Vulnerability Assessment Project DVWA Command Injection, Reflected XSS, and File Upload.

Cybersecurity and Secure Programming Module

Divya Guruvaiah Naidu

G00473080

Atlantic Technological University

January 14, 2026

Contents

1 Environment and Scope

1.1 Target Application

Damn Vulnerable Web Application (DVWA) is an intentionally insecure PHP web application intended for learning and controlled security testing. The analysis in this report focuses on the following DVWA modules:

- **Command Injection** (`/vulnerabilities/exec/`)
- **Reflected XSS** (`/vulnerabilities/xss_r/`)
- **File Upload** (`/vulnerabilities/upload/`)

1.2 Deployment

DVWA was deployed using Docker:

```
docker run --rm -it -p 80:80 vulnerables/web-dvwa
```

The application was accessed at:

```
http://localhost/
```

1.3 Security Level and Testing Approach

Testing was performed primarily at DVWA **Security Level: Low** to demonstrate vulnerabilities and validate the exploitability. Remediations were applied by updating the corresponding source files inside the running container and then verifying that the PoCs no longer succeeded.

1.4 Tools

- Web browser for manual testing and evidence capture
- VS Code for code review and edits
- **OWASP ZAP** (Manual Explore) as an intercepting proxy to capture and inspect HTTP requests/responses for reflected XSS testing

2 Threat Model and Risk Summary

Table 1 summarises the vulnerabilities assessed and their impact.

Table 1: Risk summary of assessed vulnerabilities.

Vulnerability	**Category (OWASP)**	**Impact**
Command Injection	Injection	OS command execution
Reflected XSS	Injection	Script execution in victim browser
Insecure File Upload	Software & Data Integrity / Upload Handling	Remote code execution

3 Vulnerability 1: Command Injection

3.1 Location and Root Cause

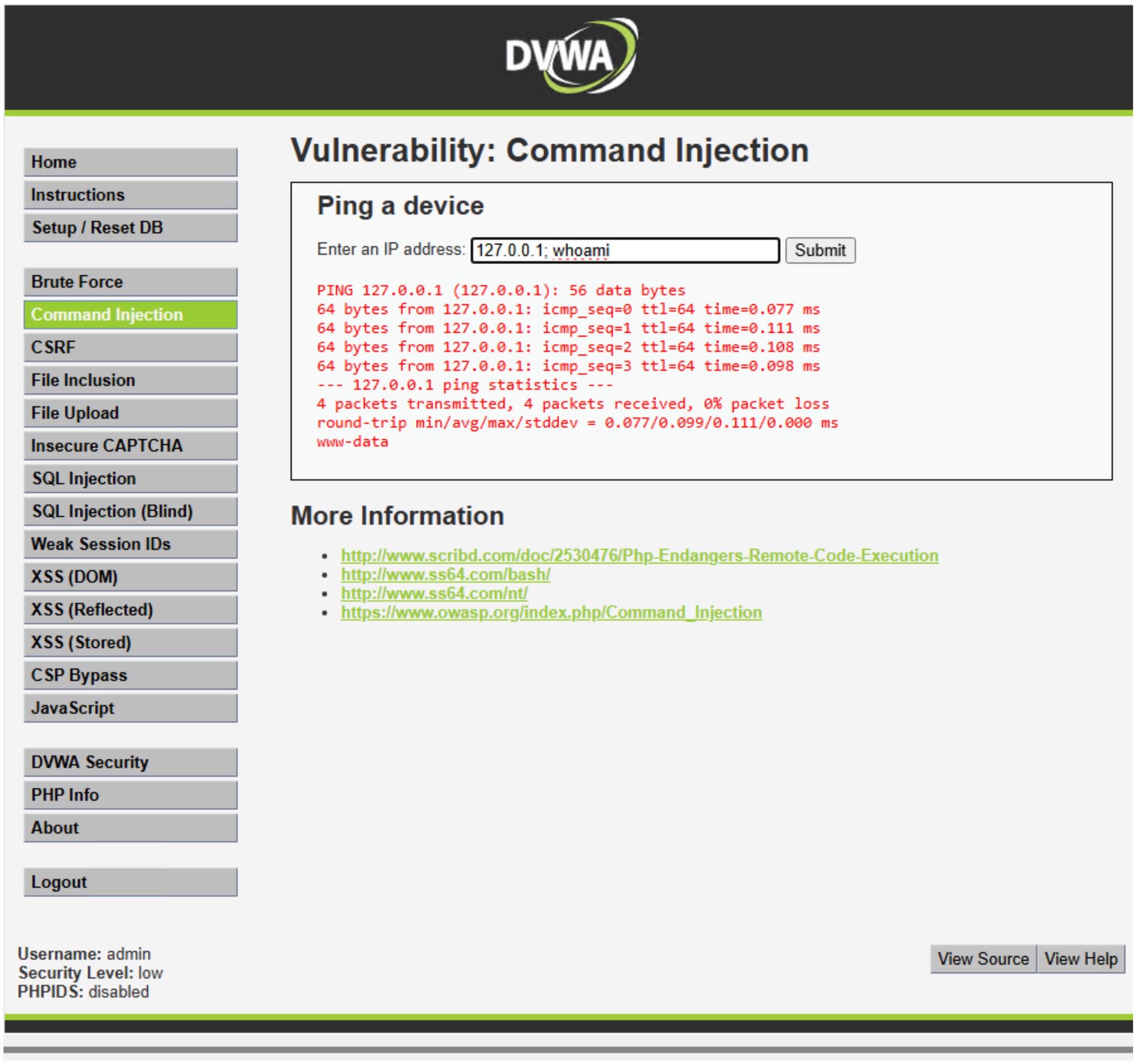
Module: Command Injection
File: `/vulnerabilities/exec/source/low.php`

The vulnerability occurs because user-controlled input (the `ip` parameter) is concatenated into a shell command and executed using `shell_exec`. Without strong validation, an attacker can inject shell metacharacters to execute arbitrary commands.

3.2 Proof of Concept (PoC)

A command injection PoC was demonstrated by appending a second command to the ping input (e.g., using command separators). Evidence was captured in the DVWA output showing unexpected command execution (e.g., `whoami` returning the web server user).

Evidence screenshot placeholder:



Vulnerability: Command Injection

Ping a device

Enter an IP address: 127.0.0.1; whoami

```
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.077 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.111 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.108 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.098 ms
--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.077/0.099/0.111/0.000 ms
www-data
```

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
Security Level: low
PHPIDS: disabled

Figure 1: Command Injection PoC output demonstrating unintended command execution.

3.3 Vulnerable Code (Before)

```php
<?php
if( isset( $_POST[ 'Submit' ]  ) ) {
    $target = $_REQUEST[ 'ip' ];

    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    $html .= "<pre>{$cmd}</pre>";
}
?>
```

3.4 Remediation (Secure Fix)

The remediation applies **allowlist validation** to accept only valid IP addresses and reduces command injection risk by **escaping shell arguments**. Invalid input is rejected with a safe user message.

3.5 Fixed Code (After)

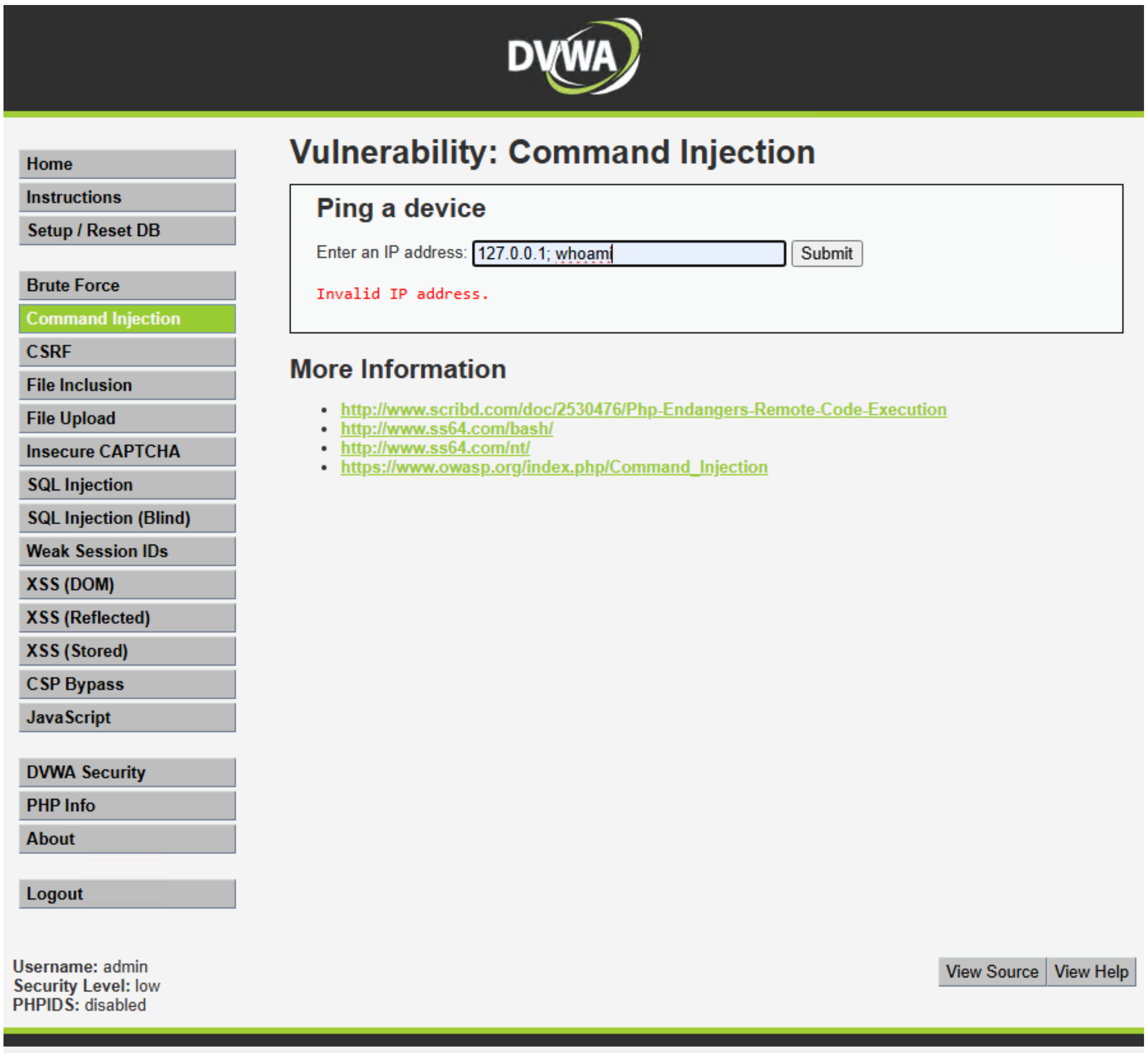
```php
<?php
if (isset($_POST['Submit'])) {

    // DVWA uses REQUEST here
    $target = $_REQUEST['ip'];

    // 1) Allowlist validation: only valid IPv4/IPv6 addresses
    if (filter_var($target, FILTER_VALIDATE_IP) === false) {
        $html .= "<pre>Invalid IP address.</pre>";
        return;
    }

    // 2) Escape argument to prevent shell metacharacter
       injection
    $safeTarget = escapeshellarg($target);

    // Determine OS and execute a fixed ping command
    if (stripos(php_uname('s'), 'Windows') !== false) {
        $cmd = shell_exec("ping $safeTarget");
    } else {
        $cmd = shell_exec("ping -c 4 $safeTarget");
    }

    $html .= "<pre>" . htmlspecialchars($cmd, ENT_QUOTES |
       ENT_SUBSTITUTE, 'UTF-8') . "</pre>";
```

```
}
?>
```

3.6 Verification

After applying the fix, injection strings were rejected (e.g., “Invalid IP address”) while valid IP inputs continued to function. This indicates that untrusted input is no longer able to influence the shell command structure.

Evidence screenshot placeholder:



Vulnerability: Command Injection

Ping a device

Enter an IP address: 127.0.0.1; whoami

```
Invalid 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

Username: admin
Security Level: low
PHPIDS: disabled

Figure 2: Command Injection mitigation verified: malicious input rejected; safe behaviour preserved.

4 Vulnerability 2: Reflected XSS

4.1 Location and Root Cause

Module: Reflected XSS
File: `/vulnerabilities/xss_r/source/low.php`

Reflected XSS occurs when user input is reflected into an HTML response without output encoding. When the input is interpreted as HTML/JavaScript by the browser, scripts can execute in the victim context.

4.2 Proof of Concept (PoC)

A PoC payload such as `<script>alert(1)</script>` was submitted and executed, confirming reflected XSS.

Evidence screenshot placeholder:

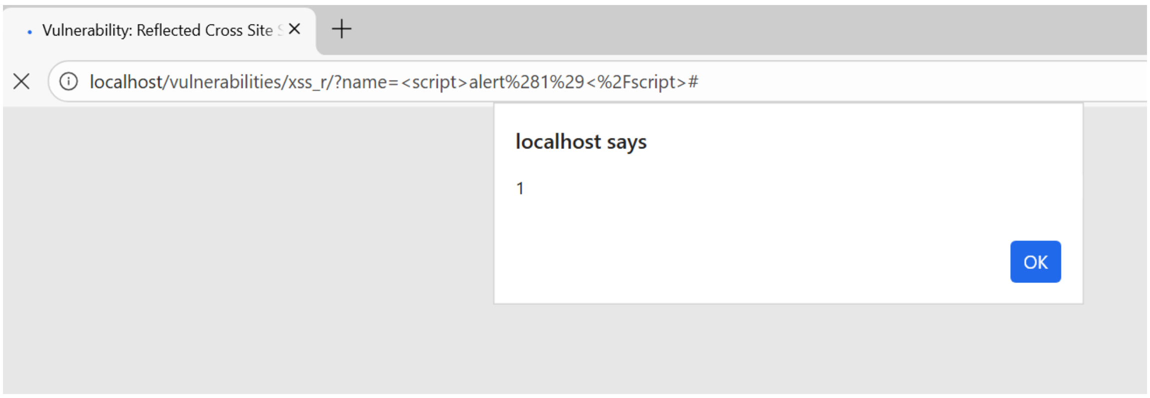

Vulnerability: Reflected Cross Site
localhost/vulnerabilities/xss_r/?name=<script>alert%281%29<%2Fscript>#
localhost says
1
OK

Figure 3: Reflected XSS proof-of-concept demonstrating execution of injected JavaScript via the `name` URL parameter.

4.3 Remediation (Secure Fix)

The remediation applies **context-appropriate output encoding** using `htmlspecialchars` (HTML context). This ensures user input is rendered as text rather than interpreted as HTML.

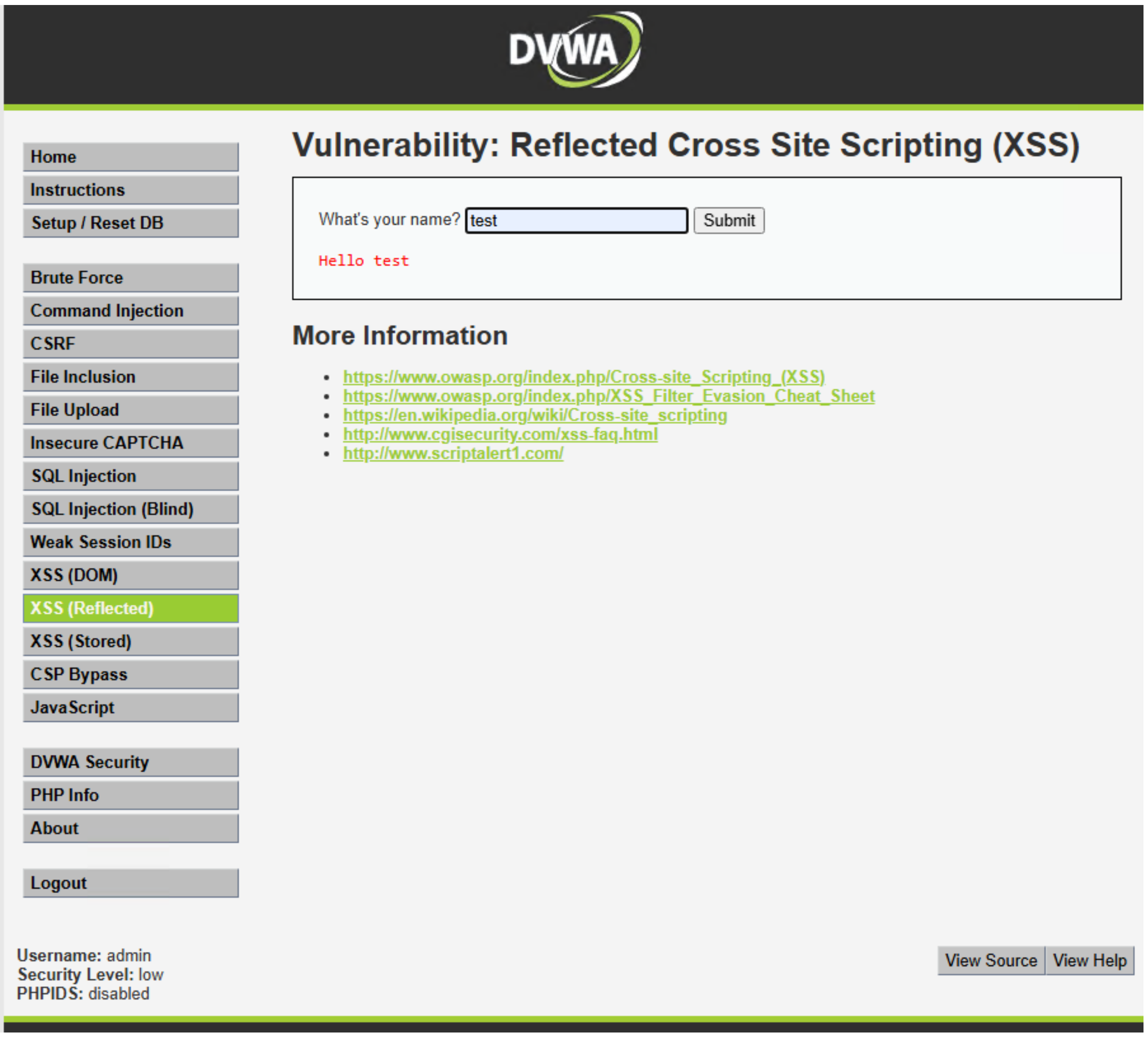
4.4 Fixed Code (After)

```php
<?php
if (isset($_GET['name'])) {
    $name = $_GET['name'];

    // Output encoding prevents script execution (HTML context)
    $safeName = htmlspecialchars($name, ENT_QUOTES |
       ENT_SUBSTITUTE, 'UTF-8');

    echo "<pre>Hello {$safeName}</pre>";
}
?>
```

4.5 Verification

After the fix, the same payload no longer executed as JavaScript and instead appeared as escaped text in the page output.

Evidence screenshot placeholder:



Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name? test

Hello test

More Information

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

Username: admin
Security Level: low
PHPIDS: disabled

Figure 4: Reflected XSS mitigation verified: payload rendered as text, not executed.

5 OWASP ZAP Proxy Inspection (Reflected XSS)

OWASP ZAP was used as a security analysis tool to intercept, inspect, and validate reflected user input at the HTTP layer, supporting both vulnerability identification and verification.

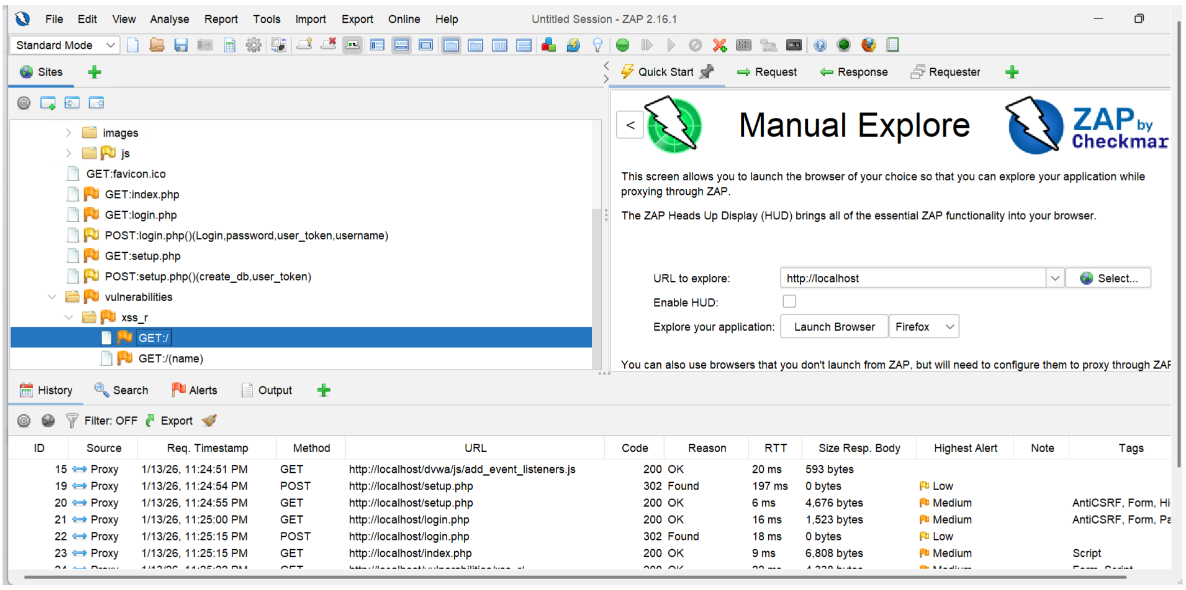
5.1 Method

OWASP ZAP was used in **Manual Explore** mode as an intercepting proxy. Browser traffic to DVWA was routed through ZAP to capture and inspect HTTP requests and responses. This supported vulnerability identification by making the reflected input visible at the HTTP layer.

5.2 Evidence

ZAP captured the request to the reflected XSS endpoint (`/vulnerabilities/xss_r/`) and displayed the injected parameter value (URL-encoded). This confirms tool-assisted inspection rather than browser-only testing.

Evidence screenshot placeholder (your ZAP screenshot):



images
js
GET:favicon.ico
GET:index.php
GET:login.php
POST:login.php()(Login,password,user_token,username)
GET:setup.php
POST:setup.php()(create_db,user_token)
vulnerabilities
xss_r
GET:/
GET:/(name)

This screen allows you to launch the browser of your choice so that you can explore your application while proxying through ZAP.

The ZAP Heads Up Display (HUD) brings all of the essential ZAP functionality into your browser.

URL to explore: http://localhost

Enable HUD: ☐

Explore your application: Launch Browser Firefox

You can also use browsers that you don't launch from ZAP, but will need to configure them to proxy through ZAF

ID	Source	Req. Timestamp	Method	URL	Code	Reason	RTT	Size Resp. Body	Highest Alert	Note	Tags
15	Proxy	1/13/26, 11:24:51 PM	GET	http://localhost/dvwa/js/add_event_listeners.js	200	OK	20 ms	593 bytes			
19	Proxy	1/13/26, 11:24:54 PM	POST	http://localhost/setup.php	302	Found	197 ms	0 bytes	Low		
20	Proxy	1/13/26, 11:24:55 PM	GET	http://localhost/setup.php	200	OK	6 ms	4,676 bytes	Medium		AntiCSRF, Form, Hi
21	Proxy	1/13/26, 11:25:00 PM	GET	http://localhost/login.php	200	OK	16 ms	1,523 bytes	Medium		AntiCSRF, Form, Pa
22	Proxy	1/13/26, 11:25:15 PM	POST	http://localhost/login.php	302	Found	18 ms	0 bytes	Low		
23	Proxy	1/13/26, 11:25:15 PM	GET	http://localhost/index.php	200	OK	9 ms	6,808 bytes	Medium		Script

Figure 5: OWASP ZAP (Manual Explore): intercepted request to `/vulnerabilities/xss_r/` showing reflected input parameter.

6 Vulnerability 3: Insecure File Upload

6.1 Location and Root Cause

Module: File Upload

Files analysed:

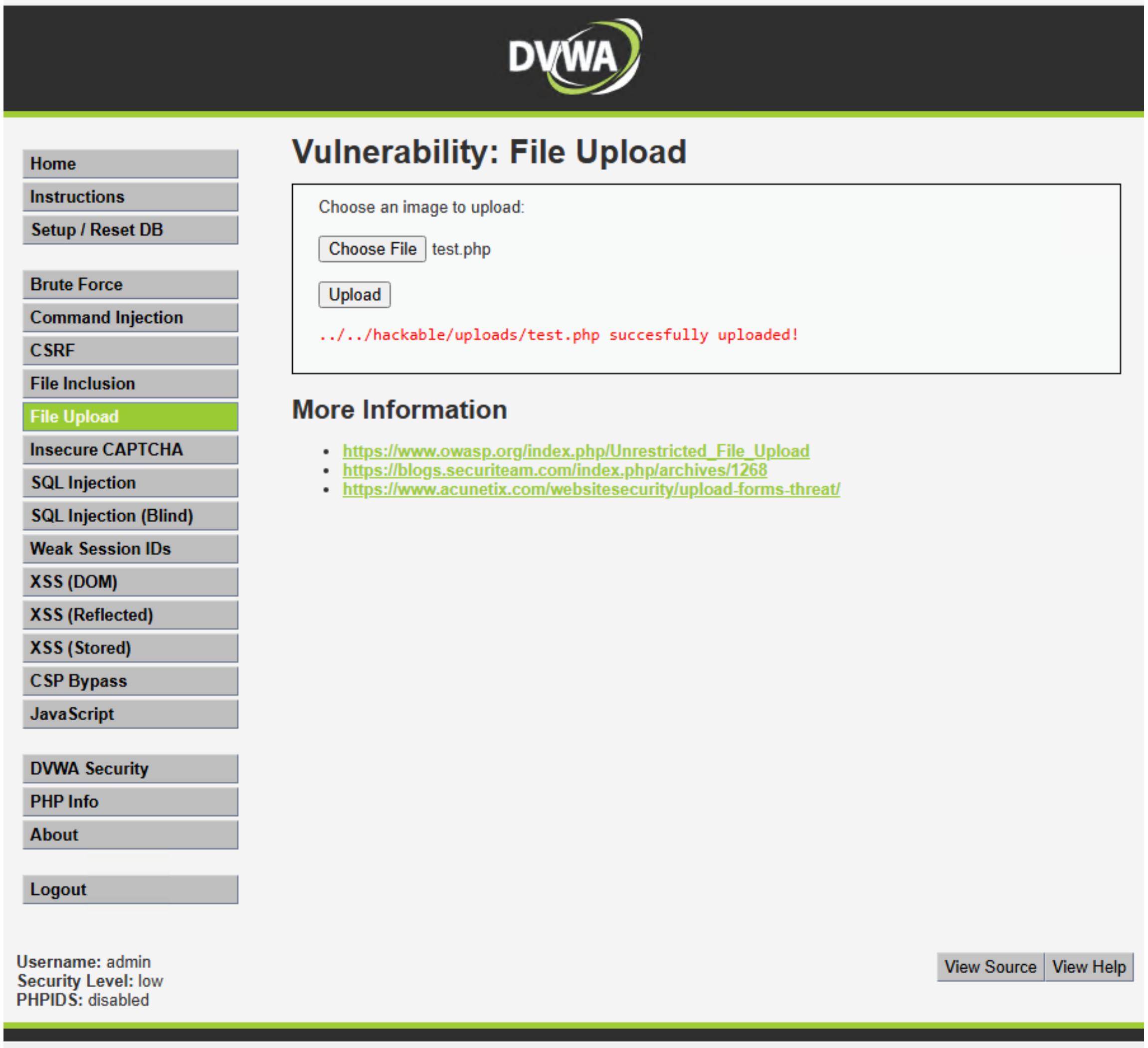
- `/vulnerabilities/upload/source/low.php`
- `/vulnerabilities/upload/source/high.php`

At low security, insufficient server-side validation allows non-image files (including executable PHP) to be uploaded into a web-accessible directory. If the server executes the uploaded file, this can lead to remote code execution (RCE).

6.2 Proof of Concept (PoC)

A PHP file was uploaded and then executed through the uploads directory, demonstrating that the upload path was web-accessible and that server-side controls were inadequate.

Evidence screenshot placeholders:



Vulnerability: File Upload

Choose an image to upload:

Choose File test.php

Upload

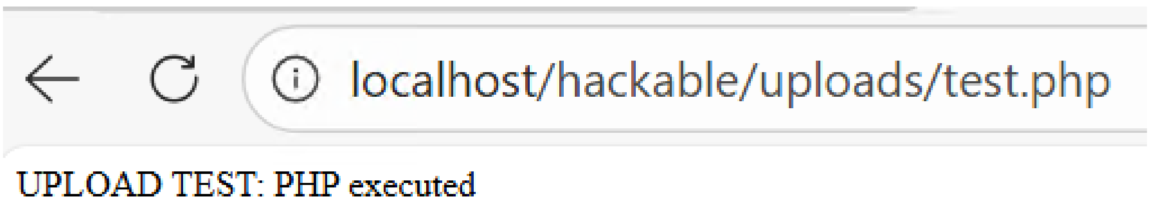
```
../../hackable/uploads/test.php succesfully uploaded!
```

More Information

- 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/

Username: admin
Security Level: low
PHPIDS: disabled

Figure 6: File Upload PoC: malicious file accepted by the application.



localhost/hackable/uploads/test.php

```
UPLOAD TEST: PHP executed
```

Figure 7: File Upload impact: uploaded PHP executed via web-accessible path (RCE condition).

6.3 Analysis of `high.php`

The file `/vulnerabilities/upload/source/high.php` was reviewed to understand stronger server-side validation patterns used by DVWA at a higher security setting. The high-security implementation demonstrates stricter restrictions compared to `low.php` (e.g., tighter type checks and more defensive upload handling). This file was analysed as a reference for secure coding practices; remediation efforts focused on fixing the vulnerable low-security implementation.

6.4 Remediation (Secure Fix)

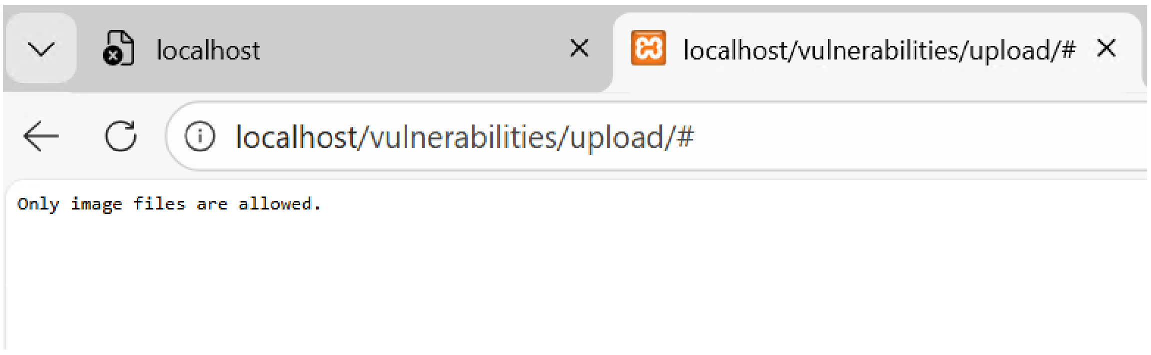
A defense-in-depth approach was applied to `low.php`:

- **Allowlist** only image extensions (e.g., `jpg`, `jpeg`, `png`, `gif`)
- Validate MIME type server-side (not trusting client headers)
- Use image parsing checks where applicable (e.g., `getimagesize`)
- Rename files to a random name to prevent path manipulation and predictable execution
- Reject any non-image file and return a safe error message

6.5 Verification

After the fix, attempts to upload a PHP file were rejected with a restrictive message (e.g., "Only image files are allowed."), confirming the attack was mitigated.

Evidence screenshot placeholder:


localhost
localhost/vulnerabilities/upload/#
localhost/vulnerabilities/upload/#

```
Only image files are allowed.
```

Figure 8: File Upload mitigation verified: non-image/executable file rejected after applying server-side validation.

7 Discussion

7.1 Security Principles Applied

Across the three vulnerabilities, the following secure engineering principles were applied:

- **Allowlist validation:** Accept only known-good inputs for command execution and uploads.
- **Output encoding:** Treat all user input as untrusted and encode before rendering.
- **Defense-in-depth:** Combine extension checks, MIME checks, and content validation for uploads.

- **Least functionality:** Reduce exposure by rejecting inputs that are not required for the feature.

7.2 Ethical Considerations

All testing was performed in a controlled environment (DVWA in Docker) for educational purposes. No testing was conducted against real systems or unauthorised targets.

8 Conclusion

This report demonstrated the practical lifecycle of web vulnerability handling: locating vulnerable code, confirming exploitability, applying secure fixes, and verifying mitigations. Command injection was mitigated through strict IP allowlisting and argument escaping. Reflected XSS was mitigated through consistent output encoding. Insecure file upload was mitigated using defense-in-depth validation controls. OWASP ZAP was used as a proxy to capture and inspect HTTP requests/responses, providing tool-assisted evidence for the reflected XSS testing workflow.

9 References

References

[1] DVWA Project. *Damn Vulnerable Web Application (DVWA).*
Available: `https://github.com/digininja/DVWA`

[2] ATU Cybersecurity and Secure Programming Module, Secure Coding Lecture Notes.

[3] OWASP Foundation. *Cross Site Scripting (XSS).*
Available: `https://owasp.org/www-community/attacks/xss/`

[4] OWASP Foundation. *Command Injection.*
Available: `https://owasp.org/www-community/attacks/Command_Injection`

[5] OWASP Foundation. *Unrestricted File Upload.*
Available: `https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload`

[6] OWASP Foundation. *OWASP ZAP (Zed Attack Proxy).*
Available: `https://www.zaproxy.org/`

Appendix A: Project Repository

The complete source code, including vulnerable implementations, applied security fixes, and version history, is available in the project GitHub repository. The repository also contains supporting documentation and demonstrates the development process through commit history.

GitHub Repository:

`https://github.com/divya-392/DVWA_Project`

Repository Contents

The repository includes:

- Original DVWA source code
- Modified source files after vulnerability remediation
- Clear commit history distinguishing pre-fix and post-fix states
- Supporting documentation for the security assessment