2018年强网杯初赛 逆向题目 hide writeup (超详细)

原创

j1rufeng 于 2018-03-28 23:41:33 发布
分类专栏： CTF WP 文章标签： CTF 逆向 强网杯 write up

这道题有壳，strings搜索是upx3.91。但是不能用upx -d。只好手动脱壳了。

难点在于本题目有反调试，有些函数不能步过。

首先研究壳。

```
LOAD:00000000044EFB0                public start
LOAD:00000000044EFB0 start          proc near               ; DATA XREF: LOAD:0000000000400018↑o
LOAD:00000000044EFB0                call    loc_44F230
LOAD:00000000044EFB5                push    rbp
LOAD:00000000044EFB6                push    rbx
LOAD:00000000044EFB7                push    rcx
LOAD:00000000044EFB8                push    rdx
LOAD:00000000044EFB9                add     rsi, rdi
LOAD:00000000044EFBC                push    rsi
LOAD:00000000044EFBD                mov     rsi, rdi
LOAD:00000000044EFC0                mov     rdi, rdx
LOAD:00000000044EFC3                xor     ebx, ebx
LOAD:00000000044EFC5                xor     ecx, ecx
LOAD:00000000044EFC7                or      rbp, 0FFFFFFFFFFFFFFFFh
LOAD:00000000044EFCB                call    sub_44F020
LOAD:00000000044EFD0                add     ebx, ebx
LOAD:00000000044EFD2                jz      short loc_44EFD6
LOAD:00000000044EFD4                rep retn
LOAD:00000000044EFD6 ; ---------------------------------------------------------------------------
LOAD:00000000044EFD6
LOAD:00000000044EFD6 loc_44EFD6:                            ; CODE XREF: start+22↑j
LOAD:00000000044EFD6                mov     ebx, [rsi]
LOAD:00000000044EFD8                sub     rsi, 0FFFFFFFFFFFFFFFCh
LOAD:00000000044EFDC                adc     ebx, ebx
LOAD:00000000044EFDE                mov     dl, [rsi]
LOAD:00000000044EFE0                rep retn
LOAD:00000000044EFE0 start          endp ; sp-analysis failed
LOAD:00000000044EFEA
```

第一句call必须步入。

```
LOAD:000000000044F230 loc_44F230:                             ; CODE XREF: start↑p
LOAD:000000000044F230                 pop     rbp
LOAD:000000000044F231                 lea     rax, [rbp-9]
LOAD:000000000044F235                 mov     r15d, [rax]
LOAD:000000000044F238                 mov     edx, 0C8h
LOAD:000000000044F23D                 sub     rax, r15
LOAD:000000000044F240                 sub     r15d, edx
LOAD:000000000044F243                 lea     rcx, [rax+rdx]
LOAD:000000000044F247                 call    sub_44F1C5
LOAD:000000000044F24C                 jb      short loc_44F254
LOAD:000000000044F24C ; ---------------------------------------------------------------------------
LOAD:000000000044F24E                 dw 0
LOAD:000000000044F250                 db 6Ch, 5, 2 dup(0)
LOAD:000000000044F254 ; ---------------------------------------------------------------------------
LOAD:000000000044F254
LOAD:000000000044F254 loc_44F254:                             ; CODE XREF: LOAD:000000000044F24C↑j
LOAD:000000000044F254                 or      [rcx+19h], cl
LOAD:000000000044F257                 add     dh, bh
LOAD:000000000044F257 ; ---------------------------------------------------------------------------
LOAD:000000000044F259                 db 2 dup(0FFh), 0E6h, 0E8h, 5Fh, 0, 2Fh
LOAD:000000000044F260 aProcSelfExe    db 'proc/self/exe',0
```

Sub_44f1c5必须步入。

Sub_44F1C5中，

```
LOAD:000000000044F1E5                 pop     r10             ; flags
LOAD:000000000044F1E7                 sub     r8d, r8d        ; fd
LOAD:000000000044F1EA                 push    9
LOAD:000000000044F1EC                 pop     rax
LOAD:000000000044F1ED                 syscall                 ; LINUX - sys_mmap
LOAD:000000000044F1EF                 cmp     edi, eax
LOAD:000000000044F1F1                 jnz     loc_44F0EB
LOAD:000000000044F1F7                 mov     esi, offset dword_400000
LOAD:000000000044F1FC                 mov     edx, edi
LOAD:000000000044F1FE                 sub     edx, esi
LOAD:000000000044F200                 jz      short loc_44F217
LOAD:000000000044F202                 add     ebp, edx        ; 增加偏移的地址差
LOAD:000000000044F204                 add     [rsp+28h+var_20], edx
LOAD:000000000044F208                 add     [rsp+28h+var_10], edx
LOAD:000000000044F20C
LOAD:000000000044F20C loc_44F20C:                             ; CODE XREF: LOAD:000000000044F1A5↑j
LOAD:000000000044F20C                 mov     ecx, ebx
LOAD:000000000044F20E                 sub     ecx, esi
LOAD:000000000044F210                 shr     ecx, 3
LOAD:000000000044F213                 cld
LOAD:000000000044F214                 rep movsq
LOAD:000000000044F217
LOAD:000000000044F217 loc_44F217:                             ; CODE XREF: sub_44F1C5+3B↑j
LOAD:000000000044F217                 xchg    eax, edi
LOAD:000000000044F218                 mov     rsi, rbx
LOAD:000000000044F21B                 push    rax             ; 这是84f248，复制的最后一个地址，作为返回地址。这句汇编是一个call函数。call调到44f2ac
LOAD:000000000044F21C                 xchg    eax, edx
LOAD:000000000044F21D                 lodsd
LOAD:000000000044F21E                 push    rax
LOAD:000000000044F21F
LOAD:000000000044F21F loc_44F21F:                             ; CODE XREF: LOAD:000000000044F1A8↑j
LOAD:000000000044F21F                 mov     rcx, rsp
LOAD:000000000044F222                 lodsd
LOAD:000000000044F223                 xchg    eax, edi
LOAD:000000000044F224                 lodsd
LOAD:000000000044F225                 movzx   r8d, al
LOAD:000000000044F229                 xchg    rdi, rsi
LOAD:000000000044F22C                 call    rbp             ; 可以不进入，不停止
LOAD:000000000044F22E                 pop     rcx
LOAD:000000000044F22F                 retn
LOAD:000000000044F22F sub_44F1C5      endp ; sp-analysis failed
```

0x44f214的rep指令把0x400000到0x44f248复制到0x800000位置。

0x44f22c处，rbp是0x84efb5。这句话是start处的第二条指令。这里不必步入。如果步入，

在44EFCB处的call　sub_44F020，再步过就会卡死（不知道为什么，奇怪！如果卡死，按ctrl+c可以只是终止而不退出gdb）；步入之后可以使用finish命令运行到函数外，即0x44F22E处pop　rcx。

在0x44f22f retn之后，到达0x84f248: call　0x84f2ac。

0x84f2ac是一开始没有复制的地址，可以考虑是0x84efb5的代码形成的新代码。此处内存开辟是在系统调用mmap处，刚开辟后此处全为0.

此时可以dump出来0x800000处的内存，见dump2.（我修改了ep，va，filesize）

```
LOAD:000000000084F2D7
LOAD:000000000084F2D7 loc_84F2D7:                              ; CODE XREF: start+33↓j
LOAD:000000000084F2D7                 cmp     qword ptr [rsi], 0
LOAD:000000000084F2DB                 movsq
LOAD:000000000084F2DD                 movsq
LOAD:000000000084F2DF                 jnz     short loc_84F2D7
LOAD:000000000084F2E1                 lea     r15, [rdi-8]
LOAD:000000000084F2E5                 mov     [rdx], rdi
LOAD:000000000084F2E8                 mov     eax, 3D202020h
LOAD:000000000084F2ED                 stosd
LOAD:000000000084F2EE                 mov     edx, 1000h      ; bufsiz
LOAD:000000000084F2F3                 mov     rsi, rdi        ; buf
LOAD:000000000084F2F6                 mov     rdi, r9         ; path
LOAD:000000000084F2F9                 push    59h
LOAD:000000000084F2FB                 pop     rax
LOAD:000000000084F2FC                 syscall                 ; LINUX - sys_readlink
LOAD:000000000084F2FE                 test    eax, eax
LOAD:000000000084F300                 js      short loc_84F306
LOAD:000000000084F302                 mov     byte ptr [rsi+rax], 0
LOAD:000000000084F306
LOAD:000000000084F306 loc_84F306:                              ; CODE XREF: start+54↑j
LOAD:000000000084F306                 add     r9, 0Fh
LOAD:000000000084F30A                 pop     rcx
LOAD:000000000084F30B                 pop     rsi
LOAD:000000000084F30C                 pop     rdi
LOAD:000000000084F30D                 sub     rsp, 800h
LOAD:000000000084F314                 mov     rdx, rsp
LOAD:000000000084F317                 mov     r8, rbp
LOAD:000000000084F31A                 push    0
LOAD:000000000084F31C                 call    sub_84F782      ; 可以步过
LOAD:000000000084F321                 pop     rdx
LOAD:000000000084F322                 add     rsp, 800h
LOAD:000000000084F329                 pop     rsi
LOAD:000000000084F32A                 pop     rdi
LOAD:000000000084F32B                 pop     rcx
LOAD:000000000084F32C                 pop     rcx
LOAD:000000000084F32D                 shl     ecx, 0Ch
LOAD:000000000084F330                 add     rdi, rcx
LOAD:000000000084F333                 sub     esi, ecx
LOAD:000000000084F335                 push    rax
LOAD:000000000084F336                 push    0Bh
LOAD:000000000084F338                 pop     rax
LOAD:000000000084F339                 jmp     qword ptr [r15]
LOAD:000000000084F339 start           endp
LOAD:000000000084F339
```

接着运行到0x84f31c: call 0x84f782，可以步过。

在0x84f339: jmp QWORD PTR [r15]，跳转到0x40000c

此时可以dump处0x400000处的内存，见文件bindump.so。这就是脱壳后的文件了，但是不能运行，不清楚原因。可以看到ep是0x400890。

0x40000c: syscall（调用号11，sys_munmap，解除内存映射）

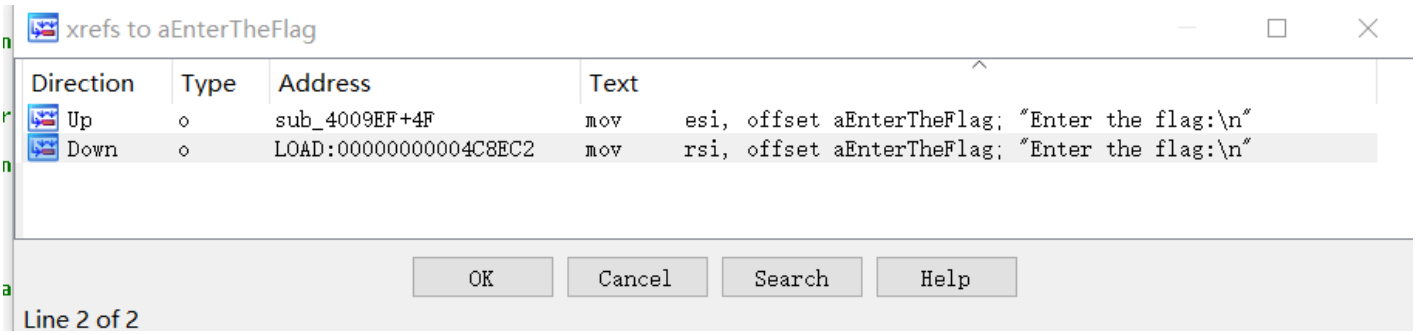
0x40000e: ret

上面两处是elf头部的padding中存储的，感叹做的壳之精准。

ret之后到达：

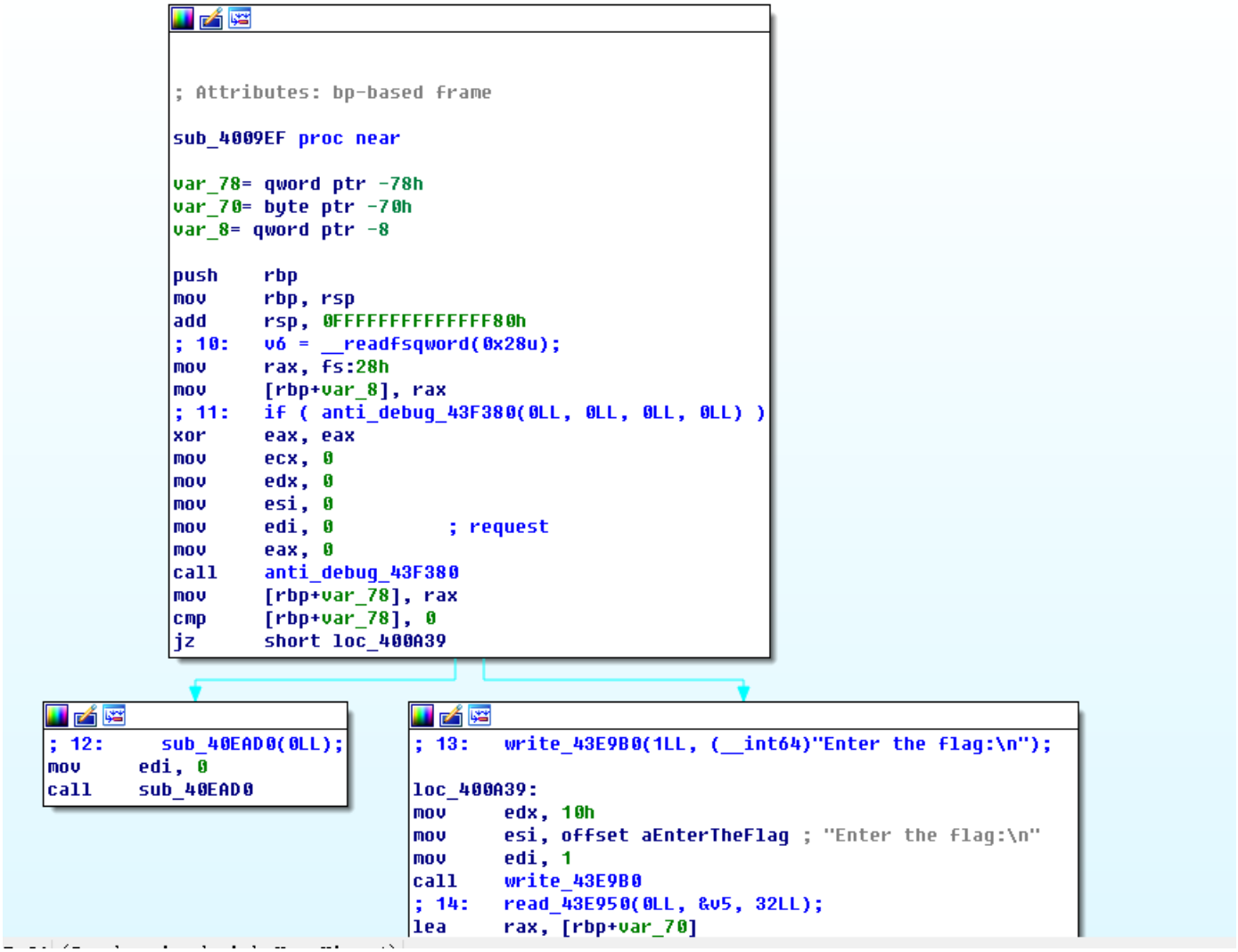
0x400890: xor ebp,ebp

下一部分，分析主逻辑。



发现Enter the flag出现了两次。

按照程序执行过程只会执行上面一个。



```
__int64 sub_4009EF()
{
  const char *v0; // rsi
  __int64 v1; // rdx
  __int64 result; // rax
  __int64 v3; // rcx
  unsigned __int64 v4; // rt1
  char v5; // [rsp+10h] [rbp-70h]
  unsigned __int64 v6; // [rsp+78h] [rbp-8h]

  v6 = __readfsqword(0x28u);
  if ( anti_debug_43F380(0LL, 0LL, 0LL, 0LL) )
    sub_40EAD0(0LL);
  write_43E9B0(1LL, (__int64)"Enter the flag:\n");
  read_43E950(0LL, &v5, 32LL);
  if ( (unsigned int)sub_4009AE((__int64)&v5) != 0 )
  {
    v0 = "You are right\n";
    write_43E9B0(1LL, (__int64)"You are right\n");
  }
  else
  {
    v0 = "You are wrong\n";
    write_43E9B0(1LL, (__int64)"You are wrong\n");
  }
  result = 0LL;
  v4 = __readfsqword(0x28u);
  v3 = v4 ^ v6;
  if ( v4 != v6 )
    sub_442480(1LL, v0, v1, v3);
  return result;
}
```

从流程图上可以看到左枝非正常退出，猜测和反调试有关，进入43f380函数，看到调用了ptrace的系统调用，确实是反调试。由于没有脱壳成功，这里没办法修改判断条件，只好用gdb脚本修改eflags寄存器了。

接着分析0x4009AE函数：

```
_BOOL8 __fastcall sub_4009AE(__int64 a1)
{
  return (unsigned int)sub_400360(a1, (__int64)"qwb{this_is_wrong_flag}") == 0;
}
```

这个时候就应该认识到这是错误的分支了，毕竟告诉你是错的flag了。但是我继续分析了，

```
__int64 __fastcall sub_400360(__int64 a1, __int64 a2)
{
  return qword_6C9050(a1, a2);
}
```

具体函数还没有写入。根据动态调试，最后比较字符串的函数是42D820.

```
33    __asm
34    {
35      movdqu  xmm1, xmmword ptr [rdi]; 简单说就是mov
36      movdqu  xmm0, xmmword ptr [rsi]
37      pcmpeqb xmm0, xmm1; 比较，相同byte置ff. 字符串的首字节存在最后byte
38      pminub  xmm0, xmm1; 感觉是逐字节取小的
39      pxor    xmm1, xmm1
40      pcmpeqb xmm0, xmm1
41      pmovmskb eax, xmm0; 取每个字节的最高位
42    }
43    if ( _RAX )
44      goto LABEL_19;
45    __asm
46    {
47      movdqu  xmm6, xmmword ptr [rdi+10h]; 自己输入
48      movdqu  xmm3, xmmword ptr [rsi+10h]
49      movdqu  xmm5, xmmword ptr [rdi+20h]
50      pcmpeqb xmm3, xmm6
51      movdqu  xmm2, xmmword ptr [rsi+20h]
52      pminub  xmm3, xmm6
53      pcmpeqb xmm3, xmm1; xmm1 是0
54      movdqu  xmm4, xmmword ptr [rdi+30h]
55      pcmpeqb xmm2, xmm5
56      pmovmskb edx, xmm3
57      movdqu  xmm0, xmmword ptr [rsi+30h]
58      pminub  xmm2, xmm5
59      pcmpeqb xmm2, xmm1
60      pcmpeqb xmm0, xmm4
61      pmovmskb eax, xmm2
62      pminub  xmm0, xmm4
63      pcmpeqb xmm0, xmm1
64      pmovmskb ecx, xmm0
65    }
```

仔细分析逻辑，需要保证输入地址的64字节和目标地址的64字节相等，而只能输入32字节，根本不能实现。

废了很久时间之后，想到了另一处Enter the flag。

```
LOAD:00000000004C8E92                 align 20h
LOAD:00000000004C8EA0                 xor     rdi, rdi
LOAD:00000000004C8EA3                 xor     rsi, rsi
LOAD:00000000004C8EA6                 xor     rdx, rdx
LOAD:00000000004C8EA9                 xor     r10, r10
LOAD:00000000004C8EAC                 mov     eax, 65h
LOAD:00000000004C8EB1                 syscall                 ; LINUX - sys_ptrace
LOAD:00000000004C8EB3                 cmp     eax, 0
LOAD:00000000004C8EB6                 jnz     locret_4C8FDB
LOAD:00000000004C8EBC                 xor     rdi, rdi
LOAD:00000000004C8EBF                 inc     rdi
LOAD:00000000004C8EC2                 mov     rsi, offset aEnterTheFlag ; "Enter the flag:\n"
LOAD:00000000004C8EC9                 mov     rdx, 10h
LOAD:00000000004C8ED0                 xor     eax, eax
LOAD:00000000004C8ED2                 inc     eax
LOAD:00000000004C8ED4                 syscall                 ; LINUX - sys_write
LOAD:00000000004C8ED6                 xor     rdi, rdi
LOAD:00000000004C8ED9                 xor     eax, eax
LOAD:00000000004C8EDB                 mov     rsi, offset input_6CCDB0
LOAD:00000000004C8EE2                 mov     rdx, 20h
LOAD:00000000004C8EE9                 syscall                 ; LINUX - sys_read
LOAD:00000000004C8EEB                 cmp     eax, 0
LOAD:00000000004C8EEE                 jle     loc_4C8FA9
LOAD:00000000004C8EF4 ; 7:   if ( strlen(input_6CCDB0) == 21
LOAD:00000000004C8EF4 ; 8:     && input_6CCDB0[1] == 'w'
LOAD:00000000004C8EF4 ; 9:     && input_6CCDB0[2] == 'b'
LOAD:00000000004C8EF4 ; 10:     && input_6CCDB0[3] == '{'
LOAD:00000000004C8EF4 ; 11:     && input_6CCDB0[20] == '}' )
LOAD:00000000004C8EF4
LOAD:00000000004C8EF4 ; =============== S U B R O U T I N E =======================================
LOAD:00000000004C8EF4
LOAD:00000000004C8EF4
LOAD:00000000004C8EF4 sub_4C8EF4      proc near
LOAD:00000000004C8EF4                 mov     rdi, offset input_6CCDB0
LOAD:00000000004C8EFB                 mov     rcx, 0FFFFFFFFFFFFFFFFh
LOAD:00000000004C8F02                 xor     eax, eax
LOAD:00000000004C8F04                 repne scasb
LOAD:00000000004C8F06                 not     rcx
LOAD:00000000004C8F09                 sub     rcx, 1
```

这里并未被ida识别成函数，0x4c8ef4处是我标记的函数，从这里标记可以使函数能使用f5.由于原程序并不会执行到此处，需要gdb脚本中修改pc值。

```
signed __int64 sub_4C8EF4()
{
  _BYTE *v0; // rdi
  __int64 *v1; // rsi
  unsigned __int64 v2; // rdx
  signed __int64 result; // rax

  if ( strlen(input_6CCDB0) == 21
    && input_6CCDB0[1] == 'w'
    && input_6CCDB0[2] == 'b'
    && input_6CCDB0[3] == '{'
    && input_6CCDB0[20] == '}' )
  {
    encrypt1_4C8CC0((__int64)&input_4_6CCDB4);
    sub_4C8E50((__int64)&input_4_6CCDB4);
    encrypt1_4C8CC0((__int64)&input_4_6CCDB4);
    sub_4C8E50((__int64)&input_4_6CCDB4);
    encrypt1_4C8CC0((__int64)&input_4_6CCDB4);
    v0 = &input_4_6CCDB4;
    sub_4C8E50((__int64)&input_4_6CCDB4);
    v1 = qword_4C8CB0;                          // target
    v2 = 0LL;
    while ( v2 < 0x10 && *v0 == *(_BYTE *)v1 )
    {
      ++v2;
      ++v0;
      v1 = (__int64 *)((char *)v1 + 1);
    }
  }
  __asm { syscall; LINUX - sys_write }
  result = 60LL;
  __asm { syscall; LINUX - sys_exit }
  return result;
}
```

下面是可以快速进入关键位置的gdb脚本。

```
#!/bin/bash
file hide
b *0x44f22f
r
si
si
d
b *0x84f339
#jmp    QWORD PTR [r15];0x40000c
c
d
si
si
si
b* 0x4009EF
c
set $rip=0x4C8EBC
b *0x4c8f11
c
set $eflags = $eflags |(1<<6)
5
b *0x4C8D09
c
c
d
b *0x4C8DFB
c
```

下面是本题目的exp

```
#coding=utf-8
import struct
import string
def u32(data):
    return struct.unpack("<I",data)[0]

def p32(data):
    return struct.pack("<I",data)

def u64(data):
    return struct.unpack("<Q",data)[0]

def p64(data):
    return struct.pack("<Q",data)

input1 = '1234567890123456'
input1 = bytearray(input1)
CONST_STR = 's1IpP3rEv3Ryd4Y3'
CONST = 0X676E696C
v4 = 0
v4_4=0
v4_4_arr = [0 for i in range(0,9)]
for i in range(1,9):
    v4_4_arr[i] = (v4_4_arr[i-1]+CONST)&0XFFFFFFFF

def re_block(byte_arr_8):
```

```
    i = 0
    v3 = u32(byte_arr_8[0:4])
    v4 = u32(byte_arr_8[4:8])
    print 'round', v3, v4
    for i in range(7,-1,-1):
        v30 = (v3 << 4) & 0xffffffff
        v2c = v3 >> 5
        edx = v30 ^ v2c
        v30 = (v3 + edx) & 0xffffffff

        v28 = (v4_4_arr[i+1] >> 11) & 3
        edx = u32(CONST_STR[v28 * 4:(v28 + 1) * 4])
        v2c = (v4_4_arr[i+1] + edx) & 0xffffffff  # xxxx
        # print 'v2c',hex(v2c)
        print v30 ^ v2c
        v4 = (v4+0x100000000-(v30 ^ v2c)) & 0xffffffff

        v30 = (v4 << 4) & 0xffffffff
        v2c = v4 >> 5
        edx = v30 ^ v2c

        v30 = (v4 + edx) & 0xffffffff

        v28 = v4_4_arr[i] & 3
        edx = u32(CONST_STR[v28 * 4:(v28 + 1) * 4])
        v2c = (v4_4_arr[i] + edx) & 0xffffffff

        v3 = (v3+0x100000000-(v30 ^ v2c)) & 0xffffffff

        print 'round',i,v3,v4
    byte_arr_8[0:4] = p32(v3)
    byte_arr_8[4:8] = p32(v4)
    return byte_arr_8
def xor16(byte_arr_16):
    for i in range(0,16):
        byte_arr_16[i]^=i
def re_all(str16):
    byte_arr = bytearray(str16)
    xor16(byte_arr)#传入整个bytearray，就是传入地址
    byte_arr[0:8] = re_block(byte_arr[0:8])#传入部分bytearray，就是复制之后再传入
    byte_arr[8:16] = re_block(byte_arr[8:16])
    xor16(byte_arr)
    byte_arr[0:8] = re_block(byte_arr[0:8])
    byte_arr[8:16] = re_block(byte_arr[8:16])
    xor16(byte_arr)
    byte_arr[0:8] = re_block(byte_arr[0:8])
    byte_arr[8:16] = re_block(byte_arr[8:16])
    return str(byte_arr)


def block(str8):
    i=0
    input1=bytearray(str8)
    v3 = u32(input1[8 * i:8 * i + 4])
    v4 = u32(input1[8 * i + 4:8 * (i + 1)])
    v4_4 = 0  ##0000
    for j in range(0, 8):

        v30 = (v4 << 4) & 0xffffffff
        v2c = v4 >> 5
```

```
        edx = v30 ^ v2c

        v30 = (v4 + edx) & 0xffffffff

        v28 = v4_4 & 3
        edx = u32(CONST_STR[v28 * 4:(v28 + 1) * 4])
        v2c = (v4_4 + edx) & 0xffffffff

        v3 = ((v30 ^ v2c) + v3) & 0xffffffff


        v4_4 = (v4_4 + CONST) & 0xffffffff

        v30 = (v3 << 4) & 0xffffffff
        v2c = v3 >> 5
        edx = v30 ^ v2c
        v30 = (v3 + edx) & 0xffffffff

        v28 = (v4_4 >> 11) & 3
        edx = u32(CONST_STR[v28 * 4:(v28 + 1) * 4])
        v2c = (v4_4 + edx) & 0xffffffff  # xxxx
        print v30 ^ v2c
        v4 = ((v30 ^ v2c) + v4) & 0xffffffff

        print 'round', j, v3, v4

    input1[8 * i:8 * i + 4] = p32(v3)
    input1[8 * i + 4:8 * (i + 1)] = p32(v4)
    str8_1=str(input1)
    return str8_1
def block2(str8):
    input1 = bytearray(str8)
    for i in range(0,8):
        input1[i]=input1[1]^i
    return str(input1)
des = ('52B8137F358CF21B'+'F46386D2734F1E31').decode('hex')
print len(des)

print block('12345678').encode('hex')
des1 = '5b90ef3f91b58fe6'.decode('hex')
print re_all(bytearray(des))
```