Web shell upload via Content-Type restriction bypass

Name: Mduduzi Raadebe

Date: February 5, 2026

Lab Title: Web shell upload via Content-Type restriction bypass

Level: Apprentice

1. Objective

The primary goal was to bypass a server-side file upload restriction to gain **Remote Code Execution (RCE)**. Specifically, the objective was to upload a PHP web shell, execute it to read the contents of /home/carlos/secret, and exfiltrate that data to complete the lab.

2. Tools Used

- **Burp Suite Community Edition:** Used for intercepting and modifying HTTP requests.
- **Burp Browser:** The target-aware Chromium instance used to interact with the web application.
- **Notepad++:** Used for creating the PHP exploit script and modifying Burp configuration files.
- **Java Runtime Environment (JRE):** The underlying engine for Burp Suite, manually tuned for performance.

3. Methodology

1. **Environment Optimization:** Before testing, the Burp Suite environment was tuned for a low-resource machine. This included setting the memory limit to **768 MB** in the BurpSuiteCommunity.vmoptions file and disabling **Live Passive Crawl** to prevent system lag.
2. **Payload Creation:** A simple PHP script was authored to read a specific system file: <?php echo file_get_contents('/home/carlos/secret'); ?>.
3. **Initial Interception:** Logged in using the provided credentials (wiener:peter). Navigated to the avatar upload function and toggled **Intercept ON** in Burp Suite.
4. **Request Modification:** Attempted to upload exploit.php. The intercepted request showed the browser identifying the file as application/octet-stream.
5. **Bypass Execution:** Manually edited the **Content-Type** header from application/octet-stream to **image/jpeg** to trick the server's validation logic.
6. **Exfiltration:** Accessed the uploaded file directly via the browser. The server executed the PHP code, rendering Carlos's secret string in the response.

4. Findings & Vulnerabilities

Vulnerability	Category	Severity
CWE-434: Unrestricted Upload of File with Dangerous Type	Broken Access Control	**High**
Improper Validation of Content-Type Header	Input Validation	**Medium/High**

How it Worked

The application employed a "blacklist" or "trust-based" validation model. Instead of inspecting the file's extension or its actual content (magic bytes), it relied on the **Content-Type** header sent by the client. Because this header is entirely user-controllable, I was able to lie to the server, claiming a PHP script was actually a JPEG image.

5. Prevention & Remediation

To secure this function, the following measures should be implemented:

- **Magic Byte Validation:** The server should inspect the file's header (magic bytes) to verify its true type, rather than trusting the HTTP header.
- **Extension Whitelisting:** Implement a strict whitelist (e.g., only .jpg, .png, .gif) and reject everything else.
- **Filename Randomization:** Rename uploaded files to random strings (e.g., a8f3.jpg) to prevent attackers from predicting the execution path.
- **Disable Execution:** Ensure the directory where files are stored has execution permissions disabled (e.g., NoExec in Apache).

6. Lessons Learned & Reflection

My Struggles

- **System Latency:** The machine was initially unable to handle Burp's default background tasks, causing the browser to hang.
- **Connection Timeouts:** During the manual modification of the request, I encountered a "Stream failed to close correctly" error because the connection was held open too long while I was editing.

What I Would Do Differently

- **Automated Modification:** Instead of manually typing image/jpeg every time, I could use **Burp Match and Replace** rules to automatically swap the Content-Type whenever a .php file is detected.
- **Speed:** I learned that in intercept-heavy labs, you must be quick with your edits to prevent the browser or server from timing out.