# DVWA Security Level Comparison Project

**Objective**

The objective of this project is to systematically evaluate how different security hardening techniques affect real vulnerabilities by using the **DVWA** learning environment. For two chosen vulnerabilities i.e.,**SQL Injection and Cross site scripting**, the project will: demonstrate working exploit payloads at the Low security setting; replay the same payloads at Medium, High, and Impossible settings and record behavioral differences; analyze DVWA’s source or responses to identify the exact defenses introduced at each level (e.g., input sanitization, output encoding, parameterized queries); and produce a concise comparison report with PoC artifacts and developer-facing remediation guidance.

## Introduction

## DVWA (Damn Vulnerable Web Application) provides configurable security levels intended for training. This project uses DVWA to compare defensive mechanisms across its security levels for two representative vulnerabilities: SQL Injection and Reflected XSS. The methodology is: craft and document proof-of-concept payloads that exploit the vulnerabilities on Low; execute identical payloads on Medium, High, and Impossible while capturing raw HTTP requests/responses and screenshots; inspect DVWA source code and response transformations to identify what protection was added at each level; and summarize effectiveness, bypassability, and recommended real-world mitigations. This comparison helps bridge lab learning to practical secure-coding decisions.

## Scope

**Target:**

* DVWA

## Steps performed

1. Reconnaissance: Found open ports, discovered hosts, services and web paths through nmap.

2. Scanning: Ran automated checks using Nikto, dirb .

3. Manual testing: Used Burp Suite to confirm issues and build PoCs (proofs of concept).

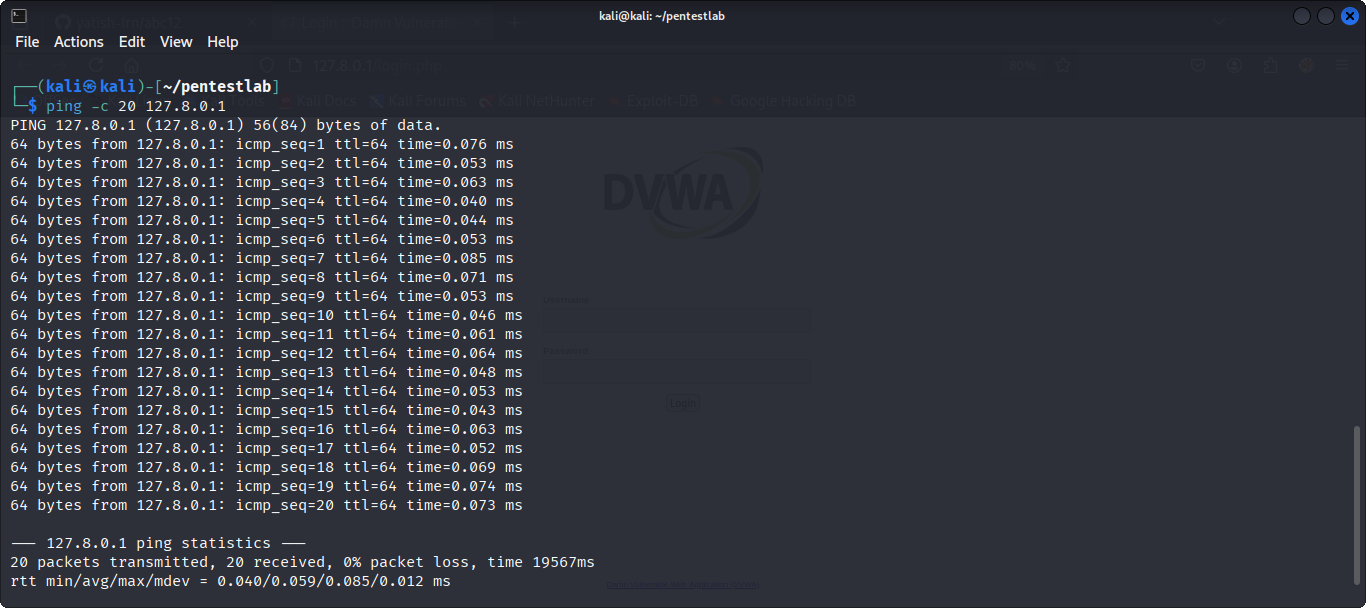
4.. Reporting: Captured screenshots, saved raw scan outputs, and wrote fixes.

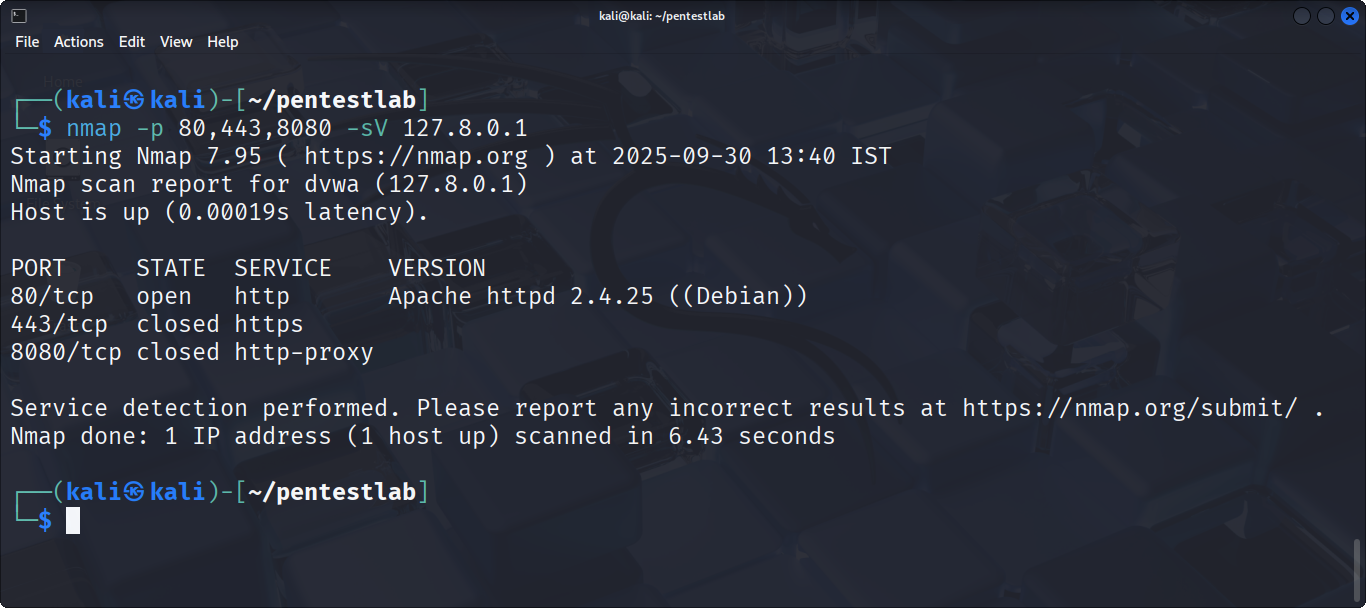
**Tools used:** Nmap, nikto, dirb, Burp Suite.

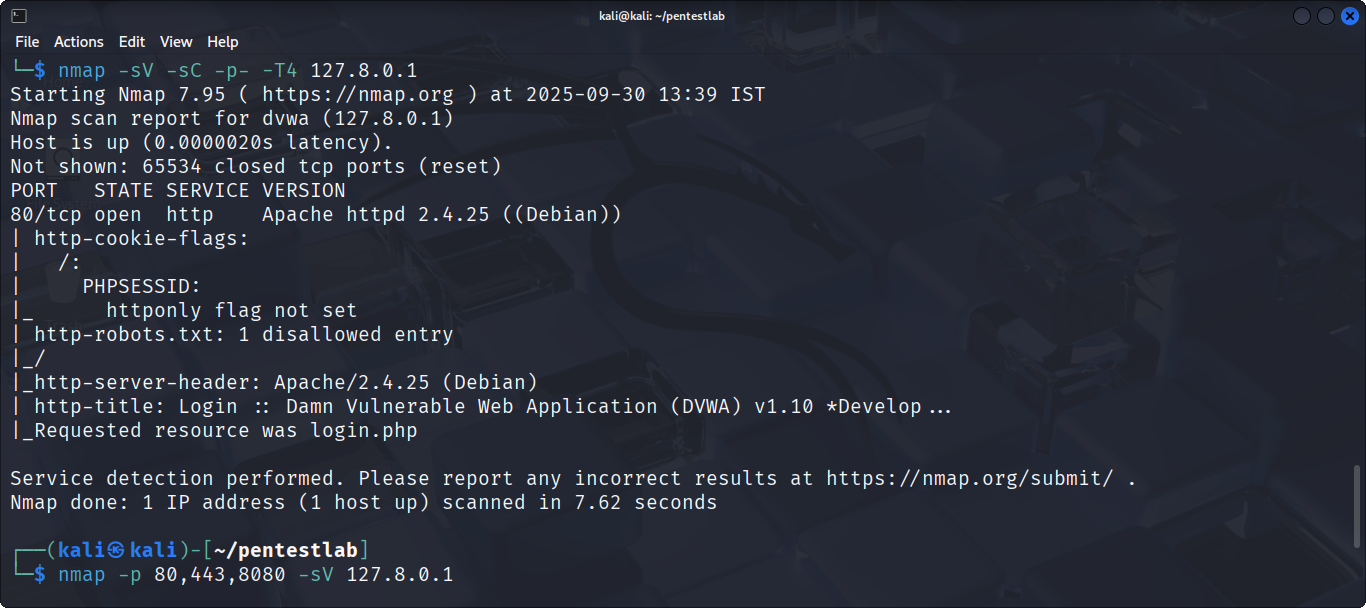
# Findings on DVWA

1. **Reconnaissance**

**i. Basic ping**

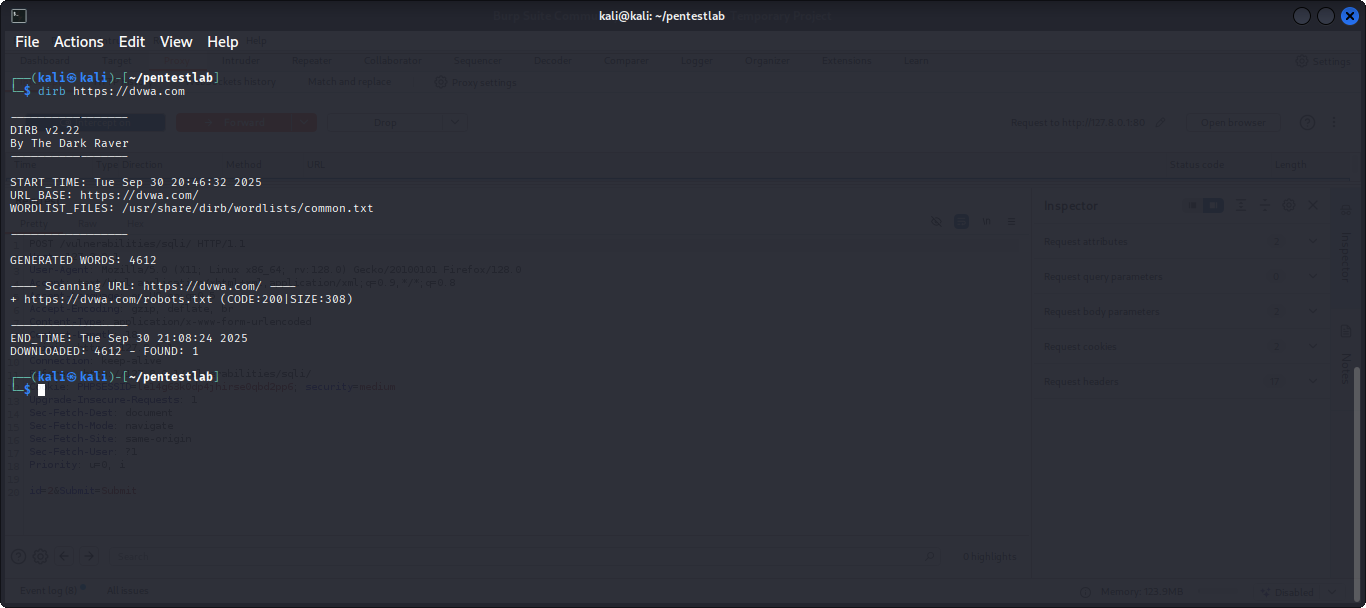
**ii.Port and Service scan**

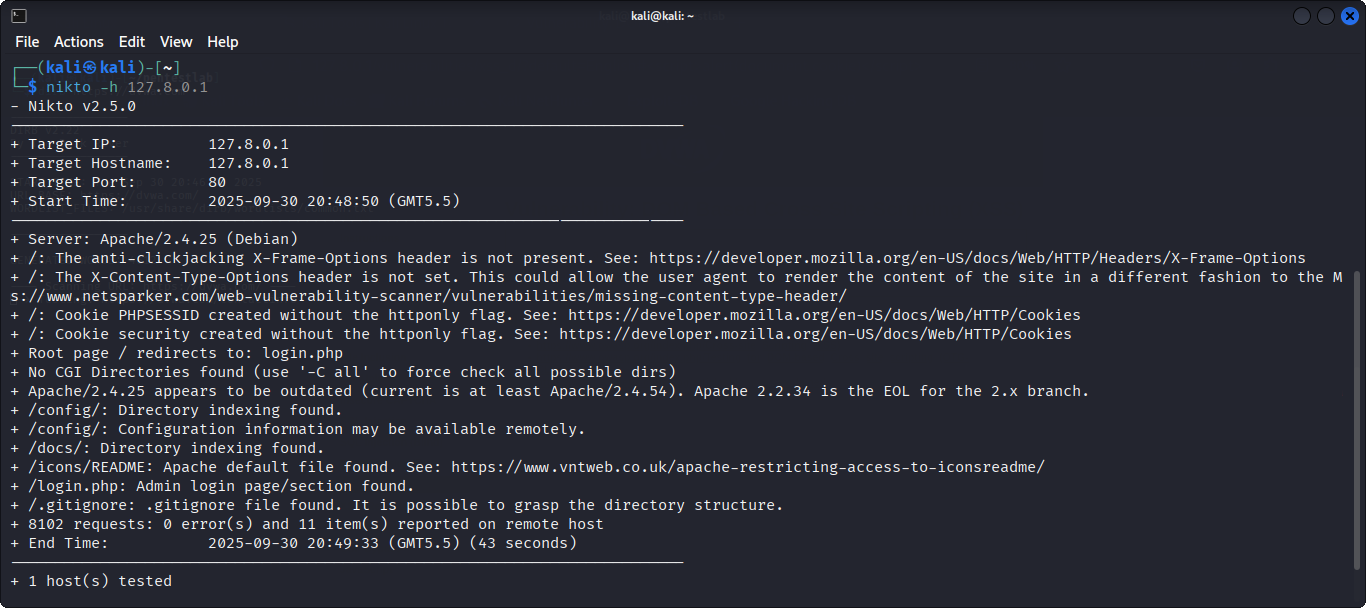
**iii.Web Focused scan**

****Captured: Open ports, service versions, server headers.

1. **Web Enumeration**

* **Directory Bruteforcing**

****

**3.Automated Scanning**

**4.Manual Testing**

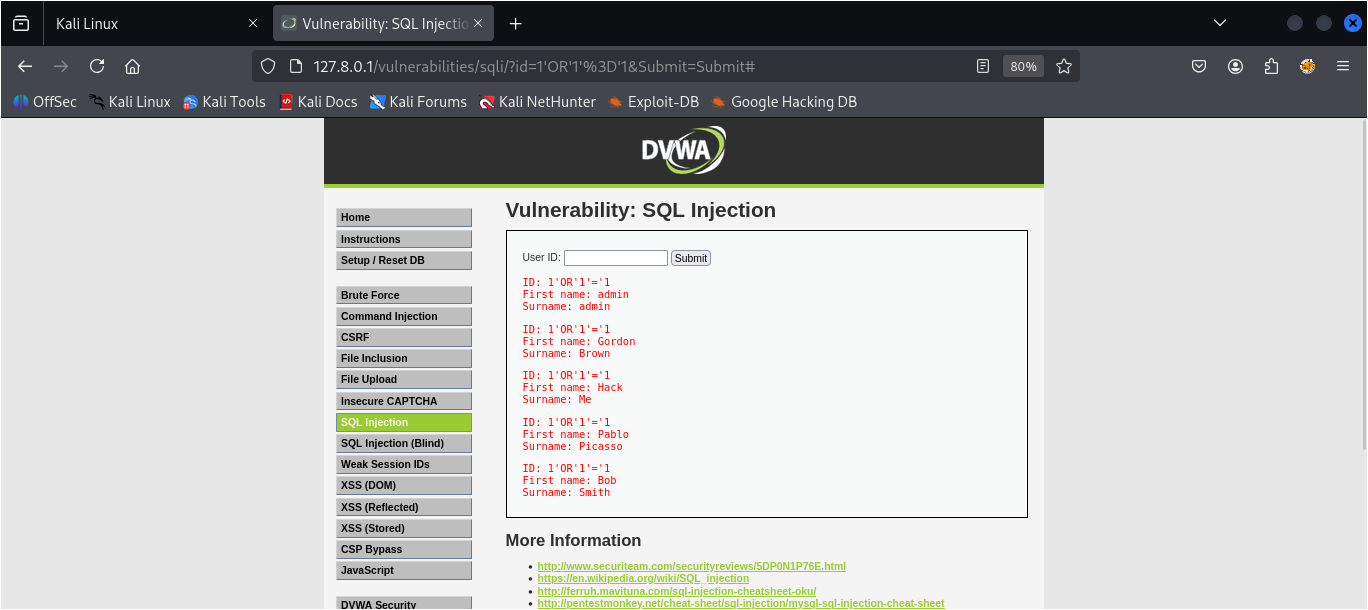
**A. SQL injection**

SQL Injection in DVWA login — can allow authentication bypass and data exposure.

**Low Security**

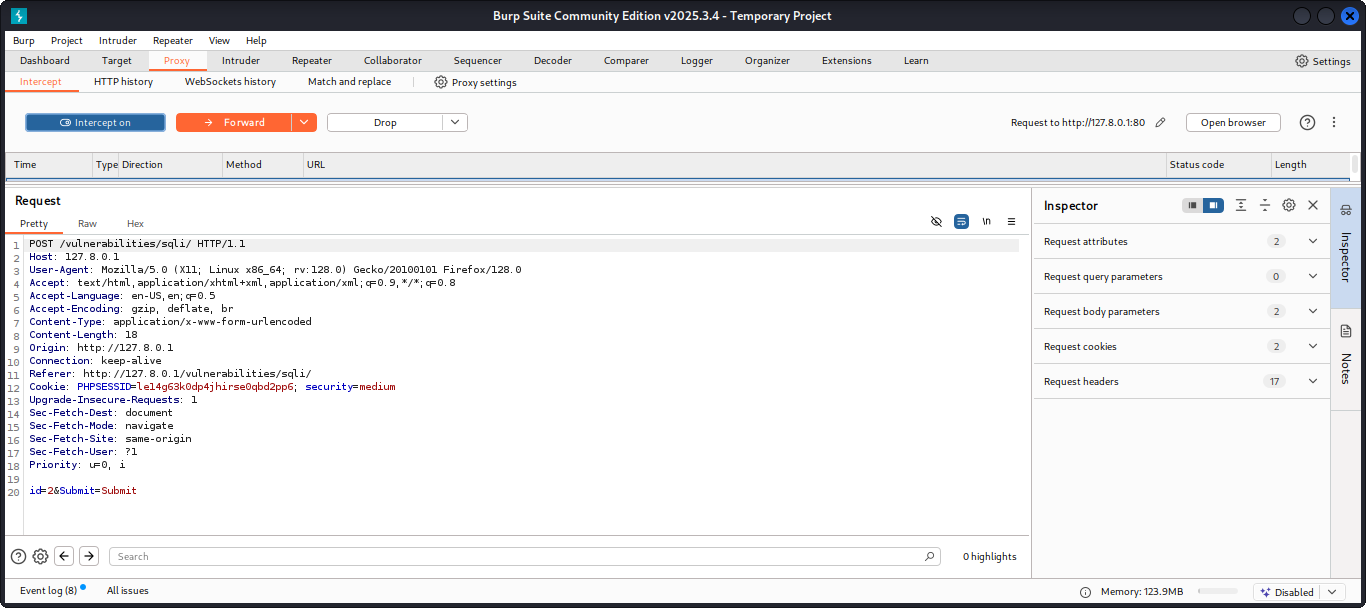
* **Go to DVWA Security-->set to LOW-->SQL Injection page-->enter payload**

**1’OR’1’=’1**

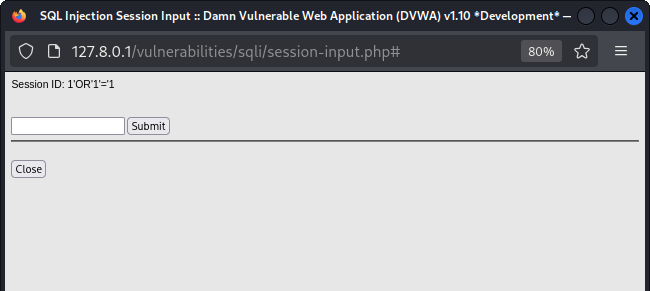
****

* **Result:All user data will be displayed**

****Medium Security****

**Use burpsuite to intercept the traffic-->enable foxyproxy-->in burpsuite go to proxy and set intercept to on-->select ID in DVWA**

**High Security**

**DVWA does not allow to perform SQL injection directly.**

* **Result:No user data can be seen-->DVWA prevented the attack.**

Why it matters: An attacker can bypass authentication and access restricted pages or data as an authenticated user, this can lead to data theft, privilege escalation, or further compromise.

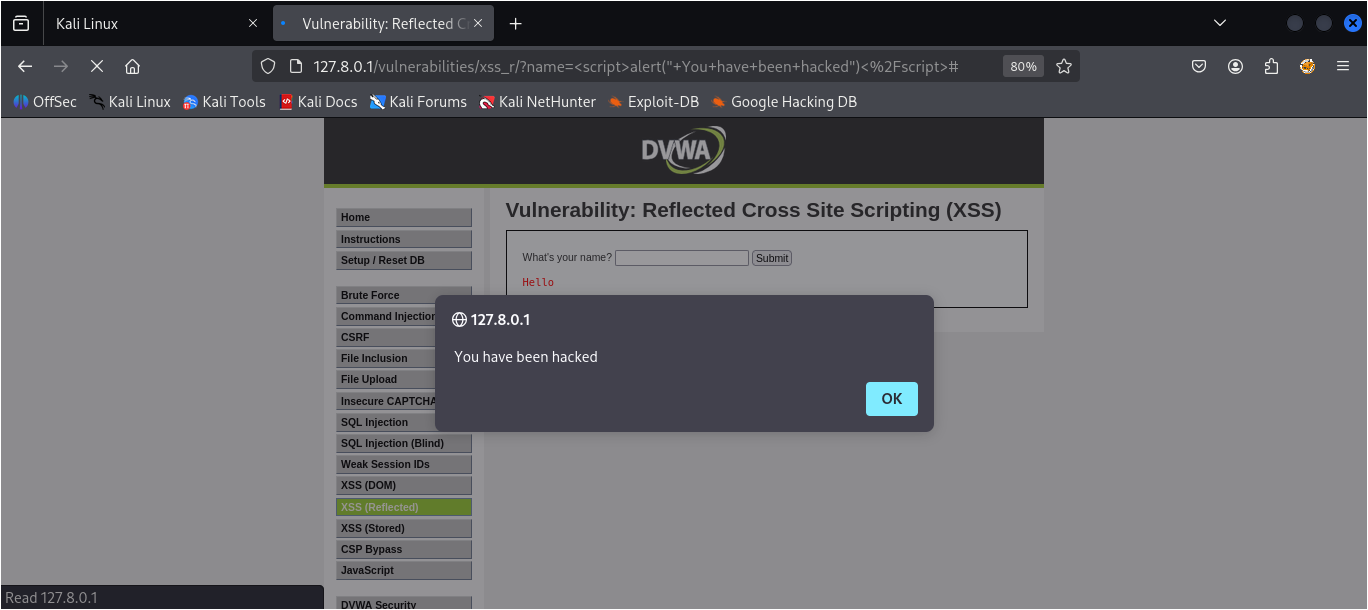
Cause : User input is directly concatenated into an SQL query without parameterization or sanitization.

**ii.Remediation**

* Use parameterized queries / prepared statements Validate & sanitize inputs server-side.
* Block suspicious payloads with a WAF rule and enforce strong authentication.
* After fix, try the same payload — it should not log you in and should be handled as data, not code.

1. **Cross Site Scripting**

Reflected XSS and stored XSS scenarios — could let an attacker run scripts in users’ browsers.

**Low Security**

* **Go to DVWA Security-->set to LOW-->go to XSS(Reflected )page-->enter the script or payload:**

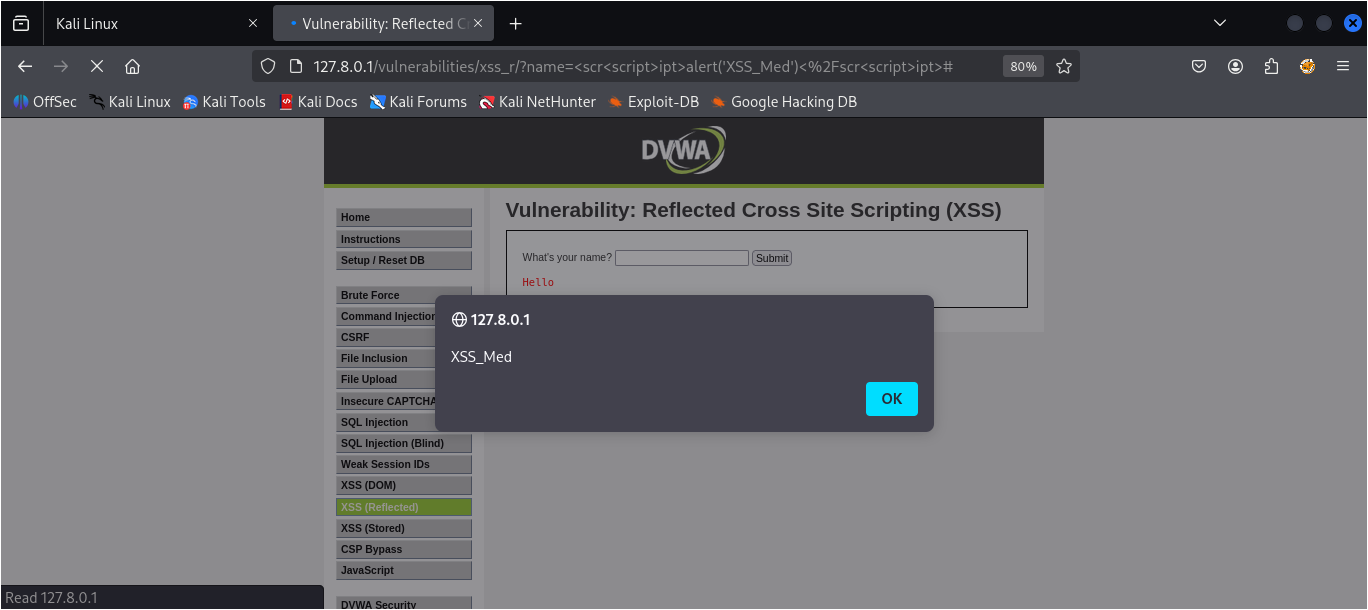
**<script>alert(“You have been hacked”)</script>**

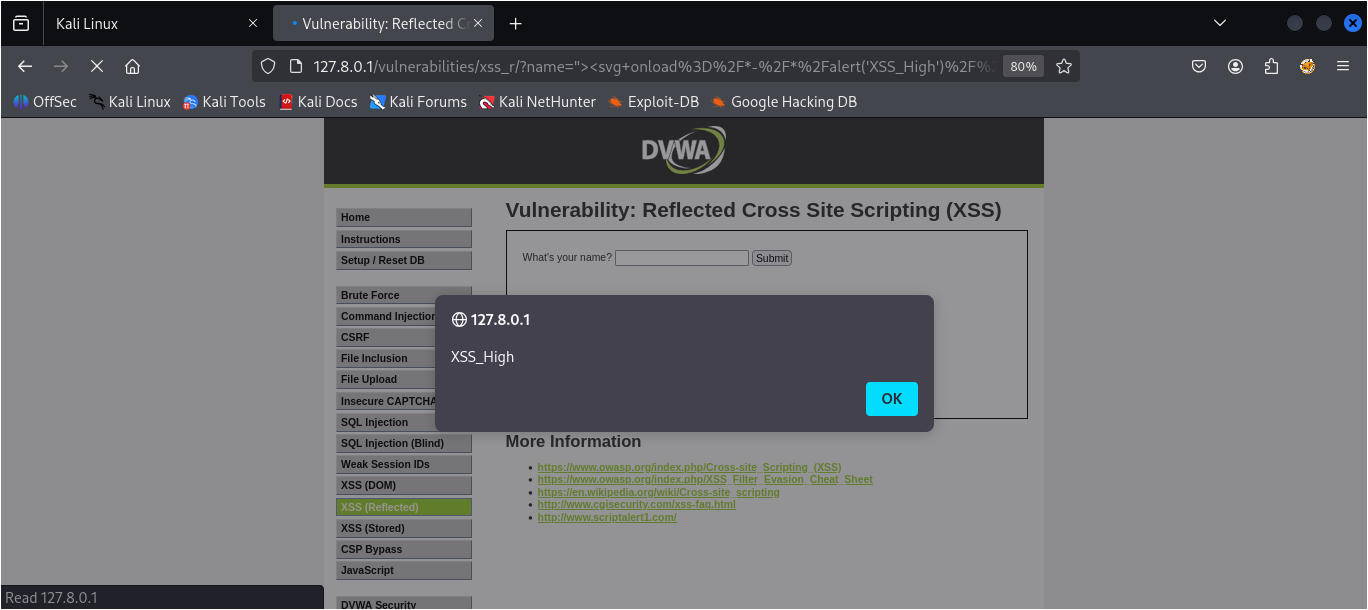
* **Result:The alert “You have been hacked” is displayed in the browser.**

**Medium Secuity**

* **Go to DVWA Security-->set to MEDIUM-->go to XSS(Reflected )page-->enter the script or payload:**

**<scr<script>ipt>alert(“XSS\_Med”)</scr<script>ipt>**

* **Result:The alert “XSS\_Med” is displayed in the browser.**

**High Security**

* **Go to DVWA Security-->set to HIGH-->go to XSS(Reflected )page-->enter the script or payload:**
* **Result:The alert “You have been hacked” is displayed in the browser.**

“><svg onload=/\*-/\*/alert(‘XSS\_High)//

**Observation**

### A. SQL Injection

**Low**

Plain payloads such as ' OR '1'='1 and admin' -- succeed. Manual auth bypass and data enumeration are possible.

**Medium**

Input frequently subject to basic escaping or simple filtering. Some payloads blocked.Error messages may be sanitized but not all injection points are eliminated.

**High**

Stronger input normalization and filtering applied. Many attack strings are removed or transformed so SQL syntax is no longer interpreted.

Low → no protection (vulnerable).

Medium → addslashes

High → broader sanitization or stricter input validation.

### B. Cross-Site Scripting (reflected and stored)

**Low**

<script>alert(1)</script> execute easily in reflected.

**Medium**

Basic HTML-encoding or tag-stripping applied. Raw <script> tags usually rendered as escaped characters , preventing execution..

**High**

More consistent sanitization: tags removed, dangerous attributes stripped, and input normalized. Event handlers are often removed or neutralized.

Low → no encoding/sanitization.

Medium → output escaping (htmlspecialchars()), naive strip\_tags().

High → broader sanitizer/whitelist and attribute stripping.

**Conclusion**

This DVWA security-level comparison shows how incremental server-side protections progressively reduce exploitability: Medium blocks the simplest attacks, High significantly hardens the application, and Impossible implements robust techniques that prevent exploitation in normal flows. However, the experiment also demonstrates that superficial or inconsistent defenses are brittle — attackers can often find alternate injection paths or exploit unsafe client-side behaviors. The most reliable security comes from applying correct, context-aware controls (parameterized queries for database access, contextual output encoding for HTML/JS/CSS/attribute sinks), enforcing defense-in-depth (CSP, secure headers, least privilege), and embedding security tests in the development lifecycle to detect regressions.