第三届上海大学生网络安全大赛 junkcode writeup

原创

Flying_Fatty 于 2018-02-05 17:20:15 发布 1690 收藏 1
分类专栏： CTF之旅 reverse 文章标签： CTF

这个题题目名字是junkcode，给人的第一反应是要去除花指令～～～

其实做题的过程和花指令并没有半毛钱关系，只是干扰了IDA的使用，从而无法进行静态分析而已

因为重点思路总是习惯性的放在了去花上，导致了其实很简单的算法，没有跟踪到数据的起始和目标节点，浪费了很多分析时间。

首先运行，看到error in open flag.

打开IDA，静态分析

```
.rdata:00407304 a_Flag          db './flag',0           ; DATA XREF: .text:00403054↑o
.rdata:0040730B                 align 4
.rdata:0040730C aErrorInOpenFla db 'error in open flag.',0
.rdata:00407320 aS_2hhx         db '%s%.2hhx',0
```

```
.text:0040303E                  push    0B8h
.text:00403043                  lea     ecx, [ebp-958h]
.text:00403049                  call    sub_4038A0
.text:0040304E                  push    1
.text:00403050                  push    40h
.text:00403052                  push    1
.text:00403054                  push    offset a_Flag   ; "./flag"
.text:00403059                  lea     ecx, [ebp-958h]
.text:0040305F                  call    sub_4039F0
.text:00403064                  mov     dword ptr [ebp-4], 0
.text:0040306B                  lea     ecx, [ebp-828h]
.text:00403071                  call    sub_401E00
.text:00403076                  mov     byte ptr [ebp-4], 1
.text:0040307A                  nop
.text:0040307B                  nop
.text:0040307C                  nop
.text:0040307D                  pusha
.text:0040307E                  jmp     loc_4030B4
.text:0040307E ; ---------------------------------------------------------------------------
.text:00403083                  db 8Bh
.text:00403084                  dd 68FF6AECh, 112233h, 33221168h, 0A16400h, 50000000h
.text:00403084                  dd 258964h, 58000000h, 0A364h, 58580000h, 0E88B5858h, 401050B8h
.text:00403084                  dd 0C3E85000h
.text:00403084 ; ---------------------------------------------------------------------------
```

这就是花指令的作用，看起来心烦，不知道程序在干什么。于是OD来动态调试

```
011730BC   ?   B0 08          mov al, 0x8
011730BE   ?   0000            add byte ptr ds:[eax], al
011730C0   .   0FB6C0          movzx eax, al
011730C3   .   85C0            test eax, eax
011730C5   .v  75 29           jnz short juckcode.011730F0
011730C7   .v  0F8E 0700000    jle juckcode.011730D4
011730CD   .v  0F85 0100000    jnz juckcode.011730D4
011730D3       E8              db E8
011730D4   >   68 0C731701     push juckcode.0117730C          ASCII 65,"rror in open flag."
011730D9   .   8B0D 0471170    mov ecx, dword ptr ds:[<&MSVCP140.std::cout>]   msvcp140.std::cout
011730DF   .   51              push ecx                        juckcode.<ModuleEntryPoint>
011730E0   .   E8 EB1F0000     call juckcode.011750D0
011730E5   .   83C4 08         add esp, 0x8
011730E8   .   6A 00           push 0x0                        status = 0x0
011730EA   .   FF15 D071170    call near dword ptr ds:[<&api-ms-win-crt-runtime-l1-1   exit
```


根据字符串提示以及exit的退出，我们猜测，程序需要读取名为flag的内容，从而与flag.enc的内容进行运算后的比对。

于是在当前目录下新建个flag文件，写入任意字符，成功绕过检验。得知我们需要填入的就是flag

于是，跟随着一堆花指令，我们看到了这个

```
011733D5   >   61              popad
011733D6   .   8D8D D8F7FFF    lea ecx, dword ptr ss:[ebp-0x828]
011733DC   .   E8 7FE8FFFF     call juckcode.01171C60
011733E1   .   50              push eax
011733E2   .   8D8D D8F7FFF    lea ecx, dword ptr ss:[ebp-0x828]
011733E8   .   E8 D3040000     call juckcode.011738C0
011733ED   .   50              push eax
011733EE   .   8D95 90F7FFF    lea edx, dword ptr ss:[ebp-0x870]
011733F4   .   52              push edx
011733F5   .   E8 56DFFFFF     call juckcode.01171350
011733FA   .   83C4 0C         add esp, 0xC
011733FD   .   C645 FC 05      mov byte ptr ss:[ebp-0x4], 0x5
01173401   .   33C0            xor eax, eax
01173403   .   85C0            test eax, eax
01173405   .v  0F8E 0700000    jle juckcode.01173412
0117340B   .v  0F85 0100000    jnz juckcode.01173412
01173411       E8              db E8
01173412   >   C685 F0FBFFF    mov byte ptr ss:[ebp-0x410], 0x0
01173419   .   68 FF030000     push 0x3FF                      n = 3FF (1023.)
0117341E   .   6A 00           push 0x0                        c = 00
01173420   .   8D85 F1FBFFF    lea eax, dword ptr ss:[ebp-0x40F]
01173426   .   50              push eax                        s = 0000001A
01173427   .   E8 DC360000     call <jmp.&VCRUNTIME140.memset>  memset
0117342C   .   83C4 0C         add esp, 0xC
0117342F   .   C785 A4F6FFF    mov dword ptr ss:[ebp-0x95C], 0x0
```


memset告诉我们，已经开始了初始化，所以很快要开始进行计算和验证判断了

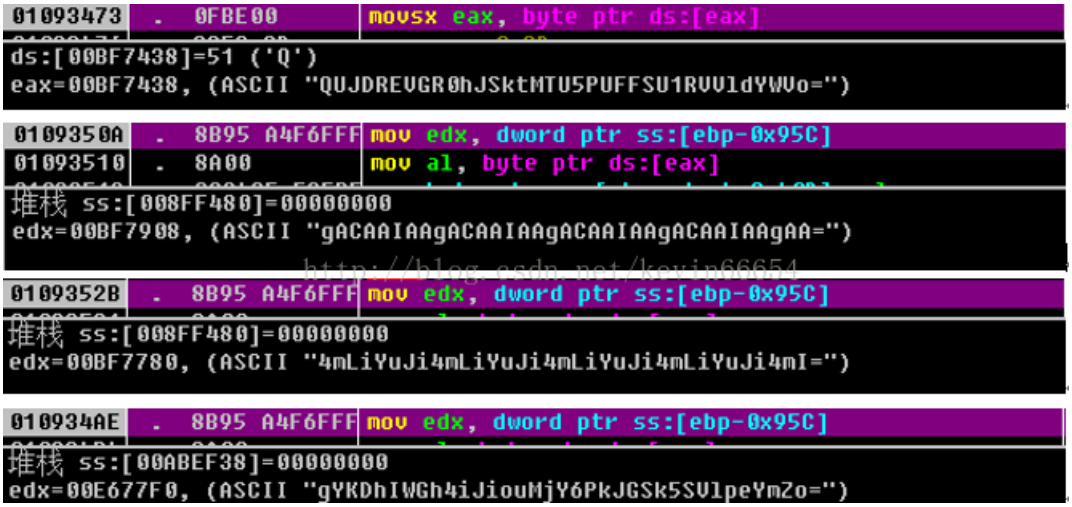
看到了这个，一个=，马上想到base64加密

```
01173473   .   0FBE00          movsx eax, byte ptr ds:[eax]
01173476   .   83F8 3D         cmp eax, 0x3D
01173479   .v  74 21           je short juckcode.0117349C
0117347B   .   8B8D A4F6FFF    mov ecx, dword ptr ss:[ebp-0x95C]
01173481   .   51              push ecx
01173482   .   8D8D C0F7FFF    lea ecx, dword ptr ss:[ebp-0x840]
01173488   .   E8 F3E7FFFF     call juckcode.01171C80
0117348D   .   8B95 A4F6FFF    mov edx, dword ptr ss:[ebp-0x95C]
01173493   .   8A00            mov al, byte ptr ds:[eax]
01173495   .   888495 F0FBF    mov byte ptr ss:[ebp+edx*4-0x410], al
0117349C   >   8B8D A4F6FFF    mov ecx, dword ptr ss:[ebp-0x95C]
011734A2   .   51              push ecx
ds:[00287278]=51 ('Q')
eax=00287278, (ASCII "QUJDREVGR0hJSktMTU5PUFFSU1RUVldYWVo=")
```


验证一下，解密之后是我输入的ABCDEFGHIJKLMNOPQRSTUVWXYZ

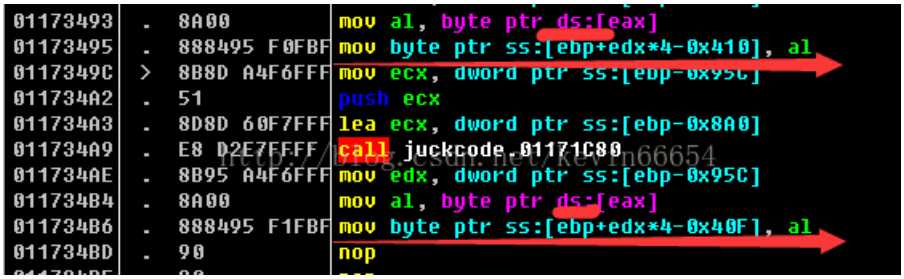
接下来是自己经验不足，做题太少，分析习惯不好而导致的多次重复分析浪费时间

接下来会在EAX中找到如下的三个重复字符串



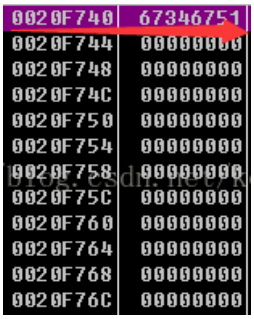
经过多次调试，发现，除了第一个字符串是根据我输入的flag经过base64加密从而生成的字符串外，其他三个字符串都是常数字符串！

我关注的重点一直放在了这三个字符串是使用什么算法生成的，因为一直没能找到，尝试在数据窗口下断一直错误。但是，其实，关注的重点应该是，这四个字符串合起来一直在做什么



注意到这个，数据的转移时从ds到了ss，也就是说，从数据窗口进入了堆栈！

那么我们需要观察循环的次数以及规律



很简单的把四个字符串的首字符连接了起来

到了3550这里就是当前循环的终止点

那么去我们的ss段也就是局部变量的地方查看一下

```
006BF6C8  67346751
006BF6CC  416D5955
006BF6D0  434C4B4A
006BF6D4  41694444
006BF6D8  41596852
006BF6DC  49754945
006BF6E0  414A5756
006BF6E4  41694747
006BF6E8  67346852
006BF6EC  416D3430
006BF6F0  434C6968
006BF6F4  41694A4A
006BF6F8  41596953
006BF6FC  49756F6B
006BF700  414A7574
```

可以在堆栈段看到数据，以字符串形式查看就是这样

```
堆栈地址=006BF6C8, (ASCII "Qg4gUYmAJKLCDDiARhYAEIuIVWJAGGiARh4g04mAhiLCJJiASiYAkouItuJAMMiATj4gUYmA56LCPPiAUkYAFJuIFGJASSiAUk4g")
ecx=006BF298
跳转来自 0106345B
```


然后一路没有发现太多的有特征的东西，直到这个字符串

```
00A736D7  .  0FB600        movzx eax, byte ptr ds:[eax]
00A736DA  .  50            push eax
00A736DB  .  8D8D F0F7FFF  lea ecx, dword ptr ss:[ebp-0x810]
00A736E1  .  51            push ecx
00A736E2  .  68 2073A700   push juckcode.00A77320                ASCII 25,"s%.2hhx"
00A736E7  .  68 00040000   push 0x400
00A736EC  .  8D95 F0F7FFF  lea edx, dword ptr ss:[ebp-0x810]
00A736F2  .  52            push edx
00A736F3  .  E8 A8240000   call juckcode.00A75BA0
```


这里是把一个字符按照16进制的形式转化。比如 ‘a’ = 0x61，则转化为61

所以根据这个lea edx , dword ptr ss:[ebp - 0x810]，可以找到我们得到的中间数据保存在了这里

```
00B7F464  65303234
00B7F468  00003032
00B7F46C  00000000
00B7F470  00000000
00B7F474  00000000
00B7F478  00000000
00B7F47C  00000000
00B7F480  00000000
00B7F484  00000000
00B7F488  00000000
```

数据来源在这里

```
地址      HEX 数据                                          ASCII
00EBEA20  42 0E 20 51 89 80 24 A2 C2 0C 38 80 46 16 00 10  B■ Q墅$18..8■F■.■
00EBEA30  8B 88 55 62 40 18 68 80 46 1E 20 D3 89 80 86 22  嫻Ub@■h■F■ 訊■?
00EBEA40  C2 24 98 80 4A 26 00 92 8B 88 B6 E2 40 30 C8 80  ?榍J&.拫垹釦0葺
00EBEA50  4E 3E 20 51 89 80 E7 A2 C2 3C F8 80 52 46 00 14  N> Q墅纰?鴢RF.■
00EBEA60  9B 88 14 62 40 49 28 80 52 4E 20 D7 99 80 45 22  洂■b@I(■RN 讬■E"
00EBEA70  C2 55 58 80 56 56 00 96 9B 88 75 E2 40 61 88 80  耆X■VV.柜坲釦a坱
00EBEA80  5A 6E 20 55 99 80 A2 82 00 00 00 00 B8 CB EB 00  Zn U櫬  ....杆?
00EBEA90  31 91 39 62 05 AC 00 00 58 B9 EB 00 D0 E9 EB 00  1?b¥?.X闰.虚?
00EBEAA0  00 00 00 00 E8 ED EB 00 19 00 00 00 A0 68 FC 73  ....栓?■...燐靨
```

这个就很好理解了，把0x42的数据转成两位，变成了0x3432~~~

那么我们需要知道420E这一堆字符串是从哪里来的

```python
import base64
s = 'Qg4g'
s = base64.b64decode(s)
for i in s:
    print hex(ord(i))[2:]
```

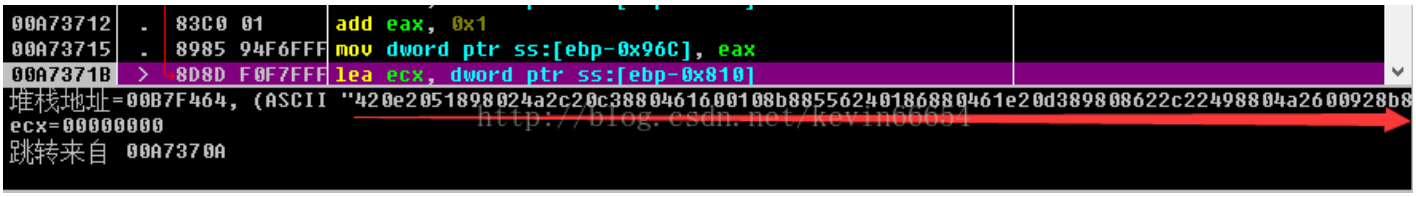
```
42
e
20
```

合理的脑洞猜测：这个题既然使用了base64的加密，那么很可能顺手调用个解密（于是我们可以猜测，是不是用了base32和base64的很多功能）

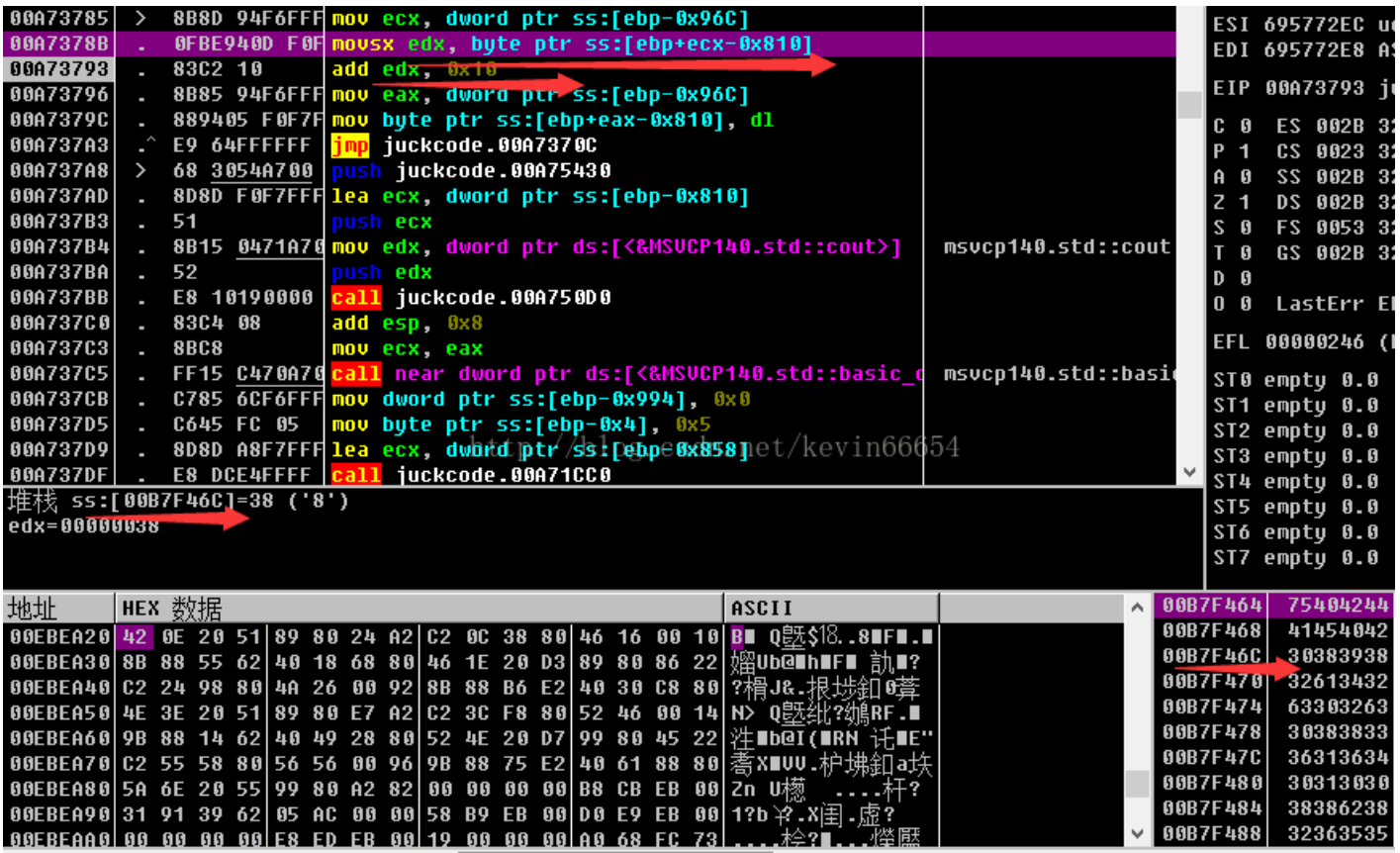
跟踪多次后，发现这里是循环终止条件

就是某个栈中的局部循环变量和字符串长度做比较

运行到了这里发现思路暂时正确



然后到下一步处理



就是每一个位置加上0x10

然后看到这里

就得到了这个奇怪的字符串

所以逆向这个简单算法就好

正向：把flag用base64进行加密，用加密后的字符串+三个常数字符串进行拼接，再base64解密，字符转化，每个字符增加0x10

逆向过程直接上代码

```
def hextonum(x):
 if x>='A' and x<='F':
  return ord(x) - ord('A') + 10
 elif x>='a' and x<='f':
  return ord(x) - ord('a') + 10
 else:
  return ord(x) - ord('0')

#hextonum(x) <---> int(x,16)

import base64
s = 'FFIF@@IqqIH@sGBBsBHFAHH@FFIuB@tvrrHHrFuBD@qqqHH@GFtuB@EIqrHHCDuBsBqurHH@EuGuB@trqrHHCDuBsBruvHH@FFIF@@
a = ''
for i in xrange(0,len(s),2):
 n1 = chr(ord(s[i]) - 0x10)
 n2 = chr(ord(s[i + 1]) - 0x10)
 number = 16 * int(n1,16) + int(n2,16)
 #m = n1 + n2
 #number = int(m,16)
 a += chr(number)
a = base64.b64encode(a)
b = ''
for i in xrange(0,len(a),4):
 b += a[i]
for i in xrange(0,4 - len(b) % 4):
 b += '='
print base64.b64decode(b)
```

参考链接：

https://www.52pojie.cn/thread-658440-1-1.html

http://www.freebuf.com/articles/rookie/153078.html