

Threat Actor Support Line – Detailed Write-up

CTF: Huntress CTF 2025

Category: Misc / Web

Points: 10

Author: John Hammond

Challenge Overview

The challenge presents a satirical web application: the "**Threat Actor Support Line**", a fake portal where cybercriminals can supposedly get help encrypting victims' files.

The portal allows users to upload ZIP or RAR archives, claims to "encrypt" them, and then provides a link to download the processed files. The theme is tongue-in-cheek, but the core of the challenge hides a real **web exploitation bug**.

Our task: exploit the service to retrieve the flag from the underlying host.

Step 1 – Initial Reconnaissance

Visiting the root `/` in the browser (or via `curl`) shows an HTML page with:

- Upload form (`POST /` with field `file`)
- A fake "processing" status (JS messages like `"Starting up our hacking computer"`)
- A "Download Encrypted Files" button that becomes visible after upload

The **response headers** reveal the server is running:

```
Server: Werkzeug/2.3.7 Python/3.13.7
```

This confirms a **Flask/Werkzeug** stack.

Static assets are served from `/static/` :

```
curl http://10.1.125.223/static/script.js
curl http://10.1.125.223/static/style.css
```

The JavaScript simply handles the upload button, enforces a 1 MB size limit, and after a short delay submits the form. No client-side obfuscation of the download URL is done – the download logic is handled server-side.

Step 2 – Understanding the Workflow

1. Upload:

`POST /` with multipart form data (file field).

```
curl -F "file=@test.zip" http://10.1.125.223/
```

2. Processing:

The backend extracts the archive, "encrypts" files, and adds a ransom note.

3. Download:

A processed archive is exposed at `/download/`.

This `/download/` route is where things get interesting: it suggests the server is serving arbitrary files based on a path parameter.

Step 3 - First Exploitation Attempts

Initial attempts with standard path traversal patterns failed:

```
curl "http://10.1.125.223/download/../../../.././flag.txt"
curl "http://10.1.125.223/download/..%2f..%2f..%2fflag.txt"
```

Both returned 404.

Double-encoding was also tested:

```
curl "http://10.1.125.223/download/%252e%252e%252f%252e%252e%252f%252e%252e%252f%252e%252e%252f"
```

This produced a 302 redirect back to / with an error message.

Clearly, the app had some basic sanitization in place.

Step 4 - Filter Bypass

A well-known traversal bypass is to use `....//` instead of `../`.

Why this works:

- Many developers blacklist or regex-match only `../`.
- `....//` slips past because it does not literally equal `../`.
- After path normalization, `....//` collapses into `../`.

Payload:

```
curl -i -L "http://10.1.125.223/download/../../../../../../../../flag.txt"
```

Step 5 – Root Cause Analysis

The vulnerability lies in how Flask's route handler processed filenames.

- It likely used something like `send_from_directory("downloads", filename)` without proper sanitization.
- A naive check for `../` was bypassed using crafted strings (`....//` → normalized into `../`).
- This allowed us to escape the intended folder and access arbitrary files on disk.

Step 6 - Mitigation

To prevent this class of vulnerability:

1. **Strict whitelist** - only allow known good filenames, no slashes.
 2. **Canonicalize paths** - resolve absolute path (`os.path.realpath()`) and check it's still within the allowed directory.
 3. **Never rely on blacklist checks** - e.g. `replace("../", "")` or regexes are trivial to bypass.
 4. **Use secure helpers** like Flask's `safe_join()` (Werkzeug).
-

Key Takeaways

- Even joke-themed challenges hide real bugs.
- Always test for path traversal variants:
 - `../`
 - `%2e%2e/`
 - `%252e%252e/`
 - `....//`
 - Backslashes `%5c` (Windows)
- Proper sanitization requires whitelisting + path canonicalization.