

Penetration Testing Report

Full Name: **KAOUTHAR BELKEBIR**
Program: **HCPT**
Date:**15/02/2025**

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 {#} 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: {count} Sub-labs

High	Medium	Low
{4}	{4}	{7}

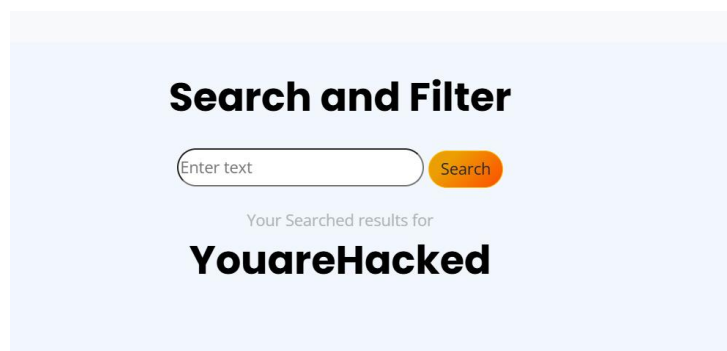
High	-	4
Medium	-	4
Low	-	7

1. {{HTML Injection}}

1.1. {HTML's Are Easy!}

Reference	Risk Rating
{HTML's Are Easy!}	Low
Tools Used	
MANUAL	
Vulnerability Description	
HTML injection is a type of vulnerability that occurs when a website allows user input to be directly included on its pages without proper validation or encoding. This can allow attackers to inject arbitrary HTML code into the web page, which will be rendered and executed by the browser of anyone viewing the infected page. The consequences of such an attack can range from benign, such as modifying the appearance of the webpage, to malicious activities like stealing user session cookies, redirecting the user to phishing sites, or executing scripts in the context of the website (cross-site scripting, or XSS).	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_1/html_injection_1.php	
Consequences of not Fixing the Issue	
<ul style="list-style-type: none">Attackers can exploit this vulnerability to inject malicious code into a web page, steal sensitive information, or compromise the security of a web application. This can result in data breaches, loss of sensitive information, and damage to the reputation of the organization	
Suggested Countermeasures	
To mitigate HTML injection risks, it's essential to enforce strict input validation, employ output encoding to neutralize malicious code, use secure APIs that automatically handle input safely, implement Content Security Policy (CSP) to restrict resource loading, and conduct regular security audits. Additionally, educating developers on secure coding practices is crucial. These measures collectively help safeguard web applications against HTML injection, protecting both the integrity of the websites and the privacy of their users	
References	
https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/11-Client_Side_Testing/03-Testing_for_HTML_Injection	

Proof of Concept



1.2. {Let Me Store Them!}

Reference	Risk Rating
{Let Me Store Them!}	Low
Tools Used	
Manual	

Vulnerability Description
Stored HTML injection, a subtype of HTML injection, occurs when malicious HTML code is permanently stored on a target server, such as in a database, message forum, visitor log, or comment field. This code is then served to users whenever they access the affected page. The consequences of such an attack can be more severe than those of a typical HTML injection because the malicious code can affect multiple users over a longer period without the need for repeated injection attempts.
How It Was Discovered
Manual Analysis
Vulnerable URLs
https://labs.hacktify.in/HTML/html_lab/lab_2/register.php
Consequences of not Fixing the Issue
Stored HTML injection can lead to persistent cross-site scripting (XSS) attacks, where attackers can steal cookies, session tokens, or other sensitive information from users. It can also allow attackers to deface websites, spread malware, perform phishing attacks, and potentially gain unauthorized access to user accounts or sensitive data stored on the server.
Suggested Countermeasures
To defend against stored HTML injection, it's crucial to rigorously validate and sanitize user inputs before storing them, encode outputs to neutralize malicious HTML code, implement Content Security Policy (CSP) for added security, and regularly perform security audits to identify vulnerabilities. Educating developers on secure coding practices is also essential to prevent such attacks. These measures collectively help safeguard web applications from the persistent threats posed by stored HTML injection, ensuring the protection of user data and maintaining the integrity of the web platform.
References
https://www.esecforte.com/responsible-vulnerability-disclosure-cve-2019-12863-stored-html-injection-vulnerability-in-solarwinds-orion-platform-2018-4-hf3-npm-12-4-netpath-1-1-4/

Proof of Concept



User Profile

First Name:

"/>

Last Name:

"/>

Email:

Password:

Confirm Password:

UpdateLog out

1.3. {File Names are also vulnerable!}

Reference	Risk Rating
{File Names are also vulnerable!}	Low
Tools Used	
Manual	
Vulnerability Description	
File name vulnerability in HTML injection refers to a specific scenario where an application allows users to upload files with names that include HTML code or scripts. When these file names are displayed on a webpage without proper sanitization or encoding, the embedded HTML or script can be executed in the context of the user's browser. This form of vulnerability is particularly concerning because it combines the risks associated with file uploads and HTML injection, potentially leading to cross-site scripting (XSS) attacks or other malicious outcomes.	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_3/html_injection_3.php	
Consequences of not Fixing the Issue	
Execute scripts in the context of other users' sessions, leading to session hijacking, data theft, or spreading malware. Phish for sensitive information by injecting forms or redirecting users to malicious sites. Deface the website by injecting arbitrary HTML content.	
Suggested Countermeasures	
To counter file name vulnerabilities in HTML injection, enforce strict input validation to filter or sanitize file names during uploads, use output encoding to neutralize executable code when displaying file	

names on web pages, and implement Content Security Policy (CSP) to restrict script execution. These strategies collectively strengthen web application defenses, preventing attackers from exploiting these vulnerabilities to conduct malicious activities.

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

Proof of Concept

Upload a File

Choose File

No file chosen

File Upload

File Uploaded JAVA.txt

```
<nav class="navbar navbar-expand-lg navbar-light
<section class="pager-section">
  <div class="container">
    <center>
      <div class="containers">
        <div class="contain">
          <h1>Upload a File</h1>
          <br>
          <div class="welcome-image"><img alt="Welcome image" data-bbox="300 780 400 800"/></div>
        <center>
          <br>
          <h2>
            "File Uploaded "
            <b>JAVA.txt</b> == $0
          </h2>
```

1.4. {File Content and HTML Injection a perfect pair!}

Reference	Risk Rating
{File Content and HTML Injection a perfect pair!}	Medium
Tools Used	
Manual	
Vulnerability Description	
File Content and HTML Injection vulnerabilities occur when an application allows user-supplied content, which can include malicious HTML or script code, to be uploaded and then served or executed without proper sanitization. This can happen in various contexts, such as uploading documents, images with metadata, or any file type where the content or metadata is not securely validated or encoded before being displayed to users or executed by the web application.	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_4/html_injection_4.php	
Consequences of not Fixing the Issue	
The consequences of these vulnerabilities are similar to those of traditional HTML injection, but with potentially broader impact due to the nature of file content being served or executed. Cross-site	

Scripting (XSS) attacks, where attackers execute scripts in the context of other users' sessions. Phishing attacks by injecting malicious content that appears legitimate. Session hijacking and redirection to malicious sites. Stealing or manipulating sensitive information. Defacing websites or displaying unauthorized content.

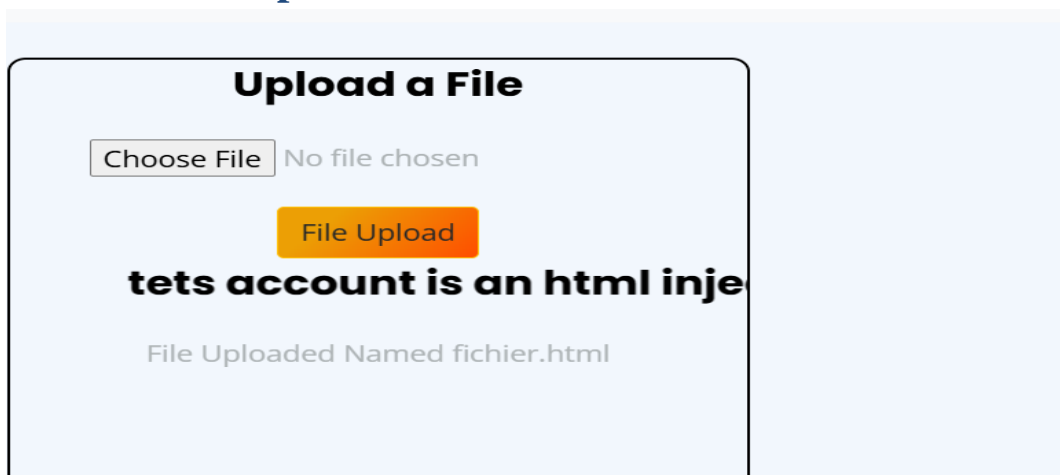
Suggested Countermeasures

To mitigate File Content and HTML Injection vulnerabilities, ensure comprehensive validation and sanitization of all user-supplied content, including file uploads and metadata, to prevent malicious code execution. Implementing Content Security Policy (CSP) is crucial for limiting the impact of any injection, by restricting resource loading and script execution from unauthorized sources. Secure file handling practices, such as isolating user-uploaded content on a separate domain and encoding all output displayed to users, are essential. Regular security assessments of file upload and handling processes further help in identifying and addressing vulnerabilities, solidifying the security of web applications against these types of attacks.

References

<https://www.acunetix.com/vulnerabilities/web/html-injection/>



Proof of Concept



1.5. {Injecting HTML using URL!}

Reference	Risk Rating
{Injecting HTML using URL!}	Medium
Tools Used	
Manual	
Vulnerability Description	
HTML injection through URL parameters is a web security vulnerability that occurs when a web application accepts unvalidated or unencoded user input from the URL query string and dynamically includes it in the webpage content. Attackers can exploit this by crafting a malicious URL with HTML or JavaScript code as part of the query parameters. When a user visits this URL, the code is executed within the context of the webpage, leading to potential security breaches such as Cross-Site Scripting (XSS).	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_5/html_injection_5.php	
Consequences of not Fixing the Issue	
HTML injection through URL parameters can have severe consequences, including Cross-Site Scripting (XSS) attacks that allow attackers to steal sensitive information, hijack user sessions, and spread malware. This vulnerability undermines the security and integrity of web applications, eroding user trust and potentially exposing organizations to legal and financial repercussions. It emphasizes the critical need for robust input validation and output encoding practices to protect against such security threats.	
Suggested Countermeasures	
To guard against HTML injection via URL parameters, enforce strict input validation to ensure only expected data is processed. Utilize output encoding to escape HTML entities, preventing execution of malicious code. Implementing a Content Security Policy (CSP) can also help mitigate potential attacks by restricting the sources from which scripts can be executed. These measures, combined with regular security audits and adopting secure coding practices, can significantly reduce the risk of HTML injection, safeguarding both the application and its users.	
References	
https://www.cgisecurity.com/articles/xss-faq.shtml	

Proof of Concept



Your URL is

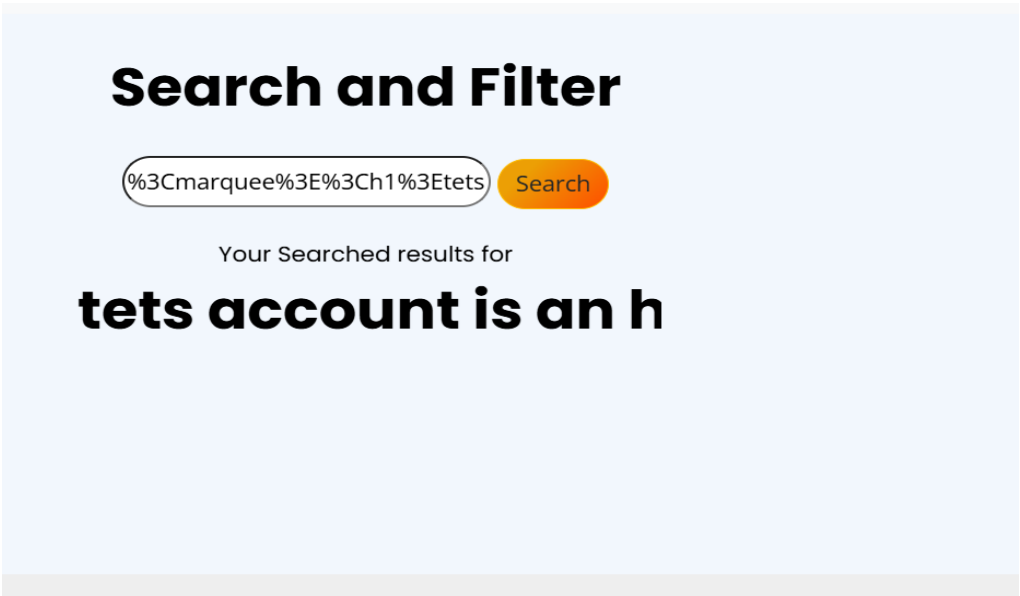
http://labs.hacktify.in/HTML/html_lab/lab_5/html_injection_5.php?=

this account was be hacked

1.6. {Encode IT!}

Reference	Risk Rating
{Encode IT!}	High
Tools Used	
Manual	
Vulnerability Description	
<p>Blind SSRF attacks are typically carried out by sending a crafted request to a vulnerable web application. The request is designed to cause the web application to make a request to an external server. The attacker can then use the response from the external server to infer information about the web application's internal network or to carry out further attacks.</p>	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/html_lab/lab_6/html_injection_6.php	
Consequences of not Fixing the Issue	
<p>Ignoring blind Server-Side Request Forgery (SSRF) vulnerabilities can have dire consequences, including unauthorized access to internal networks, bypassing of security mechanisms, data breaches, service disruptions, and potential server compromise. Such vulnerabilities expose sensitive systems to attackers, leading to regulatory and compliance issues, financial penalties, and reputational damage.</p>	
Suggested Countermeasures	
<p>To mitigate blind SSRF vulnerabilities, adopt strict input validation and sanitization measures, and enforce allowlists for outbound requests to limit interactions to trusted destinations. Employ network segmentation and the principle of least privilege to restrict server access within the network. Additionally, implement robust monitoring and logging of outbound requests to quickly identify and respond.</p>	
References	
https://portswigger.net/web-security/ssrf/blind	

Proof of Concept

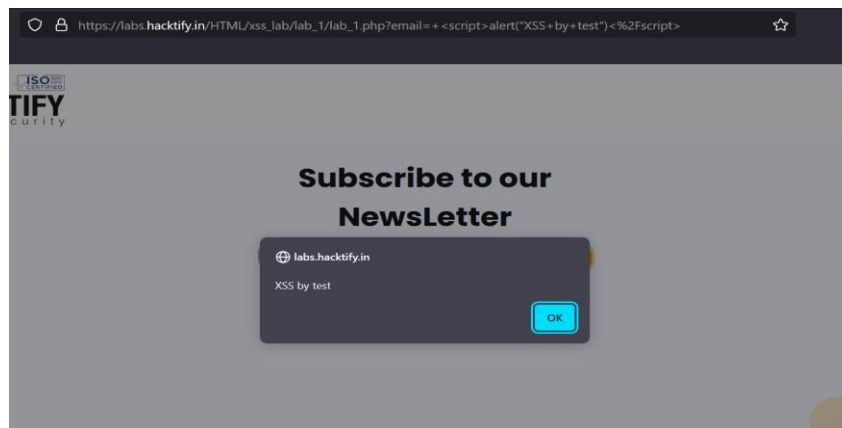


2. Cross Site Scripting

2.1. {Let's Do It!}

Reference	Risk Rating
Let's do it?	Low
Tools Used	
Firefox Browser	
Vulnerability Description	
Reflected Cross-Site Scripting (XSS) is a type of security vulnerability that occurs in web applications. It allows an attacker to inject malicious scripts into the content of a trusted website, which are then executed by the victim's browser. Reflected XSS attacks are typically delivered via email or other web-based communication that includes a link with malicious script in the URL. When the victim clicks on the link, the malicious script is sent to the vulnerable website, which reflects the attack back to the victim's browser. The browser then executes the script because it appears to be a script from the trusted website.	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_1/lab_1.php	
Consequences of not Fixing the Issue	
Not fixing a Reflected Cross-Site Scripting (XSS) vulnerability can lead to severe consequences, including theft of sensitive information, account takeovers, phishing attacks, and malware distribution. Such vulnerabilities compromise user trust and can damage an organization's reputation. Additionally, there may be legal and financial repercussions, especially if the breach involves personally identifiable information. Ensuring web applications are secure against XSS attacks is crucial to protect both users and organizations from these potential impacts.	
Suggested Countermeasures	
To counteract Reflected XSS vulnerabilities, it's essential to adopt secure coding practices that include validating and sanitizing all user inputs, encoding outputs to prevent malicious content from executing, and implementing Content Security Policies (CSP) to restrict the sources of executable scripts. Additionally, employing Web Application Firewalls (WAFs) can help detect and block XSS attacks. Regular security testing and keeping software up to date are also critical measures to identify vulnerabilities early and mitigate potential risks effectively.	
References	
https://portswigger.net/web-security/cross-site-scripting/reflected#:~:text=What%20is%20reflected%20cross%2Dsite,response%20in%20an%20unsafe%20way.	

Proof of Concept



2.2. { Balancing is Important in Life!}

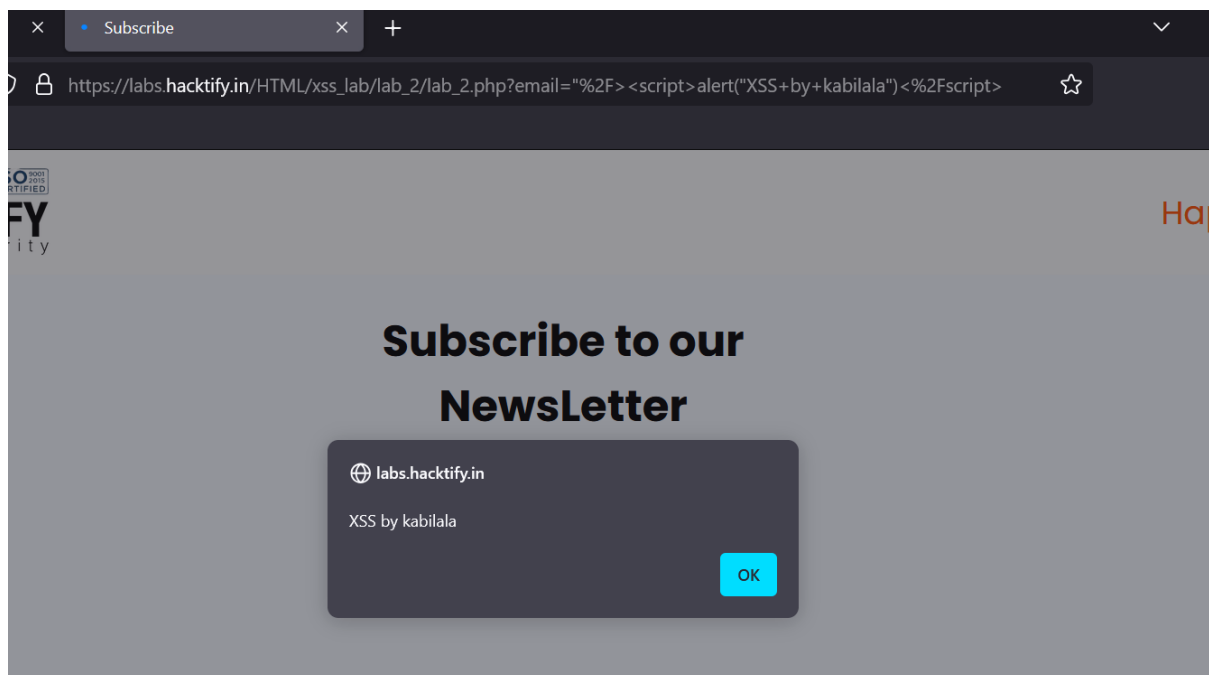
Reference	Risk Rating
{ Balancing is Important in Life!}	Low
Tools Used	
Firefox Browser	
Vulnerability Description	
<p>The XSS payload : <code>"</><script>alert("XSS by kabilala")</script></code> is a simple but effective demonstration of a Cross-Site Scripting attack, where <code>"<script></code> breaks out of HTML context to inject a JavaScript <code><script></code> tag, <code>alert(...)</code> executes a JavaScript alert function, showing the vulnerability to arbitrary script execution. This highlights the importance of sanitizing and validating user inputs in web applications to prevent attackers from injecting malicious scripts that can lead to data theft, session hijacking, and other security breaches.</p>	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_2/lab_2.php	
Consequences of not Fixing the Issue	
<p>Not fixing an XSS (Cross-Site Scripting) vulnerability can have serious consequences for a website and its users. This security flaw allows attackers to inject malicious scripts into webpages viewed by other users. If not addressed, it can lead to a wide range of issues including theft of cookies, session tokens, or other sensitive information that the browser stores. Attackers can also manipulate or deface the website content displayed to users, redirect visitors to malicious sites, and perform actions on behalf of users without their consent. This not only compromises user security and privacy but can also damage the reputation of the website, erode user trust, and potentially lead to legal and financial repercussions for the entity responsible for the website.</p>	
Suggested Countermeasures	
<p>To counter XSS vulnerabilities, it's crucial to adopt a multifaceted approach. Primarily, input sanitization and validation should be enforced, ensuring that user inputs are checked, cleaned, and confirmed as safe before being used or displayed. Employing Content Security Policy (CSP) headers can also significantly reduce the risk by specifying which sources are trusted, thus preventing the browser from executing malicious scripts from unauthorized sources. Escaping user input is essential to ensure that any data received is treated as data, not executable code, especially in places where input is inserted, into HTML, JavaScript, or database queries. Regular security audits and code reviews can help identify and rectify potential vulnerabilities. Additionally, using frameworks and libraries that automatically handle these security measures can further safeguard against XSS attacks by reducing</p>	

the amount of security-critical code developers need to write themselves.

References

- OWASP XSS Prevention Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/XSS_Prevention_Cheat_Sheet.html
- Mozilla Developer Network (MDN) - XSS: https://developer.mozilla.org/en-US/docs/Glossary/Cross-site_scripting
- Hacktify Labs: <https://labs.hacktify.in>

Proof of Concept

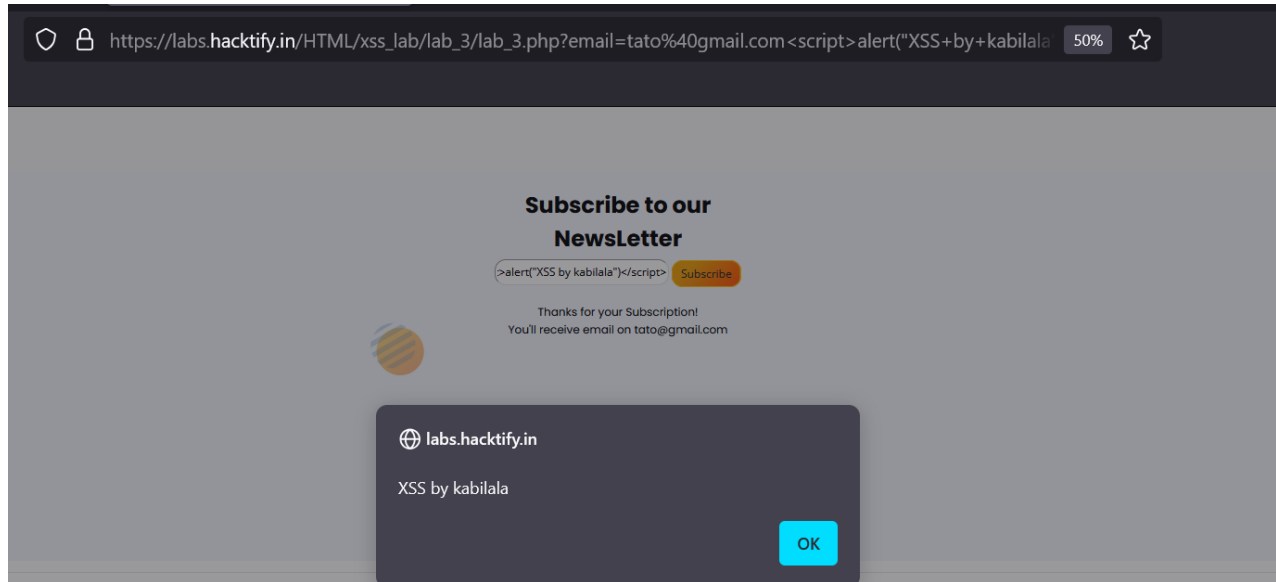


2.3. XSS Is Everywhere!

Reference	Risk Rating
XSS Is Everywhere!	Low
Tools Used	
Firefox Browser	
Vulnerability Description	
<p>Using regular expressions (regex) to detect XSS (Cross-Site Scripting) attacks involves crafting patterns that identify typical malicious inputs, such as <code><script></code>, <code>javascript:</code>, and other tags or attributes commonly used in these attacks. While regex can help flag potential XSS injections by searching for these patterns, it's not a foolproof method due to the sophistication and variety of attack vectors. False positives and negatives are common, and regex cannot account for the context of user input, making it a limited tool in XSS defense. Effective XSS prevention requires a comprehensive approach that includes input validation, output encoding, employing Content Security Policy (CSP), and adhering to secure coding practices. Relying solely on regex is insufficient; it should be part of a layered security strategy that is continuously updated to address new vulnerabilities.</p>	
How It Was Discovered	
Manual Analysis	

Vulnerable URLs
https://labs.hacktify.in/HTML/xss_lab/lab_3/lab_3.php
Consequences of not Fixing the Issue
<p>Not addressing XSS (Cross-Site Scripting) vulnerabilities can lead to severe consequences, including the theft of sensitive information such as cookies, session tokens, and personal data. Attackers can exploit these vulnerabilities to execute malicious scripts in the browsers of unsuspecting users, potentially leading to unauthorized actions on their behalf, defacement of web content, and redirection to malicious sites. This not only compromises the security and privacy of users but also damages the reputation of the affected website, eroding trust among its visitors. Furthermore, the website owners might face legal and financial repercussions for failing to safeguard user data. In summary, neglecting XSS vulnerabilities exposes both users and organizations to significant risks.</p>
Suggested Countermeasures
<p>Mitigating XSS vulnerabilities through regex involves carefully crafting expressions that identify and neutralize potential attack vectors in user input. However, solely relying on regex for XSS defense is not advisable due to its limitations in handling complex and obfuscated attacks. A more effective approach includes:</p> <p>Input Sanitization: Cleanse user input by removing or escaping potentially harmful characters, especially in contexts where HTML or JavaScript could be executed.</p> <p>Content Security Policy (CSP): Implement CSP to control the sources from which content can be loaded, effectively reducing the risk of executing unauthorized scripts.</p> <p>Output Encoding: Encode output data to ensure that any input reflected back to the user is treated as data, not executable code.</p> <p>Use of Secure Frameworks: Employ frameworks and libraries that automatically apply these security measures to reduce the risk of developer oversight.</p> <p>Regular Security Training: Educate developers on secure coding practices and the importance of regular updates to security measures.</p>
References
<ul style="list-style-type: none">OWASP XSS Prevention Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/XSS_Prevention_Cheat_Sheet.html

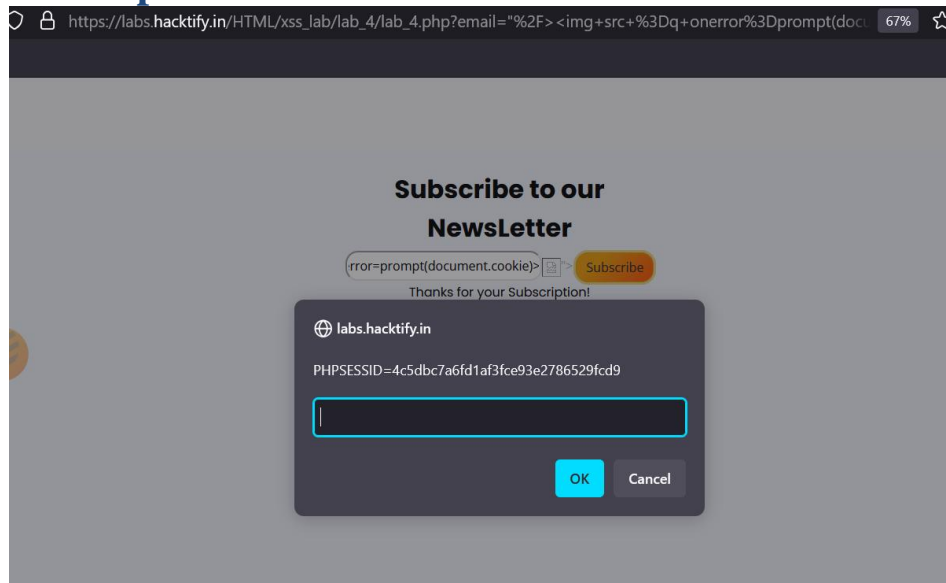
Proof of Concept



2.4. Alternatives Are Must!

Reference	Risk Rating
Alternatives Are Must!	Medium
Tools Used	
Firefox Browser	
Vulnerability Description	
<p>Payload: <code>"></code></p> <p>This payload appears to be an example of an XSS (Cross-Site Scripting) payload designed to demonstrate how malicious scripts can be injected into web applications. This specific payload attempts to execute JavaScript code (<code>confirm(1)</code>) when interpreted by a web browser, which could be part of an attack aimed at exploiting XSS vulnerabilities in a web application. In this case, if a web application improperly handles user input and directly includes this string in its output (for example, in an HTML page), the JavaScript code within the <code><script></code> tags could be executed in the context of the user's session. This execution could lead to various security issues, such as stealing cookies, session hijacking, redirecting the user to a malicious website, or performing actions on behalf of the user without their consent.</p>	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_4/lab_4.php	
Consequences of not Fixing the Issue	
<p>Not addressing the XSS vulnerability demonstrated by the payload <code>"></code> can have significant consequences. This specific attack vector can be used to execute arbitrary JavaScript code in the context of an unsuspecting user's browser. If exploited, it could lead to unauthorized actions such as session hijacking, where attackers gain control over a user's active session. It could also lead to the theft of sensitive information, including personal data and authentication credentials, as the script has the potential to access cookies and local storage. Additionally, this vulnerability can undermine the integrity and reputation of the affected website, eroding user trust and potentially resulting in legal and financial repercussions for failing to protect user data. It highlights the critical need for diligent web application security practices, including input validation and output encoding, to safeguard against XSS attacks.</p>	
Suggested Countermeasures	
<p>To effectively counter the XSS vulnerability exemplified by the payload <code>"></code>, a multi-layered security approach is essential. First, input validation should be rigorously applied to reject or sanitize inputs containing potentially dangerous characters or patterns. Second, output encoding is crucial; all user-supplied data displayed on web pages should be HTML-encoded to prevent the browser from interpreting it as executable code. Implementing Content Security Policy (CSP) can further restrict the execution of scripts to only trusted sources, significantly reducing the risk of XSS attacks. Additionally, adopting secure coding practices and frameworks that automatically handle these security concerns can help prevent such vulnerabilities from arising. Regular security audits and vulnerability assessments are also recommended to identify and remediate any potential XSS vulnerabilities proactively.</p>	
References	
<ul style="list-style-type: none">OWASP XSS Prevention Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/XSS_Prevention_Cheat_Sheet.html	

Proof of Concept



2.5. Developer Hates Scripts!

Reference	Risk Rating
Developer Hates Scripts!	High
Tools Used	
Firefox Browser	
Vulnerability Description	
<p>The XSS payload <code>"></code> is designed to exploit Cross-Site Scripting vulnerabilities by injecting malicious JavaScript code into a web page. This payload works by inserting an <code></code> tag with an invalid <code>src</code> attribute (<code>q</code>) and leveraging the <code>onerror</code> event to execute <code>prompt(document.cookie)</code>. This technique can be used to steal cookies, hijack user sessions, and execute arbitrary JavaScript within a victim's browser. The payload is particularly dangerous because it can bypass certain basic security filters that do not properly escape user input or fail to implement Content Security Policy (CSP) restrictions.</p>	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_5/lab_5.php	
Consequences of not Fixing the Issue	
<p>Failure to address this XSS vulnerability can have severe security consequences, including:</p> <ul style="list-style-type: none">Session Hijacking: Attackers can steal session cookies, allowing unauthorized access to user accounts.Data Theft: Sensitive information stored in cookies, such as authentication tokens, can be stolen.Website Defacement: Attackers can modify the appearance or content of the website.User Redirection: Users may be redirected to malicious sites leading to phishing attacks or malware infections.Reputation Damage: The organization may lose user trust and face legal and financial consequences.	
Suggested Countermeasures	

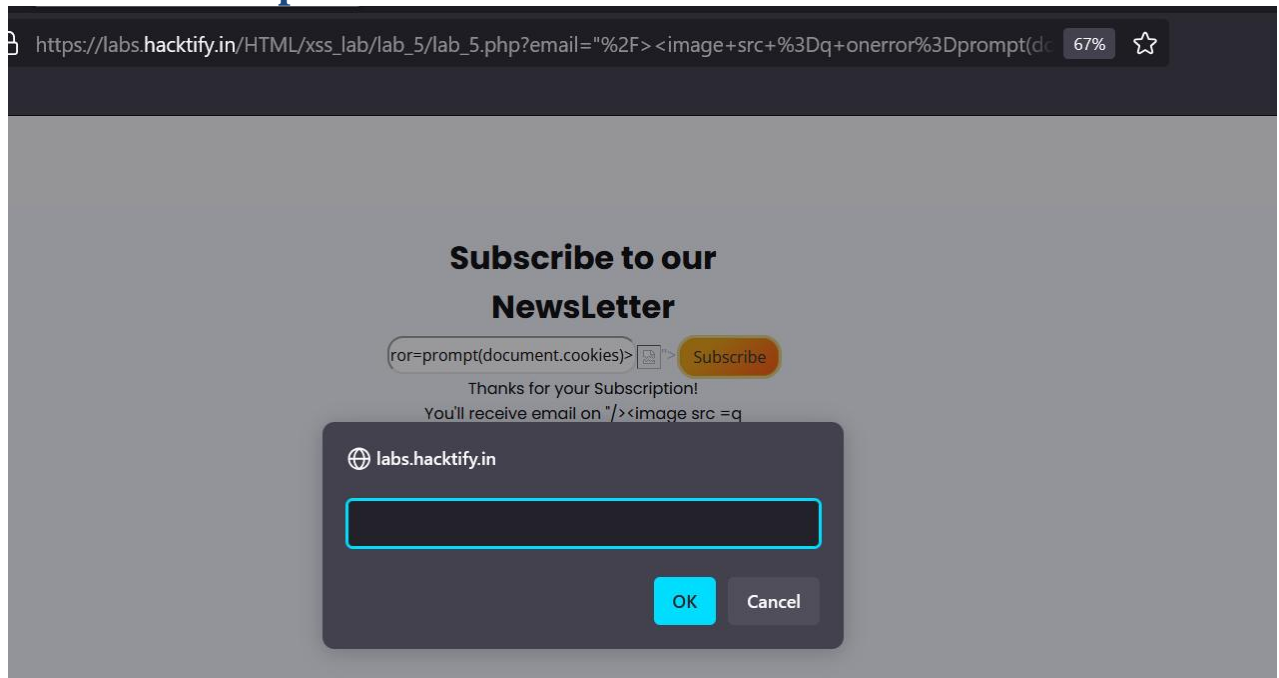
to prevent XSS attacks, it is essential to implement robust security measures, including:

- Input Sanitization & Validation:** Ensure user input is properly filtered and encoded before being stored or displayed.
- Content Security Policy (CSP):** Restrict the execution of inline scripts by allowing only trusted sources.
- Output Encoding:** Escape user-generated content to prevent script execution in HTML, JavaScript, and URL contexts.
- HTTPOnly & Secure Cookies:** Mark cookies as HttpOnly to prevent JavaScript access and use the Secure flag to enforce HTTPS-only transmission.
- Regular Security Audits:** Conduct penetration testing and code reviews to detect and mitigate vulnerabilities before exploitation.

References

<https://www.hackerone.com/knowledge-center/how-xss-payloads-work-code-examples-preventing-them>

Proof of Concept



The XSS payload `" /><image src =q onerror=prompt(document.cookies)>` exploits a Cross-Site Scripting (XSS) vulnerability by injecting an `<image>` tag into a vulnerable web page. This payload works by:

Using the `<image>` tag instead of ``, which can bypass some security filters.

Setting an invalid `src` attribute (`q`) to trigger the `onerror` event.

Executing `prompt(document.cookies)`, which can be used to steal a user's session cookies.

When injected into a vulnerable website that fails to properly sanitize input, this payload can execute malicious JavaScript within a user's browser session. This makes it possible for attackers to steal session cookies, hijack accounts, and execute unauthorized actions on behalf of victims.

How It Was Discovered

Manual Analysis

Vulnerable URLs

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

Consequences of not Fixing the Issue

Failure to mitigate this XSS vulnerability can result in:

Session Hijacking: Attackers can steal session tokens, allowing them to impersonate users.

Data Theft: User credentials, personal data, and authentication tokens may be stolen.

Website Defacement: Attackers can modify website content, misleading users or damaging the brand's reputation.

Phishing Attacks & Malware Distribution: Users can be redirected to malicious websites.

Legal and Financial Risks: Organizations may face legal actions and financial losses due to data breaches.

Suggested Countermeasures

To prevent XSS attacks, implement the following security measures:

Input Validation & Sanitization: Ensure that all user inputs are properly sanitized to remove or neutralize potentially harmful code.

Output Encoding: Encode any user-generated content before displaying it to prevent script execution.

Content Security Policy (CSP): Restrict the execution of inline scripts and allow only trusted sources.

Use HTTPOnly & Secure Cookies: Mark cookies as `HttpOnly` to prevent JavaScript access and enforce Secure cookies over HTTPS.

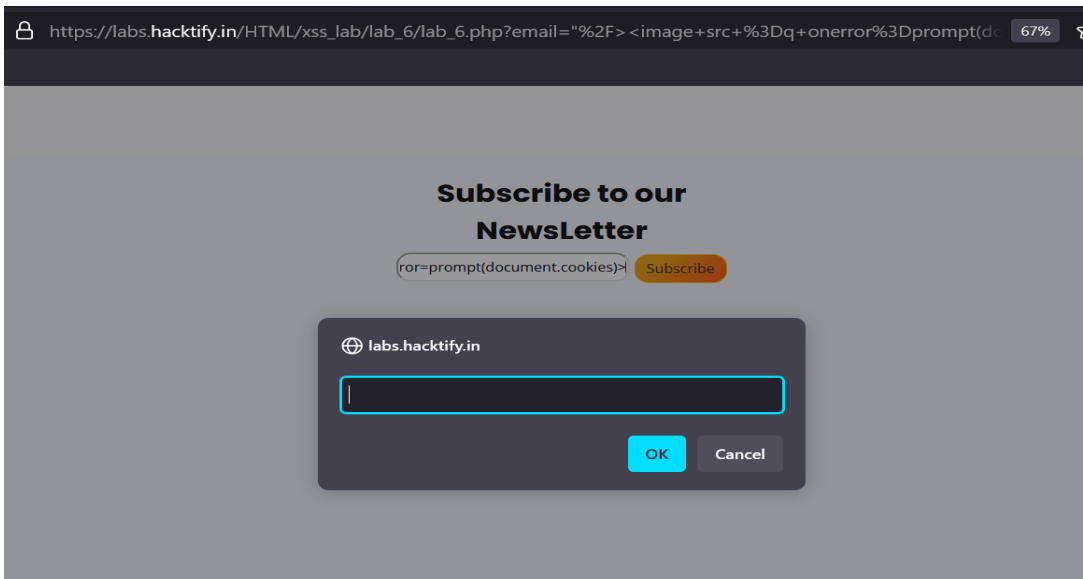
Adopt Secure Web Frameworks: Use modern frameworks that automatically handle XSS protections.

Regular Security Audits: Perform penetration testing and code reviews to detect vulnerabilities early.

References

XSS Prevention Cheat Sheet-/-OWASP XSS Filter Evasion Cheat Sheet

Proof of Concept



2.7. Encoding Is The Key?

Reference	Risk Rating
Encoding Is The Key?	Medium
Tools Used	
Firefox Browser	
Vulnerability Description	
<p>Payload: %3Cscript%3Ealert%281%29%3C%2Fscript%3E%40gmail.com</p> <p>The encoded string %3Cscript%3Ealert%281%29%3C%2Fscript%3E%40gmail.com represents an XSS payload that has been URL-encoded to bypass basic security filters and input validation checks. When decoded, it translates to <script>alert(1)</script>@gmail.com, embedding a JavaScript <script> tag that executes the alert(1) function. This technique is often used to inject malicious scripts into web applications that do not properly decode or sanitize user inputs before rendering them on a page. The payload aims to exploit Cross-Site Scripting (XSS) vulnerabilities by executing arbitrary JavaScript code in the context of the victim's browser, potentially leading to unauthorized actions such as cookie theft, session hijacking, and personal data exposure. The presence of @gmail.com suggests an attempt to embed the payload within an email context, possibly to bypass filters that specifically look for malicious content in email addresses or inputs expecting email formats. This underscores the importance of thoroughly sanitizing and validating all user inputs, including decoding encoded strings, to protect against XSS attacks.</p>	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_7/lab_7.php	
Consequences of not Fixing the Issue	
<p>The encoded XSS payload %3Cscript%3Ealert%281%29%3C%2Fscript%3E%40gmail.com represents a URL-encoded form of a script injection attempt, where %3C, %3E, and other similar sequences are URL-encoded equivalents of <, >, and other characters crucial for HTML syntax. When decoded, it</p>	

translates to `<script>alert(1)</script>@gmail.com`, aiming to execute a JavaScript `alert(1)` function within a web page. This type of payload can bypass simple input validation mechanisms that do not decode input before checking for malicious patterns. By embedding such encoded scripts in inputs that are reflected back into web pages, attackers can execute arbitrary JavaScript in the context of the user's browser session. This technique underscores the sophistication of XSS attacks and the necessity for web applications to implement robust decoding and sanitization routines in their input processing logic to prevent the execution of malicious scripts.

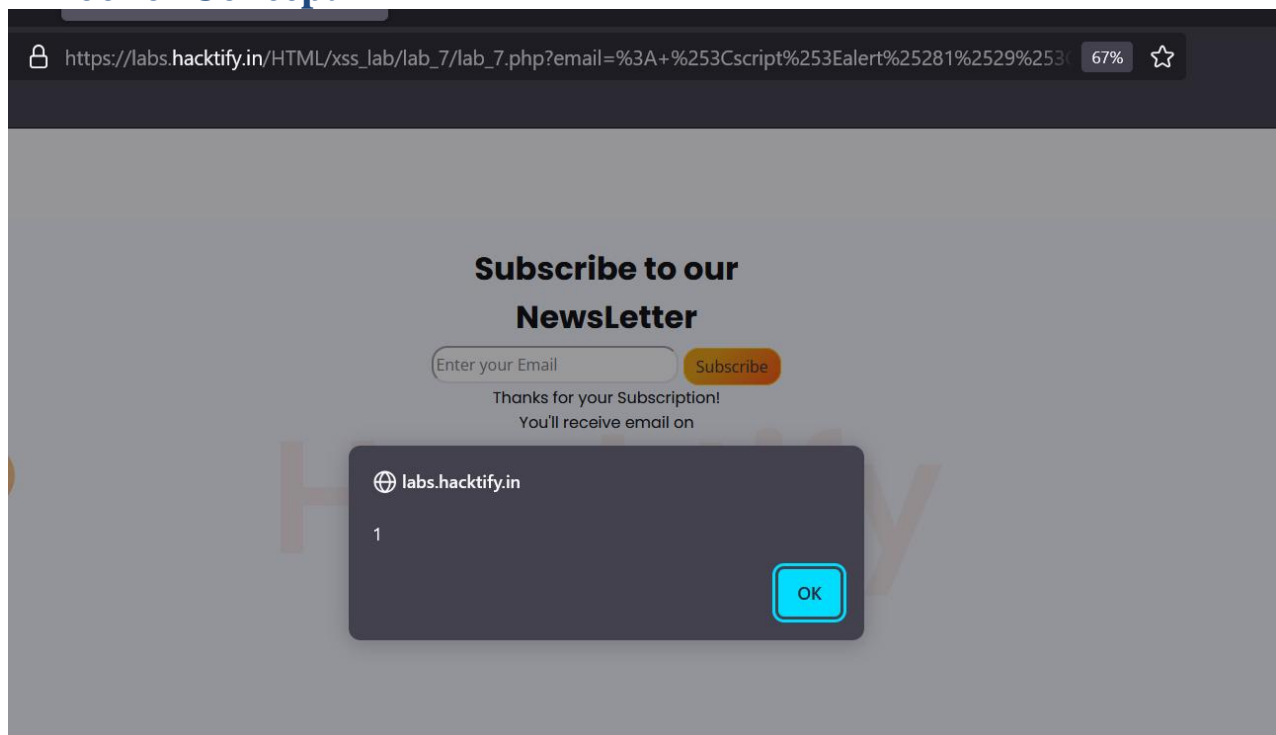
Suggested Countermeasures

The payload `%3Cscript%3Ealert%281%29%3C%2Fscript%3E%40gmail.com` uses URL encoding to disguise a script injection attempt as part of an email address. If not properly handled, it can enable attackers to execute arbitrary JavaScript in users' browsers, leading to security breaches such as data theft, session hijacking, and malicious redirection. To mitigate these risks, it's essential to sanitize and validate inputs, decode encoded inputs before processing, and implement Content Security Policy (CSP). Employing secure coding practices, regular security audits, and utilizing frameworks that automatically manage XSS protections are also key strategies in defending against such encoded XSS attacks, ensuring the safety of web applications and their users.

References

<https://medium.com/@zhyarr/double-url-encoded-xss-c3f72ff32ea8>

Proof of Concept



2.8. XSS with File Upload (File Name)

Reference	Risk Rating
XSS with File Upload (File Name)	Low
Tools Used	
Firefox Browser	
Vulnerability Description	

Payload: filename=">"test.txt"

This payload could be used to exploit a system where filenames or file paths are dynamically inserted into the HTML output without proper sanitization or encoding. For instance, if a web application displays user-uploaded file names directly in the HTML and an attacker uploads a file with the name ``, the browser would attempt to execute the JavaScript within the context of the page, leading to an XSS attack.

How It Was Discovered

Manual Analysis

Vulnerable URLs

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

Consequences of not Fixing the Issue

The consequences of such an attack can vary from minor nuisances to significant security breaches, including stealing cookies, session tokens, or other sensitive information from users, redirecting users to malicious sites, or even performing actions on behalf of the users without their consent.

Suggested Countermeasures

To mitigate this type of vulnerability, it is crucial to:

Sanitize and validate all user inputs, filenames included, to ensure that potentially harmful characters or code snippets are neutralized or rejected.

Encode or escape output to prevent the browser from interpreting it as executable code rather than as plain text or data.

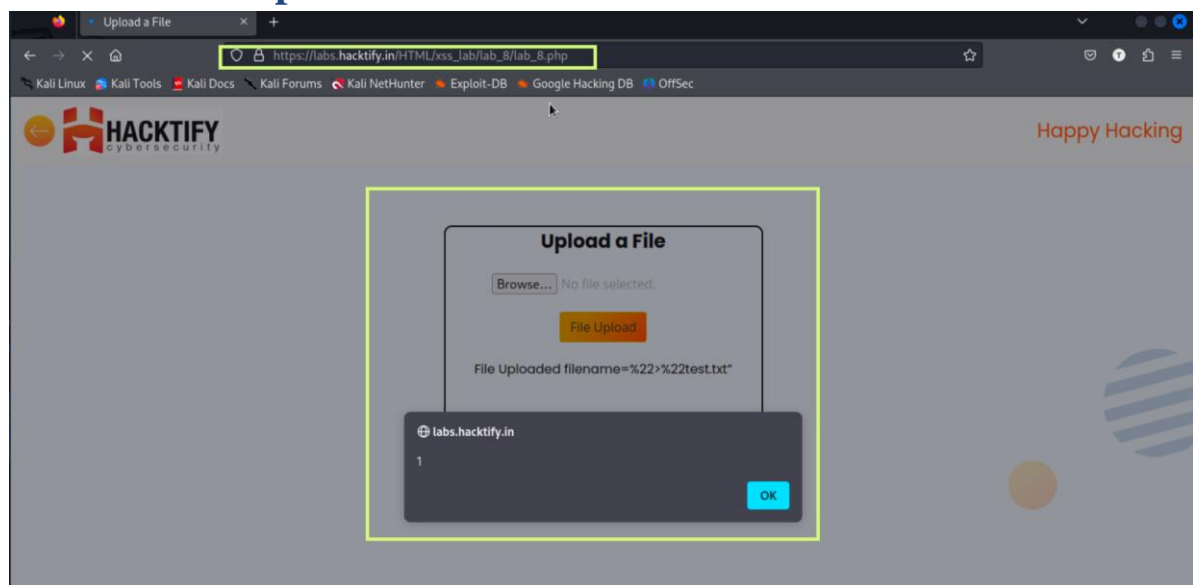
Adopt a Content Security Policy (CSP) to add an additional layer of protection by specifying which sources are valid for executing scripts, thus limiting the potential for XSS attacks.

Regularly conduct security reviews and vulnerability assessments to identify and rectify potential XSS vectors in existing applications.

References

<https://medium.com/@sarang6489/file-upload-xss-using-filename-f2f53e10033d>

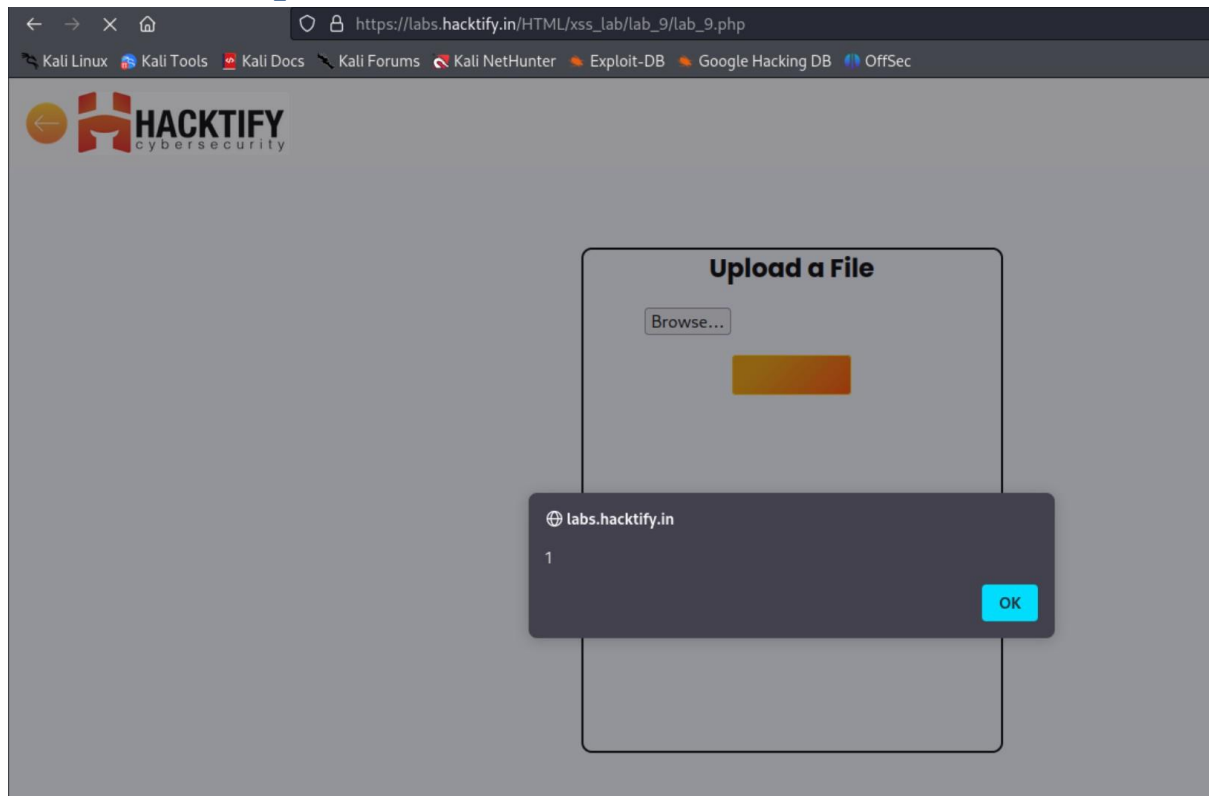
Proof of Concept



2.9. XSS with File Upload (File Content)

Reference	Risk Rating
XSS with File Upload (File Content)	Medium
Tools Used	
Firefox Browser	
Vulnerability Description	
<p>Payload:<script>alert(1)</script> <script>confirm(1)</script> "><script>alert(1)</script> "><script>confirm(1)</script> "></p> <p>In a typical file upload XSS attack scenario, an attacker would first upload a specially crafted file that contains malicious JavaScript code. They would then attempt to execute the JavaScript code by tricking a victim user into visiting a URL that references the uploaded file.</p> <p>When the victim user's browser loads the malicious file, it will execute the JavaScript code contained within the file. This can potentially allow the attacker to perform a variety of malicious actions, such as stealing the victim user's session cookies, redirecting the victim user to a malicious website, or even executing arbitrary system commands on the victim user's machine.</p>	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_9/lab_9.php	
Consequences of not Fixing the Issue	
<p>Not fixing XSS vulnerabilities with file uploads can have serious consequences. Attackers can exploit these vulnerabilities to execute malicious scripts in the context of the victim's browser, leading to data breaches, theft of sensitive information, session hijacking, and potentially gaining unauthorized access to the system. This can damage the reputation of the organization, lead to legal consequences, and result in financial losses due to the compromise of user data and potential fines. It is essential to address these vulnerabilities promptly to maintain the security and integrity of the application and protect users' data.</p>	
Suggested Countermeasures	
<p>To mitigate XSS vulnerabilities with file uploads, consider the following countermeasures:</p> <p>Input Validation: Ensure that file uploads are strictly validated for type, size, and content. Only allow specific file types and reject any files that do not meet the criteria.</p> <p>Content Sanitization: Sanitize the content of uploaded files to remove any potentially malicious code. This can include stripping out JavaScript or other executable code from file contents.</p> <p>File Type Restrictions: Restrict the types of files that can be uploaded to reduce the risk of malicious files being accepted. For example, only allow image files (e.g., .jpg, .png) and reject executable files.</p> <p>Secure File Handling: Store uploaded files in a secure location, separate from the application's execution path. This prevents attackers from being able to execute uploaded files on the server.</p> <p>Content-Type Headers: Set appropriate Content-Type headers when serving uploaded files to ensure that browsers treat them as the intended file type, rather than executable code.</p>	
References	
https://medium.com/@ms-official5878/stored-xss-via-file-upload-svg-file-content-ba230c1448f6	

Proof of Concept



2.10. Stored Everywhere!

Reference	Risk Rating
Stored Everywhere!	Low
Tools Used	
Firefox Browser	
Vulnerability Description	
<p>Payload: "><script>alert(1)</script></p> <p>The XSS payload "><script>alert(1)</script>" is a simple example of a cross-site scripting attack. It works by breaking out of an HTML attribute context (e.g., an input value or a link) and injecting a <script> tag into the webpage. When the browser renders the page, it executes the JavaScript within the <script> tag, in this case, displaying an alert box with the number 1.</p>	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_10/lab_10.php	
Consequences of not Fixing the Issue	
If the Cross-Site Scripting (XSS) vulnerability that allows the injection of the <script>alert(1)</script> payload is not properly addressed and fixed, it can lead to a number of serious consequences.	

Unauthorized Access: An attacker could potentially use the XSS vulnerability to gain unauthorized access to sensitive information on the vulnerable web application.

Data Theft: An attacker could potentially steal sensitive data from the vulnerable web application, such as user credentials, session cookies, or other confidential information.

Session Hijacking: An attacker could potentially hijack a victim user's session by stealing their session cookies. This would allow the attacker to impersonate the victim user and perform actions on their behalf.

Redirection: An attacker could potentially redirect victim users to a malicious website, where they could be tricked into downloading malware, providing sensitive information, or performing other actions that could harm the victim users.

Suggested Countermeasures

To mitigate XSS vulnerabilities, such as the one demonstrated by the payload "><script>alert(1)</script>", consider the following countermeasures:

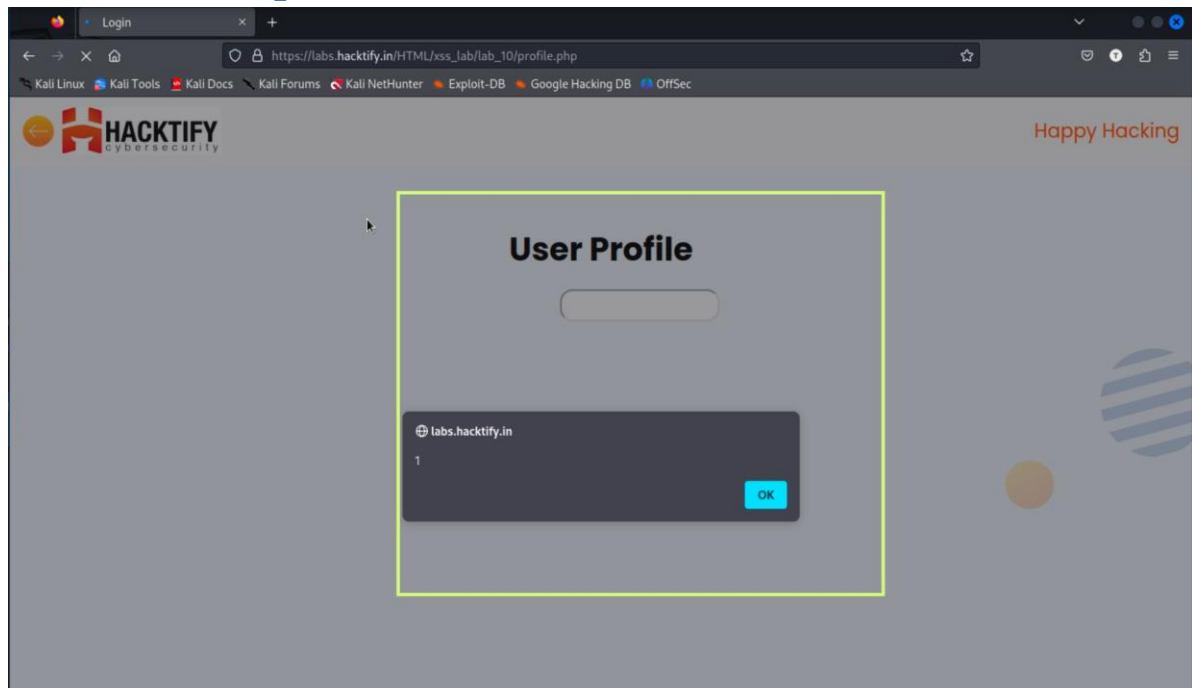
Input Validation: Validate user input on both the client and server sides to ensure that it conforms to expected formats and does not contain malicious content.

Output Encoding: Encode user-generated output before rendering it in the browser. This prevents potentially harmful characters from being interpreted as executable code.

References

<https://medium.com/@cyberw1ng/13-8-lab-stored-xss-into-anchor-href-attribute-with-double-quotes-html-encoded-2023-d9f698f43817>

Proof of Concept



2.11. DOM's are love!

Reference	Risk Rating
DOM's are love!	High
Tools Used	
Firefox Browser	
Vulnerability Description	
<p>Payload: ?name=</p> <p>The XSS payload ?name= is an example of a cross-site scripting attack that injects an HTML image tag with a malicious JavaScript event handler. In this payload:</p> <p>?name= is a query parameter that could be part of a URL.</p> <p> is the malicious script injected into the application.</p> <p>The payload includes an tag with a non-existent image source (src=x). The 'onerror' attribute is used to execute JavaScript code (alert(1)) when the image fails to load, which is guaranteed since x is not a valid image source. When the browser renders this injected code, it triggers the 'onerror' event and executes the JavaScript, resulting in an alert box displaying the number 1.</p>	
How It Was Discovered	
Manual Analysis	
Vulnerable URLs	
https://labs.hacktify.in/HTML/xss_lab/lab_11/lab_11.php	
Consequences of not Fixing the Issue	
<p>Not fixing the XSS vulnerability demonstrated by the payload ?name= can lead to several severe consequences:</p> <p>Data Theft: Attackers can use XSS attacks to steal sensitive information such as login credentials, personal data, or financial details from users.</p> <p>Session Hijacking: XSS can be exploited to steal session cookies, allowing attackers to impersonate users and gain unauthorized access to their accounts.</p> <p>Malware Distribution: Malicious scripts injected through XSS can be used to distribute malware to users' devices, leading to further security breaches.</p> <p>Phishing: XSS vulnerabilities can be leveraged to create convincing phishing attacks by injecting malicious content into trusted websites.</p>	
Suggested Countermeasures	
<p>For the specific payload ?name=, the same countermeasures apply. The payload attempts to inject an HTML image tag () with an 'onerror' attribute set to alert(1) JavaScript code. This can potentially allow the attacker to execute arbitrary JavaScript code in the context of the vulnerable web page. Therefore, it's important to implement the above mentioned countermeasures to help prevent such XSS attacks.</p>	
References	
https://medium.com/bugbountywriteup/write-up-dom-xss-in-innerhtml-sink-using-source-location-search-portswigger-academy-94c6691f89b0	

Proof of Concept

