Vulnerability Analysis Report

File Analyzed: upload_1742300534.php

Total Vulnerabilities: 25

Vulnerability Summary:

Code Injection: 1

RFI (Remote File Inclusion): 1

Shell Injection Vulnerability: 1

SQL Injection: 12

Cross-Site Scripting (XSS): 10

Detailed Vulnerabilities:

Type: Code Injection
Pattern: system($cmd)
Line: 7

Type: SQL Injection
Pattern: SELECT * FROM
Line: 17

Type: SQL Injection
Pattern: $sql = "SELECT * FROM users WHERE id = '$id'"
Line: 17

Type: SQL Injection
Pattern: $_GET['cmd']
Line: 5

Type: SQL Injection
Pattern: $_GET['cmd']
Line: 6

Type: SQL Injection
Pattern: $_GET['id']
Line: 11

Type: SQL Injection
Pattern: $_GET['id']
Line: 12

Type: SQL Injection
Pattern: $_GET['search']
Line: 26

Type: SQL Injection
Pattern: $_GET['search']
Line: 27

```
Type: SQL Injection
Pattern: $_GET['file']
Line: 32

Type: SQL Injection
Pattern: $_GET['file']
Line: 33

Type: SQL Injection
Pattern: $_GET['page']
Line: 38

Type: SQL Injection
Pattern: $_GET['page']
Line: 39

Type: RFI (Remote File Inclusion)
Pattern: include($file)
Line: 34

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['cmd']
Line: 5

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['cmd'];■
Line: 6

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['id']
Line: 11

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['id'];■
Line: 12

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['search']
Line: 26

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['search'];■
Line: 27

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['file']
Line: 32
```

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['file'];■
Line: 33

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['page']
Line: 38

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['page'];■
Line: 39

Type: Shell Injection Vulnerability
Pattern: system($cmd)
Line: 7

Mitigations:

Type: Code Injection
Code Injection (Shell Injection)**
Pattern: system($cmd), Line: 7
Pattern: system($cmd)
*Type:
Type: RFI (Remote File Inclusion)
RFI (Remote File Inclusion)**
Pattern: include($file), Line: 34

1. **Mitigation Strategy:**
   * **Completely avoid using `include()`, `require()`, `include_once()`, and
   * **Disable `allow_url_include` in `php.ini`:** Setting `allow_url_include

2. **Corrected Code Example:**

~~~~
```php
// Instead of:
// $file = $_GET['file'];
// include($file);

// Use a whitelist of allowed files:
$allowed_files = array('page1.php', 'page2.php', 'page3.php');
$file = $_GET['file'];

if (in_array($file, $allowed_files)) {
    include($file);
} else {
    echo "Invalid file."; // Or handle the error appropriately
}
~~~~

3. **Best Practices:**

* **Avoid Dynamic Includes:** Whenever possible, avoid including files dy
* **Whitelist:** If you must include files dynamically, use a strict white
* **Disable `allow_url_include`:** Ensure `allow_url_include = Off` in you
* **File Extension Validation:** If including local files, verify the file

**Type:
Type: Shell Injection Vulnerability
Shell Injection Vulnerability*

1. **Mitigation Strategy:**
    * **Avoid using `system()`, `exec()`, `shell_exec()`, `passthru()`, and si

2. **Corrected Code Example:**

```php
// Instead of:
// $cmd = $_GET['command'];
// system($cmd);

// Try to achieve the same result using PHP's built-in functions.
// Example: If the intention is to list files in a directory:

$directory = '/path/to/directory'; // Hardcode or validate directory
if (is_dir($directory)) {
    $files = scandir($directory);
    if ($files !== false) {
        foreach ($files as $file) {
            echo htmlspecialchars($file) . "<br>"; // Output safely
        }
    } else {
        echo "Error reading directory.";
    }
} else {
    echo "Invalid directory.";
}
```

3. **Best Practices:**

* **Input Validation:** Validate that user input conforms to the expected
* **Escaping:** If you absolutely have to use `system()` or similar, esca
* **Principle of Least Privilege:** Run the web server process with the mi
* **Consider alternatives:** Explore PHP's built-in functions or libraries

*   **Disable Dangerous Functions:** In `php.ini`, disable dangerous functio

**Type:
Type: SQL Injection
SQL Injection**
*Pattern: SELECT * FROM, Line: 17*
Pattern: $sql = "SELECT * FROM users WHERE id = '$id'", Line: 17
Pattern: $_GET['cmd'], Line: 5
Pattern: $_GET['cmd'], Line: 6
Pattern: $_GET['id'], Line: 11
Pattern: $_GET['id'], Line: 12
Pattern: $_GET['search'], Line: 26
Pattern: $_GET['search'], Line: 27
Pattern: $_GET['file'], Line: 32
Pattern: $_GET['file'], Line: 33
Pattern: $_GET['page'], Line: 38
Pattern: $_GET['page'], Line: 39

1. **Mitigation Strategy:**
    *   **Use Prepared Statements (Parameterized Queries):** This is the *most e
    *   **Input Validation:** Validate user input to ensure it matches the expec
    *   **Escaping:** If prepared statements are not possible (though they shoul

2. **Corrected Code Example (using Prepared Statements with MySQLi):**

    ```php
    // Assuming $mysqli is a valid MySQLi connection object
    $id = $_GET['id'];

    // Validate that $id is an integer
    if (!is_numeric($id)) {
        die("Invalid ID"); // Or handle the error appropriately
    }

    $stmt = $mysqli->prepare("SELECT * FROM users WHERE id = ?");
    $stmt->bind_param("i", $id); // "i" indicates that $id is an integer
    $stmt->execute();
    $result = $stmt->get_result();

    while ($row = $result->fetch_assoc()) {
        // Process the data
        echo htmlspecialchars($row['username']) . "<br>"; // Output safely
    }

    $stmt->close();
    ```

3. **Best Practices:**

    *   **Always use Prepared Statements:** Make this your default approach for
    *   **Least Privilege:** Grant database users only the minimum necessary pr
    *   **Input Validation:** Validate all user input to ensure it conforms to
    *   **Error Handling:** Avoid displaying raw database errors to the user, a
    *   **Code Review:** Regularly review code for potential SQL injection vulne
    *   **Web Application Firewalls (WAFs):** Use a WAF to detect and block SQL

**Type:
Type: Cross-Site Scripting (XSS)
Cross-Site Scripting (XSS)**
Pattern: $_GET['cmd'], Line: 5
Pattern: $_GET['cmd'];, Line: 6
Pattern: $_GET['id'], Line: 11
Pattern: $_GET['id'];, Line: 12
Pattern: $_GET['search'], Line: 26
Pattern: $_GET['search'];, Line: 27
Pattern: $_GET['file'], Line: 32
Pattern: $_GET['file'];, Line: 33
Pattern: $_GET['page'], Line: 38
Pattern: $_GET['page'];, Line: 39

1. **Mitigation Strategy:**
    *   **Output Encoding (Escaping):** Encode all user-supplied data *before* d
    *   **Input Sanitization:** Sanitize user input to remove or escape potentia
    *   **Content Security Policy (CSP):** Implement a CSP to restrict the sourc

2. **Corrected Code Example (HTML Context - using `htmlspecialchars()`):**

    ```php
    $search_term = $_GET['search'];
    echo "<p>You searched for: " . htmlspecialchars($search_term) . "</p>";
    ```

3. **Best Practices:**

    *   **Output Encoding:** Always encode data before outputting it. Use the a
        *   **HTML:** `htmlspecialchars()` (most common)
        *   **URL:** `urlencode()`
        *   **JavaScript:** `json_encode()` (for data passed to JavaScript)
        *   **CSS:** Use CSS escaping techniques or avoid directly embedding us
    *   **Context-Aware Encoding:** Choose the correct encoding function based
    *   **Content Security Policy (CSP):** Implement a CSP to restrict the sourc
    *   **Input Validation/Sanitization:** Sanitize input as a defense-in-depth
    *   **HTTPOnly Cookies:** Set the `HttpOnly` flag on cookies to prevent Java
    *   **Regularly Update Libraries:** Keep your web frameworks and libraries

*   **Consider a Template Engine:** Modern template engines often have built

In summary, prioritize prepared statements for SQL injection, avoid shell comman