Vulnerability Analysis Report

File Analyzed: upload_1742820348.php

Total Vulnerabilities: 61

Vulnerability Summary:

RFI (Remote File Inclusion): 1

Code Injection: 5

SQL Injection: 23

File Upload Vulnerability: 5

Cross-Site Scripting (XSS): 24

Shell Injection Vulnerability: 1

Cryptographic Vulnerability: 2

Detailed Vulnerabilities:

Type: Code Injection
Pattern: exec($command)
Line: 41

Type: Code Injection
Pattern: exec("echo " . $user_input . " > output.txt")
Line: 61

Type: Code Injection
Pattern: system($command)
Line: 8

Type: Code Injection
Pattern: exec($command)
Line: 41

Type: Code Injection
Pattern: exec("echo " . $user_input . " > output.txt")
Line: 61

Type: SQL Injection
Pattern: SELECT * FROM
Line: 15

Type: SQL Injection
Pattern: $query = "SELECT * FROM users WHERE username = '$username' AND password = '$password'"
Line: 15

Type: SQL Injection
Pattern: exec("echo " . $
Line: 61

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

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

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

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

Type: SQL Injection
Pattern: $_GET['username']
Line: 13

Type: SQL Injection
Pattern: $_GET['password']
Line: 14

Type: SQL Injection
Pattern: $_GET['file']
Line: 20

Type: SQL Injection
Pattern: $_GET['file']
Line: 21

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

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

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

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

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

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

Type: SQL Injection
Pattern: $_GET['password']
Line: 45

Type: SQL Injection
Pattern: $_GET['password']
Line: 46

Type: SQL Injection
Pattern: $_GET['url']
Line: 52

Type: SQL Injection
Pattern: $_GET['url']
Line: 53

Type: SQL Injection
Pattern: $_GET['input']
Line: 59

Type: SQL Injection
Pattern: $_GET['input']
Line: 60

Type: RFI (Remote File Inclusion)
Pattern: include($file)
Line: 22

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

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

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['username']
Line: 12
```

```
Type: Cross-Site Scripting (XSS)
Pattern: $_GET['password']
Line: 12

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['username'];
Line: 13

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['password'];
Line: 14

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['file']
Line: 20

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['file'];
Line: 21

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

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

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

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

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

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

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['password']
Line: 45
```

```
Type: Cross-Site Scripting (XSS)
Pattern: $_GET['password'];
Line: 46

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['url']
Line: 52

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['url'];
Line: 53

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['input']
Line: 59

Type: Cross-Site Scripting (XSS)
Pattern: $_GET['input'];
Line: 60

Type: Cross-Site Scripting (XSS)
Pattern: $_FILES['file']
Line: 65

Type: Cross-Site Scripting (XSS)
Pattern: $_FILES['file']
Line: 66

Type: Cross-Site Scripting (XSS)
Pattern: $_FILES['file']
Line: 67

Type: Cross-Site Scripting (XSS)
Pattern: $_FILES['file']
Line: 68

Type: Cryptographic Vulnerability
Pattern: md5(
Line: 47

Type: Cryptographic Vulnerability
Pattern: password = '$password'"
Line: 15

Type: Shell Injection Vulnerability
Pattern: system($command)
Line: 8
```

Type: File Upload Vulnerability
Pattern: move_uploaded_file($_FILES['file']['tmp_name'], "/uploads/" . $filename);
Line: 68

Type: File Upload Vulnerability
Pattern: $_FILES['file']['name']
Line: 67

Type: File Upload Vulnerability
Pattern: $_FILES['file']['tmp_name']
Line: 68

Type: File Upload Vulnerability
Pattern: "echo " . $user_input . " > output.txt"
Line: 40

Type: File Upload Vulnerability
Pattern: "echo " . $user_input . " > output.txt"
Line: 61

Mitigations:

Type: RFI (Remote File Inclusion)
Mitigation for RFI (Remote File Inclusion) not provided by the API.
Type: Code Injection
Code Injection**

*   **Vulnerability:** The `exec()` and `system()` functions execute operating s

*   **Lines:** 8, 41, 61

*   **Mitigation:**

    1. **Avoid using `exec()`, `system()`, `passthru()`, `shell_exec()` wheneve
    2. **Input Validation and Sanitization:** If executing shell commands is ab
    3. **Escaping:** Use `escapeshellarg()` to properly escape arguments passe
    4. **Principle of Least Privilege:** Ensure the web server process runs wi

*   **Corrected Code (Example for Line 41, assuming `$command` is derived from `

    ```php
    //Option 1: Avoid shell execution entirely (Best)
    //If possible, find a PHP function that does what you need without shell acc

    //Option 2:  Strict validation and escaping (Use only if absolutely necessar
    $cmd = $_GET['cmd'];
    // Validate $cmd.  Example:  Allow only specific commands and arguments.
    ```

```
if (preg_match('/^(command1|command2) -[a-z]+$/', $cmd)) { // Example whitel
    $command = escapeshellcmd($cmd); // Be cautious using this function. Pre
    exec($command, $output, $return_var);
    // Process $output and $return_var
} else {
    // Log the invalid command attempt
    error_log("Invalid command attempt: " . $cmd);
    // Display an error message to the user (without revealing details)
    echo "Invalid input.";
}

//Corrected Code (Example for Line 61, preventing arbitrary file writes)
$user_input = $_GET['input'];
$safe_filename = 'output.txt'; //Hardcode the filename to prevent injection
$escaped_input = escapeshellarg($user_input);
exec("echo " . $escaped_input . " > " . $safe_filename, $output, $return_var
```

*   **Best Practices:**

    *   **Principle of Least Privilege:** Run the web server with the fewest nec
    *   **Input Validation:** Validate all input, even from trusted sources.
    *   **Output Encoding:** Encode output to prevent XSS (see below).
    *   **Regular Security Audits:** Review code regularly for potential vulner
    *   **Use a Web Application Firewall (WAF):** WAFs can detect and block many

**
Type: SQL Injection
SQL Injection**

*   **Vulnerability:** SQL injection occurs when user-supplied data is incorpor

*   **Lines:** 5, 6, 12, 13, 14, 15, 20, 21, 26, 27, 32, 33, 38, 39, 45, 46, 52,

*   **Mitigation:**

    1. **Prepared Statements (Parameterized Queries):** This is the *most effec
    2. **Escaping (Deprecated):** While `mysqli_real_escape_string()` (or the
    3. **Input Validation:** Validate user input to ensure it conforms to the e
    4. **Principle of Least Privilege:** Use database accounts with the minima
    5. **Error Handling:** Avoid displaying detailed database error messages t

*   **Corrected Code (Example for Line 15, using prepared statements with MySQLi

    ```php
    // Assuming $username and $password come from $_GET or $_POST
    $username = $_GET['username'];
    ```

```
$password = $_GET['password'];

// Establish database connection (replace with your actual credentials)
$mysqli = new mysqli("localhost", "username", "password", "database");

// Check connection
if ($mysqli->connect_error) {
    die("Connection failed: " . $mysqli->connect_error);
}

// Prepare the SQL statement
$query = "SELECT * FROM users WHERE username = ? AND password = ?";
$stmt = $mysqli->prepare($query);

if ($stmt) {
    // Bind parameters
    $stmt->bind_param("ss", $username, $password); // "ss" indicates two str

    // Execute the query
    $stmt->execute();

    // Get the result
    $result = $stmt->get_result();

    // Process the result
    if ($result->num_rows > 0) {
        // User authenticated
        while($row = $result->fetch_assoc()) {
            // Process user data
            echo "Welcome " . htmlspecialchars($row["username"]); // Encode
        }
    } else {
        // Authentication failed
        echo "Invalid username or password.";
    }

    // Close the statement and connection
    $stmt->close();
    $mysqli->close();
} else {
    // Handle the error (e.g., log it)
    echo "Error preparing statement: " . $mysqli->error;
}
```

*   **Best Practices:**

    *   **Always use prepared statements (parameterized queries).**
    *   **Validate input data.**
    *   **Use a database abstraction layer (e.g., PDO) for portability and secur
    *   **Regularly update your database software.**
    *   **Follow the principle of least privilege for database access.**

Remote File Inclusion (RFI)

*   **Vulnerability:** RFI allows an attacker to include and execute remote file

*   **Line:** 22

*   **Mitigation:**

    1. **Disable `allow_url_include`:** The *most effective* mitigation is to
    2. **Whitelist Local Files:** If you need to include files, strictly whitel
    3. **Input Validation:** Validate the file path provided by the user to ens
    4. **Never trust user input for file paths.**

*   **Corrected Code (Example for Line 22, assuming $file comes from $_GET['file

    ```php
    $file = $_GET['file'];

    // Whitelist approach
    $allowed_files = array(
        'file1.php',
        'file2.php',
        'includes/header.php'
    );

    if (in_array($file, $allowed_files)) {
        include($file);
    } else {
        // Log the attempt
        error_log("RFI attempt: " . $file);
        echo "Invalid file.";
    }

    //Alternative using realpath and a base directory:
    $base_dir = '/path/to/your/includes/';
    $safe_file = realpath($base_dir . basename($file)); // basename prevents pat

    if (strpos($safe_file, realpath($base_dir)) === 0) { // Check if the file is
        include($safe_file);
    } else {
        error_log("RFI attempt: " . $file);
    ```

        echo "Invalid file.";
    }
    ~~~

*   **Best Practices:**

    *   **Disable `allow_url_include` in `php.ini`.**
    *   **Use a whitelist of allowed files for inclusion.**
    *   **Never directly use user input to specify the file to include.**
    *   **Use `realpath()` to resolve file paths and prevent directory traversal

**
Type: File Upload Vulnerability
File Upload Vulnerability**

*   **Vulnerability:** Unrestricted file uploads can allow attackers to upload m

*   **Lines:** 40, 61, 67, 68

*   **Mitigation:**

    1. **Validate File Type:** Check the file's MIME type using `mime_content_t
    2. **Validate File Extension (with caution):** While MIME type checking is
    3. **Sanitize Filename:** Sanitize the filename to remove or replace any po
    4. **Limit File Size:** Restrict the maximum allowed file size to prevent d
    5. **Store Uploads Outside Web Root:** Store uploaded files outside the web
    6. **Disable Execution:** If possible, disable script execution in the uplo
    7. **Content Security Policy (CSP):** Use CSP to restrict the execution of

*   **Corrected Code (Example for Lines 67 and 68):**

    ```php
    $allowed_extensions = array('jpg', 'jpeg', 'png', 'gif');
    $max_file_size = 204800; // 200KB
    $upload_dir = '/path/to/your/uploads/'; //Outside web root

    $filename = $_FILES['file']['name'];
    $file_tmp = $_FILES['file']['tmp_name'];
    $file_size = $_FILES['file']['size'];

    // 1. Validate file type (MIME type)
    $finfo = finfo_open(FILEINFO_MIME_TYPE);
    $file_mime_type = finfo_file($finfo, $file_tmp);
    finfo_close($finfo);

    if (!in_array($file_mime_type, ['image/jpeg', 'image/png', 'image/gif'])) {
        echo "Invalid file type.";
    ~~~

```
    exit;
}

// 2. Validate file extension (as a secondary check)
$file_ext = strtolower(pathinfo($filename, PATHINFO_EXTENSION));
if (!in_array($file_ext, $allowed_extensions)) {
    echo "Invalid file extension.";
    exit;
}

// 3. Validate file size
if ($file_size > $max_file_size) {
    echo "File size exceeds the limit.";
    exit;
}

// 4. Sanitize filename
$new_filename = uniqid() . '.' . $file_ext; // Generate a unique filename
$destination = $upload_dir . $new_filename;

// 5. Move the uploaded file
if (move_uploaded_file($file_tmp, $destination)) {
    echo "File uploaded successfully.";
} else {
    echo "Error uploading file.";
}
```

*   **Best Practices:**

    *   **Validate file type using MIME type checking.**
    *   **Sanitize filenames to prevent directory traversal and other attacks.**
    *   **Limit file size to prevent denial-of-service attacks.**
    *   **Store uploads outside the web root and serve them through a script.**
    *   **Disable script execution in the upload directory.**
    *   **Use Content Security Policy (CSP) to restrict script execution.**

By implementing these mitigation strategies and following the best practices, yo

Type: Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS)**

*   **Vulnerability:** XSS allows attackers to inject malicious scripts into web

*   **Lines:** 5, 6, 12, 13, 14, 20, 21, 26, 27, 32, 33, 38, 39, 45, 46, 52, 53,

*   **Mitigation:**

1. **Output Encoding/Escaping:** The *most important* defense is to encode
    *   `htmlspecialchars()`: For general HTML output.
    *   `urlencode()`: For URLs.
    *   `json_encode()`: For JSON data.
    *   For JavaScript contexts, use `json_encode()` and ensure proper quoti
2. **Input Validation:** While not a primary defense against XSS, validatin
3. **Content Security Policy (CSP):** CSP is a browser security mechanism t
4. **HTTPOnly Cookies:** Set the `HttpOnly` flag on cookies to prevent Java

*   **Corrected Code (Example for Line 5, assuming `$input` comes from `$_GET['i

```php
$input = $_GET['input'];
echo htmlspecialchars($input, ENT_QUOTES, 'UTF-8'); // Encode for HTML outpu
```

*   **Best Practices:**

    *   **Encode all output based on context.** Don't just blindly use `htmlspe
    *   **Implement Content Security Policy (CSP).**
    *   **Set `HttpOnly` flag on cookies.**
    *   **Validate input data.**
    *   **Use a templating engine that automatically escapes output.**

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

*   **Vulnerability:** The `system()` function executes operating system command

*   **Line:** 8

*   **Mitigation:**

    1. **Avoid using `system()`, `exec()`, `passthru()`, `shell_exec()` wheneve
    2. **Input Validation and Sanitization:** If executing shell commands is ab
    3. **Escaping:** Use `escapeshellarg()` to properly escape arguments passe
    4. **Principle of Least Privilege:** Ensure the web server process runs wi

*   **Corrected Code (Example for Line 8, assuming `$command` is derived from `$

```php
//Option 1: Avoid shell execution entirely (Best)
//If possible, find a PHP function that does what you need without shell acc

//Option 2:  Strict validation and escaping (Use only if absolutely necessar
$cmd = $_GET['cmd'];
```

```
// Validate $cmd.  Example:  Allow only specific commands and arguments.
if (preg_match('/^(command1|command2) -[a-z]+$/', $cmd)) { // Example whitel
    $command = escapeshellcmd($cmd); // Be cautious using this function. Pre
    system($command, $return_var);
    // Process $return_var
} else {
    // Log the invalid command attempt
    error_log("Invalid command attempt: " . $cmd);
    // Display an error message to the user (without revealing details)
    echo "Invalid input.";
}
```

*   **Best Practices:**

    *   **Principle of Least Privilege:** Run the web server with the fewest nec
    *   **Input Validation:** Validate all input, even from trusted sources.
    *   **Output Encoding:** Encode output to prevent XSS (see above).
    *   **Regular Security Audits:** Review code regularly for potential vulner
    *   **Use a Web Application Firewall (WAF):** WAFs can detect and block many

**
Type: Cryptographic Vulnerability
Cryptographic Vulnerability**

*   **Vulnerability:** Using `md5()` for password hashing is extremely insecure.

*   **Lines:** 15, 47

*   **Mitigation:**

    1. **Use `password_hash()`:** Use PHP's built-in `password_hash()` function
    2. **Use `password_verify()`:** When verifying a password, use `password_v

*   **Corrected Code (Example for Line 47 and general password handling):**

```php
// Registration (Hashing the password)
$password = $_GET['password']; //Password from input
$hashed_password = password_hash($password, PASSWORD_DEFAULT);

// Store $hashed_password in the database instead of the plaintext password.

// Login (Verifying the password)
// Retrieve the hashed password from the database.
$hashed_password_from_db = "..."; // Retrieve password from DB
```

~~~
$password = $_GET['password']; //Password from input
if (password_verify($password, $hashed_password_from_db)) {
    // Password is correct
    echo "Login successful!";
} else {
    // Password is incorrect
    echo "Login failed.";
}
```
~~~

*   **Best Practices:**

    *   **Never store passwords in plaintext.**
    *   **Use `password_hash()` and `password_verify()` for password management.
    *   **Use a strong hashing algorithm (bcrypt is recommended).**
    *   **Regularly review and update your cryptographic practices.**

**