Penetration Testing Report

Full Name: Isha Sangpal
Program: HCPT
Date: 19/02/2025

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	**Html Injection, 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	5	8

High - **Number of Sub-labs with hard difficulty level**

Medium - **Number of Sub-labs with Medium difficulty level**

Low - **Number of Sub-labs with Easy difficulty level**

1. Html Injection Labs

1.1. HTML's are easy!

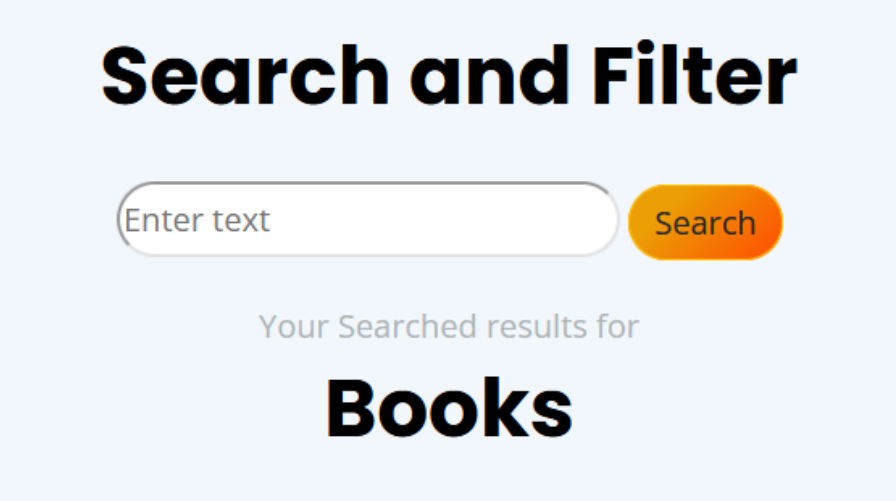
Reference	Risk Rating
HTML's are easy!	**Low**
Tools Used	
Web Browser (Mozilla Firefox), Developer's Tool	
Vulnerability Description	
This vulnerability occurs when user input is not properly sanitized, allowing HTML tags to be executed in the browser. This can lead to unintended rendering of elements, manipulation of page structure, or potential security risks like stored or reflected HTML injection.	
How It Was Discovered	
Manual Analysis: Entered HTML tags (e.g., Books) in the input field and observed the rendered output.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_1/html_injection_1.php	
Consequences of not Fixing the Issue	
HTML injection can disrupt the page layout and lead to unintended UI changes. Can be exploited for phishing attacks by tricking users with misleading content. Potential defacement of web pages if attackers insert malicious HTML elements.	
Suggested Countermeasures	
Input Validation & Sanitization: Strip or encode HTML tags before processing user input. Use Content Security Policy (CSP): Restrict execution of untrusted scripts and elements. Escape User Input: Convert < to < and > to > before rendering. Use a Secure Framework: Implement secure libraries that handle input sanitization by default.	
References	
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/03-Testing_for_HTML_Injection , https://www.acunetix.com/vulnerabilities/web/html-injection/ , https://www.imperva.com/learn/application-security/html-injection/	

Proof of Concept

Input field: books

We observe that the data entered in the input field is displayed below the input field.

Input field: <h1>Books</h1>The input field is not properly sanitized, allowing HTML tags to be executed. As a result, the `<h1>` tag is rendered, displaying "Books" as a heading instead of treating it as plain text.



1.2. Let me Store them!

Reference	Risk Rating
Let me Store them!	**Low**
Tools Used	
Web Browser (Mozilla Firefox), Developer's Tools (Browser DevTools)	
Vulnerability Description	
This vulnerability occurs when user input is stored in the application database without proper sanitization. If HTML or script tags are stored and later executed when retrieved, it can lead to HTML Injection. This allows an attacker to inject malicious scripts that execute whenever a user accesses the affected page.	
How It Was Discovered	
Manual Analysis: Stored Test in input fields, then checked if the stored data executed upon retrieval.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_2/html_injection_2.php	
Consequences of not Fixing the Issue	
Persistent Attack: Since the input is stored in the database, it affects all users who access the page. Account Hijacking: If an attacker injects JavaScript to steal session cookies, it can lead to unauthorized access. Defacement & Misinformation: Malicious content could be displayed to mislead users. Phishing Attacks: Injected content could be used to trick users into entering sensitive inform	
Suggested Countermeasures	
Input Validation & Encoding: Ensure that stored data is properly sanitized using HTML entities encoding before being displayed. Use Secure Libraries: Implement secure frameworks that automatically handle sanitization. Content Security Policy (CSP): Restrict execution of inline scripts to mitigate HTML injection attacks. Database-Level Security: Store user input in a format that prevents execution (e.g., escaping special characters).	
References	

https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/03-Testing_for_HTML_Injection , https://www.acunetix.com/vulnerabilities/web/html-injection/ , https://www.imperva.com/learn/application-security/html-injection/

Proof of Concept

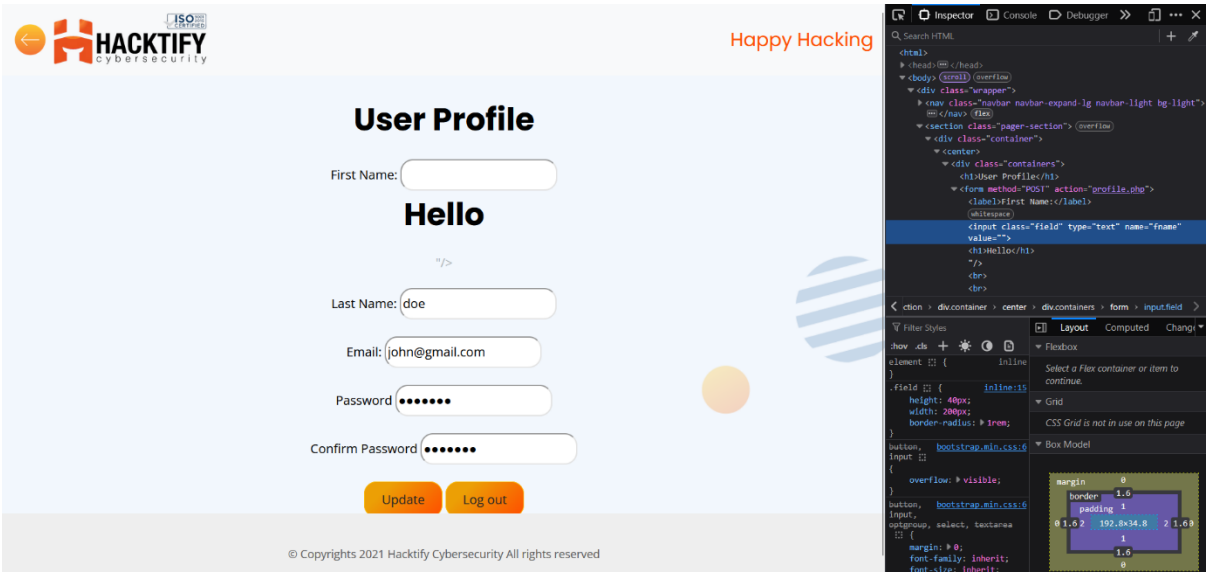
In this lab, on this webpage when I hit register and enter the information, a new user is created.

When I try to login with the created user credentials and hit login, User profile shows up with the details.

When we enter `"<h1>Hello</h1>"` directly into the input field, the webpage treats it as plain text rather than rendering it as an HTML heading, indicating that the input is properly sanitized.

Input: "/><h1>Hello</h1>

If we enter `"/> <h1>Hello</h1>` as input, it breaks out of the value parameter, causing the `<h1>` tag to be executed successfully instead of being treated as plain text. This indicates a lack of proper input sanitization, making the field vulnerable to HTML injection.



1.3. File Names are also vulnerable!

Reference	Risk Rating
File Names are also vulnerable!	**Low**
Tools Used	

Web Browser (Mozilla Firefox), Developer's Tools, Burp Suite
Vulnerability Description
This vulnerability occurs when file names are not properly sanitized before being processed or displayed on the webpage. If an attacker injects HTML or script tags in the filename, they can manipulate the rendering of the page, potentially leading to HTML Injection or Stored Cross-Site Scripting (XSS). This can be used for UI manipulation, phishing attacks, or other forms of exploitation.
How It Was Discovered
Manual Analysis: Used Developer's Tools to check how the filename is rendered on the webpage. Sent a POST request using Burp Suite to modify the filename parameter. Injected the payload Hello in the filename field and observed that it was executed successfully instead of being displayed as plain text.
Vulnerable URLs
https://labs.hacktify.in/HTML/html_lab/lab_3/html_injection_3.php
Consequences of not Fixing the Issue
HTML Injection: Malicious HTML elements can alter the page structure and mislead users. Phishing Attacks: Attackers can inject misleading content into the filename to trick users into providing sensitive information.
Suggested Countermeasures
Input Validation & Sanitization: Strip or encode HTML tags before storing or rendering file names. Escape User Input: Convert < to < and > to > to prevent unintended HTML execution. Use Secure Libraries: Implement frameworks that handle input sanitization by default (e.g., OWASP Java Encoder). Content Security Policy (CSP): Restrict execution of untrusted scripts to mitigate XSS risks. Whitelist File Extensions: Ensure only allowed file types and properly formatted filenames are accepted.
References
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/03-Testing_for_HTML_Injection , https://www.acunetix.com/vulnerabilities/web/html-injection/ , https://www.imperva.com/learn/application-security/html-injection/

Proof of Concept

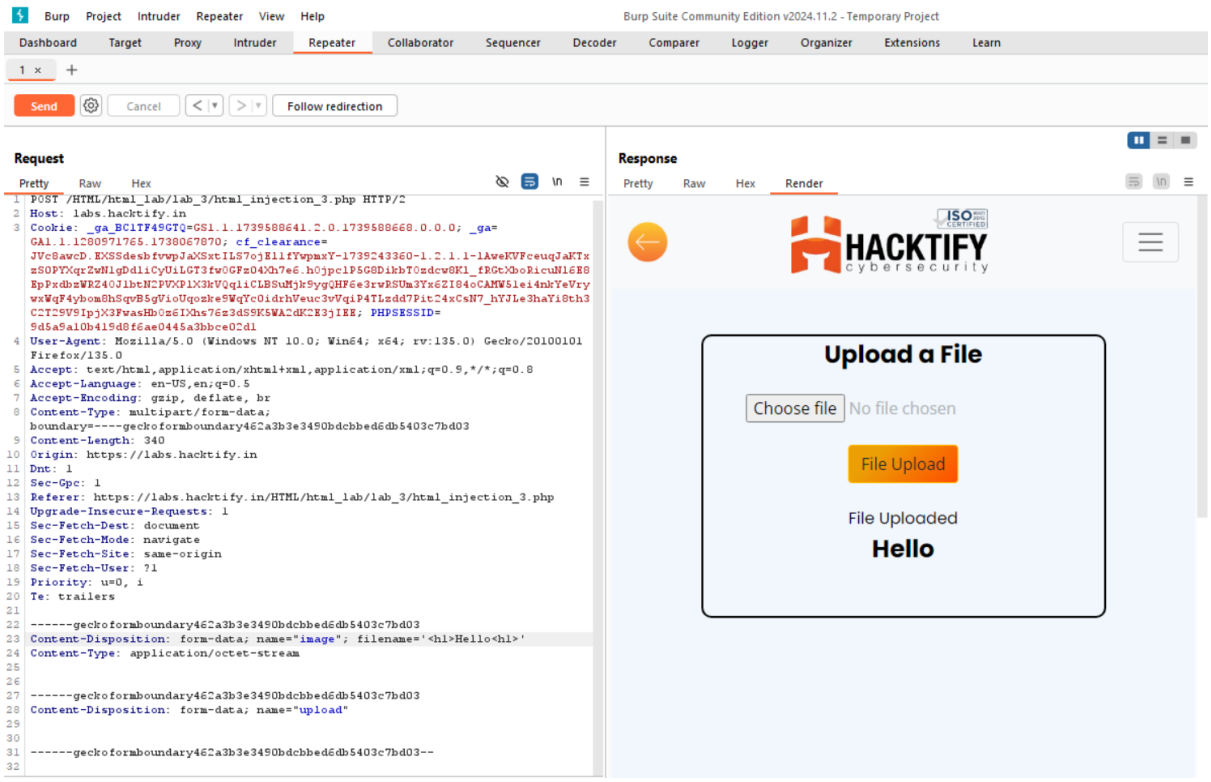
First I started with exploring the input field by uploading a file from my local system.

I observed that the name of the file which is selected is displayed after clicking on the File Upload button.

I attempted to exploit the file name displayed on the screen using the Developer's Tool. However, I found that the input is properly sanitized, causing the webpage to treat it as plain text and display it as-is.

Input field: `<h1>Hello<h1>`

I then sent a POST request to the repeater using Burp Suite and injected the payload `<h1>Hello<h1>` into the filename field. As a result, "Hello" was successfully displayed as an H1 heading, confirming that the input was not properly sanitized.



1.4. File Content and HTML Injection a perfect pair!

Reference	Risk Rating
File Content and HTML Injection a perfect pair!	**Low**
Tools Used	
Web Browser (Mozilla Firefox), Developer Tools	
Vulnerability Description	
This vulnerability occurs when the web application allows file uploads without properly validating or sanitizing the file content. If an attacker uploads an HTML file, the content inside may get executed instead of being treated as a regular file, leading to HTML Injection or Stored XSS (Cross-Site Scripting).	
How It Was Discovered	
Manual Analysis: Created an HTML script and saved it as malicious.html. Uploaded the file through the input field on the webpage. After clicking the "File Upload" button, the content inside the HTML file (This is a malicious script) was executed instead of being displayed as plain text, confirming the vulnerability.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_4/file_upload.php	
Consequences of not Fixing the Issue	
HTML Injection: Attackers can manipulate the UI, insert misleading content, or modify the website layout.	

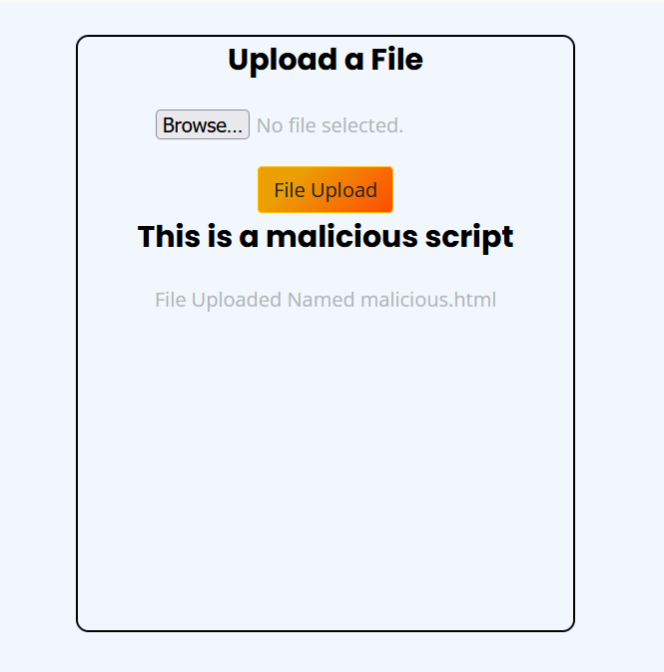
Stored XSS: If JavaScript is allowed in uploaded files, an attacker could inject scripts to steal session cookies, perform keylogging, or execute unauthorized actions on behalf of users. Malware Hosting: The vulnerability could allow attackers to upload phishing pages, malicious scripts, or malware, potentially compromising users. Data Breach & Reputation Damage: Sensitive data could be exposed, and unauthorized users may exploit the flaw to inject harmful content.
Suggested Countermeasures
Restrict File Types: Allow only specific file types (e.g., .jpg, .png, .pdf) and block .html, .js, .php or other executable file formats. Sanitize File Content: Prevent the execution of uploaded files by serving them as downloadable attachments instead of rendering them in the browser. Use Proper MIME-Type Validation: Check the file type at both client-side and server-side to ensure only allowed formats are accepted. Set Secure Headers: Implement Content-Disposition: attachment and X-Content-Type-Options: nosniff to prevent browsers from executing file content. Use a Secure File Storage Mechanism: Store uploaded files outside the web root directory to prevent direct access.
References
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/03-Testing_for_HTML_Injection , https://www.acunetix.com/vulnerabilities/web/html-injection/ , https://www.imperva.com/learn/application-security/html-injection/

Proof of Concept

I created an HTML script and saved it as `malicious.html`, then uploaded it through the input field on the webpage to test whether the file gets executed or if its contents are displayed as plain text.

Malicious.html : <h1>This is a malicious script</h1>

After selecting the file and clicking the "File Upload" button, the uploaded HTML script was executed, and its content was rendered on the screen instead of being treated as a regular file.



1.5. Injecting HTML using URL

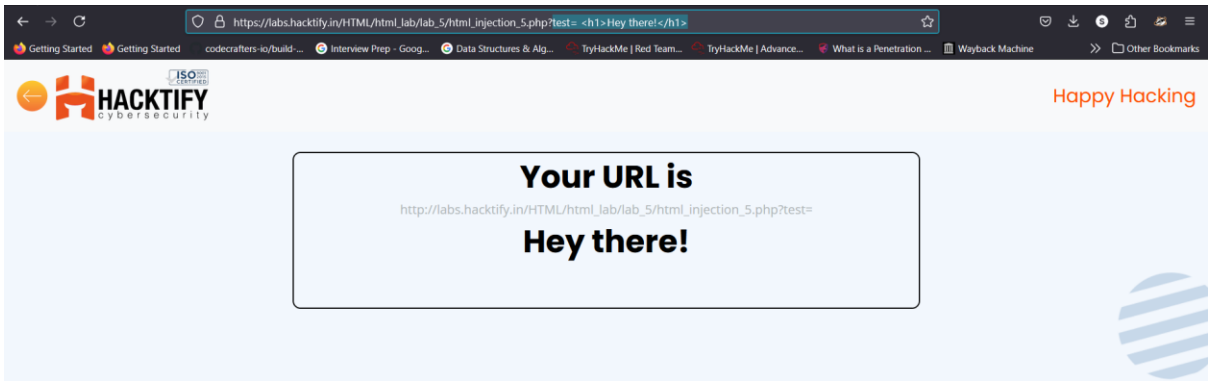
Reference	Risk Rating
Injecting HTML using URL	**Low**
Tools Used	
Web Browser (Mozilla Firefox), Developer Tools	
Vulnerability Description	
The application is vulnerable to HTML Injection via URL parameters. When user input is passed through the query parameter without proper sanitization, the injected HTML gets executed in the browser instead of being treated as plain text. This flaw can be exploited to manipulate the webpage content, perform phishing attacks, or facilitate Cross-Site Scripting (XSS).	
How It Was Discovered	
Manual Analysis: A crafted payload was injected into the test parameter in the URL. Upon accessing the URL, the injected HTML was rendered on the webpage instead of being treated as plain text.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_5/html_injection_5.php?test= Hey there!	
Consequences of not Fixing the Issue	
HTML Injection: Attackers can manipulate the UI, display misleading information, or modify page content. Stored/Reflected XSS Risk: If JavaScript is allowed, attackers could steal session cookies, perform keylogging, or execute unauthorized actions on behalf of users. Phishing Attacks: Users could be redirected to malicious pages through deceptive content.	
Suggested Countermeasures	
Use Secure Output Encoding: Implement functions like htmlspecialchars() in PHP to prevent HTML execution.	

Implement Content Security Policy (CSP): Restrict inline scripts and unauthorized content execution. Use HTTP Response Headers: Enforce security headers such as X-XSS-Protection: 1; mode=block to mitigate injection risks.
References
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/03-Testing_for_HTML_Injection , https://www.acunetix.com/vulnerabilities/web/html-injection/ , https://www.imperva.com/learn/application-security/html-injection/

Proof of Concept

https://labs.hacktify.in/HTML/html_lab/lab_5/html_injection_5.php
Into this URL, payload: ?test=<h1>Hey there!</h1> was injected.
As a result, the HTML code was executed instead of being treated as plain text, confirming that the input is not properly sanitized and the application is vulnerable to HTML Injection

Final URL:
https://labs.hacktify.in/HTML/html_lab/lab_5/html_injection_5.php?test=<h1>Hey there!</h1>



1.6. Encode IT!

Reference	Risk Rating
Encode IT!	**Low**
Tools Used	
Web Browser (Mozilla Firefox), Developer Tools, URL Encoder/Decoder (To test encoded input handling)	
Vulnerability Description	
This vulnerability arises when the application improperly processes encoded user input. While direct HTML injection is sanitized, URL-encoded values are decoded and rendered as HTML, bypassing security filters. This leads to HTML Injection and potentially Cross-Site Scripting (XSS).	
How It Was Discovered	

Manual Analysis: Tested direct input: Input: books → Displayed correctly as books. Input: Books → Partially sanitized, displayed as ;h1;Books;/h1;. Input: Books → Rendered as Books , indicating improper decoding. Bypassed sanitization using URL encoding: Input: %3Ch1%3EHello%3C%2Fh1%3E (URL-encoded version of Hello) Output: Hello was displayed in an H1 tag, proving the vulnerability.
Vulnerable URLs
https://labs.hacktify.in/HTML/html_lab/lab_6/html_injection_6.php
Consequences of not Fixing the Issue
HTML Injection: Attackers can modify page content, mislead users, or manipulate the UI. Cross-Site Scripting (XSS): If JavaScript execution is possible, attackers can steal session cookies, perform keylogging, or execute unauthorized actions on behalf of users. Phishing Attacks: Malicious content can be injected into the page, tricking users into providing sensitive information. Security Filter Bypass: Encoding-based sanitization bypasses can be used to evade weak security measures.
Suggested Countermeasures
Proper Input Validation: Ensure all user input is validated before processing. Sanitize and Encode Output: Prevent double-decoding by encoding user input properly. Use Secure Encoding Functions: Instead of just filtering special characters, encode them using: echo htmlspecialchars($_GET['test'], ENT_QUOTES, 'UTF-8'); Disable Automatic URL Decoding: Ensure the backend does not automatically decode input before processing. Implement Content Security Policy (CSP): Restrict execution of inline scripts and untrusted content.
References
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/03-Testing_for_HTML_Injection , https://www.acunetix.com/vulnerabilities/web/html-injection/ , https://www.imperva.com/learn/application-security/html-injection/

Proof of Concept

Payload Analysis and Exploitation

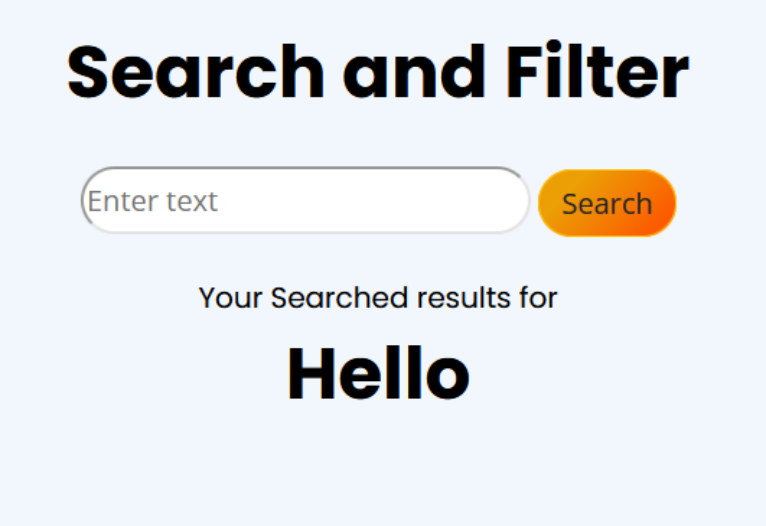
- Input: `books` → Displayed on screen: `books` (Handled correctly)
- Input: `<h1>Books</h1>` → Displayed on screen: `;h1;Books;/h1;` (Indicates basic sanitization by replacing `< >` with `;`)
- Input: `<h1>Books</h1>` → Displayed on screen: `<h1>Books</h1>` (Suggests improper decoding, allowing execution of encoded HTML)

Although direct HTML input is sanitized, bypassing the filter using URL encoding was successful:

- Encoded Input (via urlencoder.org):
  %3Ch1%3EHello%3C%2Fh1%3E

- Displayed on screen:
  <h1>Hello</h1>

This confirms that the application decodes URL-encoded input before rendering, exploiting the vulnerability and allowing HTML Injection.



2. Cross Site Scripting

2.1. Let's Do IT!

Reference	Risk Rating
Let's Do IT!	**Medium**
Tools Used	
Web Browser (Mozilla Firefox)	
Vulnerability Description	
Cross-Site Scripting (XSS) is a client-side vulnerability that allows an attacker to inject malicious scripts into a trusted website. When a user inputs JavaScript code in the newsletter subscription form and submits it, the website does not sanitize the input properly. This leads to the execution of JavaScript code in the user's browser, potentially leading to data theft, session hijacking, or website defacement.	
How It Was Discovered	
Manual Analysis (Injecting XSS payloads)	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_2/lab_2.php	
Consequences of not Fixing the Issue	
Session Hijacking: Attackers can steal session cookies and impersonate users. Phishing Attacks: Malicious JavaScript can redirect users to phishing pages. Keylogging: Attackers can log keystrokes and steal credentials. Defacement: The attacker can modify the webpage content dynamically.	
Suggested Countermeasures	
Input Validation: Allow only expected input (e.g., valid email format). Output Encoding: Encode special characters (< > " ' &) using HTML entity encoding. Content Security Policy (CSP): Restrict inline JavaScript execution.	

Sanitization Libraries: Use libraries like DOMPurify to remove dangerous scripts.
References
Awesome XSS : https://github.com/s0md3v/AwesomeXSS Cross Site Scripting by PortSwigger : https://portswigger.net/web-security/cross-site-scripting OWASP XSS : https://owasp.org/www-community/attacks/xss/

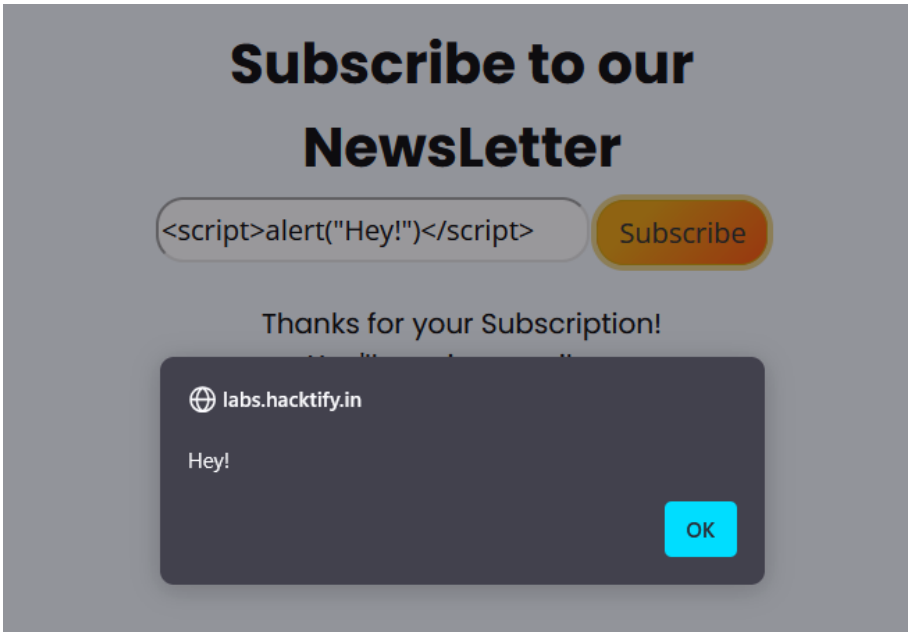
Proof of Concept

Scenario 1: Normal Input
example@gmail.com

Thanks for your Subscription!
You'll receive email on example@gmail.com
Scenario 2: Injecting XSS Payload
<script>alert("Hey!")</script>

Observed Behavior:

The script executes, displaying an alert box with the message "Hey!".
This confirms that the application is vulnerable to Stored XSS.



2.2. Balancing is Important in Life!

Reference	Risk Rating

Balancing is Important in Life!	**Medium**
Tools Used	
Web Browser (Mozilla Firefox)	
Vulnerability Description	
This XSS vulnerability occurs in the newsletter subscription form, where user input is being partially sanitized. The application attempts to filter some input but does not fully escape or validate special characters. This results in an exploitable Reflected XSS vulnerability when a crafted URL is used.	
How It Was Discovered	
Manual Analysis (Injecting XSS payloads)	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_2/lab_2.php	
Consequences of not Fixing the Issue	
User Impersonation: Attackers can steal cookies via document.cookie and hijack user sessions. Phishing Attacks: Malicious JavaScript can redirect users to fake login pages. Malware Injection: Attackers can inject hidden iframes or malicious scripts to exploit users. Sensitive Data Exposure: Keylogging and credential harvesting attacks are possible.	
Suggested Countermeasures	
Proper Input Validation: Allow only valid email formats using regex. Output Encoding: Convert special characters (< > " ' &) to HTML entities before rendering them. Use HTTP Headers: Set Content Security Policy (CSP) to block inline scripts. Use X-XSS-Protection: 1; mode=block. Sanitize User Input: Use server-side filtering (e.g., htmlspecialchars() in PHP). Implement escaping functions (e.g., encodeForHTML() in security libraries). WAF (Web Application Firewall): Implement rules to detect and block XSS payloads.	
References	
Awesome XSS : https://github.com/s0md3v/AwesomeXSS Cross Site Scripting by PortSwigger : https://portswigger.net/web-security/cross-site-scripting OWASP XSS : https://owasp.org/www-community/attacks/xss/	

Proof of Concept

Scenario 1: Normal Input
example@gmail.com
Output:
Thanks for your Subscription!
You'll receive email on example@gmail.com

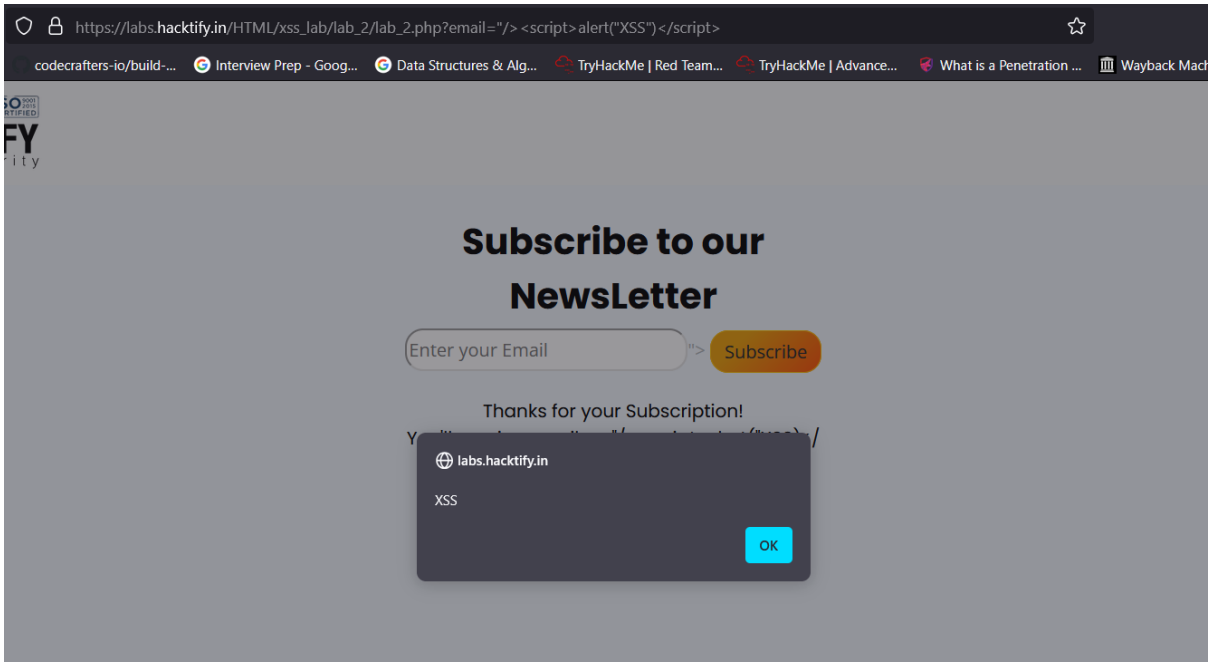
Scenario 2: Injecting XSS Payload (Partially Sanitized)
<script>alert("XSS")</script>
Observed Output:
Thanks for your Subscription!
You'll receive email on <script>alert("XSS")</script>

This indicates partial sanitization, but the script is still stored in the response.

Scenario 3: Bypassing Filters
"/><script>alert("XSS")</script>

Observed Behavior:

The script successfully executes in the browser, displaying an alert box.
Confirms a Reflected XSS vulnerability due to improper sanitization.



2.3. XSS is everywhere!

Reference	Risk Rating
XSS is everywhere!	**Medium**
Tools Used	
Web Browser (Mozilla Firefox)	
Vulnerability Description	
This XSS vulnerability occurs in the newsletter subscription form, where email validation is implemented but not properly sanitized for injected scripts when appending text after an email.	
How It Was Discovered	
Manual Analysis: Manual Testing with different payloads to bypass email validation. Initial email-only payloads were blocked, but appending an XSS payload after an email led to script execution.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_3/lab_3.php	
Consequences of not Fixing the Issue	
Session Hijacking: Attackers can steal authentication cookies. Phishing Attacks: Malicious JavaScript can redirect users to phishing pages.	

Malware Injection: The attacker can inject malicious scripts to compromise user security. Stored XSS Risk: If the input is stored in logs or databases and later rendered, it could lead to Stored XSS
Suggested Countermeasures
Strict Input Validation: Use server-side validation to enforce proper email formats. Restrict input to only allow characters required for a valid email (regex-based filtering). Output Encoding: Encode user input to prevent script execution. Use HTML entity encoding (< > " ' &) when displaying user input. Content Security Policy (CSP): Enforce CSP rules to block inline JavaScript execution. Sanitization Libraries: Use security libraries such as DOMPurify to remove malicious script tags. Implement a Web Application Firewall (WAF): Configure WAF rules to detect and block XSS payloads.
References
Awesome XSS : https://github.com/s0md3v/AwesomeXSS Cross Site Scripting by PortSwigger : https://portswigger.net/web-security/cross-site-scripting OWASP XSS : https://owasp.org/www-community/attacks/xss/

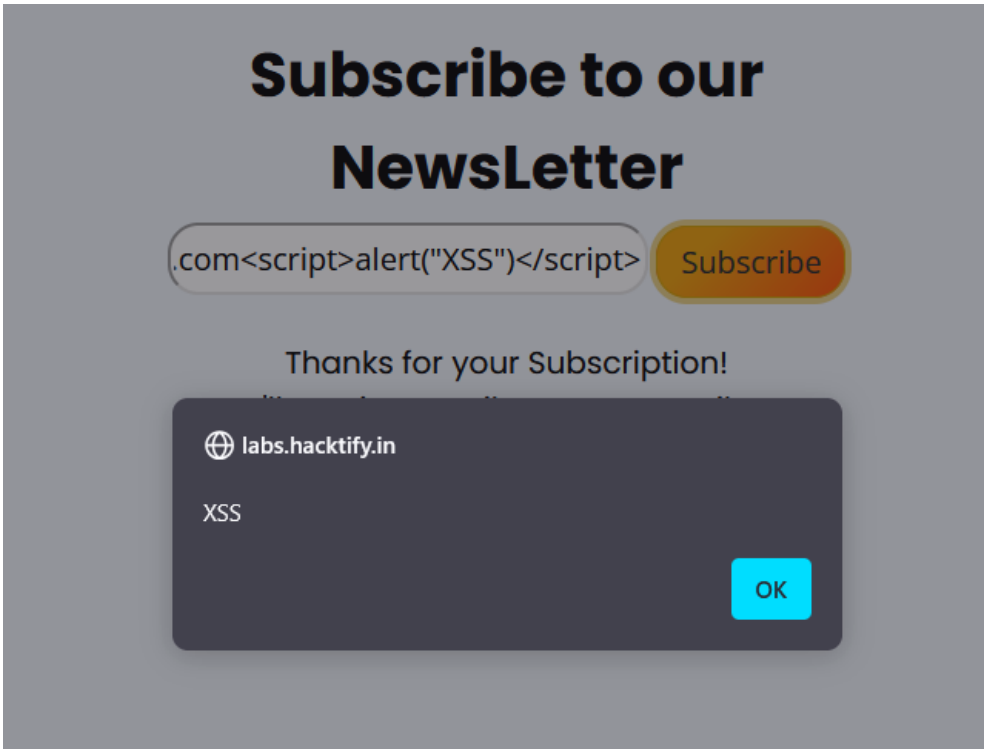
Proof of Concept

Scenario 1: Normal Input
example@gmail.com
Expected Output:
Thanks for your Subscription!
You'll receive email on example@gmail.com

Scenario 2: Injecting XSS Payload Alone (Blocked)
<script>alert("XSS")</script>
Observed Output:
Please Enter Valid Email address
Conclusion: Basic email validation is implemented.

Scenario 3: Bypassing Validation Using Appended XSS Payload
example@gmail.com<script>alert("XSS")</script>

Observed Behavior:
The script successfully executes in the browser, displaying an alert box.
This confirms that the validation is incomplete, and appended input is not sanitized properly.



2.4. Alternatives are must!

Reference	**Risk Rating**
Alternatives are must!	**Medium**
Tools Used	
Web Browser (Mozilla Firefox)	
Vulnerability Description	
This XSS vulnerability is present in the newsletter subscription form, where user input is being reflected without proper sanitization. Unlike previous labs, this test demonstrates an advanced attack scenario where cookies can be stolen, allowing an attacker to hijack user sessions.	
How It Was Discovered	
Manual Testing using script-based XSS payloads was initially tested. The payload was reflected in the response but not executed (suggesting partial sanitization). An image-based XSS payload (onerror=prompt(document.cookie)) successfully executed, proving that the input sanitization is incomplete.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_4/lab_4.php	
Consequences of not Fixing the Issue	
Session Hijacking: Attackers can steal session cookies and impersonate users. Account Takeover: Stolen cookies allow unauthorized access to accounts. Sensitive Data Theft: Attackers can extract user authentication details. Malware Injection: Malicious JavaScript can be injected to further exploit victims. Wormable XSS: This attack could be automatically propagated to other users if the input is stored in a database.	
Suggested Countermeasures	
Strict Input Validation: Allow only valid email formats.	

Use regex-based filtering to strictly validate email inputs. Output Encoding: Encode user input to prevent script execution. Use HTML entity encoding (< > " ' &) when displaying user input. Content Security Policy (CSP): Enforce CSP rules to block inline JavaScript execution. Example CSP header: Content-Security-Policy: default-src 'self'; script-src 'none'; Sanitization Libraries: Use security libraries such as DOMPurify to remove malicious script tags. In PHP, use htmlspecialchars() or htmlentities() for proper escaping. HttpOnly and Secure Flags for Cookies: Prevent JavaScript from accessing session cookies by setting: Set-Cookie: PHPSESSID=xyz; HttpOnly; Secure Web Application Firewall (WAF): Deploy a WAF to detect and block XSS payloads.
References
Awesome XSS : https://github.com/s0md3v/AwesomeXSS Cross Site Scripting by PortSwigger : https://portswigger.net/web-security/cross-site-scripting OWASP XSS : https://owasp.org/www-community/attacks/xss/

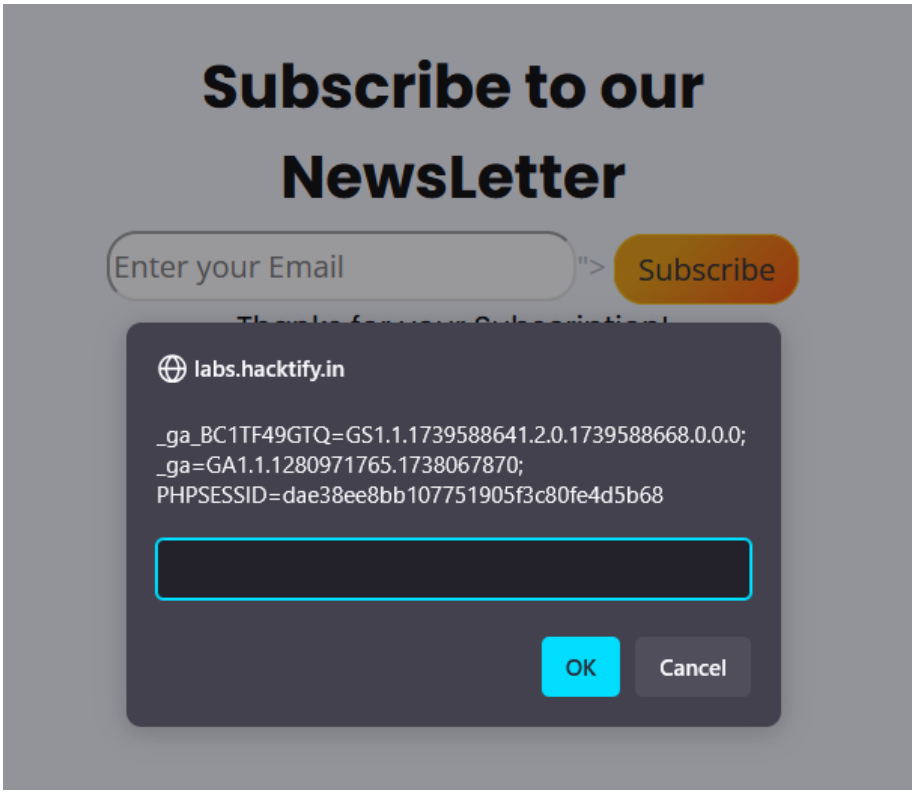
Proof of Concept

Scenario 1: Normal Input
example@gmail.com
Expected Output:
Thanks for your Subscription!
You'll receive email on example@gmail.com

Scenario 2: Injecting XSS Payload (Partially Sanitized)
<script>alert("XSS")</script>
Observed Output:
Thanks for your Subscription!
You'll receive email on <script>alert("XSS")</script>
Conclusion: The input is stored but not executed.

Scenario 3: Stealing Cookies with Image-Based XSS
"/><img src =q onerror=prompt(document.cookie)>

Observed Behavior:
The script successfully executes in the browser, displaying an alert box.
The alert reveals session cookies, including:
_ga_BC1TF49GTQ=GS1.1.1739588641.2.0.1739588668.0.0;
_ga=GA1.1.1280971765.1738067870;
PHPSESSID=dae38ee8bb107751905f3c80fe4d5b68
This proves session hijacking is possible, making the vulnerability high-risk.



2.5. Developer hates scripts!

Reference	Risk Rating
Developer hates scripts!	**Medium**
Tools Used	
Web Browser (Mozilla Firefox)	
Vulnerability Description	
This XSS vulnerability is present in the newsletter subscription form, where user input is being reflected without proper sanitization. Unlike previous labs, this test demonstrates an advanced attack scenario where cookies can be stolen, allowing an attacker to hijack user sessions.	
How It Was Discovered	
Manual Testing using script-based XSS payloads was initially tested. The payload was reflected in the response but not executed (suggesting partial sanitization). An image-based XSS payload (onerror=prompt(document.cookie)) successfully executed, proving that the input sanitization is incomplete.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_5/lab_5.php	
Consequences of not Fixing the Issue	
Session Hijacking: Attackers can steal session cookies and impersonate users. Account Takeover: Stolen cookies allow unauthorized access to accounts. Sensitive Data Theft: Attackers can extract user authentication details. Malware Injection: Malicious JavaScript can be injected to further exploit victims. Wormable XSS: This attack could be automatically propagated to other users if the input is stored in a database.	
Suggested Countermeasures	

Strict Input Validation: Allow only valid email formats. Use regex-based filtering to strictly validate email inputs. Output Encoding: Encode user input to prevent script execution. Use HTML entity encoding (< > " ' &) when displaying user input. Content Security Policy (CSP): Enforce CSP rules to block inline JavaScript execution. Example CSP header: Content-Security-Policy: default-src 'self'; script-src 'none'; Sanitization Libraries: Use security libraries such as DOMPurify to remove malicious script tags. In PHP, use htmlspecialchars() or htmlentities() for proper escaping. HttpOnly and Secure Flags for Cookies: Prevent JavaScript from accessing session cookies by setting: Set-Cookie: PHPSESSID=xyz; HttpOnly; Secure Web Application Firewall (WAF): Deploy a WAF to detect and block XSS payloads.
References
Awesome XSS : https://github.com/s0md3v/AwesomeXSS Cross Site Scripting by PortSwigger : https://portswigger.net/web-security/cross-site-scripting OWASP XSS : https://owasp.org/www-community/attacks/xss/

Proof of Concept

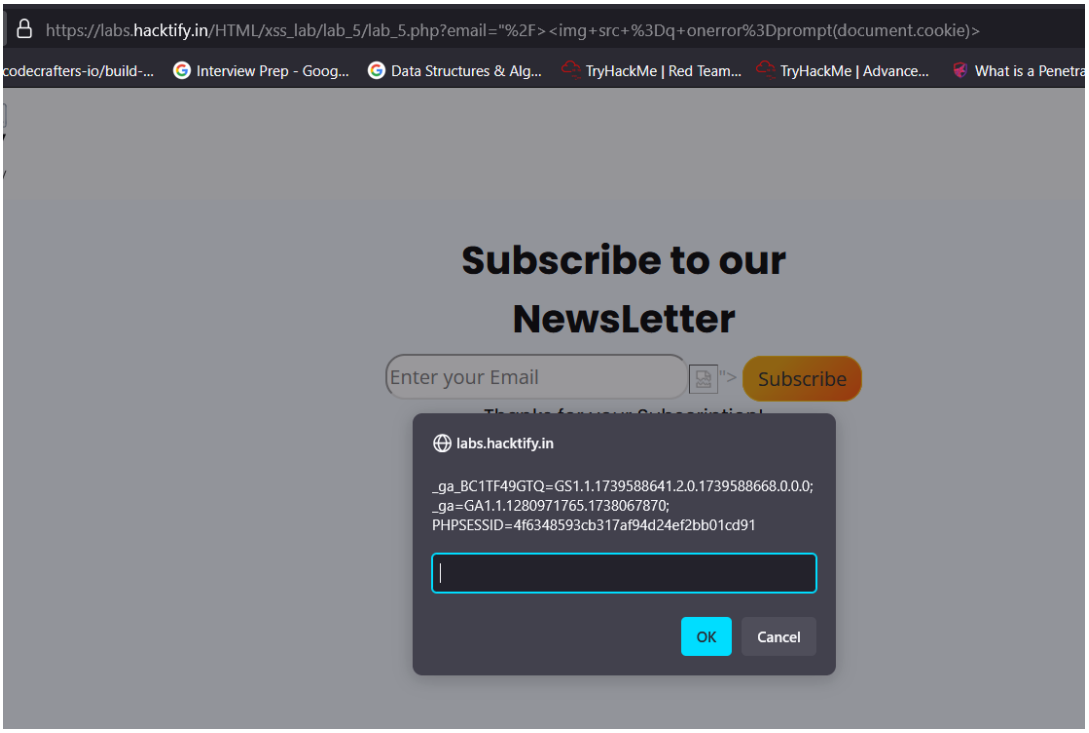
Scenario 1: Normal Input
example@gmail.com
Expected Output:
Thanks for your Subscription!
You'll receive email on example@gmail.com

Scenario 2: Injecting XSS Payload (Partially Sanitized)
<script>alert("XSS")</script>
Observed Output:
Thanks for your Subscription!
You'll receive email on <script>alert("XSS")</script>
Conclusion: The input is stored but not executed.

Scenario 3: Stealing Cookies with Image-Based XSS
"/><img src =q onerror=prompt(document.cookie)>

Observed Behavior:
The script successfully executes in the browser, displaying an alert box.
The alert reveals session cookies, including:
_ga_BC1TF49GTQ=GS1.1.1739588641.2.0.1739588668.0.0;
_ga=GA1.1.1280971765.1738067870;

PHPSESSID=4f6348593cb317af94d24ef2bb01cd91
This proves session hijacking is possible, making the vulnerability high-risk.



2.6. Change the Variation!

Reference	**Risk Rating**
Change the Variation!	**Medium**
Tools Used	
Web Browser (Mozilla Firefox)	
Vulnerability Description	
This XSS vulnerability is present in the newsletter subscription form, where user input is being reflected without proper sanitization. Unlike previous labs, this test demonstrates an advanced attack scenario where cookies can be stolen, allowing an attacker to hijack user sessions.	
How It Was Discovered	
Manual Testing using script-based XSS payloads was initially tested. The payload was reflected in the response but not executed (suggesting partial sanitization). An image-based XSS payload (onerror=prompt(document.cookie)) successfully executed, proving that the input sanitization is incomplete.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_6/lab_6.php	
Consequences of not Fixing the Issue	
Session Hijacking: Attackers can steal session cookies and impersonate users. Account Takeover: Stolen cookies allow unauthorized access to accounts. Sensitive Data Theft: Attackers can extract user authentication details.	

Malware Injection: Malicious JavaScript can be injected to further exploit victims. Wormable XSS: This attack could be automatically propagated to other users if the input is stored in a database.
Suggested Countermeasures
Strict Input Validation: Allow only valid email formats. Use regex-based filtering to strictly validate email inputs. Output Encoding: Encode user input to prevent script execution. Use HTML entity encoding (< > " ' &) when displaying user input. Content Security Policy (CSP): Enforce CSP rules to block inline JavaScript execution. Example CSP header: Content-Security-Policy: default-src 'self'; script-src 'none'; Sanitization Libraries: Use security libraries such as DOMPurify to remove malicious script tags. In PHP, use htmlspecialchars() or htmlentities() for proper escaping. HttpOnly and Secure Flags for Cookies: Prevent JavaScript from accessing session cookies by setting: Set-Cookie: PHPSESSID=xyz; HttpOnly; Secure Web Application Firewall (WAF): Deploy a WAF to detect and block XSS payloads.
References
Awesome XSS : https://github.com/s0md3v/AwesomeXSS Cross Site Scripting by PortSwigger : https://portswigger.net/web-security/cross-site-scripting OWASP XSS : https://owasp.org/www-community/attacks/xss/

Proof of Concept

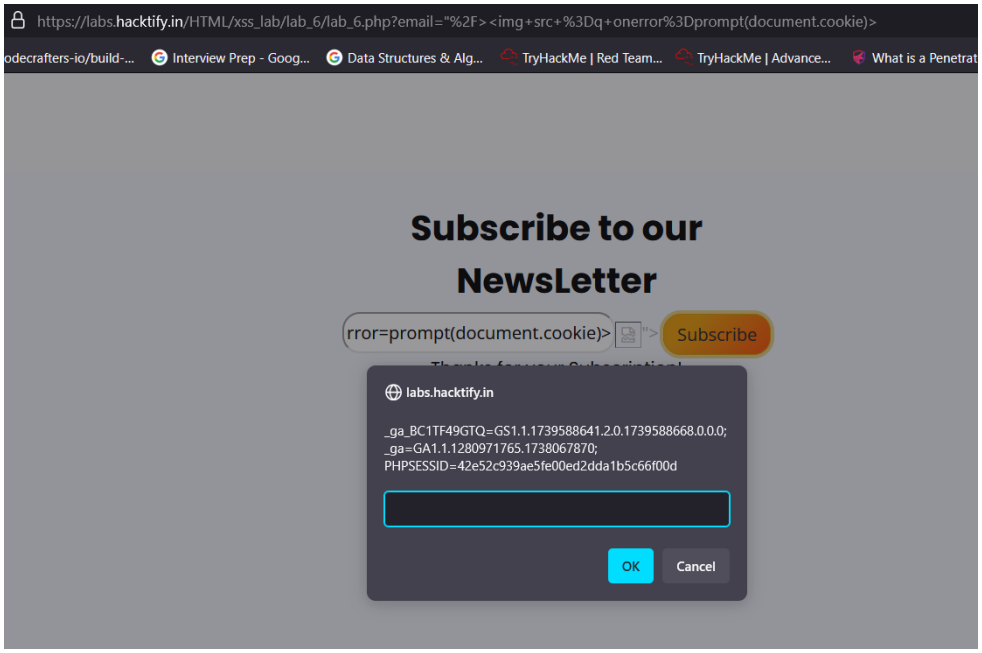
Scenario 1: Normal Input
example@gmail.com
Expected Output:
Thanks for your Subscription!
You'll receive email on example@gmail.com

Scenario 2: Injecting XSS Payload (Partially Sanitized)
<script>alert("XSS")</script>
Observed Output:
Thanks for your Subscription!
You'll receive email on <script>alert("XSS")</script>
Conclusion: The input is stored but not executed.

Scenario 3: Stealing Cookies with Image-Based XSS
"/><img src =q onerror=prompt(document.cookie)>

Observed Behavior:
The script successfully executes in the browser, displaying an alert box.

The alert reveals session cookies, including:
_ga_BC1TF49GTQ=GS1.1.1739588641.2.0.1739588668.0.0;
_ga=GA1.1.1280971765.1738067870;
PHPSESSID=42e52c939ae5fe00ed2dda1b5c66f00d
This proves session hijacking is possible, making the vulnerability high-risk.



2.7. Encoding is the key?

Reference	**Risk Rating**
Encoding is the key?	**Medium**
Tools Used	
Tools that you have used to find the vulnerability.	
Vulnerability Description	
This XSS vulnerability occurs in the newsletter subscription form, where user input is reflected with improper sanitization. The application attempts to filter some characters but fails to properly encode URL-decoded script injections, leading to successful Reflected XSS.	
How It Was Discovered	
Manual Testing revealed that direct <script> tags were filtered. URL-encoding the payload bypassed the filter, proving Reflected XSS vulnerability. The browser executed the JavaScript, leaking session cookies.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_7/lab_7.php?email=%253Cscript%253Ealert%2528document.cookie%2529%253C%252Fscript%253E	
Consequences of not Fixing the Issue	
Session Hijacking: Attackers can steal session cookies and gain unauthorized access. Persistent Account Takeover: If cookies are stored long-term, attackers can maintain control indefinitely. Credential Theft: If stored authentication details are accessible, they can be stolen. Malware Injection: Malicious JavaScript can be injected to infect users.	

Automated Exploits: Attackers can send phishing links containing pre-encoded XSS payloads.
Suggested Countermeasures
Proper Input Validation Ensure only valid email formats are accepted using strict regex. Reject characters like < > " ' ; to prevent injection. Output Encoding Convert user input to safe HTML entities before rendering: htmlspecialchars($_POST['email'], ENT_QUOTES, 'UTF-8'); This ensures <script> tags do not get executed. Enforce Content Security Policy (CSP) Block inline scripts and restrict external JavaScript execution: Content-Security-Policy: default-src 'self'; script-src 'none'; Set HttpOnly and Secure Flags for Cookies Prevent JavaScript from accessing session cookies: session_set_cookie_params(['httponly' => true, 'secure' => true, 'samesite' => 'Strict']); Sanitization Libraries Use DOMPurify in JavaScript or OWASP Java Encoder to remove dangerous input. Implement a Web Application Firewall (WAF) Use ModSecurity or Cloudflare WAF to detect and block XSS payloads.
References
OWASP XSS Prevention Guide: https://cheatsheetseries.owasp.org/cheatsheets/XSS_Prevention_Cheat_Sheet.html Mozilla's Guide on Content Security Policy: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP Awesome XSS : https://github.com/s0md3v/AwesomeXSS Cross Site Scripting by PortSwigger : https://portswigger.net/web-security/cross-site-scripting OWASP XSS : https://owasp.org/www-community/attacks/xss/

Proof of Concept

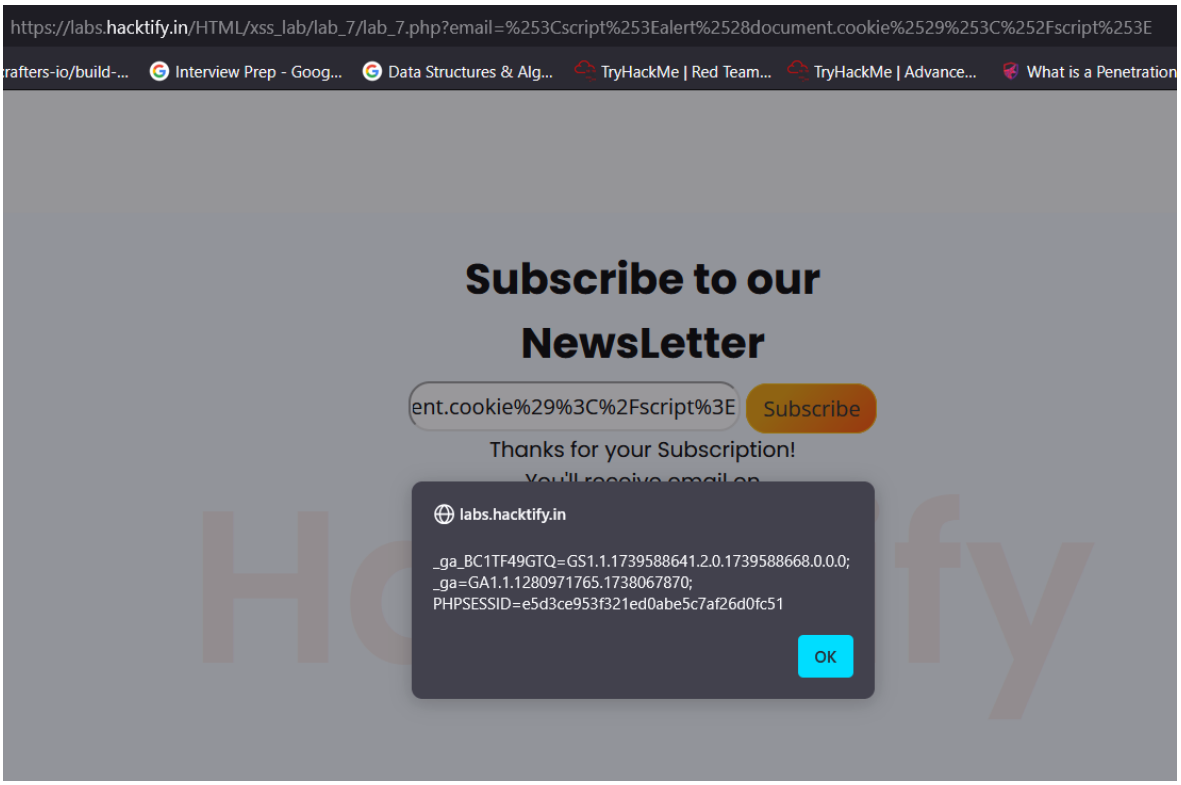
Scenario 1: Normal Input
example@gmail.com
Expected Output:
Thanks for your Subscription!
You'll receive email on example@gmail.com

Scenario 2: Injecting XSS Payload (Filtered)

<script>alert(document.cookie)</script>
Observed Output:
for your Subscription!
You'll receive email on scriptalertdocument.cookiescript
Conclusion: The application attempts to sanitize input but does not encode all characters properly.

Scenario 3: Bypassing Filters with URL-Encoding
%3Cscript%3Ealert%28document.cookie%29%3C%2Fscript%3E
Observed Behavior:

The script successfully executes, displaying an alert box with session cookies.
The extracted PHPSESSID can be used for session hijacking.



2.8. XSS with File Upload (file name)

Reference	Risk Rating
XSS with File Upload (file name)	**Medium**
Tools Used	
Web Browser (Mozilla Firefox), Burp Suite (for request modification)	
Vulnerability Description	
This XSS vulnerability exists in the file upload feature, where user-controlled filename attributes are not properly sanitized. Attackers can inject XSS payloads into the filename, leading to Reflected XSS execution when the filename is displayed.	
How It Was Discovered	
Manual Testing using PortSwigger's filename injection technique.	

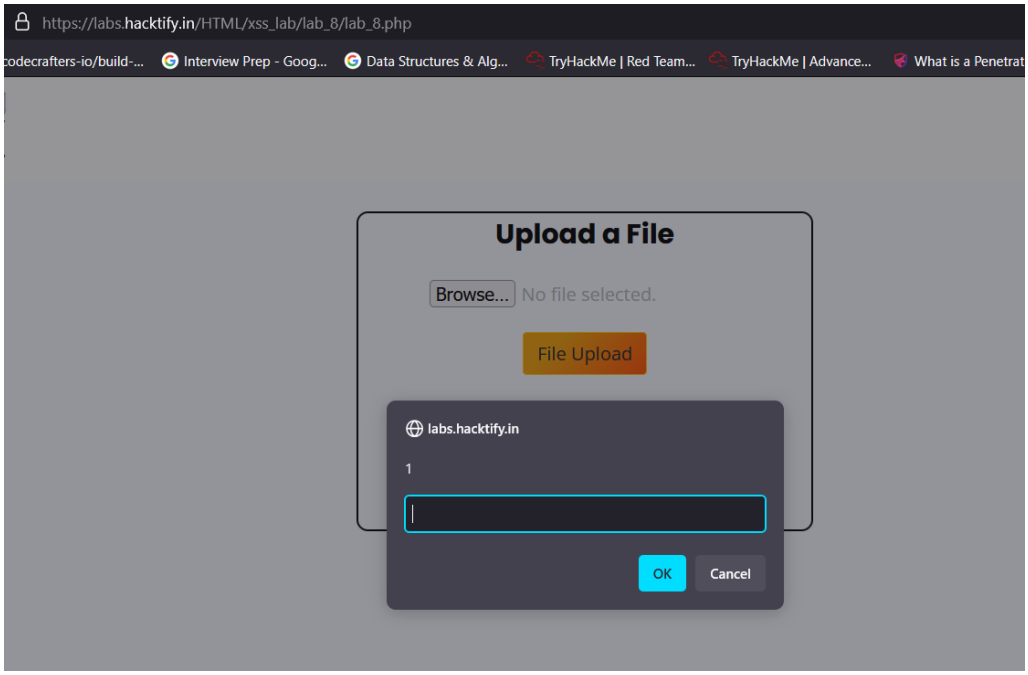
Initial normal file uploads were tested to analyze the behavior. Injecting an XSS payload into the filename resulted in script execution. This confirms that filename inputs are not sanitized.
Vulnerable URLs
https://labs.hacktify.in/HTML/xss_lab/lab_8/lab_8.php The filename attribute in the file upload module is vulnerable.
Consequences of not Fixing the Issue
Stored XSS Risk: If the filename is stored and later rendered, it could lead to a persistent XSS attack. Session Hijacking: Attackers can steal cookies using document.cookie. Malware Injection: JavaScript payloads can be used to serve malicious redirects or exploit users. Phishing Attacks: An attacker could craft a malicious filename to trick users into clicking.
Suggested Countermeasures
Strict Input Validation Restrict filenames to only alphanumeric characters and safe symbols. Use regex filtering to prevent XSS payloads: $filename = preg_replace('/[^a-zA-Z0-9_\.-]/', '', $_FILES['file']['name']); Output Encoding Encode user input before displaying filenames: htmlspecialchars($filename, ENT_QUOTES, 'UTF-8'); This prevents the execution of malicious JavaScript. Use Content Security Policy (CSP) Block inline scripts and prevent execution of injected scripts: Content-Security-Policy: default-src 'self'; script-src 'none'; Rename Uploaded Files Assign randomized filenames to prevent stored XSS: $new_filename = uniqid() . '.pdf'; move_uploaded_file($_FILES['file']['tmp_name'], "uploads/" . $new_filename); Implement a Web Application Firewall (WAF) Detect and block malicious payloads in filenames.
References
Awesome XSS : https://github.com/s0md3v/AwesomeXSS Cross Site Scripting by PortSwigger : https://portswigger.net/web-security/cross-site-scripting OWASP XSS : https://owasp.org/www-community/attacks/xss/

Proof of Concept

Scenario 1: Normal File Upload
Filename: 1807.04320v2.pdf
Expected Output:
File Uploaded: 1807.04320v2.pdf
No XSS execution occurs.

Scenario 2: Injecting XSS Payload in Filename
"/><img src =q onerror=prompt(1)>

Observed Behavior:
The script executes, displaying an alert box.
Confirms XSS vulnerability due to improper filename sanitization.



2.9. XSS with File Upload (File Content)

Reference	**Risk Rating**
XSS with File Upload (File Content)	**Medium**
Tools Used	
Web Browser (Mozilla Firefox)	
Vulnerability Description	
This Stored XSS vulnerability exists in the file upload functionality, where user-uploaded files are not properly validated. Attackers can upload malicious HTML files containing JavaScript, which execute when accessed.	
How It Was Discovered	
A basic JavaScript payload was inserted into an HTML file and uploaded. The uploaded file was stored and accessible via the web server. Opening the file in a browser executed the script, confirming Stored XSS vulnerability	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_9/lab_9.php The file upload endpoint is vulnerable.	
Consequences of not Fixing the Issue	
Stored XSS Attacks: Malicious scripts can persist on the server and execute for any user who accesses the file. Session Hijacking: Attackers can steal authentication cookies. Defacement Attacks: Uploaded HTML files can alter the website's appearance. Malware Distribution: Attackers can upload drive-by download exploits. Phishing Attacks: Users can be tricked into entering credentials on fake login pages.	

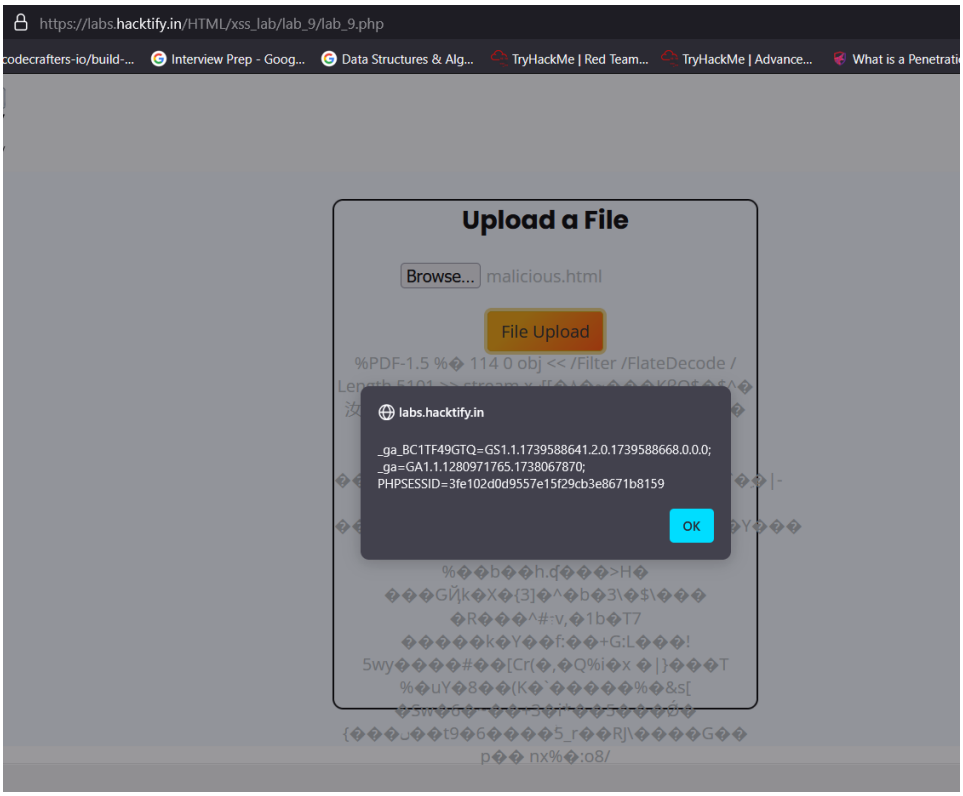
Suggested Countermeasures
Restrict File Upload Types Allow only safe file formats (e.g., PDF, JPEG, PNG). Block execution of HTML, JavaScript, and executable files. Rename Uploaded Files Assign randomized filenames and remove original extensions: $new_filename = uniqid() . '.pdf'; move_uploaded_file($_FILES['file']['tmp_name'], "uploads/" . $new_filename); Validate File Contents Check file MIME type and content inspection: $allowed_types = ['application/pdf', 'image/png', 'image/jpeg']; if (!in_array(mime_content_type($_FILES['file']['tmp_name']), $allowed_types)) { die("Invalid file type"); } Store Files in a Secure Directory Store uploaded files outside the web root and serve via scripts. Disable Execution of Uploaded Files Add rules in .htaccess to prevent execution: <FilesMatch "\.(html\|htm\|shtml\|php\|js\|exe\|sh\|pl\|cgi\|asp\|aspx)$"> ForceType application/octet-stream Header set Content-Disposition attachment </FilesMatch> Use Content Security Policy (CSP) Block inline scripts from executing: Content-Security-Policy: default-src 'self'; script-src 'none'; Implement a Web Application Firewall (WAF) Detect and block XSS payloads in file uploads.
References
Awesome XSS : https://github.com/s0md3v/AwesomeXSS Cross Site Scripting by PortSwigger : https://portswigger.net/web-security/cross-site-scripting OWASP XSS : https://owasp.org/www-community/attacks/xss/ PortSwigger's Guide on File Upload Attacks: https://portswigger.net/web-security/file-upload

Proof of Concept

Scenario 1: Normal File Upload
Filename: document.pdf
Expected Output:
File Uploaded: document.pdf
No XSS execution occurs.

Scenario 2: Uploading a Malicious HTML File
File Contents (malicious.html):
<script>alert(document.cookie)</script>

Observed Behavior:
The script executes, displaying an alert box.
Confirms Stored XSS vulnerability via file upload.



2.10. Stored Everywhere!

Reference	Risk Rating
Stored Everywhere!	**High**
Tools Used	
Web Browser (Mozilla Firefox)	
Vulnerability Description	
This Stored XSS vulnerability exists in the user registration form, where input fields (name, last name) are not properly sanitized. Attackers can inject malicious JavaScript code, which gets stored in the database and executes when the user logs in.	
How It Was Discovered	
During registration, an XSS payload was injected into the name fields. The application stored the input without sanitization. Upon logging in, the stored payload executed, proving Stored XSS vulnerability.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_10/profile.php	

The name and last name fields in the registration form are vulnerable.
Consequences of not Fixing the Issue
Stored XSS Attacks: The script executes whenever any user logs in, making it persistent. Session Hijacking: Attackers can steal PHP session cookies. Defacement Attacks: Stored scripts can be used to alter website content. Credential Theft: JavaScript keyloggers can be injected to steal user passwords. Wormable XSS: If an attacker injects self-replicating JavaScript, it could spread across multiple users.
Suggested Countermeasures
Sanitize User Input Encode and sanitize user input before storing in the database: $name = htmlspecialchars($_POST['name'], ENT_QUOTES, 'UTF-8'); $lastname = htmlspecialchars($_POST['lastname'], ENT_QUOTES, 'UTF-8'); This ensures that <script> tags do not execute. Use Prepared Statements Prevent XSS and SQL Injection by using parameterized queries: $stmt = $pdo->prepare("INSERT INTO users (name, lastname, email, password) VALUES (?, ?, ?, ?)"); $stmt->execute([$name, $lastname, $email, password_hash($password, PASSWORD_DEFAULT)]); Output Encoding Before displaying user data, apply encoding: echo htmlentities($user_name, ENT_QUOTES, 'UTF-8'); Enforce Content Security Policy (CSP) Prevent inline script execution: Content-Security-Policy: default-src 'self'; script-src 'none'; Implement a Web Application Firewall (WAF) Detect and block XSS payloads in user inputs.
References
Awesome XSS : https://github.com/s0md3v/AwesomeXSS Cross Site Scripting by PortSwigger : https://portswigger.net/web-security/cross-site-scripting OWASP XSS : https://owasp.org/www-community/attacks/xss/ Mozilla's Guide on Content Security Policy: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

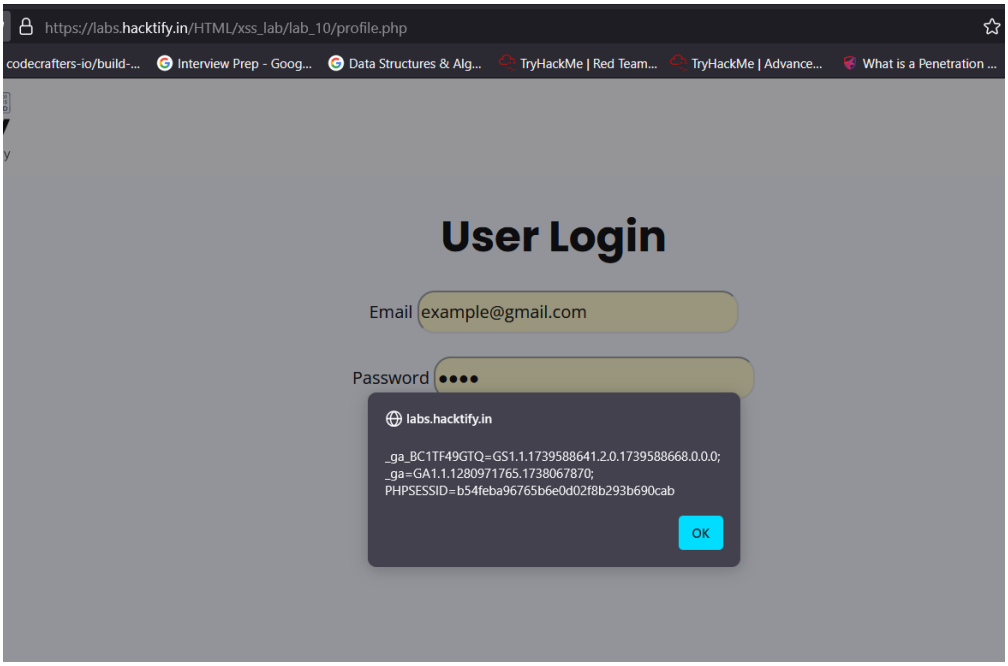
Proof of Concept

Scenario 1: Normal Registration
Name: John
Last Name: Doe
Email: example@gmail.com
Password: 1234
Expected Output:
User registered successfully.
No XSS execution occurs.

Scenario 2: Injecting XSS Payload in Name Field
Payload:

"/><script>alert(document.cookie)</script>

Observed Behavior:
The script executes during login, confirming Stored XSS vulnerability.



2.11 DOM's are love!

Reference	Risk Rating
DOM's are love!	**Medium**
Tools Used	
Web Browser (Mozilla Firefox)	
Vulnerability Description	
This DOM-Based XSS vulnerability exists in the JavaScript file dom.js, where the name parameter from the URL is inserted directly into the DOM without sanitization.	
How It Was Discovered	
Viewing the page source (view-source:) revealed that user input is directly reflected. The JavaScript file (dom.js) does not sanitize input. By injecting a <script> tag in the URL, arbitrary JavaScript executed.	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?name=%3Cimage%20src=q%20onerror=prompt(document.cookie)%3E	
Consequences of not Fixing the Issue	
Session Hijacking: Attackers can steal authentication cookies via document.cookie. Credential Theft: Malicious JavaScript can be injected to log keystrokes. Phishing Attacks: Attackers can redirect users to a fake login page. Malware Injection: JavaScript payloads can serve drive-by downloads. Automated Attacks: Attackers can craft malicious links and send them to victims.	
Suggested Countermeasures	
1. Avoid innerHTML for User Input	

Use textContent or innerText instead of innerHTML:
document.getElementById("p1").textContent = "Hello Hacker, " + username + "!";
This prevents HTML injection.
2. Sanitize User Input Before Displaying
Use a DOM XSS Sanitization Library, such as DOMPurify:
document.getElementById("p1").innerHTML = DOMPurify.sanitize(username);
This removes malicious scripts before inserting them into the page.
3. Escape Special Characters in User Input
Ensure that HTML special characters are encoded before inserting into the DOM:

```
function escapeHTML(str) {
  return str.replace(/&/g, "&amp;")
            .replace(/</g, "&lt;")
            .replace(/>/g, "&gt;")
            .replace(/"/g, "&quot;")
            .replace(/'/g, "&#039;");
}

document.getElementById("p1").innerHTML = "Hello Hacker, " + escapeHTML(username) + "!";
```

This prevents JavaScript execution.
4. Implement Content Security Policy (CSP)
Block inline script execution:
Content-Security-Policy: default-src 'self'; script-src 'none';
This prevents injected JavaScript from running.
5. Implement a Web Application Firewall (WAF)
Detect and block malicious script injections in URL parameters.

References

Awesome XSS : https://github.com/s0md3v/AwesomeXSS
Cross Site Scripting by PortSwigger : https://portswigger.net/web-security/cross-site-scripting
OWASP XSS : https://owasp.org/www-community/attacks/xss/
OWASP DOM XSS Prevention Guide: https://owasp.org/www-community/attacks/DOM_Based_XSS
Mozilla's Guide on Secure JavaScript: https://developer.mozilla.org/en-US/docs/Web/Security/Information_Leakage

Proof of Concept

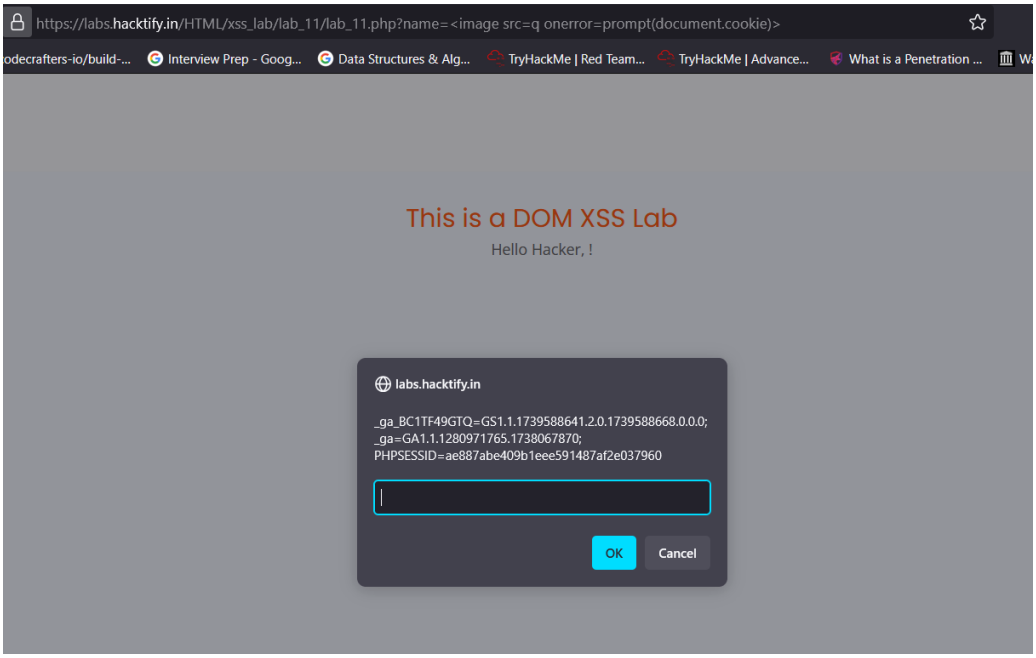
Scenario 1: Normal Input
https://labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?name=example
Expected Output:
Hello Hacker, example!
No XSS execution occurs.

Scenario 2: Injecting XSS Payload
https://labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?name=<script>alert('XSS')</script>
Observed Behavior:

The script executes, confirming DOM-Based XSS vulnerability.
Scenario 3: Stealing Cookies with XSS
https://labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php?name=<image src=q onerror=prompt(document.cookie)>

Observed Behavior:

The script executes, revealing session cookies.
Extracted Session ID:
PHPSESSID=ae887abe409b1eee591487af2e037960
Confirms session hijacking vulnerability.



Conclusion - XSS & HTML Injection Lab

The **XSS and HTML Injection vulnerabilities** identified in this lab demonstrate **critical security flaws** due to **improper input validation and output encoding**. Attackers can exploit these weaknesses to **steal session cookies, inject malicious scripts, hijack user accounts, and alter webpage content**.

Key Takeaways:

- **Stored, Reflected, and DOM-Based XSS** were successfully exploited.
- **Session hijacking, phishing attacks, and malware injection** are major risks.
- **Lack of input sanitization and direct DOM manipulation** enable these attacks.