Web Application Security: Practical Testing with DVWA

German del rio guzman A01641976

🛡️ DVWA Vulnerability Report: Command Injection

---

1. OWASP Reference

- OWASP Category: Command Injection (OS Command Injection)

---

2. Vulnerability Summary

📌 Description:

Command Injection is a critical vulnerability where an attacker can execute arbitrary operating system commands on the server hosting the application, often via unsanitized user input passed to shell commands.

⚠️ Impact and Risk Level:

- Impact: High – may allow full system compromise.
- Risk Level: Critical (CVSS 9.8+)
- Attack Consequences: Reading sensitive files, remote code execution, privilege escalation.

🎯 Common Attack Vectors:

- User input fields (forms, URLs, headers) passed directly to `system()`, `exec()`, or shell commands without proper sanitization.
- Using special characters like `;`, `&&`, `|`, or backticks to inject commands.

---

3. Vulnerability Demonstration

🔁 Step-by-Step Reproduction Guide:
Launch DVWA via Docker:

bash
CopiarEditar
```
docker run --platform linux/amd64 -p 8080:80 vulnerables/web-dvwa
```

1.

2. Open http://localhost:8080
   Log in with:
   - Username: `admin`
   - Password: `password`
3. Go to `DVWA Security` → Set Security Level to Low.
4. Navigate to `Command Injection` module.

In the input box, enter:

bash
CopiarEditar
```
127.0.0.1; ls
```

5.
6. Click Submit.

✅ Expected Behavior:

Only the result of `ping 127.0.0.1` should be shown.

❌ Actual Behavior:

The server executes both `ping 127.0.0.1` and `ls`, displaying directory contents – proof of arbitrary command execution.

---

4. Code Analysis

📂 Vulnerable File:
plaintext
CopiarEditar
```
/dvwa/vulnerabilities/exec/source/low.php
```

🔍 Code Snippet:
php
CopiarEditar
```
$target = $_REQUEST['ip'];
$cmd = shell_exec("ping -c 4 " . $target);
echo "<pre>{$cmd}</pre>";
```

⚠️ **Why It's Vulnerable:**

- User input (`$_REQUEST['ip']`) is directly concatenated into a shell command.
- No sanitization or escaping.
- An attacker can inject shell operators like `;`, `&&`, etc.

---

5. Remediation

✅ **Secure Code Example:**

php
CopiarEditar
```
$target = escapeshellarg($_REQUEST['ip']);
$cmd = shell_exec("ping -c 4 {$target}");
echo "<pre>{$cmd}</pre>";
```

🔐 **Best Practices:**

- Always validate and sanitize user input.
- Use safe APIs that don't invoke the shell, like PHP's `proc_open()` with argument arrays.
- Avoid directly passing user input into system-level commands.

🛡️ **Additional Security Measures:**

- Implement Web Application Firewalls (WAF).
- Apply principle of least privilege on the web server.
- Perform static code analysis and security audits.
- Use Content Security Policy (CSP) and logging for suspicious input.

🛡️ DVWA Vulnerability Report: SQL Injection

---

1. OWASP Reference

- **OWASP Category: SQL Injection**
- **OWASP Top 10: A1: Injection (2017), A03:2021 - Injection**

---

2. Vulnerability Summary

📌 Description:

SQL Injection occurs when an attacker manipulates user input that is concatenated into SQL queries without proper validation, allowing them to inject arbitrary SQL code into database operations.

⚠️ Impact and Risk Level:

- **Impact: Extremely high – can lead to full database compromise.**
- **Risk Level: Critical (CVSS 9.8+)**
- **Attack Consequences: Unauthorized data access, bypassing authentication, data deletion, remote code execution (in some cases).**

🎯 Common Attack Vectors:

- **Login forms**
- **Search fields**
- **URL parameters**
- **Any SQL query that uses unsanitized user input**

---

3. Vulnerability Demonstration

🔁 Step-by-Step Reproduction Guide:
Run DVWA using Docker:

bash
CopiarEditar
```
docker run --platform linux/amd64 -p 8080:80 vulnerables/web-dvwa
```

1.
2. Visit http://localhost:8080
   - Username: `admin`
   - Password: `password`
3. Set Security Level to Low from the `DVWA Security` tab.
4. Go to the SQL Injection module.

Enter the following in the User ID field:

bash
CopiarEditar
```
1' OR '1'='1
```

5.
6. Click Submit.

✅ Expected Behavior:

Returns information only about user with ID 1.

❌ Actual Behavior:

Returns all users — indicating that the query logic has been bypassed.

---

4. Code Analysis

📂 Vulnerable File:

plaintext
CopiarEditar
```
/vulnerabilities/sqli/source/low.php
```

🔍 Code Snippet:

php
CopiarEditar
```
$id = $_GET['id'];
```

```
$getid = "SELECT first_name, last_name FROM users WHERE user_id =
'$id';";
$result = mysqli_query($GLOBALS["___mysqli_ston"], $getid);
```

⚠️ **Why It's Vulnerable:**

- **User input (`$_GET['id']`) is directly embedded in the SQL query.**

Query string becomes:

sql
CopiarEditar

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

-
- **This causes the `WHERE` clause to always evaluate to true, returning all rows.**

---

5. Remediation

✅ **Secure Code Example (Using Prepared Statements):**
php
CopiarEditar

```
$id = $_GET['id'];
$stmt = $conn->prepare("SELECT first_name, last_name FROM users
WHERE user_id = ?");
$stmt->bind_param("i", $id);
$stmt->execute();
$result = $stmt->get_result();
```

🔐 **Best Practices:**

- **Always use parameterized queries or prepared statements.**
- **Never concatenate raw user input into SQL strings.**
- **Apply input validation (e.g., numeric only for IDs).**
- **Implement least privilege database roles.**

- Log and monitor abnormal database queries.

🛡️ Additional Security Measures:

- Enable Web Application Firewalls (WAFs).
- Disable detailed error messages in production.
- Periodically scan with tools like sqlmap, Burp Suite, or ZAP.
- Use an ORM (Object-Relational Mapper) with built-in protections.

🛡️ DVWA Vulnerability Report: Cross-Site Scripting (XSS)

---

1. OWASP Reference

- OWASP Category: Cross-Site Scripting (XSS)
- OWASP Top 10: A7:2017 - Cross-Site Scripting, A03:2021 - Injection

---

2. Vulnerability Summary

📌 Description:

Cross-Site Scripting (XSS) allows attackers to inject malicious JavaScript into web pages viewed by other users. This script then executes in the context of the victim's browser, potentially compromising their session or data.

⚠️ Impact and Risk Level:

- Impact: High – especially when stealing session tokens or executing arbitrary JavaScript in another user's browser.
- Risk Level: High

- Attack Consequences: Credential theft, session hijacking, redirecting victims to malicious websites, defacing the site.

🎯 Common Attack Vectors:

- Input fields (search bars, comments, message boxes) rendered without sanitization
- URLs or parameters reflected directly on the page
- Unescaped HTML or JavaScript output

---

3. Vulnerability Demonstration

🔁 Step-by-Step Reproduction Guide:
Run DVWA with Docker:

bash
CopiarEditar
```
docker run --platform linux/amd64 -p 8080:80 vulnerables/web-dvwa
```

1.
2. Open: http://localhost:8080
   Login with:
   - Username: `admin`
   - Password: `password`
3. Set Security Level to Low in `DVWA Security`.
4. Navigate to the XSS (Stored) or XSS (Reflected) module.

Enter the following payload into the input field (e.g., name, message):

html
CopiarEditar
```
<script>alert('XSS')</script>
```

5.
6. Submit the form.

✅ **Expected Behavior:**

Input should be shown as plain text or HTML-escaped.

❌ **Actual Behavior:**

A pop-up alert appears — confirming that JavaScript is executing in the browser.

---

4. Code Analysis

📂 **Vulnerable File:**
plaintext
CopiarEditar
```
/vulnerabilities/xss_r/source/low.php (Reflected)
/vulnerabilities/xss_s/source/low.php (Stored)
```

🔍 **Code Snippet (Reflected XSS):**
php
CopiarEditar
```
$name = $_GET['name'];
echo "Hello $name";
```

⚠️ **Why It's Vulnerable:**

- **User input is output directly into the page without escaping.**
- **The browser interprets `<script>` tags and runs the JavaScript.**
- **No use of `htmlspecialchars()` or encoding functions.**

---

5. Remediation

✅ **Secure Code Example (PHP):**
php
CopiarEditar
```
$name = htmlspecialchars($_GET['name'], ENT_QUOTES, 'UTF-8');
echo "Hello $name";
```

🔐 **Best Practices:**

- Escape user input using context-appropriate encoding:
    - HTML: `htmlspecialchars()` or template engines like Twig/React
    - JS: `JSON.stringify()`, `encodeURIComponent()`
- Use Content Security Policy (CSP) to limit script execution.
- Sanitize input with allowlists for expected characters or formats.
- Avoid using `innerHTML` in frontend JavaScript.

🛡️ Additional Security Measures:

- Implement CSP headers to block inline scripts.
- Use modern frameworks (e.g., React, Angular) that automatically escape HTML.
- Validate and encode data on both server and client side.
- Perform regular vulnerability scanning (e.g., ZAP, Burp Suite).