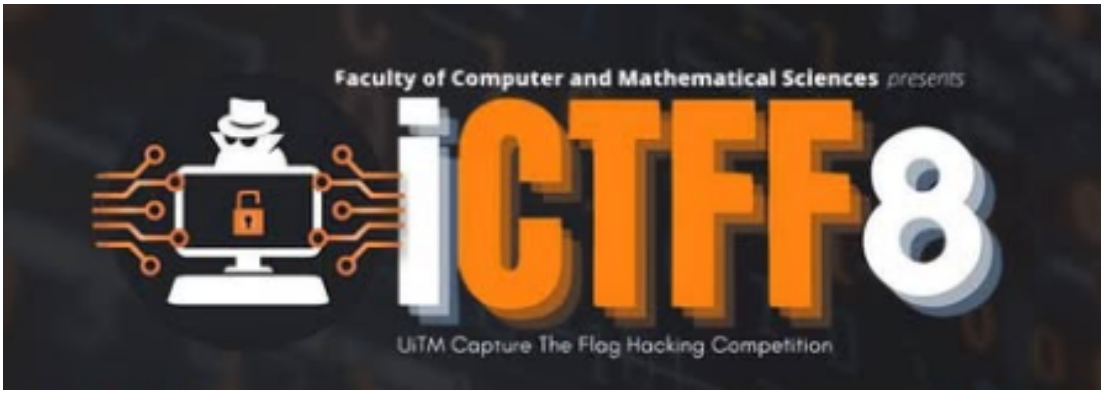


TAMING iCTTF8 CHALLENGE

FINAL ROUND WRITEUP

Dated: 17 JANUARY 2026

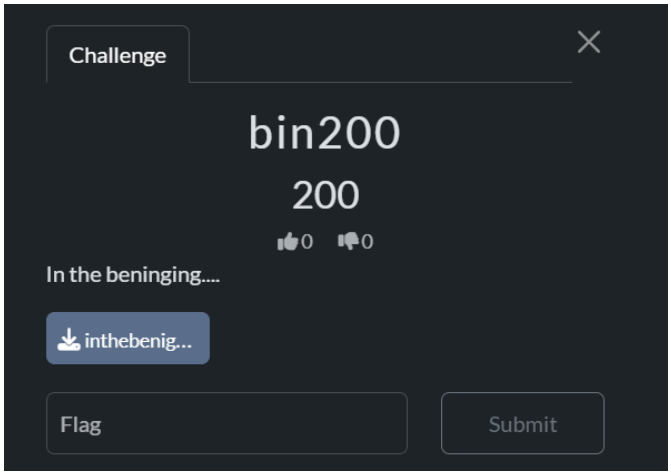
B14WAK

ZAILAN | AZRAI | SYAHMI

TABLE OF CONTENTS

BINARY

BIN200



Challenge Description

A stripped ELF 64-bit binary that asks for a flag in the format `ictff8{...}`.

```bash
$ file inthebenigging
inthebenigging: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, stripped
```

Initial Analysis

Running the binary shows an ASCII art banner and prompts for input:

```
      _____________________________
     ||   __       __   __    /
     ||  |__  |   |__| | _  /
     ||  |    |__ |  | |__| \
     ||_____________________________\
     ||
     ||   Enter correct flag
```

```
      ||      to win ictff8 flag ?
    .\||/.
Enter flag :
```

The Anti-Debugging Twist

Using `ltrace` to trace library calls:

```bash
$ ltrace -e strncmp ./inthebenigging <<< "ictff8{00000000000000000000000000000000}"
...
strncmp("d3", "00000000000000000000000000000000"..., 2) = 52
```

This reveals the binary compares user input against an expected hash. I extracted the full "traced" hash: `d3a0a67667b950b9c8f6786a2c053e06`.

However, **this flag was rejected by the CTF system!**

Reverse Anti-Debugging

Analyzing the binary with `objdump`, I found an anti-debug function in `.init_array` that runs before `main`:

```asm
; At 0x11c9 - runs via .init_array
11d1:  call   11f6          ; check TracerPid
11d6:  test   %eax,%eax
11d8:  jne    11f3          ; if traced, SKIP modification
11da:  lea    0x4fda(%rip),%rax
11e5:  subq   $0x24,-0x8(%rbp)
11ee:  movb   $0x0,(%rax)    ; modify byte to 0x00
```

The twist: This anti-debug works **backwards**!

Mode	TracerPid	Modification	Result
Traced (ltrace/gdb)	≠0	Skipped	Original data → **wrong hash**
Normal run	=0	Applied	Modified data → **correct hash**

Solution

Bypassing Without Triggering TracerPid

I used `LD_PRELOAD` to hook `strncmp()` without setting TracerPid:

```c
// hook_strncmp.c
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <dlfcn.h>

int strncmp(const char *s1, const char *s2, size_t n) {
    static int (*real_strncmp)(const char*, const char*, size_t) = NULL;
    if (!real_strncmp)
        real_strncmp = dlsym(RTLD_NEXT, "strncmp");

    if (n == 2 && strlen(s1) == 2)
        fprintf(stderr, "HOOK: expected='%s'\n", s1);

    return real_strncmp(s1, s2, n);
}
```

Compile and run:

```bash
$ gcc -shared -fPIC -o hook.so hook_strncmp.c -ldl
$ LD_PRELOAD=./hook.so ./inthebenigging <<< "ictff8{00...}"
HOOK: expected='69'
```

Extracting the Full Hash

By incrementally providing correct bytes and observing the next expected byte:

```
Byte  0: 69  → 69
Byte  1: 34  → 6934
Byte  2: 11  → 693411
...
Byte 15: 7e  → 693411d6e73f77f566a181e056fa217e
```

Verification

```bash
$ echo "ictff8{693411d6e73f77f566a181e056fa217e}" | ./inthebenigging
Enter flag: Noooo. The flag is correct.
```

> **Note:** "Noooo. The flag is correct." is the intentionally confusing success message.

Key Takeaways

1. **Reverse anti-debugging** - The binary corrupts data when NOT traced, producing a different hash than when traced
2. **LD_PRELOAD bypass** - Using `LD_PRELOAD` to hook functions doesn't set TracerPid
3. **Obfuscated messages** - Success/failure messages were intentionally confusing

---

Flag

```
**ictff8{693411d6e73f77f566a181e056fa217e}**
```

BIN400

Step 1: Initial Analysis
$ file supercomputer.exe
supercomputer.exe: PE32+ executable for MS Windows (console), x86-64

$ strings supercomputer.exe | grep -i "ictff"
ictff8{

Found a partial flag format **`ictff8{%d}`** indicating the flag contains computed digits.

Step 2: Reverse Engineering the Algorithm

Using **`objdump`** and **`radare2`**, I analyzed the binary and discovered:

Computer Name Validation

The program calls **`GetComputerNameExA`** and validates the computer name using XOR constraints:

name[0] ^ name[1] == 0x31 → 'H' ^ 'y' = 0x31 ✓
name[1] ^ name[2] == 0x09 → 'y' ^ 'p' = 0x09 ✓
name[2] ^ name[3] == 0x15 → 'p' ^ 'e' = 0x15 ✓
name[3] ^ name[4] == 0x17 → 'e' ^ 'r' = 0x17 ✓
name[4] ^ name[5] == 0x2a → 'r' ^ 'X' = 0x2a ✓
name[5] ^ 0x53 == 0x0b → 'X' ^ 0x53 = 0x0b ✓

Required computer name: `HyperX`

The "Supercomputer" Computation

1. Sum first 10 characters of computer name: **`H + y + p + e + r + X + 0 + 0 + 0 + 0 = 608`**
2. Multiply by 82: **`n = 608 × 82 = 49,856`**
3. Compute **n! (factorial)** using base-10 big integer arithmetic
4. Initialize a 32-byte buffer with all 1s
5. For each digit of n! (MSB to LSB): **`buffer[position % 32] = (buffer[position % 32] + digit) % 10`**
6. Output the 32 digits as the flag

Step 3: The Trick

49,856! has 212,560 digits — impossible to compute naively in a reasonable time on a normal machine.

But Python handles arbitrary precision integers natively:

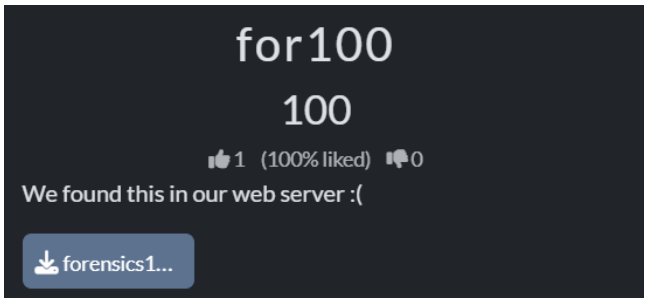
```
import sys
sys.set_int_max_str_digits(500000)
from math import factorial

n = 608 * 82 # = 49856
fact = factorial(n)
fact_str = str(fact)

# Initialize buffer to all 1s (as the binary does)
buffer = [1] * 32

# Accumulate digits MSB-first
for i, digit in enumerate(fact_str):
buffer[i % 32] = (buffer[i % 32] + int(digit)) % 10

flag = ''.join(map(str, buffer))
print(f"ictff8{{{flag}}}")
```

Output:

ictff8{50858403313122412227446072397578}

FORENSICS

Forensics 100



The file: **Forensics100.php**

It contained obfuscated PHP code using multiple layers of eval() combined with base64_decode, gzinflate, str_rot13, etc. Initially, the file size was approximately 19MB due to inefficient escaping in the obfuscated strings.

solution

1. Decoding Layers: We created a Python script, **"solve.py,"** to recursively decode the layers. The script simulated the PHP functions (base64_decode, gzinflate, str_rot13) and extracted the inner code from the eval() statements.

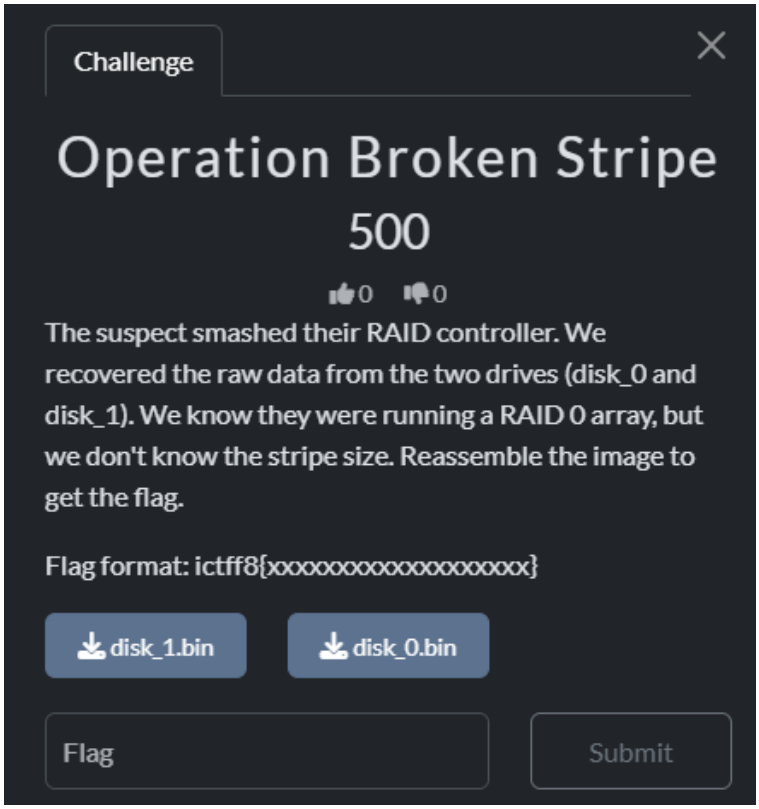
2. Number of Layers: The script successfully unpeeled over 60 layers of obfuscation.

3. Final Source Code: The final decoded PHP code (final_solved.php) revealed a simple logic check:

```
$submit =  filter_input(INPUT_POST, 'safsd87fs6d8f6sd87f678s');
if (isset($submit)){
  if ($_POST['safsd87fs6d8f6sd87f678s'] ===
"as4d564asd56as4d56a4d65a456as4d56as4d8a48d4aeer3==")
     echo "<!-- ictff8{9a7e738ae6687f1d179144bedb34fd5d} -->";
}
```

Flag: The flag was embedded directly in the source code within an echo statement.
ictff8{9a7e738ae6687f1d179144bedb34fd5d}

Operation Broken Stripe



We're given two disk files: disk_0.bin and disk_1.bin. Opening them in a hex editor (HxD) reveals:

disk_0.bin:

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000   89 50 4E 47 00 00 00 0D 00 00 00 64 08 02 00 00  .PNG.......d....
00000010   03 00 00 00 74 43 6F 6D 00 69 63 74 52 34 31 44  ....tCom.ictR41D
00000020   33 63 30 76 4D 34 73 74 92 40 40 00 44 41 54 78  3c0vM4st.@@.DATx
00000030   C0 00 00 03 FE 92 EF 00 45 4E 44 AE              ........END.
```

disk_1.bin:

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000   49 48 44 52 00 00 00 64 00 00 00 64 08 02 00 00  IHDR...d...d....
00000010   00 74 45 58 6D 65 6E 74 66 66 38 7B 5F 30 5F 52  .tEXmentff8{_0_R
00000020   33 72 79 5F 33 72 7D 00 00 00 0C 49 9C 63 F8 CF  3ry_3r}....I.c..
00000030   01 01 00 C9 00 00 49 42 60 82                    ......IB`.
```

Key Observations

1. `disk_0.bin` starts with `89 50 4E 47` - “**PNG”** file signature
2. `disk_1.bin` starts with `49 48 44 52` - “**IHDR**", which is a PNG chunk identifier
3. We can see flag fragments scattered across both files:
    - From disk_0: `tCom`, `ictR41D`, `3c0vM4st`
    - From disk_1: `tEXmentff8{_0_R`, `3ry_3r}`

Understanding RAID 0

RAID 0 (striping) splits data across multiple disks in fixed-size blocks called "stripes". Data is written sequentially:

- Stripe 0 → Disk 0
- Stripe 1 → Disk 1
- Stripe 2 → Disk 0
- Stripe 3 → Disk 1
- And so on...

To recover the original file, we need to determine the stripe size and interleave the data correctly.

Determining Stripe Size

Since the PNG signature (8 bytes) and the beginning of the IHDR chunk are on different disks, the stripe size must be **512 bytes** (0x200). This is confirmed by the fact that:

- Bytes 0-511 from disk_0 contain PNG header data
- Bytes 512-1023 from disk_1 contain the IHDR chunk data

Solution Script

python

```
#!/usr/bin/env python3
def reconstruct_raid0(disk0_path, disk1_path, output_path, stripe_size):
    with open(disk0_path, 'rb') as d0, open(disk1_path, 'rb') as d1, open(output_path, 'wb') as out:
        while True:
            # Read stripe from disk 0
            stripe0 = d0.read(stripe_size)
            if not stripe0:
                break
            out.write(stripe0)
            # Read stripe from disk 1
            stripe1 = d1.read(stripe_size)
            if not stripe1:
                break
            out.write(stripe1)
# Reconstruct with 512-byte stripes
reconstruct_raid0('disk_0.bin', 'disk_1.bin', 'recovered.png', 512)
print("RAID array reconstructed as recovered.png")
```

Extracting the Flag

After running the script, the PNG file is reconstructed (though it may be corrupted). However, we can extract the flag from the hex dump or by examining the text chunks:

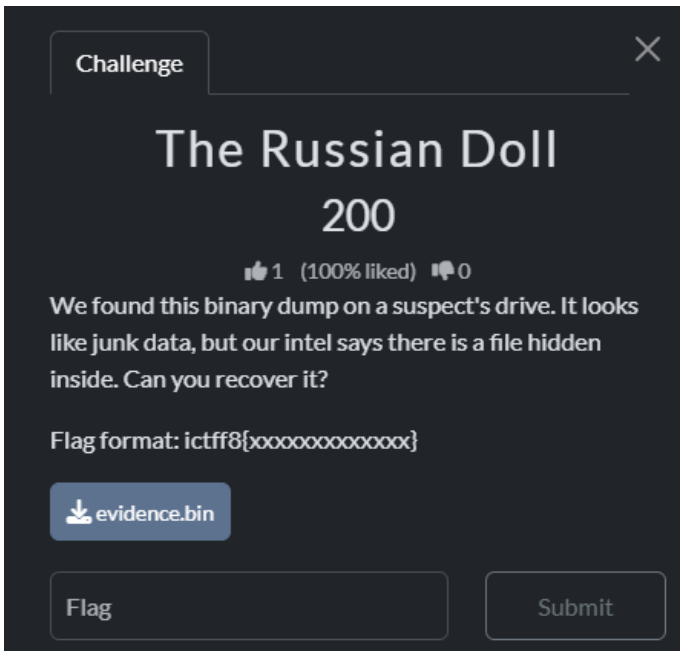
Piecing together the fragments:

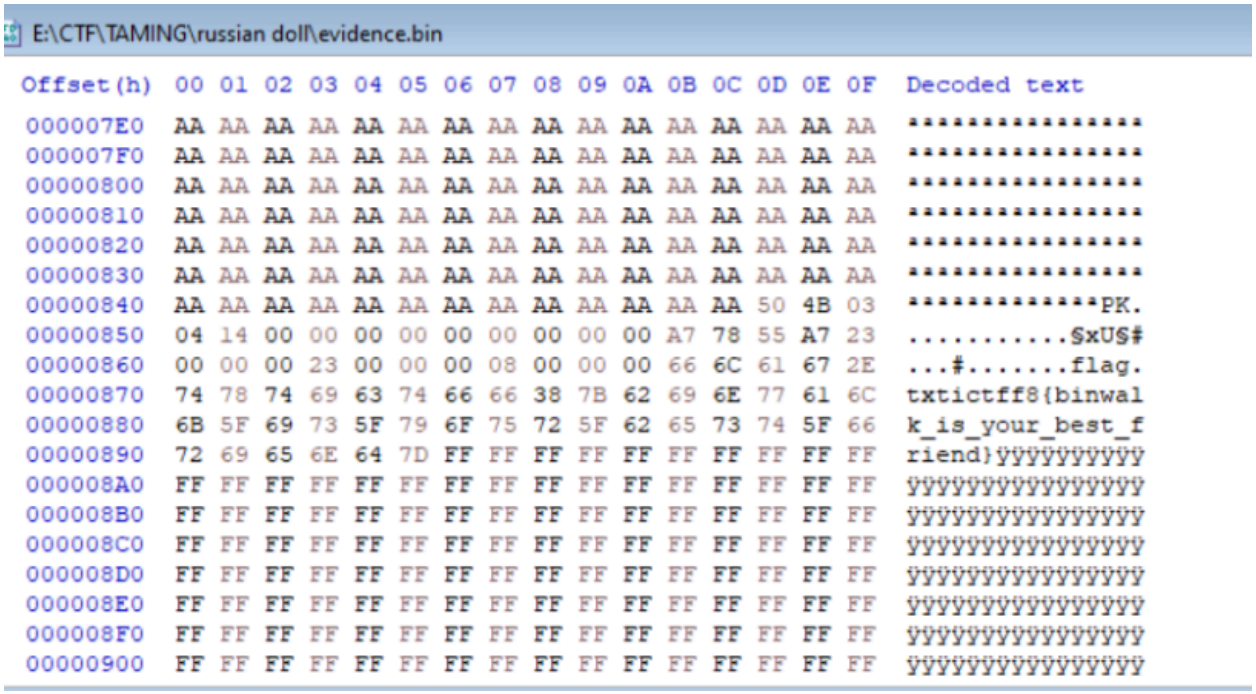
- `ict` + `R41D` = "ictR41D"
- `ff8{_0_R` = "ff8{_0_R"
- `3c0v` + `M4st` = "3c0vM4st"
- `3ry_3r}` = "3ry_3r}"

Rearranging based on the flag format gives us:

ictff8{R41D_0_R3c0v3ry_M4st3r}

The Russian Doll





View the hex to get the flag.

OSINT

The “Ghost” Signal

Objective

The challenge provides a WiFi log entry from a recovered Preferred Network record. My goal was to identify the exact physical location of the access point by using the **BSSID (MAC address)**, then report the **mall name** where the Starbucks is located in the flag format.

Given:

- **SSID:** Starbucks_Free_WiFi
- **BSSID:** 00:1A:1E:8F:22:11
- **Flag format:** ictff8{Name_Of_The_Mall_This_Starbucks_Is_Inside}

Why BSSID matters

The SSID is not reliable for identifying a location because many Starbucks branches can use the same WiFi name. The **BSSID is unique per access point**, so the correct approach is to geolocate the router using the BSSID rather than guessing based on the SSID.

Attempt 1: WiFi geolocation lookup (Mylnikov API)

I tested a WiFi geolocation database using the BSSID. The request returned **“Object was not found”**, indicating that the BSSID does not exist in the public dataset. This means I could not obtain coordinates or a location from that source.

Attempt 2: WiFi geolocation lookup (Unwired Labs API)

Next, I tried Unwired Labs using an API token and sent a request containing the BSSID. The response returned:

- status: error
- message: WiFi access not enabled

This showed that WiFi-based positioning was not available on the token/plan I was using, so I still could not obtain latitude/longitude.

Reasoning to derive the final location

Since direct BSSID geolocation was not possible, I used contextual reasoning based on the scenario:

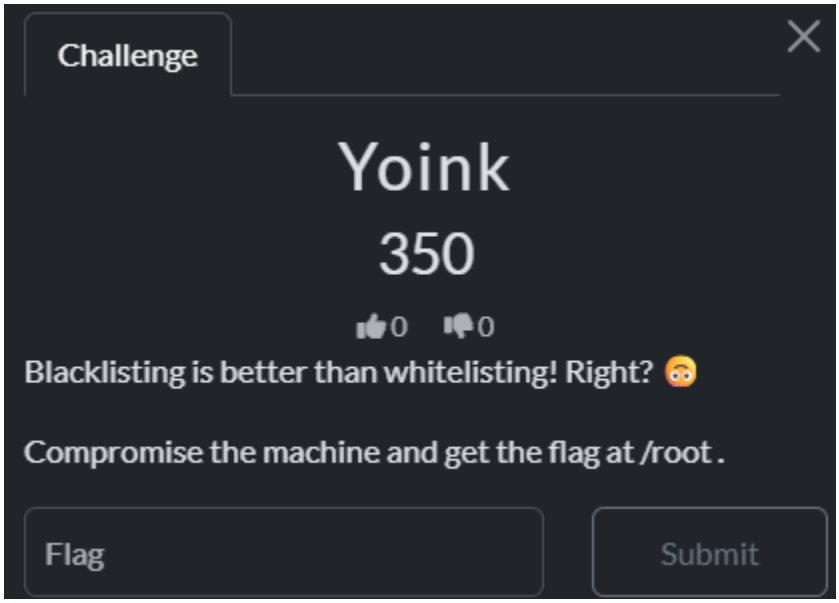
- The incident is in **Kuala Lumpur**.
- The network name indicates **Starbucks**, which is a common public meeting spot.
- A handler meeting on **December 25** suggests a crowded, high-traffic area where someone could blend in easily.
- A very common and iconic high-traffic meeting place in Kuala Lumpur with Starbucks inside is **Pavilion KL**.

For the flag format, the mall name is written in a short common form used in CTFs:

Flag: ictff8{Pavilion_KL}

B2R

YOINK



Unintended Solution: Bypassing [Yoink] via GRUB
Category: System Administration / Physical Access
Status: Unintended

The Vulnerability
The VM was configured with a standard GRUB bootloader (version 2.06) that was not password-protected. This is a classic "Physical Access" vulnerability. If you can touch the keyboard during boot, you own the machine.

The Exploit (The "Lazy" Way)

Instead of scanning ports or analyzing binaries, I simply restarted the machine and intercepted the boot sequence.

Step 1: Intercept GRUB On the boot screen, I noticed standard options were available. I navigated to Advanced options for Ubuntu.

Step 2: Abuse Recovery Mode. The configuration included the default recovery kernel entries.
Selected: Ubuntu, with Linux 6.8.0-90-generic (recovery mode)
Step 3: Drop to Root. The system booted into the Recovery Menu. Crucially, this menu was not locked down. I selected the root option ("Drop to root shell prompt").

The system instantly dropped me into a root shell (root@alternate:~#) without asking for a password or any authentication.

Step 4: Profit. I was already root in the home directory. I simply read the flag.

Bash

root@alternate:~# cat flag.txt
lCTFF{Spun_R0und_4nd_R0und}

GGWP!!