REVERSE-PRACTICE-BUUCTF-17

原创

P1umH0 于 2021-02-27 00:10:34 发布 87 收藏

分类专栏： Reverse-BUUCTF

本文链接：https://blog.csdn.net/weixin_45582916/article/details/114155871

REVERSE-PRACTICE-BUUCTF-17

[网鼎杯 2020 青龙组]jocker

exe程序，运行后提示输入flag，无壳，用ida分析

main函数平衡栈后，F5反汇编

主要逻辑为

读取输入，检验输入长度是否为24，对输入进行变换，与已知数组比较，执行一段从encrypt函数起始地址开始的SMC，分析可知，finally函数也会完全被SMC，未经变换的输入作为参数，调用encrypt函数和finally函数

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char input; // [esp+12h] [ebp-96h]
  char input_copy; // [esp+44h] [ebp-64h]
  DWORD flOldProtect; // [esp+94h] [ebp-14h]
  size_t v7; // [esp+98h] [ebp-10h]
  int i; // [esp+9Ch] [ebp-Ch]

  __main();
  puts("please input you flag:");
  if ( VirtualProtect(encrypt, 0xC8u, 4u, &flOldProtect) == 0 )
    exit(1);
  scanf("%40s", &input);
  v7 = strlen(&input);
  if ( v7 != 24 )                               // input的长度必须为24
  {
    puts("Wrong!");
    exit(0);
  }
  strcpy(&input_copy, &input);
  wrong(&input);                                // 对input进行变换，下标为奇数，input减去下标，下标为偶数，input与下标异或
  omg(&input);                                  // 与已知数组比较，验证input
  for ( i = 0; i <= 186; ++i )                  // 从encrypt函数起始地址开始的SMC，分析可知，finally函数也会完全被SMC
    *((_BYTE *)encrypt + i) ^= 0x41u;
  if ( encrypt(&input_copy) != 0 )              // 未经变换的input作为参数，调用encrypt函数
    finally(&input_copy);
  return 0;
}
```


先写input经过wrong变换和omg验证的逆运算脚本，得到的是假flag

```
unk_4030C0=[0x66, 0x6b, 0x63, 0x64, 0x7f, 0x61, 0x67, 0x64,
            0x3b, 0x56, 0x6b, 0x61, 0x7b, 0x26, 0x3b, 0x50,
            0x63, 0x5f, 0x4d, 0x5a, 0x71, 0xc, 0x37, 0x66]
s=""
for i in range(len(unk_4030C0)):
    if i%2==1:
        s+=chr(unk_4030C0[i]+i)
    else:
        s+=chr(unk_4030C0[i]^i)
print(s)
```

```
test2
D:\python27-x64\python2.exe D:/Python/pycharm/pycfile/
flag{fak3_alw35_sp_me!!}
```


往下走，在ida中使用idapython脚本完成SMC自修改代码

```
from idaapi import *
from idautils import *
start_addr = 0x401500
key = 0x41
for i in range(start_addr,start_addr+187):
    PatchByte(i,Byte(i)^key)
```

SMC前，encrypt函数和finally函数各自包含了一大段数据

```
.text:00401531 -8452                   retf    4904h
.text:00401531     __Z7encryptPc       endp ; sp-analysis failed
.text:00401531
.text:00401531     ; ---------------------------------------------------------------------------
.text:00401534                         dd 0F74E9140h, 0A504CA51h, 1015344h, 41F74E41h, 0FF4E9170h
.text:00401534                         dd 0A504CA91h, 0D5C405CAh, 5E358378h, 41654586h, 0A9410101h
.text:00401534                         dd 414152A1h, 41A10486h, 86414141h, 41416545h, 0ECA94141h
.text:00401534                         dd 0C2414152h, 0C240A504h, 3F53A53Ch, 0A13CC2F0h, 864D3440h
.text:00401534                         dd 1496545h, 0F0A94101h, 0CA414152h, 85C2A104h, 1E1F1A3Dh
.text:00401598     ; ---------------------------------------------------------------------------
.text:00401598                         sbb     al, 82h
.text:0040159A
.text:0040159A     ; =============== S U B R O U T I N E =======================================
.text:0040159A
.text:0040159A
.text:0040159A     ; _DWORD __cdecl finally(char *)
.text:0040159A                         public __Z7finallyPc
.text:0040159A     __Z7finallyPc       proc near               ; CODE XREF: _main+121↓p
.text:0040159A 000                     adc     al, 0C8h
.text:0040159C 000                     movsb
.text:0040159D 000                     retn    69ADh
.text:0040159D     __Z7finallyPc       endp
.text:0040159D
.text:0040159D     ; ---------------------------------------------------------------------------
.text:004015A0                         dd 64AA0487h, 35AD0487h, 31AC0487h, 67AF0487h, 7BAE0487h
.text:004015A0                         dd 41654586h, 0E8414141h, 1368h, 0E8240489h, 1368h, 136BE8h
.text:004015A0                         dd 0BAC18900h, 51EB851Fh, 0EAF7C889h, 8905FAC1h, 1FF8C1C8h
.text:004015A0                         dd 0D089C229h, 8BF44589h, 0C06BF445h, 89C12964h, 0F44589C8h
.text:004015A0                         dd 0F045C7h, 83000000h, 7F04F07Dh, 0EB558D3Eh, 1F0458Bh
.text:004015A0                         dd 10B60FD0h, 8BF04D8Bh, 0C8010845h, 3800B60Fh, 0C0950FC2h
.text:004015A0                         dd 3BC0B60Fh, 0E74F445h, 2C2404C7h, 0E8004040h, 1310h
.text:004015A0                         dd 4C70DEBh, 40405C24h, 1302E800h, 0C9900000h
.text:00401640     ; ---------------------------------------------------------------------------
.text:00401640                         retn
.text:00401641
```


SMC后，encrypt函数和finally函数的指令间都有一些db指令，实际上为花指令

encrypt函数：

```
.text:00401500
.text:00401500
.text:00401500     ; _DWORD __cdecl encrypt(char *)
.text:00401500                         public __Z7encryptPc
.text:00401500     __Z7encryptPc       proc far                ; CODE XREF: _main+10D↓p
.text:00401500                                                 ; DATA XREF: _main+3C↓o ...
.text:00401500 000                     push    ebp
.text:00401500     ; ---------------------------------------------------------------------------
.text:00401501 000                     db  89h
.text:00401502 000                     in      eax, 57h
.text:00401504 000                     push    esi
.text:00401505 004                     push    ebx
.text:00401505     ; ---------------------------------------------------------------------------
.text:00401506 008                     db  83h
.text:00401507     ; ---------------------------------------------------------------------------
.text:00401507 008                     in      al, dx
.text:00401508 008                     jl      short near ptr loc_4014CD+4
.text:0040150A 008                     inc     ebp
.text:0040150B 008                     loopne  near ptr unk_40150E
.text:0040150B     ; ---------------------------------------------------------------------------
.text:0040150D 008                     db    0
.text:0040150E 008 unk_40150E          db    0                 ; CODE XREF: encrypt(char *)+B↑j
.text:0040150F 008                     db    0
.text:00401510 008                     db  8Dh
.text:00401511 008                     db  45h ; E
.text:00401512     ; ---------------------------------------------------------------------------
```

```
.text:00401512 008                xchg    eax, esp
.text:00401513 008                mov     ebx, offset unk_403040
.text:00401518 008                mov     edx, 13h
.text:0040151D 00C                mov     edi, eax
.text:0040151F 00C                mov     esi, ebx
.text:00401521 -36EE              mov     ecx, edx
.text:00401523 -36EE              rep movsd
.text:00401525 -844A              mov     dword ptr [ebp-1Ch], 0
.text:0040152C -844A              jmp     short loc_401577
```


finally函数：

```
.text:0040159A
.text:0040159A     ; _DWORD __cdecl finally(char *)
.text:0040159A                    public __Z7finallyPc
.text:0040159A     __Z7finallyPc  proc near               ; CODE XREF: _main+121↓p
.text:0040159A 000                push    ebp
.text:0040159A     ; ---------------------------------------------------------------------------
.text:0040159B 000                db  89h
.text:0040159C 000                in      eax, 83h        ; DMA page register 74LS612:
.text:0040159C                                            ; Channel 1 (address bits 16-23)
.text:0040159E 000                in      al, dx
.text:0040159E     ; ---------------------------------------------------------------------------
.text:0040159F 000                db  28h ; (
.text:0040159F     __Z7finallyPc  endp ; sp-analysis failed
.text:0040159F
.text:004015A0     ; ---------------------------------------------------------------------------
.text:004015A0                    mov     byte ptr [ebp-15h], 25h
.text:004015A4                    mov     byte ptr [ebp-14h], 74h
.text:004015A8                    mov     byte ptr [ebp-13h], 70h
.text:004015AC                    mov     byte ptr [ebp-12h], 26h
.text:004015B0                    mov     byte ptr [ebp-11h], 3Ah
.text:004015B4                    mov     dword ptr [esp], 0
.text:004015BB                    call    _time
.text:004015C0                    mov     [esp], eax
.text:004015C3                    call    _srand
.text:004015C8                    call    _rand
.text:004015CD                    mov     ecx, eax
.text:004015CF                    mov     edx, 51EB851Fh
.text:004015D4                    mov     eax, ecx
.text:004015D6                    imul    edx
.text:004015D8                    sar     edx, 5
.text:004015DB                    mov     eax, ecx
.text:004015DD                    sar     eax, 1Fh
.text:004015E0                    sub     edx, eax
.text:004015E2                    mov     eax, edx
.text:004015E4                    mov     [ebp-0Ch], eax
.text:004015E7                    mov     eax, [ebp-0Ch]
```


手动nop掉多余的db指令，让ida能够自动分析成代码

完成nop后，发现encrypt函数，在两端黑色代码中间插有一段红色代码

```
.text:00401530 004                nop
.text:00401531 000                mov     eax, [ebp+8]
.text:00401531     __Z7encryptPc  endp ; sp-analysis failed
.text:00401531
.text:00401534                    add     eax, edx
.text:00401536                    movzx   edx, byte ptr [eax]
.text:00401539                    mov     eax, [ebp-1Ch]
.text:0040153C                    add     eax, offset aHahahahaDoYouF ; "hahahaha_do_you_find_me?"
.text:00401541                    movzx   eax, byte ptr [eax]
.text:00401544                    xor     eax, edx
.text:00401546                    movsx   edx, al
.text:00401549                    mov     eax, [ebp-1Ch]
.text:0040154C                    mov     eax, [ebp+eax*4-6Ch]
```

```
.text:00401550                  cmp     edx, eax
.text:00401552                  jz      short loc_401573
.text:00401554                  mov     dword ptr [esp], offset aWrong_0 ; "wrong ~"
.text:0040155B                  call    _puts
.text:00401560                  mov     dword ptr [ebp-20h], 0
.text:00401567                  mov     dword ptr [esp], 0
.text:0040156E                  call    _exit
.text:00401573      ; ---------------------------------------------------------------------------
.text:00401573
.text:00401573      loc_401573:                             ; CODE XREF: .text:00401552↑j
.text:00401573                  add     dword ptr [ebp-1Ch], 1
.text:00401577      ; START OF FUNCTION CHUNK FOR __Z7encryptPc
.text:00401577
.text:00401577      loc_401577:                             ; CODE XREF: encrypt(char *)+2C↑j
.text:00401577 000              cmp     dword ptr [ebp-1Ch], 12h
.text:0040157B 000              jle     short loc_40152E
```


这时的处理方法是，在encrypt函数起始地址0x401500处，右键->undefine，变成黄色的数据，再按c转成代码，这时，encrypt函数起始地址与finally函数起始地址之间的代码全部为红色，再在encrypt函数起始地址0x401500处，右键->create function，平衡栈后，F5反汇编

这里encrypt函数反汇编后，好像没有用到未经变换的input，可能是patch代码的时候出错了，不过可以猜测为input和字符串“hahahaha_do_you_find_me?”异或，再与已知的unk_403040数组比较

```
1 int __usercall encrypt@<eax>(char a1@<sf>, char a2@<of>, unsigned __int16 a3@<dx>, int a4@<ebp>)
2 {
3   unsigned __int32 v4; // eax
4   int v5; // ebp
5   signed int v6; // edx
6   int v8; // [esp+4h] [ebp-Ch]
7
8   v4 = __indword(0x57u);
9   LOBYTE(v4) = __inbyte(a3);
10  JUMPOUT(a1 ^ a2, (char *)&loc_4014CD + 4);
11  v5 = a4 + 2;
12  v6 = 19;
13  qmemcpy(&v8, &unk_403040, 76u);               // unk_403040已知
14  for ( *(_DWORD *)(v5 - 28) = 0; *(_DWORD *)(v5 - 28) <= 18; ++*(_DWORD *)(v5 - 28) )
15  {
16    v6 = (char)(*(_BYTE *)(v6 + *(_DWORD *)(v5 + 8)) ^ aHahahahaDoYouF[*(_DWORD *)(v5 - 28)]);
17    if ( v6 != *(_DWORD *)(v5 + 4 * *(_DWORD *)(v5 - 28) - 108) )
18    {
19      puts("wrong ~");
20      *(_DWORD *)(v5 - 32) = 0;
21      exit(0);
22    }
23  }
24  if ( *(_DWORD *)(v5 - 32) == 1 )
25    puts("come here");
26  return *(_DWORD *)(v5 - 32);
27 }
```


写encrypt函数的逆运算脚本，得到部分flag，之所以是部分flag，是因为长度为24的unk_403040的最后5个元素为0，“d_me?”与0异或不变，或者v6=19说明了只异或了19次，而且提交失败

```
s="hahahaha_do_you_find_me?"
unk_403040=[0xe, 0xd, 0x9, 0x6, 0x13, 0x5, 0x58, 0x56, 0x3e, 0x6,
    0xc, 0x3c, 0x1f, 0x57, 0x14, 0x6b, 0x57, 0x59, 0xd,
    0x0, 0x0, 0x0, 0x0, 0x0]
data=[]
for i in range(len(s)):
    data.append(ord(s[i])^unk_403040[i])
print(''.join(chr(i) for i in data))
```

test1

```
test1
D:\python27-x64\python2.exe D:/Python/pycharm/pycfile/test1.py
flag{d07abccf8a410cd_me?
```


同encrypt函数的解析步骤，对finally函数，先nop掉多余的db指令，undefine掉黑色代码，转成数据，按c再转成代码，右键->create function，F5反汇编

没看懂这个函数，看了其他师傅的wp，说这里也是异或，能看到的5个值[37,116,112,38,58]是异或后要比较的值，由于input的最后一位一定是"}"，它异或一个值（"71"）后与58相等，而且它之前的四位也是和这个值（"71"）异或，结果是58之前的四个值

```
int __usercall finally@<eax>(unsigned __int16 a1@<dx>, int a2@<ebp>)
{
  unsigned __int32 v2; // eax
  unsigned int v3; // eax
  int v4; // eax
  int result; // eax

  v2 = __indword(0x83u);
  LOBYTE(v2) = __inbyte(a1);
  *(_BYTE *)(a2 - 21) = 37;
  *(_BYTE *)(a2 - 20) = 116;
  *(_BYTE *)(a2 - 19) = 112;
  *(_BYTE *)(a2 - 18) = 38;
  *(_BYTE *)(a2 - 17) = 58;
  v3 = time(0);
  srand(v3);
  v4 = rand();
  *(_DWORD *)(a2 - 12) = v4 / 100;
  result = v4 - 100 * *(_DWORD *)(a2 - 12);
  *(_DWORD *)(a2 - 12) = result;
  *(_DWORD *)(a2 - 16) = 0;
  if ( *(_DWORD *)(a2 - 16) <= 4 )
  {
    if ( (*(_BYTE *)(a2 - 21 + *(_DWORD *)(a2 - 16)) != *(_BYTE *)(*(_DWORD *)(a2 - 16) + *(_DWORD *)(a2 + 8))) == *(_DWORD *)(a2 - 12) )
      result = puts("Really??? Did you find it?OMG!!!");
    else
      result = puts("I hide the last part, you will not succeed!!!");
  }
  return result;
}
```


写finally函数的逆运算脚本得到后半部分的flag，替换掉"flag{d07abccf8a410cd_me?"的后5位，flag即为"flag{d07abccf8a410cb37a}"

```
arr=[37,116,112,38,58]
num=58^ord('}')
data=[]
for i in arr:
    data.append(i^num)
print(''.join(chr(i) for i in data))
```

```
test2
D:\python27-x64\python2.exe D:/Python/pycharm/pycfile/test2.py
b37a}
```


[2019红帽杯]childRE

exe程序，运行后直接输入，无壳，ida分析

交叉引用字符串"flag{MD5(your input)}"来到sub_140001610函数

sub_140001610函数主要的逻辑为

读取输入，验证输入的长度是否为31，对输入进行位置变换处理，位置变换的结果放到v2（name），经调试可知，输入为abcdefghijklmnopqrstuvwxyz12345时，v2（name）的内容为pqhrsidtujvwkebxylz1mf23n45ogca

对v2（name）调用UnDecorateSymbolName函数处理，结果放到outputstring，验证outputstring的长度62及其内容

```
signed __int64 sub_7FF6DC281610()
```

```
{
  signed __int64 input_len; // rax
  _QWORD *v1; // rax
  const CHAR *v2; // r11
  __int64 v3; // r10
  __int64 v4; // r9
  const CHAR *v5; // r10
  signed __int64 outputstring_len; // rcx
  __int64 v7; // rax
  signed __int64 result; // rax
  unsigned int v9; // ecx
  __int64 v10; // r9
  int v11; // er10
  __int64 v12; // r8
  __int128 input; // [rsp+20h] [rbp-38h]
  __int128 v14; // [rsp+30h] [rbp-28h]

  input = 0i64;
  v14 = 0i64;
  sub_7FF6DC281080((__int64)"%s", &input);      // 读取输入
  input_len = -1i64;
  do
    ++input_len;
  while ( *((_BYTE *)&input + input_len) );
  if ( input_len != 31 )                         // 输入的长度为31
  {
    while ( 1 )
      Sleep(0x3E8u);
  }
  v1 = sub_7FF6DC281280(&input);                 // 对input的处理，通过调试可知，从31行到41行，是对input的位置变换，变换后的结果放入v2，也就是name
                                                 // 例如，输入abcdefghijklmnopqrstuvwxyz12345，name="pqhrsidtujvwkebxylz1mf23n45ogca"
  v2 = name;                                     // v2==name
  if ( v1 )
  {
    sub_7FF6DC2815C0((unsigned __int8 *)v1[1]);
    sub_7FF6DC2815C0(*(unsigned __int8 **)(v3 + 16));
    v4 = dword_7FF6DC2857E0;
    v2[v4] = *v5;
    dword_7FF6DC2857E0 = v4 + 1;
  }
  UnDecorateSymbolName(v2, outputString, 0x100u, 0);// v2，也就是name，经过UnDecorateSymbolName函数处理，结果放到outputstring中
  outputstring_len = -1i64;
  do
    ++outputstring_len;
  while ( outputString[outputstring_len] );
  if ( outputstring_len == 62 )                  // outputstring的长度为62
  {
    v9 = 0;
    v10 = 0i64;
    do                                           // do循环体在验证outputstring的内容
    {
      v11 = outputString[v10];
      v12 = v11 % 23;
      if ( table[v12] != *(_BYTE *)(v10 + 0x7FF6DC283478i64) )// (_@4620!08!6_0*0442!@186%%0@3=66!!974*3234=&0^3&1@=&0908!6_0*&
        _exit(v9);
      if ( table[v11 / 23] != *(_BYTE *)(v10 + 0x7FF6DC283438i64) )// 5556565325555222556556555524346633465366355
```

```
4442656555525555222
        _exit(v9 * v9);
      ++v9;
      ++v10;
    }
    while ( v9 < 0x3E );
    sub_7FF6DC281020("flag{MD5(your input)}\n", v11 / 23, v12, v10);
    result = 0i64;
  }
  else
  {
    v7 = sub_7FF6DC2818A0(std::cout);
    std::basic_ostream<char,std::char_traits<char>>::operator<<(v7, sub_7FF6DC281A60);
    result = 0xFFFFFFFFi64;
  }
  return result;
}
```

先写验证outputstring的长度62及其内容的逆运算脚本，得到outputstring，为一个未修饰的C++符号名

```
outputstring_data=[]
table="1234567890-=!@#$%^&*()_+qwertyuiop[]QWERTYUIOP{}asdfghjkl;'ASDFGHJKL:\"ZXCVBNM<>?zxcvbnm,./"
str_3438="555656532555522255655655552434663346536635444265655555525555222"
str_3478="(_@4620!08!6_0*0442!@186%%0@3=66!!974*3234=&0^3&1@=&0908!6_0*&"
for i in range(len(str_3438)):
    v11_div_23=table.find(str_3438[i])
    v12=table.find(str_3478[i])
    v11=v12+v11_div_23*23
    outputstring_data.append(v11)
print(''.join(chr(i) for i in outputstring_data))
```

```
est1
D:\python27-x64\python2.exe D:/Python/pycharm/pycfile/test1.py
private: char * __thiscall ROPxx::My_Aut0_PWN(unsigned char *)
```


UnDecorateSymbolName函数反修饰指定已修饰的 C++ 符号名，参考：UnDecorateSymbolName

第1个参数为已修饰的 C++ 符号名，此名称能以始终为问号 (?) 的首字符鉴别，本题中为v2（name），第2个参数指向字符串缓冲区的指针，该缓冲区接收未修饰的名字，本题中为outputstring

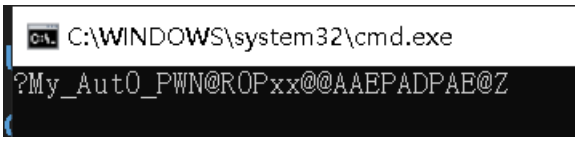
可知需要对outputstring符号名修饰才能得到v2（name），参考：c/c++函数名修饰规则

或者通过写C++代码，调用__FUNCDNAME__宏（**FUNCDNAME**：只有在函数内部才有效，返回该函数经编译器修饰后的名字。如果编译器选项中设定了/EP或/P，则__FUNCDNAME__是未定义。）直接得到函数经编译器修饰后的符号名，需要注意的是函数所在的类，域，返回类型，函数名，参数类型都必须和已知完全相同

```cpp
#include<iostream>
using namespace std;
class R0Pxx {//函数所在的类
public:
 R0Pxx() {//该类的构造函数
  unsigned char a;
  My_Aut0_PWN(&a);
 }

private://函数属于私有域
 //函数的返回类型 函数名 以及参数类型
 char * My_Aut0_PWN(unsigned char*) {
  char * ret = NULL;
  //调用宏 输出该函数经编译器修饰后的符号名
  printf("%s", __FUNCDNAME__);
  return ret;
 }
};
int main() {
 new R0Pxx();
 getchar();
 return 0;
}
```

运行结果即为v2（name），注意是在x86架构下运行的结果



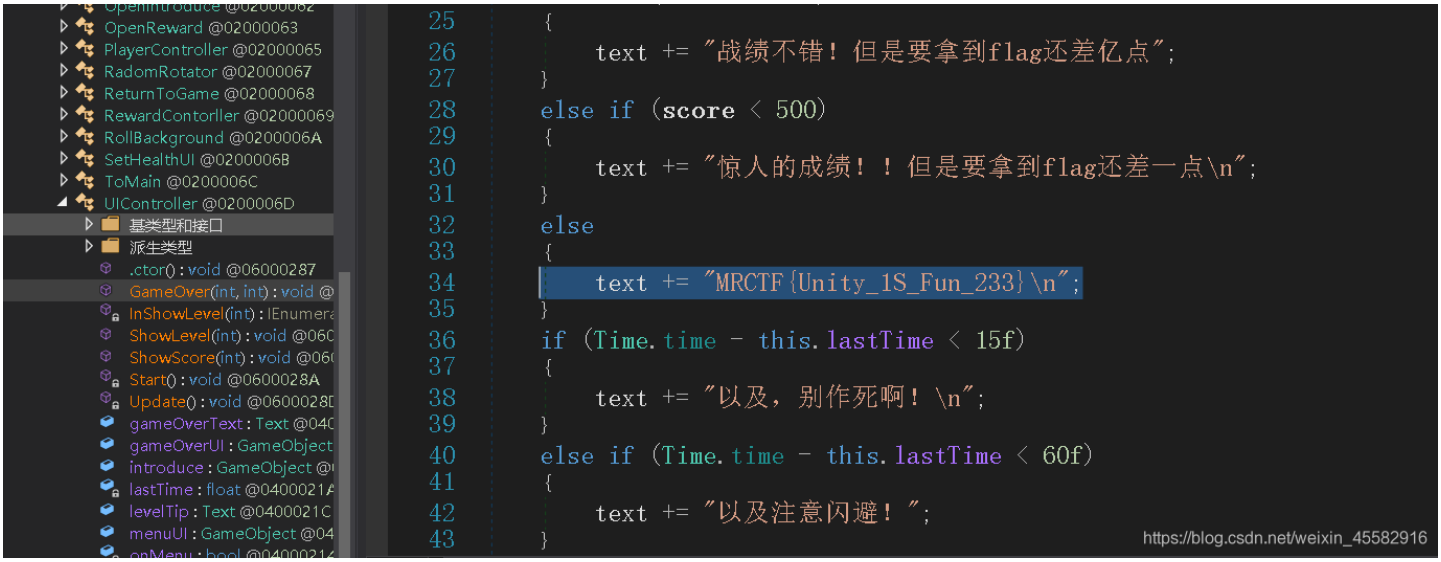
写逆位置变换脚本即可得到输入，对输入进行md5散列即可得到flag

```python
import hashlib
ori="abcdefghijklmnopqrstuvwxyz12345"
change="pqhrsidtujvwkebxylz1mf23n45ogca"
name="?My_Aut0_PWN@R0Pxx@@AAEPADPAE@Z"
flag=[0]*len(name)
for i in range(len(name)):
    flag[ori.find(change[i])]=ord(name[i])
flag_str=''.join(chr(i) for i in flag)
print(flag_str)
h=hashlib.md5()
h.update(flag_str.encode(encoding='utf-8'))
print(h.hexdigest())
```

```
est2
D:\python27-x64\python2.exe D:/Python/pycharm/
Z0@tRAEyuP@xAAA?M_A0_WNPx@@EPDP
63b148e750fed3a33419168ac58083f5
```


[MRCTF2020]PixelShooter

apk文件，jadx-gui打开，什么都没发现
用Apktool Box反编译apk后，在PixelShooter->lib->armeabi-v7a目录下发现3个.so文件，依次分析，同样什么都没发现
突然想到这是个安卓unity游戏，而安卓unity游戏的核心逻辑一般位于assets\bin\Data\Managed\Assembly-CSharp.dll
用dnSpy打开，在类UIController的GameOver方法中找到flag



[ACTF新生赛2020]SoulLike

elf文件，无壳，ida分析
main函数，读取输入，验证输入的前5个字符是否为“actf{”，将输入{}内的字符放入v8，可知输入{}内的长度为12，调用sub_83A函数验证v8，且输入的最后一个字符为“}”

```
15  v13 = __readfsqword(0x28u);
16  printf("input flag:", a2, a3);
17  scanf("%s", input);
18  v9 = 'ftca';
19  v10 = '{';
20  v5 = 1;
21  for ( i = 0; i <= 4; ++i )                    // 验证输入的前5个字符是否为actf{
22  {
23    if ( *((_BYTE *)&v9 + i) != input[i] )
24    {
25      v5 = 0;
26      goto LABEL_6;
27    }
28  }
29  if ( !v5 )
30    goto LABEL_19;
31LABEL_6:
32  for ( j = 0; j <= 11; ++j )                   // 将input的{}内的字符放入v8
33    v8[j] = input[j + 5];
34  v3 = (unsigned __int8)sub_83A(v8) && v12 == '}' ? 1 : 0;// 验证v8，以及输入的最后一个字符为“}”
35  if ( v3 )
36  {
37    printf("That's true! flag is %s", input);
38    result = 0LL;
39  }
40  else
41  {
42LABEL_19:
43    printf("Try another time...");
44    result = 0LL;
45  }
46  return result;
47}
```

分析sub_83A函数，可以看到是对input{}内12个字符的超多异或运算

```
signed __int64 __fastcall sub_83A(_DWORD *a1)
{
  _DWORD *v1; // ST08_8
  signed int i; // [rsp+1Ch] [rbp-44h]
  int v4; // [rsp+20h] [rbp-40h]
  int v5; // [rsp+24h] [rbp-3Ch]
  int v6; // [rsp+28h] [rbp-38h]
  int v7; // [rsp+2Ch] [rbp-34h]
  int v8; // [rsp+30h] [rbp-30h]
  int v9; // [rsp+34h] [rbp-2Ch]
  int v10; // [rsp+38h] [rbp-28h]
  int v11; // [rsp+3Ch] [rbp-24h]
  int v12; // [rsp+40h] [rbp-20h]
  int v13; // [rsp+44h] [rbp-1Ch]
  int v14; // [rsp+48h] [rbp-18h]
  int v15; // [rsp+4Ch] [rbp-14h]
  unsigned __int64 v16; // [rsp+58h] [rbp-8h]

  v1 = a1;                                              // 参数a1为input{}内的字符，长度为12
  v16 = __readfsqword(0x28u);
  *a1 ^= 0x2Bu;
  v1[1] ^= 0x6Cu;
  v1[2] ^= 0x7Eu;
  v1[3] ^= 0x56u;
  v1[4] ^= 0x39u;
  v1[5] ^= 3u;
  v1[6] ^= 0x2Du;
  v1[7] ^= 0x28u;
  v1[8] ^= 8u;
  ++v1[9];
  v1[10] ^= 0x2Fu;
  v1[11] ^= 0xAu;
  ++*v1;
  v1[1] ^= 0xDu;
  v1[2] ^= 0x73u;
  v1[3] ^= v1[2];
```


在sub_83A函数的最后，是input{}内的12个字符经过超多异或运算后的值与已知的v4到v15比较，因为会有类似input[3]^=input[2]的情况，所以不能通过调试得到每个字符最后等效的异或值是什么
不过当比较为不相同时，程序会打印输入是哪个位置的字符错了，于是可以写脚本爆破出flag

```
  v4 = 126;
  v5 = 50;
  v6 = 37;
  v7 = 88;
  v8 = 89;
  v9 = 107;
  v10 = 53;
  v11 = 110;
  v12 = 0;
  v13 = 19;
  v14 = 30;
  v15 = 56;
  for ( i = 0; i <= 11; ++i )
  {
    if ( *(&v4 + i) != a1[i] )
    {
      printf("wrong on #%d\n", (unsigned int)i);
      return 0LL;
```

```
    }
  }
  return 1LL;
}
```


爆破脚本

```
#coding:utf-8
from itertools import *
import subprocess

flag=""
t=""
for i in range(12):
    for j in range(32,126):
        flag ="actf{"+t+chr(j)+"}"
        p = subprocess.Popen(["./SoulLike"], stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PI
PE)
        p.stdin.write(flag)
        p.stdin.close()
        out=p.stdout.read()
        p.stdout.close()

        if "#"+str(i) not in out:
            t+=chr(j)
            break

print(flag)
```

运行结果

```
zihaoli@zihaoliworld:~/CTF$ python z_exp.py
actf{b0Nf|Re_LiT!}
```