Project 1: DVWA Security Level Comparative Analysis

Objective: The primary goal of this project is to analyze and compare the behavior of the **Damn Vulnerable Web Application (DVWA)** across its four security tiers: Low, Medium, High, and Impossible.
Methodology: Our research focuses on two specific vulnerabilities.

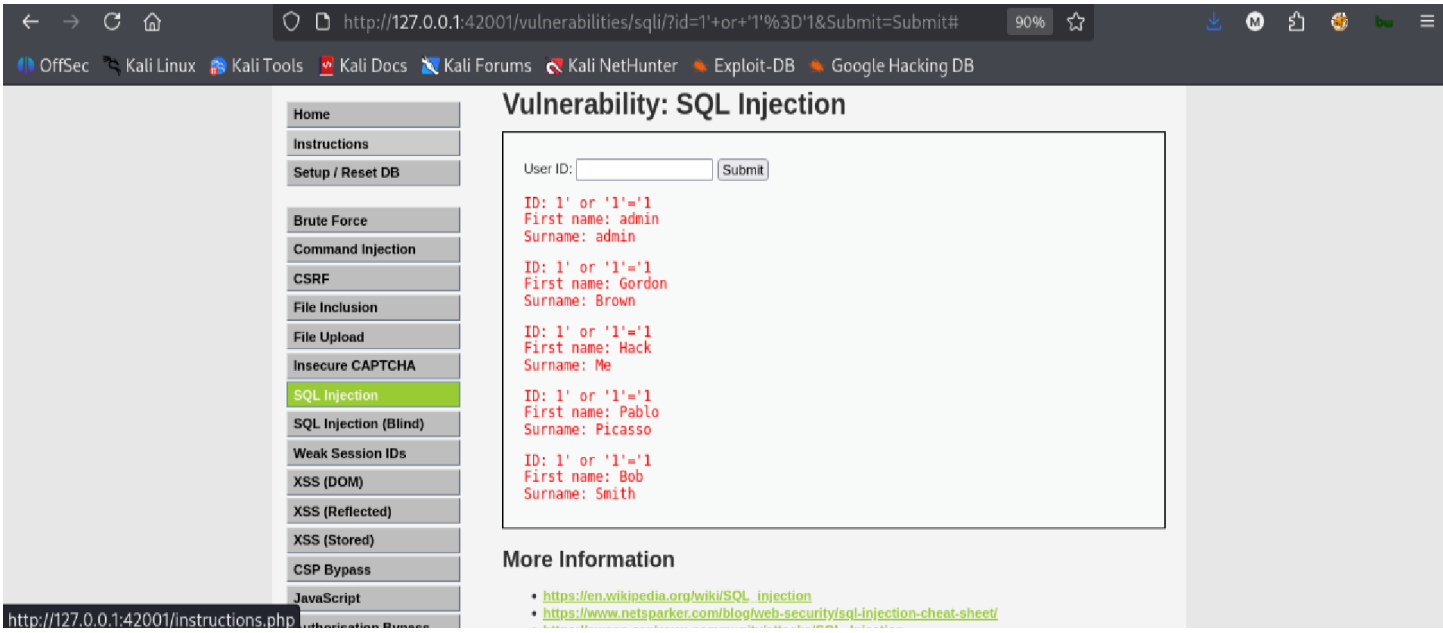
1. SQL Injection (SQLi)

Definition: This vulnerability occurs when user-supplied data (such as login credentials or URL parameters) is directly concatenated into an SQL query without proper sanitization. This flaw allows an attacker to inject and execute malicious SQL commands within the backend database.

Experimental Execution

- **Target Level:** Low Security
- **Payload Used:** 1' OR '1'= '1-- -
- **Observation:** At this level, the application lacks input validation, allowing the payload to bypass authentication or extract unauthorized data by making the SQL statement always evaluate to "true."

  "User ID" field: 1' or '1'='1.
- **The Logic:** In SQL, the condition '1'='1' is always true.
- **The Result:** By injecting this string, we alter the backend query logic. Instead of searching for a single specific ID, the database sees a command that effectively says, "Show me the record where the ID is 1 OR where 1 equals 1".



The Outcome

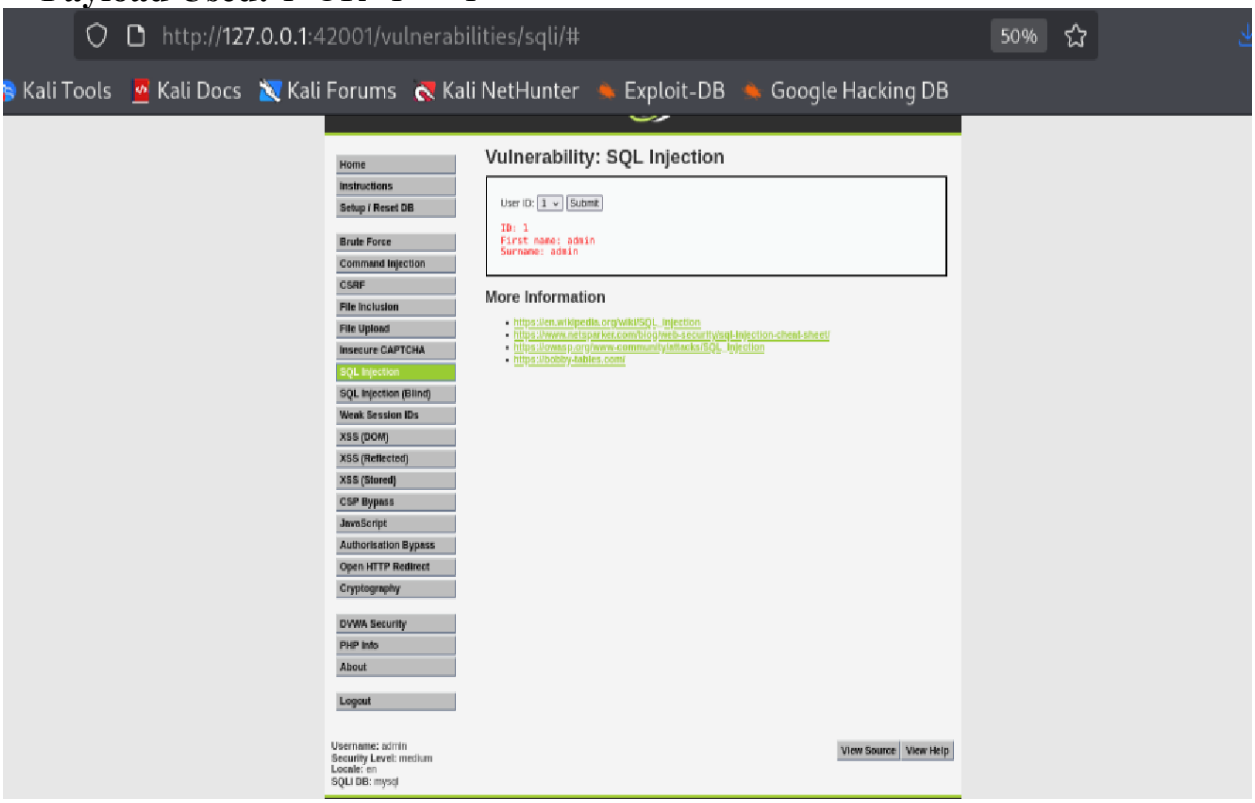
Because the condition is always true, the database bypasses the standard filtering and returns **every user record** in the table rather than just one. The red text in the results area confirms this by displaying data for multiple users:

- **Admin**
- **Gordon Brown**
- **Hack Me**
- **Pablo Picasso**
- **Bob Smith**

  This screenshot demonstrates a classic **authentication bypass/data leakage** scenario. The application directly concatenates the input into the query, allowing the attacker to dump the entire user database.

2. Medium Security

- **Target Level:** Medium Security
- **Payload Used:** 1' OR '1'= '1-- -

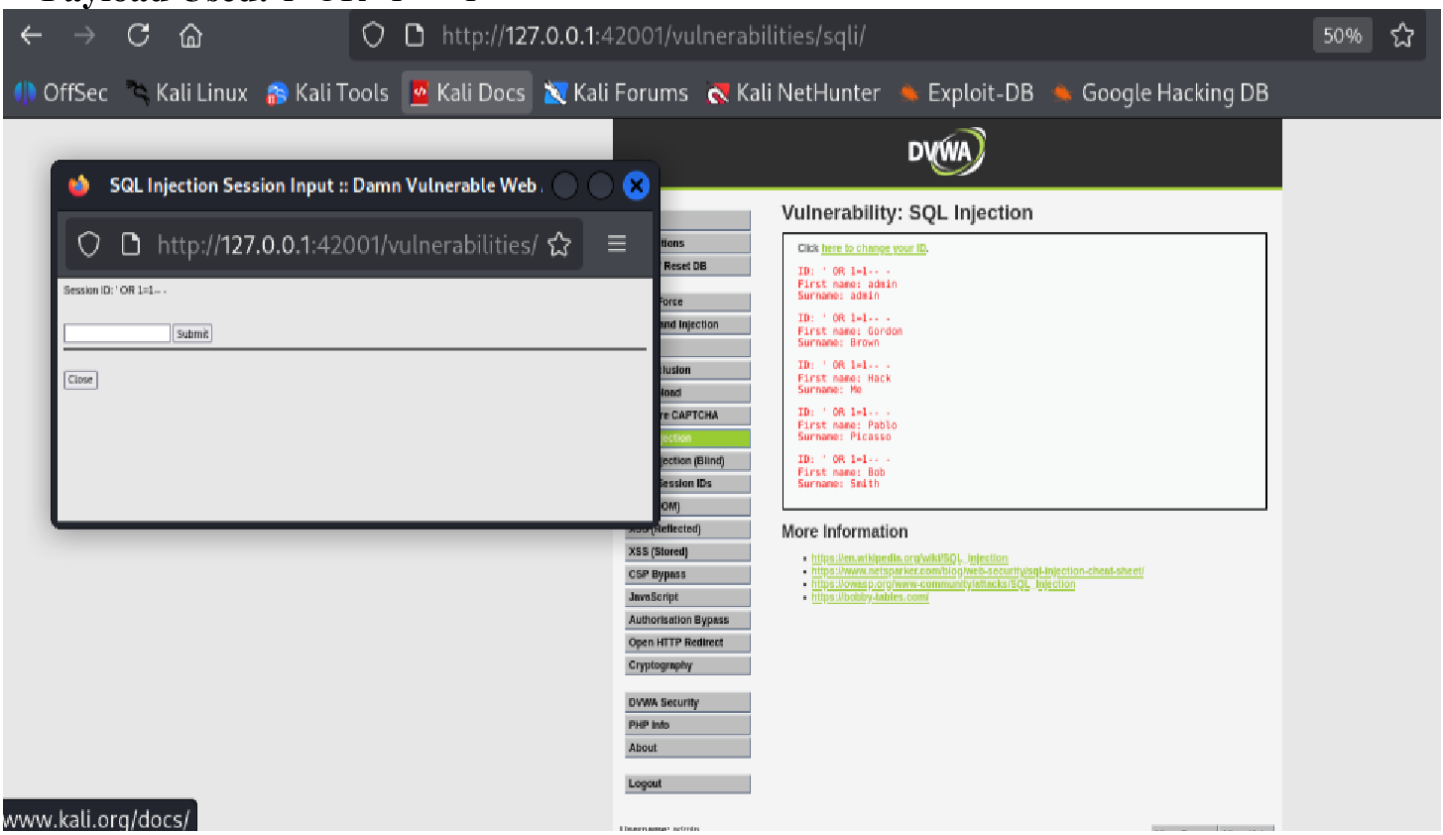


The Outcome

- The interface changes to include a dropdown selection menu rather than a free-text field.
- **Defense Mechanism:** This is a client-side restriction intended to limit users to specific, valid IDs.
- **Result:** By selecting "1" from the dropdown, the application only returns the single record for the "admin" user.
- **Weakness:** While it prevents simple text-entry injection, the underlying PHP code often uses basic escaping (like mysqli_real_escape_string) which can still be bypassed through request interception (e.g., Burp Suite) if variables are not properly typed

3. High Security

- **Target Level:** High Security
- **Payload Used:** 1' OR '1'= '1-- -



The Outcome

In this tier, the input mechanism is moved to a separate popup window or a different page to decouple the input from the results page.

- **Defense Mechanism:** This is designed to break automated exploitation tools and prevent simple URL manipulation.
- **The Exploit:** Despite the separate window, the user has successfully injected the payload ' OR 1=1-- -.
- **Result:** The injection is still successful, as evidenced by the main window displaying the full list of database users again.
- **Analysis:** This level demonstrates that "security by obscurity" (changing the UI flow) is ineffective if the backend query still directly concatenates user session input into the SQL statement.

Comparison Summary Table

Security Level	Input Method	Defense Type	Outcome
Low	Free Text Box	None	Full Data Leak
Medium	Dropdown Menu	Client-side restriction	Restricted to single record
High	Separate Popup Window	Indirect input flow	Security by Obscurity

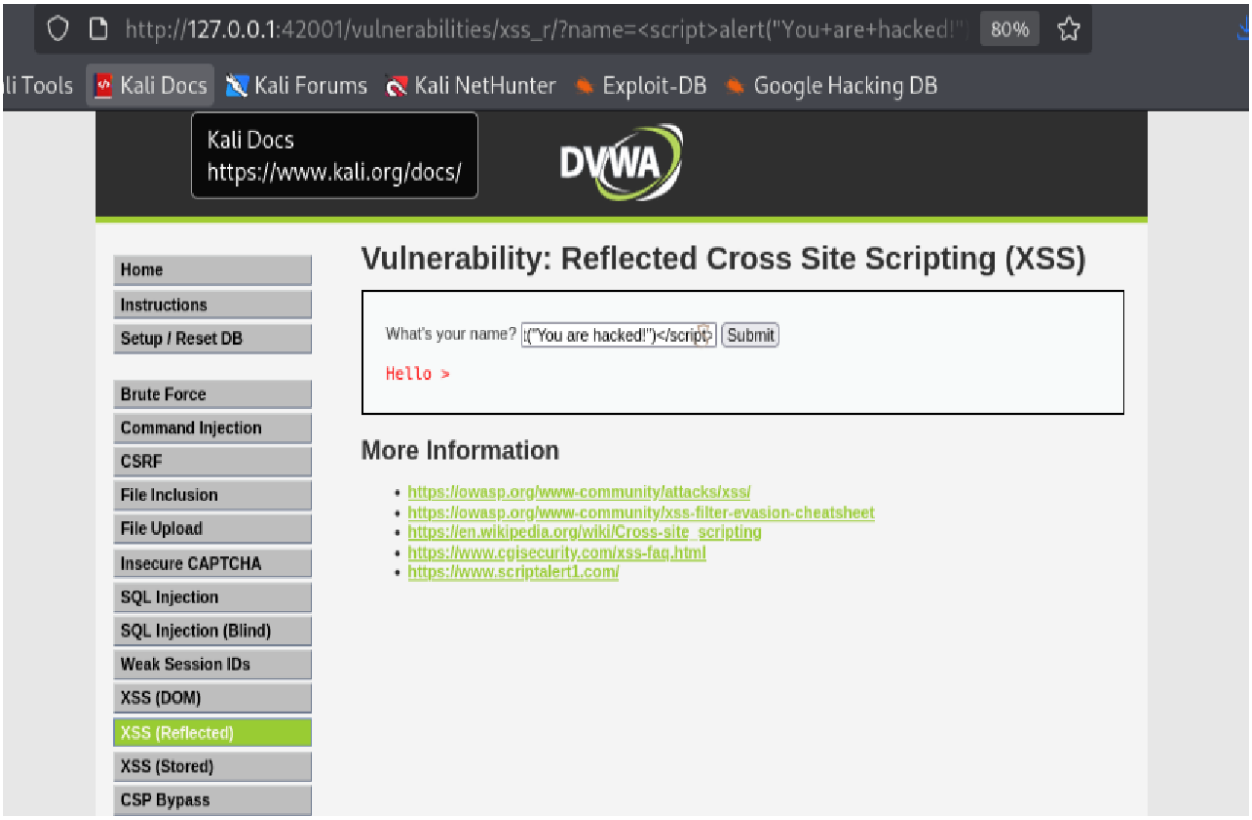
Vulnerability 2: Reflected Cross-Site Scripting (XSS)

1. Low Security Level

At this level, the application is highly vulnerable because it takes user input and reflects it back onto the page without any validation or encoding.

- **Exploitation:** When a JavaScript payload is entered into the name field: <script>alert("You are hacked!")</script>.
- **Mechanism:** Instead of treating the input as plain text (a name), the web browser interprets the <script> tags as executable code..
- **Result:** The page displays "Hello" followed by a blank space where the script was executed, proving that the malicious code was successfully injected into the Document Object Model (DOM).

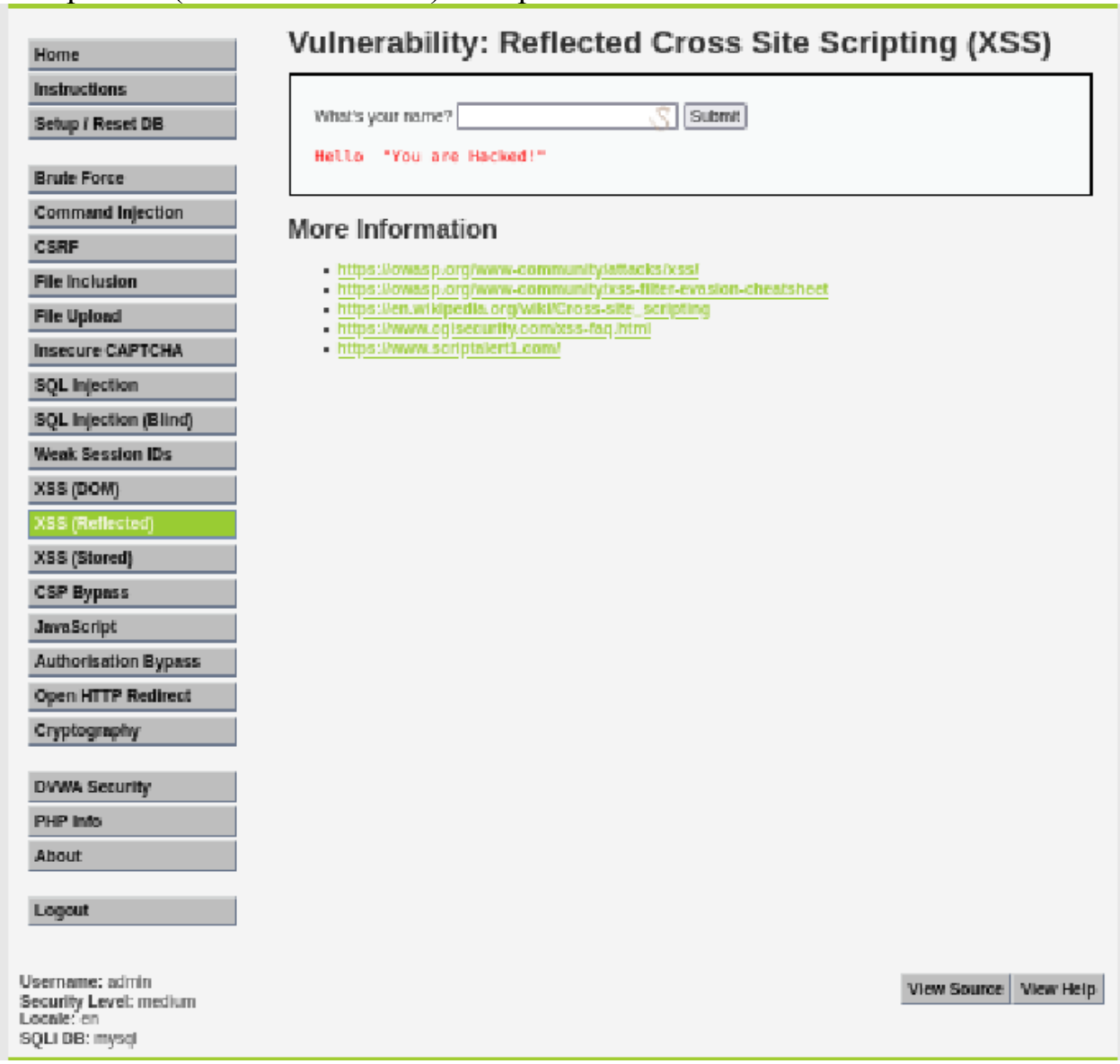
Security level:**low**
When the following payload is entered
<script>alert(“You are hacked!”)</script>



2. Medium Security: Basic Blacklisting

- **Mechanism**: The application attempts to strip out the <script> tag using basic string replacement functions.
- **Exploitation**: The screenshot shows the output Hello "You are Hacked!" in red text. The tags have been removed, preventing the script from running.
- **Result**: While it stops the basic payload, it can often be bypassed using different tags (like <img>) or mixed-case tags.

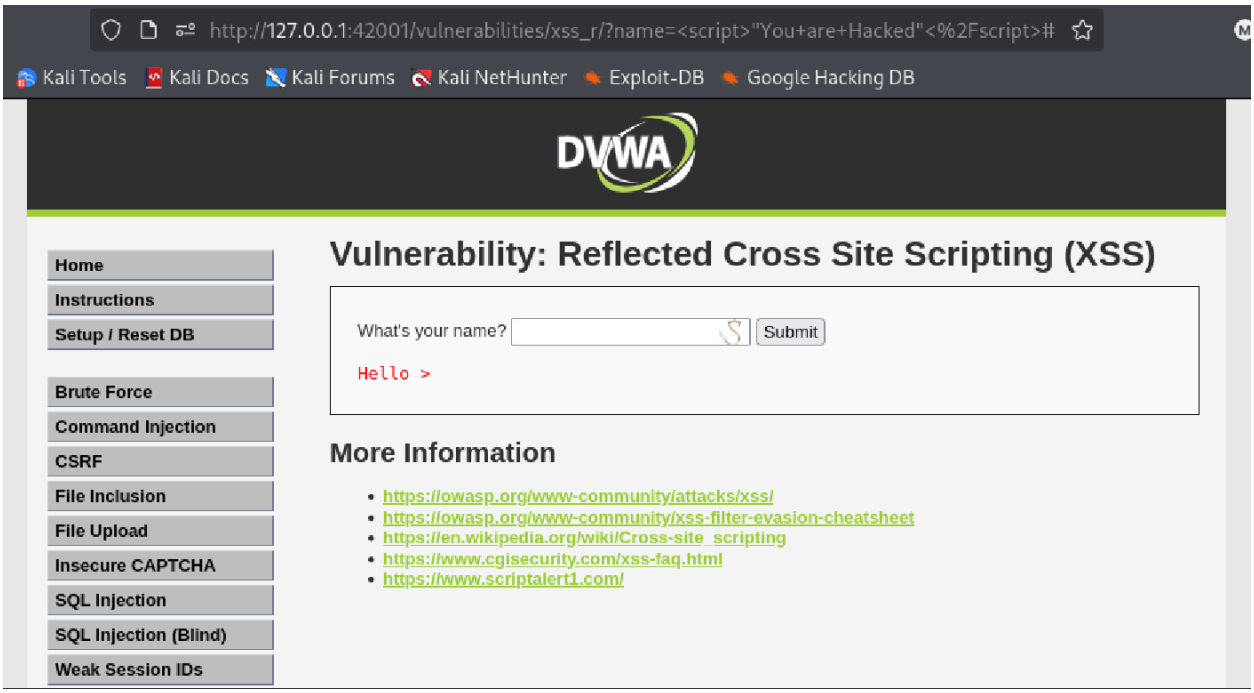
Security level:**Medium**
When the following payload is entered
<script>alert(“You are hacked!”)</script>



3. High Security: Advanced Filtering

- **Mechanism**: The application uses more complex regular expressions to remove anything that looks like a script tag.
- **Observation**: The output reflects Hello >, showing that the filter successfully stripped the majority of the payload.
- **Result**: Most common XSS bypasses are neutralized, making exploitation significantly more difficult at this level.

Security level:**Hard**
When the following payload is entered
<script>alert(“You are hacked!”)</script>



Summary

1. Low Security Level

At this level, there are **no security measures** in place. User input is used raw in the underlying code, making the application fully vulnerable and easy to exploit.

- **SQL Injection (SQLi):** Raw user input is directly concatenated into the SQL query, making it susceptible to classic injection payloads like ' OR '1'='1.
- **Cross-Site Scripting (XSS):** Any script input (e.g., <script>alert('XSS')</script>) is embedded in the page without any filtering or encoding, causing it to execute in the user's browser.

2. Medium Security Level

The Medium level features rudimentary, **insufficient security attempts** that are often bypassable. It serves to illustrate bad security practices.

- **SQLi:** Simple input sanitization functions (e.g., addslashes() or mysql_real_escape_string()) might be used, but the underlying query structure may still be flawed (e.g., missing quotes around parameters), allowing skilled attackers to bypass the filters using different techniques like numeric injection or manipulating hidden form fields.
- **XSS:** Basic string replacement is used to filter out common patterns like <script> tags. Attackers can bypass this by using variations in input case, encoding, or alternative tags like <img> to execute JavaScript.

3. High Security Level

This level employs **stronger, but still imperfect, security controls** that significantly reduce exploitability, often simulating real-world scenarios or CTF challenges.

- **SQLi:** The application might use session variables or different input methods to obscure the vulnerability, making direct URL injection harder, but the core issue of improper sanitization might still exist in the backend code.
- **File Inclusion:** Most common XSS bypasses are neutralized, making exploitation significantly more difficult at this level but potential vulnerabilities might still exist through clever techniques or misconfigurations.