Web Application Security Assessment Report

Target Application: "DVWA (Damn Vulnerable Web Application

Company: "Future Interns"  
Date: 02/05/2025  
Author: Adjei Jeffery Amoafo

**Acknowledgement**

I would like to express my sincere gratitude to Future Interns for the opportunity to undertake this web application penetration testing project as part of my internship. Special thanks to the security and mentorship team for their support, resources, and constructive feedback throughout the learning process. This project has significantly enhanced my practical understanding of web vulnerabilities and ethical hacking techniques.

**Abstract**

This report presents a practical assessment of common web application vulnerabilities using the *Damn Vulnerable Web Application (DVWA)* as the target system. The project focused on identifying and exploiting weaknesses such as SQL Injection, Cross-Site Scripting (XSS), and authentication flaws, followed by proper documentation of findings and recommendations. The aim was to simulate real-world attack scenarios to understand exploitation techniques, assess application behavior under attack, and recommend mitigations to enhance web security posture.

**Introduction**

In the evolving threat landscape, web applications are among the most frequently targeted assets. As part of my internship with Future Interns, I was tasked with conducting a hands-on penetration test on a deliberately insecure web platform — DVWA. The objective was to identify, exploit, and document vulnerabilities commonly found in web applications, including but not limited to SQL Injection, Reflected and Stored XSS, and Session Management issues. This project helped bridge the gap between theoretical cybersecurity knowledge and practical offensive security skills, reinforcing the importance of secure coding practices and robust defensive mechanisms.

Table 1 Tools Used

**The following tools and technologies were employed during this penetration testing project:**

| **Tool / Technology** | **Purpose** |
| --- | --- |
| DVWA (Damn Vulnerable Web App) | Target environment designed for web security testing |
| Docker | To containerize and run DVWA locally in a secure sandboxed setup |
| Burp Suite Community Edition | Intercepting, modifying, and replaying HTTP requests for manual testing |
| sqlmap | Automated SQL injection and database takeover tool |
| Web Browser (Chrome) | Manual interaction and XSS testing |
| cURL | Basic HTTP request testing via CLI |
| Notepad / Word | Report documentation |
| Excel | Vulnerability tracking and risk categorization |
| Wireshark *(optional)* | For traffic sniffing and debugging (if needed in future expansions) |

# **Finding: SQL Injection – Authenticated User Credential Disclosure**

**Vulnerability Type**  
SQL Injection (GET parameter id in vulnerable.php)

**Affected URL/Endpoint**  
http://127.0.0.1/dvwa/vulnerabilities/sqli/?id=1&Submit=Submit

**Parameter**  
id

**Impact**  
An attacker can extract sensitive data directly from the backend database, including usernames and password hashes, leading to full compromise of application accounts.

**Proof of Concept (PoC)**  
A payload such as:

?id=1' OR 1=1-- -

was submitted via the vulnerable endpoint. Using **sqlmap**, we automated data extraction:

sqlmap -r sqli\_request.txt --dbs

sqlmap -r sqli\_request.txt -D dvwa --tables

sqlmap -r sqli\_request.txt -D dvwa -T users –dump

Table 2 Extracted Data Sample

|  |  |  |
| --- | --- | --- |
| user | avatar | password |
| admin | /hackable/users/admin.jpg | 5f4dcc3b5aa765d61d8327deb882cf99 (password) |
| gordonb | /hackable/users/gordonb.jpg | e99a18c428cb38d5f260853678922e03 (abc123) |
| 1337 | /hackable/users/1337.jpg | 8d3533d75ae2c3966d7e0d4fcc69216b (charley) |

## **Remediation**

* Use parameterized queries (e.g., PDO, prepared statements).
* Employ proper input validation and escaping.
* Set WAF rules to detect SQLi payloads.
* Limit DB permissions.

# **Finding: Reflected Cross-Site Scripting (XSS) – Input Handling Vulnerability**

**Vulnerability Type**  
Reflected Cross-Site Scripting (XSS)

**Severity**  
Medium

**Tested URL**  
http://127.0.0.1/dvwa/vulnerabilities/xss/?name=<script>alert('XSS')</script>

**Description**  
The application fails to properly sanitize user-supplied input before reflecting it back in the HTTP response. When a specially crafted payload such as <script>alert('XSS')</script> is entered into the name parameter and submitted, the payload is executed by the browser, demonstrating the vulnerability.

This confirms the application is vulnerable to **Reflected XSS**, which could allow attackers to trick users into clicking malicious links that execute arbitrary JavaScript in their browsers.

## **Proof of Concept (PoC)**

1. Navigate to the vulnerable URL.
2. Enter the following input in the name field:  
   <script>alert('XSS')</script>
3. Submit the form.
4. A JavaScript alert box pops up with the message XSS

**Impact**  
Successful exploitation of this vulnerability can lead to:

* Cookie/session theft
* Credential harvesting
* Redirecting users to malicious sites
* Browser-based exploitation

**Affected Parameter**  
name

## **Recommendation**

* Sanitize and encode user inputs properly using frameworks or libraries that handle output encoding (e.g., OWASP Java Encoder, Microsoft AntiXSS).
* Use Content-Security-Policy (CSP) headers to reduce script execution risk.
* Implement server-side input validation and output escaping.
* Avoid reflecting user input without encoding in HTML response.

# **Stored XSS – Cross-Site Scripting Vulnerability**

**Vulnerability Description**

Stored Cross-Site Scripting (XSS) occurs when user input containing malicious scripts is stored by the application and reflected to users in HTTP responses without proper sanitization. This type of vulnerability is particularly dangerous because the malicious script is stored on the server and can affect multiple users, triggering the script every time they access the affected page.

**How We Tested for It**

1. **Injected Payload:** We injected typical XSS payloads such as <script>alert('XSS')</script> into user input fields (e.g., comment sections, profile fields, etc.).
2. **Submission:** After submitting the input, we observed if the input was reflected back in the HTTP response without proper sanitization.
3. **Execution of Script:** Upon visiting the page, we monitored if the browser executed the malicious JavaScript payload, which would confirm the vulnerability.

**Findings**

* **Vulnerability Identified:** The application failed to sanitize or encode user input before storing it, leading to the execution of injected JavaScript when the page was loaded.
* **Payload Tested:** <script>alert('XSS')</script>
* **Effect:** When the page was loaded, the browser executed the script, showing an alert box with the message "XSS," confirming the vulnerability.

**Impact**

* **Data Theft:** Attackers could steal sensitive data such as session cookies or user credentials stored in the browser.
* **Phishing Attacks:** Attackers could inject malicious scripts that redirect users to fake websites, leading to credential theft.
* **Defacement or Disruption:** Attackers could inject scripts that alter the appearance or content of the page, potentially misleading or confusing users.
* **Remote Code Execution:** In some cases, the attacker could inject scripts that interact with the backend, making the attack more severe.

**Recommendation**

* **Sanitize User Input:** Ensure proper sanitization of user input before storing it, filtering out unsafe characters or patterns (like <, >, script, etc.).
* **Output Encoding:** Use output encoding (e.g., HTML entity encoding) to prevent scripts from being executed when reflected back to users.
* **Implement Content Security Policy (CSP):** Apply a robust CSP to block the execution of untrusted scripts, reducing the potential impact of XSS vulnerabilities.
* **Use Web Application Firewalls (WAFs):** Deploy WAFs to detect and block malicious input, providing an additional layer of defense.

**Example Payloads for Testing**

* <script>alert('XSS')</script>
* <img src="x" onerror="alert('XSS')">
* <div onmouseover="alert('XSS')">Hover me!</div>
* <a href="javascript:alert('XSS')">Click me!</a>

**Tools Used for Testing**

* **Manual Testing**: We manually injected malicious JavaScript payloads into input fields (such as form fields or comment sections) and observed the results.
* **Browser Developer Tools**: Used the browser console to observe and analyze the execution of JavaScript payloads in real-time.

**Conclusion**

The application is vulnerable to **Stored XSS**, which could lead to a variety of attacks including session hijacking, phishing, and data theft. Immediate remediation should be implemented to sanitize user inputs, encode outputs, and apply a Content Security Policy (CSP) to prevent future attacks.

**Vulnerability: Broken Authentication – Session Fixation / Session Hijacking**

**Description**

The application fails to invalidate previously issued session IDs. By reusing an old session token (PHPSESSID), it was possible to gain unauthorized access to a user’s session, even after logout. This indicates improper session handling and opens up the application to session hijacking attacks.

**Proof of Concept (PoC)**

1. Log in to the DVWA app as a user.
2. Capture the session cookie (PHPSESSID) using Burp Suite.
3. Log out of the account to simulate session end.
4. Open Burp Repeater or a new browser tab.
5. Send a request with the *old* session ID set in the Cookie header:

makefile

CopyEdit

Cookie: PHPSESSID=6ka5fgj9i229jpnr7l3pvropq5

1. Observe that the server still recognizes the session as valid and grants access.

# **Affected Parameter**

**PHPSESSID** cookie

**Impact**

An attacker who gets access to a user's session ID (e.g., via XSS or sniffing on insecure connections) can reuse that session to log in without credentials — leading to full account takeover.

**Recommendation**

* Regenerate session IDs on each login.
* Invalidate sessions on logout and after a timeout period.
* Tie sessions to IP/user agent (optional but helpful).
* Implement HttpOnly, Secure, and SameSite flags for session cookies.

**Conclusion**

The **Broken Authentication** vulnerability due to **Session Fixation** and **Session Hijacking** in the DVWA app highlights poor session management practices. The inability to invalidate session IDs after logout allows attackers to reuse old session tokens, enabling unauthorized access and potential account takeover. This indicates a critical flaw in session handling, which can be exploited through XSS, sniffing, or other methods of stealing session IDs. To mitigate such risks, the application must ensure proper session management, including regenerating session IDs on login, invalidating sessions after logout or timeout, and securing cookies with appropriate flags.