Vulnerability Analysis Report

File Analyzed: upload_1742829123.php

Total Vulnerabilities: 32

Vulnerability Summary:

File Upload Vulnerability: 8

Code Injection: 3

Command Injection: 1

SQL Injection: 11

Cross-Site Scripting (XSS): 7

Shell Injection Vulnerability: 2

Detailed Vulnerabilities:

Type: Code Injection
Pattern: eval($_GET["cmd"])
Line: 212

Type: Code Injection
Pattern: eval($_GET["cmd"])
Line: 212

Type: Code Injection
Pattern: eval($_GET["cmd"])
Line: 212

Type: SQL Injection
Pattern: SELECT * FROM
Line: 77

Type: SQL Injection
Pattern: DELETE FROM
Line: 87

Type: SQL Injection
Pattern: OR '1'='1"
Line: 73

Type: SQL Injection
Pattern: $maliciousUsername = "' OR '1'='1"
Line: 73

Type: SQL Injection
Pattern: $maliciousInput = "'; alert('xss'); '"
Line: 100

```
Type: SQL Injection
Pattern: $_POST['username']
Line: 199

Type: SQL Injection
Pattern: $_POST['password']
Line: 200

Type: SQL Injection
Pattern: $_POST['username']
Line: 202

Type: SQL Injection
Pattern: $_POST['password']
Line: 202

Type: SQL Injection
Pattern: $_POST['password']
Line: 203

Type: SQL Injection
Pattern: $_GET["cmd"]
Line: 212

Type: Cross-Site Scripting (XSS)
Pattern: <script>alert("xss")</script>
Line: 92

Type: Cross-Site Scripting (XSS)
Pattern: $_POST['username']
Line: 199

Type: Cross-Site Scripting (XSS)
Pattern: $_POST['password']
Line: 200

Type: Cross-Site Scripting (XSS)
Pattern: $_POST['username']
Line: 202

Type: Cross-Site Scripting (XSS)
Pattern: $_POST['password']
Line: 202

Type: Cross-Site Scripting (XSS)
Pattern: $_POST['password']
Line: 203
```

Type: Cross-Site Scripting (XSS)
Pattern: $_GET["cmd"]
Line: 212

Type: Command Injection
Pattern: eval($_GET["cmd"])
Line: 212
Mitigation: Validate and sanitize user inputs. Use allowlists instead of executing raw input commands.

Type: Shell Injection Vulnerability
Pattern: eval($_GET["cmd"])
Line: 212

Type: Shell Injection Vulnerability
Pattern: eval($_GET["cmd"])
Line: 212

Type: File Upload Vulnerability
Pattern: ../
Line: 2

Type: File Upload Vulnerability
Pattern: ../
Line: 3

Type: File Upload Vulnerability
Pattern: ../
Line: 145

Type: File Upload Vulnerability
Pattern: ../
Line: 153

Type: File Upload Vulnerability
Pattern: '/../vendor/autoload.php'
Line: 2

Type: File Upload Vulnerability
Pattern: '/../uploads/secure_version.php'
Line: 3

Type: File Upload Vulnerability
Pattern: '../config.php'
Line: 145

Type: File Upload Vulnerability
Pattern: '/../includes/header.php'
Line: 153

Mitigations:

Type: File Upload Vulnerability

File Upload Vulnerability (Various Patterns, Lines: 2, 3, 145, 153)**

*   **Mitigation Strategy:**
    *   **Validate the file extension:** Only allow specific, safe file types (e
    *   **Check the file's MIME type:** Verify that the MIME type matches the e
    *   **Randomize the filename:** Rename the uploaded file to a randomly gene
    *   **Store uploaded files outside the web root:** This prevents direct acc
    *   **Disable execution of uploaded files:** Configure your web server to p
    *   **Limit file size:** Restrict the maximum file size that can be uploade
    *   **Sanitize the filename:** Remove or replace any potentially dangerous c

*   **Corrected Code (Example - Basic File Upload with Validation):**

```php
$allowed_extensions = ['jpg', 'jpeg', 'png', 'gif'];
$upload_dir = '/path/to/secure/uploads/';  // Outside the web root!

if (isset($_FILES['upload'])) {
    $file_name = $_FILES['upload']['name'];
    $file_tmp = $_FILES['upload']['tmp_name'];
    $file_size = $_FILES['upload']['size'];
    $file_error = $_FILES['upload']['error'];

    $file_ext = strtolower(pathinfo($file_name, PATHINFO_EXTENSION));

    if (in_array($file_ext, $allowed_extensions)) {
        if ($file_error === 0) {
            if ($file_size <= 2097152) { // 2MB limit
                $file_new_name = uniqid('', true) . '.' . $file_ext; // Rand
                $file_destination = $upload_dir . $file_new_name;

                if (move_uploaded_file($file_tmp, $file_destination)) {
                    echo "File uploaded successfully!";
                } else {
                    echo "Error uploading file.";
                }
            } else {
                echo "File size exceeds the limit.";
            }
        } else {
            echo "Error uploading file.";
        }
    } else {
        echo "Invalid file type.";
    }
}
```

~~~

*   **Best Practices:**
    *   **Use a whitelist approach for allowed file extensions.**
    *   **Verify the MIME type server-side.**
    *   **Randomize filenames.**
    *   **Store uploaded files outside the web root.**
    *   **Disable execution of uploaded files.**
    *   **Limit file size.**
    *   **Sanitize filenames.**
    *   Implement robust error handling and logging.
    *   Use a dedicated file storage service (e.g., Amazon S3) if possible.
    *   Regularly scan the upload directory for malicious files.

By implementing these mitigations and following the recommended best practices,

Type: Code Injection

Code Injection (eval($_GET["cmd"]), Line: 212)**

*   **Mitigation Strategy:** **NEVER** use `eval()` with user-supplied data. I

*   **Corrected Code (Example - Hypothetical Scenario):**

    ```php
    // Assuming the intent was to perform a mathematical operation
    if (isset($_GET['operation']) && isset($_GET['number1']) && isset($_GET['num
        $operation = $_GET['operation'];
        $number1 = floatval($_GET['number1']); // Convert to float for safety
        $number2 = floatval($_GET['number2']); // Convert to float for safety

        switch ($operation) {
            case 'add':
                $result = $number1 + $number2;
                break;
            case 'subtract':
                $result = $number1 - $number2;
                break;
            case 'multiply':
                $result = $number1 * $number2;
                break;
            case 'divide':
                if ($number2 != 0) {
                    $result = $number1 / $number2;
                } else {
                    $result = "Division by zero error!";
                }
                break;
            default:
~~~

~~~
                    $result = "Invalid operation!";
            }
            echo "Result: " . $result;
        } else {
            echo "Please provide operation, number1, and number2 parameters.";
        }
        ```

    *   **Best Practices:**
        *   Avoid `eval()` entirely.
        *   Use parameterized queries or prepared statements for database interactio
        *   Use specific functions (e.g., `intval()`, `floatval()`) for type casting
        *   Implement a strict allowlist of permitted characters or operations if us
        *   Consider using a secure templating engine that automatically escapes out

    **2.
Type: Command Injection
    Mitigation for Command Injection not provided by the API.
Type: SQL Injection
    SQL Injection (Various Patterns, Lines: 73, 77, 87, 100, 199, 200, 202, 203)**

    *   **Mitigation Strategy:** Use **Prepared Statements with Parameterized Queri

    *   **Corrected Code (Example using Prepared Statements - PDO):**

        ```php
        // Assuming a database connection $pdo exists
        $username = $_POST['username'];
        $password = $_POST['password'];

        $stmt = $pdo->prepare("SELECT * FROM users WHERE username = :username AND pa
        $stmt->bindParam(':username', $username);
        $stmt->bindParam(':password', $password); // Properly hash and salt password
        $stmt->execute();

        $user = $stmt->fetch(PDO::FETCH_ASSOC);

        if ($user) {
            // Successful login
            echo "Login successful!";
        } else {
            echo "Invalid username or password.";
        }
        ```

    *   **Best Practices:**
        *   **Always use prepared statements with parameterized queries.**
~~~

    *   Use a database abstraction layer (DBAL) for portability and security.
    *   Apply the principle of least privilege to database user accounts.
    *   Regularly update database drivers and the database server itself.
    *   Implement input validation to prevent unexpected data types or formats.
    *   Use a strong hashing algorithm (e.g., bcrypt, Argon2) to store passwords

**3.

Type: Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) (Various Patterns, Lines: 92, 199, 200, 202, 203, 212

*   **Mitigation Strategy:** **Output Encoding/Escaping**. Encode all user-sup

*   **Corrected Code (Example using `htmlspecialchars()` - HTML context):**

    ```php
    $username = $_POST['username'];
    echo "Welcome, " . htmlspecialchars($username, ENT_QUOTES, 'UTF-8') . "!"; /
    ```

*   **Best Practices:**
    *   **Encode all user-supplied data before outputting it.**
    *   Use context-aware encoding functions (e.g., `htmlspecialchars()`, `urlen
    *   Consider using a Content Security Policy (CSP) to restrict the sources f
    *   Use a templating engine with automatic escaping.
    *   Implement input validation to filter out potentially malicious character
    *   Set the `HttpOnly` flag on cookies to prevent JavaScript from accessing
    *   Sanitize HTML input using a library like HTML Purifier if you need to al

4. Command/Shell Injection (eval($_GET["cmd"]), Line: 212)

*   **Mitigation Strategy:** As stated above, **NEVER** use `eval()` with user-s

*   **Corrected Code (Example using `proc_open()` with Validation - Highly Hypot

    ```php
    if (isset($_GET['command'])) {
        $command = $_GET['command'];

        // **EXTREMELY IMPORTANT:  This is just an example. You MUST carefully v
        // **DO NOT use this code in a production environment without thorough s
        // **A whitelist of allowed commands is essential.**

        // Example: Allow only 'ls' command with a specific directory
        if (strpos($command, 'ls /safe/directory') === 0) {
            $descriptorspec = array(
               0 => array("pipe", "r"),  // stdin is a pipe that the child will
               1 => array("pipe", "w"),  // stdout is a pipe that the child will
    ```

```
           2 => array("pipe", "w")   // stderr is a pipe that the child will
        );

        $process = proc_open($command, $descriptorspec, $pipes);

        if (is_resource($process)) {
            // $pipes now looks like this:
            // 0 => writeable handle connected to child stdin
            // 1 => readable handle connected to child stdout
            // Any error output will be appended to /tmp/error-output.txt

            $stdout = stream_get_contents($pipes[1]);
            fclose($pipes[1]);

            $stderr = stream_get_contents($pipes[2]);
            fclose($pipes[2]);

            $return_value = proc_close($process);

            echo "Output: " . htmlspecialchars($stdout) . "<br>";
            if (!empty($stderr)) {
                echo "Error: " . htmlspecialchars($stderr) . "<br>";
            }
            echo "Return code: " . $return_value . "<br>";
        } else {
            echo "Failed to execute command.";
        }
    } else {
        echo "Invalid command.";
    }
}
```

*   **Best Practices:**
    *   **Avoid executing shell commands whenever possible.** Use PHP's built-i
    *   If you must execute shell commands, use `proc_open()` instead of `system
    *   **Thoroughly validate and sanitize all input.**
    *   Implement a strict whitelist of allowed commands.
    *   Use the principle of least privilege for the user account running the we
    *   Disable shell access for the web server user if possible.

**5.

Type: Shell Injection Vulnerability

Mitigation for Shell Injection Vulnerability not provided by the API.