

这个题目是鹏程杯预赛里的一道Reverse，是一个比较隐蔽的vm类型题目，赛后战队review被安排到了这个题目，网上的WP都解释的都很简单，正好就以该题目为例，自己写个wp。

vm类型题目一般是自己写程序来解释执行一些基本的汇编程序指令，这种类型题目的解题方法基本是一样的，就是程序复杂程度的区别，简单的话直接读指令就能猜出来程序逻辑。

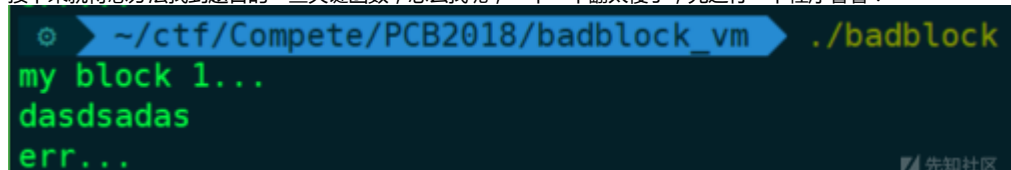
题目分析

拿到题目，拖进IDA里，找到main函数F5看一下反汇编结果：

```
12  unsigned __int64 v13; // [rsp+B8h] [rbp-30h]
13
14  v13 = __readfsqword(0x28u);
15  sub_1E50(&v5, a2, a3);
16  v3 = (_QWORD *)operator new(8uLL);
17  *v3 = off_2059F0;
18  sub_1C50(v3);
19  (*(void (__fastcall **)(_QWORD *))v3)(v3);
20  std::ostream_insert<char,std::char_traits<char>>(&std::cout, "my block 1...", 13LL);
21  std::endl<char,std::char_traits<char>>(&std::cout);
22  v9 = 332847724520534330LL;
23  v8 = 9LL;
24  v10 = 46;
25  v11 = 0;
26  v7 = &v9;
27  sub_2CB0((__int64)&v12, 1, &v7);
28  sub_2160((unsigned int *)&v5, (__int64)&v12);
29  sub_2400((__int64)&v12);
30  if ( v7 != &v9 )
31      operator delete(v7);
32  sub_39D0(&v6);
33  return 0LL;
```

发现看不出什么逻辑，只能看出来是c++的代码，有一句输出，点开每个函数简单看看，发本看不出什么明显的逻辑，但是发现在cout语句前面的几个函数中有两个反调用了。

接下来就得想办法找到题目的一些关键函数，怎么找呢，一个一个翻太慢了，先运行一下程序看看：



需要我们输入一串字符，然后输出错误提示，看来关键逻辑应该是在我们输入之后，所以要先找到cin函数，直接去ida的import视图下找没找到，就从'err...'这个字符串入手。

:0000000000205A10	off_205A10	dq offset sub_41A0
:0000000000205A18		dq offset sub_41D0
:0000000000205A20		dq offset sub_4200
:0000000000205A28		dq offset sub_4240
:0000000000205A30		dq offset sub_4260
:0000000000205A38		dq offset sub_4290
:0000000000205A40		dq offset sub_42C0
:0000000000205A48		dq offset sub_42F0
:0000000000205A50		dq offset sub_43C0
:0000000000205A58		dq offset sub_4320
:0000000000205A60		dq offset sub_4350
:0000000000205A68		dq offset sub_4380
:0000000000205A70		dq offset sub_43B0
:0000000000205A78		dq offset sub_43E0
:0000000000205A80		dq offset sub_43F0

猜想可能是根据偏移量来调用函数的，再寻找off_205A10的调用关系，终于发现了cin：

```

std::operator>><char,std::char_traits<char>,std::allocator<char>>>(&std::cin, &v46);
v48 = &v49;
sub_2BE0(&v48, v46, *((_QWORD *)&v46 + 1) + v46);
sub_2D50(&v67, &v48);
v6 = sub_2DD0((__int64 *)&v67);
if ( v67 != &v69 )
    operator delete(v67);
if ( v48 != &v49 )
    operator delete(v48);

```

先知社区

那么主要逻辑应该就是在cin之后的几个函数。

一个一个分析，首先是sub_2BE0函数：

```

if ( !a2 && a3 )
    std::__throw_logic_error("basic_string::_M_construct null not valid");
v4 = a3 - (_QWORD)a2;
v9 = a3 - (_QWORD)a2;
if ( (unsigned __int64)(a3 - (_QWORD)a2) > 0xF )
{
    v6 = (void *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_create(a1, &v9, 0LL);
    *v3 = v6;
    v3[2] = v9;
LABEL_9:
    memcpy(v6, a2, v4);
    v5 = (_BYTE *)*v3;
    goto LABEL_7;
}
v5 = *a1;
v6 = *a1;
if ( v4 == 1 )
{
    *v5 = *a2;
    v5 = (_BYTE *)*v3;
    goto LABEL_7;
}
if ( v4 )
    goto LABEL_9;
LABEL_7:
v7 = v9;
v3[1] = v9;
v5[v7] = 0;
return __readfsqword(0x28u) ^ v10;

```

先知社区

分析一下，应该是自己实现的一个str_cpy功能函数，把输入的字符串拷贝到另一个地址中。

然后是sub_2D50函数：

```

    result = a1;
    v3 = 4;
    do
    {
        for ( i = 1LL; a2[1] > i; ++i )
            *(_BYTE *)(*a2 + i) ^= *(_BYTE *)(*a2 + i - 1);
        --v3;
    }
    while ( v3 );
    v5 = (_QWORD *)*a2;
    *a1 = a1 + 2;
    if ( v5 == a2 + 2 )
    {
        v7 = a2[3];
        a1[2] = a2[2];
        a1[3] = v7;
    }
    else
    {
        *a1 = v5;
        a1[2] = a2[2];
    }
    v6 = a2[1];
    *a2 = a2 + 2;
    a2[1] = 0LL;
    *((_BYTE *)a2 + 16) = 0;
    a1[1] = v6;
    return result;
}

```

分析一下大概的逻辑是对字符串做了4轮的异或加密，加密逻辑也比较简单，然后将结果赋给另一个变量，分析发现，传入的参数并不是一个简单的字符串，应该是一个结构

```

000 lk_str          struc ; (sizeof=0x20,
000 s              dq ?
008 len           dq ?
010 field_10      dq ?
018 field_18      dq ?
020 lk_str        ends

```

对于不清楚具体含义的域可以先空着，之后分析用到了再去补充，之后再去看这个函数，就会清晰很多：

```

10 result = out;
11 round = 4;
12 do
13 {
14     for ( i = 1LL; in->len > i; ++i )
15         in->s[i] ^= in->s[i - 1];
16     --round;
17 }
18 while ( round );
19 v5 = in->s;
20 out->s = (char *)&out->field_10;
21 if ( v5 == (char *)&in->field_10 )
22 {
23     v7 = in->field_18;
24     out->field_10 = in->field_10;
25     out->field_18 = v7;
26 }
27 else
28 {
29     out->s = v5;
30     out->field_10 = in->field_10;
31 }
32 v6 = in->len;
33 in->s = (char *)&in->field_10;
34 in->len = 0LL;
35 LOBYTE(in->field_10) = 0;
36 out->len = v6;
37 return result;
38 }

```

接下来是sub_2DD0函数，就是在这个函数中，调用了之前发现的function table：

```

1 __BOOL8 __fastcall sub_2DD0(__int64 *a1)
2 {
3     _QWORD *v1; // rax
4     _QWORD *v2; // rbx
5     __int64 v3; // rax
6     __int64 v4; // rsi
7     __int64 v5; // r8
8     __int64 v6; // rdx
9
10    v1 = (_QWORD *)operator new(0x18uLL);
11    v2 = v1;
12    *v1 = off_205A10;
13    v3 = operator new(0x68uLL);
14    v4 = a1[1];
15    if ( v4 )
16    {
17        v5 = *a1;
18        v6 = 0LL;
19        do
20        {
21            *(_WORD *) (v3 + 2 * v6 + 24) = *(char *) (v5 + v6);
22            *(_WORD *) (v3 + 2 * v6 + 64) = aMg[v6];
23            ++v6;
24        }
25        while ( v4 != v6 );
26    }
27    v2[2] = v3;
28    *(_QWORD *) (v3 + 8) = &unk_206020;
29    *(_QWORD *) (v3 + 16) = &unk_206340;
30    return (*(unsigned int (__fastcall **)(__QWORD *))(*v2 + 104LL))(v2) == 1;
31 }

```

分析函数里的操作，指针操作有些多，看着是一些赋值操作，猜测应该也是一些结构体变量的初始化，我们先跳过直接看function call调用的function table里的函数：

```

while ( 1 )
{
    v5 = (unsigned __int8 *)(v4 + 6LL * v3);
    v6 = v5[1];
    if ( (_BYTE)v6 && v6 != v1[1] )
        goto LABEL_3;
    v7 = *v5;
    if ( *v5 == 7 )
    {
        (*(void (__fastcall **)(_QWORD *))*v2 + 48LL)(v2);
        v1 = (int *)v2[2];
        v3 = *v1;
        v4 = *((_QWORD *)v1 + 1);
        goto LABEL_3;
    }
    if ( v7 <= 7u )
    {
        if ( v7 == 3 )
        {
            (*(void (__fastcall **)(_QWORD *))*v2 + 16LL)(v2);
            v1 = (int *)v2[2];
            v3 = *v1;
            v4 = *((_QWORD *)v1 + 1);
        }
        else if ( v7 <= 3u )
        {
            if ( v7 == 1 )
            {
                (*(void (__fastcall **)(_QWORD *))*v2)(v2);
                v1 = (int *)v2[2];
                v3 = *v1;
            }
        }
    }
}

```

主要部分是一个while循环，找到了！！VM的主要逻辑就在这里，简单分析一下，循环中的分支路径应该就是不同的操作函数，根据不同的opcode调用function table里的函数作为对应的Handler，直接分析while循环，会发现指针操作，十分混乱，很难理清逻辑，需要把相关的变量和结构体分析标注清楚来帮助分析。

VM代码分析

想要分析清楚VM代码的主要逻辑，必须要搞清楚程序和汇编指令相关的几种变量，首先是程序流，通常是用数组或字符串表示。然后是pc指针，一般在程序中会是一个寄存器，比如eax，1,2

可以理解为regs[1]+=regs[2]。然后是mem和flag，程序会从一段已知的内存中取值到寄存器中，这个已知的数组一般就是mem，和regs使用的方法一样，只不过regs在寄存器中，而mem在内存中。

分析出程序的pc, regs, mem, flag等变量, 然后对相关的结构体进行标注, 程序就会变得十分容易理解, 这是我标记的相关结构体, 不同题目会有差异, 但本质上寻找的

```
00000000 vmobj          struc ; (sizeof=0x18, r
00000000 func_table     dq ?
00000008 result        dd ?
0000000C field_C       dd ?
00000010 lkk_env       dq ?
00000018 vmobj          ends
00000018
00000000 ; -----
00000000
00000000 lkk_env          struc ; (sizeof=0x68, r
00000000 pc              dd ?
00000004 flag          dd ?
00000008 vmcode        dq ?
00000010 regs          dq ?
00000018 mem           dw 20 dup(?)
00000040 enc_flag       dw 20 dup(?)
00000068 lkk_env          ends
00000068
```

再回头分析相关函数, 就会清晰很多:

```
1 BOOL8 __fastcall VM(lk_str *a1)
2 {
3     vmobj *v1; // rax
4     vmobj *vmobj; // rbx
5     lkk_env *enc; // rax
6     __int64 len; // rsi
7     char *v5; // r8
8     __int64 i; // rdx
9
10    v1 = (vmobj *)new(0x18uLL);
11    vmobj = v1;
12    v1->func_table = (int *)func_table;
13    enc = (lkk_env *)new(0x68uLL);
14    len = a1->len;
15    if ( len )
16    {
17        v5 = a1->s;
18        i = 0LL;
19        do
20        {
21            enc->mem[i] = v5[i];
22            enc->enc_flag[i] = *((_WORD *)enc_flag + i);
23            ++i;
24        }
25        while ( len != i );
26    }
27    vmobj->lkk_env = enc;
28    enc->vmcode = &vmcode;
29    enc->regs = &regs;
30    return (*((unsigned int (__fastcall **)(vmobj *))vmobj->func_table + 0xD))(vmobj) == 1;
31 }
```

```

13 env = vmobj->lkk_env;
14 v2 = vmobj;
15 v3 = 0;
16 env->pc = 0;
17 vmobj->result = 1;
18 v4 = env->vmcode;
19 while ( 1 )
20 {
21     opcode_ = (unsigned __int16 *)&v4[3 * v3];
22     flag = *((unsigned __int8 *)opcode_ + 1);
23     if ( (_BYTE)flag && flag != env->flag )
24         goto LABEL_3;
25     opcode = *(_BYTE *)opcode_;
26     if ( *(_BYTE *)opcode_ == 7 )
27     {
28         (*((void (__fastcall **)(vmobj *))v2->func_table + 6))(v2);
29         env = v2->lkk_env;
30         v3 = env->pc;
31         v4 = env->vmcode;
32         goto LABEL_3;
33     }
34     if ( opcode <= 7u )
35     {
36         if ( opcode == 3 )
37         {
38             (*((void (__fastcall **)(vmobj *))v2->func_table + 2))(v2);
39             env = v2->lkk_env;
40             v3 = env->pc;
41             v4 = env->vmcode;
42         }
43         else if ( opcode <= 3u )

```

之后就是对每一个Handler函数进行分析，这里以只举一个例子，function table里的第一个函数：

```

1 _WORD *__fastcall func1_add(vmobj *a1)
2 {
3     lkk_env *v1; // rax
4     signed __int64 v2; // rcx
5     signed __int16 *vmcode; // rdx
6     _WORD *regs; // rax
7
8     v1 = a1->lkk_env;
9     v2 = 3LL * v1->pc;
10    vmcode = v1->vmcode;
11    regs = v1->regs;
12    regs[vmcode[v2 + 1]] += regs[vmcode[v2 + 2]];
13    return regs;
14 }

```

这是一个汇编的add命令，操作数是两个寄存器在寄存器数组中的索引，vmcode是程序流，一个word类型的数组，pc指针在每次乘3（其他Handler中也是），说明指令是reg1,reg2。

用同样的方法对function table中的所有函数分析，得到如下结果：

```
00205A10 func_table      dq offset func1_add      ; DATA XREF: VM
00205A18                dq offset func2_sub
00205A20                dq offset func3_cmp_regs
00205A28                dq offset func4_jmpoffset
00205A30                dq offset func5_mov_reg2reg
00205A38                dq offset func6_mov_pc2reg
00205A40                dq offset func7_jmpreg
00205A48                dq offset func8_mov_value2reg
00205A50                dq offset func9_err_res0
00205A58                dq offset func10_load_mem2reg
00205A60                dq offset func11_load_encflag2reg
00205A68                dq offset func12_xor_regreg
00205A70                dq offset func13_check_res_no0
00205A78                dq offset vm_main_loop
```

值得注意的是，分析发现程序中opcode的种类只有有限的几种，但是程序流vmcode中出现了一些无法匹配的opcode，仔细分析程序发现，vmcode是word类型的数组，

```
0x0003, 0x0006, 0x0005,
0x0104, 0xFFFF5, 0x0000,
0x0204, 0x0001, 0x0000,
```

终于做完了所有的准备工作，进入正题，这些vm命令到底做了什么事情呢，我自己写了一个脚本来将程序流解释为比较直观的类汇编命令，代码如下：

```
#!/usr/bin/perl -u
#-*-coding:utf-8-*-
code=[
0x0008, 0x0000, 0x0014,
0x0008, 0x0001, 0x0000,
0x0008, 0x0002, 0x0001,
0x0008, 0x0007, 0x0009,
0x0008, 0x0008, 0x0000,
0x0008, 0x0009, 0x0000,
0x0001, 0x0009, 0x0008,
0x0001, 0x0008, 0x0002,
0x0003, 0x0007, 0x0008,
0x0204, 0xFFFFC, 0x0000,
0x0005, 0x0003, 0x0009,
0x0003, 0x0001, 0x0000,
0x0104, 0x000A, 0x0000,
0x0005, 0x0004, 0x0001,
0x0001, 0x0004, 0x0003,
0x0001, 0x0004, 0x0004,
0x000A, 0x0005, 0x0001,
0x000C, 0x0005, 0x0004,
0x000B, 0x0006, 0x0001,
0x0001, 0x0001, 0x0002,
0x0003, 0x0006, 0x0005,
0x0104, 0xFFFF5, 0x0000,
0x0204, 0x0001, 0x0000,
0x00FF, 0x0000, 0x0000,
0x0009, 0x0000, 0x0000,
0x00FF, 0x0000, 0x0000,
0x0000, 0x0000]

def interpreter():
    inst = {
        1: "add",
        2: "sub",
        3: "cmp_regs(== ->1,!= ->2)",
```

```

4: "jmpoffset_newpc",
5: "mov_reg2reg",
6: "mov_pc_reg",
7: "jmpreg_newpc",
8: "mov_value2reg",
9: "err_res0",
10: "load_mem2reg",
11: "load_encflag2reg",
12: "xor_reg^reg",
13: "check_res_no0",
255: "exit"
}

```

```
flag=0
```

```
for i in range(0,len(code),3):
```

```
    if(i+2>=len(code)):
```

```
        break
```

```
    opcode=code[i]
```

```
    opl=code[i+1]
```

```
    op2=code[i+2]
```

```
    if not inst.has_key(opcode):
```

```
        print hex(opcode)
```

```
        flag=opcode>>8
```

```
        opcode&=0xff
```

```
    else:
```

```
        flag=0
```

```
    print ("%02d: %04x %04x %04x -- %s %d %d" % (i/3,opcode,opl,op2,inst[opcode],opl,op2))
```

```
interpreter()
```

得到结果如下：

```
00: 0008 0000 0014 -- mov_value2reg 0 20
01: 0008 0001 0000 -- mov_value2reg 1 0
02: 0008 0002 0001 -- mov_value2reg 2 1 //i
03: 0008 0007 0009 -- mov_value2reg 7 9
04: 0008 0008 0000 -- mov_value2reg 8 0
05: 0008 0009 0000 -- mov_value2reg 9 0
06: 0001 0009 0008 -- add 9 8
07: 0001 0008 0002 -- add 8 2
08: 0003 0007 0008 -- cmp_regs(== ->1,!= ->2) 7 8
0x204
09: 0004 fffc 0000 -- jmpoffset_newpc 65532(-4) 0 //循环算出一个常数36到r9中
10: 0005 0003 0009 -- mov_reg2reg 3 9
11: 0003 0001 0000 -- cmp_regs(== ->1,!= ->2) 1 0
0x104
12: 0004 000a 0000 -- jmpoffset_newpc 10 0
13: 0005 0004 0001 -- mov_reg2reg 4 1
14: 0001 0004 0003 -- add 4 3
15: 0001 0004 0004 -- add 4 4
16: 000a 0005 0001 -- load_mem2reg 5 1
17: 000c 0005 0004 -- xor_reg^reg 5 4
18: 000b 0006 0001 -- load_encflag2reg 6 1
19: 0001 0001 0002 -- add 1 2
20: 0003 0006 0005 -- cmp_regs(== ->1,!= ->2) 6 5
0x104
21: 0004 fff5 0000 -- jmpoffset_newpc 65525(-11) 0
0x204
22: 0004 0001 0000 -- jmpoffset_newpc 1 0
23: 00ff 0000 0000 -- exit 0 0
24: 0009 0000 0000 -- err_res0 0 0
25: 00ff 0000 0000 -- exit 0 0
```



对汇编命令分析就简单了，这个程序也不复杂，首先是通过一个循环，累加0-9，得到一个常数36，之后用36对字符串中的每一位异或 $(36+i) \times 2$ 。然后与mem中固定的字符串

解题

总结一下程序逻辑，首先读输入的字符串做了4轮的异或加密，之后对加密后的字符串每一位都异或 $(36+i) \times 2$ ，然后与固定的字符串进行比较，如果一致，则通过检查。我们

```
def solve():
    data=[0x002E, 0x0026, 0x002D, 0x0029, 0x004D, 0x0067, 0x0005, 0x0044, 0x001A, 0x000E, 0x007F, 0x007F, 0x007D, 0x0065, 0x007F]
    res=[]
    for i in range(len(data)):
        res.append(data[i]^(36+i)*2)
    print res
    for i in range(4):
        for j in range(19,0,-1):
            res[j]^=res[j-1]
    print ''.join(map(chr,res))

solve()
#flag{Y0u_ar3_S0co0L}
```

总结

分析完程序之后发现，程序逻辑实际上很简单，其实大多VM类型题目的逻辑都不会特别复杂，很多都是异或一个常数、异或字符串中的前一个字符等比较直观的处理。当然

这类题目十分考验基本的逆向能力，没有多少技术上的要求，只要能读懂汇编，有耐心，抓住重点，分析出vm指令的pc、regs等相关变量，就能分析出程序的大概逻辑，除

有什么问题欢迎留言讨论~~

点击收藏 | 2 关注 | 1

[上一篇：s-cms代码审计----任意文件下载](#) [下一篇：Windows Privilege...](#)

1. 0 条回复
- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)