Penetration Testing Report

Full Name: RUSHIKESH AVIREDDY
Program: HCPT
Date: 16/02/2025

Introduction

This report document hereby describes the proceedings and results of a Black Box security assessment conducted against the **Week {1} Labs**. The report hereby lists the findings and corresponding best practice mitigation actions and recommendations.

1. Objective

The objective of the assessment was to uncover vulnerabilities in the **Week {1} Labs** 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.

2. Scope

This section defines the scope and boundaries of the project.

Application Name	{Lab 1 – HTML Injection}, {Lab 2 – Cross Site Scripting}

3. Summary

Outlined is a Black Box Application Security assessment for the **Week {1} Labs**.

Total number of Sub-labs: {17} Sub-labs

High	Medium	Low
{4}	{3}	{8}

High - **4 Sub-labs with hard difficulty level**

Medium - **3 Sub-labs with Medium difficulty level**

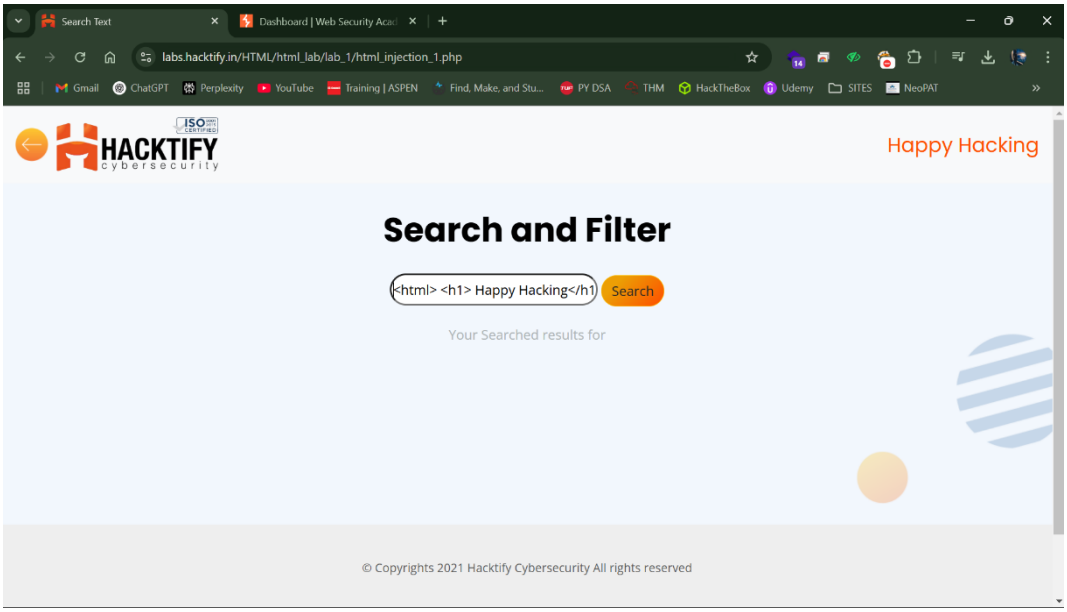
Low - **8 Sub-labs with Easy difficulty level**

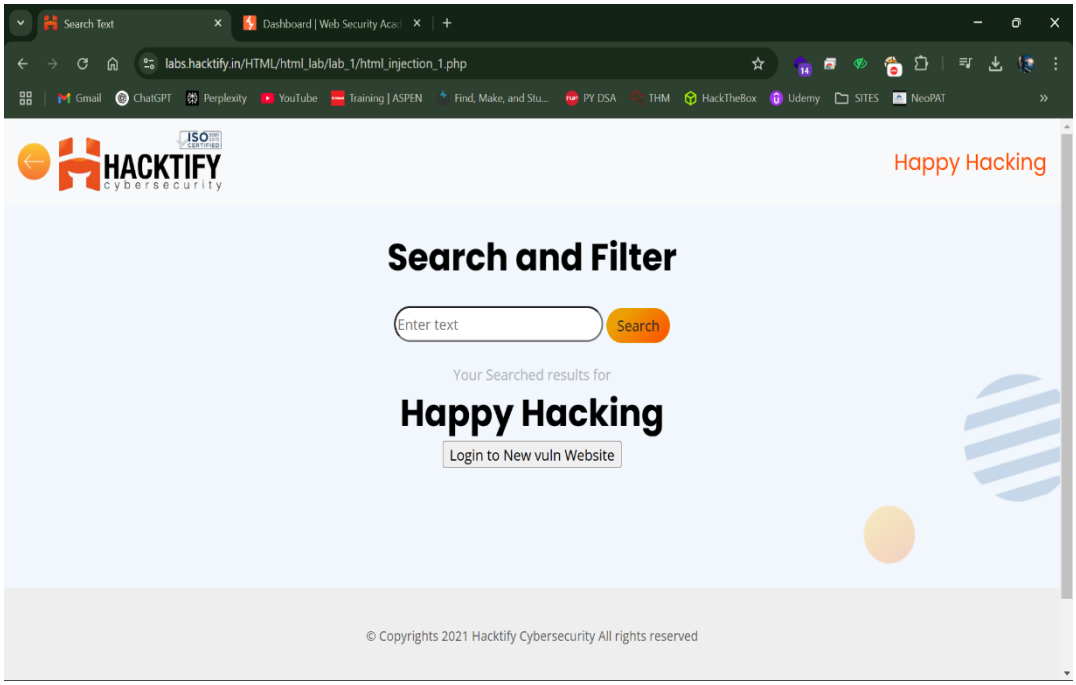
1. HTML Injection

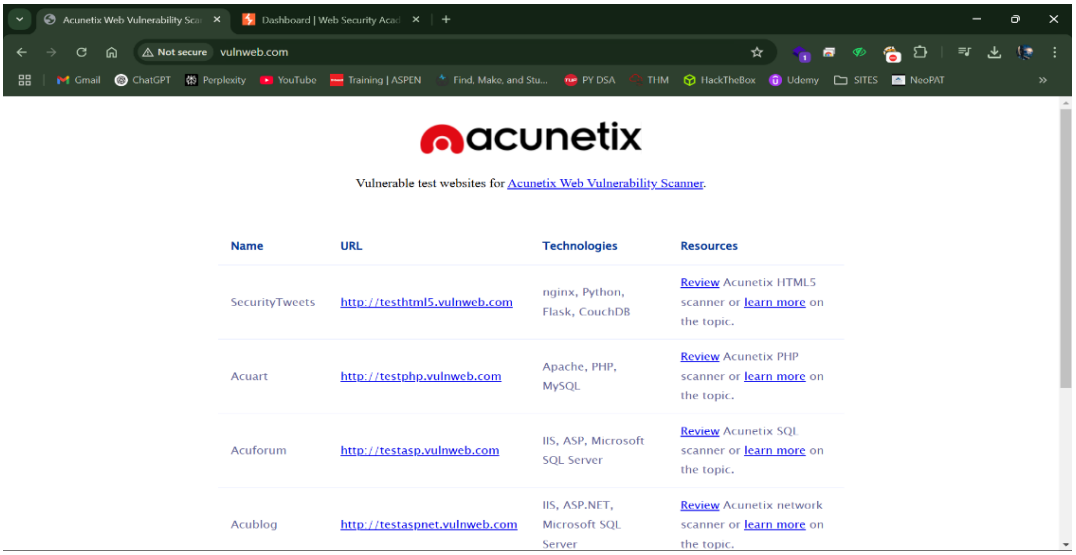
1.1. HTML's are easy!

Reference	Risk Rating
HTML's are easy!	**Low**
Tools Used	
Browser "Inspector" is used to find the vulnerability.	
Vulnerability Description	
HTML Injection is a web security vulnerability that occurs when an attacker injects malicious HTML code into a web application, typically through user input fields that are not properly sanitized. This can lead to content manipulation, defacement, and even phishing attacks.	
How It Was Discovered	
Manual Analysis – Viewing Page Source	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_1/html_injection_1.php	
Consequences of not Fixing the Issue	
IIf the HTML Injection vulnerability is not fixed , then attackers can alter the appearance or content of web pages, damaging the reputation of the website or organization.	
Suggested Countermeasures	
Input Validation:- Validate all user inputs based on expected formats (e.g., alphanumeric, length restrictions).Strip or encode HTML tags from user input before storing or rendering.Use libraries like DOMPurify (for JavaScript) or OWASP Java Encoder (for Java) to sanitize input.	
References	
https://www.acunetix.com/vulnerabilities/web/html-injection/	

Proof of Concept



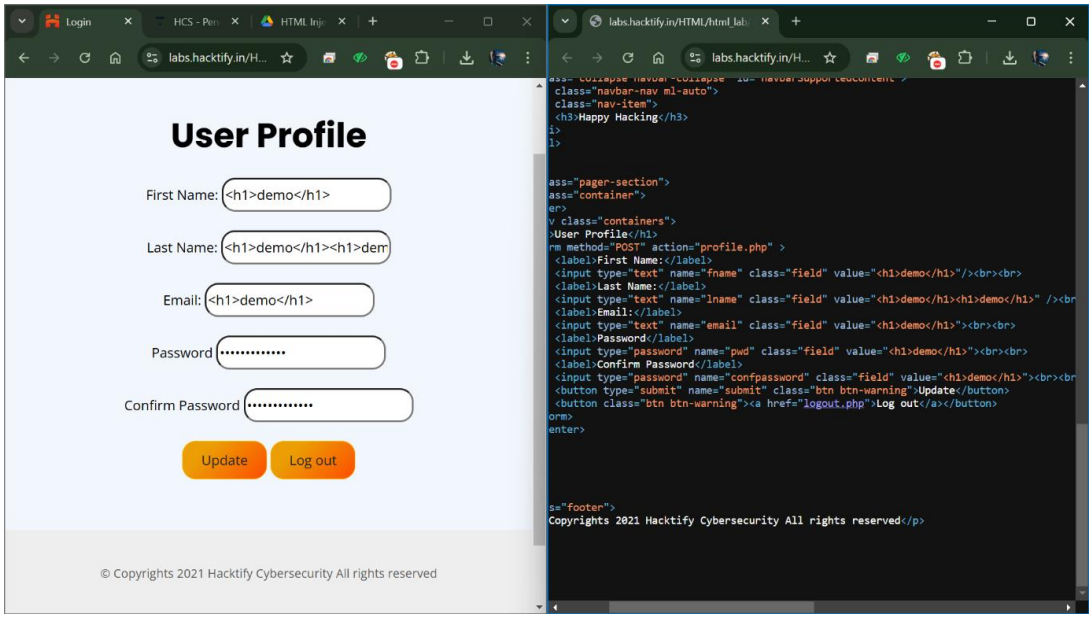

Search Text
Dashboard | Web Security Acad
labs.hacktify.in/HTML/html_lab/lab_1/html_injection_1.php
Gmail
ChatGPT
Perplexity
YouTube
Training | ASPEN
Find, Make, and Stu...
PY DSA
THM
HackTheBox
Udemy
SITES
NeoPAT
HACKTIFY
cybersecurity
Happy Hacking
Search and Filter
Enter text
Search
Your Searched results for
Happy Hacking
Login to New vuln Website
© Copyrights 2021 Hacktify Cybersecurity All rights reserved


Acunetix Web Vulnerability Scan
Dashboard | Web Security Acad
Not secure
vulnweb.com
Gmail
ChatGPT
Perplexity
YouTube
Training | ASPEN
Find, Make, and Stu...
PY DSA
THM
HackTheBox
Udemy
SITES
NeoPAT
acunetix
Vulnerable test websites for Acunetix Web Vulnerability Scanner.
Name
URL
Technologies
Resources
SecurityTweets
http://testhtml5.vulnweb.com
nginx, Python, Flask, CouchDB
Review Acunetix HTML5 scanner or learn more on the topic.
Acuart
http://testphp.vulnweb.com
Apache, PHP, MySQL
Review Acunetix PHP scanner or learn more on the topic.
Acuforum
http://testasp.vulnweb.com
IIS, ASP, Microsoft SQL Server
Review Acunetix SQL scanner or learn more on the topic.
Acublog
http://testaspnet.vulnweb.com
IIS, ASP.NET, Microsoft SQL Server
Review Acunetix network scanner or learn more on the topic.

1.2. Let me Store them!

Reference	Risk Rating
Let me Store them!	**Low**
Tools Used	
Browser “Inspector” is used to find the vulnerability.	
Vulnerability Description	
The application is vulnerable to **HTML Injection**, allowing attackers to inject and render HTML tags due to improper input sanitization, leading to defacement, phishing.	
How It Was Discovered	
Manual Analysis – Inspecting the Page Source Code	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_2/html_injection_2.php	
Consequences of not Fixing the Issue	
Not fixing this issue can lead to webpage defacement, phishing attacks, session hijacking, malware injection compromising user data and trust.	
Suggested Countermeasures	
Preventing HTML Injection requires input validation, output encoding, CSP implementation, restricting HTML in inputs, and regular security testing.	
References	
https://www.acunetix.com/vulnerabilities/web/html-injection https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/03-Testing_for_HTML_Injection.	

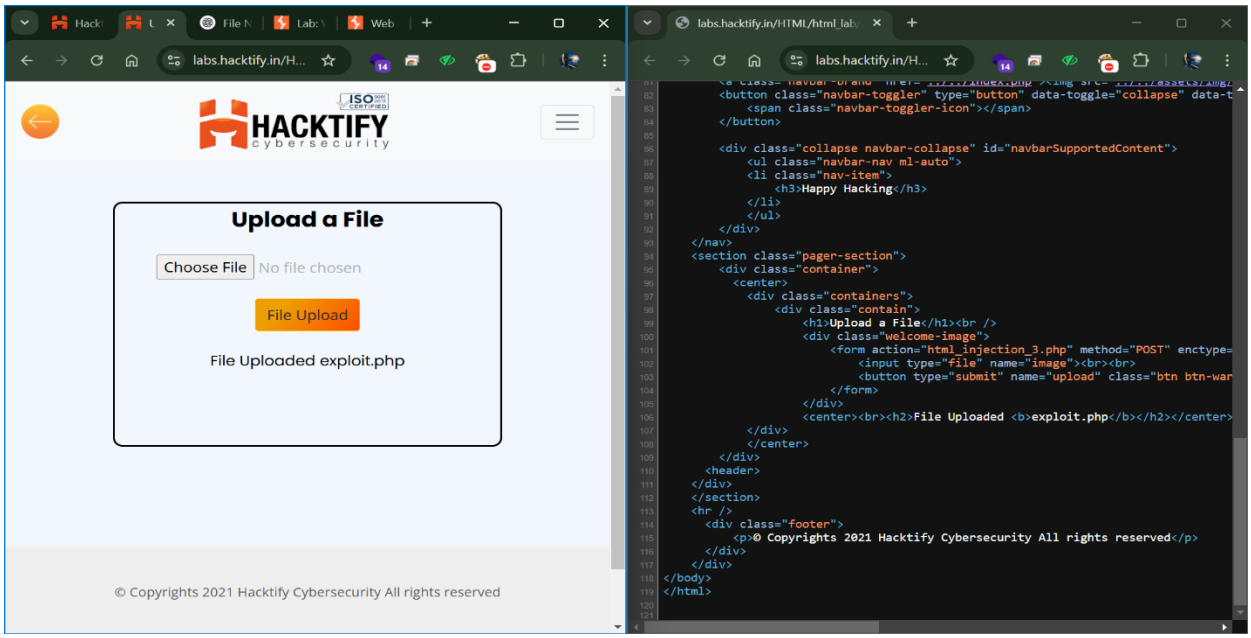
Proof of Concept



1.3 File Names are also Vulnerable!

Reference	Risk Rating
File Names are also Vulnerable	**Low**
Tools Used	
Browser “Inspector” is used to find the vulnerability.	
Vulnerability Description	
Filename vulnerabilities occur when applications fail to sanitize or validate filenames during uploads. Attackers can exploit this for **Log Injection, Path Traversal, or Command Injection**. Unsanitized filenames can execute scripts (XSS), corrupt logs, or manipulate file paths and system commands, leading to **unauthorized access or code execution**. Proper validation and sanitization are essential to prevent these attacks.	
How It Was Discovered	
Manual Analysis – By Inspecting page source code	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_3/html_injection_3.php	
Consequences of not Fixing the Issue	
Ignoring filename vulnerabilities can lead to **Log Injection, Path Traversal, or Command Injection**, risking **data breaches, system compromise, and reputational damage**. Attackers can execute malicious scripts, corrupt logs, manipulate file paths, or run unauthorized commands, potentially leading to **Remote Code Execution (RCE)** and full server control.	
Suggested Countermeasures	
Validate filenames by allowing only alphanumeric characters and limiting length. Sanitize output, use secure storage paths to block path traversal, and validate file types to prevent script execution. Escape special characters in logs and enforce least privilege access to restrict file operations.	
References	
https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload https://portswigger.net/web-security/file-upload	

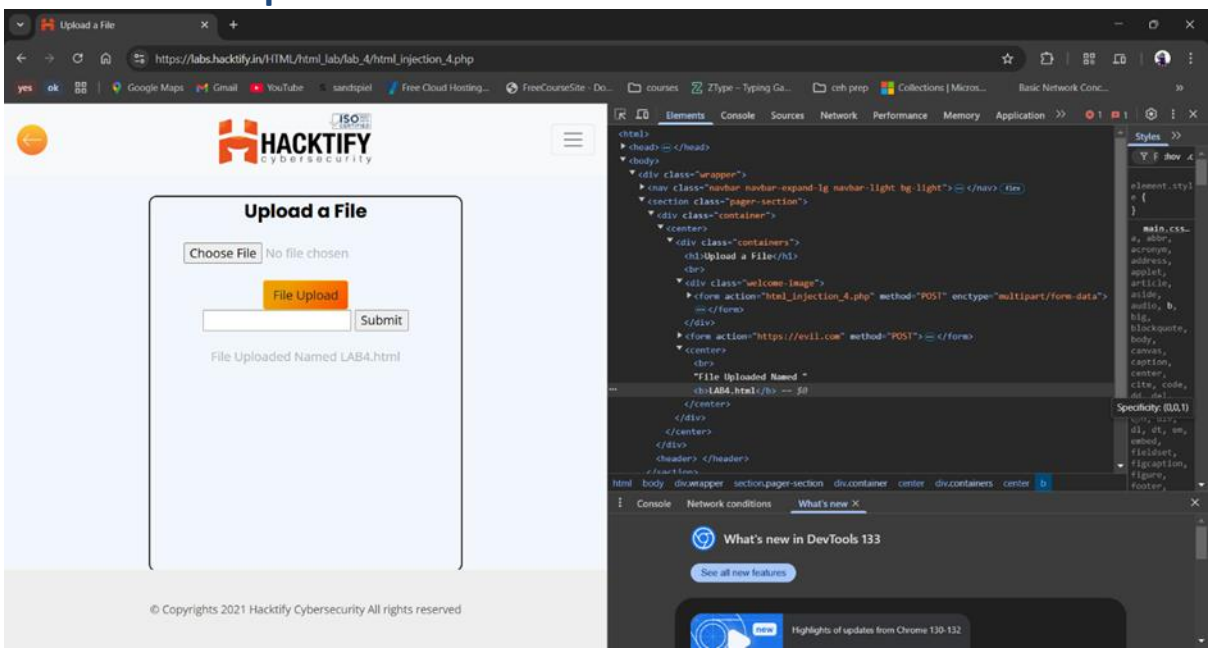
Proof of Concept



1.4 File Content and HTML Injection a perfect parir!

Reference	Risk Rating
File ContentandHTML Injection a perfect pair!	**Low**
Tools Used	
Browser Developer Tools(F12) – To inspect rendered HTML and test injected Content.	
Vulnerability Description	
This vulnerability occurs when an application renders uploaded file contents as HTML instead of plain text. Attackers can inject malicious HTML elements like forms, iframes, or buttons, leading to UI manipulation, phishing, or misleading content display. The issue arises due to lack of proper sanitization and output encoding when processing file uploads.	
How It Was Discovered	
Manual Analysis by uploading HTML file containing form elements	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_4/html_injection_4.php	
Consequences of not Fixing the Issue	
Attackers can inject **fake forms, misleading content, or alter the UI**, leading to **phishing attacks, data theft, or defacement**. This can harm user trust and expose sensitive information.	
Suggested Countermeasures	
To mitigate this issue, sanitize uploaded file content to prevent HTML rendering and apply output encoding to escape special characters. Implement a Content Security Policy (CSP) to restrict unwanted content execution.	
References	
https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload	

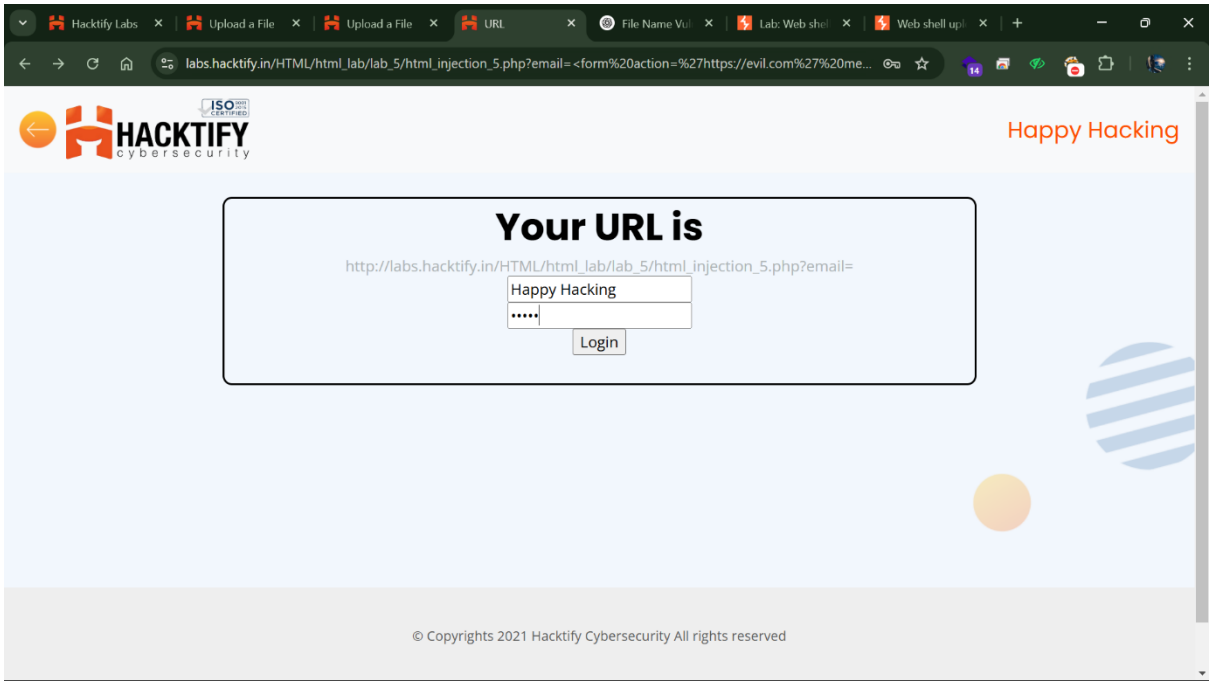
Proof of Concept

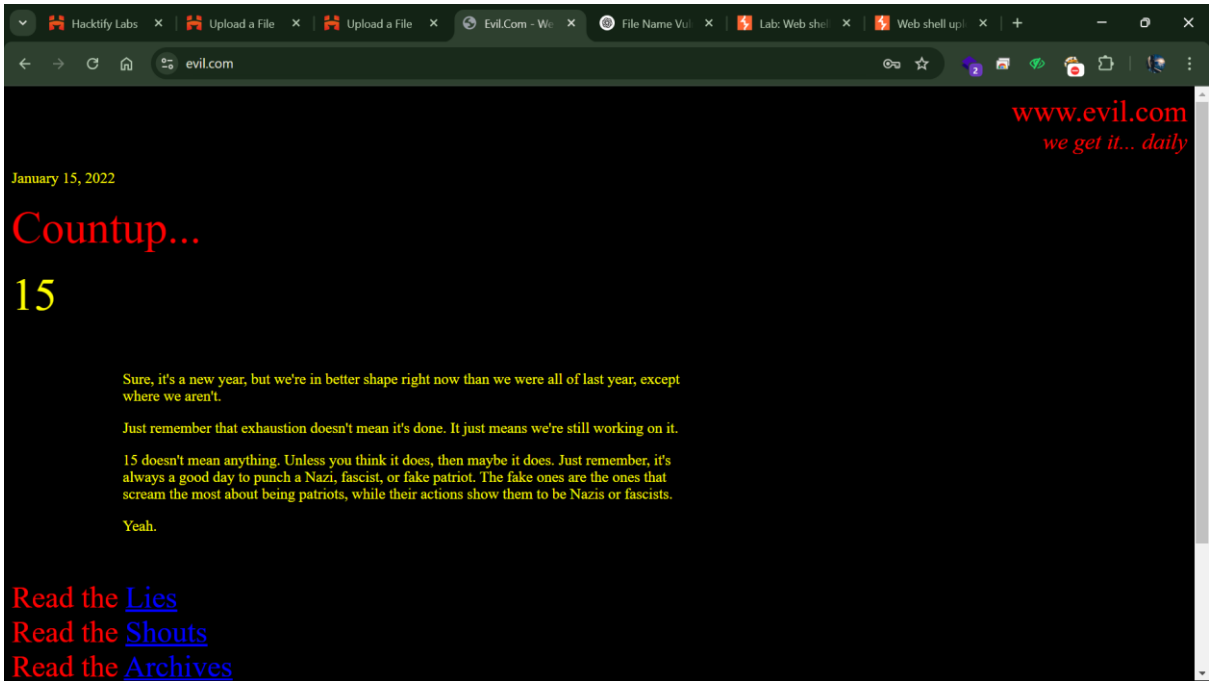


1.5 Injecting HTML using URL

Reference	Risk Rating
Injecting HTML using URL	**Medium**
Tools Used	
Browser Developer Tools(F12) – To inspect rendered HTML and test injected Content.	
Vulnerability Description	
Injecting HTML using a URL happens when user input in the URL is not properly sanitized, allowing attackers to insert HTML elements into the webpage. This can lead to content manipulation, fake login forms, phishing attacks, or misleading users. Proper input validation and output encoding can prevent this issue.	
How It Was Discovered	
Manual Analysis by injecting HTML elements into URL	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_5/html_injection_5.php	
Consequences of not Fixing the Issue	
If not fixed, attackers can manipulate content, create fake login forms for phishing, and mislead users, leading to data theft and reputational damage.	
Suggested Countermeasures	
Implement input validation to reject HTML tags, use output encoding to prevent rendering injected content, and apply Content Security Policy (CSP) to restrict unauthorized scripts and elements.	
References	
https://owasp.org/www-community/attacks/HTML_Injection **https://portswigger.net/web-security/html-injection**	

Proof of Concept

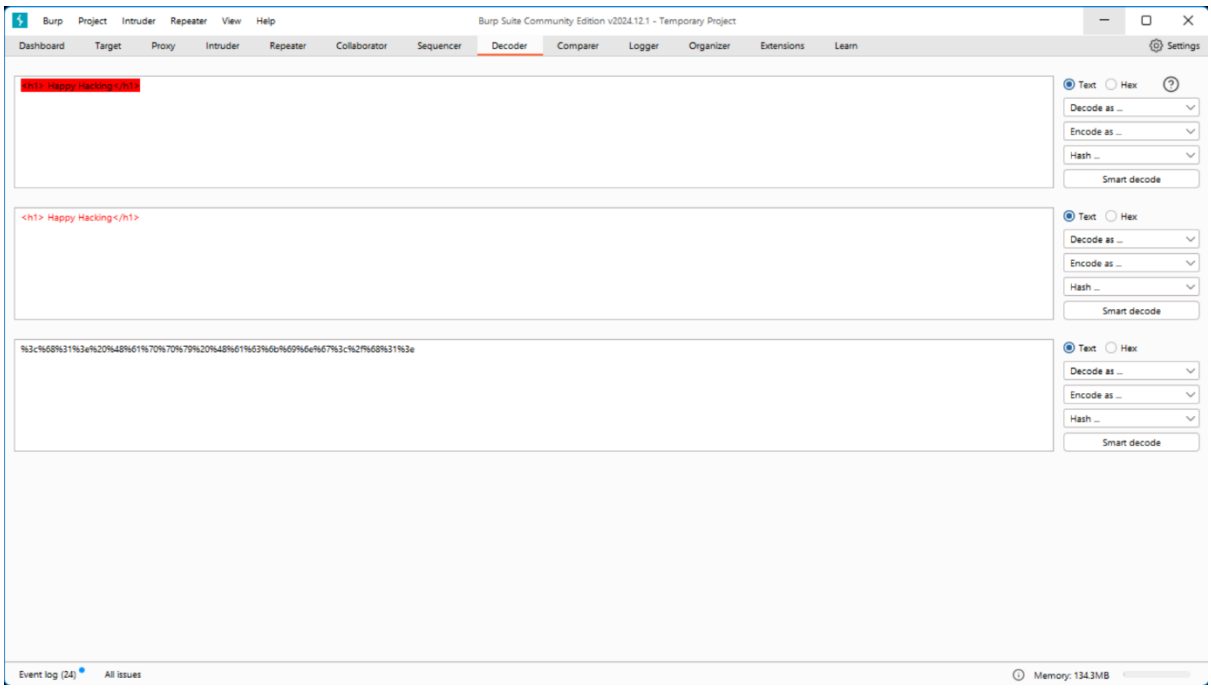


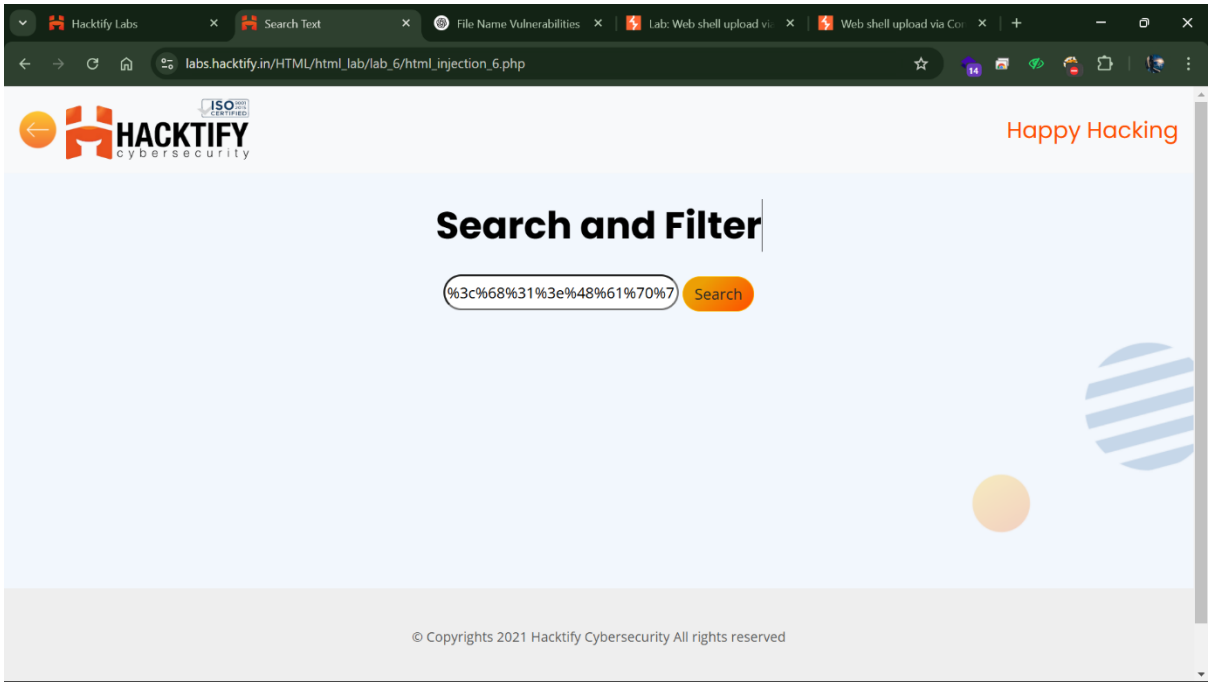


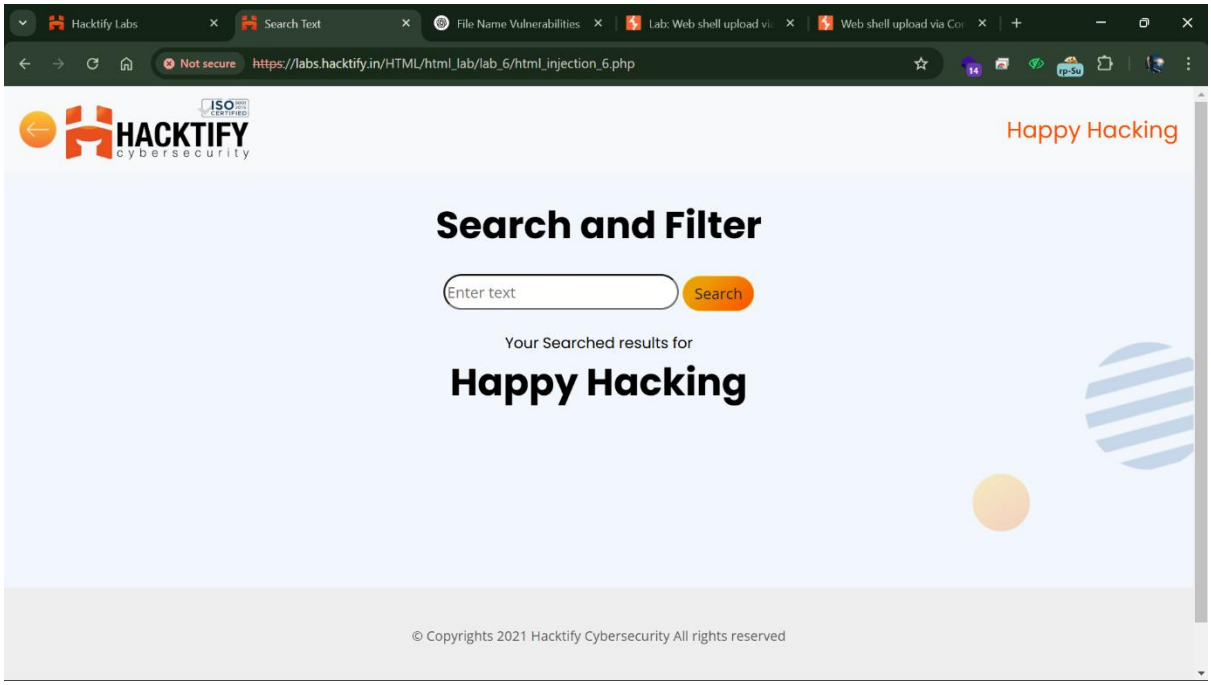
1.6 Encode It!

Reference	Risk Rating
Encode it!	High
Tools Used	
BurpSuite – Repeater and Decoder	
Vulnerability Description	
The vulnerability you exploited is HTML Injection, which occurs when a web application improperly handles user input and renders it as actual HTML. In this case, you URL-encoded an HTML tag (Happy Hacking), and the server decoded and rendered it instead of treating it as plain text. This happens due to lack of proper input sanitization and output encoding, allowing attackers to inject and modify webpage content. If JavaScript execution was possible, it could escalate to Cross-Site Scripting (XSS), leading to more severe attacks like session hijacking or data theft.	
How It Was Discovered	
Manual Analysis – By observing the output in Burpsuite	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_6/html_injection_6.php	
Consequences of not Fixing the Issue	
If this HTML Injection is not fixed, attackers can modify webpage content, deface the site, or trick users into entering sensitive information (phishing). They can also inject malicious scripts to steal cookies, hijack user sessions, or spread malware. This can lead to data breaches, loss of user trust, and legal consequences for the website owner.	
Suggested Countermeasures	
Sanitize and encode user input, validate input properly, use CSP and WAF, and store files securely. Regular security testing helps prevent attacks.	
References	
https://owasp.org/www-community/attacks/HTML_Injection	

Proof of Concept





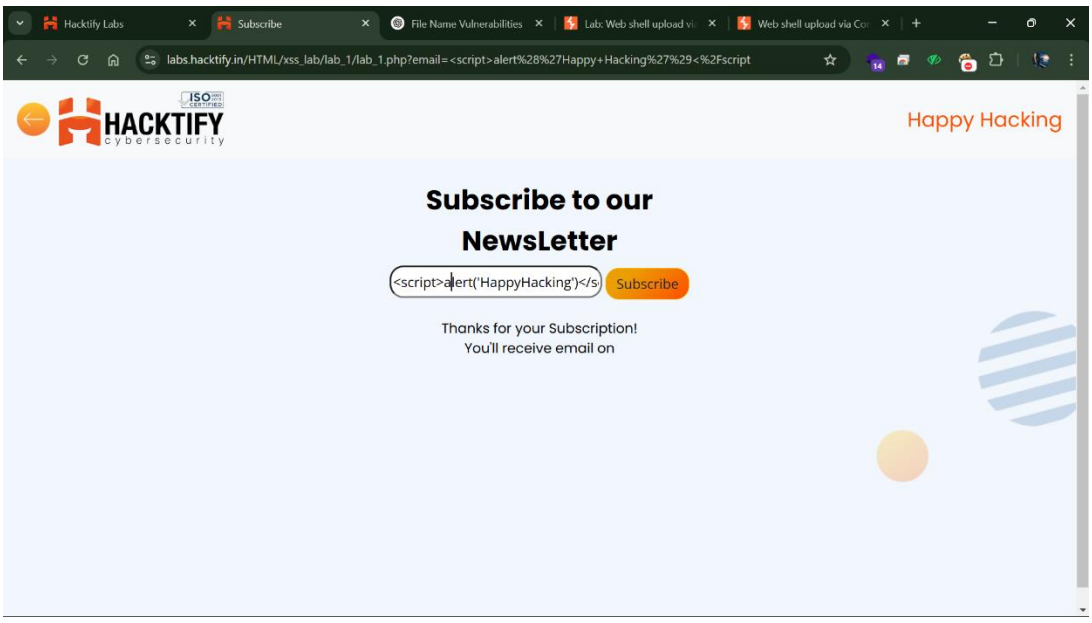

Hacktify Labs
Search Text
File Name Vulnerabilities
Lab: Web shell upload vi
Web shell upload via Cor
Not secure
https://labs.hacktify.in/HTML/html_lab/lab_6/html_injection_6.php
HACKTIFY
cybersecurity
Happy Hacking
Search and Filter
Enter text
Search
Your Searched results for
Happy Hacking
© Copyrights 2021 Hacktify Cybersecurity All rights reserved

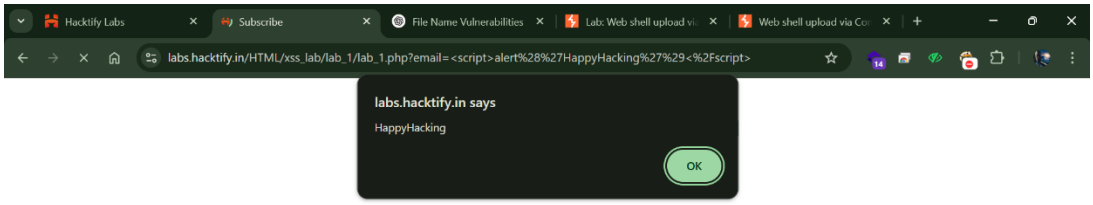
2. Cross-Site Scripting

2.1 Let's Do It!

Reference	Risk Rating
Let's Do It!	**Low**
Tools Used	
XSS payloads - Scripts	
Vulnerability Description	
The vulnerability you exploited is **Reflected Cross-Site Scripting (XSS)**, where user input is not properly sanitized and gets executed as JavaScript in the browser. This allows attackers to inject malicious scripts, which can steal cookies, deface webpages, or perform unauthorized actions on behalf of users. Fixing this requires **proper input encoding**, **CSP implementation**, and **disabling inline scripts**.	
How It Was Discovered	
Manual Analysis – using XSS payloads or Scripts.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_1/lab_1.php	
Consequences of not Fixing the Issue	
If **Reflected XSS** is not fixed, attackers can steal user cookies, hijack sessions, deface websites, redirect users to malicious sites, or execute unauthorized actions on behalf of victims. This can lead to **data theft, loss of user trust, and security breaches**.	
Suggested Countermeasures	
To prevent **Reflected XSS**, always **sanitize and encode user input** before displaying it. Use htmlspecialchars() in PHP, implement **Content Security Policy (CSP)** to block inline scripts, and enable **input validation** to reject harmful characters. A **Web Application Firewall (WAF)** can also help detect and block attacks.	
References	
https://owasp.org/www-community/attacks/xss/	

Proof of Concept

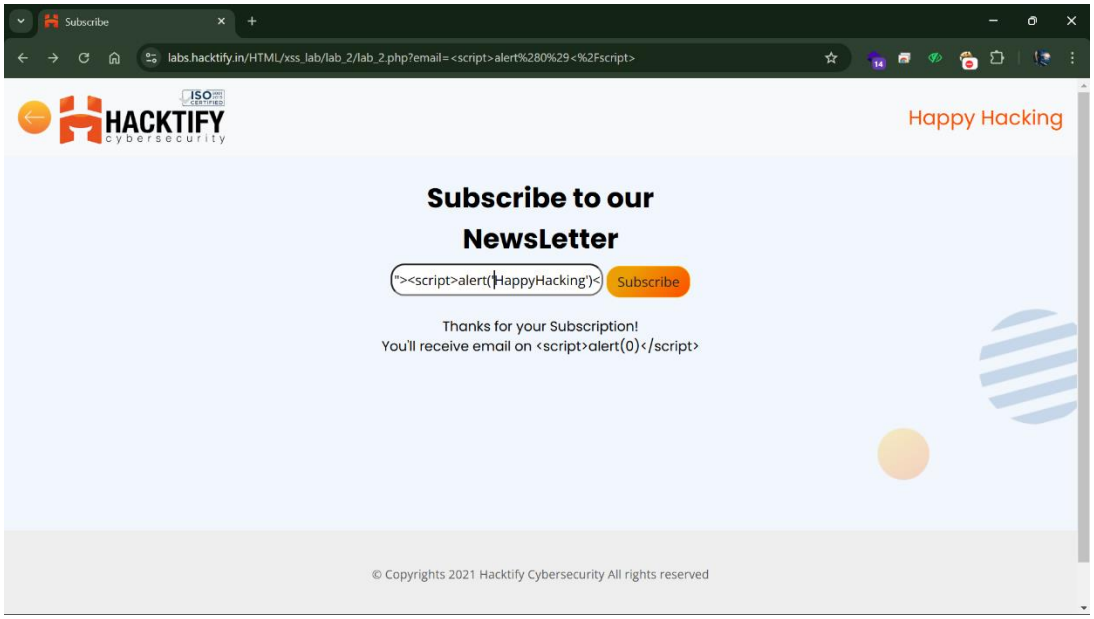


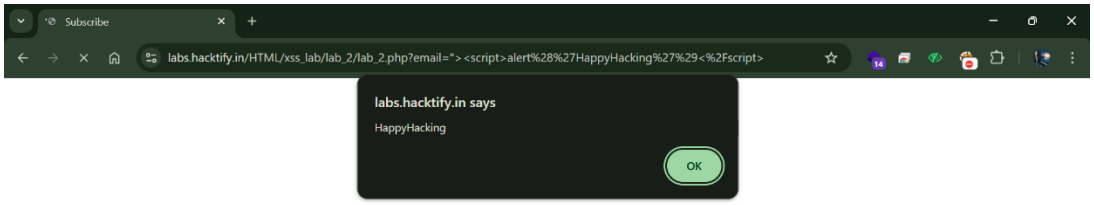


2.2 Balancing is Important in Life!

Reference	Risk Rating
Balancing is Important in Life!	**Low**
Tools Used	
XSS payloads - Scripts	
Vulnerability Description	
The vulnerability you exploited is **Reflected Cross-Site Scripting (XSS)**, where user input is not properly sanitized and is executed as JavaScript in the browser. Your payload injected a <script> tag, causing the browser to run the malicious code. Attackers can use this to **steal cookies, hijack sessions, redirect users, or deface webpages**. Fixing this requires **proper input encoding, CSP implementation, and input validation** to block harmful scripts.	
How It Was Discovered	
Manual Analysis – using XSS payloads or Scripts.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_2/lab_2.php	
Consequences of not Fixing the Issue	
If **Reflected XSS** is not fixed, attackers can steal user cookies, hijack accounts, deface webpages, redirect users to phishing sites, or execute malicious actions on behalf of victims. This can lead to **data breaches, financial loss, reputational damage, and legal consequences** for the website owner.	
Suggested Countermeasures	
To prevent **Reflected XSS**, **sanitize and encode user input** using htmlspecialchars() in PHP, implement **Content Security Policy (CSP)** to block inline scripts, and use **input validation** to reject malicious characters. Enable a **Web Application Firewall (WAF)** to detect and block attacks, and conduct regular security testing to identify vulnerabilities.	
References	
https://portswigger.net/web-security/cross-site-scripting	

Proof of Concept

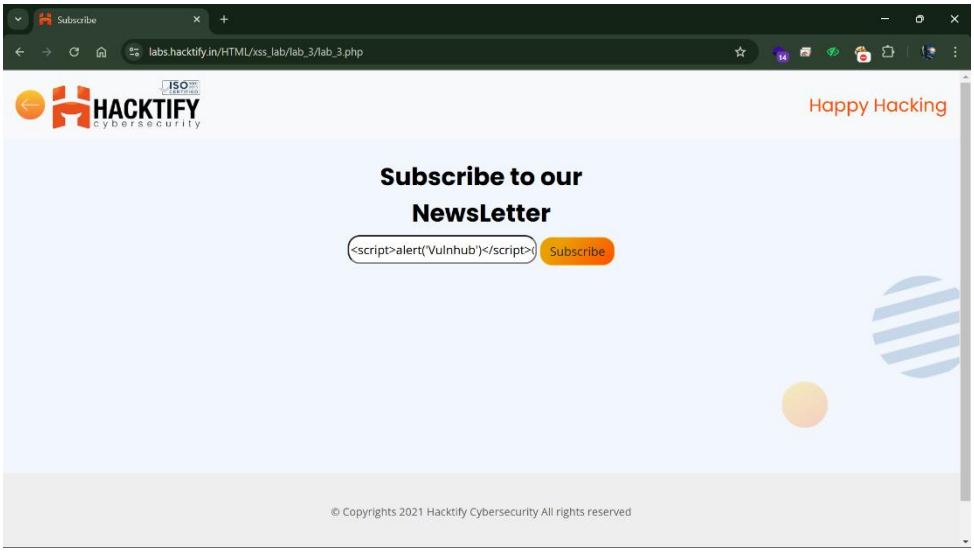


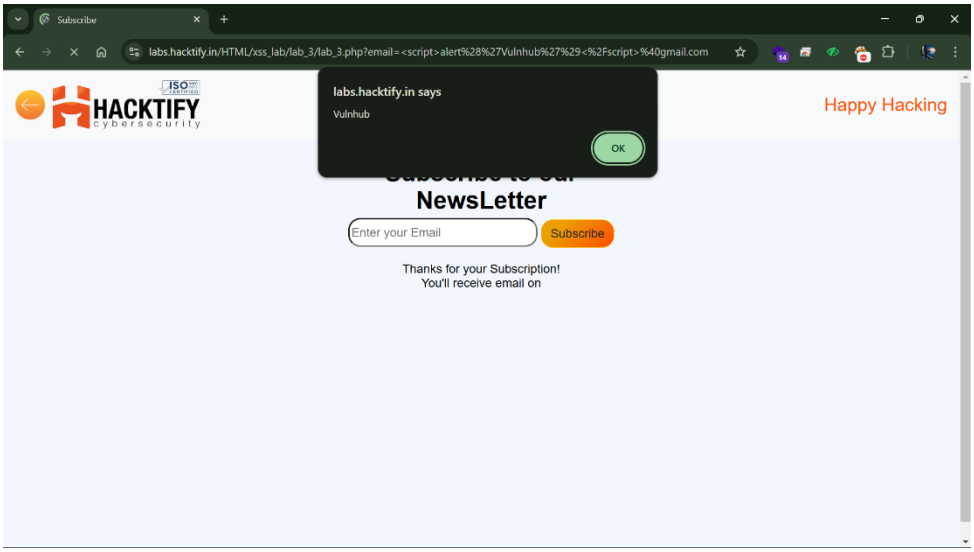


2.3 XSS is everywhere!

Reference	Risk Rating
XSS is everywhere!	**Low**
Tools Used	
XSS payloads - Scripts	
Vulnerability Description	
The vulnerability you exploited is **Reflected Cross-Site Scripting (XSS)**, where user input is not properly sanitized and gets executed as JavaScript in the browser. Your payload contained a <script> tag within an email input field, tricking the application into running malicious code. Attackers can use this to **steal cookies, hijack sessions, inject phishing forms, or manipulate website content**. Fixing this requires **proper input validation, output encoding, and CSP (Content Security Policy) implementation** to block malicious scripts.	
How It Was Discovered	
Manual Analysis – using XSS payloads with mail extension (@gmail.com).	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_3/lab_3.php	
Consequences of not Fixing the Issue	
If **Reflected XSS** is not fixed, attackers can execute malicious scripts to **steal cookies, hijack user sessions, deface webpages, redirect users to phishing sites, or inject fake login forms**. This can lead to **data breaches, identity theft, reputational damage, and financial loss** for both users and the website owner.	
Suggested Countermeasures	
To prevent **Reflected XSS**, **sanitize and encode user input** using htmlspecialchars() in PHP, implement **To prevent Reflected XSS, sanitize and encode user input using htmlspecialchars() in PHP, implement Content Security Policy (CSP) to block inline scripts, and use input validation to reject malicious characters. Additionally, enable a Web Application Firewall (WAF) to detect and block attacks, and conduct regular security testing to identify vulnerabilities before attackers exploit them.**	
References	
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html	

Proof of Concept

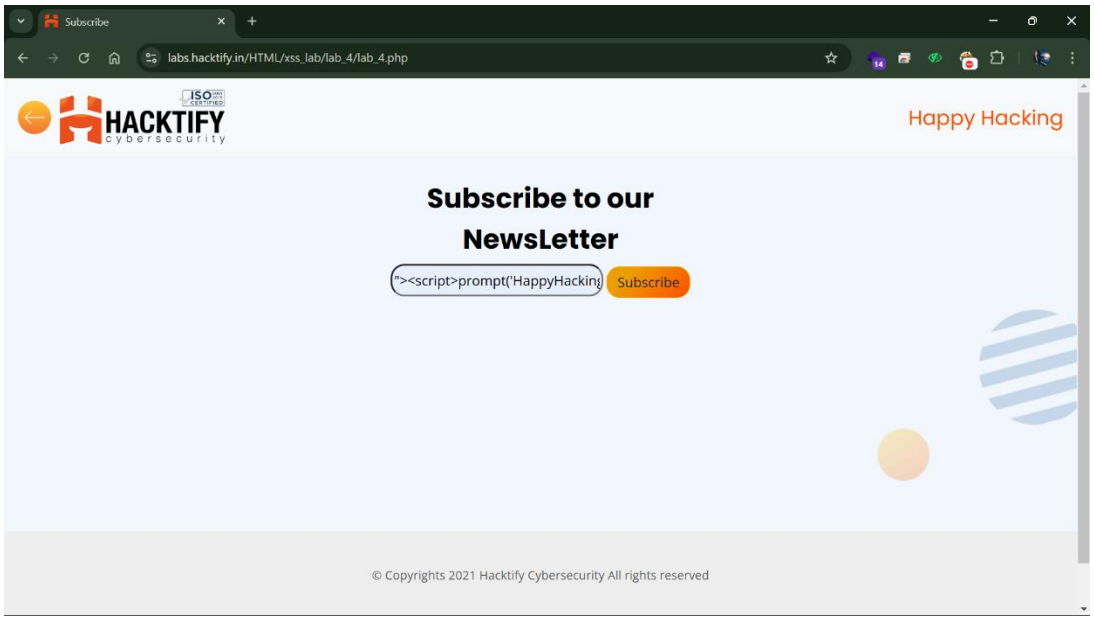


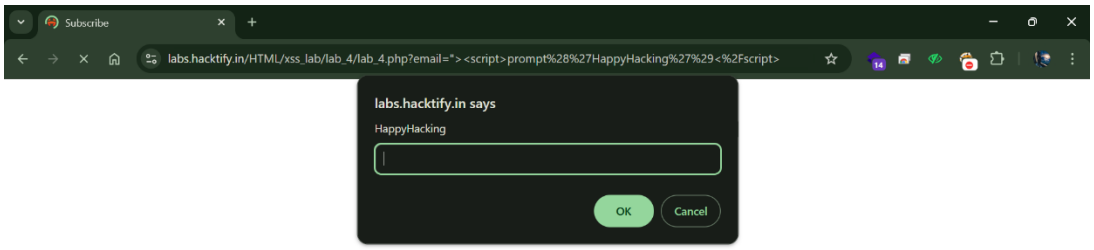


2.4 Alternatives are must!

Reference	Risk Rating
Alternatives are must!	**Medium**
Tools Used	
XSS payloads - Scripts	
Vulnerability Description	
The vulnerability you exploited is **Reflected Cross-Site Scripting (XSS)**, where user input is not properly sanitized and gets executed as JavaScript in the browser. The payload injected a <script> tag, causing the browser to run the malicious code. Attackers can use this to **steal cookies, hijack sessions, inject phishing forms, or manipulate website content**. Fixing this requires **proper input validation, output encoding, and CSP (Content Security Policy) implementation** to block malicious scripts.	
How It Was Discovered	
Manual Analysis – understanding the page source.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_4/lab_4.php	
Consequences of not Fixing the Issue	
If **Reflected XSS** is not fixed, attackers can execute malicious scripts to **steal cookies, hijack user sessions, deface webpages, redirect users to phishing sites, or inject fake login forms**. This can lead to **data breaches, identity theft, reputational damage, and financial loss** for both users and the website owner.	
Suggested Countermeasures	
To prevent **Reflected XSS**, **sanitize and encode user input**, implement **Content Security Policy (CSP)** to block inline scripts, and use **input validation** to reject malicious characters. Additionally, enable a **Web Application Firewall (WAF)** to detect and block attacks, and conduct **regular security testing** to identify vulnerabilities before attackers exploit them.	
References	
https://stackoverflow.com/questions/12072124/whats-the-alternate-for-alert-function-in-javascript	

Proof of Concept

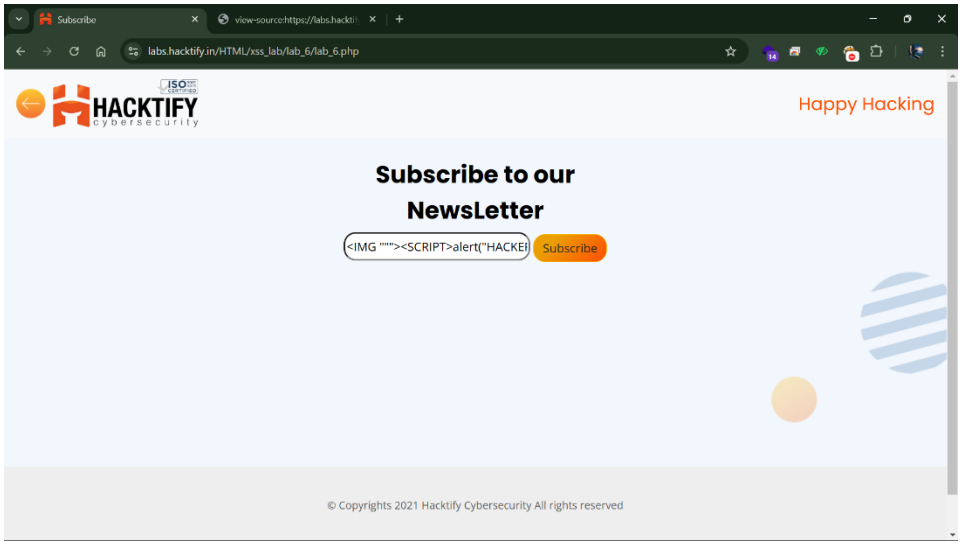


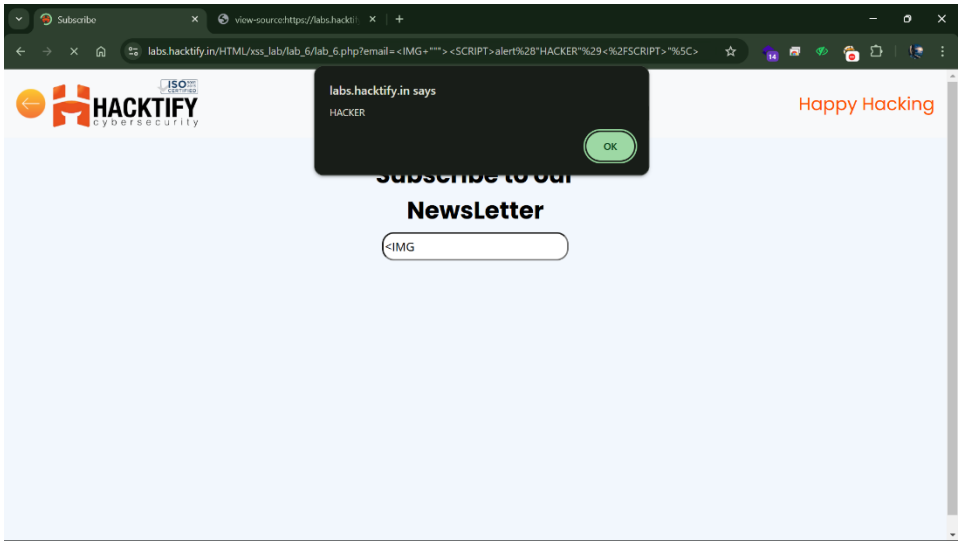


2.5 Developer hates Scripts!

Reference	Risk Rating
Developer hates Scripts!	**High**
Tools Used	
XSS payloads - Scripts	
Vulnerability Description	
The vulnerability you exploited is **Reflected Cross-Site Scripting (XSS)**, where the application fails to properly sanitize user input, allowing malicious JavaScript to execute in the browser. Your payload used an <IMG> tag with a broken attribute to inject a <SCRIPT> element, triggering an alert. Attackers can exploit this to **steal cookies, hijack user sessions, redirect users to phishing sites, or manipulate webpage content**.	
How It Was Discovered	
Manual Analysis – understanding the page source.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_5/lab_5.php	
Consequences of not Fixing the Issue	
If **Reflected XSS** is not fixed, attackers can exploit it to **execute arbitrary JavaScript in users' browsers**, leading to **session hijacking, credential theft, unauthorized actions on behalf of users, and redirection to malicious sites**. This can compromise **user trust, expose sensitive data, and damage the website's integrity**, making it a target for further exploitation.	
Suggested Countermeasures	
To prevent **Reflected XSS**, applications should **sanitize and encode user input** before rendering it in the browser. Implement **Content Security Policy (CSP)** to restrict script execution, use **input validation** to block dangerous characters, and employ **HTTP-only and secure cookies** to prevent session hijacking. Additionally, enabling a **Web Application Firewall (WAF)** and conducting **regular security audits** can help detect and mitigate XSS vulnerabilities.	
References	
https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html	

Proof of Concept

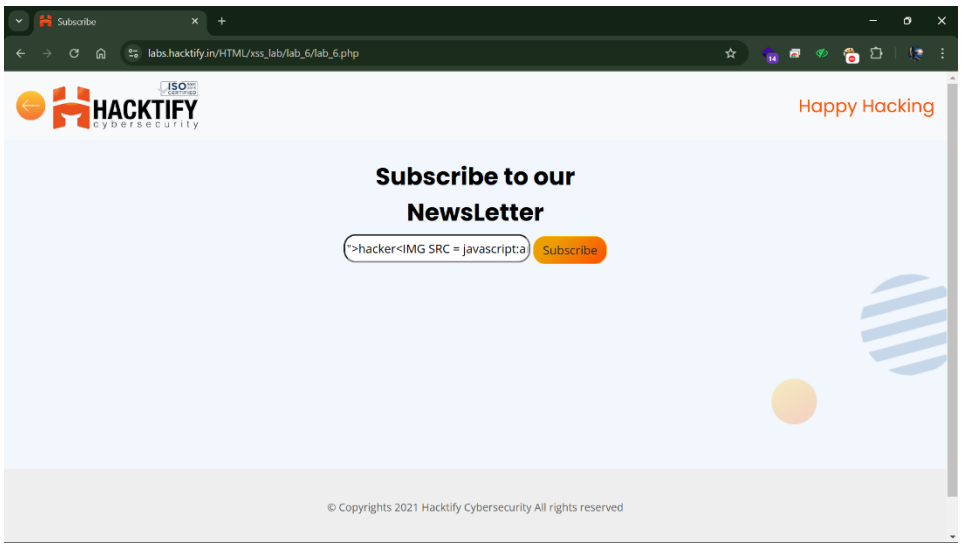


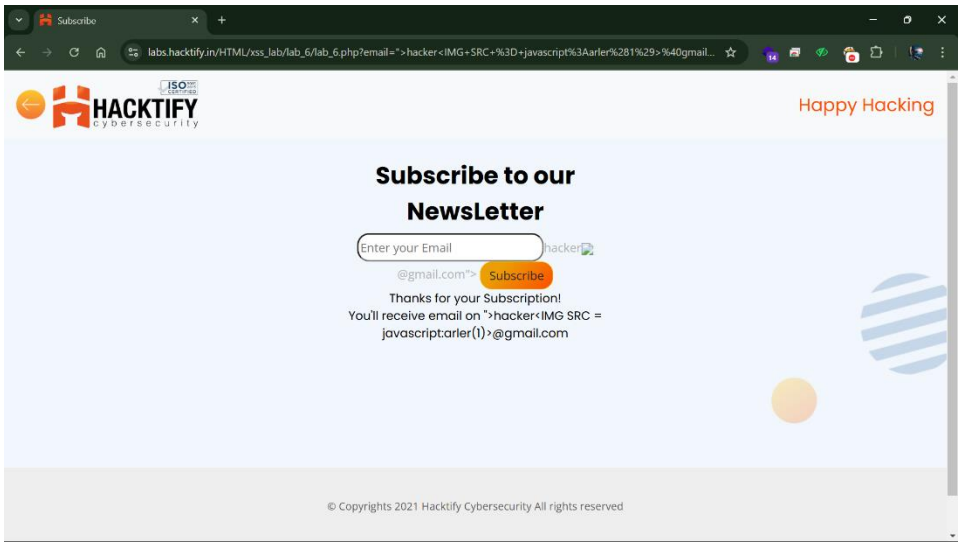


2.6 Change the Variation!

Reference	Risk Rating
Change the Variation!	**High**
Tools Used	
XSS payloads – using javascripts and HTML tags	
Vulnerability Description	
Stored Cross-Site Scripting (XSS) occurs when a web application **stores malicious JavaScript** in its database and serves it to users, executing the script in their browsers. This can lead to **session hijacking, data theft, phishing attacks, malware distribution, and website defacement**. Proper security measures like **input sanitization, output encoding, Content Security Policy (CSP), and Web Application Firewalls (WAF)** help prevent this vulnerability.	
How It Was Discovered	
Manual Analysis – understanding the page source.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_6/lab_6.php	
Consequences of not Fixing the Issue	
If this **Stored XSS** is not fixed, attackers can inject malicious scripts that execute every time a user visits the affected page. This can lead to **account takeovers, data theft, malware distribution, phishing attacks, and website defacement**, causing **reputational damage, financial loss, and security breaches**.	
Suggested Countermeasures	
To prevent **Stored XSS**, applications should **sanitize and validate all user input**, **encode output before rendering**, and implement **Content Security Policy (CSP)** to block unauthorized scripts. Using **HTTP-only cookies**, enabling a **Web Application Firewall (WAF)**, and conducting **regular security audits** can further reduce the risk.	
References	
https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html	

Proof of Concept

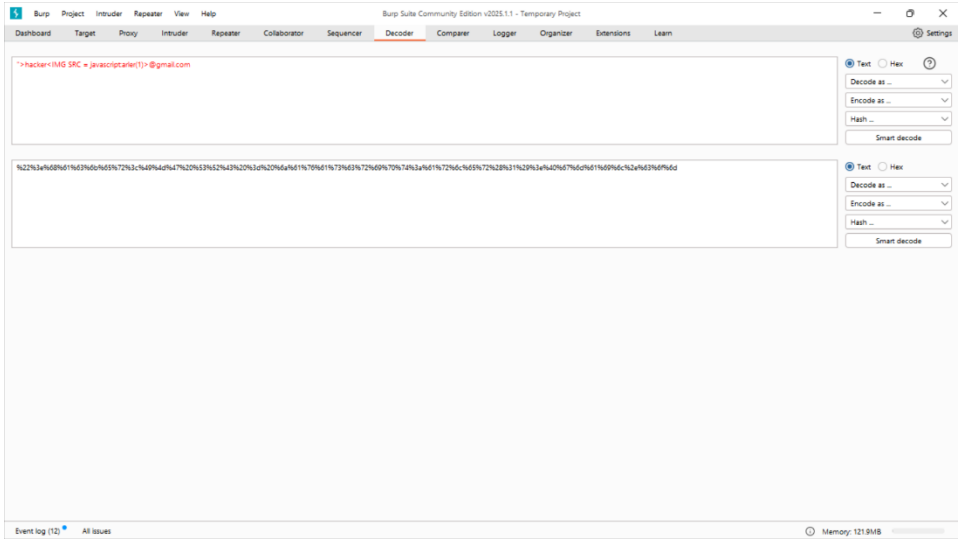


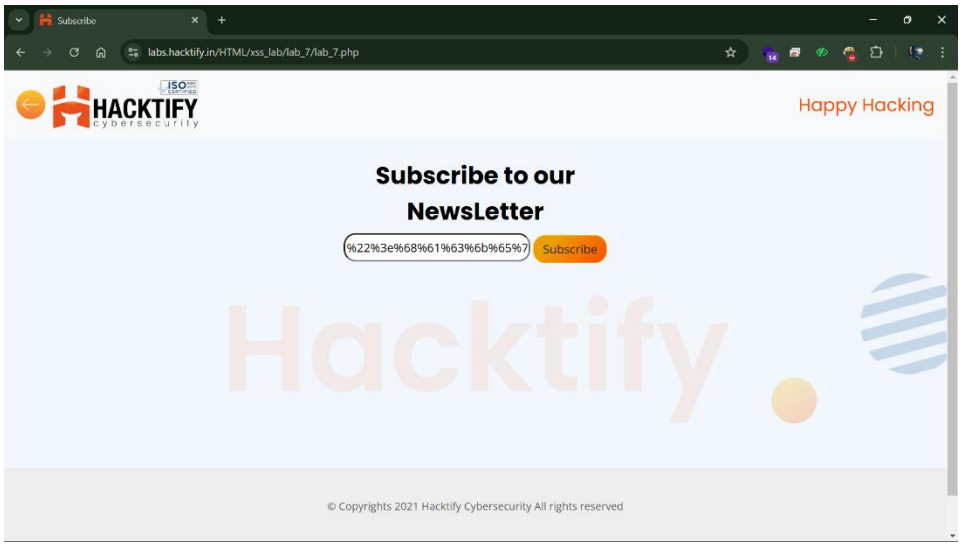


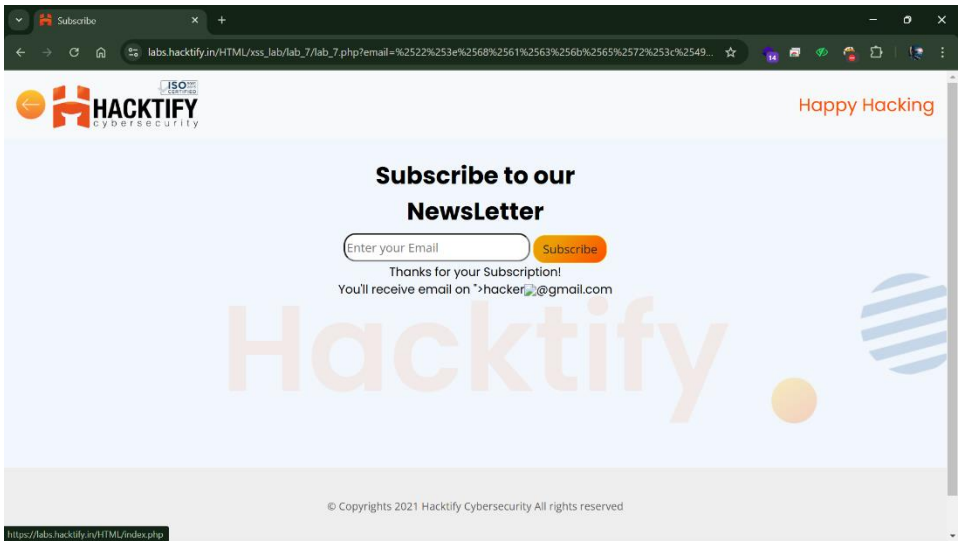
2.7 Encoding is the key?

Reference	Risk Rating
Encoding is the key?	**Medium**
Tools Used	
BurpSuite – Repeater and Decoder	
Vulnerability Description	
Stored Cross-Site Scripting (XSS) occurs when a web application stores maliciously encoded input, which later gets decoded and executed in the browser. Attackers use encoding techniques, such as URL encoding, to bypass security filters and inject harmful scripts. Once stored, these scripts can execute automatically whenever a user accesses the affected page, leading to account hijacking, data theft, phishing attacks, and malware distribution. Proper input validation, output encoding, and security mechanisms like Content Security Policy (CSP) and Web Application Firewalls (WAF) are essential to prevent such attacks.	
How It Was Discovered	
Manual Analysis – By Intercepting and manipulating the request.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_7/lab_7.php	
Consequences of not Fixing the Issue	
Ignoring **Stored XSS** allows attackers to inject persistent malicious scripts, which can automatically execute for every user accessing the compromised page. This can result in **unauthorized access, data breaches, phishing scams, malware spread, and website manipulation**. The consequences may include **user account compromises, financial losses, reputational damage, and legal liabilities** for the affected platform.	
Suggested Countermeasures	
To prevent **Stored XSS**, applications should **validate and sanitize all user inputs**, **encode output before displaying it**, and implement **Content Security Policy (CSP)** to restrict script execution. Additionally, using **HTTP-only cookies**, enabling a **Web Application Firewall (WAF)**, and conducting **regular security audits** help detect and block potential attacks.	
References	
https://www.imperva.com/learn/application-security/cross-site-scripting-xss-attacks/	

Proof of Concept



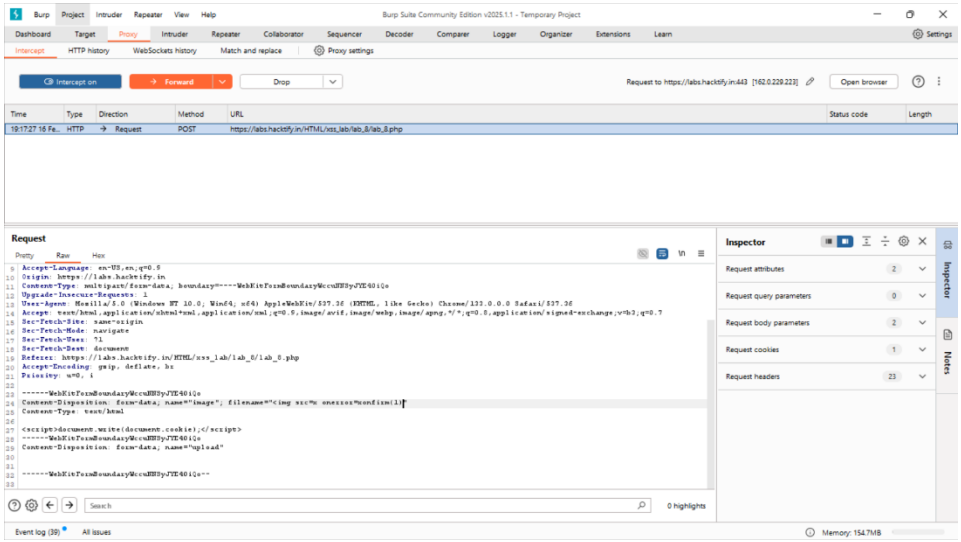


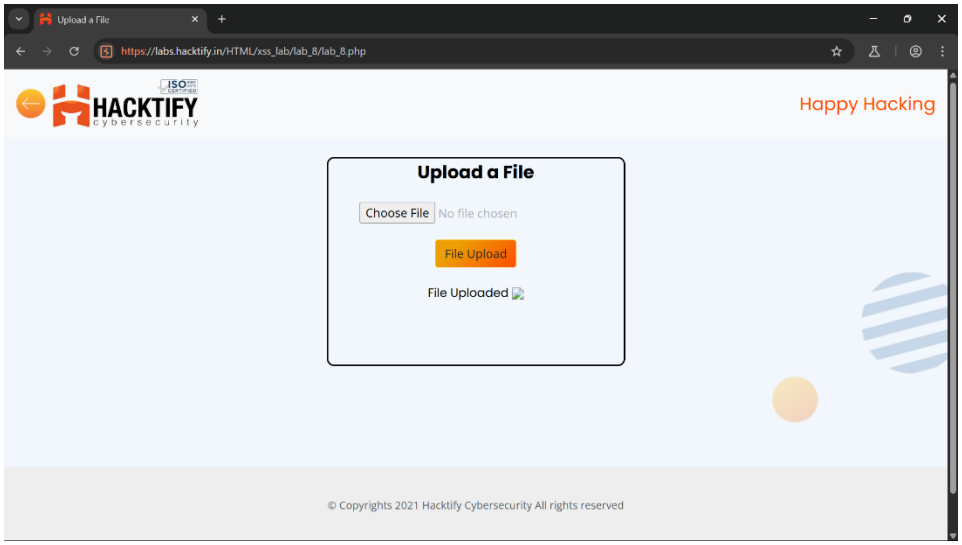


2.8 XSS with File Upload (file name)

Reference	Risk Rating
XSS with File Upload (file name)	**Low**
Tools Used	
BurpSuite – Repeater and Render	
Vulnerability Description	
XSS with File Upload (Filename-Based XSS) occurs when a web application fails to sanitize user-supplied filenames, allowing attackers to upload files with maliciously crafted names containing JavaScript code. When the application later displays the filename on a webpage, the script executes in the user's browser. This can lead to session hijacking, data theft, phishing attacks, or malware execution. Proper input validation, output encoding, and Content Security Policy (CSP) implementation can prevent this vulnerability.	
How It Was Discovered	
Manual Analysis – By modifying the request .	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_8/lab_8.php	
Consequences of not Fixing the Issue	
If this **Filename-Based XSS** is not fixed, attackers can upload files with **malicious JavaScript in the filename**, which executes when viewed in a web page. This can lead to **session hijacking, data theft, phishing attacks, website defacement, and malware distribution**, causing **security breaches, reputational damage, and financial loss**.	
Suggested Countermeasures	
To prevent **Filename-Based XSS**, applications should **sanitize and encode filenames** before displaying them, **restrict special characters**, and **validate file uploads** by allowing only trusted extensions. Implementing **Content Security Policy (CSP)** and using **proper output encoding** can further mitigate the risk.	
References	
https://portswigger.net/web-security/file-upload	

Proof of Concept

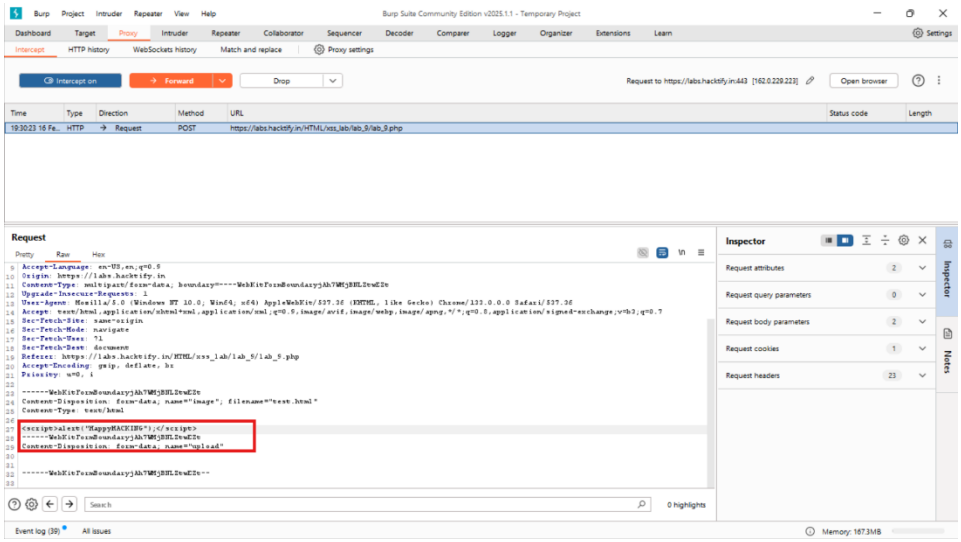


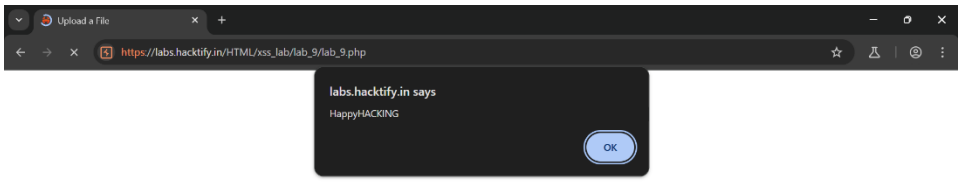


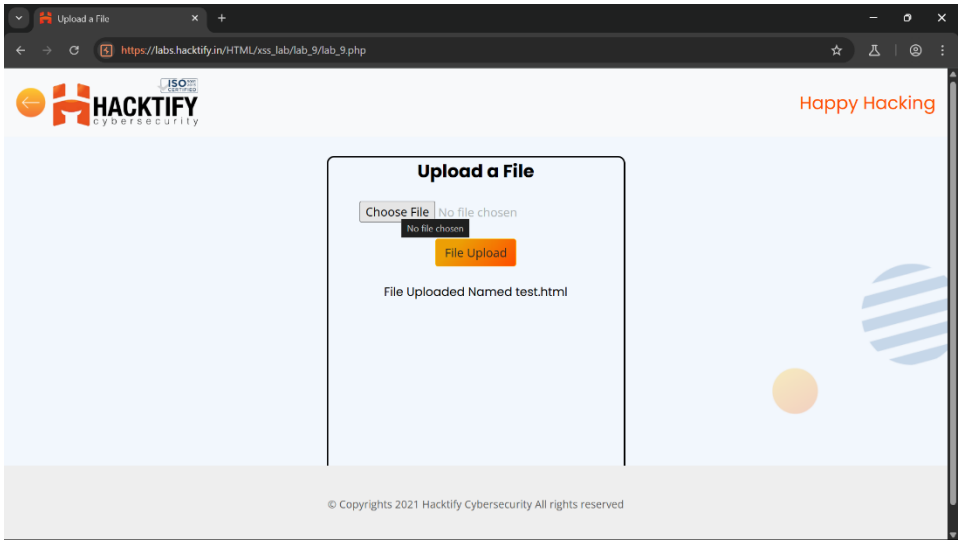
2.9 XSS with File Upload (File Content)

Reference	Risk Rating
XSS with File Upload (File Content)	**Low**
Tools Used	
BurpSuite	
Vulnerability Description	
XSS with File Upload (File Content-Based XSS) occurs when a web application fails to validate and sanitize uploaded files, allowing attackers to upload malicious scripts inside HTML, JavaScript, or other executable file types. When the file is accessed and executed by a user, the script runs in their browser, leading to session hijacking, data theft, phishing attacks, or malware distribution. Proper MIME type validation, content filtering, and serving uploaded files from a different domain can help prevent this vulnerability.	
How It Was Discovered	
Manual Analysis – By observing the request .	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_9/lab_9.php	
Consequences of not Fixing the Issue	
If **File Content-Based XSS** is not fixed, attackers can upload malicious files that execute JavaScript when opened, leading to **session hijacking, data theft, phishing attacks, malware infections, and website compromise**. This can result in **loss of sensitive information, reputational damage, financial loss, and legal consequences** for the affected platform.	
Suggested Countermeasures	
To prevent **File Content-Based XSS**, applications should **restrict allowed file types**, **validate MIME types**, and **block executable scripts**. Serving uploaded files from a **separate domain or sandboxed environment** and enforcing **Content Security Policy (CSP)** can further reduce risks. Regular **security audits and monitoring** help detect potential threats.	
References	
https://portswigger.net/web-security/file-upload	

Proof of Concept



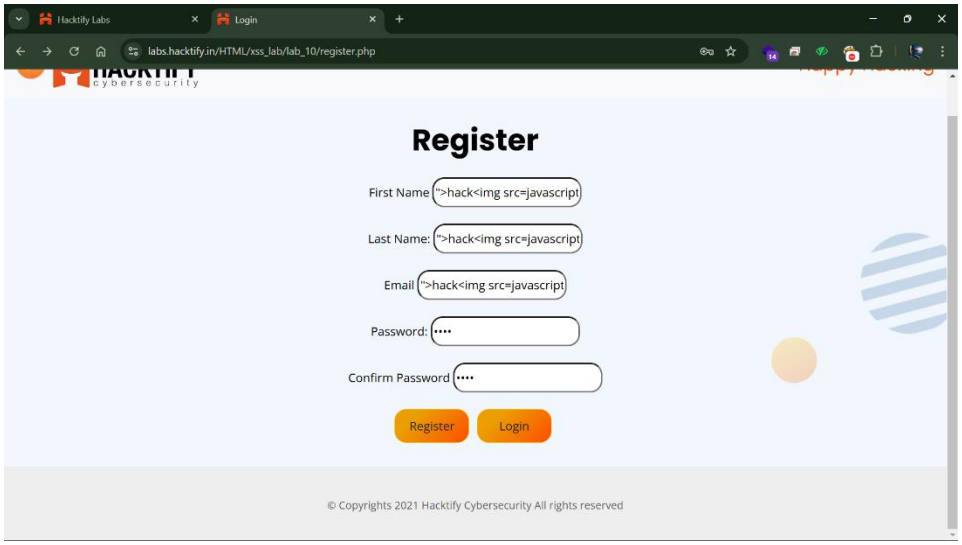


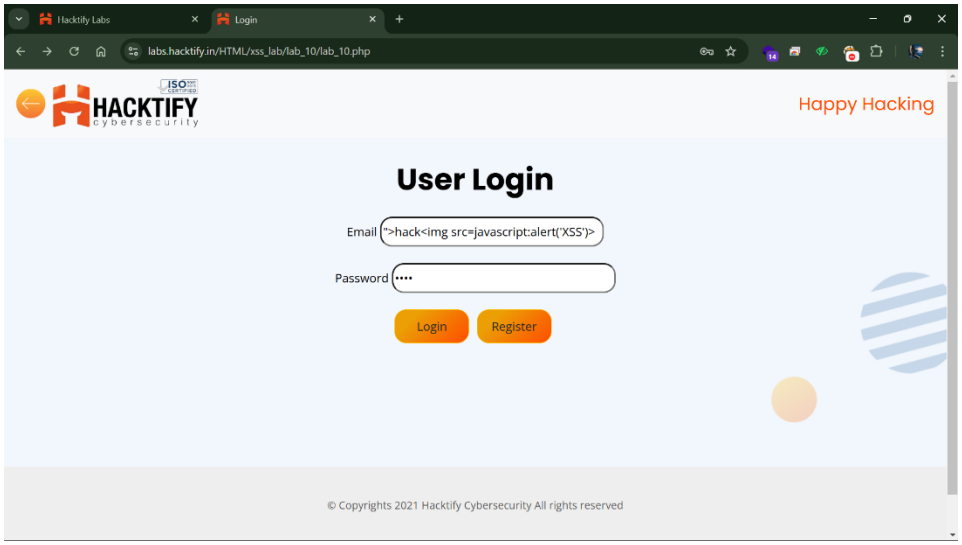


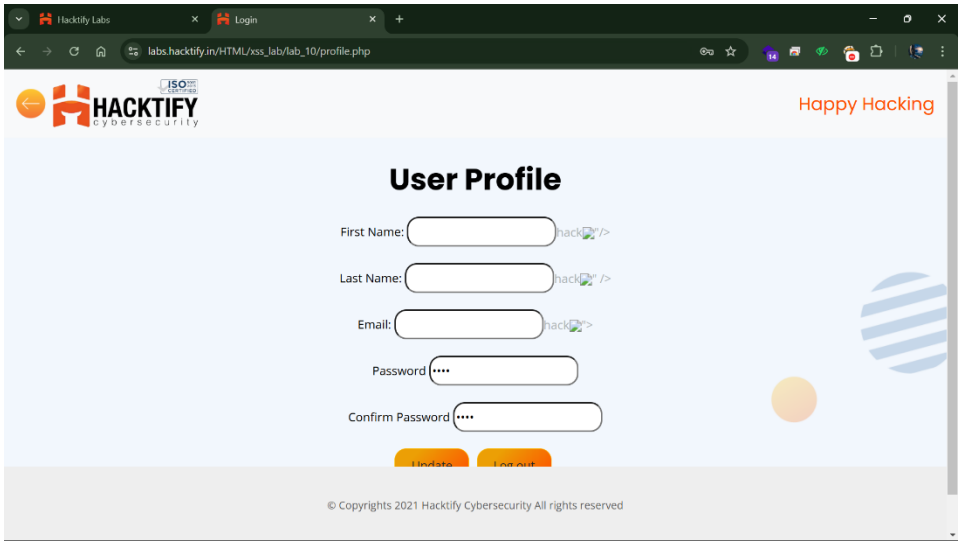
2.10 Stored Everywhere!

Reference	Risk Rating
Stored Everywhere!	**Low**
Tools Used	
Browser Developr Tools - Inspector	
Vulnerability Description	
Stored Cross-Site Scripting (XSS) occurs when a web application stores malicious JavaScript in its database and later displays it on a web page without proper sanitization. When users visit the affected page, the script executes in their browsers, allowing attackers to steal session cookies, hijack accounts, deface websites, spread malware, or launch phishing attacks. Preventing this requires input validation, output encoding, Content Security Policy (CSP), and regular security audits.	
How It Was Discovered	
Manual Analysis – By observing the page source code.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_10/lab_10.php	
Consequences of not Fixing the Issue	
If this **Stored XSS** is not fixed, attackers can inject malicious scripts that execute whenever users visit the affected page. This can lead to **session hijacking, data theft, phishing attacks, malware infections, and website defacement**, causing **loss of user trust, financial damage, reputational harm, and potential legal consequences** for the organization.	
Suggested Countermeasures	
To prevent **Stored XSS**, applications should **validate and sanitize user input**, **encode output before displaying it**, and enforce a **Content Security Policy (CSP)** to block unauthorized scripts. Using **HttpOnly cookies, Web Application Firewalls (WAF), and regular security audits** can further reduce the risk.	
References	
https://www.acunetix.com/websitesecurity/cross-site-scripting/	

Proof of Concept



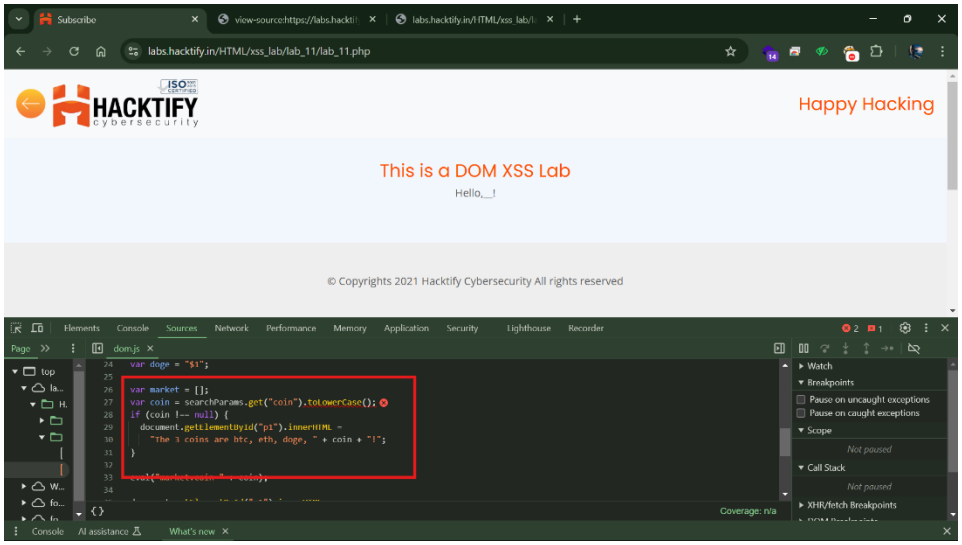

Hacktify Labs
Login
labs.hacktify.in/HTML/xss_lab/lab_10/lab_10.php
HACKTIFY
cybersecurity
Happy Hacking
User Login
Email
">hack<img src=javascript:alert('XSS')>
Password
Login
Register
© Copyrights 2021 Hacktify Cybersecurity All rights reserved

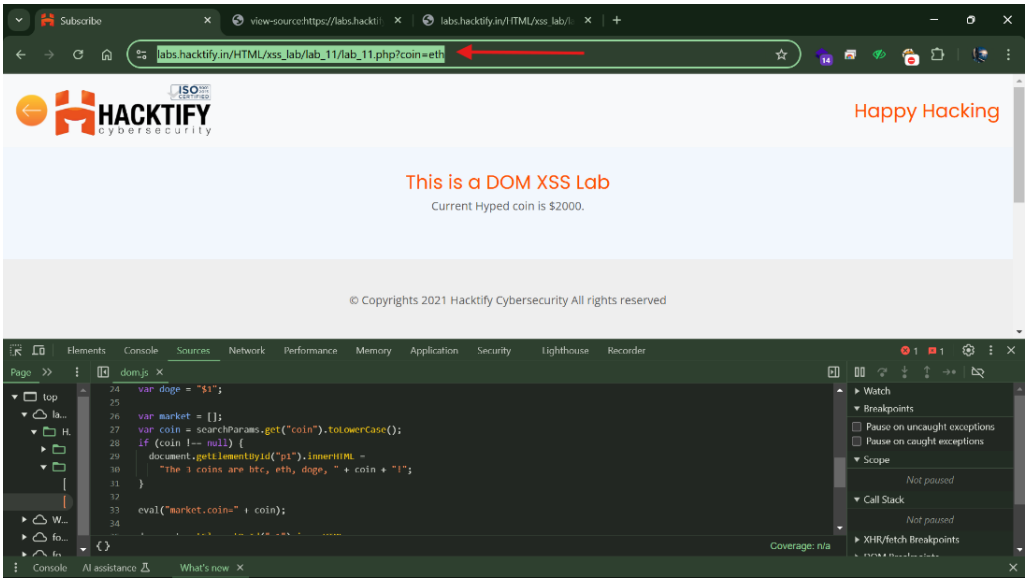

Hacktify Labs
Login
labs.hacktify.in/HTML/xss_lab/lab_10/profile.php
HACKTIFY
cybersecurity
Happy Hacking
User Profile
First Name:
Last Name:
Email:
Password
Confirm Password
© Copyrights 2021 Hacktify Cybersecurity All rights reserved

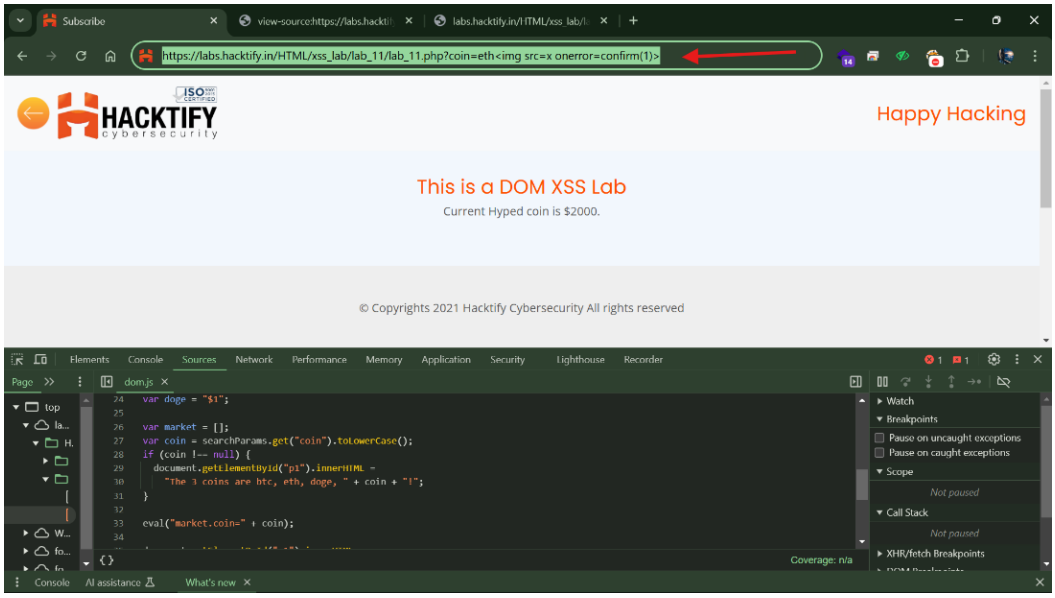
2.11 DOM's are love!

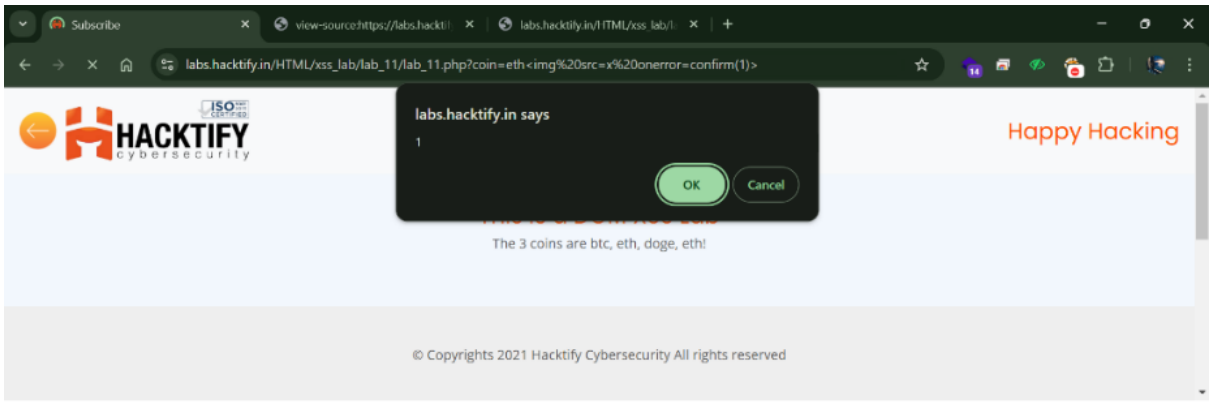
Reference	Risk Rating
DOM's are love!	**High**
Tools Used	
Browser Developr Tools - Inspector	
Vulnerability Description	
DOM-Based XSS occurs when malicious JavaScript is injected and executed within the browser's DOM (Document Object Model) instead of the server. The vulnerability arises when client-side scripts process untrusted user input (e.g., from window.location, document.referrer, or innerHTML) without proper sanitization. This allows attackers to manipulate the page dynamically, leading to session hijacking, data theft, phishing attacks, or malware execution. Proper input validation, output encoding, and security controls like Content Security Policy (CSP) can help prevent this attack.	
How It Was Discovered	
Manual Analysis – By observing the page source code.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php	
Consequences of not Fixing the Issue	
If this **DOM XSS** is not fixed, attackers can manipulate client-side scripts to execute malicious code in users' browsers. This can lead to **session hijacking, credential theft, phishing attacks, unauthorized actions on behalf of users, and malware injection**. The impact includes **loss of sensitive data, reputational damage, financial loss, and legal consequences** for the affected organization.	
Suggested Countermeasures	
To prevent **DOM XSS**, avoid using **untrusted user input** directly in the DOM. Use **secure JavaScript methods** like textContent instead of innerHTML, validate and sanitize inputs, and implement **Content Security Policy (CSP)** to block malicious scripts. Regular **security audits and penetration testing** help detect vulnerabilities early.	
References	
https://owasp.org/www-community/attacks/DOM_Based_XSS	

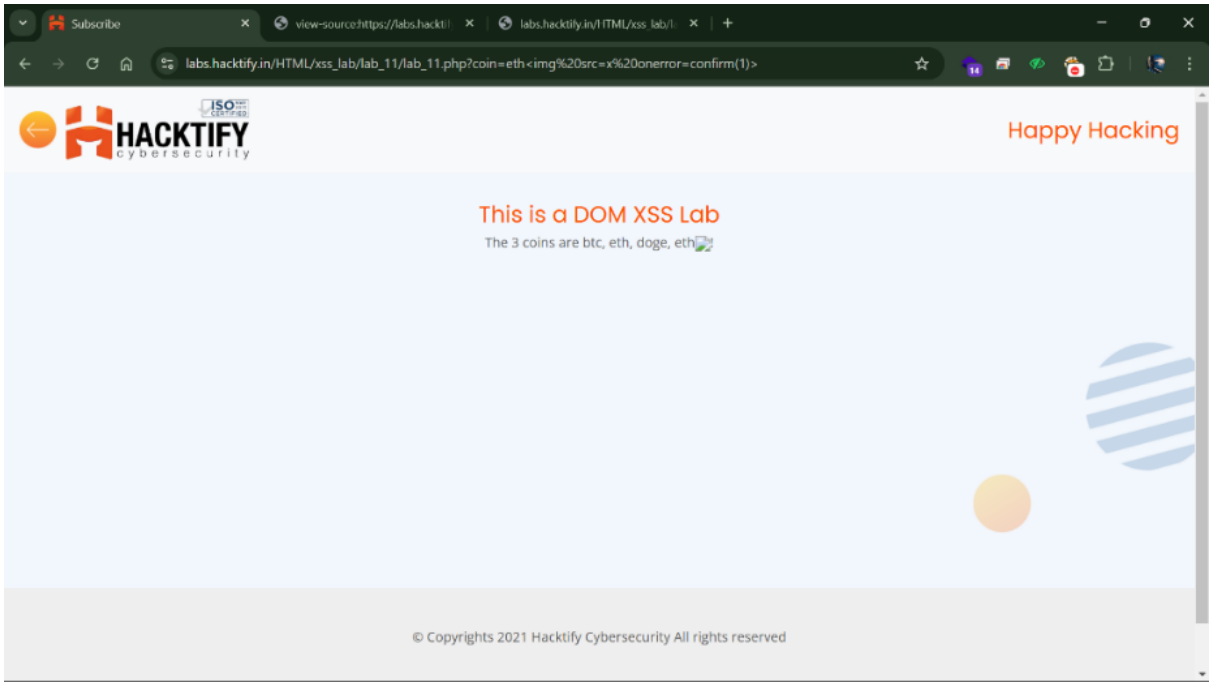
Proof of Concept




Subscribe
labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?coin=eth
HACKTIFY
cybersecurity
Happy Hacking
This is a DOM XSS Lab
Current Hyped coin is $2000.
© Copyrights 2021 Hacktify Cybersecurity All rights reserved
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Recorder
dom.js
var doge = "$1";
var market = [];
var coin = searchParams.get("coin").toLowerCase();
if (coin !== null) {
document.getElementById("p1").innerHTML =
"The 3 coins are btc, eth, doge, " + coin + "!";
}
eval("market.coin=" + coin);
Watch
Breakpoints
Pause on uncaught exceptions
Pause on caught exceptions
Scope
Not paused
Call Stack
Not paused
XHR/fetch Breakpoints
Coverage: n/a
Console
AI assistance
What's new


Subscribe
https://labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?coin=eth<img src=x onerror=confirm(1)>
HACKTIFY
cybersecurity
Happy Hacking
This is a DOM XSS Lab
Current Hyped coin is $2000.
© Copyrights 2021 Hacktify Cybersecurity All rights reserved
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Recorder
dom.js
var doge = "$1";
var market = [];
var coin = searchParams.get("coin").toLowerCase();
if (coin !== null) {
document.getElementById("p1").innerHTML =
"The 3 coins are btc, eth, doge, " + coin + "!";
}
eval("market.coin=" + coin);
Watch
Breakpoints
Pause on uncaught exceptions
Pause on caught exceptions
Scope
Not paused
Call Stack
Not paused
XHR/fetch Breakpoints
Coverage: n/a
Console
AI assistance
What's new


Subscribe
labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?coin=eth<img%20src=x%20onerror=confirm(1)>
HACKTIFY
cybersecurity
labs.hacktify.in says
1
OK
Cancel
Happy Hacking
The 3 coins are btc, eth, doge, eth!
© Copyrights 2021 Hacktify Cybersecurity All rights reserved


Subscribe
labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?coin=eth<img%20src=x%20onerror=confirm(1)>
HACKTIFY
cybersecurity
Happy Hacking
This is a DOM XSS Lab
The 3 coins are btc, eth, doge, eth
© Copyrights 2021 Hacktify Cybersecurity All rights reserved

Penetration Testing Report

Full Name: AVIREDDY RUSHIKESH
Program: HCS - Penetration Testing Internship Week-2
Date: 25/02/25

Introduction

This report document hereby describes the proceedings and results of a Black Box security assessment conducted against the **Week {2} Labs**. The report hereby lists the findings and corresponding best practice mitigation actions and recommendations.

1. Objective

The objective of the assessment was to uncover vulnerabilities in the **Week {2} Labs** 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.

2. Scope

This section defines the scope and boundaries of the project.

Application Name	**Lab 1 Name – SQL Injection, Lab 2 Name – Insecure Direct Object Reference .**

3. Summary

Outlined is a Black Box Application Security assessment for the **Week {#} Labs**.

Total number of Sub-labs: {count} Sub-labs

High	Medium	Low
4	6	6

High - **4**

Medium - **6**

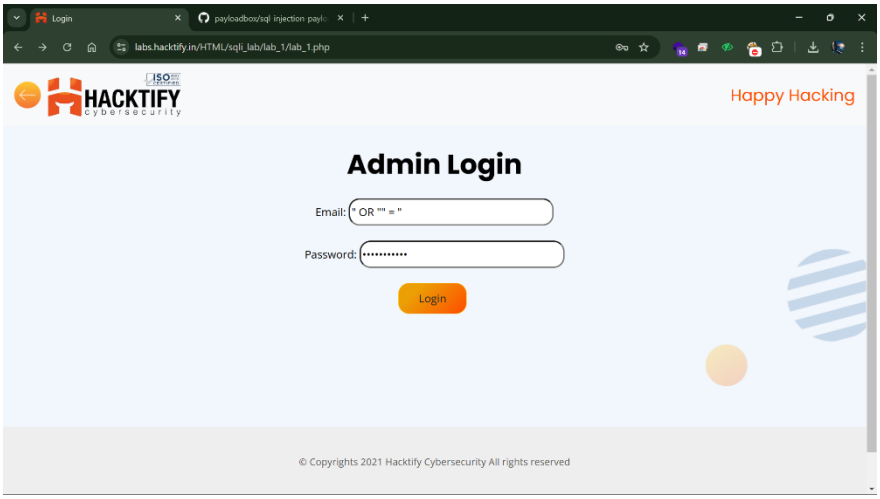
Low - 6

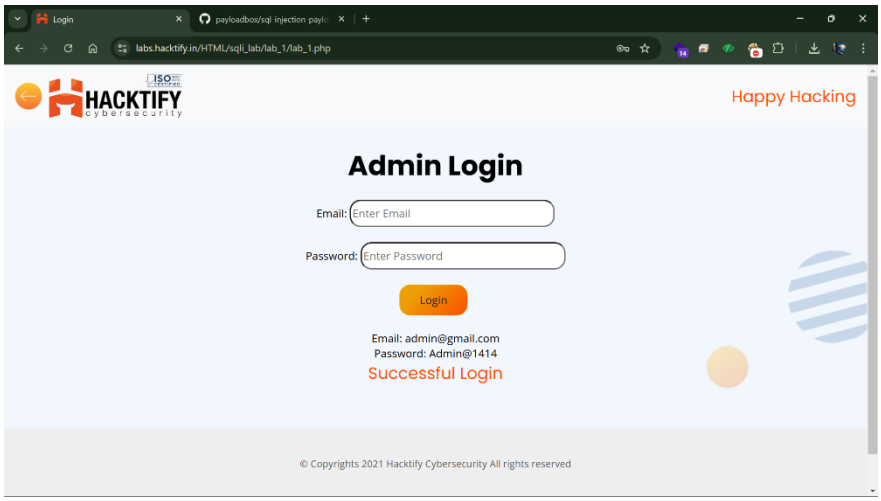
1. SQL Injection

1.1. Strings & Errors Part 1

Reference	Risk Rating
Strings & Errors Part 1	**Low**
Tools Used	
SQL Injection – SQL payloads	
Vulnerability Description	
SQL Injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It occurs when an application improperly handles user input, enabling malicious SQL statements to be injected and executed within the database. This can lead to unauthorized access, data manipulation, or even complete control over the database.	
How It Was Discovered	
Manual Analysis – SQL payloads	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_1/lab_1.php	
Consequences of not Fixing the Issue	
Failure to fix SQL Injection (SQLi) vulnerabilities can lead to severe consequences, including unauthorized data breaches, exposure of sensitive information (user credentials, financial data, personal records), and identity theft. Attackers can manipulate or delete critical data, bypass authentication, and gain administrative access to the database, potentially compromising the entire system. This can result in financial losses, legal penalties (due to GDPR, HIPAA, or PCI-DSS violations), reputational damage, and operational disruptions, ultimately endangering business continuity and user trust.	
Suggested Countermeasures	
To prevent SQL Injection (SQLi), use **prepared statements (parameterized queries)** and **stored procedures** to separate SQL code from user input. Implement **input validation** to restrict unexpected characters and enforce strict data types. Apply the **principle of least privilege (PoLP)** to limit database user permissions. Use **web application firewalls (WAFs)** to detect and block SQLi attempts. Regularly perform **security testing (penetration testing, code reviews, and automated scans)** to identify and remediate vulnerabilities before attackers exploit them.	
References	
https://github.com/payloadbox/sql-injection-payload-list	

Proof of Concept

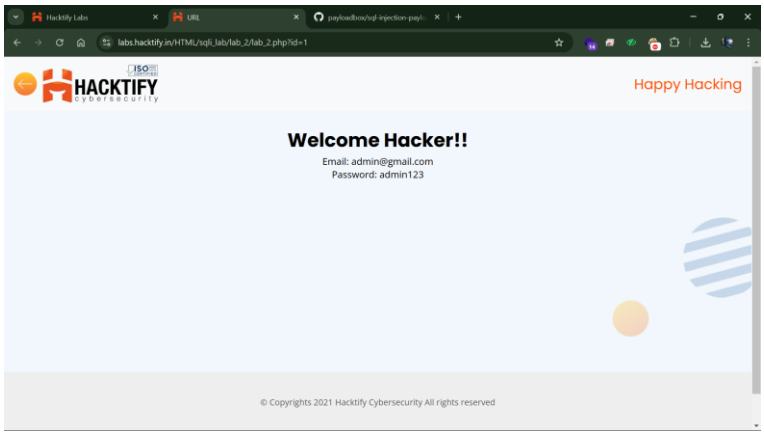


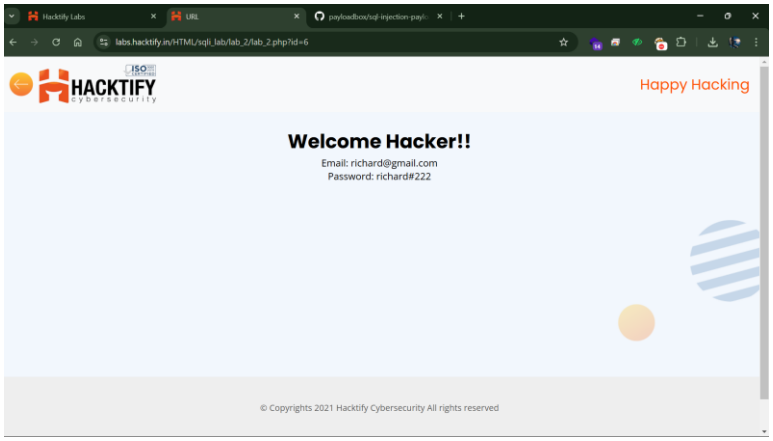


1.2 Strings & Errors Part 2

Reference	Risk Rating
Strings & Errors Part 2	**Low**
Tools Used	
SQL Injection – SQL Payloads	
Vulnerability Description	
SQL Injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It occurs when an application improperly handles user input, enabling malicious SQL statements to be injected and executed within the database. This can lead to unauthorized access, data manipulation, or even complete control over the database.	
How It Was Discovered	
Manual Analysis – SQL Payloads :- by changing id number.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_2/lab_2.php	
Consequences of not Fixing the Issue	
Failure to fix SQL Injection (SQLi) vulnerabilities can lead to severe consequences, including unauthorized data breaches, exposure of sensitive information (user credentials, financial data, personal records), and identity theft. Attackers can manipulate or delete critical data, bypass authentication, and gain administrative access to the database, potentially compromising the entire system. This can result in financial losses, legal penalties (due to GDPR, HIPAA, or PCI-DSS violations), reputational damage, and operational disruptions, ultimately endangering business continuity and user trust.	
Suggested Countermeasures	
To prevent SQL Injection (SQLi), use **prepared statements (parameterized queries)** and **stored procedures** to separate SQL code from user input. Implement **input validation** to restrict unexpected characters and enforce strict data types. Apply the **principle of least privilege (PoLP)** to limit database user permissions. Use **web application firewalls (WAFs)** to detect and block SQLi attempts. Regularly perform **security testing (penetration testing, code reviews, and automated scans)** to identify and remediate vulnerabilities before attackers exploit them.	
References	
https://github.com/payloadbox/sql-injection-payload-list	

Proof of Concept

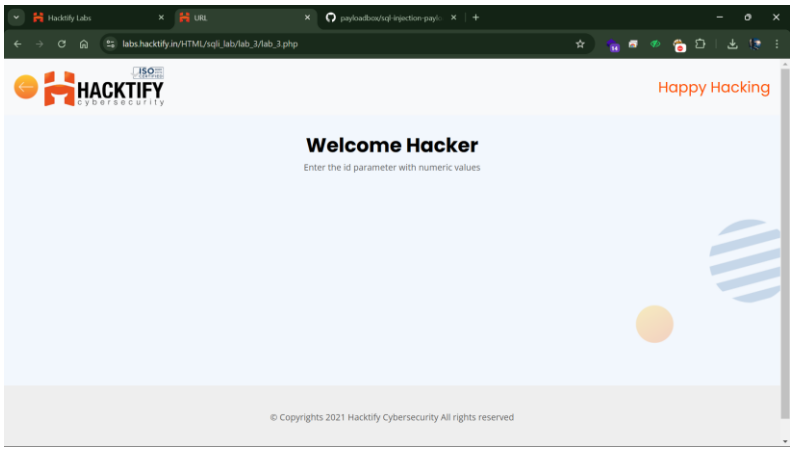


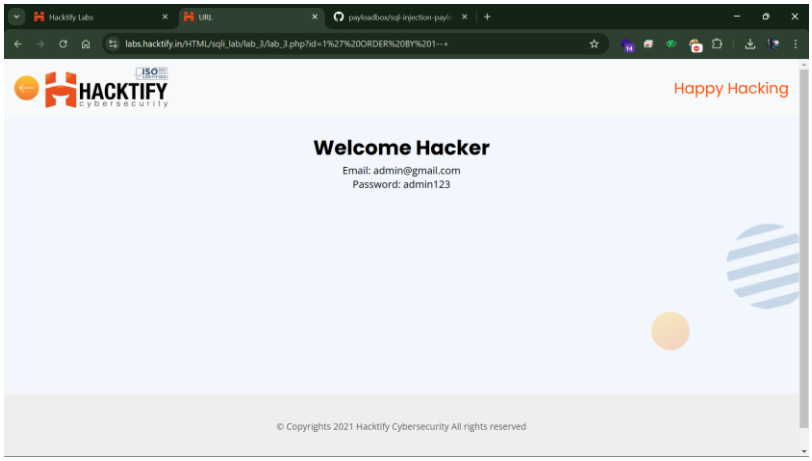


1.3 Strings & Errors Part 3

Reference	Risk Rating
Strings & Errors Part 3	**Low**
Tools Used	
SQL Injection – SQL Payloads	
Vulnerability Description	
SQL Injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It occurs when an application improperly handles user input, enabling malicious SQL statements to be injected and executed within the database. This can lead to unauthorized access, data manipulation, or even complete control over the database.	
How It Was Discovered	
Manual Analysis – SQL Payloads :- by changing id parameter with numeric values.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_3/lab_3.php	
Consequences of not Fixing the Issue	
Failure to fix SQL Injection (SQLi) vulnerabilities can lead to severe consequences, including unauthorized data breaches, exposure of sensitive information (user credentials, financial data, personal records), and identity theft. Attackers can manipulate or delete critical data, bypass authentication, and gain administrative access to the database, potentially compromising the entire system. This can result in financial losses, legal penalties (due to GDPR, HIPAA, or PCI-DSS violations), reputational damage, and operational disruptions, ultimately endangering business continuity and user trust.	
Suggested Countermeasures	
To prevent SQL Injection (SQLi), use **prepared statements (parameterized queries)** and **stored procedures** to separate SQL code from user input. Implement **input validation** to restrict unexpected characters and enforce strict data types. Apply the **principle of least privilege (PoLP)** to limit database user permissions. Use **web application firewalls (WAFs)** to detect and block SQLi attempts. Regularly perform **security testing (penetration testing, code reviews, and automated scans)** to identify and remediate vulnerabilities before attackers exploit them.	
References	
https://github.com/payloadbox/sql-injection-payload-list	

Proof of Concept

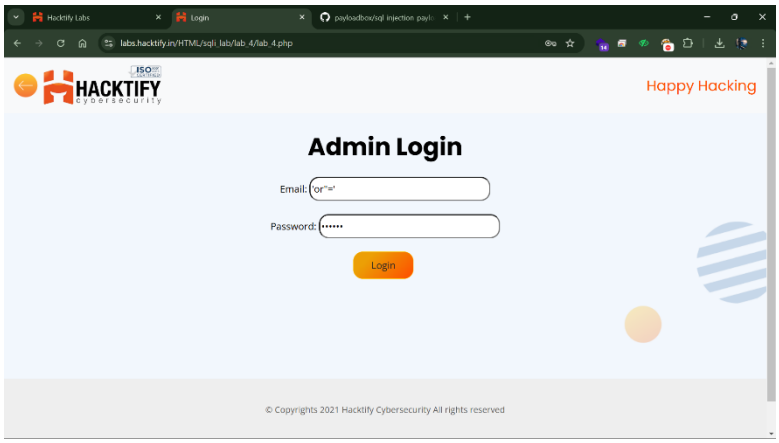


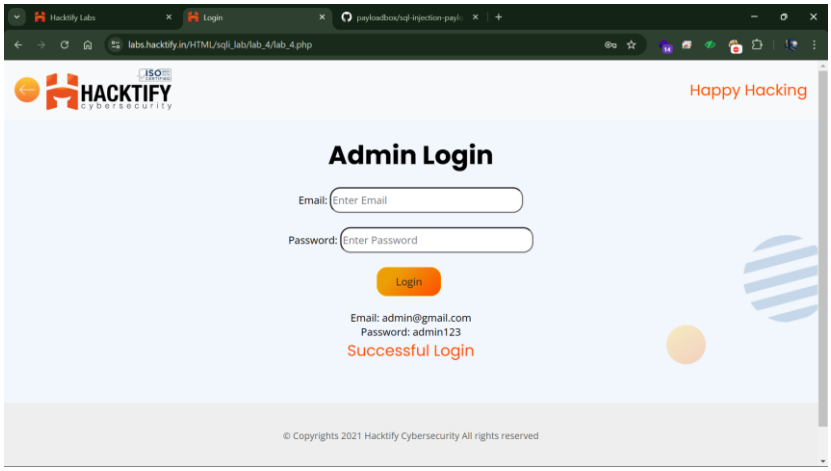


1.4 Let's Trick 'em!

Reference	Risk Rating
Let's Trick 'em!	**Medium**
Tools Used	
SQL Injection – SQL payloads	
Vulnerability Description	
SQL Injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It occurs when an application improperly handles user input, enabling malicious SQL statements to be injected and executed within the database. This can lead to unauthorized access, data manipulation, or even complete control over the database.	
How It Was Discovered	
Manual Analysis – SQL payloads	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_4/lab_4.php	
Consequences of not Fixing the Issue	
Failure to fix SQL Injection (SQLi) vulnerabilities can lead to severe consequences, including unauthorized data breaches, exposure of sensitive information (user credentials, financial data, personal records), and identity theft. Attackers can manipulate or delete critical data, bypass authentication, and gain administrative access to the database, potentially compromising the entire system. This can result in financial losses, legal penalties (due to GDPR, HIPAA, or PCI-DSS violations), reputational damage, and operational disruptions, ultimately endangering business continuity and user trust.	
Suggested Countermeasures	
To prevent SQL Injection (SQLi), use **prepared statements (parameterized queries)** and **stored procedures** to separate SQL code from user input. Implement **input validation** to restrict unexpected characters and enforce strict data types. Apply the **principle of least privilege (PoLP)** to limit database user permissions. Use **web application firewalls (WAFs)** to detect and block SQLi attempts. Regularly perform **security testing (penetration testing, code reviews, and automated scans)** to identify and remediate vulnerabilities before attackers exploit them.	
References	
https://github.com/payloadbox/sql-injection-payload-list	

Proof of Concept

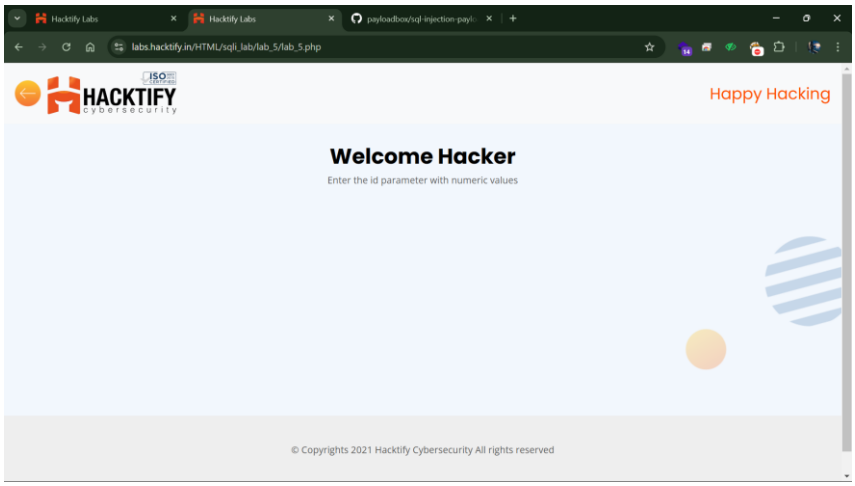


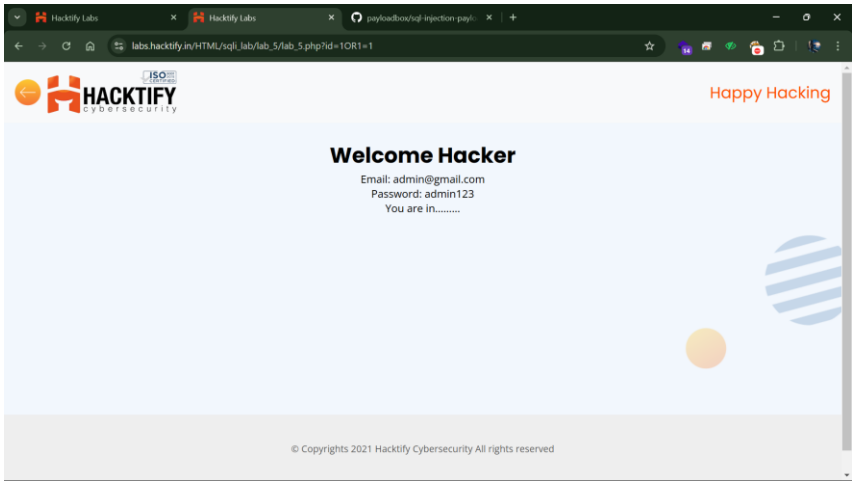


1.5 Booleans and Blind!

Reference	Risk Rating
Booleans and Blind!	**Medium**
Tools Used	
SQL Injection – SQL Payloads	
Vulnerability Description	
SQL Injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It occurs when an application improperly handles user input, enabling malicious SQL statements to be injected and executed within the database. This can lead to unauthorized access, data manipulation, or even complete control over the database.	
How It Was Discovered	
Manual Analysis – SQL Payloads :- by changing id parameter with numeric values.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_5/lab_5.php	
Consequences of not Fixing the Issue	
Failure to fix SQL Injection (SQLi) vulnerabilities can lead to severe consequences, including unauthorized data breaches, exposure of sensitive information (user credentials, financial data, personal records), and identity theft. Attackers can manipulate or delete critical data, bypass authentication, and gain administrative access to the database, potentially compromising the entire system. This can result in financial losses, legal penalties (due to GDPR, HIPAA, or PCI-DSS violations), reputational damage, and operational disruptions, ultimately endangering business continuity and user trust.	
Suggested Countermeasures	
To prevent SQL Injection (SQLi), use **prepared statements (parameterized queries)** and **stored procedures** to separate SQL code from user input. Implement **input validation** to restrict unexpected characters and enforce strict data types. Apply the **principle of least privilege (PoLP)** to limit database user permissions. Use **web application firewalls (WAFs)** to detect and block SQLi attempts. Regularly perform **security testing (penetration testing, code reviews, and automated scans)** to identify and remediate vulnerabilities before attackers exploit them.	
References	
https://github.com/payloadbox/sql-injection-payload-list	

Proof of Concept

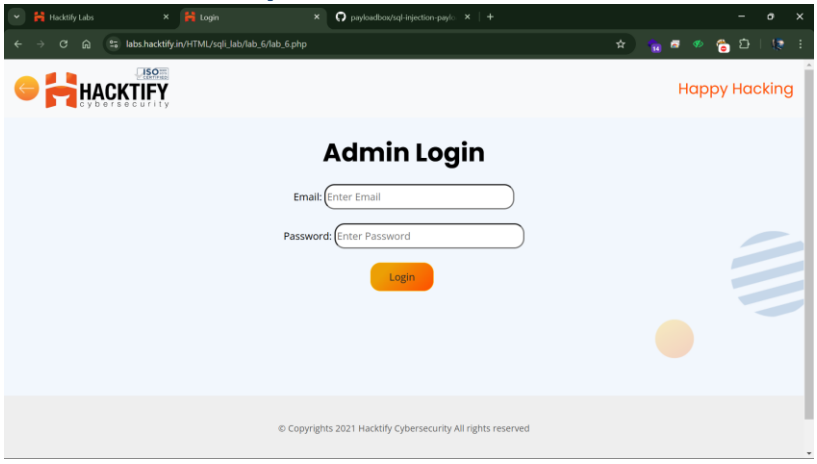


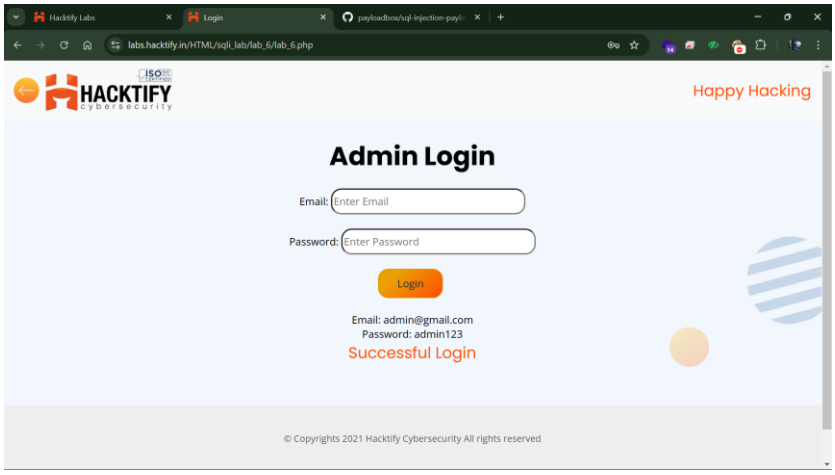


1.6 Error Based: Tricked

Reference	Risk Rating
Error Based: Tricked	**Medium**
Tools Used	
SQL Injection – SQL payloads	
Vulnerability Description	
SQL Injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It occurs when an application improperly handles user input, enabling malicious SQL statements to be injected and executed within the database. This can lead to unauthorized access, data manipulation, or even complete control over the database.	
How It Was Discovered	
Manual Analysis – SQL payloads	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_6/lab_6.php	
Consequences of not Fixing the Issue	
Failure to fix SQL Injection (SQLi) vulnerabilities can lead to severe consequences, including unauthorized data breaches, exposure of sensitive information (user credentials, financial data, personal records), and identity theft. Attackers can manipulate or delete critical data, bypass authentication, and gain administrative access to the database, potentially compromising the entire system. This can result in financial losses, legal penalties (due to GDPR, HIPAA, or PCI-DSS violations), reputational damage, and operational disruptions, ultimately endangering business continuity and user trust.	
Suggested Countermeasures	
To prevent SQL Injection (SQLi), use **prepared statements (parameterized queries)** and **stored procedures** to separate SQL code from user input. Implement **input validation** to restrict unexpected characters and enforce strict data types. Apply the **principle of least privilege (PoLP)** to limit database user permissions. Use **web application firewalls (WAFs)** to detect and block SQLi attempts. Regularly perform **security testing (penetration testing, code reviews, and automated scans)** to identify and remediate vulnerabilities before attackers exploit them.	
References	
https://github.com/payloadbox/sql-injection-payload-list	

Proof of Concept

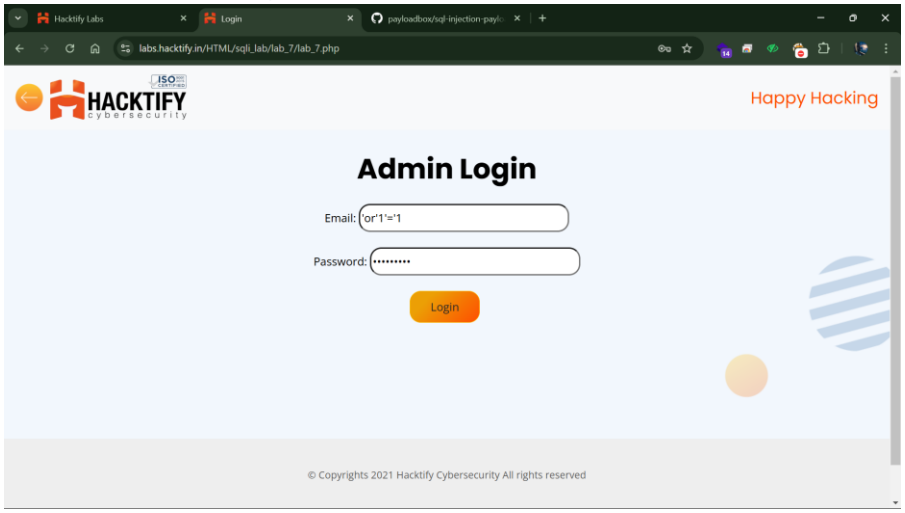


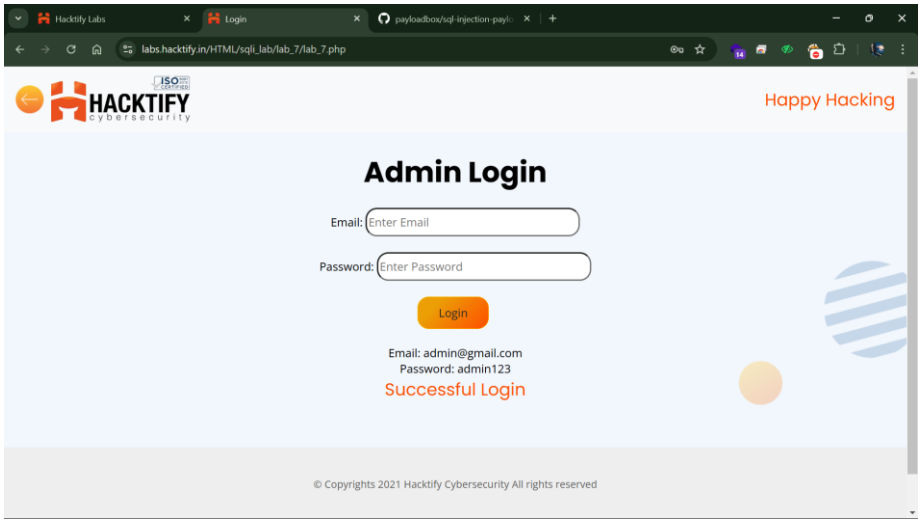


1.7 Errors and Post!

Reference	Risk Rating
Errors and Post!	**Low**
Tools Used	
SQL Injection – SQL payloads	
Vulnerability Description	
SQL Injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It occurs when an application improperly handles user input, enabling malicious SQL statements to be injected and executed within the database. This can lead to unauthorized access, data manipulation, or even complete control over the database.	
How It Was Discovered	
Manual Analysis – SQL payloads	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_7/lab_7.php	
Consequences of not Fixing the Issue	
Failure to fix SQL Injection (SQLi) vulnerabilities can lead to severe consequences, including unauthorized data breaches, exposure of sensitive information (user credentials, financial data, personal records), and identity theft. Attackers can manipulate or delete critical data, bypass authentication, and gain administrative access to the database, potentially compromising the entire system. This can result in financial losses, legal penalties (due to GDPR, HIPAA, or PCI-DSS violations), reputational damage, and operational disruptions, ultimately endangering business continuity and user trust.	
Suggested Countermeasures	
To prevent SQL Injection (SQLi), use **prepared statements (parameterized queries)** and **stored procedures** to separate SQL code from user input. Implement **input validation** to restrict unexpected characters and enforce strict data types. Apply the **principle of least privilege (PoLP)** to limit database user permissions. Use **web application firewalls (WAFs)** to detect and block SQLi attempts. Regularly perform **security testing (penetration testing, code reviews, and automated scans)** to identify and remediate vulnerabilities before attackers exploit them.	
References	
https://github.com/payloadbox/sql-injection-payload-list	

Proof of Concept

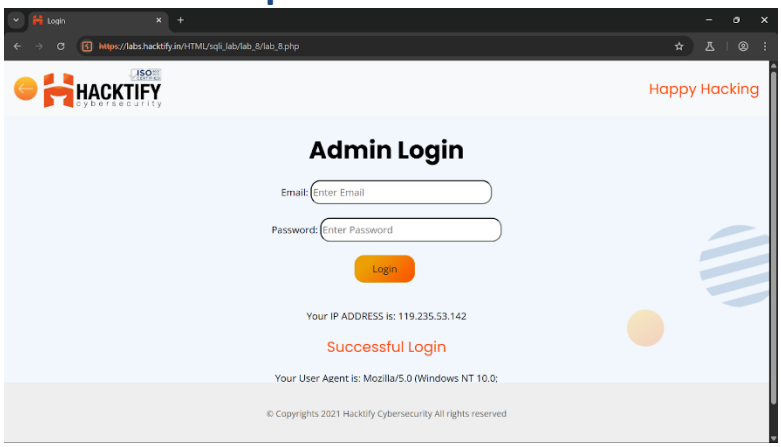


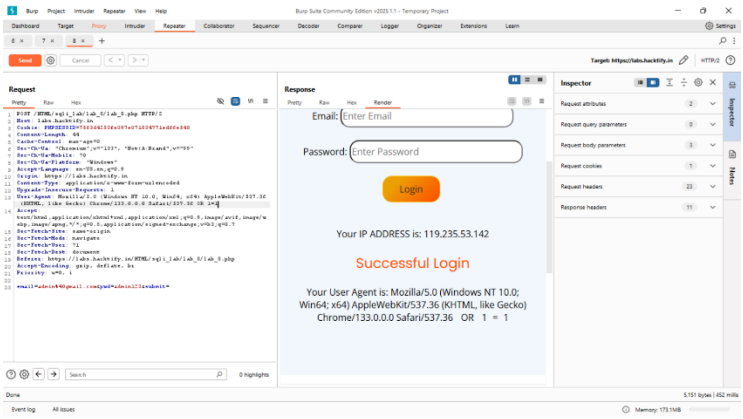


1.8 User Agents lead us!

Reference	Risk Rating
User Agents lead us!	**High**
Tools Used	
SQL payloads ,Burpsuite	
Vulnerability Description	
This type of **SQL Injection (SQLi)** occurs when a web application dynamically constructs SQL queries using **unsanitized user input**, allowing attackers to manipulate the logic of the query. By injecting a condition like **" OR "1"="1**, which always evaluates to **true**, an attacker can **bypass authentication**, access restricted data, or modify the application's behavior.	
How It Was Discovered	
Manual Analysis – SQL payloads and modifying the request.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_8/lab_8.php	
Consequences of not Fixing the Issue	
Failure to fix this **SQL Injection (SQLi) vulnerability** can lead to **authentication bypass**, allowing attackers to log in as **any user**, including administrators. This can result in **unauthorized access to sensitive data**, **data breaches**, **account takeovers**, and even **full system compromise**. Attackers may also **modify or delete records**, leading to **data loss** and **service disruptions**. Additionally, organizations risk **legal penalties** for failing to protect user data (e.g., GDPR, HIPAA) and **reputation damage**, which can result in financial losses and loss of customer trust.	
Suggested Countermeasures	
To prevent this **SQL Injection (SQLi) vulnerability**, use **prepared statements (parameterized queries)** to separate SQL logic from user input, ensuring inputs are treated as data rather than executable code. Implement **input validation** by restricting special characters and enforcing strict data types. Apply **least privilege principles** to database accounts, limiting access to only necessary operations. Use **Web Application Firewalls (WAFs)** to detect and block SQLi attempts in real time. Regularly conduct **security audits, penetration testing, and code reviews** to identify and patch vulnerabilities before attackers exploit them.	
References	
https://owasp.org/www-community/attacks/SQL_Injection	

Proof of Concept

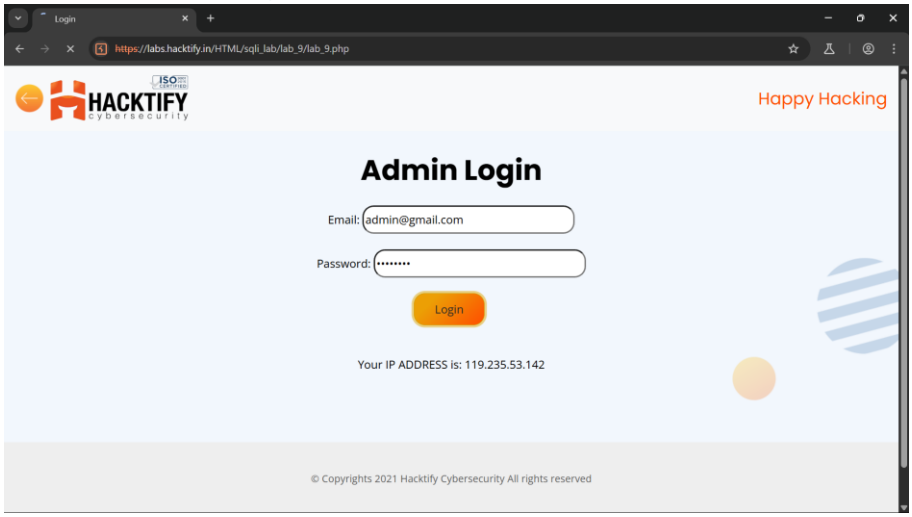


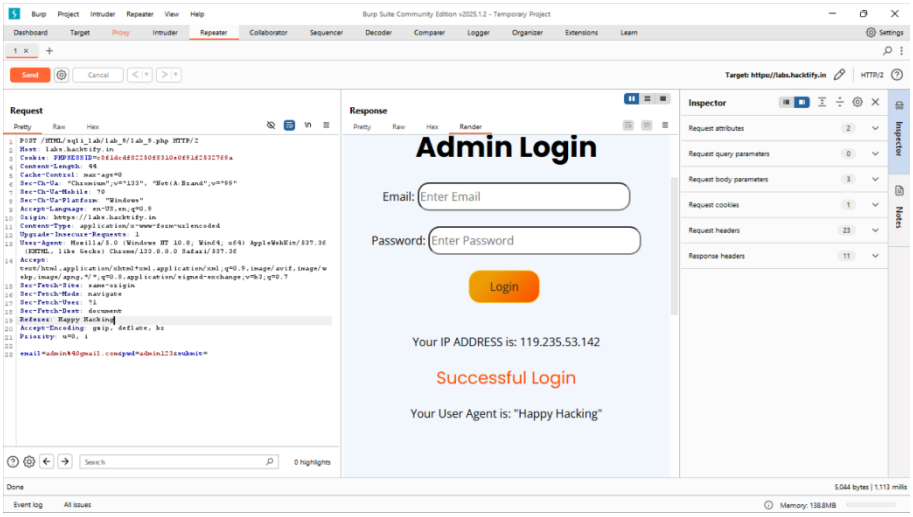


1.9 Referer lead us!

Reference	Risk Rating
User Agents lead us!	**Medium**
Tools Used	
SQL payloads ,Burpsuite	
Vulnerability Description	
This type of **SQL Injection (SQLi)** occurs when a web application dynamically constructs SQL queries using **unsanitized user input**, allowing attackers to manipulate the logic of the query. By injecting a condition like **"Happy Hacking"**, which always evaluates to **true**, an attacker can **bypass authentication**, access restricted data, or modify the application's behavior.	
How It Was Discovered	
Manual Analysis – SQL payloads and modifying the request.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_9/lab_9.php	
Consequences of not Fixing the Issue	
Failure to fix this **SQL Injection (SQLi) vulnerability** can lead to **authentication bypass**, allowing attackers to log in as **any user**, including administrators. This can result in **unauthorized access to sensitive data**, **data breaches**, **account takeovers**, and even **full system compromise**. Attackers may also **modify or delete records**, leading to **data loss** and **service disruptions**. Additionally, organizations risk **legal penalties** for failing to protect user data (e.g., GDPR, HIPAA) and **reputation damage**, which can result in financial losses and loss of customer trust.	
Suggested Countermeasures	
To prevent this **SQL Injection (SQLi) vulnerability**, use **prepared statements (parameterized queries)** to separate SQL logic from user input, ensuring inputs are treated as data rather than executable code. Implement **input validation** by restricting special characters and enforcing strict data types. Apply **least privilege principles** to database accounts, limiting access to only necessary operations. Use **Web Application Firewalls (WAFs)** to detect and block SQLi attempts in real time. Regularly conduct **security audits, penetration testing, and code reviews** to identify and patch vulnerabilities before attackers exploit them.	
References	
https://owasp.org/www-community/attacks/SQL_Injection	

Proof of Concept

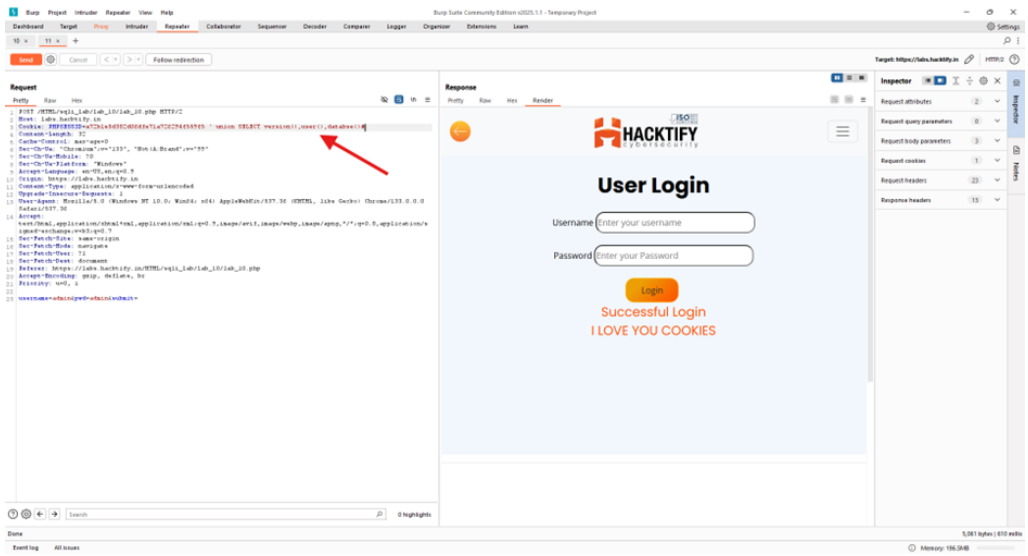




1.10 oh Cookies!

Reference	Risk Rating
Oh Cookies!	**High**
Tools Used	
SQL payloads ,Burpsuite	
Vulnerability Description	
SQL Injection via cookies occurs when user-controlled cookie values are directly used in database queries without proper validation. Attackers can modify the cookie to inject SQL payloads, allowing unauthorized access to database information. In this lab, modifying the `id` value in the cookie enabled retrieval of the database version, user, and name, demonstrating a critical security flaw.	
How It Was Discovered	
Manual Analysis – SQL payloads and modifying the request.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_10/lab_10.php	
Consequences of not Fixing the Issue	
Failing to fix this SQL Injection vulnerability can lead to data leaks, unauthorized access, and database manipulation. Attackers may escalate privileges, deface the website, or inject malware. Organizations could also face legal penalties for non-compliance with data protection regulations.	
Suggested Countermeasures	
To prevent this **SQL Injection (SQLi) vulnerability**, use **prepared statements (parameterized queries)** to separate SQL logic from user input, ensuring inputs are treated as data rather than executable code. Implement **input validation** by restricting special characters and enforcing strict data types. Apply **least privilege principles** to database accounts, limiting access to only necessary operations. Use **Web Application Firewalls (WAFs)** to detect and block SQLi attempts in real time. Regularly conduct **security audits, penetration testing, and code reviews** to identify and patch vulnerabilities before attackers exploit them.	
References	
https://owasp.org/www-project-cheat-sheets/	

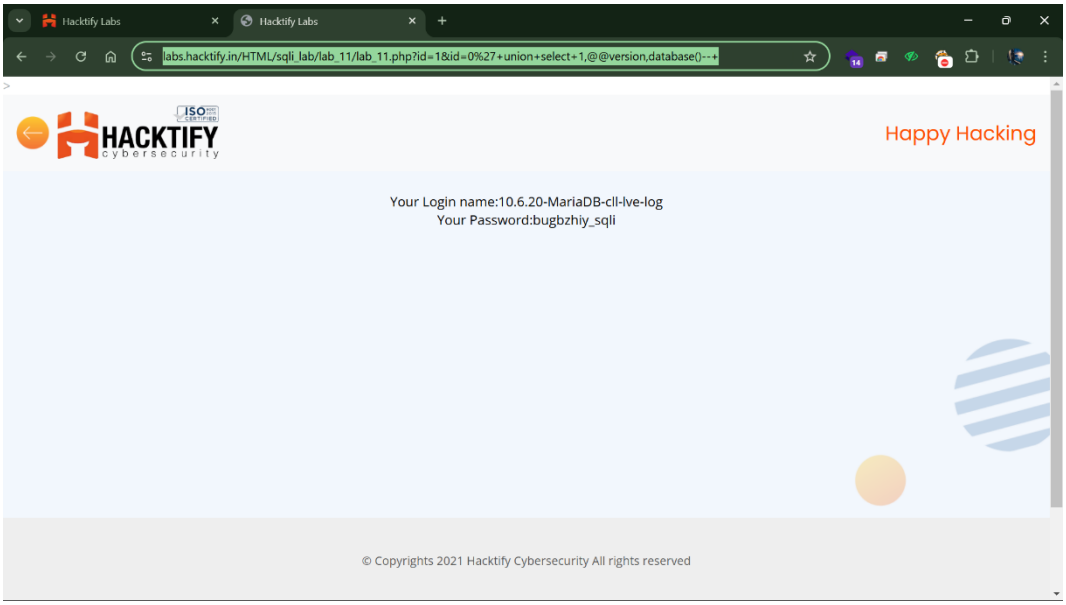
Proof of Concept



1.11 WAF's are Injected!

Reference	Risk Rating
WAF's are Injected!	**High**
Tools Used	
SQL payloads	
Vulnerability Description	
In this lab, the application is protected by a Web Application Firewall (WAF), but the underlying SQL Injection vulnerability still exists. By crafting a payload that bypasses the WAF, the attacker is able to extract sensitive data.	
How It Was Discovered	
Manual Analysis – SQL payloads and modifying the URL as **SQL payloads -- id=1&id=0' union select 1,@@version,database()--+**	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_11/lab_11.php	
Consequences of not Fixing the Issue	
If this vulnerability is left unaddressed, attackers could bypass security measures like the WAF and execute arbitrary SQL queries, potentially exposing sensitive information such as login credentials and system details. This may lead to unauthorized access, data manipulation, and full system compromise.	
Suggested Countermeasures	
To prevent SQL Injection and bypassing of security measures like WAFs, applications should implement parameterized queries and prepared statements to ensure user inputs are not directly executed in SQL queries. Input validation and sanitization should be enforced to filter and reject any malicious inputs. Additionally, WAF rules must be regularly updated and fine-tuned to detect advanced SQL Injection techniques. Developers should avoid exposing sensitive database information in error messages and restrict database privileges to minimize potential damage. Regular security audits and penetration testing should also be conducted to identify and fix vulnerabilities before attackers can exploit them.	
References	
https://www.cloudflare.com/learning/security/web-application-firewall-waf/	

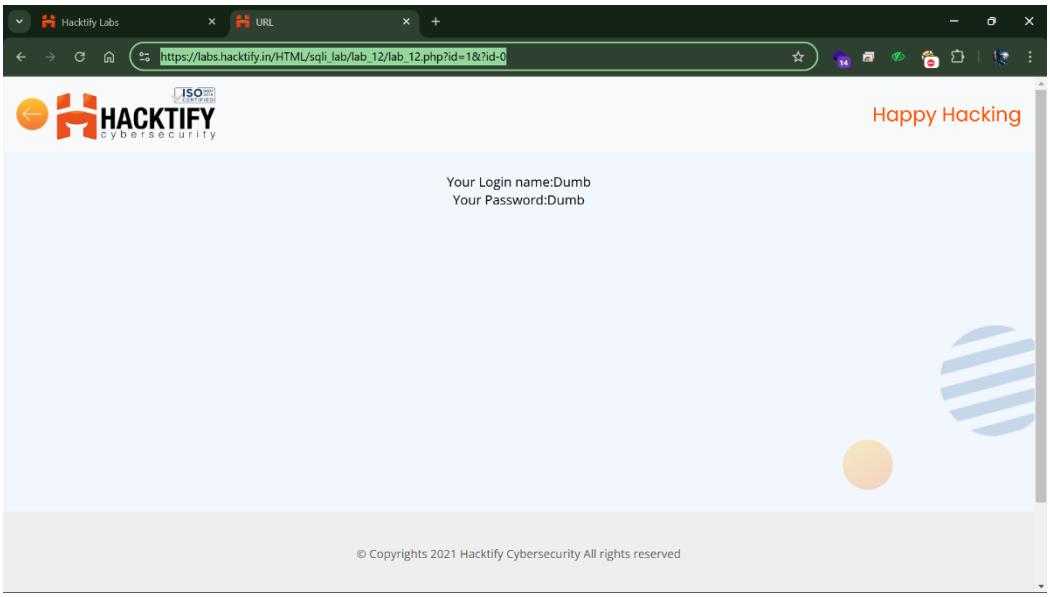
Proof of Concept



1.11 WAF's are Injected Part 2!

Reference	Risk Rating
WAF's are Injected!	**Medium**
Tools Used	
SQL payloads	
Vulnerability Description	
In this lab, the application is protected by a Web Application Firewall (WAF), but the underlying SQL Injection vulnerability still exists. By crafting a payload that bypasses the WAF, the attacker is able to extract sensitive data.	
How It Was Discovered	
Manual Analysis – SQL payloads and modifying the URL as **SQL payloads -id=1&?id-0**	
Vulnerable URLs	
https://labs.hacktify.in/HTML/sqli_lab/lab_12/lab_12.php	
Consequences of not Fixing the Issue	
If this vulnerability is left unaddressed, attackers could bypass security measures like the WAF and execute arbitrary SQL queries, potentially exposing sensitive information such as login credentials and system details. This may lead to unauthorized access, data manipulation, and full system compromise.	
Suggested Countermeasures	
To prevent SQL Injection and bypassing of security measures like WAFs, applications should implement parameterized queries and prepared statements to ensure user inputs are not directly executed in SQL queries. Input validation and sanitization should be enforced to filter and reject any malicious inputs. Additionally, WAF rules must be regularly updated and fine-tuned to detect advanced SQL Injection techniques. Developers should avoid exposing sensitive database information in error messages and restrict database privileges to minimize potential damage. Regular security audits and penetration testing should also be conducted to identify and fix vulnerabilities before attackers can exploit them.	
References	
https://www.cloudflare.com/learning/security/web-application-firewall-waf/	

Proof of Concept

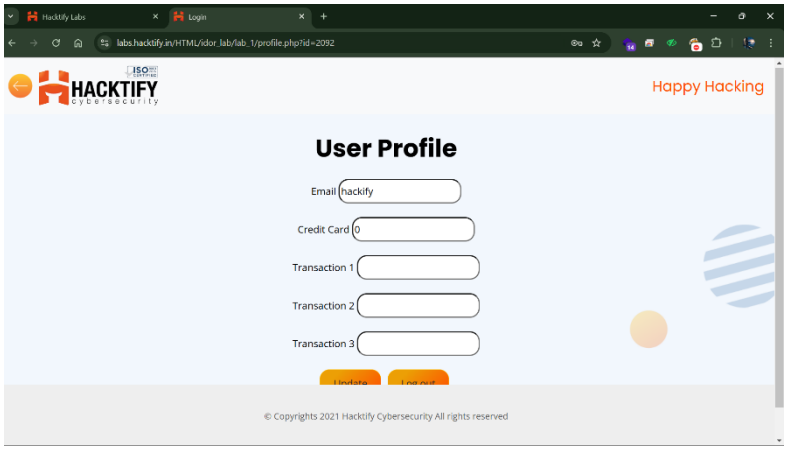


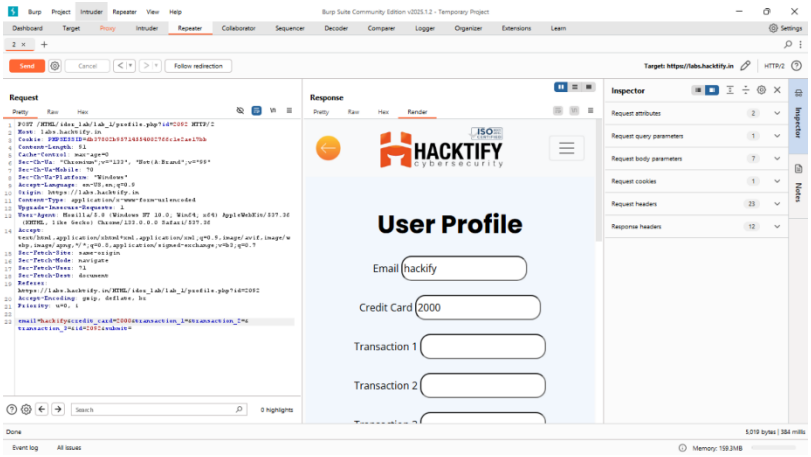
2. Insecure Direct Object References

2.1. Give me my amount!!

Reference	Risk Rating
Give me my amount!!	**Low**
Tools Used	
Burpsuite	
Vulnerability Description	
Insecure Direct Object Reference (IDOR) is a security vulnerability that occurs when an application provides direct access to objects (e.g., database records, files, or URLs) based on user-supplied input without proper authorization checks. This can allow attackers to manipulate parameters and gain unauthorized access or modify data.	
How It Was Discovered	
Automated Tools – Burpsuite: by modifying the request.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_1/profile.php?id=2092	
Consequences of not Fixing the Issue	
If an IDOR vulnerability allowing transaction modification is not fixed, it can lead to severe financial losses, as attackers may exploit it to purchase products or services at reduced prices, receive inflated refunds, or manipulate account balances. This can result in revenue loss, legal liabilities, and regulatory penalties for failing to secure financial transactions. Additionally, businesses may suffer reputational damage, loss of customer trust, and potential exploitation by fraudsters. If left unaddressed, attackers can automate the exploit, causing widespread abuse, impacting the company's financial stability, and leading to severe security breaches.	
Suggested Countermeasures	
To prevent IDOR vulnerabilities, implement **server-side validation** to ensure transaction amounts cannot be modified by users, enforce **strict access controls** to verify user permissions, and use **indirect object references** (e.g., mapping IDs instead of exposing them directly). Additionally, log and monitor suspicious activities, apply **input validation**, and conduct **regular security audits** to detect and fix vulnerabilities before exploitation.	
References	
https://owasp.org/www-community/attacks/Indirect_Object_Reference_Map	

Proof of Concept

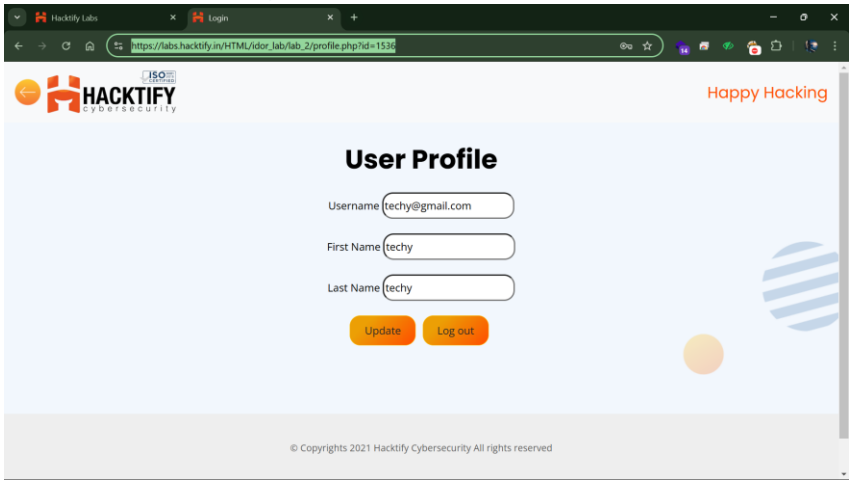


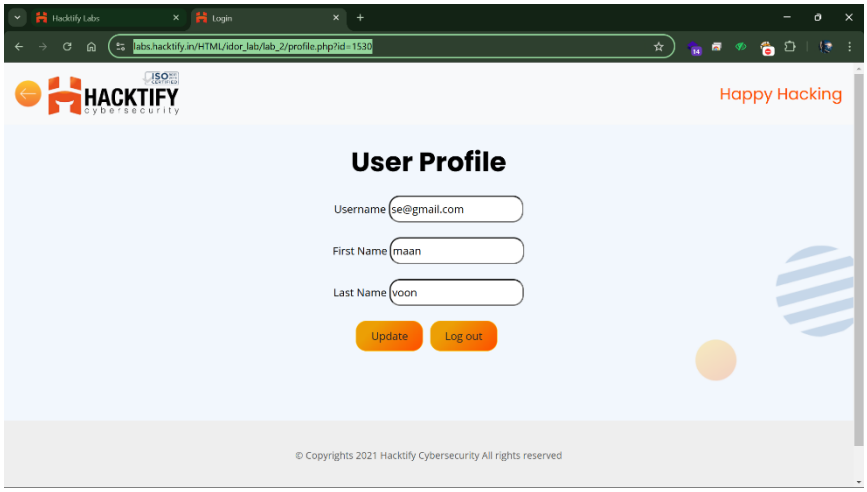


2.2 Stop Polluting my Params!

Reference	Risk Rating
Stop Polluting my Params!	**Medium**
Tools Used	
SQL Payloads	
Vulnerability Description	
The vulnerability in this scenario is **Insecure Direct Object Reference (IDOR)**, where an attacker can access other users' names by modifying the **ID parameter** in the request. This occurs because the application directly references user records based on a numerical ID without proper authorization checks. By incrementing or decrementing the ID, an attacker can enumerate and retrieve other users' personal details, leading to **privacy violations, unauthorized data access, and potential identity theft**. This flaw stems from **improper access control** and can be mitigated by implementing **proper authentication, role-based access control (RBAC), and object-level authorization checks**.	
How It Was Discovered	
Manual Analysis – By Modifying the id number.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_2/lab_2.php	
Consequences of not Fixing the Issue	
If an IDOR vulnerability exposing usernames is not fixed, attackers can enumerate and access other users' personal information, leading to **privacy breaches, identity theft, and unauthorized data exposure**. This can result in **compliance violations** (e.g., GDPR, CCPA), legal consequences, and **reputational damage** for the organization. Additionally, attackers may exploit this information for **social engineering attacks, phishing, or credential stuffing**, further compromising user security.	
Suggested Countermeasures	
To mitigate IDOR vulnerabilities, implement **proper access controls** by enforcing **object-level authorization checks** to ensure users can only access their own data. Use **indirect object references** (e.g., UUIDs instead of sequential IDs) to prevent enumeration, and apply **server-side validation** to verify user permissions before serving requested data. Additionally, enable **logging and monitoring** to detect unauthorized access attempts, conduct **regular security audits and penetration testing**, and follow **least privilege principles** to minimize data exposure risks.	
References	
https://owasp.org/Top10/A01_2021-Broken_Access_Control/	

Proof of Concept

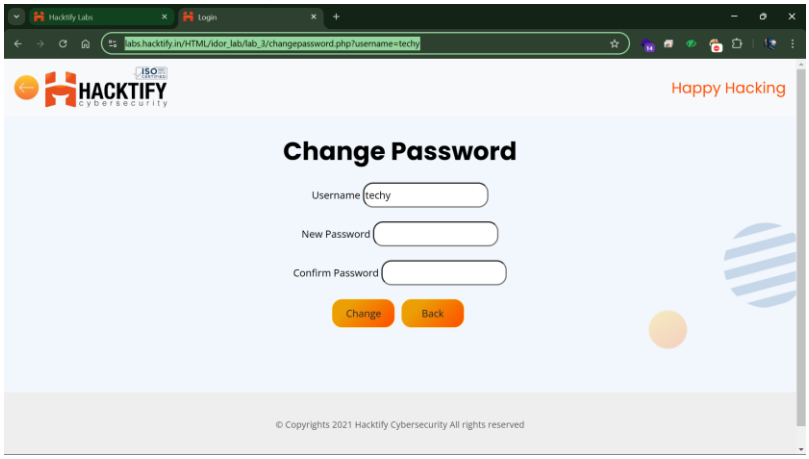


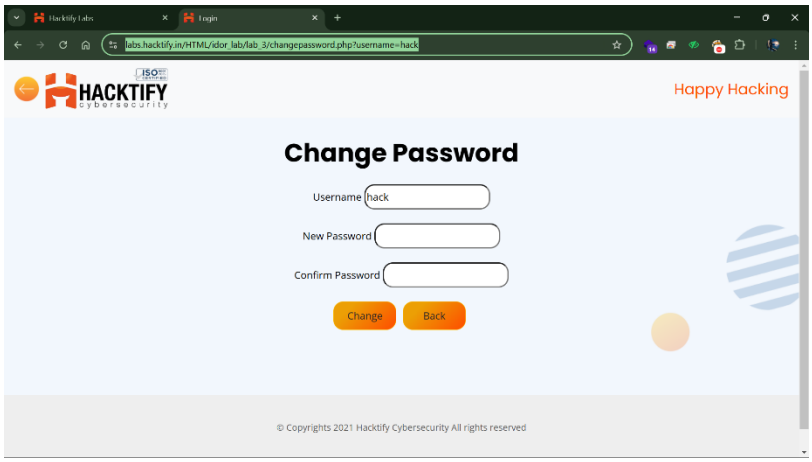


2.3 Someone Changed my Password!

Reference	Risk Rating
Someone Changed my Password!	**Low**
Tools Used	
URL Inspecter	
Vulnerability Description	
This vulnerability is a **Broken Access Control (IDOR in Password Reset Functionality)** issue, where the application allows users to change their password but fails to enforce proper authorization checks. By modifying the **username parameter in the URL**, an attacker can reset another user's password without authentication, leading to **account takeover**. This occurs due to **lack of user session validation and improper access controls** when handling password change requests. Exploiting this flaw allows an attacker to lock users out of their accounts, gain unauthorized access, and potentially escalate privileges within the system.	
How It Was Discovered	
Manual Analysis – By Modifying the usernames.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_3/lab_3.php	
Consequences of not Fixing the Issue	
If this **Broken Access Control (IDOR in Password Reset Functionality)** vulnerability is not fixed, attackers can **take over user accounts** by changing their passwords, leading to **unauthorized access, data theft, and identity fraud**. This can result in **loss of sensitive user data, financial fraud, reputational damage, and legal consequences** due to non-compliance with security regulations (e.g., GDPR, CCPA). Additionally, if administrative accounts are compromised, attackers may escalate privileges, potentially gaining **full control over the system**, leading to **widespread data breaches and service disruption**.	
Suggested Countermeasures	
To mitigate this **Broken Access Control (IDOR in Password Reset Functionality)** vulnerability, implement **strict authorization checks** to ensure users can only update their own passwords. Enforce **session-based authentication** and verify the requesting user's identity before processing password changes. Use **server-side validation** to reject unauthorized modifications to usernames in the URL or request body. Implement **multi-factor authentication (MFA)** to prevent unauthorized access even if passwords are compromised. Additionally, log and monitor password change attempts for **suspicious activity**, and conduct **regular security audits and penetration testing** to detect and fix such vulnerabilities.	
References	
https://owasp.org/Top10/A01_2021-Broken_Access_Control/	

Proof of Concept

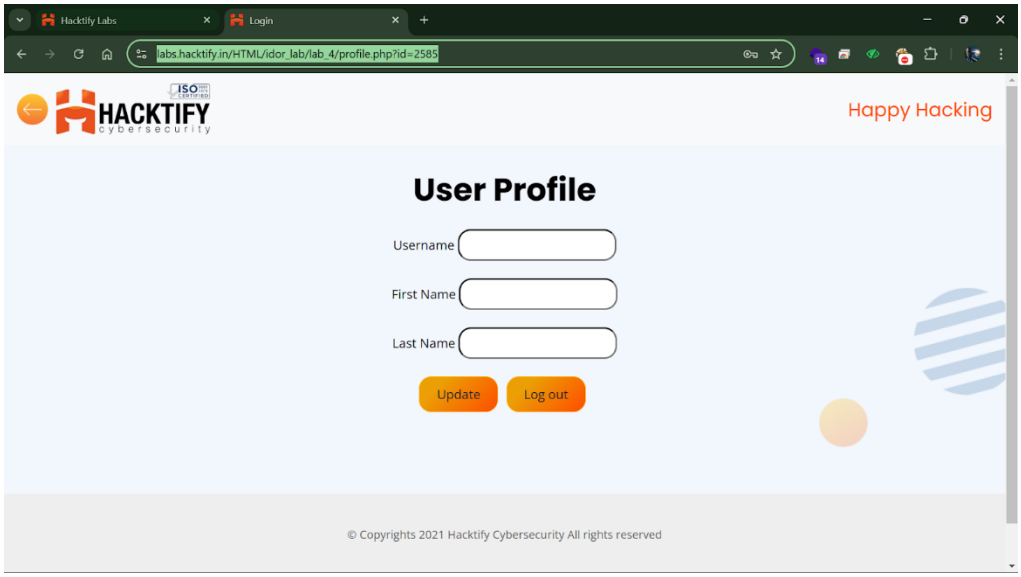


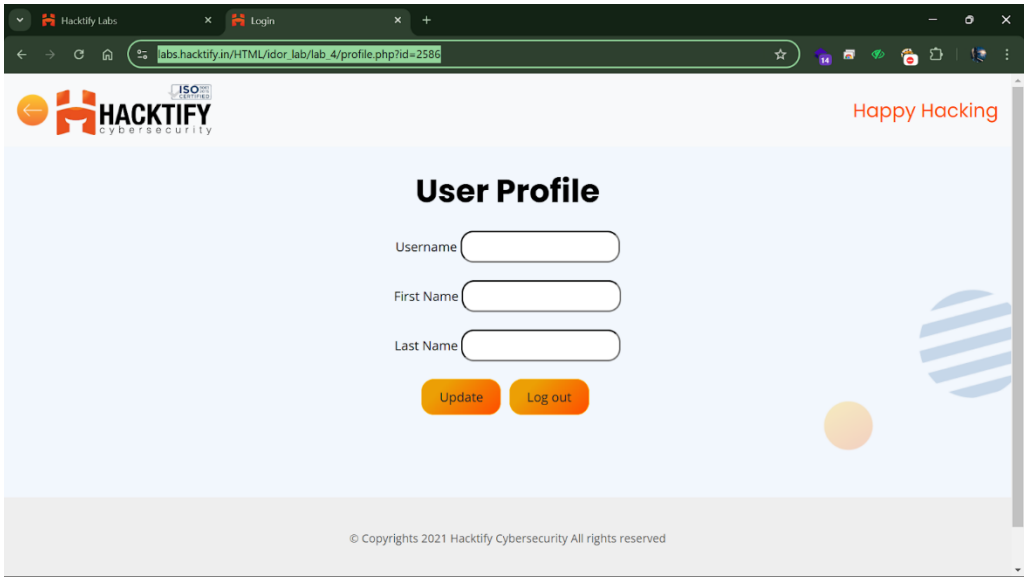


2.4 Change your Methods!

Reference	Risk Rating
Change your Methods!	**Medium**
Tools Used	
URL Inspector	
Vulnerability Description	
This vulnerability is an **Insecure Direct Object Reference (IDOR)** issue, where the application allows users to update their **first name and last name** by modifying the **ID parameter in the URL** without proper authorization checks. Since the server does not validate whether the user has permission to edit another user's profile, an attacker can change anyone's personal details by simply altering the **user ID**. This flaw results from **lack of access controls at the object level** and can lead to **identity manipulation, data integrity issues, and unauthorized profile modifications**.	
How It Was Discovered	
Manual Analysis – By Modifying the usernames.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/idor_lab/lab_4/lab_4.php	
Consequences of not Fixing the Issue	
If this **IDOR vulnerability** is not fixed, attackers can modify other users' personal details, leading to **identity manipulation, reputational damage, and data integrity issues**. Malicious actors could impersonate users, change sensitive information, or disrupt business operations by altering multiple accounts. In cases involving **regulated data (e.g., GDPR, HIPAA, CCPA)**, this can result in **legal penalties, compliance violations, and loss of user trust**. Additionally, attackers may combine this with social engineering or phishing to **escalate attacks, commit fraud, or bypass authentication mechanisms**.	
Suggested Countermeasures	
To mitigate this **IDOR vulnerability**, implement **proper access controls** by ensuring that users can only modify their own profiles through **server-side authorization checks**. Use **session-based authentication** to verify user identity before processing updates, and avoid exposing **user IDs in URLs** by using **indirect references (e.g., UUIDs or tokens)**. Implement **role-based access control (RBAC)** to restrict profile modifications, enforce **input validation and logging**, and conduct **regular security audits and penetration testing** to detect and fix such vulnerabilities proactively.	
References	
https://owasp.org/Top10/A01_2021-Broken_Access_Control/	

Proof of Concept





Penetration Testing Report

Full Name: AVIREDDY RUSHIKESH
Program: HCPT
Date:02/03/2025

Introduction

This report document hereby describes the proceedings and results of a Black Box security assessment conducted against the **Week {3} Labs**. The report hereby lists the findings and corresponding best practice mitigation actions and recommendations.

1. Objective

The objective of the assessment was to uncover vulnerabilities in the **Week {3} Labs** 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.

2. Scope

This section defines the scope and boundaries of the project.

Application Name	**Lab 1 Name – Cross-Site Request Forgery , Lab 2 Name – Cross Origin Resource Sharing**

3. Summary

Outlined is a Black Box Application Security assessment for the **Week {3} Labs**.

Total number of Sub-labs: 13

High	Medium	Low
5	3	5

High - **5**

Medium - **3**

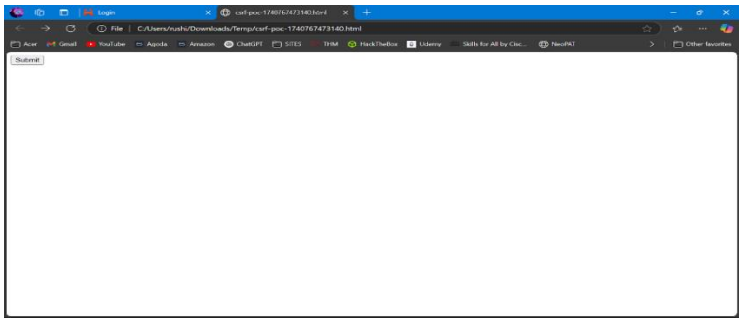
Low - 5

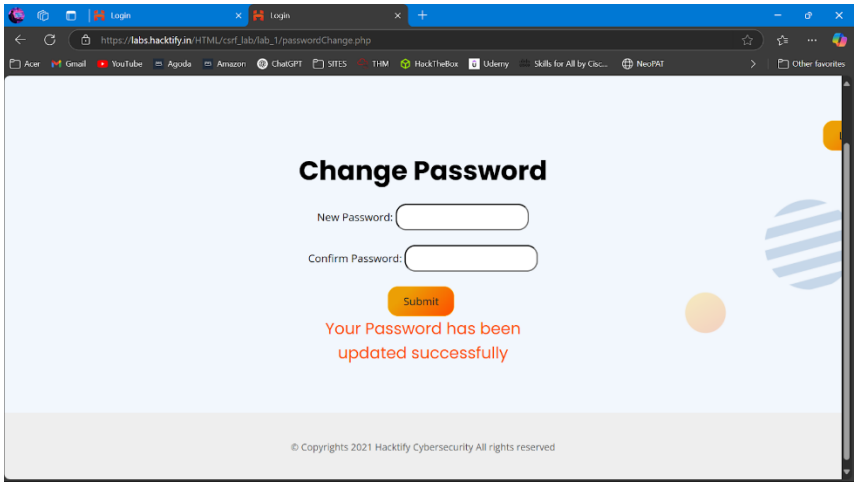
1. Cross – Site Request Forgery

1.1. Eassyy CSRF

Reference	Risk Rating
Eassyy CSRF	**Low**
Tools Used	
Hacktify CSRF Poc Generator	
Vulnerability Description	
Cross-Site Request Forgery (CSRF) is a vulnerability that allows an attacker to trick a logged-in user into unknowingly executing **unauthorized actions** on a web application. This occurs when the application **fails to verify the origin of requests**, allowing an attacker to embed malicious requests in a **link, form, or script** that executes with the victim's **session credentials**. If exploited, CSRF can lead to **account takeover, data modification, fund transfers, or privilege escalation** without the victim's consent. Proper **CSRF protection tokens** and **same-site cookie policies** are essential to prevent this attack.	
How It Was Discovered	
Automated Tools – By CSRF Poc Generator	
Vulnerable URLs	
https://labs.hacktify.in/HTML/csrf_lab/lab_1/index.php	
Consequences of not Fixing the Issue	
If a **CSRF vulnerability** is not fixed, attackers can force users to perform unintended actions like **changing account details, transferring funds, or deleting data** without their consent. This can lead to **account hijacking, financial fraud, and data loss**, causing **reputational damage and regulatory non-compliance**. In critical cases, attackers may **gain full control over user accounts or administrative functions**, leading to **mass exploitation and security breaches**.	
Suggested Countermeasures	
To prevent **CSRF attacks**, implement **CSRF tokens** in all state-changing requests to verify legitimate user actions. Use **SameSite cookies** to restrict cookie transmission across sites and enforce **user re-authentication** for sensitive actions. Implement **CORS policies** to limit cross-origin requests and use **HTTP headers like Referer and Origin** to validate request sources. Additionally, educate users to avoid clicking on untrusted links while logged in and conduct **regular security audits** to detect vulnerabilities.	
References	
https://owasp.org/www-community/attacks/csrf	

Proof of Concept

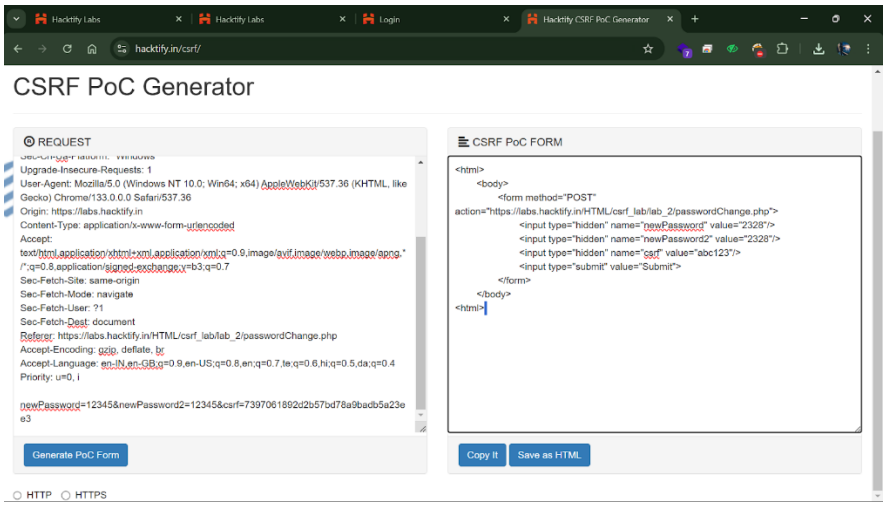


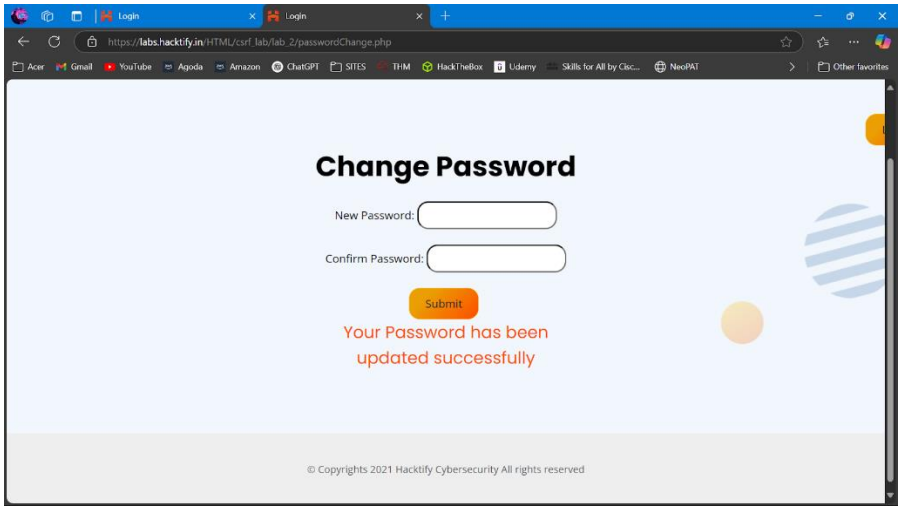


1.2. Always Validate Tokens!

Reference	Risk Rating
Always Validate Tokens	**Medium**
Tools Used	
Hacktify CSRF Poc Generator	
Vulnerability Description	
Cross-Site Request Forgery (CSRF) occurs when an attacker tricks a logged-in user into unknowingly making a **malicious request** that changes a security token, such as an **API key, authentication token, or session token**, without their consent. This happens due to the application **failing to validate request origins** and relying solely on **session-based authentication**. If exploited, an attacker can **change security tokens** linked to the victim's account, leading to **unauthorized access, privilege escalation, or account takeover**. Implementing **CSRF tokens, SameSite cookie attributes, and user re-authentication** can prevent such attacks.	
How It Was Discovered	
Automated Tools – By CSRF Poc Generator	
Vulnerable URLs	
https://labs.hacktify.in/HTML/csrf_lab/lab_2/login.php	
Consequences of not Fixing the Issue	
If a **CSRF vulnerability** is not fixed, attackers can force users to perform unintended actions like **changing account details, transferring funds, or deleting data** without their consent. This can lead to **account hijacking, financial fraud, and data loss**, causing **reputational damage and regulatory non-compliance**. In critical cases, attackers may **gain full control over user accounts or administrative functions**, leading to **mass exploitation and security breaches**.	
Suggested Countermeasures	
To prevent **CSRF attacks**, implement **CSRF tokens** in all state-changing requests to verify legitimate user actions. Use **SameSite cookies** to restrict cookie transmission across sites and enforce **user re-authentication** for sensitive actions. Implement **CORS policies** to limit cross-origin requests and use **HTTP headers like Referer and Origin** to validate request sources. Additionally, educate users to avoid clicking on untrusted links while logged in and conduct **regular security audits** to detect vulnerabilities.	
References	
https://owasp.org/www-community/attacks/csrf	

Proof of Concept

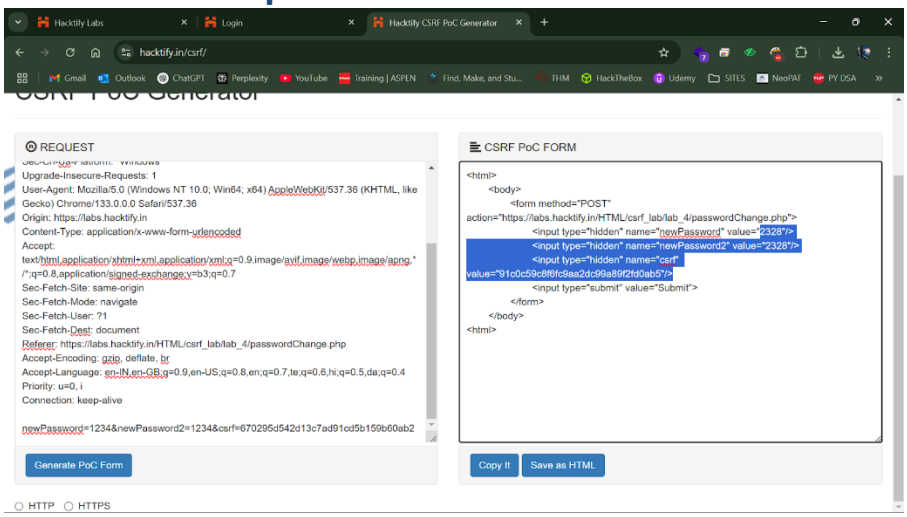


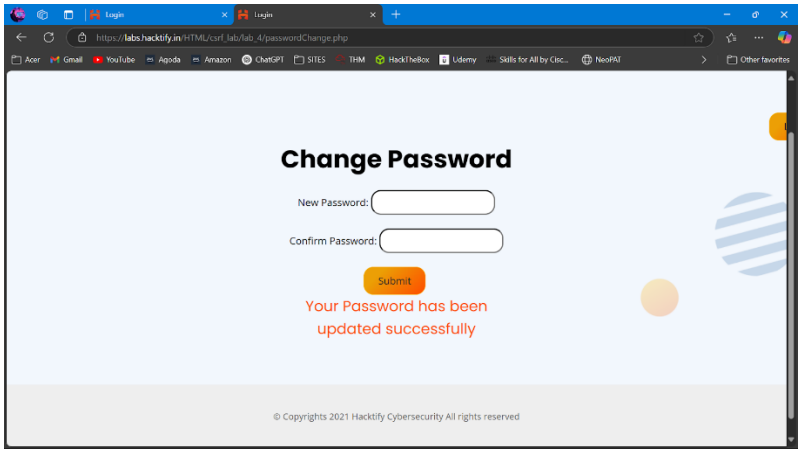


1.3. I hate when someone uses my Tokens!

Reference	Risk Rating
I hate when someone uses my Tokens!	**Low**
Tools Used	
Hacktify CSRF Poc Generator	
Vulnerability Description	
Cross-Site Request Forgery (CSRF) occurs when an attacker tricks a logged-in user into unknowingly making a **malicious request** that changes a security token, such as an **API key, authentication token, or session token**, without their consent. This happens due to the application **failing to validate request origins** and relying solely on **session-based authentication**. If exploited, an attacker can **change security tokens** linked to the victim's account, leading to **unauthorized access, privilege escalation, or account takeover**. Implementing **CSRF tokens, SameSite cookie attributes, and user re-authentication** can prevent such attacks.	
How It Was Discovered	
Automated Tools – By CSRF Poc Generator	
Vulnerable URLs	
https://labs.hacktify.in/HTML/csrf_lab/lab_4/login.php	
Consequences of not Fixing the Issue	
If a **CSRF vulnerability** is not fixed, attackers can force users to perform unintended actions like **changing account details, transferring funds, or deleting data** without their consent. This can lead to **account hijacking, financial fraud, and data loss**, causing **reputational damage and regulatory non-compliance**. In critical cases, attackers may **gain full control over user accounts or administrative functions**, leading to **mass exploitation and security breaches**.	
Suggested Countermeasures	
To prevent **CSRF attacks**, implement **CSRF tokens** in all state-changing requests to verify legitimate user actions. Use **SameSite cookies** to restrict cookie transmission across sites and enforce **user re-authentication** for sensitive actions. Implement **CORS policies** to limit cross-origin requests and use **HTTP headers like Referer and Origin** to validate request sources. Additionally, educate users to avoid clicking on untrusted links while logged in and conduct **regular security audits** to detect vulnerabilities.	
References	
https://owasp.org/www-community/attacks/csrf	

Proof of Concept

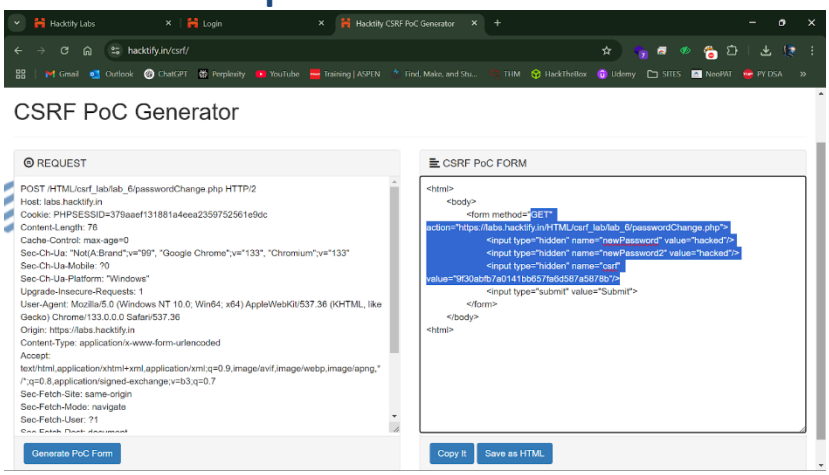


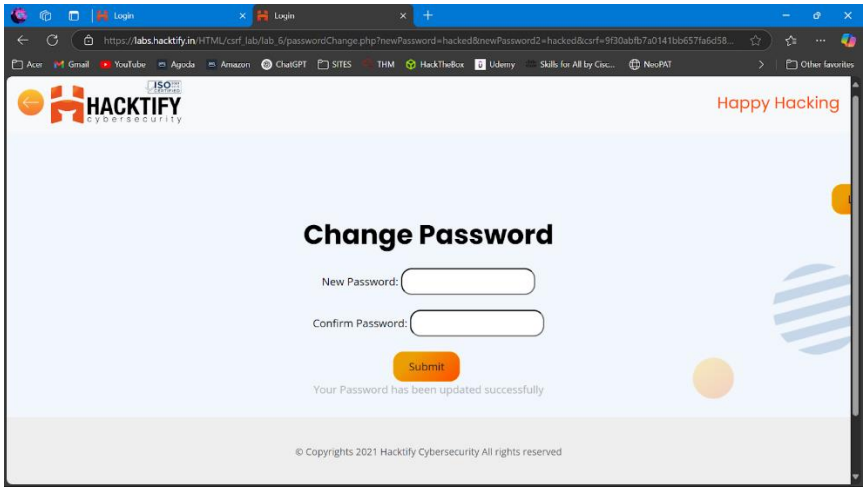


1.4. GET Me or POST ME!

Reference	Risk Rating
GET Me or POST ME!	**Low**
Tools Used	
Hacktify CSRF Poc Generator	
Vulnerability Description	
Cross-Site Request Forgery (CSRF) exploits the lack of proper request validation, allowing an attacker to force a victim's browser to perform unauthorized actions. If a web application accepts sensitive actions via both GET and POST requests, an attacker can convert a POST request to GET and embed it in an external link or image, causing unintended changes when the victim visits a malicious page. This can lead to account modifications, privilege escalation, or data exposure. Implementing strict request methods, CSRF tokens, and SameSite cookie policies can prevent such attacks.	
How It Was Discovered	
Automated Tools – By CSRF Poc Generator	
Vulnerable URLs	
https://labs.hacktify.in/HTML/csrf_lab/lab_6/login.php	
Consequences of not Fixing the Issue	
If this **CSRF vulnerability** (changing POST to GET) is not fixed, attackers can exploit it to perform **unauthorized actions** on behalf of authenticated users without their knowledge. This can lead to **account takeovers, unauthorized data modifications, financial fraud, or privilege escalations**. Since **GET requests can be triggered automatically** (e.g., by loading an image or clicking a malicious link), attackers can easily exploit users just by making them visit a compromised webpage. Failure to enforce **proper request validation** can result in **data breaches, reputational damage, and compliance violations** for the affected organization.	
Suggested Countermeasures	
To prevent this **CSRF vulnerability (changing POST to GET)**, enforce **strict request methods** by ensuring sensitive actions **only accept POST requests**. Implement **CSRF tokens** for all state-changing operations and validate them on the server side. Use **SameSite cookie attributes (Lax or Strict)** to prevent cross-origin requests from automatically sending session cookies. Additionally, require **re-authentication or multi-factor authentication (MFA)** for critical actions like password or email changes. Regular **security audits and penetration testing** can help identify and mitigate such vulnerabilities before they are exploited.	
References	
https://owasp.org/www-community/attacks/csrf	

Proof of Concept

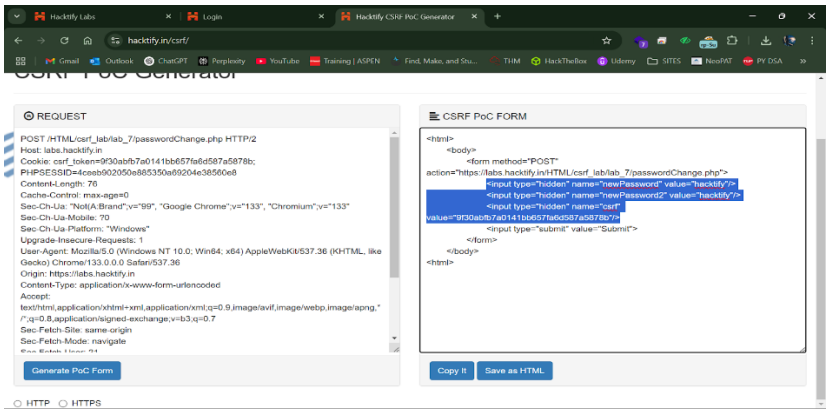


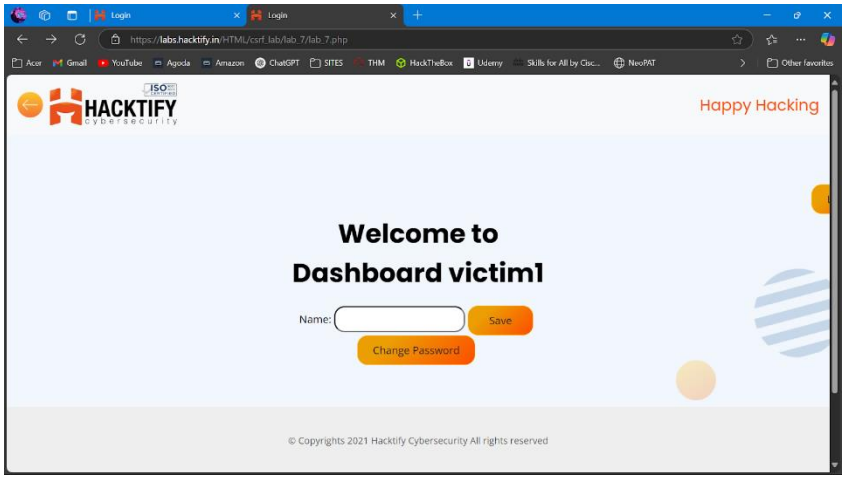


1.5. XSS the Saviour!

Reference	Risk Rating
XSS the Saviour!	**High**
Tools Used	
Burpsuite , Hacktify CSRF Poc Generator	
Vulnerability Description	
Cross-Site Request Forgery is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering an attacker may trick the users of a web application into executing actions of the attacker's choosing. Cross-Site Request Forgery is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering an attacker may trick the users of a web application into executing actions of the attacker's choosing.	
How It Was Discovered	
Automated Tools – By CSRF Poc Generator	
Vulnerable URLs	
https://labs.hacktify.in/HTML/csrf_lab/lab_7/login.php	
Consequences of not Fixing the Issue	
If a **CSRF vulnerability** is not fixed, attackers can force users to perform unintended actions like **changing account details, transferring funds, or deleting data** without their consent. This can lead to **account hijacking, financial fraud, and data loss**, causing **reputational damage and regulatory non-compliance**. In critical cases, attackers may **gain full control over user accounts or administrative functions**, leading to **mass exploitation and security breaches**.	
Suggested Countermeasures	
To prevent **CSRF attacks**, implement **CSRF tokens** in all state-changing requests to verify legitimate user actions. Use **SameSite cookies** to restrict cookie transmission across sites and enforce **user re-authentication** for sensitive actions. Implement **CORS policies** to limit cross-origin requests and use **HTTP headers like Referer and Origin** to validate request sources. Additionally, educate users to avoid clicking on untrusted links while logged in and conduct **regular security audits** to detect vulnerabilities.Additionally, require **re-authentication or multi-factor authentication (MFA)** for critical actions like password or email changes. Regular **security audits and penetration testing** can help identify and mitigate such vulnerabilities before they are exploited.	
References	
https://portswigger.net/web-security/csrf	

Proof of Concept

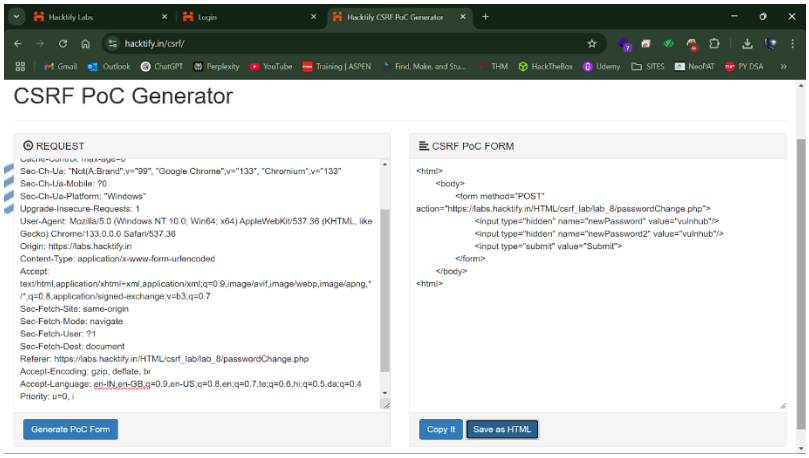


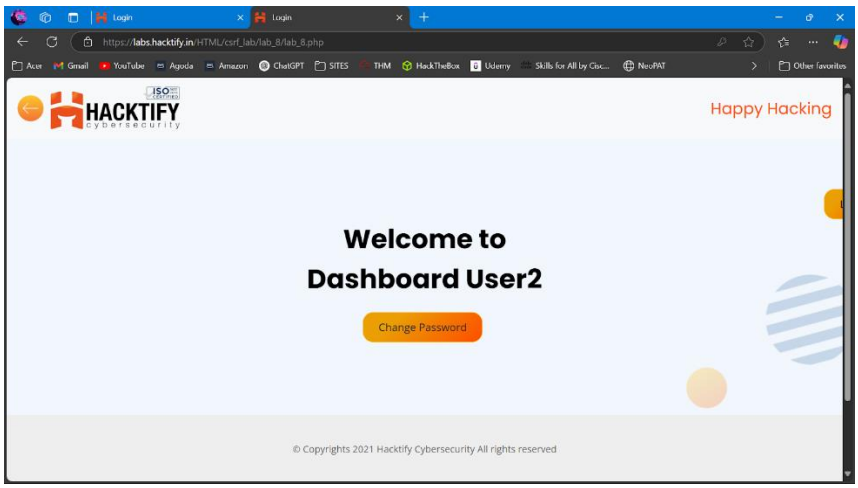


1.6. rm -rf token

Reference	Risk Rating
rm -rf token	**High**
Tools Used	
Burpsuite , Hacktify CSRF Poc Generator	
Vulnerability Description	
Cross-Site Request Forgery is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering an attacker may trick the users of a web application into executing actions of the attacker's choosing. Cross-Site Request Forgery is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering an attacker may trick the users of a web application into executing actions of the attacker's choosing.	
How It Was Discovered	
Automated Tools – By CSRF Poc Generator	
Vulnerable URLs	
https://labs.hacktify.in/HTML/csrf_lab/lab_8/login.php	
Consequences of not Fixing the Issue	
If a **CSRF vulnerability** is not fixed, attackers can force users to perform unintended actions like **changing account details, transferring funds, or deleting data** without their consent. This can lead to **account hijacking, financial fraud, and data loss**, causing **reputational damage and regulatory non-compliance**. In critical cases, attackers may **gain full control over user accounts or administrative functions**, leading to **mass exploitation and security breaches**.	
Suggested Countermeasures	
To prevent **CSRF attacks**, implement **CSRF tokens** in all state-changing requests to verify legitimate user actions. Use **SameSite cookies** to restrict cookie transmission across sites and enforce **user re-authentication** for sensitive actions. Implement **CORS policies** to limit cross-origin requests and use **HTTP headers like Referer and Origin** to validate request sources. Additionally, educate users to avoid clicking on untrusted links while logged in and conduct **regular security audits** to detect vulnerabilities.Additionally, require **re-authentication or multi-factor authentication (MFA)** for critical actions like password or email changes. Regular **security audits and penetration testing** can help identify and mitigate such vulnerabilities before they are exploited.	
References	
https://www.invicti.com/learn/cross-site-request-forgery-csrf/	

Proof of Concept



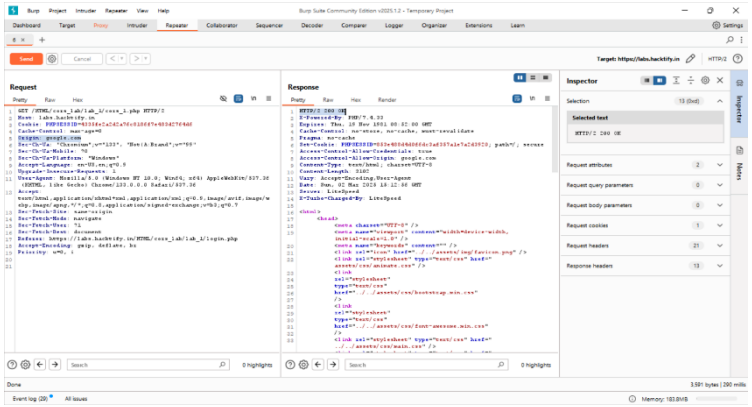


2. Cross-Origin Resource Sharing

2.1. CORS with arbitrary Origin

Reference	Risk Rating
CORS with arbitrary Origin	**Low**
Tools Used	
Burpsuite	
Vulnerability Description	
A **Cross-Origin Resource Sharing (CORS) vulnerability** occurs when a web application improperly configures its CORS policy, allowing **unauthorized origins** to access sensitive resources. If an application allows requests from **any origin (Access-Control-Allow-Origin: *) or whitelists untrusted domains**, attackers can exploit this to steal **user data, hijack sessions, or perform API abuse** via malicious websites. This is especially dangerous if **authentication credentials (cookies, tokens) are exposed to untrusted origins**, leading to **data breaches and unauthorized actions**. Properly **restricting allowed origins and methods** is crucial for preventing CORS-based attacks.	
How It Was Discovered	
Manual Analysis – Intercetping and modifying the request	
Vulnerable URLs	
https://labs.hacktify.in/HTML/cors_lab/lab_1/login.php	
Consequences of not Fixing the Issue	
If a **CORS vulnerability** is not fixed, attackers can exploit it to perform **unauthorized cross-origin requests**, leading to **data theft, session hijacking, and API abuse**. Malicious websites can trick a victim's browser into making requests to a vulnerable application, potentially exposing **sensitive user data, authentication tokens, or financial information**. If **credentials are included in CORS requests**, attackers could **bypass authentication mechanisms** and gain unauthorized access. This can result in **data breaches, compliance violations (e.g., GDPR), and reputational damage** for the affected organization.	
Suggested Countermeasures	
To prevent **CORS vulnerabilities**, configure the **CORS policy securely** by restricting Access-Control-Allow-Origin to **trusted domains** instead of using *. Avoid allowing **credentials (Access-Control-Allow-Credentials: true)** unless absolutely necessary, and never expose sensitive endpoints to untrusted origins. Implement **proper authentication and authorization checks** on the server-side rather than relying on CORS for security. Regularly audit CORS configurations and use **security headers** like Content-Security-Policy (CSP) to reduce the risk of cross-origin attacks.	
References	
https://portswigger.net/web-security/cors	

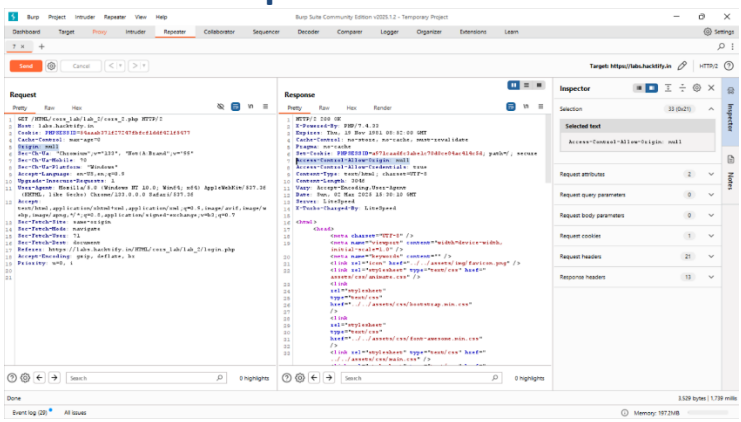
Proof of Concept



2.2. CORS with Null Origin

Reference	Risk Rating
CORS with Null Origin	**Low**
Tools Used	
Burpsuite	
Vulnerability Description	
A **null origin CORS vulnerability** occurs when a web application allows requests from the **null origin** (Access-Control-Allow-Origin: null), which is often misused in **sandboxed iframes, file:// URLs, and certain cross-origin requests**. Attackers can exploit this by embedding the vulnerable site in a **malicious iframe** or using a **local HTML file** to bypass origin restrictions, leading to **data theft, unauthorized API access, or session hijacking**. To mitigate this, developers should **restrict allowed origins to trusted domains** and avoid using null unless absolutely necessary in controlled environments.	
How It Was Discovered	
Manual Analysis – Intercetping and modifying the request	
Vulnerable URLs	
https://labs.hacktify.in/HTML/cors_lab/lab_2/login.php	
Consequences of not Fixing the Issue	
If a **null origin CORS vulnerability** is not fixed, attackers can exploit it to **bypass origin restrictions** and gain **unauthorized access to sensitive data or APIs**. Malicious websites or **local HTML files** can send requests on behalf of users, potentially leading to **data theft, session hijacking, or API abuse**. This is especially dangerous for applications handling **personal, financial, or authentication-related data**, as it can result in **data breaches, account compromises, and regulatory non-compliance**, ultimately damaging the organization's reputation and security.	
Suggested Countermeasures	
To prevent **null origin CORS vulnerabilities**, avoid using Access-Control-Allow-Origin: null unless absolutely necessary in controlled environments. Instead, explicitly define **trusted origins** and restrict cross-origin access to only those domains. Implement **strict authentication and authorization checks** on the server-side rather than relying on CORS for security. Additionally, disable null origin access for sensitive endpoints, enforce **SameSite cookies**, and use **Content Security Policy (CSP)** to restrict resource loading from untrusted sources. Regular security audits can help identify and mitigate such misconfigurations.	
References	
https://www.tenable.com/blog/understanding-cross-origin-resource-sharing-vulnerabilities	

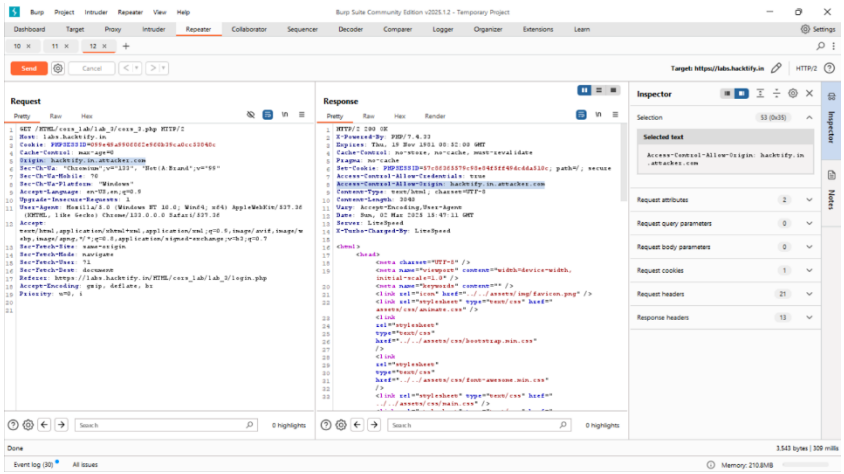
Proof of Concept

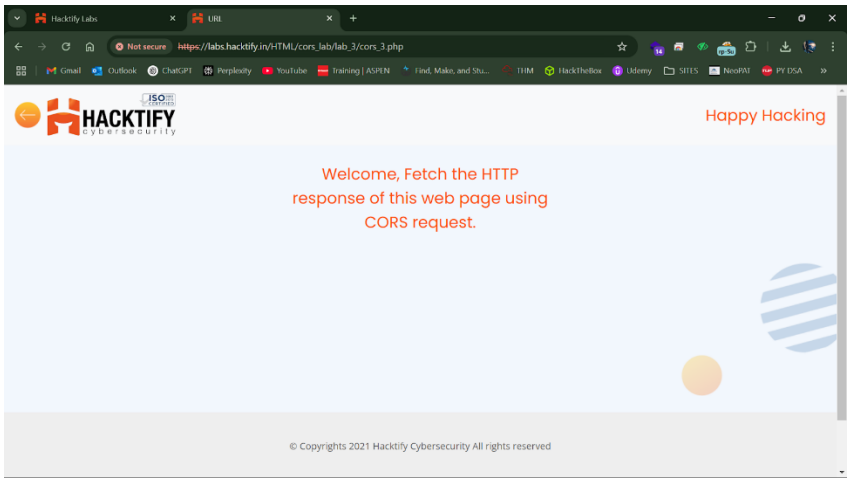


2.3. CORS with prefix match

Reference	Risk Rating
CORS with prefix match	**Medium**
Tools Used	
Burpsuite	
Vulnerability Description	
A **prefix match CORS vulnerability** occurs when a server improperly validates the Origin header by using **partial string matching** instead of an exact match. For example, if a server allows requests from "https://trusted.com" but incorrectly matches any domain starting with "https://trusted.com", an attacker can exploit this by hosting a malicious site at "https://trusted.com.attacker.com". This can lead to **unauthorized cross-origin access, data theft, and API abuse**, allowing attackers to steal **user credentials, tokens, or sensitive data** through manipulated requests.	
How It Was Discovered	
Manual Analysis – Intercetping and modifying the request	
Vulnerable URLs	
https://labs.hacktify.in/HTML/cors_lab/lab_3/login.php	
Consequences of not Fixing the Issue	
If a **prefix match CORS vulnerability** is not fixed, attackers can exploit it by hosting malicious sites with similar domain prefixes (e.g., https://trusted.com.attacker.com) to **bypass origin restrictions**. This can lead to **unauthorized API access, data leaks, session hijacking, and account takeovers**, as users' sensitive information may be exposed to **untrusted origins**. Such vulnerabilities can be particularly dangerous for applications handling **financial, authentication, or personal data**, potentially resulting in **data breaches, compliance violations, and reputational damage**.	
Suggested Countermeasures	
To prevent **prefix match CORS vulnerabilities**, always use **exact string matching** when validating the Origin header instead of partial or substring-based checks. Define a strict **allowlist of trusted origins** and avoid using wildcard (*) or dynamically reflecting origins without proper validation. Additionally, ensure **sensitive endpoints do not expose credentials** via CORS (Access-Control-Allow-Credentials: true) unless absolutely necessary. Regular security audits and testing can help detect and mitigate misconfigurations before they are exploited.	
References	
https://www.tenable.com/blog/understanding-cross-origin-resource-sharing-vulnerabilities	

Proof of Concept

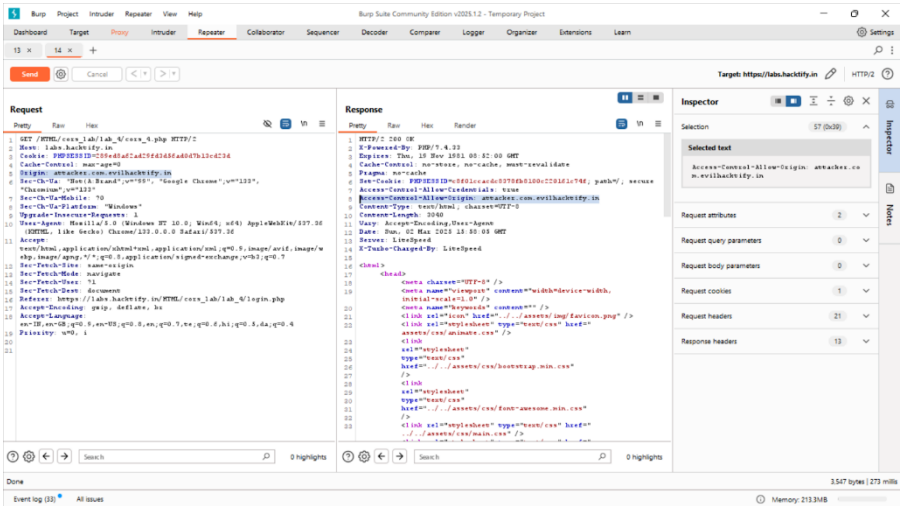


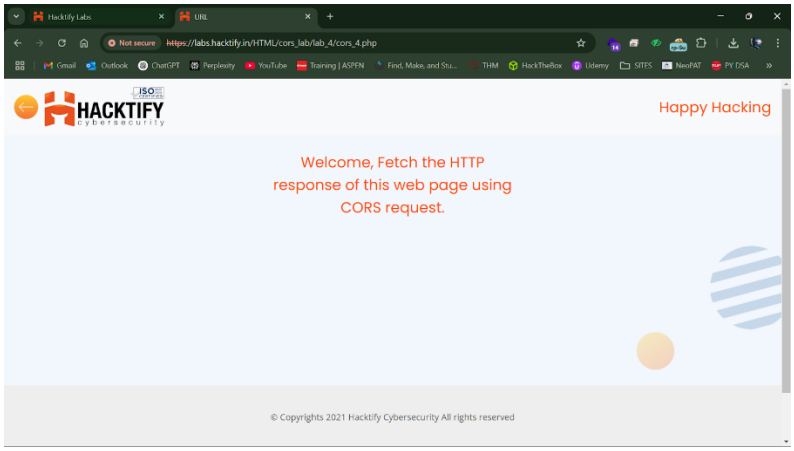


2.4. CORS with suffix match

Reference	Risk Rating
CORS with suffix match	**Medium**
Tools Used	
Burpsuite	
Vulnerability Description	
A **suffix match CORS vulnerability** occurs when a server incorrectly validates the Origin header by allowing any domain that **ends** with a trusted string (e.g., allowing example.com but also mistakenly permitting attackerexample.com). Attackers can exploit this by hosting malicious domains that **trick the server into granting cross-origin access**, leading to **data theft, unauthorized API requests, and session hijacking**.	
How It Was Discovered	
Manual Analysis – Intercetping and modifying the request	
Vulnerable URLs	
https://labs.hacktify.in/HTML/cors_lab/lab_4/login.php	
Consequences of not Fixing the Issue	
If a **suffix match CORS vulnerability** is not fixed, attackers can create deceptive domains (e.g., malicious-example.com) to **bypass security restrictions** and gain **unauthorized access to sensitive data or APIs**. This can lead to **user impersonation, data leaks, and financial fraud**, especially in applications dealing with **personal, authentication, or financial information**, ultimately causing **data breaches, compliance violations, and reputational damage**.	
Suggested Countermeasures	
To prevent **suffix match CORS vulnerabilities**, always perform **exact string matching** for trusted origins instead of relying on **partial or wildcard matching**. Use a **whitelist of fully qualified, explicitly trusted domains**, and never allow dynamic or user-controlled origins unless validated. Additionally, implement **strict authentication and authorization checks** on the server-side, enforce **SameSite cookies**, and use **security headers** like Content-Security-Policy (CSP) to mitigate risks. Regular audits should be conducted to ensure proper CORS configurations.	
References	
https://www.tenable.com/blog/understanding-cross-origin-resource-sharing-vulnerabilities	

Proof of Concept

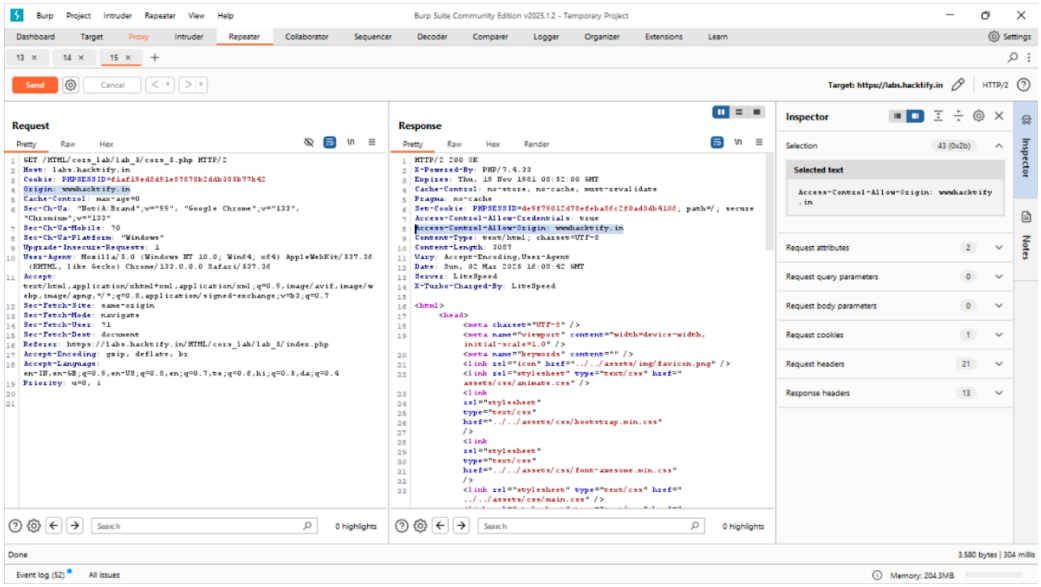


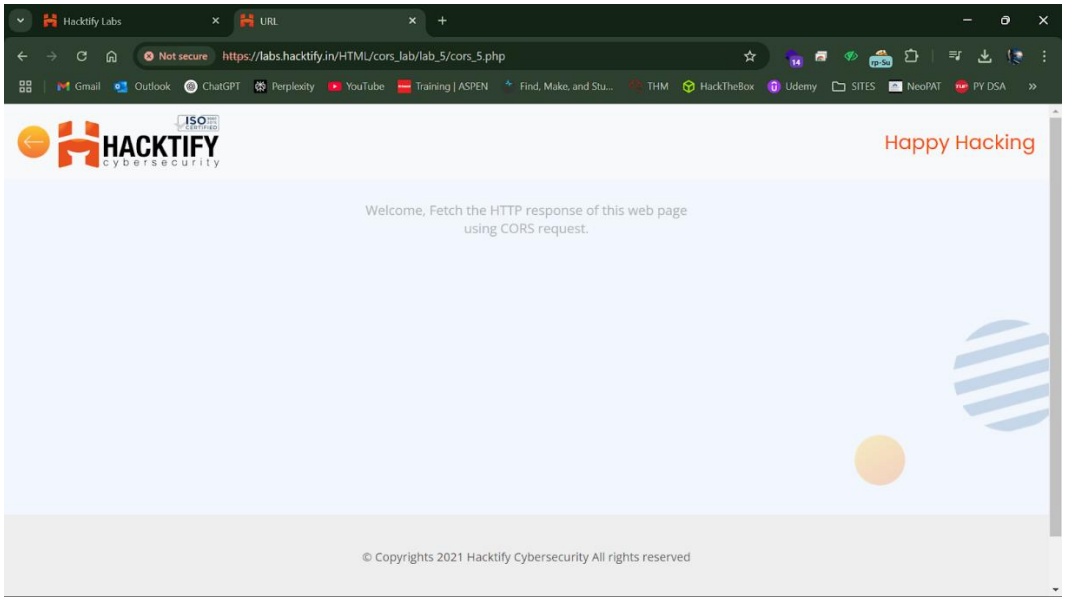


2.5. CORS with Escape dot

Reference	Risk Rating
CORS with Escape dot	**High**
Tools Used	
Burpsuite	
Vulnerability Description	
A **CORS with escape dot vulnerability** occurs when a server misinterprets domain validation due to improper handling of escaped characters (e.g., \. instead of .). Attackers can exploit this by crafting malicious subdomains (trusted\.malicious.com instead of trusted.malicious.com) that bypass CORS restrictions. This can lead to **unauthorized API access, data exfiltration, and session hijacking**, compromising sensitive user information.	
How It Was Discovered	
Manual Analysis – Intercetping and modifying the request	
Vulnerable URLs	
https://labs.hacktify.in/HTML/cors_lab/lab_5/login.php	
Consequences of not Fixing the Issue	
If not fixed, attackers can exploit **CORS misconfigurations** to **steal user data, hijack sessions, and perform unauthorized actions** on behalf of users. This can affect **authentication mechanisms, financial transactions, and private communications**, leading to **data breaches, compliance failures, and reputational damage**.	
Suggested Countermeasures	
To mitigate **CORS with escape dot vulnerabilities**, always use **exact string matching** for trusted origins and avoid **regex-based or partial matching**. Implement **strict CORS policies**, validate and sanitize input properly, and enforce **server-side authentication and authorization** rather than relying on CORS alone. Regular security audits should be conducted to detect and fix misconfigurations.	
References	
https://www.tenable.com/blog/understanding-cross-origin-resource-sharing-vulnerabilities	

Proof of Concept

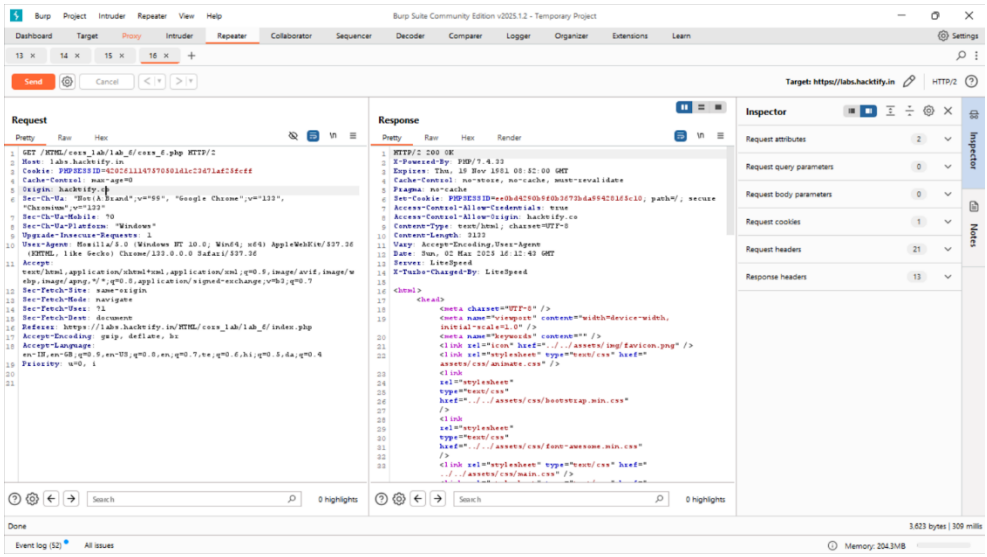


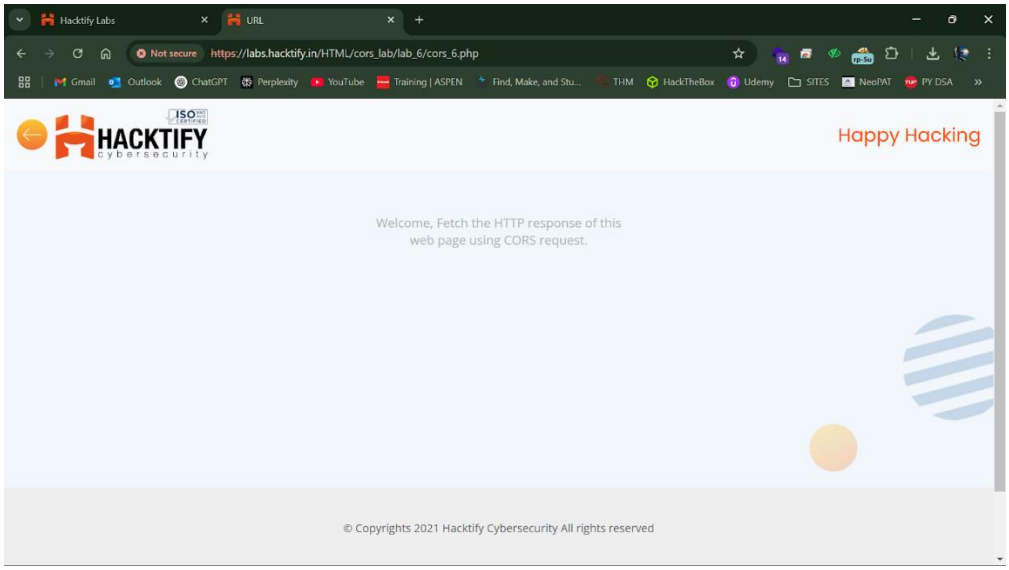


2.6. CORS with Substring match

Reference	Risk Rating
CORS with Substring match	**High**
Tools Used	
Burpsuite	
Vulnerability Description	
A **CORS with substring match vulnerability** occurs when a server improperly validates the Origin header by allowing any domain that **contains** a trusted string (e.g., permitting trusted.com but also allowing eviltrusted.com). Attackers can exploit this misconfiguration by **registering malicious domains containing the trusted string**, gaining unauthorized cross-origin access to sensitive APIs and user data.	
How It Was Discovered	
Manual Analysis – Intercetping and modifying the request	
Vulnerable URLs	
https://labs.hacktify.in/HTML/cors_lab/lab_6/login.php	
Consequences of not Fixing the Issue	
If not fixed, attackers can exploit this weakness to **steal sensitive data, perform unauthorized actions, hijack user sessions, or abuse APIs** by hosting malicious domains containing the trusted substring. This can lead to **data breaches, account takeovers, financial fraud, and compliance violations**, ultimately damaging user trust and an organization's reputation.	
Suggested Countermeasures	
To prevent **substring match vulnerabilities**, always use **exact string matching** for CORS origin validation rather than partial or regex-based matching. Maintain a strict **whitelist of fully qualified trusted domains**, avoid dynamically reflecting user-provided origins, and implement **server-side authentication and authorization** instead of relying solely on CORS. Conduct **regular security audits** to detect and fix misconfigurations.	
References	
https://www.tenable.com/blog/understanding-cross-origin-resource-sharing-vulnerabilities	

Proof of Concept

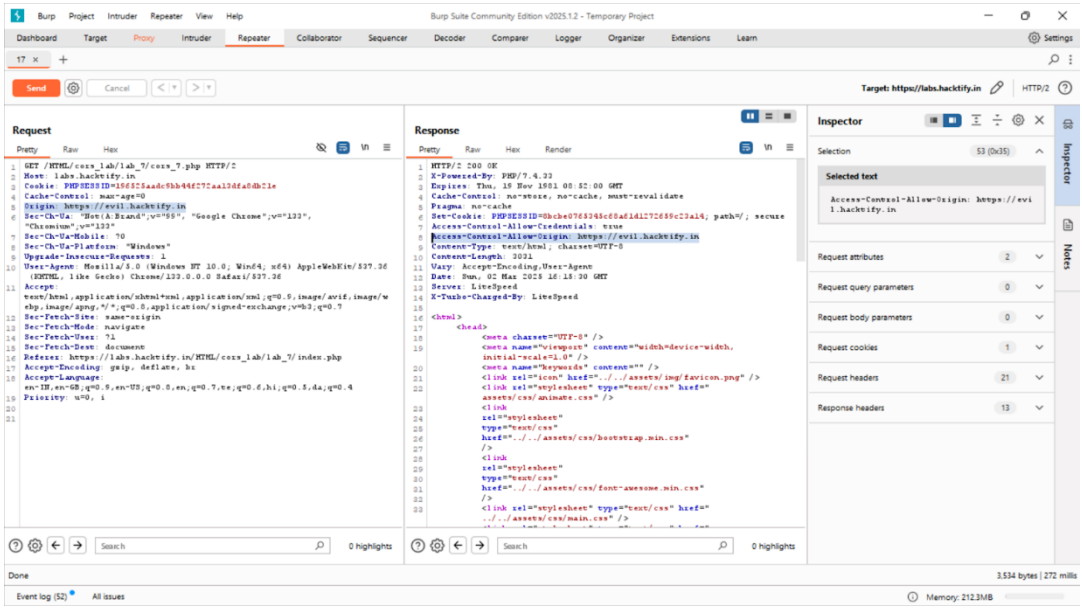


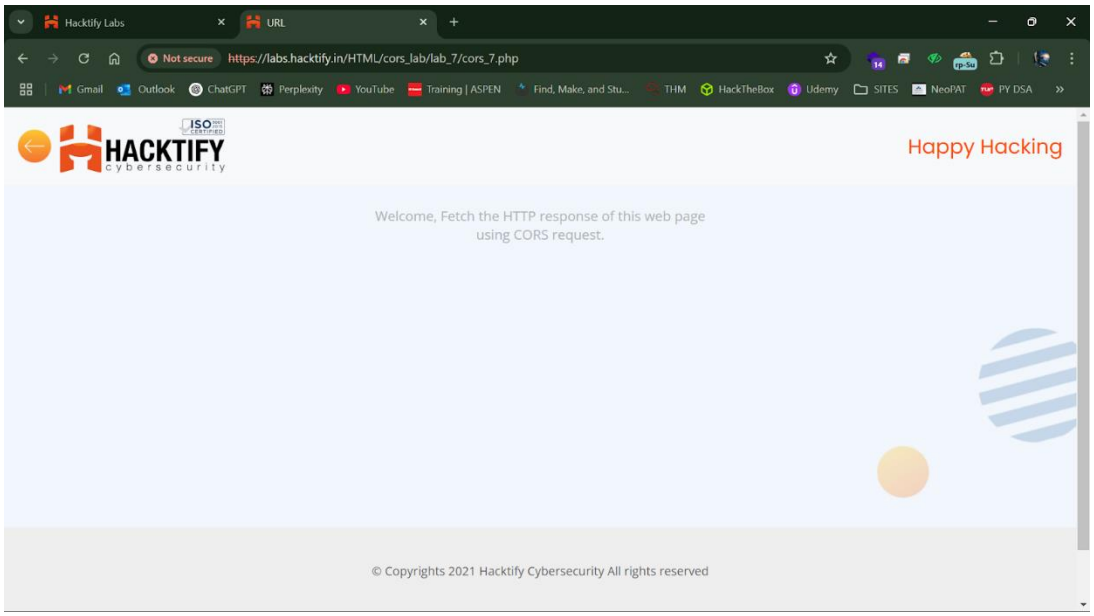


2.7. CORS with Arbitrary Subdomain

Reference	**Risk Rating**
CORS with Arbitrary Subdomain	**High**
Tools Used	
Burpsuite	
Vulnerability Description	
A **CORS with arbitrary subdomain vulnerability** occurs when a server **overly trusts all subdomains** using a wildcard (*.example.com) or improper validation, allowing any subdomain (e.g., attacker.example.com) to make cross-origin requests. If an attacker gains control of a subdomain through **subdomain takeover or misconfiguration**, they can **bypass CORS restrictions, access sensitive data, and exploit API endpoints**.	
How It Was Discovered	
Manual Analysis – Intercetping and modifying the request	
Vulnerable URLs	
https://labs.hacktify.in/HTML/cors_lab/lab_7/login.php	
Consequences of not Fixing the Issue	
If not fixed, attackers can create **malicious subdomains** to perform **data theft, session hijacking, and unauthorized API access**, leading to **account takeovers, financial fraud, and exposure of sensitive information**. This can result in **data breaches, legal consequences, and reputational damage**, especially for applications handling **authentication, financial transactions, or personal data**.	
Suggested Countermeasures	
To mitigate **CORS with arbitrary subdomain vulnerabilities**, avoid using wildcards (* or *.example.com) and enforce **exact matching of trusted origins**. Secure subdomains against **takeover attacks**, use **strict authentication and authorization checks** on the backend, and implement **security headers like Content Security Policy (CSP)** to prevent exploitation. Regularly audit DNS records and CORS policies to detect and fix misconfigurations.	
References	
https://www.tenable.com/blog/understanding-cross-origin-resource-sharing-vulnerabilities	

Proof of Concept





CTF Report

Full Name: AVIREDDY RUSHIKESH
Program: HCS - Penetration Testing 1-Month Internship
Date: 10/03/2025

Category: Web 2.0 {Lock Web} – Points: 100

Description:

Web 2.0 challenges focus on modern web applications, often involving JavaScript, APIs, authentication mechanisms, and client-side security flaws.

Challenge Overview:

The challenge presents a locked web interface requiring a numeric PIN for access. By analyzing client-side code and server responses, the goal is to bypass or extract the correct PIN to proceed.

Steps for Finding the Flag:

1. **Initial Reconnaissance:**
    - **Use Burp Suite Proxy to intercept requests and analyze website behavior.**
    - **Check for hidden parameters, input fields, and API endpoints.**
2. **Input Validation Testing:**
    - **Intercept requests and modify inputs in Repeater to test for vulnerabilities.**
3. **Brute Force & Enumeration:**
    - **Use Intruder to brute-force numeric PINs, directories, or credentials.**

- **Analyze response differences (status, length, or content changes).**

4. **Exploitation:**
    - **Modify successful responses in Repeater to manipulate the application's behavior.**
    - **Check session tokens, cookies, or redirects for access control flaws.**
5. **Flag Retrieval:**
    - **Identify the correct request/response revealing the flag.**
    - **Submit the captured flag from the response.**

Flag: flag{ V13w_r0b0t5.txt_c4n_b3_u53ful!!!}

Category: Web 2.0 {The World} – Points: 150

Challenge Description:

You've landed on a webpage that displays "Hello World!". While it looks simple, hidden paths exist within the site. The objective is to explore and uncover the flag.

Challenge Overview:

This challenge involves discovering hidden files and directories within a seemingly simple webpage. By leveraging web enumeration techniques, analyzing responses, and decoding hidden messages, the goal is to retrieve the flag.

Steps for Finding the Flag:

1. Initial Reconnaissance:

- Opened the website and inspected its structure.
- Viewed the page source (Ctrl+U) and checked for hidden elements.
- Used **Burp Suite Proxy** to capture network traffic and analyze requests.

2. Directory Enumeration:

- Ran dirb on the target URL to uncover hidden directories.
- Found **privacy.txt** and **secret.txt** files.

3. Decoding Hints:

- privacy.txt contained a Base64-encoded message:
  Encoded:
  Sm91cm5leSBvbndhcmRzIHVudGlsIHlvdSBmaW5hbGx5IGFycml2ZSBhdCB5b3VyIGRlc3RpbmF0aW9u **Decoded:** Journey onwards until you finally arrive at your destination
    - This hinted at further exploration of hidden paths.
- secret.txt contained another Base64-encoded string:
  Encoded: RkxBR3tZMHVfaGF2M180eHBsMHJlRF90aDNfVzByTGQhfQ==
  Decoded Flag: FLAG{Y0u_hav3_4xpl0reD_th3_W0rLd!}

Flag Retrieved:

FLAG{Y0u_hav3_4xpl0reD_th3_W0rLd!}

Category: Reverse Engg {Lost in the Past} – Points:150

Description:

This challenge involves reverse engineering an App Inventor project. By analyzing .scm and .bky files, identifying encoded text, and decoding it using ROT47, the flag is extracted from the app's logic.

Challenge Overview:

The challenge provides an App Inventor project that needs to be reverse-engineered. By inspecting the `.scm` and `.bky` files, identifying encoded text within the app logic, and decoding it using ROT47, the hidden flag can be retrieved.

Steps for Finding the Flag

1. **Extract and Analyze Project Files**
    - Identify key files: Screen1.scm, Scrum.bky, and project.properties.
    - Look for any text components that might display a flag.
2. **Inspect App Logic (Scrum.bky)**
    - Identify conditions triggering the flag display (e.g., specific switch combinations).
    - Locate encoded text inside the logic blocks (found in ROT47 format).
3. **Decode ROT47 Text**
    - Extract the encoded string from Scrum.bky:
    - 7=28LE__0>F490C6GbCD`?8N
    - Use a ROT47 decoder to decrypt it.
4. **Validate and Submit the Flag**
    - The decoded text is the flag. **FLAG: flag{t00_much_rev3rs1ng}**

Category: Reverse Engg {Decrypt Quest} -- Points: 200

Description:

Samarth's imaginary friend, Arjun, hands him a text file containing encrypted secret data. Arjun claims it's impossible to decode but promises a $1,000,000 reward if Samarth extracts the hidden information. However, the file is filled with irrelevant data to mislead him. The challenge requires analyzing the file, identifying useful clues, and decrypting the hidden message to obtain the flag.

Challenge Overview:

The challenge involves extracting a hidden flag from a text file filled with misleading data. By analyzing the file in CyberChef, a Java code snippet is revealed. This code contains a Google Drive link leading to an encrypted file. A hint suggests using Unix Epoch Time for decryption. Modifying the Java code to detect the 1970 timestamp helps retrieve the flag successfully.

Steps for Finding the Flag:

1. Download the file and analyze it using CyberChef to extract hidden data.

2. Identify the Java code in the extracted data, which contains a Google Drive link

3. Access the Drive link and decrypt its content using the hint: Unix Epoch Time.

4. Modify the Java code to simplify it and focus on finding the 1970 value (Epoch Time).

5. Run the corrected code, stopping at the Unix Epoch timestamp, which reveals the flag.

Flag: flag{hjwilj111970djs}