【UNCTF】逆向WriteUp以及出题笔记

原创

古月浪子 于 2020-11-16 10:20:10 发布

文章标签： CTF

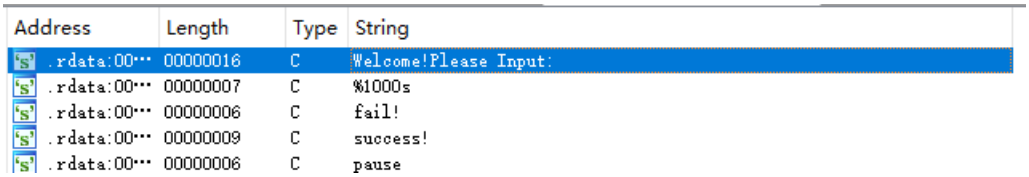
本文链接：https://blog.csdn.net/tqydyqt/article/details/109548103

【UNCTF】逆向WriteUp以及出题笔记

WriteUp

re_checkin

IDA打开，搜索字符串



Address	Length	Type	String
.rdata:00…	00000016	C	Welcome!Please Input:
.rdata:00…	00000007	C	%1000s
.rdata:00…	00000006	C	fail!
.rdata:00…	00000009	C	success!
.rdata:00…	00000006	C	pause

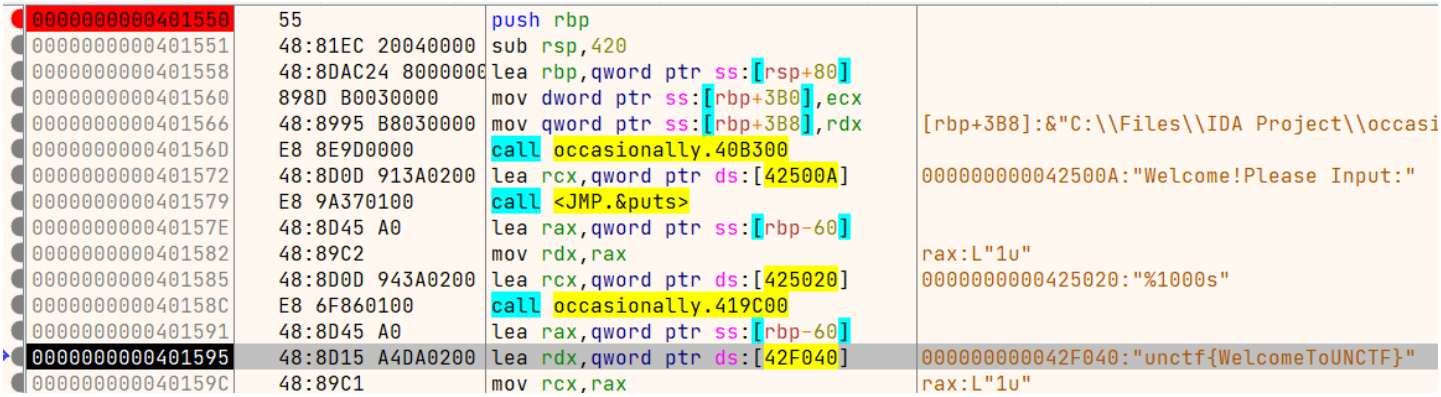
查看交叉引用，定位到主函数

```
__int64 sub_401550()
{
  char Str1; // [rsp+20h] [rbp-60h]

  sub_40B300();
  puts("Welcome!Please Input:");
  sub_419C00("%1000s");
  if ( !strcmp(&Str1, &Str2) )
    puts("success!");
  else
    puts("fail!");
  system("pause");
  return 0i64;
}
```

可以看到直接比较字符串

由于Str2在IDA中没有出现明文值，因此直接x64dbg打开，主函数下断，即可看到flag



反编译

这道题...有意思？？？

正规解题步骤参照下面的babypy，不过这道题据说运行就有flag？！

babypy

ExeinfoPe打开，发现是pyinstaller打包的，用网上的脚本解包

将解包出来的struct的前16字节添加到babypy文件头部，改名为babypy.pyc，用uncompyle6反编译，得到.py文件

```
import os, libnum, binascii
flag = 'unctf{******************}'
x = libnum.s2n(flag)

def gen(x):
    y = abs(x)
    while 1:
        if y > 0:
            yield y % 2
            y = y >> 1
    else:
        if x == 0:
            yield 0


l = [i for i in gen(x)]
l.reverse()
f = '%d' * len(l) % tuple(l)
a = binascii.b2a_hex(f.encode())
b = int(a, 16)
c = hex(b)[2:]
print(c)
os.system('pause')
```

再结合tip.txt，很容易写出逆向算法

```
flag = '3131313031303130313130313131303031313030303131303131313031303030313130303031313031303131313031313031303130
3130303031313031303030303130303030303030303131313031303030313031313131313031313031303031303131313031303130313031
3131313031313130303130303131303031303130313130303030313030313130303030313130303031303130313130303031313031303131
3131313031313030303130303131303030303030313130303030303030313130303030313031313131313031'
n = 0
for i in flag:
    if i == '1':
        n = (n << 1) + 1
    elif i == '0':
        n = n << 1
s = libnum.n2s(n)
print(s)
```

easyMaze

IDA打开，标准的迷宫题

```
__int64 sub_401550()
{
  bool v0; // al
  char v2; // [rsp+20h] [rbp-60h]
  char Dst; // [rsp+220h] [rbp+1A0h]
  __int16 v4; // [rsp+226h] [rbp+1A6h]

  sub_401AC0();
  memset(&Dst, 0, 0x200ui64);
  memset(&v2, 0, 0x200ui64);
  puts("Help Me Out!!!!!!!!");
  sub_40AE30("%200s", &Dst);
  v0 = sub_401757(&Dst) && sub_40161A(&v4);
  if ( v0 )
    puts("Yes! Escaped!");
  else
    puts("No way!");
  return 0i64;
}
```

检查输入格式

```
_BOOL8 __fastcall sub_401757(const char *a1)
{
  char *Str1; // [rsp+30h] [rbp+10h]

  Str1 = a1;
  return !strncmp(a1, "unctf{", 6ui64) && Str1[strlen(Str1) - 1] == '}';
}
```

输入wasd控制方向，Dst处是迷宫地图

```
  }
  if ( v4 < 0 || v3 < 0 || *(Dst + 10 * v3 + v4) == 68 || *(Dst + 10 * v3 + v4) == 48 )
    return 0i64;
  if ( v4 > 9 || v3 > 9 )
    return 0i64;
  if ( *(Dst + 10 * v3 + v4) == 83 )
    break;
  if ( sub_4019F4() )
  {
    puts("I See YOU!");
```

```
    exit(2);
  }
  ++v5;
}
```

根据交叉引用定位

```
char *sub_4017AA()
{
  _QWORD *v0; // rax
  char *result; // rax

  Dst = malloc(0x80ui64);
  memset(Dst, 0, 0x80ui64);
  v0 = Dst;
  *Dst = '00Do00o0';
  v0[1] = '0oooo0DS';
  v0[2] = '0D0ooooD';
  v0[3] = 'oo00o0Do';
  v0[4] = '00o00ooo';
  v0[5] = 'ooo0D0Do';
  v0[6] = 'o0o00ooo';
  v0[7] = 'ooDoo0o0';
  v0[8] = '0oDDDooo';
  v0[9] = 'oooo00o0';
  v0[10] = '0000D0Do';
  v0[11] = 'oooooDo';
  *(v0 + 24) = 'Dooo';
  result = Dst + 100;
  *(Dst + 100) = 0;
  return result;
}
```

迷宫是10x10的，这样看着不直观，而且是反的，倒过来换成每行10个稍微直观了一点

```
Oo00oD00SD
0oooo0Dooo
o0D0oD0o00
ooooo00o00
oD0D0ooooo
o00o0o0o0o
oDooooDDD
o00o00oooo
oD0D0000oD
ooooooooD
```

很容易看出来走出迷宫的路径：dsdddssaaaassssssddddddddwwaawawwddwwwdw

ICU

IDA打开，明文对比字符串

```
16  sub_40B3D0();
17  sub_49FDD0(&unk_4A6920, "This file was complied by MingW\n");
18  sub_49FDD0(&unk_4A6920, "enjoy\n");
19  sub_49FDD0(&unk_4A6920, "Please Input:\n");
20  v3 = sub_4A2860(0x10ui64);
21  sub_41F030(v3);
22  Memory = v3;
23  v4 = sub_4A2860(0x20ui64);
24  sub_4908B0(v4);
25  v13 = v4;
26  v10 = 0;
27  sub_4A0FA0(&unk_4A65C0, v4);
28  LODWORD(v4) = sub_42A820(v4);
29  v5 = sub_42A840(v13);
30  Str = sub_40180E(v5, v4);
31  v15 = 0;
32  v11 = strlen(Str);
33  while ( v15 < v11 )
34  {
35    sub_41EEC0(Memory, &v10);
36    Str[v15++] += v10;
37  }
38  if ( !memcmp(Str, "HSWEH2vXHmRtGZRJvSmKviwtviv4Ga5rD25Mvl:u6ewBUKg9", 0x30ui64) )
39    sub_49FDD0(&unk_4A6920, "You Win,but you don't enjoy it,right?\n");
40  else
41    sub_49FDD0(&unk_4A6920, "You lose,Try again\n");
42  v6 = sub_49FDD0(&unk_4A6920, Str);
43  sub_467C60(v6, sub_49D4F0);
44  system("pause");
```

看似好像是先处理一遍然后依次对单个字符进行处理，动态调试看看情况

```
0000000000401AEC    8802          mov byte ptr ds:[rdx],al          rdx:"35hqAP6m"
0000000000401AEE    8345 CC 01    add dword ptr ss:[rbp-34],1
0000000000401AF2  ^ EB BA         jmp icu.401AAE
0000000000401AF4    48:8B45 B0    mov rax,qword ptr ss:[rbp-50]     [rbp-50]:"35hqAP6m"
```

输入123456下断查看第一次处理的结果，6位变8位，看起来像base64，IDA搜索字符串可以看到base64的变表

单步即可看到第二次处理的结果

最终解密脚本

```python
import base64

table = 'UyOPef2ghvwx3ABdT7856QSijuCDFGst0LKER4ZabckHIJMNnopqrlmz1VWXY9+/'
origin = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'
t = 'HSWEH2vXHmRtGZRJvSmKviwtviv4Ga5rD25Mvl:u6ewBUKg9'
f = ''
odd = True
for i in t:
    if odd:
        f += chr(ord(i) - 1)
    else:
        f += i
    odd = not odd
flag = f.translate(str.maketrans(table, origin))
print(base64.b64decode(flag))
```

ezRust

通过试错可以发现需要2个额外的命令行参数

x64dbg打开，给2个假的命令行参数“1234”和“5678”，一路F8，遇到Error输出就F7，最后进入下面的函数

```
00007FF79DD521E0   55              push rbp
00007FF79DD521E1   48:81EC 70020000 sub rsp,270
00007FF79DD521E8   48:8DAC24 800000 lea rbp,qword ptr ss:[rsp+80]
00007FF79DD521F0   48:C785 E8010000 mov qword ptr ss:[rbp+1E8],FFFFFFF
00007FF79DD521FB   48:8D45 28       lea rax,qword ptr ss:[rbp+28]
00007FF79DD521FF   48:89C1          mov rcx,rax
00007FF79DD52202   48:8945 20       mov qword ptr ss:[rbp+20],rax
00007FF79DD52206   E8 05940000      call ezrust.7FF79DD5B610
00007FF79DD5220B   48:8B4D 20       mov rcx,qword ptr ss:[rbp+20]
00007FF79DD5220F   E8 7CA10000      call ezrust.7FF79DD5C390
00007FF79DD52214   48:8945 18       mov qword ptr ss:[rbp+18],rax
00007FF79DD52218   EB 00            jmp ezrust.7FF79DD5221A
00007FF79DD5221A   48:8D4D 28       lea rcx,qword ptr ss:[rbp+28]
00007FF79DD5221E   E8 DDF7FFFF      call ezrust.7FF79DD51A00
00007FF79DD52223   48:8B45 18       mov rax,qword ptr ss:[rbp+18]
00007FF79DD52227   48:83F8 03       cmp rax,3
00007FF79DD5222B   75 2E            jne ezrust.7FF79DD5225B
00007FF79DD5222D   48:8D85 90000000 lea rax,qword ptr ss:[rbp+90]
00007FF79DD52234   48:89C1          mov rcx,rax
00007FF79DD52237   48:8945 10       mov qword ptr ss:[rbp+10],rax
00007FF79DD5223B   E8 D0930000      call ezrust.7FF79DD5B610
00007FF79DD52240   48:8D4D 78       lea rcx,qword ptr ss:[rbp+78]     [rbp+78]:"main"
```

这一行会把第一个命令行和“YLBNB”入参，“1234”的情况下执行完call后rax等于0

```
00007FF79DD52391   48:8D15 88200200 lea rdx,qword ptr ds:[7FF79DD74420] rdx:&"YLBNB", 00007FF79DD74420:&"YLBNB"
00007FF79DD52398   48:8D8D B0000000 lea rcx,qword ptr ss:[rbp+B0]
00007FF79DD5239F   E8 1C500000      call ezrust.7FF79DD573C0
00007FF79DD523A4   8845 BF          mov byte ptr ss:[rbp-41],al
```

尝试把第一个命令行换成“YLBNB”，rax等于1

返回0会直接输出Error，返回1会进行第二次判断，同理得到第二个命令行为“RUSTPROGRAMING”

运行直接出flag（IDA也能找到对应的地方）

```
55    v8 = sub_1400073C0(&v13, &YLBNB);
56    if ( v8 )
57      v21 = sub_1400073C0(&v15, &RUSTPROGRAMING);
58    else
59      v21 = 0;
60    if ( v21 )
61    {
```

base_on_rust

（做题环境忘了保存，口头说一下做法）

这题我是动调出来的，但是我想站在上帝视角教大家一个简单的办法

程序需要一个额外的命令行输入，我们假设输入123456，会发现它会输出一段base64，将其解码，发现疑似base32，再解，发现是313233343536，正好是输入的字符的ascii码的十六进制拼起来

IDA的F5里面很多JUMPOUT，在汇编窗口手动修一下（没有解析的代码C一下，C不行的话就nop，再U整个函数，再P），可以看到疑似比较的操作，把待比较的数据dump下来，解码一下，跑脚本出flag

```
s = ''
f = '756E6374667B6261736536345F6261736533325F6261736531365F656E636F64655F5F5F7D'
i = 0
while i < len(f):
    c = f[i:i + 2]
    s += chr(int(c, 16))
    i += 2
print(s)
```

Trap

IDA打开，发现输入的flag s1和dest各留一份，在sub_400CBE后s1与s2比较

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  char v3; // r12
  FILE *s; // ST10_8
  FILE *v5; // rbx
  char *v6; // rax
  void (__fastcall *v7)(__int16 *, char *); // ST20_8
  signed int i; // [rsp+Ch] [rbp-A4h]
  void *handle; // [rsp+18h] [rbp-98h]
  char dest[48]; // [rsp+30h] [rbp-80h]
  __int16 v12; // [rsp+60h] [rbp-50h]
  __int16 v13; // [rsp+62h] [rbp-4Eh]
  int v14; // [rsp+64h] [rbp-4Ch]
  __int64 v15; // [rsp+68h] [rbp-48h]
  unsigned __int64 v16; // [rsp+98h] [rbp-18h]

  v16 = __readfsqword(0x28u);
  *dest = 0;
  *&dest[2] = 0;
  *&dest[4] = 0;
  memset(&dest[8], 0, 0x28uLL);
  puts("Welcome To UNCTF2020_RE_WORLD !!!");
  printf("Plz Input Key: ", a2);
  __isoc99_scanf("%s", s1);
  strcpy(dest, s1);
  sub_400CBE();
  if ( !strcmp(s1, s2) )
  {
    puts("Success.");
    for ( i = 0; i <= 8479; ++i )
    {
      v3 = ptr[i];
      ptr[i] = s1[i % strlen(s1)] ^ v3;
    }
    s = fopen("/tmp/libunctf.so", "wb");
    fwrite(ptr, 1uLL, 0x2120uLL, s);
```

跟进函数内部，异或后会开启线程

```
int sub_400CBE()
{
  int i; // [rsp+8h] [rbp-8h]
  int v2; // [rsp+Ch] [rbp-4h]

  v2 = strlen(s1);
  for ( i = 0; i < v2; ++i )
    s1[i] ^= 0x22u;
  pthread_create(&th, 0LL, start_routine, 0LL);
  return pthread_join(th, 0LL);
}
```

这个程序有很多地方不能正常反编译，手动修一下汇编就好了

比如说sub_400C13，一开始F5后看不懂，修好以后发现是个简单递归

```
unsigned __int64 __fastcall start_routine(void *a1)
{
```

```
unsigned __int64 result; // rax
int i; // [rsp+8h] [rbp-8h]
int v3; // [rsp+Ch] [rbp-4h]

sub_400BC0();
v3 = strlen(s1);
for ( i = 0; ; ++i )
{
  result = i;
  if ( i >= v3 )
    break;
  sub_400C13(&s1[i], v3);                       // s1[i]+=v3
}
return result;
}
```

跟进另一个函数，这个函数里面本来是有反调试的，被我nop掉了

```
__int64 sub_400BC0()
{
  __int64 result; // rax
  unsigned int i; // [rsp+8h] [rbp-8h]
  signed int v2; // [rsp+Ch] [rbp-4h]

  v2 = strlen(s2);
  for ( i = 0; ; ++i )
  {
    result = i;
    if ( i >= v2 )
      break;
    s2[i] ^= 0x33u;
  }
  return result;
}
```

对s1 s2的算法都很简单，很容易求出输入的是 941463c8-2bcb-

```
sub_400CBE();
if ( !strcmp(s1, s2) )
{
  puts("Success.");
  for ( i = 0; i <= 8479; ++i )
  {
    v3 = ptr[i];
    ptr[i] = s1[i % strlen(s1)] ^ v3;
  }
  s = fopen("/tmp/libunctf.so", "wb");
  fwrite(ptr, 1uLL, 0x2120uLL, s);
  getchar();
  handle = dlopen("/tmp/libunctf.so", 1);
  if ( !handle )
  {
    v5 = stderr;
    v6 = dlerror();
    fputs(v6, v5);
    exit(1);
  }
  v7 = dlsym(handle, "jo_enc");
  dlerror();
  v12 = 0;
```

```
    v13 = 0;
    v14 = 0;
    memset(&v15, 0, 0x28uLL);
    printf("plz Input Answer: ", "jo_enc", &v15);
    __isoc99_scanf("%s", &v12);
    v7(&v12, dest);                             // dest = 941463c8-2bcb-
  }
  else
  {
    puts("Loser!!!");
  }
  return 0LL;
}
```

运行的时候输入正确的flag会异或解密整个动态链接库文件，然后写入文件并调用jo_enc函数对接下来的输入进行检查

我的Ubuntu20不知道为什么调用动态库会失败，于是在/tmp中找到文件直接静态分析

```
__int64 __fastcall jo_enc(char *a1, char *a2)
{
  char *v2; // ST20_8
  size_t v3; // ST10_8
  int n; // [rsp+60h] [rbp-500h]
  int m; // [rsp+64h] [rbp-4FCh]
  int l; // [rsp+68h] [rbp-4F8h]
  int k; // [rsp+6Ch] [rbp-4F4h]
  int v9; // [rsp+70h] [rbp-4F0h]
  int j; // [rsp+74h] [rbp-4ECh]
  int v11; // [rsp+78h] [rbp-4E8h]
  signed int i; // [rsp+7Ch] [rbp-4E4h]
  int v13[48]; // [rsp+80h] [rbp-4E0h]
  int v14[128]; // [rsp+140h] [rbp-420h]
  int s[129]; // [rsp+340h] [rbp-220h]
  int v16; // [rsp+544h] [rbp-1Ch]
  char *v17; // [rsp+548h] [rbp-18h]
  char *v18; // [rsp+550h] [rbp-10h]

  v18 = a1;
  v17 = a2;
  v16 = 0;
  memset(s, 0, 0x200uLL);
  memset(v14, 0, 0x200uLL);
  memset(v13, 0, 0xC0uLL);
  for ( i = 0; i < 128; ++i )
  {
    s[i] = 2 * i;
    v14[i] = 2 * i + 1;
  }
  v11 = strlen(v18);
  for ( j = 0; j < v11; ++j )
  {
    v9 = v18[j];
    if ( !(v9 % 2) )
    {
```

```
    v14[i] = 2 * i + 1;
  }
  v11 = strlen(v18);
  for ( j = 0; j < v11; ++j )
  {
    v9 = v18[j];
    if ( !(v9 % 2) )
    {
```

```
    for ( k = 0; k < v9; k += 2 )
      v13[j] += s[k];
  }
  if ( v9 % 2 )
  {
    for ( l = 0; l < v9; l += 2 )
      v13[j] += v14[l];
  }
}
for ( m = 0; m < v11; ++m )
{
  v2 = v17;
  v3 = strlen(v17);
  v13[m] = (16 * v2[m % v3] & 0xE827490C | ~(16 * v2[m % v3]) & 0x17D8B6F3) ^ (v13[m] & 0xE827490C | ~v13[m] & 0x17D8B6F3);
}
for ( n = 0; n < v11; ++n )
{
  if ( v13[n] != *(off_200FD8 + n) )
  {
    v16 = 0;
    exit(1);
  }
  ++v16;
}
if ( v16 == 22 )
  puts("Win , Flag is unctf{input1+input2}");
return 0LL;
}
```

也是进行运算操作后和固定值比较，逆向脚本还算好写

```
cipher = [1668, 1646, 1856, 4118, 1899, 1752, 640, 2000, 4412, 1835, 820, 984, 968, 1189, 4353, 1646, 4348, 4561
, 1564,
          1566, 5596, 1525]
input1 = ''
input2 = '941463c8-2bcb-'
a = [i * 2 for i in range(128)]
b = [i * 2 + 1 for i in range(128)]
c = [16 * ord(input2[m % 14]) ^ cipher[m] for m in range(22)]
for d in c:
    for n in range(128):
        j = 0
        if not n % 2:
            for i in range(0, n, 2):
                j += a[i]
        else:
            for i in range(0, n, 2):
                j += b[i]
        if j == d:
            input1 += chr(n)
print('unctf{' + input2 + input1 + '}')
```

ezre

IDA打开，发现有混淆，根据字符串定位到主函数，flag有18位

```
  memset(v44, 0, 0x64u);
  v44[0] = 141;
  v44[1] = 99;
  v44[2] = 177;
  v44[3] = 201;
  v44[4] = 161;
  v44[5] = 205;
  v44[6] = 177;
  v44[7] = 177;
  v44[8] = 203;
  memset(v43, 0, sizeof(v43));
  v44[9] = 51;
```

```
v44[10] = 62;
v44[11] = 33;
v44[12] = 19;
v44[13] = 31;
v44[14] = 12;
v44[15] = 24;
v44[16] = 33;
v44[17] = 23;
memset(v45, 0, sizeof(v45));
sub_806FC80(1, "please input your flag:");
scanf("%s", v45);
if ( strlen(v45) != 18 )
{
  sub_804FB40("wrong!");
  exit(-1);
}
```

整个函数充满着混淆后的junk代码，自己调一调就能明白这些代码什么意思

```
v1 = v45[0];
v42 = 0;
v2 = 9;
while ( 1 )
{
  v3 = v45[v2];
  v4 = sub_8048EB0(-1, v1);                     // 这个函数其实就是*
  v5 = v4;
  v6 = v4;
  v7 = 128;
  do
  {
    v8 = v7 & v6;
    v6 = ~(v7 & v6) & (v7 | v6);
    v7 = 2 * v8;                                // 整个do while其实就是v6+=v7;v7=0
  }
  while ( 2 * v8 );
  v9 = v5;
  v10 = 128;
  if ( v3 < v6 )
  {
```

前面写了用来对比的数据，此处就是check的代码

```
v40 = 0;
do
{
  if ( sub_8048D70(v44[v40], v43[v40]) )        // 这个函数其实就是≠
  {
    sub_804FB40("you are not a hacker!");
    exit(-1);
  }
  ++v40;
}
while ( v40 != 18 );
sub_804FB40("flag is so good!");
```

2张截图之间的160多行代码就是具体的算法了，下面给出化简后的代码

```
v1 = v45[0];
v42 = 0;
v2 = 9;
while (1)
{
    v3 = v45[v2];
    if (v1 + v3 < 128)
    {
        v43[v42] = 128 - v3;
        v43[v42 + 9] = 2 * v3 + v1 - 128;
    }
    else
    {
        if (v1 + v3 == 128)
        {
            v43[v42] = v1;
            v43[v42 + 9] = v3;
        }
        else
        {
            v43[v42] = 2 * v1 + v3 - 128;
            v43[v42 + 9] = 128 - v1;
        }
    }
    if (++v42 == 9)
        break;
    v1 = v45[v42];
    v2 = v42 + 9;
}
```

用python写一个爆破脚本跑出flag

```
def chk(a, b):
    x = a + b
    if x < 128:
        y = 128 - b
        a, b = y, x - y
    elif x == 128:
        a = a
        b = b
    else:
        y = 128 - b
        a, b = x - y, y
    return a, b

flag = [141, 99, 177, 201, 161, 205, 177, 177, 203, 51, 62, 33, 19, 31, 12, 24, 33, 23]
flag_pre = ''
flag_post = ''
for i in range(0, 9):
    for j in range(0, 256):
        for k in range(0, 256):
            if chk(j, k) == (flag[i], flag[i + 9]):
                flag_pre += chr(k)
                flag_post += chr(j)
print(flag_pre + flag_post)
```

ezvm

IDA打开，就是一个VM

```
__int64 sub_403110()
{
  __int64 v0; // rax
  vm a1; // [rsp+20h] [rbp-28h]

  sub_4025B0();
  sub_401980();
  sub_4016E0(&a1);
  sub_4017D0(&a1);
  v0 = 0i64;
  while ( flag[v0] == *(a1.input + v0) )
  {
    if ( ++v0 == 0x15 )
    {
      puts("wuhu flag is what you input");
      return 0i64;
    }
  }
  puts("wrong! maybe you are not a hacker !");
  return 0i64;
}
```

一开始程序对VM进行了初始化

```
__int64 __fastcall sub_4016E0(vm *a1)
{
  vm *v1; // rbx
  _WORD *v2; // rax
  _WORD *v3; // rdi
  __int64 result; // rax

  v1 = a1;
  a1->field_0 = 0;
  a1->field_4 = 0;
  a1->field_8 = 0;
  a1->_rip = &unk_404080;
  v2 = malloc(0x512ui64);
  v1->input = v2;
  memset(v2, 0, 0x510ui64);
  v3 = v2 + 648;
  result = 0i64;
  *v3 = 0;
  return result;
}
```

根据初始化代码，自建一个结构体

```
00000000 vm              struc ; (sizeof=0x20, mappedto_44)
00000000                                       ; XREF: sub_403110/r
00000000 field_0         dd ?
00000004 field_4         dd ?
00000008 field_8         dd ?
0000000C field_C         dd ?
00000010 _rip            dq ?
00000018 input           dq ?                  ; XREF: sub_403110+24/r
00000020 vm              ends
```

分析每条指令作用的时候有几个地方需要注意

```
case 3:
  v8 = v2[1];
  v9 = v1->field_8;
  if ( v2[1] )
  {
    do
    {
      v10 = v8;
      v11 = v9 & v8;
      v12 = v9 ^ v10;
      v13 = 2 * v11 == 0;
      v8 = 2 * v11;
      v9 = v12;
    }
    while ( !v13 );
    v1->field_8 = v12;                  // v1->field_8 += v2[1]
  }
  else
  {
```

```
      v1->field_8 = v9;
    }
    goto LABEL_12;


case 8:
  v4 = v1->field_0;
  v5 = 3;
  v6 = 1i64;
  v7 = 7i64;
  do
  {
    if ( v7 & 1 )
      v6 = v4 * v6 - 187 * ((((631332417495941873 5i64 * (v4 * v6)) >> 64) >> 6) - (v4 * v6 >> 63));
    v7 >>= 1;
    --v5;
    v4 = v4 * v4 - 187 * ((((631332417495941873 5i64 * (v4 * v4)) >> 64) >> 6) - (v4 * v4 >> 63));
  }
  while ( v5 );
  ++v2;
  v1->field_0 = v6;
  v1->_rip = v2;
  result = *v2;
  if ( result == -7 )
    return result;
  goto LABEL_3;
default:
  puts("fxxx me ?");
  v2 = v1->_rip;
  break;
```

将opcode提取出来，并写出伪代码

```
-9 -15 6 3 -15 7 0 * -15 5 0 -8 -15 2 0 -16 20 -13 2 -12 11 -7

-7 return

-16 if [8]==next [0]=0 else [0]=([8]>=next)+1
 rip+=2

-15 if next==5 [0]=input[[8]]
 else if next==6 [4]=nnextt
 else if next==7 [8]=nnextt
 else if next==2 input[[8]]=[0]
 rip+=3

-13 [8]+=next

-12 if [0]==1 rip-=next else rip+=2

-9 input flag 21

-8 [0]???=[0]
 rip++

input flag 21
[4]=3
[8]=0
*
[0]=input[0]
[0]???=[0]
input[0]=[0]
[0]=1
[8]+=2
jmp * ([0]!=1 return)
```

最终解密脚本

```python
def func(v4):
    v5 = 3
    v6 = 1
    v7 = 7
    while True:
        if v7 & 1:
            v6 = v4 * v6 - 187 * ((((6313324174959418735 * v4 * v6) >> 64) >> 6) - (v4 * v6 >> 63))
        v7 >>= 1
        v5 -= 1
        v4 = v4 * v4 - 187 * ((((6313324174959418735 * v4 * v4) >> 64) >> 6) - (v4 * v4 >> 63))
        if v5 == 0:
            break
    return v6


flag = [150, 48, 144, 106, 159, 54, 39, 116, 179, 49, 157, 95, 142, 95, 17, 97, 157, 121, 39, 118, 131]
s = ''

for i in range(21):
    if i % 2 == 1:
        s += chr(flag[i])
    else:
        for j in range(128):
            if func(j) == flag[i]:
                s += chr(j)

print(s)
```

BetterCPU

不知道为什么，我这边运行不了这道题的附件，于是干脆纯静态分析

IDA打开，发现是VM

```
__int64 __cdecl main()
{
  ezvm *v0; // rbx

  _main();
  v0 = operator new(0x30ui64);
  ezvm::ezvm(v0);
  ezvm::run(v0);
  if ( v0 )
  {
    ezvm::~ezvm(v0);
    operator delete(v0);
  }
  return 0i64;
}
```

有符号表，很不错

```
void __cdecl ezvm::ezvm(ezvm *const this)
{
  __int64 v1; // [rsp+0h] [rbp-80h]
  unsigned __int64 enc_flag[45]; // [rsp+20h] [rbp-60h]
  ezvm *thisa; // [rsp+1A0h] [rbp+120h]

  thisa = this;
```

```
  this->memory = 0i64;
  this->stack = 0i64;
  this->rip = 0i64;
  this->rsp = 0i64;
  this->reg = 0i64;
  memcpy(&v1 + 4, &qword_409020, 0x168ui64);
  thisa->memory = operator new[](0x1000ui64);
  thisa->stack = operator new[](0x1000ui64);
  memset(thisa->stack, 0, 0x200ui64);
  memset(thisa->memory, 0, 0x200ui64);
  memcpy(thisa->memory + 336, enc_flag, 0x160ui64);
}
```

主要逻辑函数反编译不了，原因是jmp rax

```
void __cdecl ezvm::run(ezvm *const this)
{
  unsigned int v1; // eax
  ezvm *thisa; // [rsp+90h] [rbp+10h]

  thisa = this;
  v1 = ezvm::order[this->rip] - 14;
  if ( v1 <= 0x72 )
    JUMPOUT(__CS__, dword_40A020 + dword_40A020[v1]);
  puts("RUNTIME ERROR!");
  text_67("rip:    %lld\n", thisa->rip);
  exit(2);
}
```

很长很长的opcode

```
; unsigned __int8 ezvm::order[736]
_ZN4ezvm5orderE db 20h, 57h, 0, 0, 0, 0, 0, 0, 0, 22h
                                        ; DATA XREF: ezv
                                        ; ezvm::run(voi
                db 20h, 65h, 0, 0, 0, 0, 0, 0, 0, 22h
                db 20h, 31h, 0, 0, 0, 0, 0, 0, 0, 22h
                db 20h, 63h, 0, 0, 0, 0, 0, 0, 0, 22h
                db 20h, 30h, 0, 0, 0, 0, 0, 0, 0, 22h
                db 20h, 6Dh, 0, 0, 0, 0, 0, 0, 0, 22h
                db 20h, 65h, 0, 0, 0, 0, 0, 0, 0, 22h
```

用于jmp rax的表

```
; signed int dword_40A020[116]
dword_40A020    dd 0FFFF75F2h, 0FFFF7649h, 0FFFF76A0h, 0FFFF76E7h, 0FFFF7736h
                                        ; DATA XREF: ezvm::run(void)+38↑o
                                        ; ezvm::run(void)+44↑o
                dd 0FFFF7797h, 0Ch dup(0FFFF7C42h), 0FFFF77F0h, 0FFFF782Ah
                dd 0FFFF7852h, 0Dh dup(0FFFF7C42h), 0FFFF787Ah, 0FFFF7910h
                dd 0FFFF79A3h, 0FFFF7A4Fh, 0FFFF7B03h, 0Bh dup(0FFFF7C42h)
                dd 0FFFF7BB2h, 3Fh dup(0FFFF7C42h), 0FFFF7C74h, 0
```

我们先把可以跳到的地址找到，并且硬啃汇编，找到每个jmp的作用

```
0x401810 : mov_reg
 reg=*(rip+1)
 rip+=9
```

```
0x401c62 : put_error
 puts runtime error

0x401872 : putchar_reg
 putchar reg
 rip++

0x401930 : stack_sub
 rsp--
 *(stack+rsp-1)-=*(stack+rsp)
 rip++

0x40189a : stack_add
 rsp--
 *(stack+rsp-1)+=*(stack+rsp)
 rip++

0x4016c0 : push_reg
 rsp++
 *(stack+rsp)=reg
 rip++

0x401756 : pop_memory
 rsp--
 *(memory+reg)=*(stack+rsp)
 rip++

0x40184a : getchar_reg
 getchar
 reg=input
 rip++

0x4017b7 : push_memory
 rsp++
 *(stack+rsp)=*(memory+reg)
 rip++

0x401707 : pop_reg
 rsp--
 reg=*(stack+rsp)

0x401bd2 : stack_cmp
 rsp--
 a=*(stack+rsp)
 rsp--
 b=*(stack+rsp)
 this->?=a==b
 rip++

0x401669 : jmp_sub
 if this->?==0
  rip+=9
 else
  rip-=order[rip+1]

0x4019c3 : stack_xor
 rsp--
 b_1=*(stack+rsp)
 rsp--
 a_1=*(stack+rsp)
```

```
 a_1= (stack+rsp)
 rsp++
 *(stack+rsp)=a_1^b_1
 rip++

0x401612 : jmp_add
 if this->?==0
  rip+=9
 else
  rip+=order[rip+1]

0x401c94 : end
 return
```

然后写IDAPython脚本解读opcode

```
def getaddr(i):
    return -0x100000000+0x40a020+Dword(0x40a020+(Byte(i)-14)*4)

s=''
jmp=0
for i in range(0x4091a0,0x409480):
    if jmp>0:
        jmp-=1
        continue
    if Byte(i)==0x20:
        s+=str(i-0x4091a0)+' : mov_reg\n'+str(i-0x4091a0+1)+' : '+str(Qword(i+1))+' "'+chr(Qword(i+1)%256)+'" \n
'
        jmp=8
        continue
    if Byte(i)==0xf:
        s+=str(i-0x4091a0)+' : jmp_sub\n'+str(i-0x4091a0+1)+' : '+str(i-Qword(i+1)-0x4091a0)+'\n'
        jmp=8
        continue
    if Byte(i)==0xe:
        s+=str(i-0x4091a0)+' : jmp_add\n'+str(i-0x4091a0+1)+' : '+str(i+Qword(i+1)-0x4091a0)+'\n'
        jmp=8
        continue
    s+=str(i-0x4091a0)+' : '+str(hex(getaddr(i)))[:-1]+'\n'
s=s.replace('0x401c62','put_error')
s=s.replace('0x401872','putchar_reg')
s=s.replace('0x401930','stack_sub')
s=s.replace('0x40189a','stack_add')
s=s.replace('0x4016c0','push_reg')
s=s.replace('0x401756','pop_memory')
s=s.replace('0x40184a','getchar_reg')
s=s.replace('0x4017b7','push_memory')
s=s.replace('0x401707','pop_reg')
s=s.replace('0x401bd2','stack_cmp')
s=s.replace('0x4019c3','stack_xor')
s=s.replace('0x401c94','end')
f=open('op.txt','w')
f.write(s)
f.close()
```

可以得到伪代码

```
0 : mov_reg
1 : 87 "W"
9 : putchar_reg
```

```
10 : mov_reg
11 : 101 "e"
19 : putchar_reg
20 : mov_reg
21 : 49 "1"
29 : putchar_reg
30 : mov_reg
31 : 99 "c"
39 : putchar_reg
40 : mov_reg
41 : 48 "0"
49 : putchar_reg
50 : mov_reg
51 : 109 "m"
59 : putchar_reg
60 : mov_reg
61 : 101 "e"
69 : putchar_reg
70 : mov_reg
71 : 10 "
"
79 : putchar_reg
80 : mov_reg
81 : 80 "P"
89 : putchar_reg
90 : mov_reg
91 : 108 "l"
99 : putchar_reg
100 : mov_reg
101 : 101 "e"
109 : putchar_reg
110 : mov_reg
111 : 97 "a"
119 : putchar_reg
120 : mov_reg
121 : 115 "s"
129 : putchar_reg
130 : mov_reg
131 : 101 "e"
139 : putchar_reg
140 : mov_reg
141 : 32 " "
149 : putchar_reg
150 : mov_reg
151 : 73 "I"
159 : putchar_reg
160 : mov_reg
161 : 110 "n"
169 : putchar_reg
170 : mov_reg
171 : 80 "P"
179 : putchar_reg
180 : mov_reg
181 : 117 "u"
189 : putchar_reg
190 : mov_reg
191 : 116 "t"
199 : putchar_reg
200 : mov_reg
201 : 58 ":"
```

```
201 : 58 ":
209 : putchar_reg
210 : mov_reg
211 : 10 "
"
219 : putchar_reg
220 : mov_reg
221 : 0 " "
229 : push_reg
230 : pop_memory
231 : getchar_reg
232 : push_reg
233 : mov_reg
234 : 256 " "
242 : push_reg
243 : mov_reg
244 : 0 " "
252 : push_memory
253 : stack_add
254 : pop_reg
255 : pop_memory
256 : mov_reg
257 : 1 ""
265 : push_reg
266 : mov_reg
267 : 0 " "
275 : push_memory
276 : stack_add
277 : pop_memory
278 : mov_reg
279 : 0 " "
287 : push_memory
288 : mov_reg
289 : 44 ","
297 : push_reg
298 : stack_cmp
299 : jmp_sub
300 : 231
308 : mov_reg
309 : 0 " "
317 : push_reg
318 : pop_memory
319 : mov_reg
320 : 256 " "
328 : push_reg
329 : mov_reg
330 : 0 " "
338 : push_memory
339 : stack_add
340 : pop_reg
341 : push_memory
342 : mov_reg
343 : 102 "f"
351 : push_reg
352 : stack_add
353 : mov_reg
354 : 54 "6"
362 : push_reg
363 : stack_xor
364 : mov_reg
```

```
365 : 66 "B"
373 : push_reg
374 : stack_sub
375 : mov_reg
376 : 256 " "
384 : push_reg
385 : mov_reg
386 : 0 " "
394 : push_memory
395 : stack_add
396 : pop_reg
397 : pop_memory
398 : mov_reg
399 : 1 ""
407 : push_reg
408 : mov_reg
409 : 0 " "
417 : push_memory
418 : stack_add
419 : pop_memory
420 : mov_reg
421 : 0 " "
429 : push_memory
430 : mov_reg
431 : 44 ","
439 : push_reg
440 : stack_cmp
441 : jmp_sub
442 : 319
450 : mov_reg
451 : 0 " "
459 : push_reg
460 : pop_memory
461 : mov_reg
462 : 256 " "
470 : push_reg
471 : mov_reg
472 : 0 " "
480 : push_memory
481 : stack_add
482 : pop_reg
483 : push_memory
484 : mov_reg
485 : 336 "P"
493 : push_reg
494 : mov_reg
495 : 0 " "
503 : push_memory
504 : stack_add
505 : pop_reg
506 : push_memory
507 : stack_cmp
508 : jmp_add
509 : 660
517 : mov_reg
518 : 1 ""
526 : push_reg
527 : mov_reg
528 : 0 " "
```

```
536 : push_memory
537 : stack_add
538 : pop_memory
539 : mov_reg
540 : 0 " "
548 : push_memory
549 : mov_reg
550 : 44 ","
558 : push_reg
559 : stack_cmp
560 : jmp_sub
561 : 461
569 : mov_reg
570 : 83 "S"
578 : putchar_reg
579 : mov_reg
580 : 117 "u"
588 : putchar_reg
589 : mov_reg
590 : 99 "c"
598 : putchar_reg
599 : mov_reg
600 : 99 "c"
608 : putchar_reg
609 : mov_reg
610 : 101 "e"
618 : putchar_reg
619 : mov_reg
620 : 115 "s"
628 : putchar_reg
629 : mov_reg
630 : 115 "s"
638 : putchar_reg
639 : mov_reg
640 : 33 "!"
648 : putchar_reg
649 : mov_reg
650 : 10 "
"
658 : putchar_reg
659 : end
660 : mov_reg
661 : 70 "F"
669 : putchar_reg
670 : mov_reg
671 : 65 "A"
679 : putchar_reg
680 : mov_reg
681 : 73 "I"
689 : putchar_reg
690 : mov_reg
691 : 76 "L"
699 : putchar_reg
700 : mov_reg
701 : 10 "
"
709 : putchar_reg
710 : end
711 : 0x40a01f
712 : 0x40a01f
```

```
713 : 0x40a01f
714 : 0x40a01f
715 : 0x40a01f
716 : 0x40a01f
717 : 0x40a01f
718 : 0x40a01f
719 : 0x40a01f
720 : 0x40a01f
721 : 0x40a01f
722 : 0x40a01f
723 : 0x40a01f
724 : 0x40a01f
725 : 0x40a01f
726 : 0x40a01f
727 : 0x40a01f
728 : 0x40a01f
729 : 0x40a01f
730 : 0x40a01f
731 : 0x40a01f
732 : 0x40a01f
733 : 0x40a01f
734 : 0x40a01f
735 : 0x40a01f
```

如果读不懂可以再脑内解析一下

```
0:
Welcome...

220:
[0]=0

231:
push input
push 256
push [0]
add
[[0]+256]=input
push 1
push [0]
add
[0]=[0]+1
push [0]
push 44
cmp
jmp 231

308:
[0]=0

319:
push 256
push [0]
add
push [[0]+256]
push 102
add
push 54
```

```
xor
push 66
sub
push 256
push [0]
add
[0]=[0]+256
push 1
push [0]
add
[0]=[0]+1
push [0]
push 44
cmp
jmp 319

450:
[0]=0

461:
push 256
push [0]
add
push [[0]+256]
push 336
push [0]
add
push [[0]+336]
cmp
jmp 660

517:
push 1
push [0]
add
[0]=[0]+1
push [0]
push 44
cmp
jmp 461

569:
Success...
```

可以看到，读取44个字符，进行一系列操作，然后与memory+336比较

memory+336在虚拟机初始化的时候memcpy了一段数据

```
qword_409020    dq 0ABh, 0A0h, 0BDh, 0AAh, 0B8h, 95h, 4Ah, 56h, 57h, 4Dh
                                        ; DATA XREF: ezvm::ezvm(void)+65↑o
                dq 0B1h, 48h, 43h, 0B1h, 0B7h, 0ADh, 0B1h, 5Ch, 0BBh, 0AAh
                dq 0AAh, 0BBh, 0ACh, 0B1h, 4Ah, 0B6h, 0AFh, 0A0h, 0B1h, 47h
                dq 42h, 5Ch, 79h, 0ADh, 0B1h, 5Bh, 6Bh, 0B1h, 6Ch, 68h
                dq 6Bh, 5Eh, 93h, 4, 0, 0, 0, 0
```

写脚本提取出来，最终解密脚本

```
def ch(i):
    return chr((((i + 66) % 256 ^ 54) % 256 - 102) % 256)


flag = [171, 160, 189, 170, 184, 149, 74, 86, 87, 77, 177, 72, 67, 177, 183, 173, 177, 92, 187, 170, 170, 187, 1
72, 177,
        74, 182, 175, 160, 177, 71, 66, 92, 121, 173, 177, 91, 107, 177, 108, 104, 107, 94, 147, 4]

s = ''
for i in flag:
    s += ch(i)
print(s)
```

出题笔记

CTFilter

这道题呢，本来是想弄一个完整一点的透明加解密，后来因为太忙了，加上这是一个新生赛，于是搞了一个阉割版的

加密算法也很简单，只是异或

我感觉有点经验的赛棍可以直接盲猜出来答案

Hint.exe（flag.exe）的源码：

```
#include <stdio.h>

const char* s = "flag{Oh!You_found_me~}";

int main() {
 wprintf_s(L"Did you find my secret file? It's named flag.txt");
 getchar();
}
```

因为内核中使用的是UnicodeString，而等会内核会读取这里的字符，所以这里要用宽字符

CTFilter.sys头文件：

```
#include <fltKernel.h>

#define SystemProcessesAndThreadsInformation 5

typedef struct _SYSTEM_THREAD_INFORMATION {
 LARGE_INTEGER KernelTime;
 LARGE_INTEGER UserTime;
 LARGE_INTEGER CreateTime;
 ULONG         WaitTime;
 PVOID         StartAddress;
 CLIENT_ID     ClientId;
 ULONG64       Priority;
 LONG          BasePriority;
 ULONG         ContextSwitches;
 ULONG         ThreadState;
 ULONG         WaitReason;
 ULONG         PadPadAlignment;
} SYSTEM_THREAD_INFORMATION, * PSYSTEM_THREAD_INFORMATION;

typedef struct _SYSTEM_PROCESS_INFORMATION {
 ULONG                       NextEntryOffset;
 ULONG                       NumberOfThreads;
```

```
 LARGE_INTEGER            WorkingSetPrivateSize;
 ULONG                    HardFaultCount;
 ULONG                    NumberOfThreadsHighWatermark;
 ULONGLONG                CycleTime;
 LARGE_INTEGER            CreateTime;
 LARGE_INTEGER            UserTime;
 LARGE_INTEGER            KernelTime;
 UNICODE_STRING           ImageName;
 ULONG64                  BasePriority;
 HANDLE                   UniqueProcessId;
 HANDLE                   InheritedFromUniqueProcessId;
 ULONG                    HandleCount;
 ULONG                    SessionId;
 ULONG_PTR                UniqueProcessKey;
 SIZE_T                   PeakVirtualSize;
 SIZE_T                   VirtualSize;
 ULONG                    PageFaultCount;
 SIZE_T                   PeakWorkingSetSize;
 SIZE_T                   WorkingSetSize;
 SIZE_T                   QuotaPeakPagedPoolUsage;
 SIZE_T                   QuotaPagedPoolUsage;
 SIZE_T                   QuotaPeakNonPagedPoolUsage;
 SIZE_T                   QuotaNonPagedPoolUsage;
 SIZE_T                   PagefileUsage;
 SIZE_T                   PeakPagefileUsage;
 SIZE_T                   PrivatePageCount;
 LARGE_INTEGER            ReadOperationCount;
 LARGE_INTEGER            WriteOperationCount;
 LARGE_INTEGER            OtherOperationCount;
 LARGE_INTEGER            ReadTransferCount;
 LARGE_INTEGER            WriteTransferCount;
 LARGE_INTEGER            OtherTransferCount;
 SYSTEM_THREAD_INFORMATION Threads[1];
}SYSTEM_PROCESS_INFORMATION, * PSYSTEM_PROCESS_INFORMATION;

EXTERN_C_START
NTSTATUS ZwQuerySystemInformation(ULONG SystemInformationClass, PVOID SystemInformation, ULONG SystemInformation
Length, PULONG ReturnLength);
EXTERN_C_END

NTSTATUS CTFilterUnload(FLT_FILTER_UNLOAD_FLAGS);
FLT_PREOP_CALLBACK_STATUS PreWriteCallback(PFLT_CALLBACK_DATA, PCFLT_RELATED_OBJECTS, PVOID*);
BOOLEAN FindProcess();
BOOLEAN GetFileNameAndKey();
```

CTFilter.sys的主代码：

```
#include "CTFilter.h"

PWCHAR ProcessName;
HANDLE Pid;
PWCHAR FileName;
PUCHAR Key;
PFLT_FILTER FilterHandle;

CONST FLT_OPERATION_REGISTRATION Callbacks[] = {
 { IRP_MJ_WRITE, 0, PreWriteCallback, NULL },
 { IRP_MJ_OPERATION_END }
};
```

```
CONST FLT_REGISTRATION FilterRegistration = {
 sizeof(FLT_REGISTRATION), FLT_REGISTRATION_VERSION, 0, NULL, Callbacks, CTFilterUnload,
 NULL, NULL, NULL, NULL, NULL, NULL, NULL
};

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
 NTSTATUS status;
 ProcessName = ExAllocatePool(PagedPool, 0x12);
 memcpy(ProcessName, "\"D(D%D#DjD!D<D!DDD", 0x12);
 if (!FindProcess()) {
  ExFreePool(ProcessName);
  return STATUS_UNSUCCESSFUL;
 }
 if (!GetFileNameAndKey()) {
  ExFreePool(ProcessName);
  return STATUS_UNSUCCESSFUL;
 }
 status = FltRegisterFilter(DriverObject, &FilterRegistration, &FilterHandle);
 if (!NT_SUCCESS(status)) {
  ExFreePool(Key);
  ExFreePool(FileName);
  ExFreePool(ProcessName);
  return status;
 }
 status = FltStartFiltering(FilterHandle);
 if (!NT_SUCCESS(status))
  FltUnregisterFilter(FilterHandle);
 return status;
}

NTSTATUS CTFilterUnload(FLT_FILTER_UNLOAD_FLAGS Flags) {
 FltUnregisterFilter(FilterHandle);
 ExFreePool(Key);
 ExFreePool(FileName);
 ExFreePool(ProcessName);
 return STATUS_SUCCESS;
}

FLT_PREOP_CALLBACK_STATUS PreWriteCallback(PFLT_CALLBACK_DATA Data, PCFLT_RELATED_OBJECTS FltObjects, PVOID* Com
pletionContext) {
 NTSTATUS status;
 PFLT_FILE_NAME_INFORMATION info;
 ULONG len = Data->Iopb->Parameters.Read.Length;
 if (!Data)
  return FLT_PREOP_SUCCESS_NO_CALLBACK;
 status = FltGetFileNameInformation(Data, FLT_FILE_NAME_NORMALIZED | FLT_FILE_NAME_QUERY_DEFAULT, &info);
 if (!NT_SUCCESS(status))
  return FLT_PREOP_SUCCESS_NO_CALLBACK;
 status = FltParseFileNameInformation(info);
 if (!NT_SUCCESS(status)) {
  FltReleaseFileNameInformation(info);
  return FLT_PREOP_SUCCESS_NO_CALLBACK;
 }
 if (info->Name.Buffer && !wcscmp(wcsrchr(info->Name.Buffer, L'\\') + 1, FileName)) {
  if (Data->Iopb->Parameters.Write.WriteBuffer) {
   PUCHAR buffer = (PUCHAR)Data->Iopb->Parameters.Write.WriteBuffer;
   for (int i = 0; i < Data->Iopb->Parameters.Write.Length; ++i)
    buffer[i] ^= Key[i % 0x16];
  }
```

```
 }
 FltReleaseFileNameInformation(info);
 return FLT_PREOP_SUCCESS_WITH_CALLBACK;
}

BOOLEAN FindProcess() {
 ULONG size = 0;
 PVOID buffer;
 PSYSTEM_PROCESS_INFORMATION info;
 ULONG count = 0;
 for (PCHAR p = (PCHAR)ProcessName; p < ProcessName + 0x9; ++p)
  *p ^= 0x44;
 ZwQuerySystemInformation(SystemProcessesAndThreadsInformation, NULL, 0, &size);
 buffer = ExAllocatePool(PagedPool, size);
 ZwQuerySystemInformation(SystemProcessesAndThreadsInformation, buffer, size, NULL);
 info = (PSYSTEM_PROCESS_INFORMATION)buffer;
 do {
  if (info->ImageName.Buffer && !wcscmp(info->ImageName.Buffer, ProcessName)) {
   Pid = info->UniqueProcessId;
   ++count;
  }
  info = (PSYSTEM_PROCESS_INFORMATION)((ULONG64)info + info->NextEntryOffset);
 } while (info->NextEntryOffset);
 ExFreePool(buffer);
 return count == 1;
}

BOOLEAN GetFileNameAndKey() {
 NTSTATUS status;
 PEPROCESS process;
 KAPC_STATE apc;
 status = PsLookupProcessByProcessId(Pid, &process);
 if (!NT_SUCCESS(status))
  return FALSE;
 KeStackAttachProcess(process, &apc);
 FileName = ExAllocatePool(PagedPool, 0x12);
 memcpy(FileName, (PVOID)0x1400022B0, 0x12);
 Key = ExAllocatePool(PagedPool, 0x16);
 memcpy(Key, (PVOID)0x140002240, 0x16);
 KeUnstackDetachProcess(&apc);
 ObDereferenceObject(process);
 return TRUE;
}
```

主要逻辑就是，加载驱动后会查询有没有flag.exe这个进程，有的话就读取它的假flag和flag.txt到内存中

每当flag.txt被写入时，将写入的数据循环异或假flag

据此我们可以推断，没开地址随机化的Hint.exe的对应地址刚好有相符的数据，可以认定它被改名了，原名应该是flag.exe

而Unknown_data应该是flag.txt，将里面的内容再循环异或一次假flag即可得到真flag

> 这道题主要的难点应该就是不能动态调试
> 对于从来没有接触过内核驱动的人来说，加载驱动是一个难点，调试驱动又是一个难点
> 可以尝试纯静态分析

原神

首先，立正挨打

这道题把主办方的CPU跑满了，导致被迫临时下线。后来我们自己部署了一个静态的环境，才稍微好一点（然而机子还是炸了好几次）

源代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <time.h>
#include <unistd.h>

const char* ch4[] = { "丽莎", "凯亚", "安柏", "香菱", "砂糖", "诺艾尔", "凝光", "北斗", "菲谢尔", "芭芭拉", "重云", "班尼特", "行秋", "雷泽" };
const char* ch5[] = { "刻晴", "七七", "琴", "温迪", "莫娜", "迪卢克" };
const char* wp3[] = { "弹弓", "鸦羽弓", "讨龙英杰谭", "黑缨枪", "沐浴龙血的剑", "飞天御剑", "冷刃", "神射手之誓", "翡玉法球", "魔导绪论", "以理服人", "铁影阔剑", "黎明神剑" };
const char* wp4[] = { "祭礼弓", "西风猎弓", "祭礼残章", "西风秘典", "匣里灭辰", "祭礼大剑", "西风大剑", "祭礼剑", "西风剑", "弓藏", "绝弦", "昭心", "流浪乐章", "西风长剑", "雨裁", "钟剑", "匣里龙吟", "笛剑" };
const char* wp5[] = { "天空之翼", "天空之卷", "天空之脊", "天空之傲", "风鹰剑", "阿莫斯之弓", "四风原典", "和璞鸢", "狼的末路", "天空之刃" };

int ch4_n, ch5_n, wp3_n, wp4_n, wp5_n;

void ck(int n) {
 int r;
 puts("抽卡结果如下：");
 for (int i = 0; i < n; ++i) {
  r = rand() % 1000;
  if (r < 3) {
   r = rand() % 6;
   printf("★★★★★ %s\n", ch5[r]);
   ++ch5_n;
  }
  else if (r < 6) {
   r = rand() % 10;
   printf("★★★★★ %s\n", wp5[r]);
   ++wp5_n;
  }
  else if (r < 106) {
   r = rand() % 14;
   printf("★★★★ %s\n", ch4[r]);
   ++ch4_n;
  }
  else if (r < 206) {
   r = rand() % 18;
   printf("★★★★ %s\n", wp4[r]);
   ++wp4_n;
  }
  else {
   r = rand() % 13;
   printf("★★★ %s\n", wp3[r]);
   ++wp3_n;
  }
 }
}

int main() {
 int n;
```

```
int n;
char name[32];
setbuf(stdin, 0);
setbuf(stdout, 0);
setbuf(stderr, 0);
memset(name, 0, 32);
srand(time(0));
puts("欢迎使用原神抽卡模拟器！祝你好运~");
while (1) {
 puts("请选择：[1]单抽 [2]十连 [3]结束抽卡");
 scanf("%d", &n);
 if (n == 1)
  ck(1);
 else if (n == 2)
  ck(10);
 else if (n == 3)
  break;
 else
  puts("错误的选择！");
}
printf("恭喜你，一共抽到了%d个4星角色、%d个5星角色、%d个3星武器、%d个4星武器、%d个5星武器！\n请选择：[1]向好友炫耀 [2]退出\n", ch4_n, ch5_n, wp3_n, wp4_n, wp5_n);
scanf("%d", &n);
if (n == 1) {
 puts("请输入你的名字：");
 read(0, name, 88);
 puts("分享成功^_^");
}
system("echo Bye~!");
close(1);
}
```

基础的栈溢出题，可以很明显的看到程序可以进行ROP

但由于程序本身没有/bin/sh，所以不能直接打system

因为最后关闭了标准输出流，因此leak不太现实

可以想到的是，int类型的 26739 和字符串类型的 “sh” 的二进制值是一模一样的，3星武器的值是最容易增长的，于是我们可以抽这么多的3星武器，然后把记录3星武器数量的变量的地址当作参数传入system即可拿到shell

之后由于标准输出流被关闭了，但是错误输出流没有被关闭，因此通过 cat flag 1>&2 重定向输出流从而读取flag

（不知道有没有非预期，逃ε=ε=ε= ┏(゜ロ ゜;)┛ ）

> 解题过程中用到的都是基础知识点，特别适合萌新
> 数据类型混淆是一个考点，要学会巧妙的构造可以利用的点
> 对于能拿到shell的人，我觉得最后cat flag应该不是问题
> 最后，希望大家都能在原神里单抽抽到自己想要的角色~