Simulation and Analysis of Cross-Site Scripting (XSS) Vulnerability Using DVWA

Project Overview:-

This project is centered on the simulation and detailed analysis of a prevalent web application vulnerability known as **Cross-Site Scripting (XSS)**. Leveraging the intentionally vulnerable web platform **Damn Vulnerable Web Application (DVWA)**, we aim to explore the mechanics of XSS attacks in a controlled environment.

Objective:

The primary goal of this project is to understand how web applications can become vulnerable to XSS due to the failure to properly validate and sanitize user input. By exploiting these vulnerabilities within DVWA, we will:

- Demonstrate how malicious scripts can be injected and executed in a victim's browser.
- Analyze the impact and consequences of successful XSS attacks, including session hijacking, data theft, and interface manipulation.
- Document the observed behavior of such attacks.
- Design and propose effective incident response strategies and mitigation techniques to prevent or limit the damage from XSS vulnerabilities.

Relevance:

XSS remains one of the most common security flaws in modern web applications and consistently appears in the **OWASP Top 10** list. Through this simulation, we aim to enhance practical knowledge of web security and promote best practices for securing applications against client-side injection attacks.

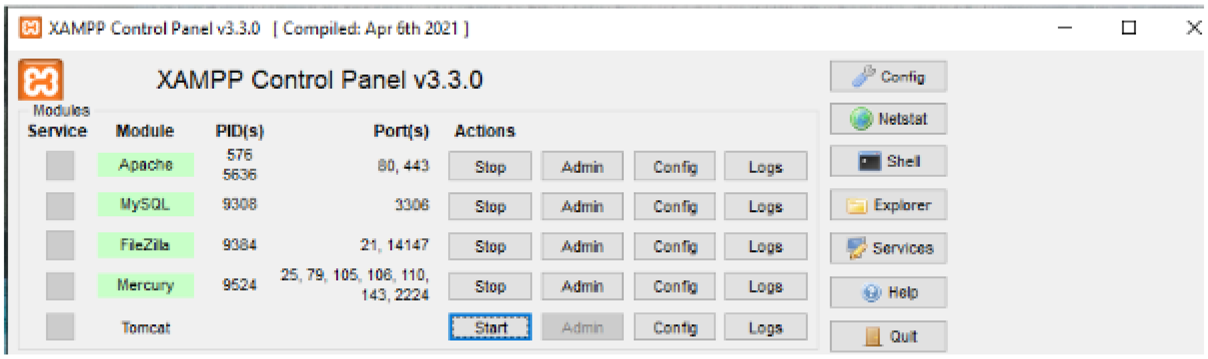
Attack and simulation

✓ **Setup Environment**:

- Installed and configured XAMPP to run an Apache web server and MySQL database locally.
- Downloaded and set up DVWA (Damn Vulnerable Web Application) in the XAMPP htdocs directory.
- Started Apache and MySQL services from the XAMPP control panel.
- Configured DVWA by accessing http://localhost/dvwa in the browser, setting up the database, and logging in using default credentials (admin / password).
- Set DVWA Security Level to "Low" under the DVWA Security settings to ensure XSS vulnerabilities are present for learning purposes.

➢ Step 1: Start XAMPP

- Open the **XAMPP Control Panel**.
- Start the **Apache** and **MySQL** services by clicking the "Start" buttons next to each.
- Ensure both services are running before proceeding to access DVWA.

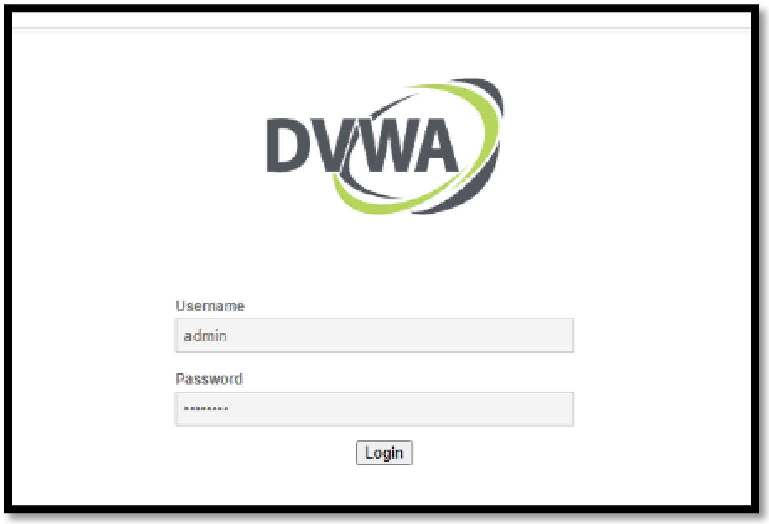

XAMPP Control Panel v3.3.0 [Compiled: Apr 6th 2021]

XAMPP Control Panel v3.3.0

Service	Module	PID(s)	Port(s)	Actions			
	Apache	576 5636	80, 443	Stop	Admin	Config	Logs
	MySQL	9308	3306	Stop	Admin	Config	Logs
	FileZilla	9384	21, 14147	Stop	Admin	Config	Logs
	Mercury	9524	25, 79, 105, 106, 110, 143, 2224	Stop	Admin	Config	Logs
	Tomcat			Start	Admin	Config	Logs

➢ **<u>Step 2: Launch DVWA</u>**

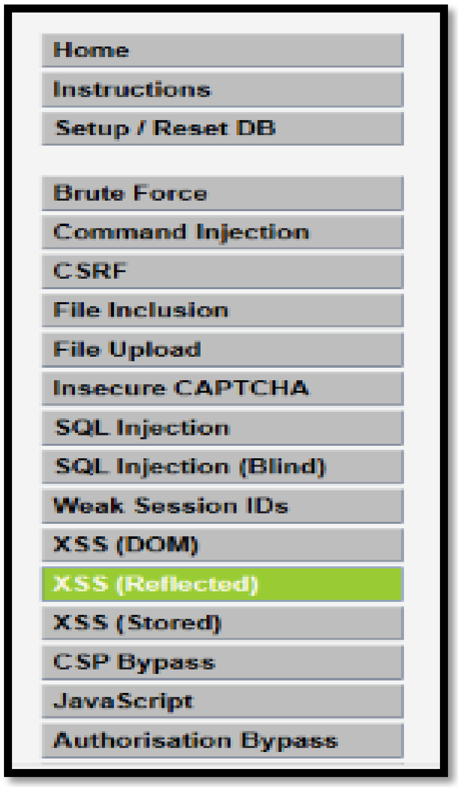
- Open DVWA in your web browser.
- Log in using your credentials (e.g., **admin / [your password]**).



✓ **Attack Execution:**

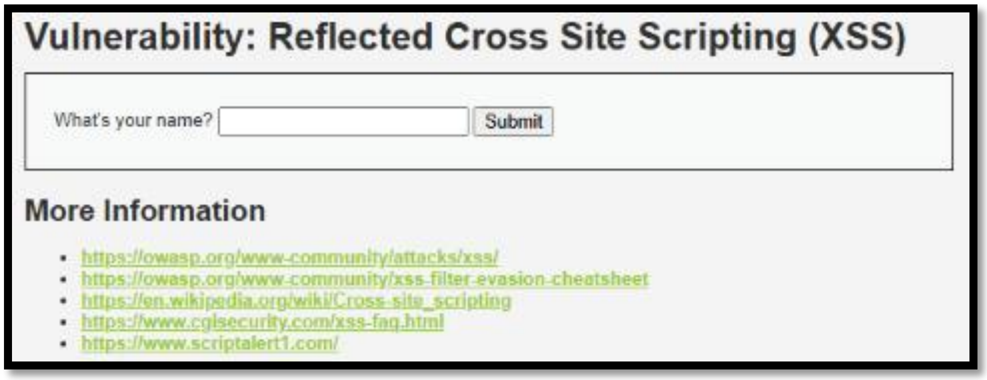
➢ **Step 3: Navigate to XSS (Reflected)**

- In the DVWA dashboard, go to the **“XSS (Reflected)”** section to begin testing the reflected XSS vulnerability.



➢ **Step 4: Inject XSS Payload**

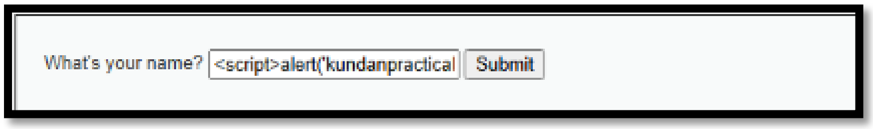
- In the input field labeled **"What's your name?"**, enter the following payload:
  <script>alert('Kundan Practical')</script>
- Submit the input to test for reflected XSS



Vulnerability: Reflected Cross Site Scripting (XSS)

More Information

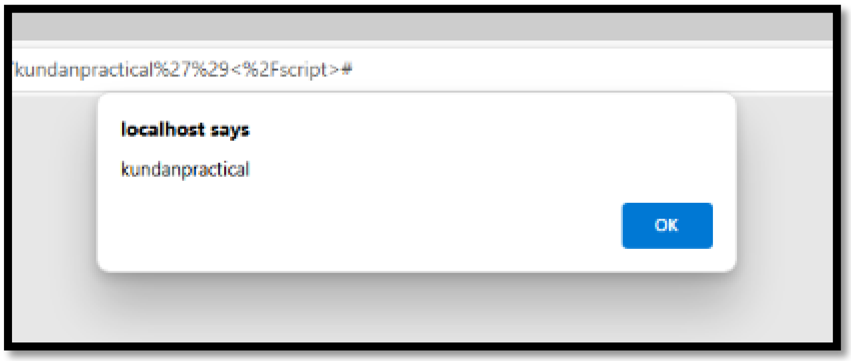
- https://owasp.org/www-community/attacks/xss/
- https://owasp.org/www-community/xss-filter-evasion-cheatsheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- https://www.cgisecurity.com/xss-faq.html
- https://www.scriptalert1.com/



```
<script>alert('kundanpractical
```

✓ Simulation Result:

- As expected, an alert box with the message "Kundanpractical" appeared in the browser.
- This confirmed that the web application reflects user input directly into the page without proper sanitization or encoding.



```
'kundanpractical%27%29<%2Fscript>#
```

```
localhost says
kundanpractical
```

✓ Conclusion:

- The successful execution of the JavaScript alert demonstrates a **reflected XSS vulnerability** in DVWA.
- This simulation shows how attackers can exploit unsanitized user inputs to run arbitrary scripts in a victim's browser, which can lead to session hijacking, credential theft, or redirection to malicious websites.

Observe and identify

Observation Phase:

After submitting the XSS payload (<script>alert('Kundanpractical')</script>) in the **DVWA's XSS (Reflected)** input form, the application returned a new page where the input was reflected directly into the HTML output without any sanitization. As a result, the JavaScript payload was executed immediately by the browser.

Identification Phase:

Based on the behavior and technical analysis, the vulnerability can be identified as a **Reflected XSS** attack due to the following reasons:

- The malicious script was not stored on the server — it was reflected directly from the user's request into the response page.
- The script execution occurred immediately upon submitting the input, without any need for interaction from another user.
- The vulnerability depends on user input being included in server responses without proper validation or encoding.

Security Implications Identified:

- **Exploitation Potential**: If an attacker crafts a URL containing the malicious XSS payload and tricks a user into clicking it, the script can execute in the victim's browser — potentially stealing session cookies or redirecting to malicious websites.
- **Session Hijacking**: If session cookies are not flagged as Http Only, they can be accessed by injected JavaScript.
- **Phishing or Defacement**: The attacker could manipulate the page content to appear legitimate while stealing data in the background.

Conclusion:

The clear sign of JavaScript execution — in the form of an unexpected alert — and the technical behavior observed in the browser confirm that the web application is vulnerable to **Reflected Cross-Site Scripting (XSS)**. This vulnerability demonstrates the danger of failing to sanitize user input before reflecting it in a web page.

Basic IR plan

Mitigation

Mitigation: Preventing XSS with Input Sanitization

What is the issue?

XSS happens when untrusted user input (like <script>alert('XSSed!')</script>) is injected into a web page and executed by the browser.
This allows attackers to run malicious scripts in the victim's browser.

What is input sanitization?

Input sanitization means converting special characters in user input into safe, displayable formats.
This ensures the browser displays the input as plain text rather than executing it as code.

How does it work?

Before outputting user input to the page:

-Convert<to&lt;
-Convert>to&gt;
-Convert"to&quot;
- Convert ' to &#39;

Example:-

User input:
<script>alert('Kundan Practical')</script>

After sanitization:
&lt;script&gt;alert('XSSed!')&lt;/script&gt;

- Browser displays this as text.
- Script does not execute.

Why this works

The browser no longer recognizes the input as a <script> tag or other HTML/JS code.
It breaks the XSS attack chain.

Simple policy

❖ **All user inputs must be validated and sanitized at the point of entry.**

- Input fields (text boxes, forms, URL parameters, API calls) must enforce strict data types and acceptable character sets.
- Inputs from untrusted sources (e.g., browser, external APIs) are considered malicious by default.

❖ **No user input should be rendered in HTML output without proper encoding.**

- Dynamic content must be escaped or encoded before being rendered in a browser context.
- JavaScript, HTML, CSS, and URL contexts must each use appropriate encoding techniques.

- **Client-side validation is permitted for user experience but must never replace server-side validation.**

- JavaScript-based checks can improve responsiveness but cannot be trusted for security enforcement.

- **Directory traversal attempts and file path manipulation must be blocked at the server level.**

- Web servers must be configured to deny requests that attempt to access parent directories (../) or protected files.

Root cause analysis

- ✓ **Summary of the Incident**

A **Reflected Cross-Site Scripting (XSS)** vulnerability was identified during a security test on a form input field within the DVWA application hosted on a local XAMPP server. The form accepted user input and reflected it back in the HTTP response without any sanitization or encoding, resulting in the execution of injected JavaScript code.

Root Cause

- **Direct Cause:**

- The application reflected raw user input directly into the HTML response. No input validation, sanitization, or output encoding was applied.
- The payload <script>alert('XSSed!')</script> was successfully injected and executed in the browser, confirming a **reflected XSS vulnerability**.

- **Underlying Causes:**

- **Lack of Input Validation:** No checks were in place to restrict or filter potentially malicious content submitted by the user.

- **Absence of Output Encoding:** Data submitted via input fields was not encoded before being embedded into the page's DOM.
- **No Content Security Policy (CSP):** The application lacked HTTP headers such as Content-Security-Policy, which could have mitigated the impact.
- **Development Misconfiguration:** The application was in "low security" mode (DVWA configuration), which is intentionally vulnerable for educational purposes. However, the same patterns are often unintentionally found in poorly coded real-world applications.

❖ Contributing Factors

- **Educational Environment (DVWA):** DVWA is intentionally designed to be insecure for learning, but the setup mimics common real-world mistakes.
- **Developer Awareness:** Developers may not have been properly trained on secure coding practices, particularly around input handling.
- **No Secure Coding Framework Used:** In the DVWA context, raw HTML and PHP were used without leveraging templating systems that escape output by default.
- **Lack of Security Testing:** If this were a real application, the vulnerability may have existed due to the absence of static/dynamic application security testing (SAST/DAST) in the development pipeline.

❖ Impact Assessment

- **Technical Impact:** Script execution demonstrated that arbitrary JavaScript can be run within the user's browser context.
- **Business Risk (in real-world scenario):**
    - Session hijacking through cookie theft.
    - Credential phishing or login form impersonation.
    - Website defacement or redirection to malicious sites.
    - Loss of customer trust and reputational damage.

❖ Remediation Actions Taken

- Applied proper **output encoding** using secure methods (e.g., htmlspecialchars() in PHP).
- Enforced **input validation** to block disallowed characters and restrict input to expected formats.
- Configured **security headers** (e.g., Content-Security-Policy, X-XSS-Protection).
- Reviewed and tested similar input fields across the application for similar vulnerabilities.
- Updated developer guidelines to include secure input/output handling policies.

➢ Conclusion

The XSS vulnerability stemmed from unsafe handling of user-supplied input, which was not uncommon in many real-world applications. This simulation with DVWA demonstrated how simple input fields can be exploited if basic security principles are not enforced. A combination of secure coding, automated testing, and policy enforcement can effectively eliminate such vulnerabilities from production systems.