low

Tools Used

Web browser, developer tools, Burp Suite

Vulnerability Description

The file upload functionality does not properly validate or sanitize the file name before displaying it on the web page. This allows an attacker to rename a file with an XSS payload (using, for example, an <img> tag with an onerror event) so that when the file name is rendered, the malicious code is executed in the user's browser. This is a classic case of stored or reflected XSS, depending on whether the file name is stored and later displayed.

How It Was Discovered

The image file uploaded was renamed with the malicious payload and uploaded via the form. When the application later displayed the file name, the payload `"><img src="x" onerror="alert('XSS')">` executed, producing a popup alert.

Vulnerable URLs

https://labs.hacktify.in/HTML/xss_lab/lab_7/lab_8.php

Consequences of not Fixing the Issue

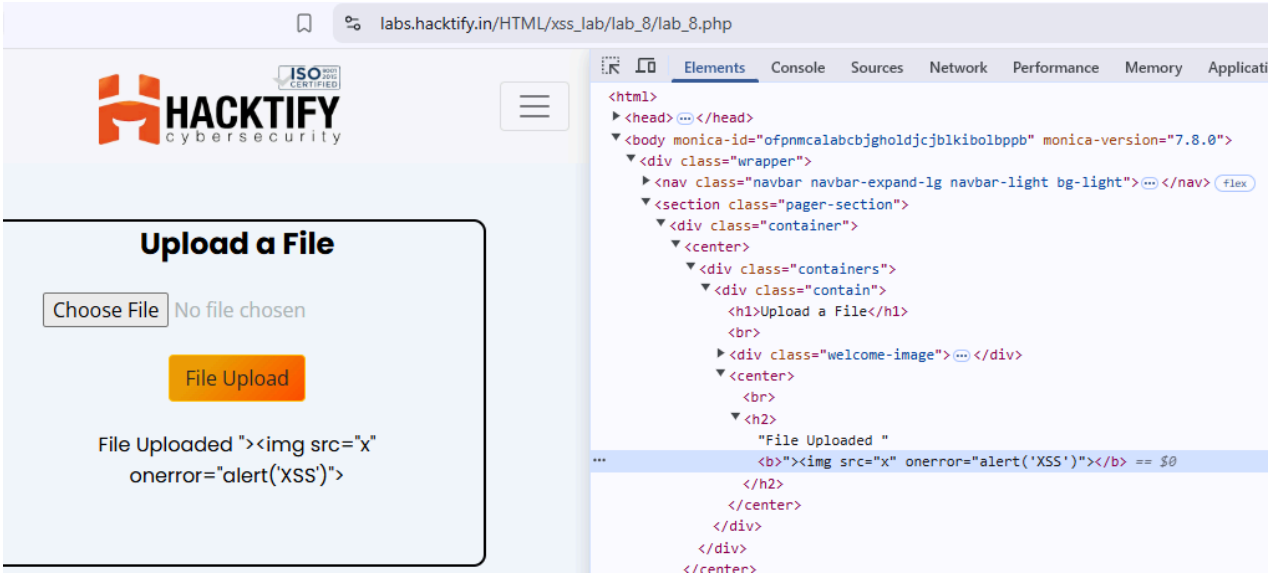
- **Arbitrary JavaScript Execution:** Attackers can execute scripts in the context of a user's browser.
- **Session Hijacking:** Malicious scripts could steal cookies or session tokens.
- **Website Defacement:** Attackers might change page content or redirect users.
- **Phishing & Malware Delivery:** Injected code may be used to redirect users to fraudulent sites or deliver malware.
- **Loss of User Trust & Reputation Damage:** Users' security may be compromised, reducing confidence in the site.

Suggested Countermeasures

1. **Input Validation:** Restrict file names to a safe set of characters (e.g., alphanumeric characters, dots, hyphens, and underscores).
Example (PHP):
php
$filename = preg_replace("/[^a-zA-Z0-9.\-_]/", "", $_FILES["image"]["name"]);
2. **Output Encoding:** When displaying the file name, ensure that any special characters are HTML-encoded.
Example (PHP):
php
echo htmlspecialchars($filename, ENT_QUOTES, 'UTF-8');
3. **Use Whitelisting:** Enforce a whitelist of acceptable file name patterns to reject malicious inputs.
4. **Store Files with Safe Identifiers:** Instead of using user-supplied names, generate unique file names on the server side and map them to the original name in a secure manner.
5. **Implement a Content Security Policy (CSP):** Use CSP headers to restrict script execution, reducing

the impact if an XSS does occur. Example: php header("Content-Security-Policy: default-src 'self'; script-src 'self'");
References
- OWASP XSS Prevention Cheat Sheet - PortSwigger Web Security Academy: File Upload - OWASP: Unrestricted File Upload

Proof of Concept



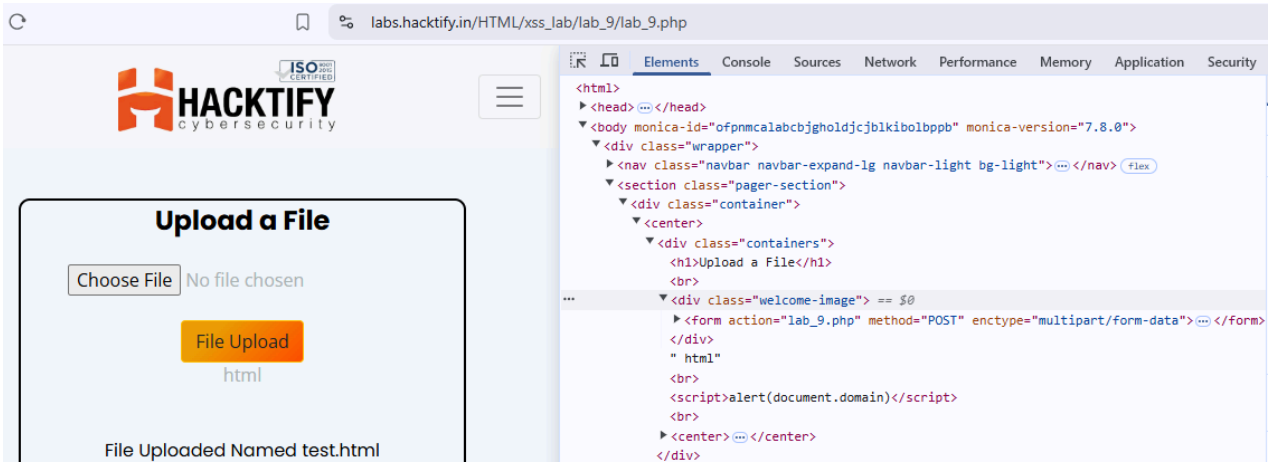
2.9: XSS with File Upload (File Content)

Reference	**Risk Rating**
Sub-lab-XSS with File Upload (File Content)	**Medium**
Tools Used	
Web browser, developer tools, Burp Suite	
Vulnerability Description	
The file upload functionality does not properly validate or sanitize the file name before displaying it on the web page. This allows an attacker to rename a file with an XSS payload so that when the file name is rendered, the malicious code is executed in the user's browser. This is a classic case of stored or reflected XSS, depending on whether the file name is stored and later displayed.	
How It Was Discovered	
The file uploaded was renamed with the malicious payload html <script>alert(document.domain)</script> and uploaded it via the form. When the application later displayed the file name, the payload executed, producing a popup alert.	

Vulnerable URLs
https://labs.hacktify.in/HTML/xss_lab/lab_7/lab_9.php
Consequences of not Fixing the Issue
- Arbitrary JavaScript Execution: Attackers can execute scripts in the context of users' browsers. - Session Hijacking: Malicious code can steal session cookies or other sensitive information. - Website Defacement: Injected scripts can alter the appearance or functionality of the site. - Phishing and Malware Delivery: Attackers may redirect users to malicious sites or load additional harmful scripts. - Loss of User Trust: Persistent vulnerabilities can damage the site's reputaion
Suggested Countermeasures
1. Input Validation: Validate and restrict file content by allowing only safe file types (e.g., actual image MIME types) and reject files with embedded HTML or JavaScript. 2. MIME Type Verification: Use server-side functions (like PHP's finfo_file) to verify the file's MIME type matches the expected type before processing the file. 3. Output Encoding: If file content or file names are to be displayed, ensure that all user-provided data is HTML-encoded using functions like htmlspecialchars(). 4. Store Files Securely: Save uploaded files with server-generated filenames outside the web-accessible directory, and serve them via a secure script that does not directly render the file content as HTML. 5. Content Security Policy (CSP): Implement CSP headers to reduce the risk of injected scripts executing.
References
- OWASP XSS Prevention Cheat Sheet - PortSwigger: File Upload Vulnerabilities - OWASP Guide to Unrestricted File Upload

Proof of Concept

1. Create a file (e.g., exploit.html) containing the payload: html
 <script>alert(document.domain)</script>
2. Navigate to the vulnerable upload page at https://labs.hacktify.in/HTML/xss_lab/lab_9/lab_9.php.
3. Upload the file via the provided form.
4. After upload, view the page where the file content or name is rendered.
5. Observe that an alert popup appears displaying the current domain, confirming that the injected script executed successfully.



2.10: Stored Everywhere!

Reference	Risk Rating
Sub-lab- Stored Everywhere!	High

Tools Used

Web browser, developer tools, Burp Suite

Vulnerability Description

Stored XSS with a severity of P2 and a CVSS score between 7 and 8.9. This vulnerability can be used to steal cookies (even of admin users), hijack sessions, and perform large-scale data theft because the payload is stored and executed every time the affected page is loaded.
The application does not sanitize or encode user input on registration fields (e.g., first name, last name, email). Malicious scripts injected into these fields are stored on the server and later rendered in pages (such as the user profile) without proper escaping. This results in stored Cross-Site Scripting (XSS), which affects all users viewing the page, including administrators.

How It Was Discovered

1. **Registration Testing:** The tester registered an account using payloads in multiple fields:

• **First Name:** ">fname<IMG SRC=javascript:alert(Hacked)>
• **Last Name:** ">lname<IMG SRC=javascript:alert(Again)>
• **Email:** "><script>alert(document.cookie)</script>@gmail.com
2. After submitting the registration form, the application redirected to the login page. The tester then logged in using the registered credentials.
3. **Observation:** Upon logging in, a popup appeared—triggered by the stored XSS payload in the profile page—confirming that the injected scripts were stored and later executed.

Vulnerable URLs

https://labs.hacktify.in/HTML/xss_lab/lab_7/lab_10.php
https://labs.hacktify.in/HTML/xss_lab/lab_10/register.php
https://labs.hacktify.in/HTML/xss_lab/lab_10/profile.php

Consequences of not Fixing the Issue

- Arbitrary JavaScript Execution: Malicious scripts execute in every user's browser who views the affected page.
- Session Hijacking: Attackers can steal session cookies (including those of administrators), leading to unauthorized access.
- Phishing and Data Theft: Injected code can redirect users to phishing pages or capture sensitive information.
- Reputation Damage: Persistent XSS vulnerabilities undermine user trust and harm the organization's credibility.

Suggested Countermeasures

1. **Input Validation:** Ensure that registration fields accept only valid, expected data. For email fields, use a regex such as:
php
if (!preg_match("/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/", $_POST['email'])) {
die("Invalid email address!");
}
2. **Output Encoding:** Use functions like htmlspecialchars() to encode output when displaying user data on pages:
php
echo htmlspecialchars($firstname, ENT_QUOTES, 'UTF-8');
3. **Sanitization:** Use built-in functions (e.g., PHP's filter_var()) to sanitize input before storing it.
4. **Content Security Policy (CSP):** Deploy CSP headers to restrict inline script execution. For example:
php
header("Content-Security-Policy: default-src 'self'; script-src 'self'");
5. **Regular Security Testing:** Conduct periodic security audits and code reviews to detect and fix XSS vulnerabilities.

References

- OWASP XSS Prevention Cheat Sheet
- OWASP Cross-Site Scripting (XSS) Overview
- PortSwigger Web Security Academy: XSS

Proof of Concept

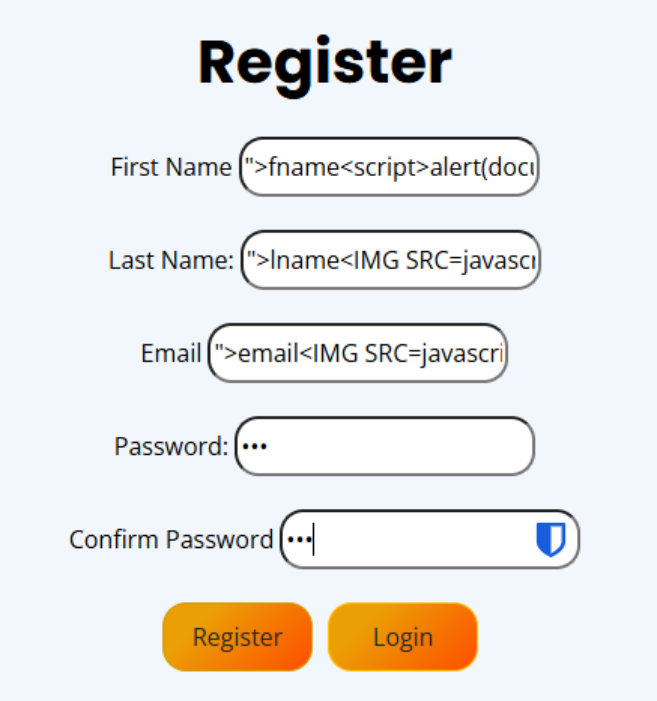
1. Navigate to the registration page and fill in the fields with the following payloads:
• **First Name:** ">fname<IMG SRC=javascript:alert(Hacked)>
• **Last Name:** ">lname<IMG SRC=javascript:alert(Again)>

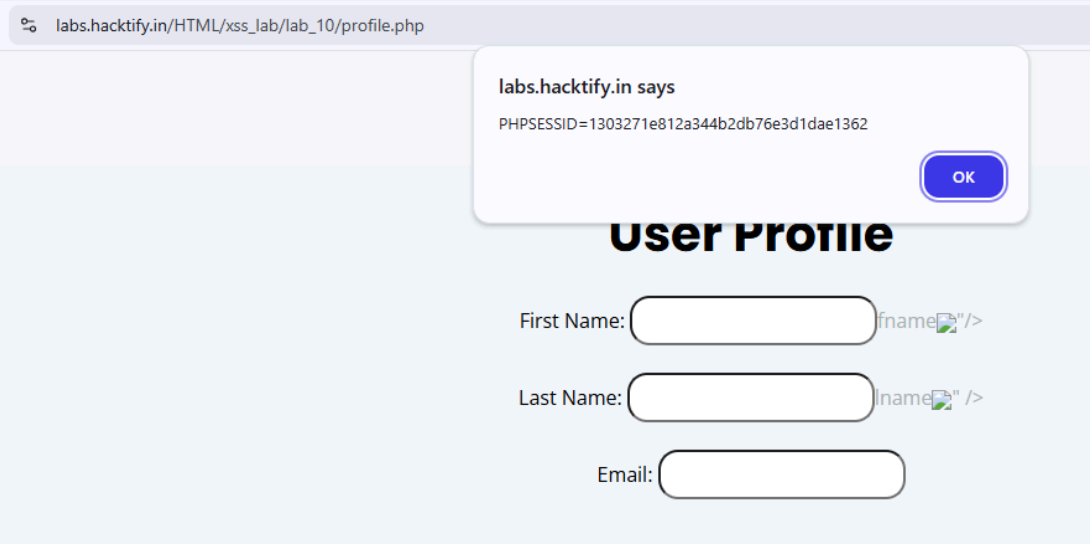
• **Email:** "><script>alert(document.cookie)</script>@gmail.com

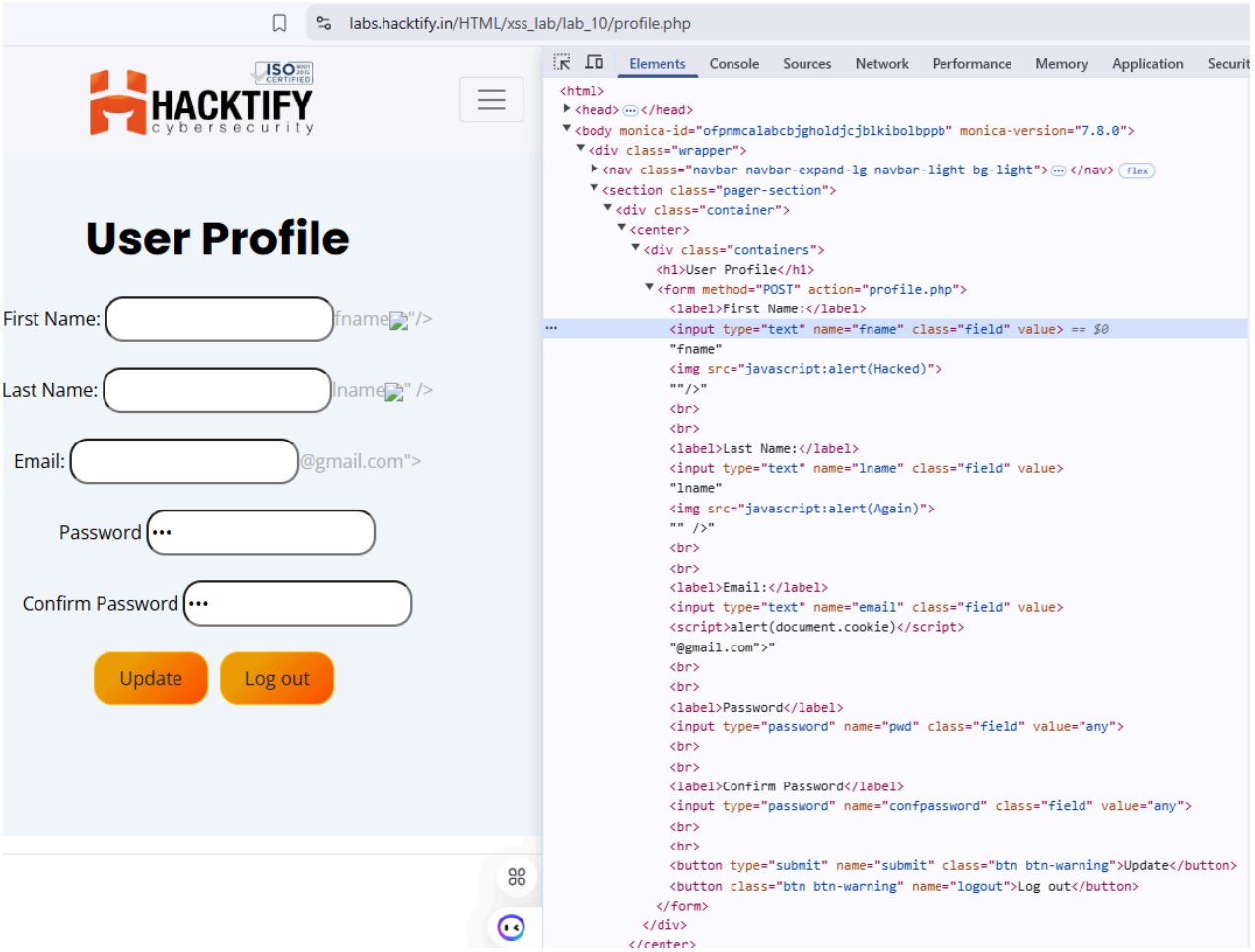
2. Complete the registration (set any password) and click **Register**.

3. After being redirected to the login page, log in using the registered credentials.

4. Upon loading the user profile page, an alert box pops up (triggered by the stored payload), confirming that the XSS payload executed and the vulnerability exists.







2.11: DOM's are love!

Reference	Risk Rating
Sub-lab-DOM's are love!	High

Tools Used

Web browser, developer tools, Burp Suite

Vulnerability Description

The analysis confirms that the web application is vulnerable to DOM-based XSS attacks through multiple entry points. DOM-based Cross-Site Scripting (DOM XSS) occurs when client-side scripts of a web application process user input without proper validation or encoding, leading to the execution of malicious scripts. In this lab, the provided dom.js script contains several potential vulnerabilities due to improper handling of user inputs.

How It Was Discovered

Similarly, by manipulating the redir and coin parameters, an attacker can execute arbitrary scripts due to the improper handling of these parameters in the dom.js script. By addressing these vulnerabilities through proper input validation, sanitization, and avoiding unsafe JavaScript functions, the application

can mitigate the risk of DOM-based XSS attacks.

Vulnerable URLs

https://labs.hacktify.in/HTML/xss_lab/lab_7/lab_11.php
https://labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?coin=doge
https://labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?coin=btc
https://labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?coin=eth
https://labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?coin=doge

Consequences of not Fixing the Issue

If these vulnerabilities are not addressed, attackers can execute arbitrary JavaScript in the context of the user's session. This can lead to:

- Theft of sensitive information (e.g., cookies, session tokens)
- Unauthorized actions on behalf of the user
- Redirection to malicious websites
- Defacement of the website

Suggested Countermeasures

Input Validation and Sanitization:

- Sanitize all user inputs, especially those from the URL parameters, to remove or encode potentially malicious content.
- Use libraries like DOMPurify to sanitize HTML content.

Avoid Dangerous Functions:

- Refrain from using functions like eval() and properties like innerHTML that can execute or render raw HTML.
- Use safer alternatives such as textContent or innerText for inserting text into the DOM.

Implement Content Security Policy (CSP):

- Define a strict CSP to restrict the execution of unauthorized scripts and resources.

URL Validation:

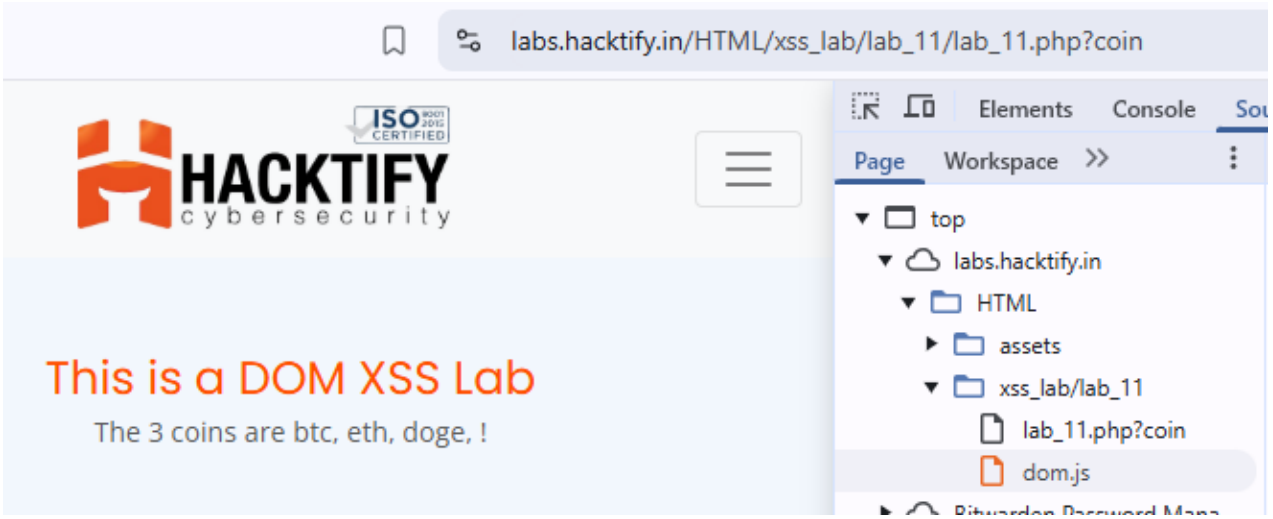
- When redirecting users based on URL parameters, ensure the destination URL is within the allowed domain list to prevent open redirects

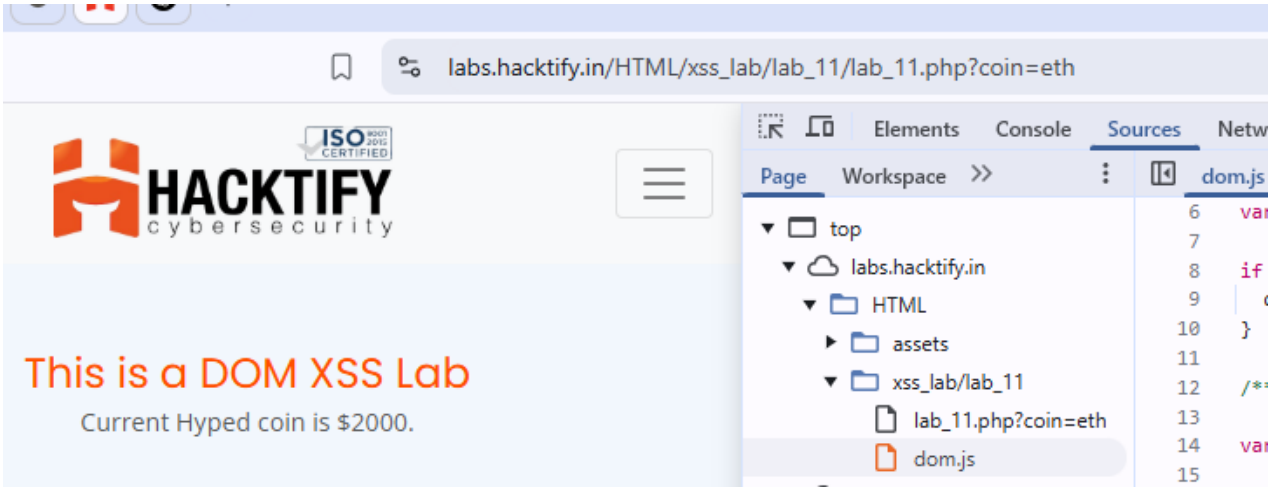
References

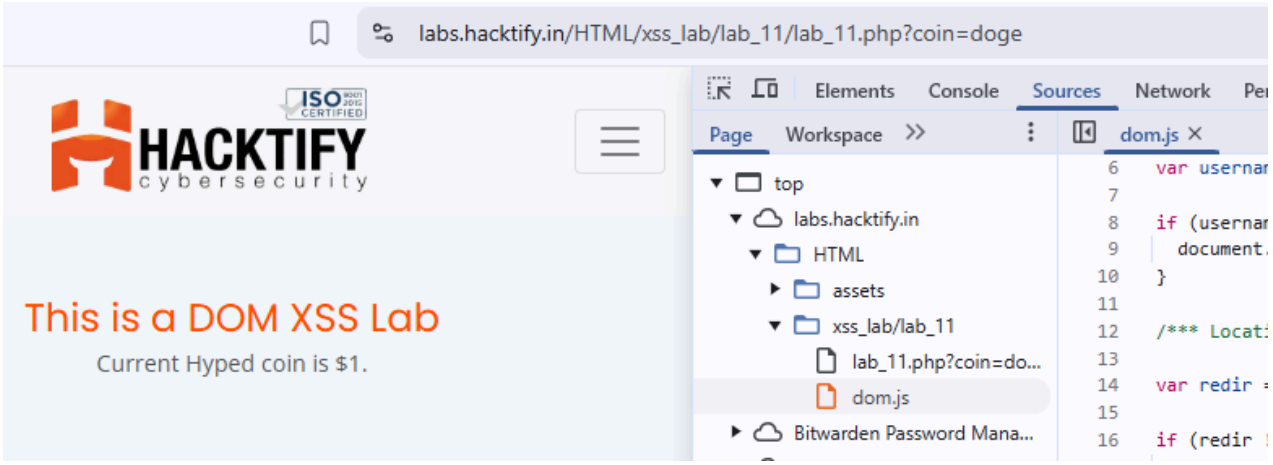
PortSwigger's DOM-based XSS Explanation: PortSwigger
Acunetix's Guide on Preventing DOM-based XSS: Acunetix
OWASP Cross-Site Scripting Prevention Cheat Sheet: OWASP Cheat Sheet Series

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab


labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?coin
HACKTIFY
cybersecurity
This is a DOM XSS Lab
The 3 coins are btc, eth, doge, !
Elements
Console
Page
Workspace
top
labs.hacktify.in
HTML
assets
xss_lab/lab_11
lab_11.php?coin
dom.js


labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?coin=eth
HACKTIFY
cybersecurity
This is a DOM XSS Lab
Current Hyped coin is $2000.
Elements
Console
Sources
Page
Workspace
dom.js
top
labs.hacktify.in
HTML
assets
xss_lab/lab_11
lab_11.php?coin=eth
dom.js


labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?coin=doge
HACKTIFY
cybersecurity
This is a DOM XSS Lab
Current Hyped coin is $1.
Elements
Console
Sources
Network
Page
Workspace
dom.js
top
labs.hacktify.in
HTML
assets
xss_lab/lab_11
lab_11.php?coin=do...
dom.js
Bitwarden Password Mana...

Week 2

3. Insecure Direct Object References

3.1. Give me my amount!!

Reference	Risk Rating
Give me my amount!!	**Low**
Tools Used	
Browser, Burp Suite	
Vulnerability Description	
An Insecure Direct Object Reference (IDOR) vulnerability exists because the application uses the customer number directly as a record index in queries to the back-end database, without proper access controls. By manipulating the id parameter in the URL, an attacker can access other users' profiles and modify their transaction details.This vulnerability allows for horizontal privilege escalation, where an attacker can access resources belonging to other users.	
How It Was Discovered	
The vulnerability was discovered through manual analysis by registering and logging in with the credentials test@test.com and test. By modifying the id parameter in the URL (profile.php?id=11 to id=11 and id=1), the profile of benep81280@whwow.com and alice@gmail.com was accessed respectively.I then modified the Transaction 1, Transaction 2, and Transaction 3 values for benep81280@whwow.com. Clicked "Update" to save changes. After logging out and repeating the steps to view benep81280@whwow.com's profile, the altered values were displayed.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_1/lab_1.php https://labs.hacktify.in/HTML/idor_lab/lab_1/profile.php?id=12 https://labs.hacktify.in/HTML/idor_lab/lab_1/profile.php?id=11 https://labs.hacktify.in/HTML/idor_lab/lab_1/profile.php?id=1	
Consequences of not Fixing the Issue	
• Privacy Violation: User data, including transaction details, can be accessed and modified by unauthorized individuals. • Data integrity: Unauthorized modification of transaction values can lead to incorrect financial records and loss of trust. • Reputational damage: The application's credibility can be severely impacted if user data is compromised. • Account Takeover: Attackers might be able to perform horizontal and vertical privilege escalation by altering the user to one with additional privileges while bypassing access controls.	
Suggested Countermeasures	
• Implement Proper Authorization Checks: Ensure that users can only access and modify their own data by verifying user permissions on the server side. • Use Indirect References: Employ indirect object references, such as mapping actual object IDs to random or hashed values, making it difficult for attackers to guess valid identifiers. • Input Validation and Sanitization: Validate and sanitize user inputs to prevent unauthorized data access. • Regular Security Audits: Conduct periodic security assessments to identify and remediate access control vulnerabilities.	

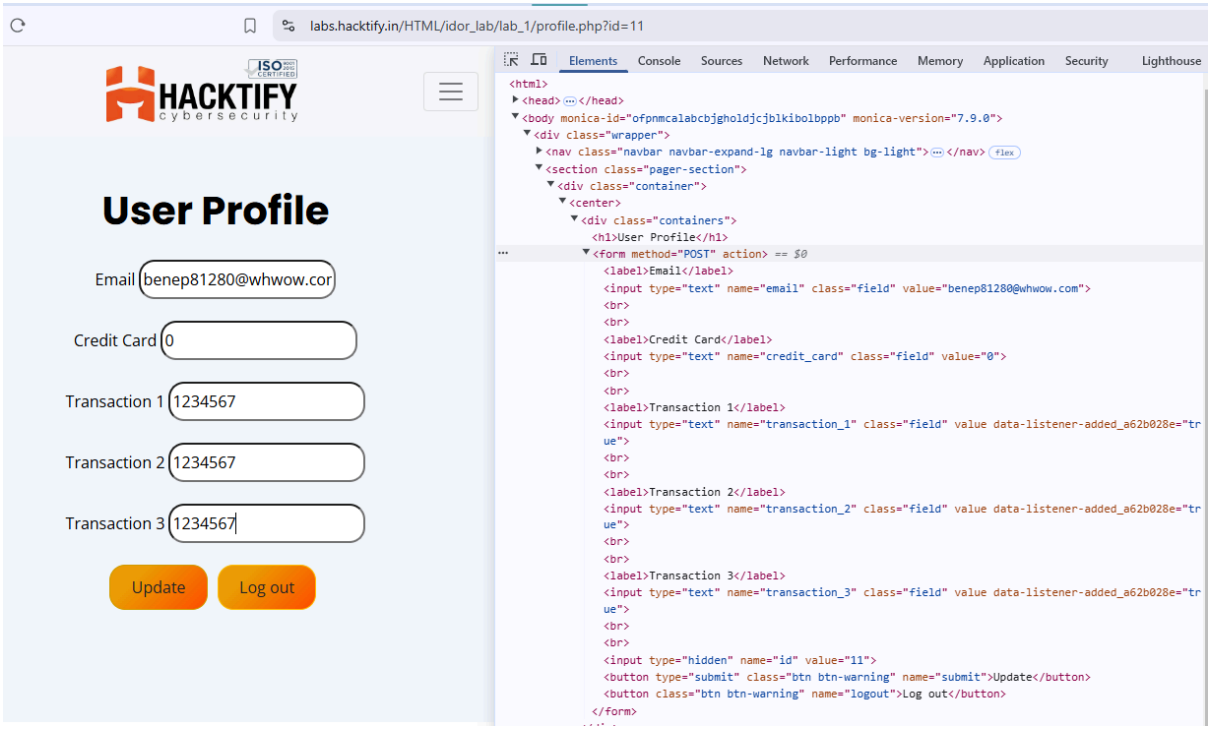
- User Session Management: Implement robust session management to track user activities and ensure that users can only perform actions within their authorized scope.

References

https://portswigger.net/web-security/access-control/idor
https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html
https://www.bugcrowd.com/blog/how-to-find-idor-insecure-direct-object-reference-vulnerabilities-for-large-bounty-rewards/

Proof of Concept

This proof of concept demonstrates the ability to access and modify another user's sensitive information due to the lack of proper authorization checks, highlighting the severity of the IDOR vulnerability present in the application.

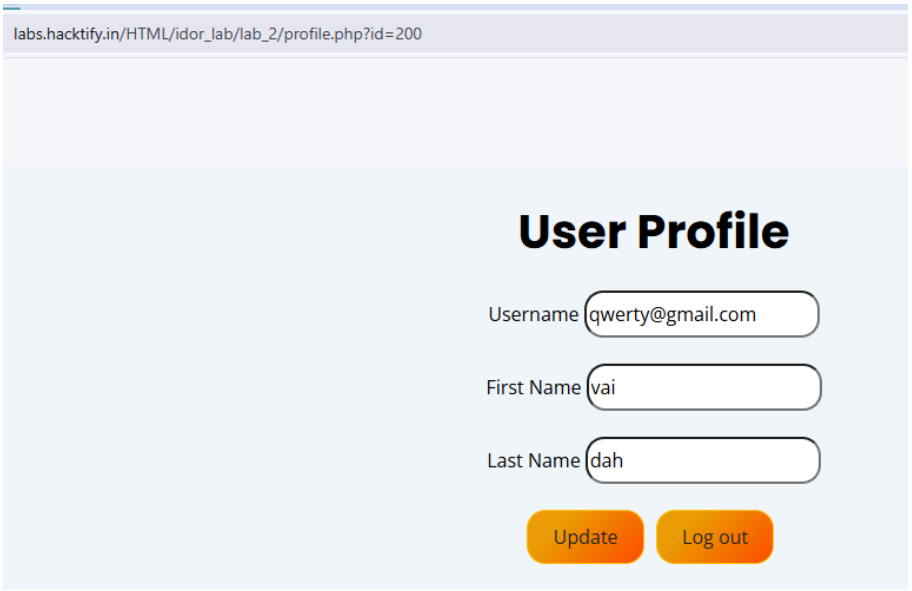


3.2. Stop polluting my params!

Reference	Risk Rating
Stop polluting my params!	**Medium**
Tools Used	
Web Browser (Chrome/Firefox)	
Vulnerability Description	
The application is vulnerable to Insecure Direct Object References (IDOR), allowing unauthorized access to user profiles by manipulating the id parameter in the URL. By incrementing or changing the id value, attackers can access other users' personal data, violating access controls. Unauthorized users can access and manipulate sensitive data by modifying object references in the URL. This can lead to privacy breaches, data leaks, and identity theft.	
How It Was Discovered	
I was able to log in with the credentials test@test.com and test. By modifying the default id parameter in the URL (profile.php?id=4 to id=1 and id=200), the profile of alice@gmail.com and qwerty@gmail.com was accessed respectively. Each User Profile displayed the Username, First Name, and Last Name values respectively.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_2/lab_2.php https://labs.hacktify.in/HTML/idor_lab/lab_2/profile.php?id=4 https://labs.hacktify.in/HTML/idor_lab/lab_2/profile.php?id=200 https://labs.hacktify.in/HTML/idor_lab/lab_2/profile.php?id=1	
Consequences of not Fixing the Issue	
• Unauthorized Data Access: Attackers can view, modify, or delete other users' information. • Privacy Violations: Exposure of personally identifiable information (PII). • Data Breaches: Large-scale exploitation can lead to regulatory penalties and reputational damage.	
Suggested Countermeasures	
• Implement Proper Access Controls – Use session-based user authentication to verify user privileges before displaying data. • Enforce Parameter Validation – Restrict direct user input in URL parameters. • Use Indirect Object References – Replace sequential id values with hashed or randomized identifiers. • Role-Based Access Control (RBAC) – Ensure that users can only access data associated with their accounts. • Conduct Regular Security Audits – Use automated tools (Burp Suite, OWASP ZAP) to detect IDOR vulnerabilities.	
References	
https://portswigger.net/web-security/access-control/idor https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html https://www.bugcrowd.com/blog/how-to-find-idor-insecure-direct-object-reference-vulnerabilities-for-large-bounty-rewards/	

Proof of Concept

The following confirms that unauthorized access is possible, proving the IDOR vulnerability exists.



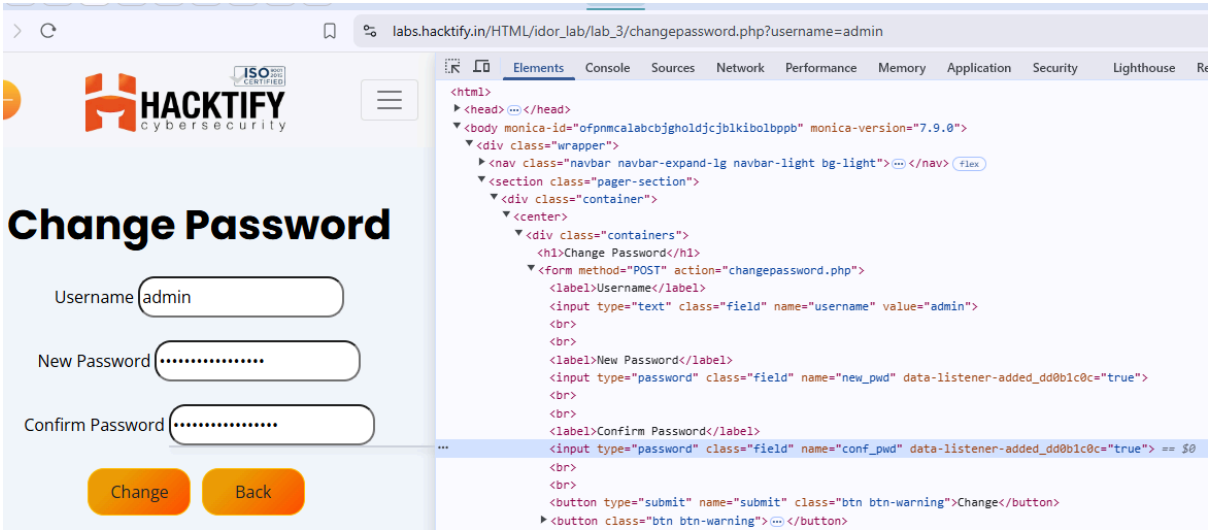
3.3. Someone changed my Password 🙀!

Reference	Risk Rating
Someone changed my Password 🙀!	**High**
Tools Used	
Web Browser (Chrome/Firefox)	
Vulnerability Description	
This lab explores an Insecure Direct Object References (IDOR) vulnerability that allows an attacker to change the password of an administrator account without authentication. The vulnerability arises due to the absence of proper access controls, allowing unauthorized users to modify the username parameter in the URL.	
How It Was Discovered	
I was able to log in without any credentials. Upon login, the User Profile page was displayed with the following fields, all of which were empty, Username, Email and Name. I clicked on the "Change password" button. On the Change Password page, I noticed the following URL (https://…username=), I modified the url to https:/…username=admin. The Change Password page for the admin account loaded, I changed the admin Password and Clicked "Change". Thus, I successfully modified the admin credentials, proving unauthorized access was possible.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_3/lab_3.php https://labs.hacktify.in/HTML/idor_lab/lab_3/profile.php https://labs.hacktify.in/HTML/idor_lab/lab_3/changepassword.php?username=test https://labs.hacktify.in/HTML/idor_lab/lab_3/changepassword.php?username=admin	
Consequences of not Fixing the Issue	
Successfully exploiting this flaw can lead to unauthorized access, privilege escalation, and potential account takeovers.	
Suggested Countermeasures	
• Implement Role-Based Access Control (RBAC): Ensure users can only modify their own accounts. • Use Session-Based Authentication: Validate sessions before allowing access to sensitive pages.	

• Enforce Parameter Validation: Restrict direct input manipulation in URL parameters. • Use Indirect Object References: Replace direct usernames with hashed or randomized identifiers. • Regular Security Audits: Conduct penetration testing to detect similar vulnerabilities.
References
• https://portswigger.net/web-security/access-control/idor • https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html • https://www.bugcrowd.com/blog/how-to-find-idor-insecure-direct-object-reference-vulnerabilities-for-large-bounty-rewards/

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



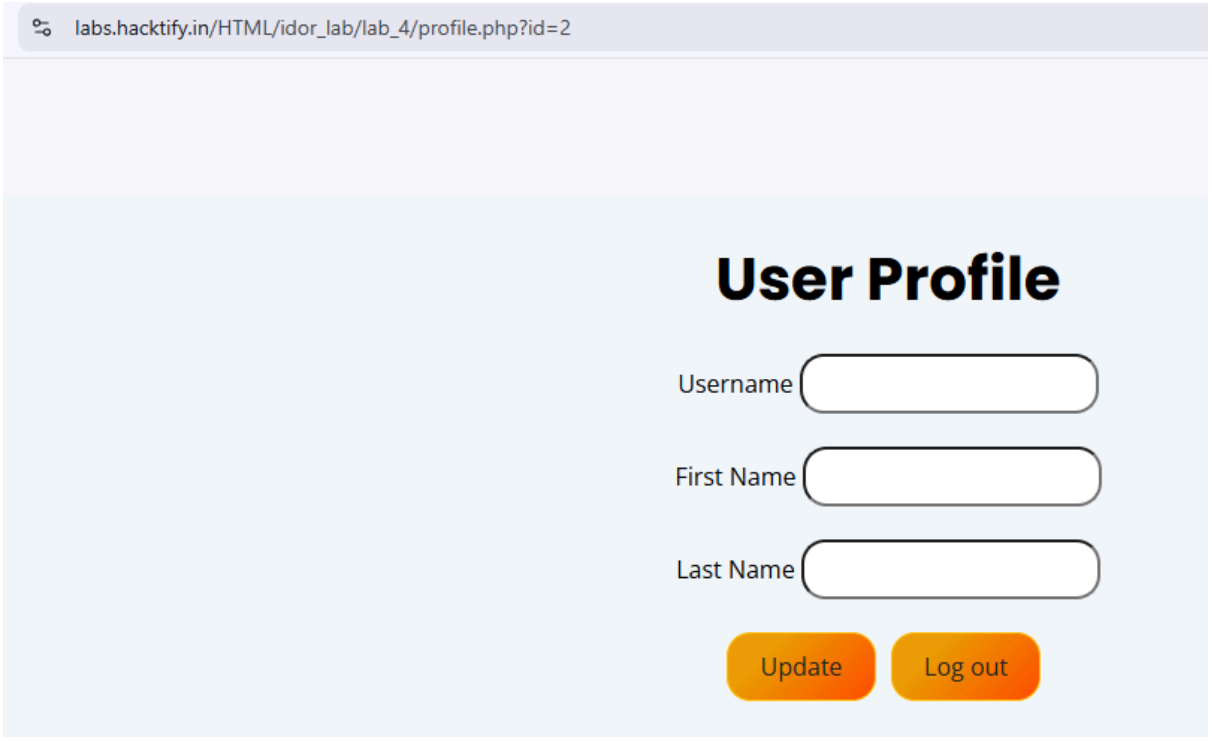
3.4. Change your methods!

Reference	Risk Rating
Change your methods!	**Medium**
Tools Used	
Web browser (Google Chrome, Mozilla Firefox)	
Vulnerability Description	
Insecure Direct Object Reference (IDOR) vulnerabilities occur when an application provides direct access to objects based on user-supplied input, without proper authorization checks. This can allow attackers to access or modify data they are not authorized to.	
How It Was Discovered	
During testing, it was observed that the user profile page URL contained a parameter id that appeared to reference user-specific data. By manipulating this parameter, it was possible to access other users' profiles without proper authorization.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_4/lab_4.php https://labs.hacktify.in/HTML/idor_lab/lab_4/profile.php?id=473 https://labs.hacktify.in/HTML/idor_lab/lab_4/profile.php?id=2	

Consequences of not Fixing the Issue
Failure to address this vulnerability can lead to unauthorized access to sensitive user information, data breaches, and potential legal ramifications.
Suggested Countermeasures
• Implement proper authorization checks to ensure users can only access resources they are permitted to. • Use indirect references (e.g., mapping actual object references to internal identifiers). • Conduct regular security assessments to identify and remediate such vulnerabilities.
References
• https://portswigger.net/web-security/access-control/idor • https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html • https://www.bugcrowd.com/blog/how-to-find-idor-insecure-direct-object-reference-vulnerabilities-for-large-bounty-rewards/

Proof of Concept

By accessing the URL id= while authenticated as user with https://…id=473, the profile information of user id=2 was displayed, confirming the IDOR vulnerability.



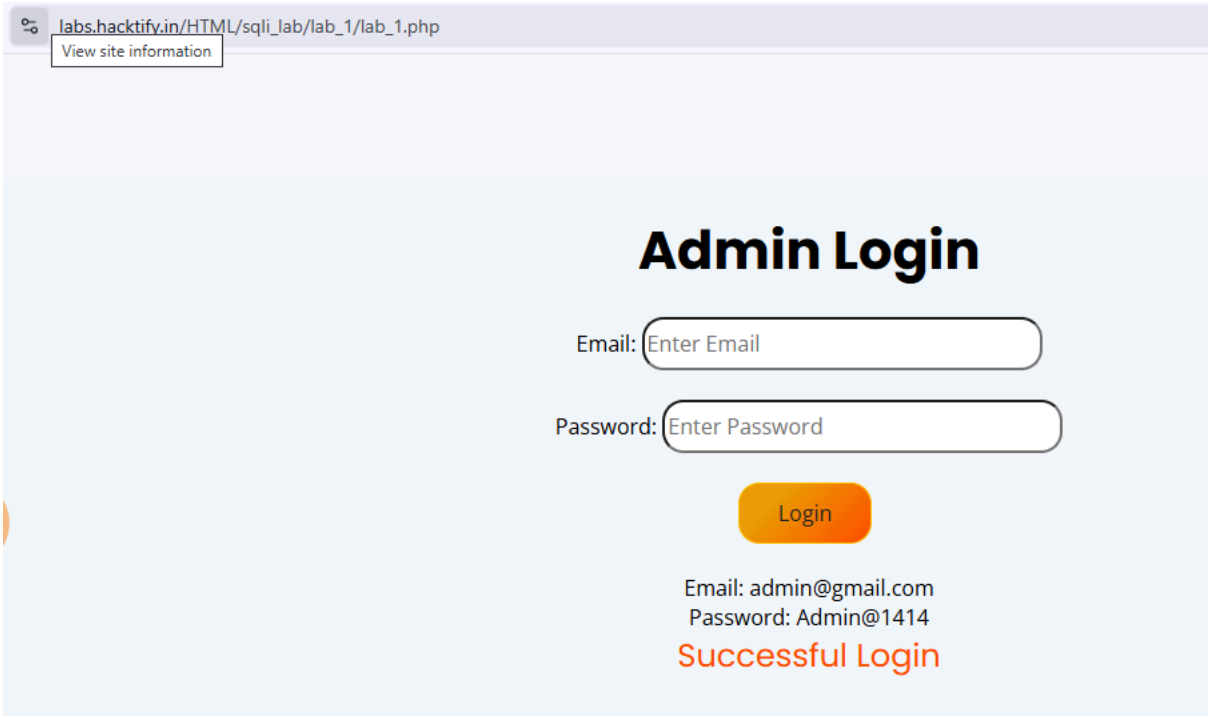
4. SQL Injection

4.1. Strings & Errors Part 1!

Reference	Risk Rating
Strings & Errors Part 1!	**Low**
Tools Used	
Web Browser Tools (Inspect and Source Code)	
Vulnerability Description	
SQL Injection is a critical security vulnerability that allows attackers to interfere with the queries an application makes to its database. This can lead to unauthorized access to sensitive data, modification of data, or even complete compromise of the database server. In this instance, the login functionality fails to properly sanitize user inputs, making it susceptible to SQL Injection attacks.	
How It Was Discovered	
During testing, the login form was identified as a potential entry point for SQL Injection. By inputting the payload (admin" or "1"="1) or (" OR "1"="1) into both the email and password fields, access was granted without valid credentials, and the real admin credentials were displayed on the screen, thus indicating a successful SQL Injection attack.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_1/lab_1.php	
Consequences of not Fixing the Issue	
If left unaddressed, this vulnerability could allow attackers to: • Bypass authentication mechanisms. • Access, modify, or delete sensitive data. • Execute administrative operations on the database. • Potentially compromise the entire server hosting the database.	
Suggested Countermeasures	
• **Input Validation:** Implement strict input validation to ensure only expected data types and formats are accepted. • **Parameterized Queries:** Use prepared statements with parameterized queries to prevent the injection of malicious SQL code. • **Error Handling:** Configure the application to handle errors gracefully without exposing detailed database error messages to users. • **Regular Security Audits:** Conduct periodic security assessments to identify and remediate potential vulnerabilities.	
References	
OWASP SQL Injection Prevention Cheat Sheet OWASP Testing Guide: SQL Injection Testing	

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



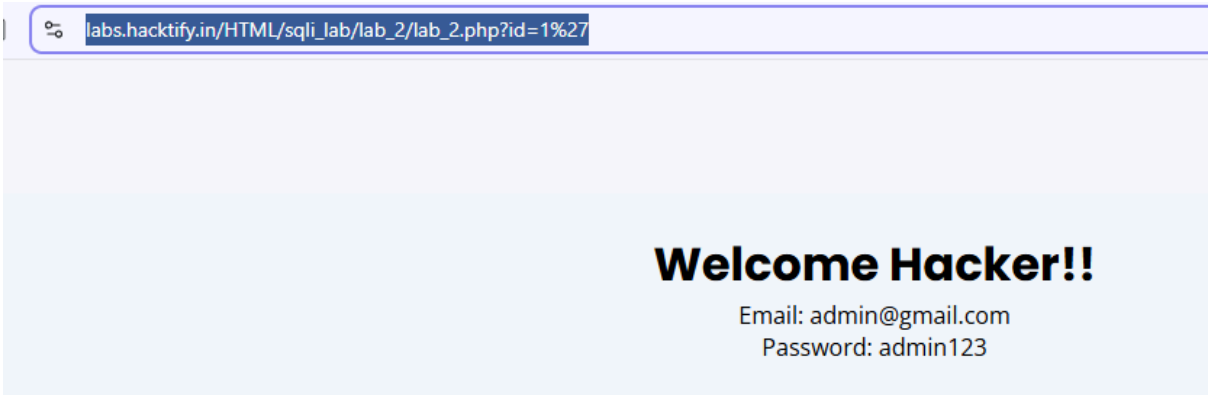
4.2. Strings & Errors Part 2!

Reference	Risk Rating
Strings & Errors Part 2!	**Low**
Tools Used	
Web Browser Tools (Inspect and Source Code)	
Vulnerability Description	
The application is vulnerable to SQL Injection, a critical security flaw that allows attackers to interfere with the queries an application makes to its database. By manipulating input parameters, attackers can access unauthorized data, modify or delete data, and execute administrative operations on the database.	
How It Was Discovered	
During testing, appending ?id=1' to the URL resulted in the display of admin credentials. This behavior indicates that the input is not properly sanitized, allowing malicious SQL code to alter the database query execution. Adding ?id=1' UNION SELECT 1,2,3,4--+ to the URL returns the database schema (id) of the Email and password	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php?id=1%27	
Consequences of not Fixing the Issue	
If unaddressed, this vulnerability could lead to severe consequences, including unauthorized access to sensitive information, data breaches, loss of data integrity, and potential full system compromise.	
Suggested Countermeasures	
• Input Validation: Implement strict input validation to ensure only expected data types and formats are accepted. • Parameterized Queries: Use prepared statements with parameterized queries to prevent the inclusion of malicious SQL code. • Stored Procedures: Utilize stored procedures to encapsulate SQL statements, reducing direct interaction with the database.	

• Error Handling: Avoid displaying detailed database error messages to users, as they can provide insights for exploitation.
References
• OWASP SQL Injection Prevention Cheat Sheet • OWASP Testing Guide: SQL Injection Testing

Proof of Concept

By appending ?id=1' to the URL https://…lab_2.php, the application reveals sensitive admin credentials, demonstrating the presence of an SQL Injection vulnerability.



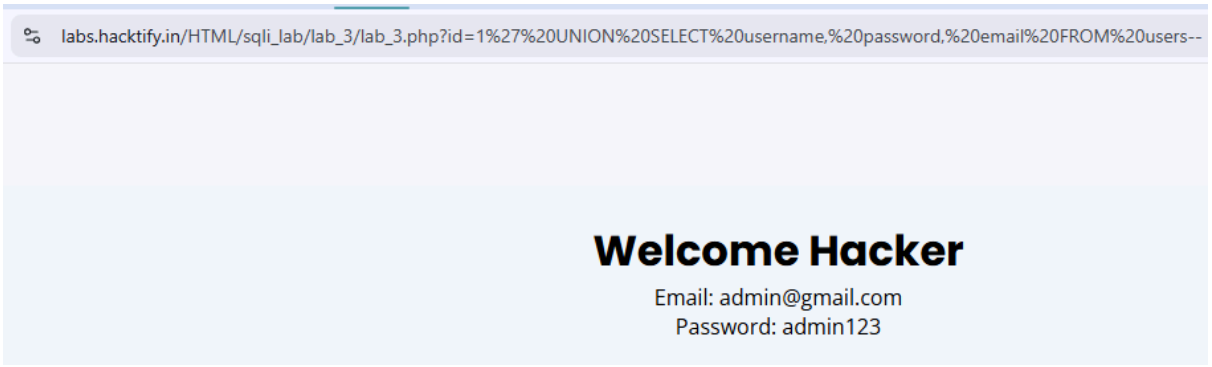
4.3. Strings & Errors Part 3!

Reference	Risk Rating
Strings & Errors Part 3!	**Low**
Tools Used	
Web Browser Tools (Inspect and Source Code)	
Vulnerability Description	
The application is vulnerable to SQL Injection, a critical security flaw that allows attackers to manipulate SQL queries by injecting malicious input through URL parameters. This can lead to unauthorized access to sensitive data, such as user credentials.	
How It Was Discovered	
By appending specific payloads to the id parameter in the URL, the application returned error messages and, eventually, sensitive data. For example:https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php?id=1' UNION SELECT username, password, email FROM users--.This payload caused the application to display admin credentials, indicating a successful SQL Injection attack.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php	
Consequences of not Fixing the Issue	
Failure to address this vulnerability can result in: Unauthorized access to sensitive information, Compromise of user accounts, Potential full system compromise, Legal and reputational damage	
Suggested Countermeasures	
Implement parameterized queries or prepared statements to prevent SQL Injection. Sanitize and validate all user inputs. Use stored procedures for database interactions.	

Employ web application firewalls (WAF) to detect and block malicious requests.
References
• OWASP SQL Injection Prevention Cheat Sheet • OWASP Testing Guide: SQL Injection Testing

Proof of Concept

The following proof confirms the presence of a SQL Injection vulnerability that exposes sensitive user credentials.



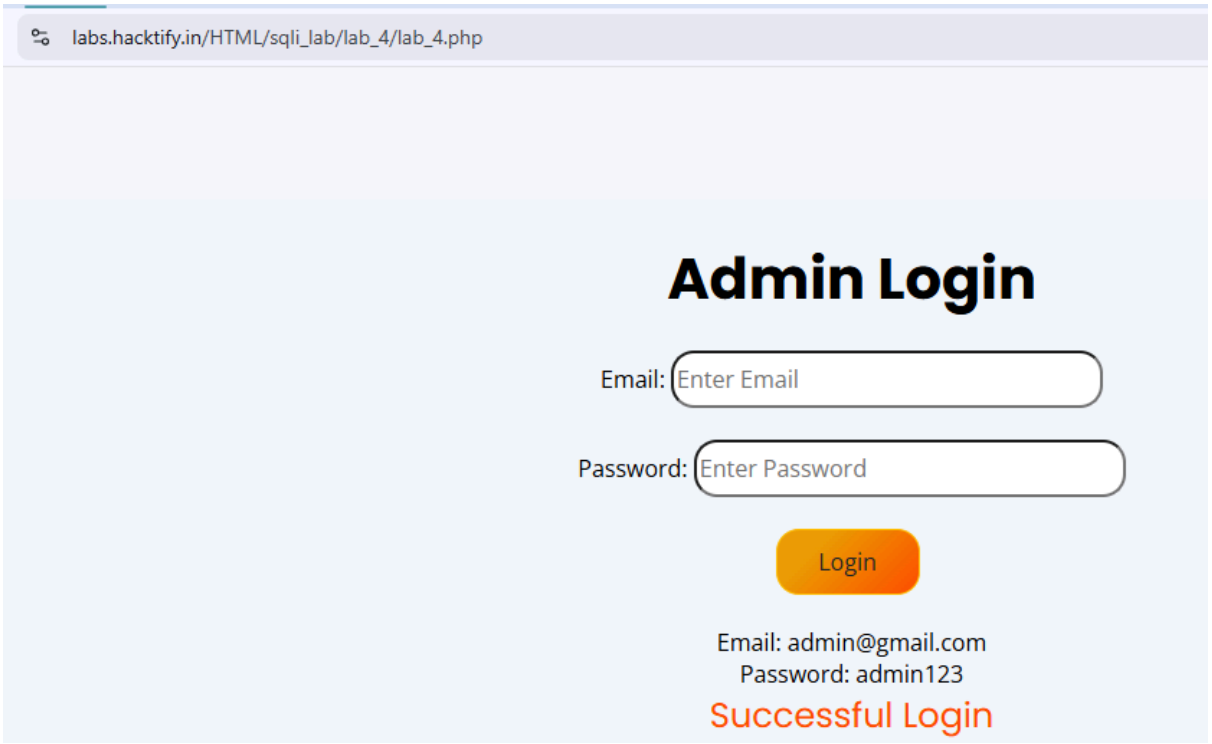
4.4. Let's Trick 'em!

Reference	Risk Rating
Sub-lab-4. Let's Trick 'em!	**Medium**
Tools Used	
Web Browser Tools (Inspect and Source Code)	
Vulnerability Description	
The application is vulnerable to SQL Injection, a critical security flaw that allows attackers to interfere with the queries an application makes to its database. This vulnerability arises when user inputs are improperly sanitized, enabling malicious SQL code execution.	
How It Was Discovered	
Each input field was tested by entering a single quote (') to detect SQL errors. The application returned error messages upon inputting the single quote, indicating potential SQL injection points. Given the discovery of the payload 1' || '1'='1 on line 103 of the source code, I used the payload 1' || '1'='1 in both the email and password fields resulted in a successful login, confirming the vulnerability.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_4/lab_4.php	
Consequences of not Fixing the Issue	
If unaddressed, attackers can: Bypass authentication mechanisms. Access, modify, or delete data in the database. Execute administrative operations on the database. Extract sensitive information, leading to data breaches.	
Suggested Countermeasures	
• Input Validation: Implement strict input validation to ensure only expected data types and formats are accepted. • Parameterized Queries: Use parameterized queries or prepared statements to separate SQL code from user inputs. • Error Handling: Configure the application to handle errors gracefully without exposing stack traces or database errors to users.	

• Regular Security Audits: Conduct periodic security assessments to identify and remediate vulnerabilities promptly.
References
• OWASP SQL Injection Prevention Cheat Sheet • OWASP Testing Guide: SQL Injection Testing

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



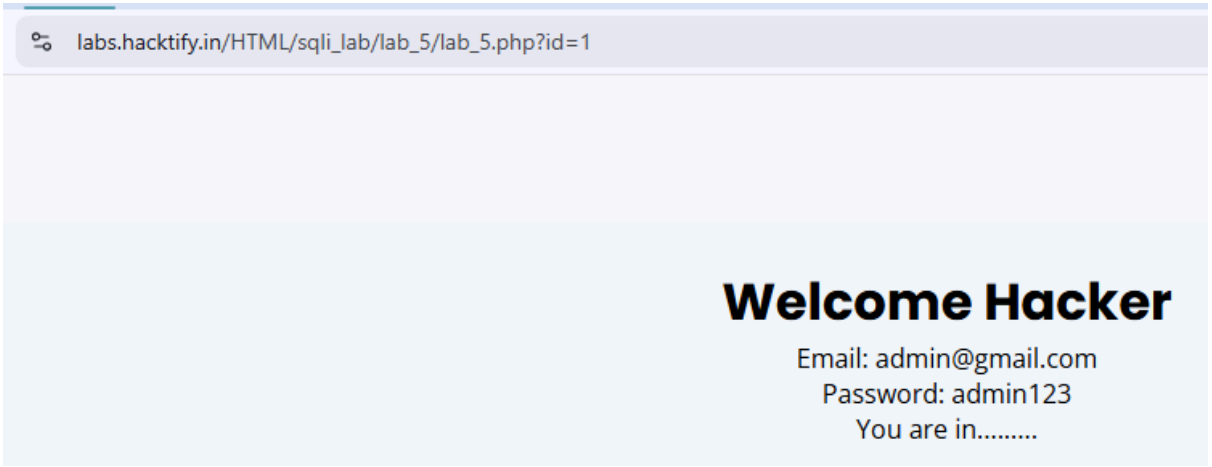
4.5. Booleans and Blind!

Reference	Risk Rating
Booleans and Blind!	**High**
Tools Used	
Web Browser Tools (Inspect and Source Code)	
Vulnerability Description	
The application is vulnerable to SQL Injection, a critical security flaw that allows attackers to interfere with the queries an application makes to its database. By manipulating input parameters, attackers can access, modify, or delete data without authorization.	
How It Was Discovered	
Appending ?id=1 to the URL resulted in the display of sensitive information. This behavior indicates that the id parameter is directly used in a database query without proper sanitization, leading to potential SQL Injection.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php?id=1	

Consequences of not Fixing the Issue
Unauthorized access to sensitive data, Data loss or corruption, Compromise of entire database Reputation damage, Legal and regulatory repercussions
Suggested Countermeasures
• Implement parameterized queries or prepared statements to prevent SQL Injection. • Validate and sanitize all user inputs. • Use ORM frameworks that abstract SQL queries. • Regularly update and patch database management systems. • Conduct security code reviews and penetration testing.
References
• OWASP SQL Injection Prevention Cheat Sheet • OWASP Testing Guide: SQL Injection Testing

Proof of Concept

By appending ?id=1 to the URL, the application reveals sensitive information, demonstrating the presence of an SQL Injection vulnerability.



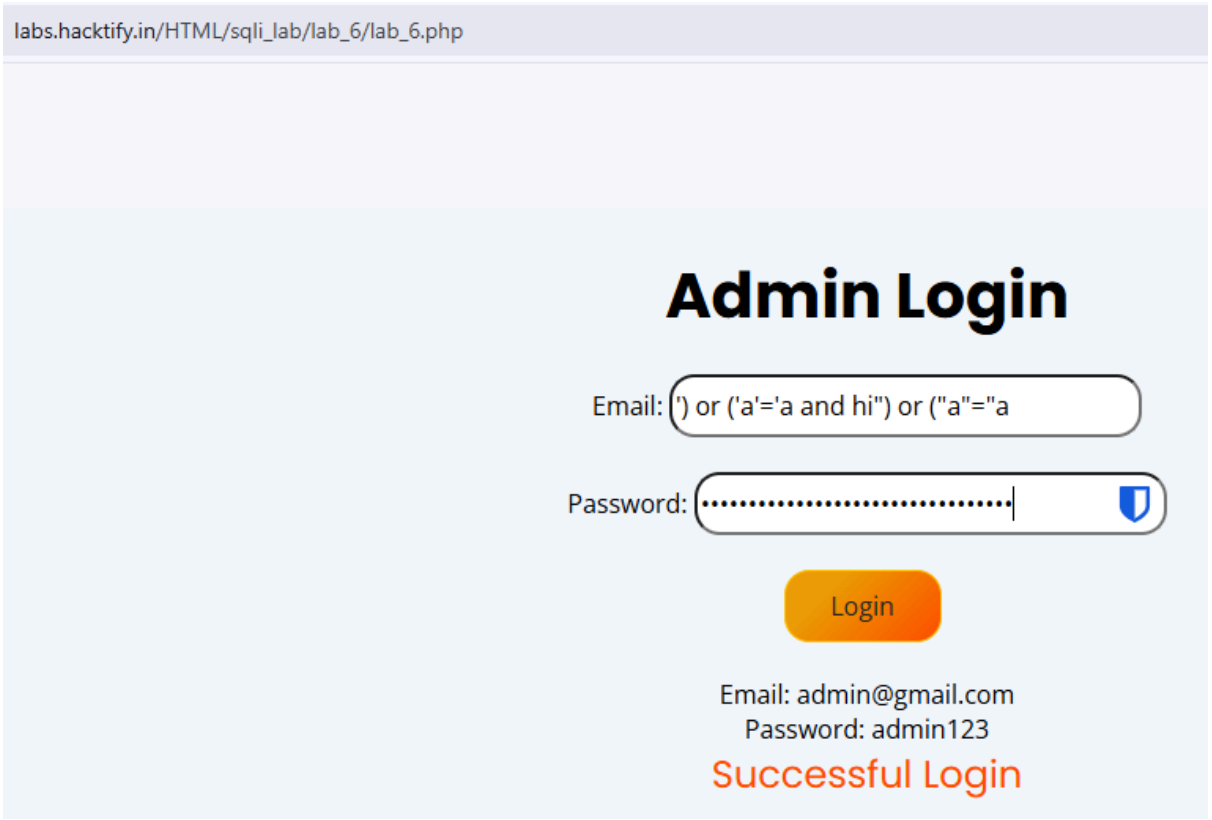
4.6. Error Based : Tricked

Reference	Risk Rating
6: Error Based : Tricked	**Medium**
Tools Used	
Web Browser Tools (Inspect and Source Code)	
Vulnerability Description	
About the vulnerability and its working	
How It Was Discovered	
During testing, Line 103 of the source code, <!-- use of payload ") or ("1")=("1 --> The following payload was input into both the email and password fields: ') or ('a'='a and hi") or ("a"="a Upon submission, the application granted access and displayed the real admin credentials. This behavior indicates that the application does not properly sanitize user inputs, allowing SQL code to alter the intended query logic.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_6/lab_6.php	
Consequences of not Fixing the Issue	
If unaddressed, this vulnerability could allow attackers to:	

• Gain unauthorized access to user accounts, including administrative accounts. • Extract, modify, or delete sensitive data from the database. • Execute administrative operations on the database, potentially compromising the entire application.
Suggested Countermeasures
• Input Validation: Implement strict input validation to ensure that user inputs conform to expected formats and reject any inputs containing SQL syntax. • Parameterized Queries: Use parameterized queries or prepared statements to separate SQL code from user inputs, preventing injection attacks. • Error Handling: Configure the application to handle database errors gracefully without exposing detailed error messages to users, which can aid attackers. • Regular Security Testing: Conduct regular security assessments, including automated and manual testing, to identify and remediate vulnerabilities promptly.
References
• OWASP SQL Injection Prevention Cheat Sheet • OWASP Testing Guide: SQL Injection Testing

Proof of Concept

This confirms the presence of a SQL Injection vulnerability in the login functionality.



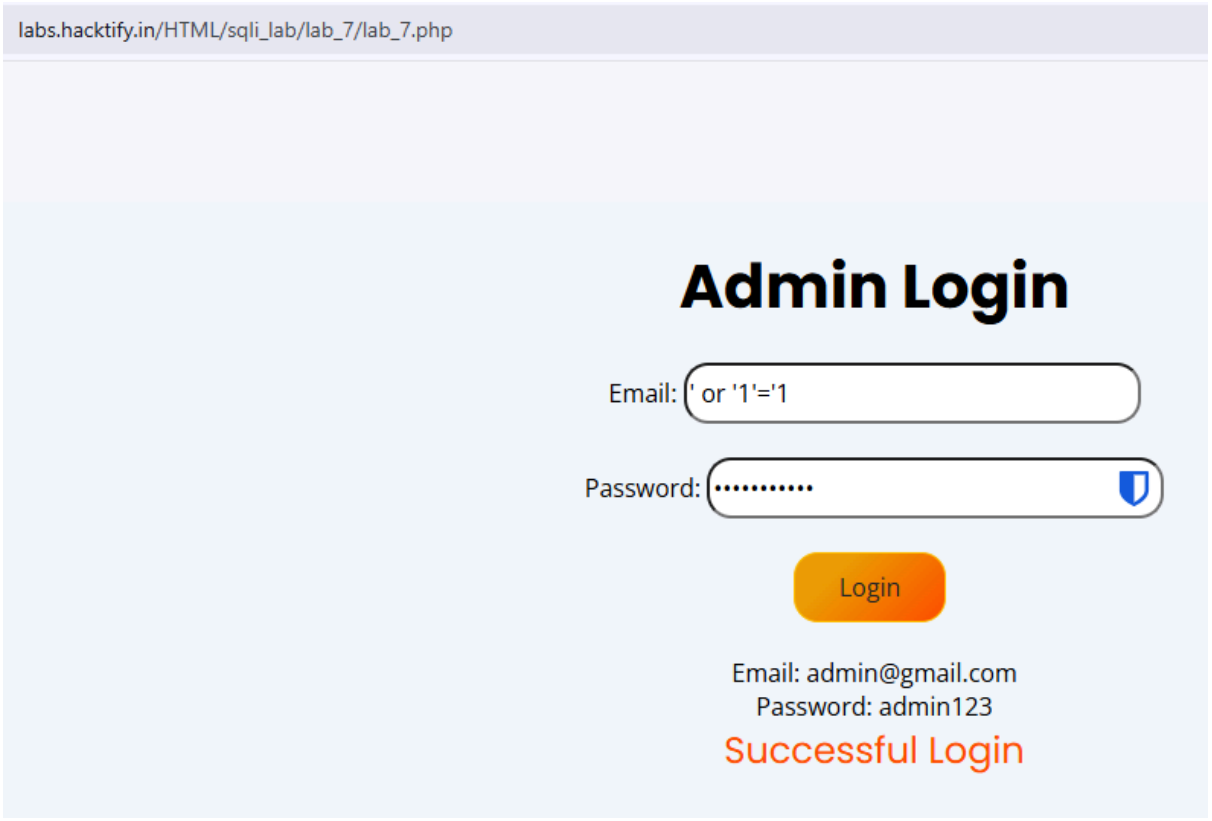
4.7. Errors and Post!

Reference	Risk Rating
Errors and Post!	**Low**
Tools Used	
Web Browser Tools (Inspect and Source Code)	

Vulnerability Description
SQL Injection occurs when an application incorporates unvalidated user input into SQL queries, allowing attackers to manipulate the query structure. The payload ' or '1'='1 exploits this flaw by injecting a condition that always evaluates to true, effectively bypassing authentication checks. This can grant unauthorized access to user accounts and sensitive information.
How It Was Discovered
During security testing, input fields were tested with the payload ' or '1'='1. The application granted access without proper credentials, indicating that user inputs were directly embedded into SQL queries without adequate sanitization or parameterization.
Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_7/lab_7.php
Consequences of not Fixing the Issue
If unaddressed, this vulnerability can lead to severe security breaches, including: • Unauthorized Data Access: Attackers can retrieve, modify, or delete sensitive information from the database. • Authentication Bypass: Malicious users can gain administrative privileges without valid credentials. • Potential System Compromise: Exploiting this flaw may allow execution of arbitrary commands, leading to complete system takeover.
Suggested Countermeasures
• Input Validation: Implement strict validation to ensure inputs conform to expected formats and types. • Parameterized Queries: Use prepared statements with parameterized queries to prevent injection attacks. • Stored Procedures: Encapsulate SQL queries within stored procedures to limit direct interaction with the database. • Least Privilege Principle: Ensure database accounts have the minimum privileges necessary for their functions. • Regular Security Audits: Conduct periodic code reviews and vulnerability assessments to identify and remediate security flaws.
References
• OWASP SQL Injection Prevention Cheat Sheet • OWASP Testing Guide: SQL Injection Testing

Proof of Concept

Upon submission, the application granted access without validating legitimate credentials, confirming the presence of an SQL Injection vulnerability.



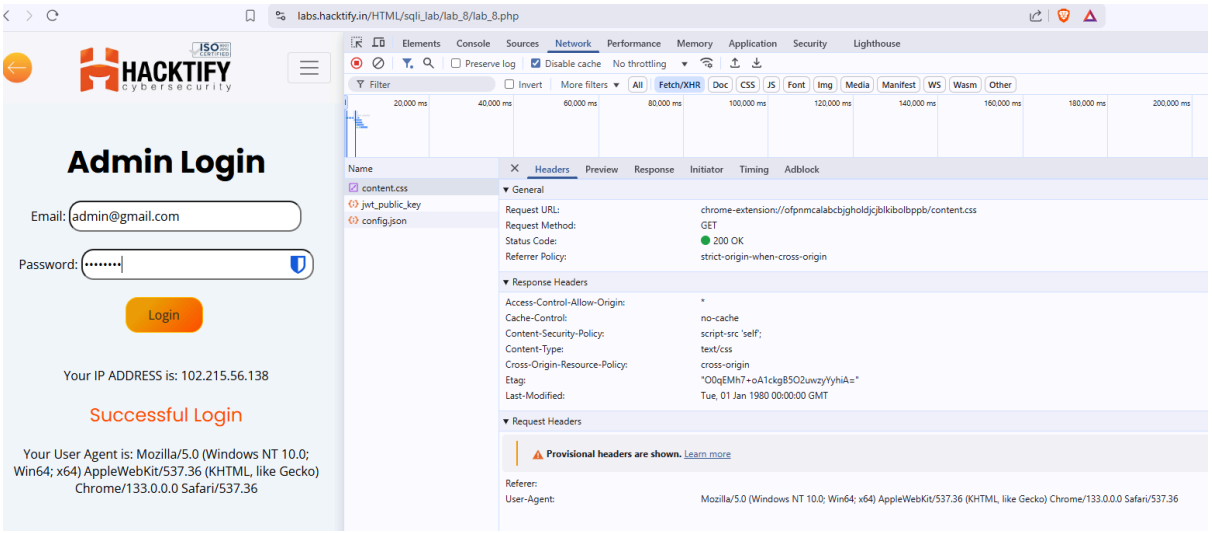
4.8. User Agents lead us!

Reference	Risk Rating
User Agents lead us!	**High**
Tools Used	
Web Browser Tools (Inspect and Source Code)	
Vulnerability Description	
The application is susceptible to SQL Injection through the User-Agent HTTP header. When a user logs in, their User-Agent string is inserted into the database without proper sanitization. An attacker can exploit this by injecting malicious SQL code into the User-Agent header, potentially compromising the database.	
How It Was Discovered	
After successfully logging in with the credentials admin@gmail.com and admin123, the application displayed the User-Agent string. By injecting a single quote (') into the User-Agent header and observing an SQL error message, it was evident that the input was not properly sanitized, confirming the SQL Injection vulnerability	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_8/lab_8.php	
Consequences of not Fixing the Issue	
If unaddressed, attackers can: Execute arbitrary SQL commands, Access, modify, or delete sensitive data, Compromise the entire database.	
Suggested Countermeasures	
• Input Validation: Sanitize all inputs, including HTTP headers, to remove or escape harmful characters. • Prepared Statements: Use parameterized queries to prevent SQL Injection.	

• Web Application Firewall (WAF): Deploy a WAF to detect and block malicious requests
References
• OWASP SQL Injection Prevention Cheat Sheet • OWASP Testing Guide: SQL Injection Testing

Proof of Concept

This section contains the proof of the above vulnerabilities as the screenshot of the vulnerability of the lab



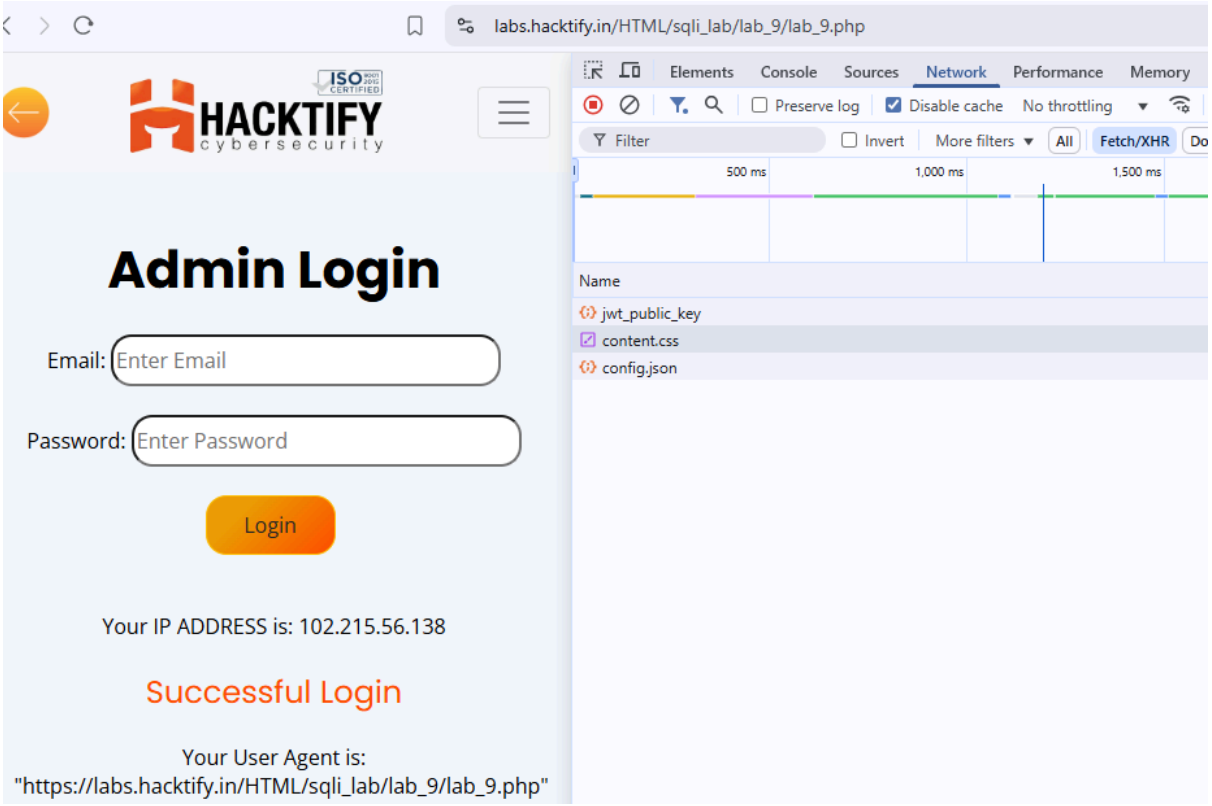
4.9. Referer lead us!

Reference	Risk Rating
Referer lead us!	**Medium**
Tools Used	
Web Browser Tools (Inspect and Source Code)	
Vulnerability Description	
The application is vulnerable to SQL Injection through the HTTP Referer header. By manipulating this header, an attacker can inject malicious SQL code, potentially gaining unauthorized access to sensitive data or compromising the database.	
How It Was Discovered	
Used browser developer tools to inspect network requests and identify the Referer header as a potential injection point. Employed Burp Suite to intercept HTTP requests and modify the Referer header with SQL injection payloads. Observed application responses for SQL errors or unexpected behavior indicative of successful injection.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_9/lab_9.php	
Consequences of not Fixing the Issue	
Unauthorized access to sensitive information. Compromise of database integrity and confidentiality. Potential for full system compromise if leveraged further.	
Suggested Countermeasures	
Implement input validation and sanitization for all user inputs, including HTTP headers. Use parameterized queries or prepared statements to prevent SQL injection.	

Avoid relying on HTTP headers for critical application logic or security decisions.
References
• OWASP SQL Injection Prevention Cheat Sheet • OWASP Testing Guide: SQL Injection Testing

Proof of Concept

This demonstrates the critical need for proper input handling and validation to prevent such vulnerabilities.



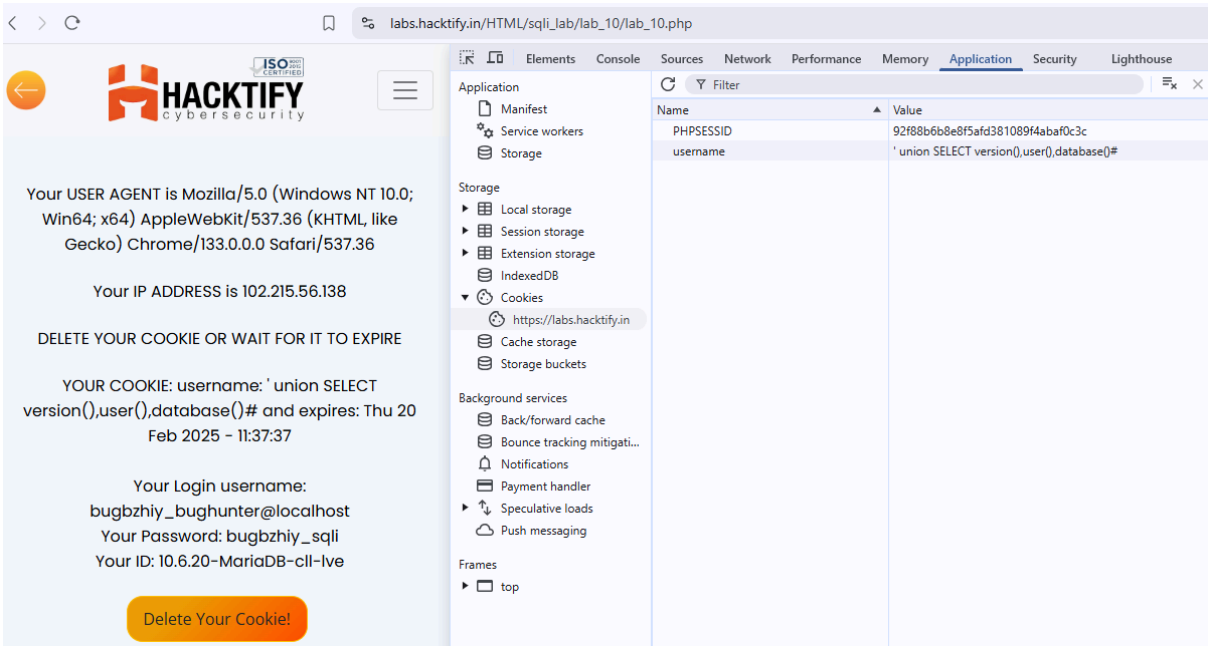
4.10. Oh Cookies!

Reference	Risk Rating
Oh Cookies!	**High**
Tools Used	
Web Browser Developer Tools (Inspect Element) SQL Injection Payloads	
Vulnerability Description	
The application is vulnerable to SQL Injection through the manipulation of cookie parameters. By altering the 'username' cookie value with malicious SQL code, an attacker can execute arbitrary SQL queries against the database. This vulnerability arises from insufficient validation and sanitization of cookie data on the server side.	
How It Was Discovered	
1. Initial Login: Logged in using credentials 'admin' for both username and password. 2. Session Analysis: Observed the 'username' cookie set upon successful login.	

3. Cookie Manipulation: Using browser's developer tools, modified the 'username' cookie to include a SQL injection payload: ' union SELECT version(),user(),database()#. 4. Upon refreshing the page, the application displayed database version, current user, and database name, confirming successful SQL injection.
Vulnerable URLs
https://labs.hacktify.in/HTML/sqli_lab/lab_10/lab_10.php
Consequences of not Fixing the Issue
Unauthorized access to sensitive data. Compromise of database integrity. Potential for complete system takeover. Damage to reputation and potential legal implications.
Suggested Countermeasures
• Input Validation: Implement strict validation and sanitization of all user inputs, including cookies. • Parameterized Queries: Use prepared statements with parameterized queries to prevent SQL injection. • Least Privilege Principle: Ensure the database user has the minimum privileges necessary. • Regular Security Audits: Conduct periodic security assessments to identify and remediate vulnerabilities.
References
• OWASP: SQL Injection • CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Proof of Concept

This demonstrates the application's susceptibility to SQL injection via cookie manipulation.



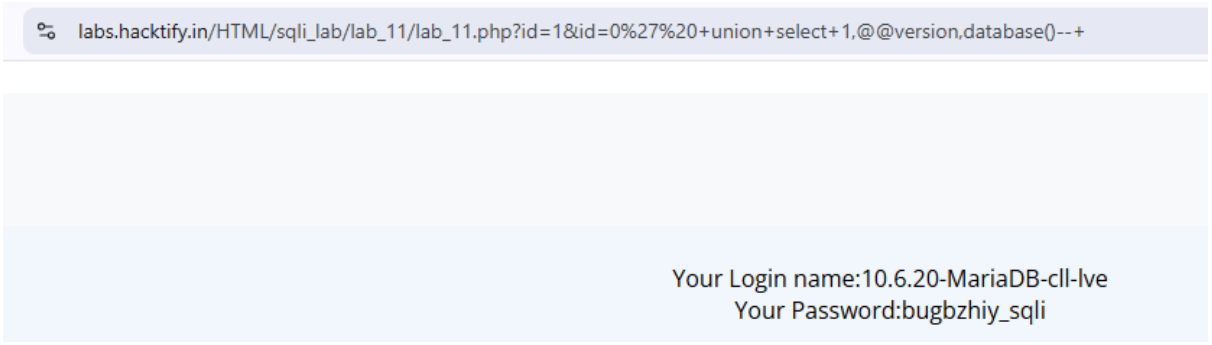
4.11. WAF's are injected!

Reference	Risk Rating

WAF's are injected Part 2!	**High**
Tools Used	
Web Browser Tools (Inspect and Source Code)	
Vulnerability Description	
The application is vulnerable to a SQL Injection attack, specifically through the id parameter in the URL. By manipulating this parameter, an attacker can execute arbitrary SQL commands, leading to unauthorized access to sensitive information stored in the database.	
How It Was Discovered	
During testing, the following payload was appended to the URL ?id=1&id=0' +union+select+1,@@version,database()--+ This payload exploits the SQL Injection vulnerability by injecting a UNION SELECT statement to retrieve the database version and name. This indicates that the application executed the injected SQL command and returned sensitive database information.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_11/lab_11.php https://labs.hacktify.in/HTML/sqli_lab/lab_11/lab_11.php?id=1&id=0%27%20+union+select+1,@@version,database()--+	
Consequences of not Fixing the Issue	
If left unaddressed, attackers can exploit this vulnerability to: Access and exfiltrate sensitive data, including user credentials and personal information. • Modify or delete data, leading to data integrity issues. • Execute administrative operations on the database. • Compromise the entire application and potentially the underlying server.	
Suggested Countermeasures	
• Input Validation: Implement strict input validation to ensure that user-supplied data conforms to the expected format and type. • Parameterized Queries: Use prepared statements with parameterized queries to separate SQL logic from data, preventing injection. • Stored Procedures: Utilize stored procedures to encapsulate SQL queries, reducing direct interaction with SQL commands. • Least Privilege Principle: Configure the database with the least privileges necessary for the application to function, limiting the potential impact of a successful injection. • Regular Security Audits: Conduct periodic code reviews and security testing to identify and remediate vulnerabilities promptly.	
References	
• OWASP SQL Injection Prevention Cheat Sheet • OWASP Testing Guide: SQL Injection Testing	

Proof of Concept

This demonstrates the ability to execute arbitrary SQL commands and retrieve sensitive information from the database using ?id=1&id=0%27%20+union+select+1,@@version,database()--+



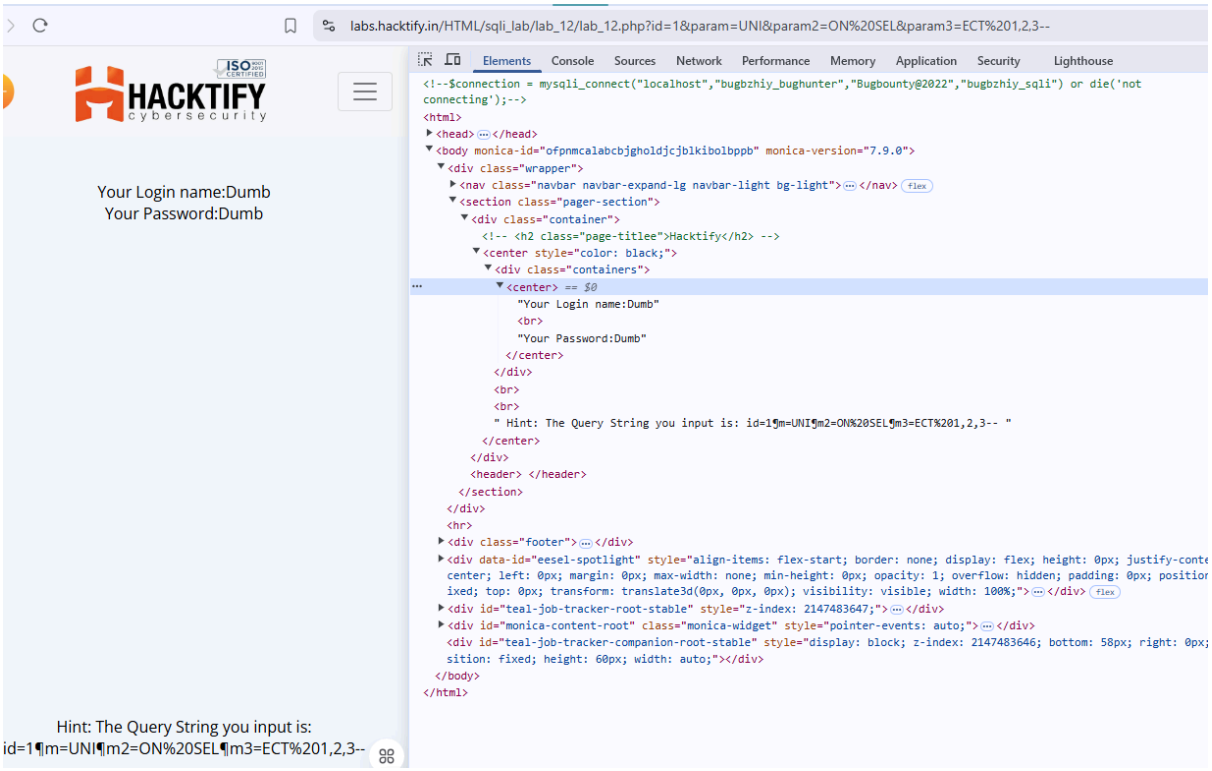
4.12. WAF's are injected Part 2!

Reference	Risk Rating
WAF's are injected Part 2!	**High**
Tools Used	
Web Browser Tools (Inspect and Source Code)	
Vulnerability Description	
The application is vulnerable to SQL Injection attacks, allowing attackers to execute arbitrary SQL code. This can lead to unauthorized access to sensitive information, such as usernames and passwords. In this instance, the Web Application Firewall (WAF) intended to protect the application was bypassed using crafted payloads.	
How It Was Discovered	
By inputting specific SQL payloads into the URL parameters, unexpected data was returned, indicating successful injection and WAF bypass. For example:http://example.com/index.php?id=1– http://example.com/index.php?id=1&param=UNI&param2=ON SEL&param3=ECT 1,2,3-- The application responded with login credentials	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php?id=1– https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php?id=1&param=UNI&param2=ON%20SEL&param3=ECT%201,2,3– https://labs.hacktify.in/HTML/sqli_lab/lab_12/hacked.php	
Consequences of not Fixing the Issue	
Failure to address this vulnerability can result in: Unauthorized access to sensitive data Compromise of user accounts, Potential full system compromise, Damage to reputation and trust	
Suggested Countermeasures	
• Implement parameterized queries to prevent SQL Injection. • Regularly update and patch the WAF to recognize and block advanced bypass techniques. • Conduct thorough input validation and sanitization. • Perform regular security assessments and code reviews.	
References	
• OWASP SQL Injection Prevention Cheat Sheet • OWASP Testing Guide: SQL Injection Testing	

Proof of Concept

This demonstrates that the WAF was bypassed, and the application is susceptible to SQL Injection attacks using

https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php?id=1&param=UNI&param2=ON%20SEL&param3=ECT%201,2,3–

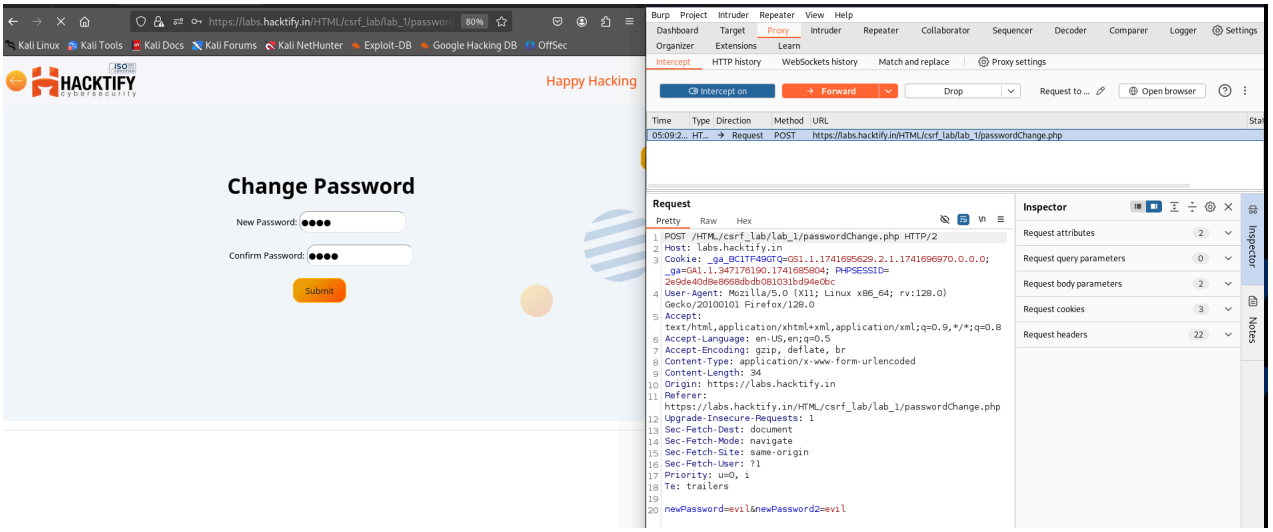


5. Cross-Site Request Forgery

5.1. Eassyy CSRF

Reference	Risk Rating
Eassyy CSRF	**Low**
Tools Used	
Web Browser Tools (Inspect and Source Code), Burp Suite Community Edition, CSRF PoC Generator, Local Web Server (e.g., Python's HTTP server)	
Vulnerability Description	
The application lacks proper CSRF protections on the password change functionality. This oversight allows an attacker to craft malicious requests that, when executed by an authenticated user, can change the user's password without their consent.	
How It Was Discovered	
• I accessed the web application at https://…/login.php and registered two accounts: one representing the victim and another as the attacker. I then logged into the attacker's account and accessed the password change functionality. • **I** enabled "Intercept" in Burp Suite and captured the HTTP request corresponding to the password change action. I then copied the captured request for analysis. • Next, I generated a CSRF PoC using the CSRF PoC Generator available at https://hacktify.in/csrf/. • After which, I disabled "Intercept" after capturing the necessary data. • Saved the generated CSRF PoC HTML to a local file and hosted it using a local web server. • **I** Logged into the victim's account and accessed the hosted CSRF PoC file via the local server. It was noted that the victim's password was changed without their interaction, confirming the CSRF vulnerability.	
Vulnerable URLs	
• https://labs.hacktify.in/HTML/csrf_lab/lab_1/lab_1.php • https://labs.hacktify.in/HTML/csrf_lab/lab_1/login.php • https://labs.hacktify.in/HTML/csrf_lab/lab_1/passwordChange.php	
Consequences of not Fixing the Issue	
• Unauthorized account modifications leading to potential account takeovers. • Loss of user trust and potential legal ramifications. • Compromise of sensitive user data.	
Suggested Countermeasures	
• Implement anti-CSRF tokens for state-changing requests to ensure that such actions are performed intentionally by authenticated users. • Enforce same-site cookie attributes to prevent cookies from being sent with cross-site requests. • Educate users about the risks of CSRF and encourage safe browsing habits.	
References	
• OWASP CSRF Prevention Cheat Sheet • https://portswigger.net/web-security/csrf • https://developer.mozilla.org/en-US/docs/Glossary/CSRF	

Proof of Concept



5.2. Always Validate Tokens

Reference	Risk Rating
Always Validate Tokens	**Medium**

Tools Used

Web Browser Tools (Inspect and Source Code), Burp Suite Community Edition, CSRF PoC Generator, Local Web Server (e.g., Python's HTTP server)

Vulnerability Description

The application implements a CSRF token mechanism; however, it fails to properly validate the authenticity of these tokens. By manipulating the token value, an attacker can forge requests that are accepted by the server, leading to unauthorized actions such as changing a user's password without their consent.

How It Was Discovered

- I accessed the web application at https://…/lab_2/login.php and registered two accounts: one representing the attacker and another as the victim.
- I then utilized Burp Suite where I enabled "Intercept" in Burp Suite.
- Next, I logged in as the victim and captured the login request, noting the CSRF token in the request parameters. This prompted me to log in as the attacker and intercepted the password change request.
- I then generated a CSRF PoC using the attacker's request but substituted the attacker's CSRF token with the victim's token, altering a character to test token validation.
- I saved the crafted CSRF PoC HTML to a local file and hosted it using a local web server.
- I logged into the victim's account again and accessed the hosted CSRF PoC file via the local server. It was noted that the victim's password was changed without their interaction, indicating improper validation of CSRF tokens.

Vulnerable URLs

https://labs.hacktify.in/HTML/csrf_lab/lab_2/login.php
https://labs.hacktify.in/HTML/csrf_lab/lab_2/login.php
https://labs.hacktify.in/HTML/csrf_lab/lab_2/passwordChange.php

Consequences of not Fixing the Issue

- Unauthorized Account Modifications: Attackers can change user account settings, including passwords, leading to potential account takeovers.

- Privilege Escalation: If an administrative account is compromised, attackers could gain elevated privileges, allowing them to alter critical application configurations or access sensitive data.
- Data Integrity Compromise: Unauthorized actions can lead to data being altered or deleted, affecting the integrity and reliability of the application's data.

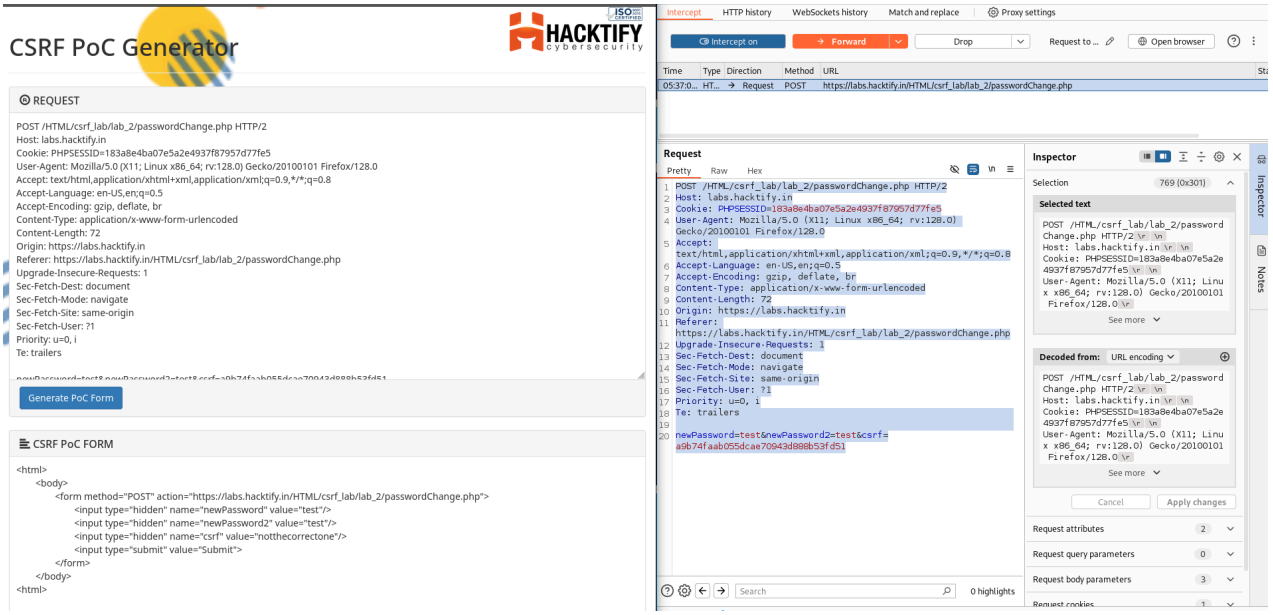
Suggested Countermeasures

- Ensure that CSRF tokens are unique per user session and are validated strictly on the server side for every state-changing request. Tokens should be tied to the user's session and should be unpredictable with high entropy.
- Generate a new CSRF token for each request to minimize the risk of token reuse attacks. This approach shortens the window of opportunity for an attacker to exploit a stolen token.
- Bind CSRF tokens to specific user sessions and ensure that any discrepancy results in request rejection. This prevents attackers from using tokens issued to other sessions.

References

- OWASP CSRF Prevention Cheat Sheet
- https://portswigger.net/web-security/csrf
- https://developer.mozilla.org/en-US/docs/Glossary/CSRF

Proof of Concept



5.3. I hate when someone uses my tokens!

Reference	Risk Rating
I hate when someone uses my tokens!	**Medium**

Tools Used

Web Browser Tools (Inspect and Source Code), Burp Suite Community Edition, CSRF PoC Generator, Local Web Server (e.g., Python's HTTP server)

Vulnerability Description

The application is vulnerable to Cross-Site Request Forgery (CSRF) due to improper validation of CSRF tokens. Specifically, the server accepts CSRF tokens from one user session to perform state-changing

actions in another user's session. This flaw allows an attacker to reuse a CSRF token from their own session to execute unauthorized actions on behalf of another user, leading to potential account compromise and unauthorized data modifications.

How It Was Discovered

- I accessed the web application at https://…/lab_4/login.php and registered two accounts: one for the attacker and another for the victim.
- I then captured the Victim's CSRF Token: Logged in as the victim and used Burp Suite to intercept and capture the CSRF token associated with the victim's session.
- I logged in as the attacker and intercepted the password change request using Burp Suite. Sent this request to the Repeater tool within Burp Suite for further manipulation.
- I then crafted the CSRF PoC: Replaced the attacker's CSRF token in the intercepted password change request with the victim's CSRF token. Generated a CSRF PoC HTML file using this modified request.
- I then saved the crafted CSRF PoC HTML file and hosted it on a local web server.
- Lastly, I logged in as the victim and accessed the hosted CSRF PoC file via the local server. Observed that the victim's password was changed without their consent, confirming the CSRF vulnerability.

Vulnerable URLs

https://labs.hacktify.in/HTML/csrf_lab/lab_4/lab_4.php
https://labs.hacktify.in/HTML/csrf_lab/lab_4/login.php
https://labs.hacktify.in/HTML/csrf_lab/lab_4/passwordChange.php

Consequences of not Fixing the Issue

- Attackers can change user account settings, including passwords, leading to potential account takeovers.
- If an administrative account is compromised, attackers could gain elevated privileges, allowing them to alter critical application configurations or access sensitive data.
- Unauthorized actions can lead to data being altered or deleted, affecting the integrity and reliability of the application's data.

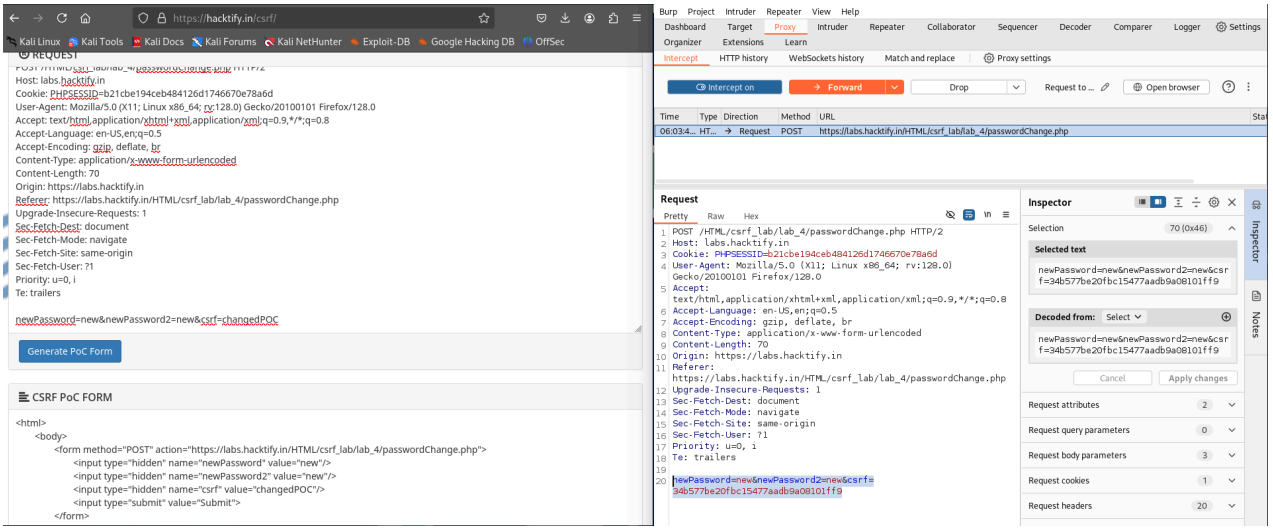
Suggested Countermeasures

- Ensure that CSRF tokens are unique per user session and are validated strictly on the server side for every state-changing request. Tokens should be tied to the user's session and should be unpredictable with high entropy.
- Generate a new CSRF token for each sensitive action to minimize the risk of token reuse attacks. This approach shortens the window of opportunity for an attacker to exploit a stolen token.
- Bind CSRF tokens to specific user sessions and ensure that any discrepancy results in request rejection. This prevents attackers from using tokens issued to other sessions.

References

- OWASP CSRF Prevention Cheat Sheet
- PortSwigger Web Security: Bypassing CSRF Token Validation
- Invicti: How To Prevent CSRF Attacks by Using Anti-CSRF Tokens

Proof of Concept



5.4. GET Me or POST ME

Reference	Risk Rating
GET Me or POST ME	**Low**

Tools Used

Web Browser Tools (Inspect and Source Code), Burp Suite Community Edition, CSRF PoC Generator, Local Web Server (e.g., Python's HTTP server)

Vulnerability Description

Vulnerability Description The application is vulnerable to Cross-Site Request Forgery (CSRF) attacks, allowing unauthorized commands to be transmitted from a user that the web application trusts. Specifically, the password change functionality can be exploited by an attacker to change a victim's password without their consent.

How It Was Discovered

- Accessed the web application at https://labs.hacktify.in/HTML/csrf_lab/lab_6/login.php and logged in as the attacker.
- Navigated to the change password section.
  Enabled "Intercept" in Burp Suite and captured the password change request.
- Sent the captured request to Burp Suite's Repeater, observing parameters such as newPassword, newPassword2, and csrf.
- Disabled "Intercept" in Burp Suite.
- Utilized the captured request details to generate a CSRF PoC using the POST method.
- Saved the generated PoC to a local HTML file and hosted it using a local web server.
- Logged into the victim's account and accessed the hosted PoC file via the local server.
- Observed that the victim's password was changed without their interaction, confirming the CSRF vulnerability.

Vulnerable URLs

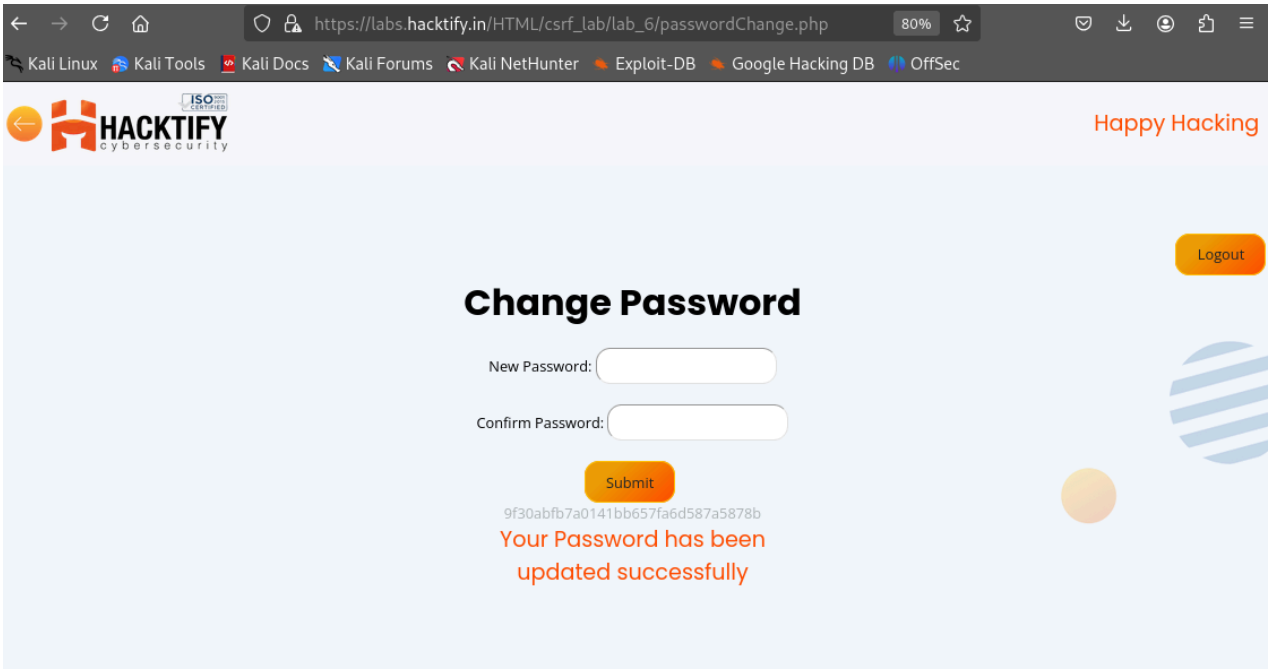
https://labs.hacktify.in/HTML/csrf_lab/lab_6/login.php
https://labs.hacktify.in/HTML/csrf_lab/lab_6/login.php
https://labs.hacktify.in/HTML/csrf_lab/lab_6/passwordChange.php

Consequences of not Fixing the Issue

• Unauthorized password changes can lead to account takeovers, compromising user data and privacy. • Attackers gaining control of accounts can perform malicious activities, damaging the application's reputation. • Potential legal and compliance issues arising from unauthorized access to user accounts.
Suggested Countermeasures
• Ensure that CSRF tokens are unique per user session and are validated strictly on the server side for every state-changing request. • Implement the `SameSite` attribute for cookies to prevent them from being sent with cross-site requests. • Require re-authentication or additional verification steps for sensitive actions like password changes to confirm the legitimacy of the request.
References
• OWASP CSRF Prevention Cheat Sheet • https://portswigger.net/web-security/csrf • https://developer.mozilla.org/en-US/docs/Glossary/CSRF

Proof of Concept

An attacker can craft a malicious HTML page that, when accessed by an authenticated victim, sends a POST request to change the victim's password without their knowledge or consent.



5.5. XSS the saviour

Reference	Risk Rating
XSS the saviour	**High**
Tools Used	
Web Browser Tools (Inspect and Source Code), Burp Suite Community Edition, CSRF PoC Generator, Local Web Server (e.g., Python's HTTP server)	

Vulnerability Description
The web application is vulnerable to Cross-Site Request Forgery (CSRF), allowing attackers to perform unauthorized actions on behalf of authenticated users. Specifically, the application fails to validate the origin of state-changing requests, making it susceptible to CSRF attacks.
How It Was Discovered
1. Accessed the web application at https://…/lab_7/login.php and logged in as the victim. 2. Navigated to the input name field within the application. 3. Enabled "Intercept" in Burp Suite and captured the request when submitting the name change. 4. Inserted the XSS payload <script>alert(document.cookie)</script> into the name parameter of the intercepted request. 5. Forwarded the modified request to the server. 6. Observed that the browser executed the script, displaying the session ID and CSRF token, confirming the vulnerability.
Vulnerable URLs
https://labs.hacktify.in/HTML/csrf_lab/lab_7/login.php https://labs.hacktify.in/HTML/csrf_lab/lab_7/login.php https://labs.hacktify.in/HTML/csrf_lab/lab_7/passwordChange.php
Consequences of not Fixing the Issue
• Attackers can perform unauthorized actions on behalf of users, leading to potential account compromise. • Exposure of sensitive information, such as session IDs and CSRF tokens, can facilitate further attacks. • Compromised user trust and potential legal implications due to data breaches.
Suggested Countermeasures
• Ensure that each form submission includes a unique, unpredictable CSRF token that the server validates. Set the SameSite attribute for cookies to Strict or Lax to prevent them from being sent in cross-origin requests. • Require re-authentication or additional verification steps for sensitive actions to confirm the legitimacy of the request.
References
• OWASP CSRF Prevention Cheat Sheet • https://portswigger.net/web-security/csrf • https://developer.mozilla.org/en-US/docs/Glossary/CSRF

Proof of Concept

By injecting the XSS payload <script>alert(document.cookie)</script> into the name parameter, the application executed the script, revealing the session ID and CSRF token.

5.6. rm -rf token

Reference	Risk Rating
rm -rf token	**High**
Tools Used	

Web Browser Tools (Inspect and Source Code), Burp Suite Community Edition, CSRF PoC Generator, Local Web Server (e.g., Python's HTTP server)
Vulnerability Description
The application is vulnerable to Cross-Site Request Forgery (CSRF) due to the absence of proper CSRF token validation. This flaw allows attackers to forge malicious requests that can perform unauthorized actions on behalf of authenticated users, such as changing passwords without the user's consent.
How It Was Discovered
1. Accessed the web application at https://…/lab_8/login.php and created two accounts: one representing the victim and another as the attacker. 2. Logged in as the attacker and navigated to the functionality that allows password changes. 3. Enabled "Intercept" in Burp Suite and captured the password change request. 4. Sent the captured request to Burp Suite's Repeater and turned off the intercept to allow normal traffic flow. 5. Removed the CSRF token parameter from the request to test if the server validates its presence. 6. Generated a CSRF PoC using the modified request information. Saved the CSRF PoC to a local HTML file and hosted it using a local web server. 7. Logged in as the victim and accessed the hosted PoC file. 8. Observed that the victim's password was changed successfully without their interaction, indicating the application's failure to validate CSRF tokens properly.
Vulnerable URLs
https://labs.hacktify.in/HTML/csrf_lab/lab_8/login.php https://labs.hacktify.in/HTML/csrf_lab/lab_8/login.php https://labs.hacktify.in/HTML/csrf_lab/lab_8/passwordChange.php
Consequences of not Fixing the Issue
• Attackers can perform unauthorized actions on behalf of users, such as changing account settings or passwords, leading to potential account takeovers. • Compromise of user accounts can result in unauthorized access to sensitive personal or financial information. • Exploitation of administrative accounts could allow attackers to alter critical application configurations or access sensitive organizational data.
Suggested Countermeasures
• Implement anti-CSRF tokens: Ensure that each form submission includes a unique, unpredictable token that the server validates. • Enforce token validation: The server should reject any state-changing requests that do not contain the correct CSRF token. • Set the SameSite attribute for cookies to Strict or Lax to prevent them from being sent with cross-site requests. • Educate users about the risks of CSRF attacks and advise them to avoid clicking on suspicious links or visiting untrusted websites while authenticated.
References
• OWASP CSRF Prevention Cheat Sheet • https://portswigger.net/web-security/csrf • https://developer.mozilla.org/en-US/docs/Glossary/CSRF

Proof of Concept

By removing the CSRF token from the password change request and executing the modified request, the application accepted the request and changed the user's password without proper validation, demonstrating the CSRF vulnerability.

6. Cross-Origin Resource Sharing (CORS)

6.1. CORS With Arbitrary Origin

Reference	Risk Rating
CORS With Arbitrary Origin	**Low**
Tools Used	
Web Browser Tools (Inspect and Source Code), Burp Suite, cURL	
Vulnerability Description	
The application accepts and processes requests from any origin without proper validation. This misconfiguration allows attackers to craft malicious websites that can interact with the vulnerable application on behalf of authenticated users, potentially leading to unauthorized data access or actions.	
How It Was Discovered	
1. Accessed the web application at https://…/lab_1/login.php and logged in using the provided credentials. 2. Launched Burp Suite and enabled the intercept feature. 3. Performed an action within the application to capture a request. 4. In the intercepted request, added the Origin header with the value https://attacker.com. 5. Forwarded the modified request to the server () and observed the response. 6. Noted that the server included the header Access-Control-Allow-Origin: https://attacker.com in its response, indicating that it trusts the specified origin. 7. Utilized cURL to replicate the request: curl -i -H "Origin: https://attacker.com" \ -H "Cookie: PHPSESSID=<session_id>" \ -X GET https://labs.hacktify.in/HTML/cors_lab/lab_1/cors_1.php 8. Confirmed that the response contained sensitive user data, demonstrating that the server processes requests from arbitrary origins.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/cors_lab/lab_1/cors_1.php	
Consequences of not Fixing the Issue	
• Unauthorized access to sensitive user information by malicious websites. • portswigger.net • Potential execution of unauthorized actions on behalf of authenticated users. • Increased risk of phishing attacks leveraging the trust relationship between the user and the vulnerable application.	
Suggested Countermeasures	
• Implement a whitelist of trusted domains that are permitted to access resources, ensuring that only legitimate origins can interact with the application.portswigger.net • Avoid using wildcard characters (*) in the Access-Control-Allow-Origin header, as this allows any domain to access resources. • Set the Access-Control-Allow-Credentials header to true only when absolutely necessary, and ensure that credentials are only shared with trusted origins. • Regularly review and update CORS policies to adapt to the evolving security landscape and maintain robust protection against cross-origin attacks.	
References	
• Cross-Origin Resource Sharing (CORS) - MDN Web Docs	

- Cross-Origin Resource Sharing (CORS) - PortSwigger
- Exploiting CORS Misconfigurations - FreeCodeCamp

Proof of Concept

By adding an Origin header with a malicious domain and observing the server's response, it was demonstrated that the application accepts and processes requests from arbitrary origins, leading to potential unauthorized data access.

6.2. CORS with Null origin

Reference	Risk Rating
CORS with Null origin	**Low**
Tools Used	
Web Browser Tools (Inspect and Source Code), Burp Suite, cURL	
Vulnerability Description	
The application improperly trusts requests with a null origin, allowing unauthorized cross-origin interactions. This misconfiguration enables attackers to craft malicious websites that can interact with the vulnerable application on behalf of authenticated users, potentially leading to unauthorized data access or actions.	
How It Was Discovered	
1. I access the web application at https://…/lab_2/login.php and log in using the provided credentials. 2. I launch Burp Suite and enable the intercept feature. 3. I perform an action within the application to capture a request. 4. In the intercepted request, I add the Origin header with the value https://attacker.com. 5. I forward the modified request to the server and observe the response. 6. I note that the server includes the header Access-Control-Allow-Origin: null in its response, indicating that it trusts the null origin. 7. I utilize cURL to replicate the request: curl -i -H "Origin: null" \ -H "Cookie: PHPSESSID=<session_id>" \ -X GET https://labs.hacktify.in/HTML/cors_lab/lab_2/cors_2.php 8. I confirm that the response contains sensitive user data, demonstrating that the server processes requests with a null origin	
Vulnerable URLs	
https://labs.hacktify.in/HTML/cors_lab/lab_2/cors_2.php	
Consequences of not Fixing the Issue	
• Unauthorized access to sensitive user information by malicious websites. • portswigger.net • Potential execution of unauthorized actions on behalf of authenticated users. • Increased risk of phishing attacks leveraging the trust relationship between the user and the vulnerable application.	
Suggested Countermeasures	
• Implement a whitelist of trusted domains that are permitted to access resources, ensuring that only legitimate origins can interact with the application.portswigger.net	

- Avoid using wildcard characters (*) in the Access-Control-Allow-Origin header, as this allows any domain to access resources.
- Set the Access-Control-Allow-Credentials header to true only when absolutely necessary, and ensure that credentials are only shared with trusted origins.
- Regularly review and update CORS policies to adapt to the evolving security landscape and maintain robust protection against cross-origin attacks.

References

- Cross-Origin Resource Sharing (CORS) - MDN Web Docs
- Cross-Origin Resource Sharing (CORS) - PortSwigger
- Exploiting CORS Misconfigurations - FreeCodeCamp

Proof of Concept

By adding an Origin header with the value null and observing the server's response, it is demonstrated that the application accepts and processes requests from a null origin, leading to potential unauthorized data access.

6.3. CORS with prefix match

Reference	Risk Rating
CORS with prefix match	**Medium**
Tools Used	
Web Browser Tools (Inspect and Source Code), Burp Suite, cURL	
Vulnerability Description	
The application accepts and trusts any origin that has a prefix matching its own domain. For example, it considers hacktify.in.attacker.com as a trusted origin because it starts with hacktify.in. This misconfiguration allows attackers to craft malicious subdomains that can interact with the application's resources, potentially leading to unauthorized data access or actions on behalf of authenticated users.	
How It Was Discovered	
• I navigated to the web application's login page and authenticated using the provided credentials. • I enabled the intercept feature in Burp Suite and captured a request to the application. • I modified the Origin header to hacktify.in.attacker.com and forwarded the request. • I observed that the server responded successfully and included the Access-Control-Allow-Origin: hacktify.in.attacker.com header, indicating that it trusts origins with prefixes matching its domain.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/cors_lab/lab_3/login.php	
Consequences of not Fixing the Issue	
• Unauthorized access to sensitive user information by malicious websites. • portswigger.net • Potential execution of unauthorized actions on behalf of authenticated users. • Increased risk of phishing attacks leveraging the trust relationship between the user and the vulnerable application.	
Suggested Countermeasures	
• Implement a whitelist of trusted domains that are permitted to access resources, ensuring that only legitimate origins can interact with the application.portswigger.net • Avoid using wildcard characters (*) in the Access-Control-Allow-Origin header, as this allows any domain to access resources. • Set the Access-Control-Allow-Credentials header to true only when absolutely necessary, and ensure that credentials are only shared with trusted origins. • Regularly review and update CORS policies to adapt to the evolving security landscape and maintain robust protection against cross-origin attacks.	
References	
• Cross-Origin Resource Sharing (CORS) - MDN Web Docs • Cross-Origin Resource Sharing (CORS) - PortSwigger • Exploiting CORS Misconfigurations - FreeCodeCamp	

Proof of Concept

By setting the Origin header to hacktify.in.attacker.com, the server grants access, demonstrating that it trusts any origin with a prefix matching its domain.

6.4. CORS with suffix match

Reference	Risk Rating
CORS with suffix match	**Medium**
Tools Used	
Web Browser Tools (Inspect and Source Code), Burp Suite, cURL	
Vulnerability Description	
The application is configured to trust any origin that ends with a specific suffix, such as "hacktify.in". This misconfiguration allows an attacker to exploit the CORS policy by using a malicious domain that ends with the trusted suffix (e.g., "evilhacktify.in"). Consequently, unauthorized websites can interact with the application's resources, leading to potential data breaches or unauthorized actions on behalf of authenticated users.	
How It Was Discovered	
• I navigated to the web application's login page at https://…/lab_4/login.php and authenticated using the provided credentials. • I enabled the intercept feature in Burp Suite and captured a request to the application. • I added the Origin header with the value evilhacktify.in to the request. • I forwarded the modified request to the repeater and turned off the interceptor. • I observed that the server responded successfully, indicating that it accepted the request from the evilhacktify.in domain. This demonstrated that the server's CORS policy trusts origins with the specified suffix, regardless of the preceding domain.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/cors_lab/lab_4/login.php	
Consequences of not Fixing the Issue	
• Unauthorized access to sensitive user information by malicious websites. • portswigger.net • Potential execution of unauthorized actions on behalf of authenticated users. • Increased risk of phishing attacks leveraging the trust relationship between the user and the vulnerable application.	
Suggested Countermeasures	
• Implement a whitelist of trusted domains that are permitted to access resources, ensuring that only legitimate origins can interact with the application.portswigger.net • Avoid using wildcard characters (*) in the Access-Control-Allow-Origin header, as this allows any domain to access resources. • Set the Access-Control-Allow-Credentials header to true only when absolutely necessary, and ensure that credentials are only shared with trusted origins. • Regularly review and update CORS policies to adapt to the evolving security landscape and maintain robust protection against cross-origin attacks.	
References	
• Cross-Origin Resource Sharing (CORS) - MDN Web Docs • Cross-Origin Resource Sharing (CORS) - PortSwigger • Exploiting CORS Misconfigurations - FreeCodeCamp	

Proof of Concept

By setting the Origin header to evilhacktify.in, the server grants access, demonstrating that it trusts any origin with a suffix matching its domain.

6.5. CORS with Escape dot

Reference	Risk Rating
CORS with Escape dot	**High**
Tools Used	
Web Browser Tools (Inspect and Source Code), Burp Suite	
Vulnerability Description	
The application is vulnerable due to improper handling of the 'Origin' header in CORS requests. Specifically, the server fails to correctly validate the 'Origin' header when dots are not properly escaped in its regular expression pattern. This misconfiguration allows an attacker to craft malicious domains that can bypass the CORS policy, leading to unauthorized access to sensitive user data.	
How It Was Discovered	
• I navigated to the web application at https://../lab_5/login.php and logged in using the provided credentials. • I launched Burp Suite and enabled the intercept feature to capture HTTP requests. • Upon capturing the login request, I sent it to the Repeater module within Burp Suite for further analysis. • In the Repeater, I added the 'Origin' header with the value 'wwwhacktify.in' to simulate a request from a subdomain that exploits the dot misconfiguration. • I forwarded the modified request and observed the server's response. • The server responded successfully, including the 'Access-Control-Allow-Origin' header set to 'wwwhacktify.in', indicating that the server incorrectly trusts the crafted origin.	
Vulnerable URLs	
• https://labs.hacktify.in/HTML/cors_lab/lab_5/login.php	
Consequences of not Fixing the Issue	
• Unauthorized access to sensitive user information by malicious websites. • portswigger.net • Potential execution of unauthorized actions on behalf of authenticated users. • Increased risk of phishing attacks leveraging the trust relationship between the user and the vulnerable application.	
Suggested Countermeasures	
• Ensure that regular expressions used in CORS policy configurations correctly escape special characters, such as dots, to prevent unintended matches. • Implement a strict whitelist of trusted origins and avoid using patterns that could match unintended domains. • Regularly audit and test CORS configurations to identify and rectify misconfigurations that could be exploited.	
References	
• Cross-Origin Resource Sharing (CORS) - MDN Web Docs • Cross-Origin Resource Sharing (CORS) - PortSwigger • Exploiting CORS Misconfigurations - FreeCodeCamp	

Proof of Concept

By adding the 'Origin' header with the value 'wwwhacktify.in' in a crafted request, the server grants access, demonstrating the CORS misconfiguration vulnerability.

6.6. CORS with Substring match

Reference	Risk Rating
CORS with Substring match	**High**
Tools Used	
Web Browser Tools (Inspect and Source Code), Burp Suite, cURL	
Vulnerability Description	
The application is configured to allow cross-origin requests from domains that contain a specific substring, such as hacktify. This approach is insecure because it permits any domain with the substring (e.g., hacktify.co) to access resources, potentially exposing sensitive information to unauthorized parties.	
How It Was Discovered	
• I navigated to https://…/lab_6/login.php and logged in using the provided credentials. • I launched Burp Suite and enabled the intercept feature to capture HTTP requests. • Upon capturing the login request, I added the Origin header with the value hacktify.co. • I forwarded the modified request to the server and observed the response. • The server responded successfully, indicating that it accepted the cross-origin request from hacktify.co.	
Vulnerable URLs	
• https://labs.hacktify.in/HTML/cors_lab/lab_6/login.php	
Consequences of not Fixing the Issue	
• Unauthorized access to sensitive user information by malicious websites. • portswigger.net • Potential execution of unauthorized actions on behalf of authenticated users. • Increased risk of phishing attacks leveraging the trust relationship between the user and the vulnerable application.	
Suggested Countermeasures	
• Implement strict origin checks by specifying exact, trusted domains in the Access-Control-Allow-Origin header rather than using substring matches. • Regularly audit CORS configurations to ensure that only legitimate domains have access to resources. • Educate developers about the risks associated with permissive CORS policies and the importance of precise origin validation.	
References	
• Cross-Origin Resource Sharing (CORS) - MDN Web Docs • Cross-Origin Resource Sharing (CORS) - PortSwigger • Exploiting CORS Misconfigurations - FreeCodeCamp	

Proof of Concept

By adding the Origin header with the value hacktify.co to the HTTP request, the server granted access, demonstrating the vulnerability.

6.7. CORS with Arbitrary Subdomain

Reference	Risk Rating
CORS with Arbitrary Subdomain	**High**
Tools Used	
Web Browser Tools (Inspect and Source Code), Burp Suite, cURL	
Vulnerability Description	
The application is configured to trust arbitrary subdomains, allowing any subdomain under hacktify.in to access resources. This misconfiguration permits an attacker-controlled subdomain to interact with the application as a trusted origin, potentially leading to unauthorized access to sensitive data.	
How It Was Discovered	
• I accessed the web application at https://…/lab_7/login.php and logged in using the provided credentials. • I enabled the intercept feature in Burp Suite and captured the login request. • I added the 'Origin' header with the value somesubdomain.hacktify.in to the request. • I forwarded the modified request to the server and observed the response. • The response included the 'Access-Control-Allow-Origin' header set to somesubdomain.hacktify.in, indicating that the server trusts this arbitrary subdomain.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/cors_lab/lab_7/login.php	
Consequences of not Fixing the Issue	
• Unauthorized access to sensitive user information by malicious websites. • portswigger.net • Potential execution of unauthorized actions on behalf of authenticated users. • Increased risk of phishing attacks leveraging the trust relationship between the user and the vulnerable application.	
Suggested Countermeasures	
• Implement a whitelist of trusted domains that are permitted to access resources, ensuring that only legitimate origins can interact with the application.portswigger.net • Avoid using wildcard characters (*) in the Access-Control-Allow-Origin header, as this allows any domain to access resources. • Set the Access-Control-Allow-Credentials header to true only when absolutely necessary, and ensure that credentials are only shared with trusted origins. • Regularly review and update CORS policies to adapt to the evolving security landscape and maintain robust protection against cross-origin attacks.	
References	
• Cross-Origin Resource Sharing (CORS) - MDN Web Docs • Cross-Origin Resource Sharing (CORS) - PortSwigger • Exploiting CORS Misconfigurations - FreeCodeCamp	

Proof of Concept

By adding an 'Origin' header with an arbitrary subdomain (e.g., somesubdomain.hacktify.in) to the request, the server includes this origin in the 'Access-Control-Allow-Origin' response header, demonstrating that it trusts arbitrary subdomains.

WEEK 4 (CTFs)

7.1 Help Me

Category: Web

Description: This challenge focuses on web security and authentication bypass techniques. This involves navigating through a login system, retrieving encoded credentials, and finding a hidden flag within the web application.

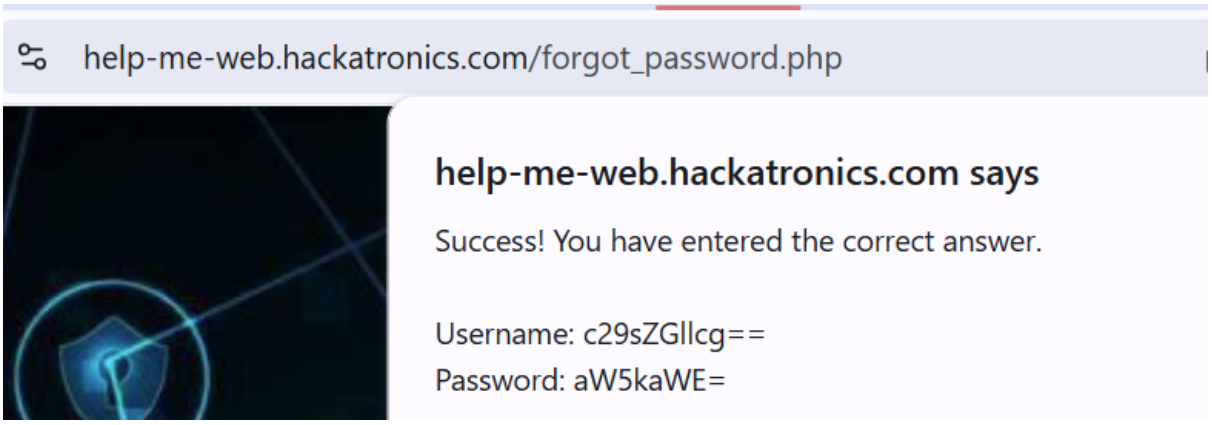
Challenge Overview: The "Help Me" challenge requires testing authentication mechanisms, decoding Base64-encoded credentials, analyzing web page sources, and decrypting encoded text to retrieve the final flag.

Steps for Finding the Flag:

1. I accessed the login page at **https://help-me-web.hackatronics.com/** and attempted to brute-force the credentials using default login credentials (admin:admin). The attempt was unsuccessful, displaying the error message: "*Invalid username or password. Please try again.*"
2. I then attempted SQL Injection using the payload `admin" or "1"="1` but received the same error message, confirming that authentication bypass via SQLi was not possible.
3. Next, I navigated to the **Forgot Password** page at **https://help-me-web.hackatronics.com/forgot_password.php**. The page required answering a security question: "*On which date is Navy Day celebrated in India?*"
4. I searched online and found the answer: **4th December**. Upon entering this answer, a pop-up appeared displaying encrypted credentials:
    - **Username:** `c29sZGllcg==`
    - **Password:** `aW5kaWE=`
5. I tested these credentials directly on the login page but received an invalid login error.
6. I then decoded the credentials using **Base64 decoding** via **https://www.base64decode.org/.** The decoded credentials were: **Username:** soldier | **Password:** india
7. I successfully logged in using these credentials at **https://help-me-web.hackatronics.com/home.php**.
8. Upon successful login, I analyzed the page source code by right-clicking and selecting *View Page Source*. Scrolling through the page source, I found a comment on **line 85** containing an encoded flag: <!-- SYNT:{1_nz_ce0h4_0s_h} -->

9. The flag was encoded using **ROT13 encryption**. I used the ROT13 decoder at https://cryptii.com/pipes/rot13-decoder to decode the text.

Flag: `{1_am_pr0u4_0f_u}`



7.2: Lock Web

Category: Web

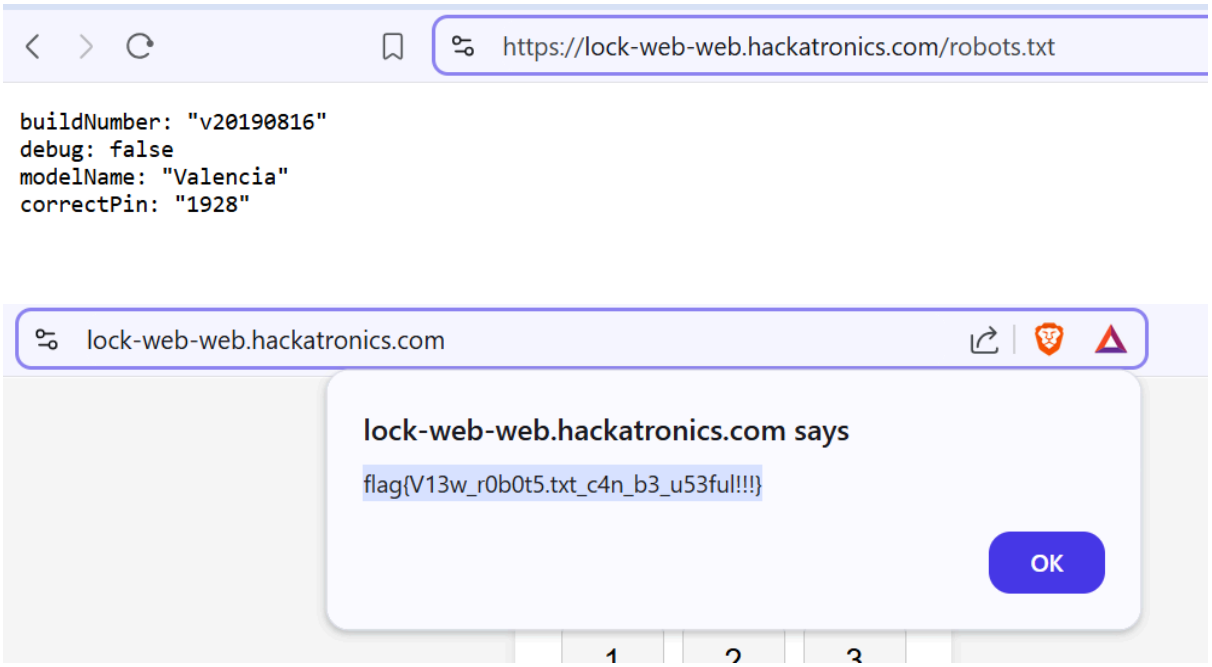
Description: This challenge involves discovering sensitive information through content discovery techniques. The goal is to bypass a PIN lock system using data extracted from the website's `robots.txt` file.

Challenge Overview: The target website, https://lock-web-web.hackatronics.com, presents a PIN Lock System that requires a 4-digit PIN for access. By leveraging robots.txt, we extracted the correct PIN and used it to unlock the flag.

Steps for Finding the Flag:

1. Since the hint suggested *"Think like a robot beep bop"*, I checked the `robots.txt` file by navigating to https://lock-web-web.hackatronics.com/robots.txt, The file contained the following
   buildNumber: "v20190816"
   debug: false
   modelName: "Valencia"
   correctPin: "1928"
2. I then returned to https://lock-web-web.hackatronics.com and entered 1928. A popup message appeared, displaying the flag.

Flag: flag{V13w_r0b0t5.txt_c4n_b3_u53ful!!!}



7.3: The World

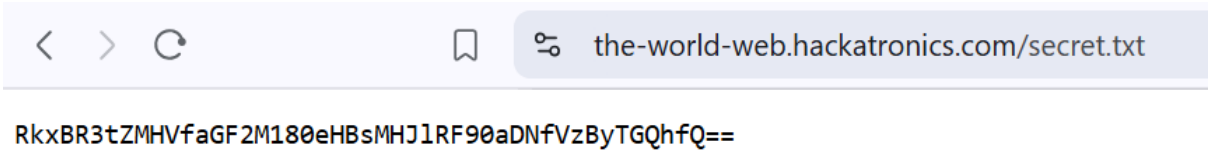
Category: Web

Description: This challenge involves performing **directory fuzzing** to uncover hidden files and extract sensitive information. The goal is to explore unlisted paths and retrieve the flag.

Challenge Overview: The target website, **https://the-world-web.hackatronics.com**, deeper exploration using **directory fuzzing** leads to the discovery of a hidden text file containing an encoded flag.

Steps for Finding the Flag:

1. The hint suggested checking **every file**, implying hidden resources. I launched **DirBuster** and a wordlist with the file extension set to `.txt` into the URL: **https://the-world-web.hackatronics.com**.
2. Among the discovered files, **`secret.txt`** stood out due to its different size. Manually navigating to **https://the-world-web.hackatronics.com/secret.txt** revealed a **Base64-encoded string** RkxBR3tZMHVfaGF2M180eHBsMHJlRF90aDNfVzByTGQhfQ==



3. I decoded it using **https://www.base64decode.org/** which decodes to

Flag: FLAG{Y0u_hav3_4xpl0reD_th3_W0rLd!}

7.4: Mail Mystery

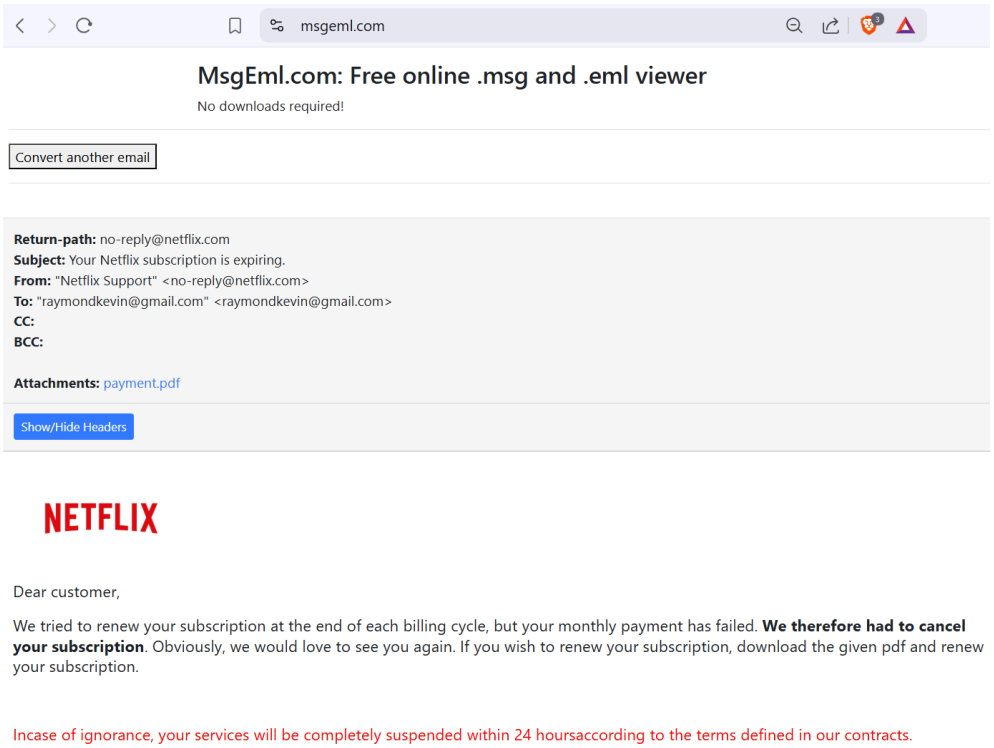
Category: Network Forensics

Description: This challenge involves analyzing an email file recovered from a system crash. The goal is to extract relevant information from the file and uncover any hidden details regarding the email's authenticity and purpose.

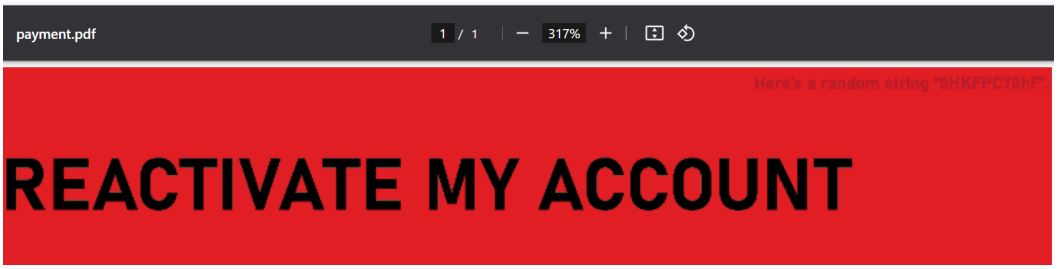
Challenge Overview: The "Mail Mystery" challenge requires identifying the email file type, extracting its contents, analyzing metadata, and investigating any hidden elements to retrieve the flag.

Steps for Finding the Flag:

1. The provided file was named `Mail_Mystery`, but the file type was unspecified. To determine the file type, on a bash terminal, I used the command file Mail_Mystery. The output indicated that the file contained ASCII text related to SMTP mail, suggesting it was a `.eml` file.
2. To confirm this, I used the `cat` command: cat Mail_Mystery. This displayed the email's raw content, confirming it was indeed an `.eml` file.
3. To properly view the email, I uploaded it to **https://msgeml.com/**. The email details were extracted.



4. The email urged the recipient to download `payment.pdf` to renew the subscription. I retrieved and analyzed the attachment using https://exif.tools/upload.php The PDF contained a button labeled "*Reactivate my account*", which linked to Netflix's official signup page: **https://www.netflix.com/signup/planform**. Running `exiftool` on the PDF revealed additional metadata, including a **Pastebin link**: The output contained: **https://pastebin.com/fh4mEK5P**.
5. The Pastebin page was password-protected. Searching through the PDF's content, I found a small hidden text near the button: **"Here's a random string '8HKFPC70hF'."**



6. Using ‘8HKFPC70hF’ as the password for the Pastebin link, I successfully accessed the hidden content.

flag{DFIR_G3N1US}.

7.5: Corrupted

Category: Network Forensics

Description: A file named chall.png-1740909420074-618633075.png was provide. Attempting to open this file results in an error indicating that the file format is not supported. The objective is to investigate and correct any issues with the file to retrieve the hidden flag.

Challenge Overview: *This is too EasyPeasyy!*

Steps for Finding the Flag:

1. I tried opening `chall.png-1740909420074-618633075.png` but encountered an error stating the file format is not supported.
2. Therefore, I then opened the file in a hex editor to inspect its header. I noticed that the header bytes did not match the standard PNG file signature (`89 50 4E 47 0D 0A 1A 0A`).
3. I modified the first eight bytes to match the correct PNG signature.
4. I then saved the changes and reopened the file using an image viewer. The image displayed contained the flag.

Flag: flag{m3ss3d_h3ad3r$}

7.6: Shadow Web

Category: Network Forensics

Description: This challenge involves analyzing network traffic to extract hidden **Form Data** from captured packets. The goal is to retrieve and decode scattered secrets to reveal the flag.

Challenge Overview: The challenge provided a **pcapng** file containing captured network traffic. Using **tshark** and **base64 decoding**, we extracted sensitive data embedded in HTTP POST requests and successfully retrieved the flag.

Steps for Finding the Flag:

1. I used **tshark** to filter and extract data from HTTP POST requests:
   tshark -r capture.pcapng -T fields -e http.file_data -Y "http.request.method == POST"
2. This command revealed **WebKit boundaries** containing the hidden content. I then used a Python script to extract the relevant content from the **WebKit boundaries**. The extracted content was Base64-encoded:

ZmxhZ3ttdWx0MXBsM3A0cnRzYzBuZnVzM3N9

3. I used the following command to decode the Base64 string:

echo ZmxhZ3ttdWx0MXBsM3A0cnRzYzBuZnVzM3N9 | base64 -d

This returned the final flag

FLAG: flag{mult1pl3p4rtsc0nfus3s}

.7: It's easy, y'know

Category: Reverse Engineering

Description: This challenge involves analyzing a binary executable file to extract an embedded password using reverse engineering techniques.

Challenge Overview: The `crackme2` file is an ELF executable that contains a hidden password. The goal is to reverse-engineer the file to extract the password and retrieve the flag.

Steps for Finding the Flag:

1. I identified the file type using the command: file crackme2. The output confirmed that it was an **ELF** executable. Running the executable displayed the message: "Good luck, read the source." This hinted that the password was embedded in the binary.
2. I loaded the file into **Ghidra** for further analysis. The **`my_secure_test`** function contained a parameter named `param_1`, which stored a text string. The string "**1337_pwd**" was identified in `param_1`, indicating that it was the password.
3. Using this password, I retrieved the flag.

FLAG: flag{1337_pwd}

7.8: Lost in the Past

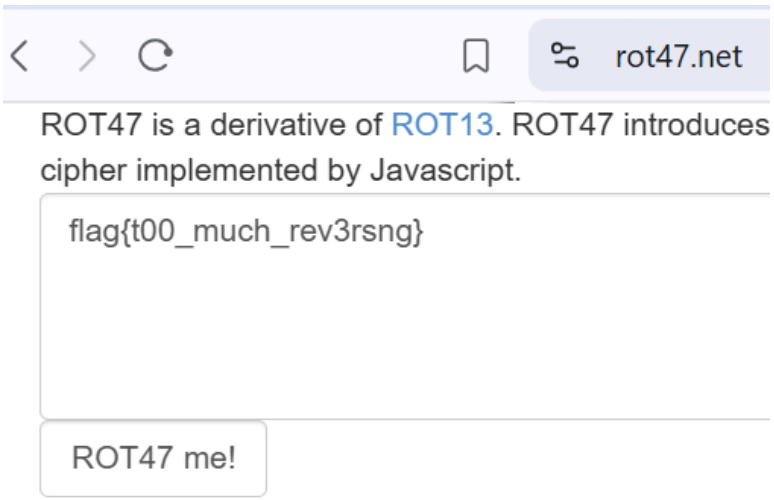
Category: Reverse Engineering

Description:

Challenge Overview: I enjoyed making small projects when I was at a young age! I used to love hiding random funny texts in my projects that no one else could understand but myself. Coincidentally, I found a project file of something I made at that time. But it's been so long, I can't find that text. Can you help me find it?

Steps to Find the Flag:

1. **I** rename the `CTF.aia-1740910120281-691025702.aia` file to `CTF.zip` and extract its contents.
2. **I n**avigated to find the Scrum.bky file then open it, the encoded flag is in textbox 1 "7=28LE__0>F490C6GbCD?8N"
3. I then used https://rot47.net/and decode the st7=28LE__0>F490C6GbCD?8N sting.



flag{t00_much_rev3rs1ng}

7.9: Decrypt Quest

Category: Reverse Engineering

Description: A zip file which contains a text file purportedly holding encrypted secrets amidst a plethora of irrelevant data. The goal is to sift through the noise to find and decrypt the concealed information.

Challenge Overview: One day, one of Samarth's imaginary friends, Arjun, mysteriously hands him a text file claiming it holds encrypted secret data impossible to decode! Arjun dangles a $1,000,000 reward if Samarth manages to extract the information. However, Arjun enjoys mischief and attempts to trick Samarth by flooding the file with loads of irrelevant data. Would you assist Samarth in unlocking this top-secret information? He pledges to split the reward with you if successful!!

Steps to Find the Flag:

1. **I** unzipped the ZIP File `Answer.zip-1740910433987-931520235.zip` to obtain `encrypted_data.txt`.
2. I then opened `encrypted_data.txt` and looked for patterns or anomalies that stand out from the irrelevant data. I also scanned the file for sequences that resemble encoded data, such as base64 strings, hexadecimal sequences, or other encoding schemes.
3. Finally, I used an online decoder to decode the original message. The decrypted text revealed the flag.

Flag: flag{hjwilj111970djs}

7.10: Raccoon

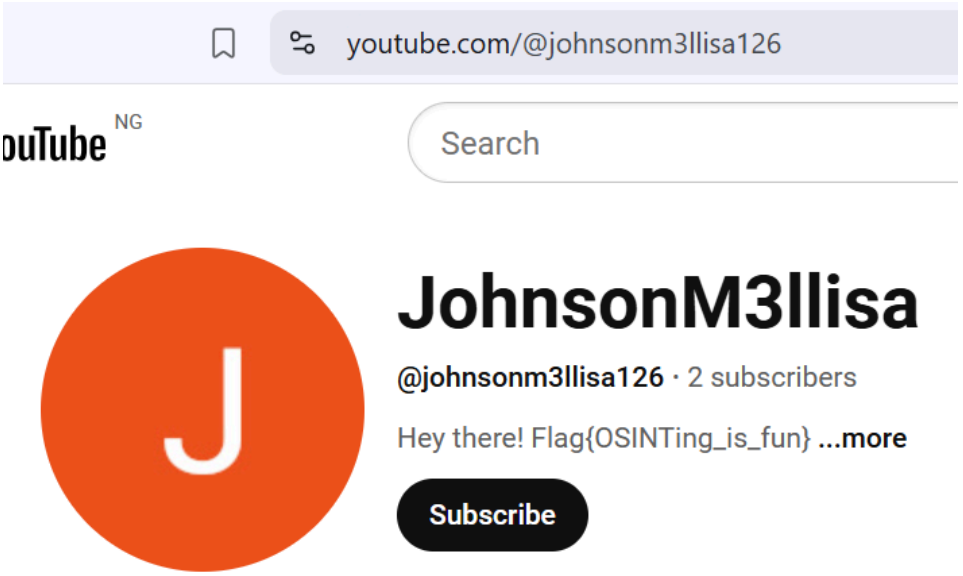
Category: OSINT

Description: This challenge requires finding the online presence of a pet raccoon named "racckoonn" through open-source intelligence (OSINT) techniques.

Challenge Overview: The given information hints that the raccoon's name, racckoonn, may be used on social media platforms. The objective is to track the digital footprint associated with the pet raccoon.

Steps for Finding the Flag:

1. Since this was an OSINT challenge, I searched for the username racckoonn on Google.
2. I found an Instagram account matching the name: https://www.instagram.com/racckoonn. The Instagram bio mentioned that the raccoon's owner was @johnsonm3llisa126, and she had a YouTube channel.
3. I then searched for the YouTube username: https://www.youtube.com/@johnsonm3llisa126
4. The About section of the YouTube channel contained the flag.



FLAG: flag{OSINTing_is_fun}

7.11: Time Machine

Category: OSINT

Description:

Challenge Overview: Mr. TrojanHunt has power to travel time. He is hiding some extremely confidential file from the government. Can you help NIA to get secrets of TrojanHunt?

Steps to Find the Flag:

1. **I** conducted an online search for "Mr. TrojanHunt" to gather background information. I found Mr. TrojanHunt's Internet Archive's Wayback Machine on https://archive.org/details/secret_202103 to view archived versions of Mr. TrojanHunt's website. Here, I discovered a text file. I then accessed the file and found the flag within its contents.

Flag: flag{Tr0j3nHunt_t1m3_tr4v3l}

7.12: Snapshot Whispers

Category: OSINT

Description: The task requires investigating the origin of an image to determine if it's genuinely taken by the friend or sourced from elsewhere. This involves using OSINT techniques to trace the image back to its original creator.

Challenge Overview: Skeptical about my friend's recent travel claims, I requested a photo, and to my surprise, they sent me an image that seemed more like a generic internet find than a personal capture. Help me find out the name of the photographer who clicked the photo.

Steps to Find the Flag:

1. I uploaded the Image.png-1740911096032-151980933.png to Google Images and search for concert hall in Sydney opera house google reviews where I saw a similar image.

2. I download the image and use tools like exiftool to extract metadata using the exiftool Image.png
3. I searched for fields such as "Artist" or "Copyright," which might contain the photographer's name and Alas, I found it.

flag{Jeffrey_Seidman}

7.13: Time Traveller

Category: Crypto

Description: This challenge involves decrypting an encrypted message secured through two stages: a permutation-based encryption and a time-seeded XOR operation. The challenge requires reversing both stages to recover the original flag.

Challenge Overview: The "Time Traveller" challenge encrypts a message using an 8-element permutation followed by XOR-based obfuscation with a time-based seed. The task involves reversing these encryption stages by extracting the time, undoing XOR, and brute-forcing the key permutations.

Steps for Finding the Flag:

1. The encrypted flag file `flag.enc` and the encryption script `chall.py` were provided for analysis.
2. The encryption involved two stages:
    - **Stage 1:** Encryption using a random permutation of `[0,8)`.
    - **Stage 2:** Appending the current timestamp (18 bytes) and applying an XOR operation with `0x42`.
3. The first step was extracting the last **18 bytes** of `flag.enc`, which contained the encrypted timestamp. Each byte was XORed with `0x42` to recover the original timestamp.
4. Using this timestamp as a seed, a pseudo-random number generator was initialized to regenerate the encryption key. The first **part of the encrypted message** (excluding the last 18 bytes) was decrypted using the generated key by reversing the XOR operation.
5. The decrypted message was still permuted due to Stage 1 encryption. To recover the original text, a brute-force approach was used by iterating through all **8! (40320) permutations** of `[0,1,2,3,4,5,6,7]`.
6. For each permutation, the `dec` function was applied **42 times** (as per the encryption pattern in the script) to attempt message reconstruction. The valid plaintext flag was identified when it matched the format `flag{...}`.

FLAG: {T1m3_15_pr3C10u5_s0_Enj0y_ur_L1F5!!!}

7.14: Wh@t7he####

Category: Crypto

Description: The challenge presents an enigmatic prompt: "Change my mind!" with a flag format specified as `flag{!@#$%^&*()_+}`.

Challenge Overview: The task involves deciphering the cryptic message to uncover the flag. The unusual flag format suggests that the flag itself might consist of special characters

Steps to Find the Flag:

1. I used the file command in a terminal to determine the file's type: file chall.file-1740911737469-67832877.file
2. I then inspected the file contents by using a hex editor to view its hexadecimal representation: xxd chall.file-1740911737469-67832877.file
3. Open the provided file (chall.file-1740911737469-67832877.file) using a text editor. I observed that the content consists of characters like +, -, [,], >, <, ., and ,, which are characteristic of Brainfuck or its derivatives.
4. Navigate to an online ReverseFuck decoder, such as https://www.dcode.fr/reversefuck-language and Paste the content of the file into the decoder and execute it to obtain the output.
5. The decoder outputs the flag

Flag {R3Vers3ddd_70_g3t_m3}

7.15: Success Recipe

Category: Crypto

Description: A chef friend sent a recipe written in an unfamiliar language or code. The task is to decipher the recipe to uncover the hidden flag.

Challenge Overview: My friend who is a Chef sent me this recipe (recipe.txt-1740911941466-466385354.txt) but i can't understand it He likes to write in weird languages Can you help me?

Steps to Find the Flag:

1. **I o**pened the `recipe.txt-1740911941466-466385354.txt` file using a text editor. The content resembles a cooking recipe, indicating it is written in the Chef programming language.

2. I then used an Online Chef Interpreter esolangpark.vercel.app, where I pasted the content of the recipe file into the interpreter's code editor.
3. After resolving any syntax issues, run the code in the interpreter. The interpreter will output a sequence of characters, typically consisting of symbols like `+`, `-`, `[`, `]`, `>`, `<`, `.`, and `,`.
4. I then navigated to an online Brainfuck interpreter https://www.dcode.fr/reversefuck-language where I pasted the Brainfuck code into the interpreter and execute it to obtain the decoded message.
5. The decoded message reveals the flag

`Flag{y0u_40+_s3rv3d!}`