CTF小白学习笔记(Reverse)-i春秋 juckcode

原创

I still … 于 2020-11-15 16:01:00 发布 286 收藏

分类专栏： CTF的writeup 文章标签： 信息安全

本文链接：https://blog.csdn.net/qq_44370676/article/details/109700516

这道题主要考察花指令和动态调试。(看了很多大神的wp才看出门道)

动态调试也是看着别人的wp慢慢调出来的（还是太菜了）

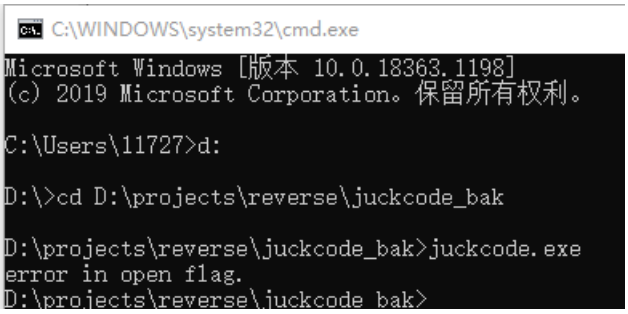
对花指令不太了解的朋友可以参考以下视频：

IDA如何patch掉花指令？

大纲

一.解题过程

拿到题目后解压，发现一个是flag.enc，一个exe，运行exe。



去除花指令

提示打不开flag文件，用ida打开看看。

```
.text:00403010
.text:00403010 ; int __cdecl main(int argc, const char **argv, const char
.text:00403010 _main:                                  ; CODE XREF: __scr
.text:00403010                 push    ebp
.text:00403011                 mov     ebp, esp
.text:00403013                 push    0FFFFFFFFh
.text:00403015                 push    offset sub_406D03
.text:0040301A                 mov     eax, large fs:0
.text:00403020                 push    eax
.text:00403021                 sub     esp, 9D0h
.text:00403027                 mov     eax, ___security_cookie
.text:0040302C                 xor     eax, ebp
.text:0040302E                 mov     [ebp-10h], eax
.text:00403031                 push    ebx
.text:00403032                 push    esi
.text:00403033                 push    edi
.text:00403034                 push    eax
.text:00403035                 lea     eax, [ebp-0Ch]
.text:00403038                 mov     large fs:0, eax
.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 aFlag    ; "./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]
```


花指令1

```
loc_4030B4:                            ; CODE XREF: .text:0040307E↑j
                popa
                lea     ecx, [ebp-958h]
                call    sub_403970
                movzx   eax, al
                test    eax, eax
                jnz     short loc_4030F0
                jle     near ptr loc_4030D3+1
                jnz     near ptr loc_4030D3+1

loc_4030D3:                            ; CODE XREF: .text:004030C7↑j
                                       ; .text:004030CD↑j
                call    near ptr 40B33D40h
                add     [ebx+4071040Dh], cl
                add     [ecx-18h], dl
                jmp     short near ptr loc_403100+2
; ---------------------------------------------------------------------------
                align 4
                dd 8C48300h, 15FF006Ah
                dd offset __imp_exit
```

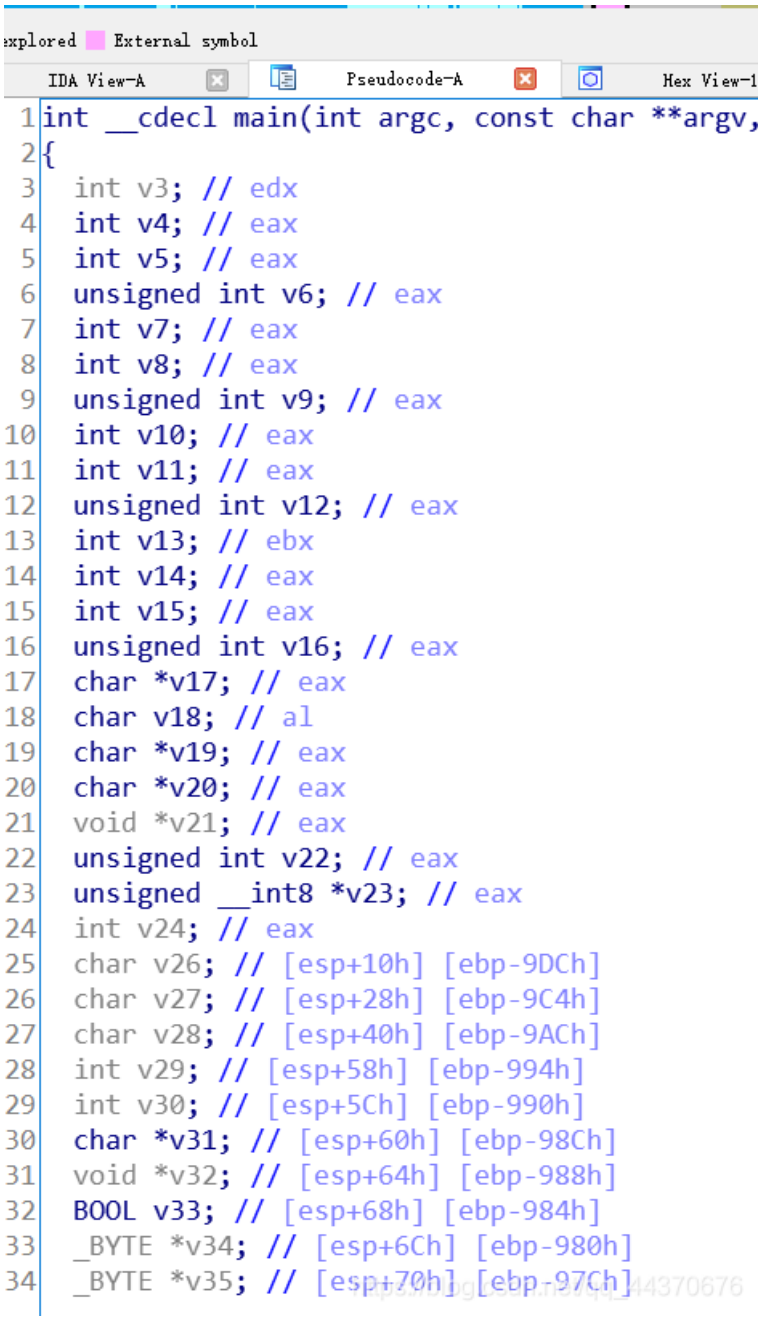

花指令2

```
30F2                test    eax, eax
30F4                jle     near ptr loc_403100+1
30FA                jnz     near ptr loc_403100+1
3100
3100 loc_403100:                            ; CODE XREF: .text:004030F4↑j
3100                                        ; .text:004030FA↑j ...
3100                call    near ptr 0F818C692h
3100 ; ---------------------------------------------------------------------------
3105                db 2 dup(0FFh), 52h
3108                dd 0F6A8858Dh, 0E850FFFFh, 22FCh, 0F08C483h, 78Eh, 1850F00h
3108                dd 0E8000000h, 0F7D88D8Dh, 31E8FFFFh, 50FFFFEBh, 0F7D88D8Dh
3108                dd 85E8FFFFh, 50000007h, 0F7C08D8Dh, 0E851FFFFh, 0FFFFE208h
3108                dd 0C60CC483h, 9002FC45h, 0E9609090h, 31h, 0FF6AEC8Bh
3108                dd 11223368h, 22116800h, 0A1640033h, 0
316C                dd 25896450h, 0
3174                dd 0A36458h, 58000000h, 8B585858h, 1050B8E8h, 0E8500040h
3174                dd 85C761C3h, 0FFFFF690h, 0
3194                dd 958B0FEBh, 0FFFFF690h, 8901C283h, 0FFF69095h, 0D88D8DFFh
3194                dd 0E8FFFFF7h, 0FFFFEAB0h, 0F6908539h, 2E73FFFFh, 0F690858Bh
3194                dd 8D50FFFFh, 0FFF7D88Dh, 0EAB6E8FFh, 8589FFFFh, 0FFFFF684h
3194                dd 0F6848D8Bh, 0BE0FFFFFh, 40C28311h, 0F684858Bh, 1088FFFFh
3194                dd 9090B0EBh, 31E96090h, 8B000000h, 68FF6AECh, 112233h
3194                dd 33221168h, 0A16400h, 50000000h, 258964h, 58000000h
3194                dd 0A364h, 58580000h, 0E88B5858h, 401050B8h, 0C3E85000h
3194                dd 0D88D8D61h, 0E8FFFFF7h, 0FFFFEA34h, 0D88D8D50h, 0E8FFFFF7h
```

发现一大堆花指令，有个字符串./flag，不过error in open flag提示找不到引用，所以猜测程序的输入可能是flag文件（不是flag.enc）,我们先创建一个flag文件，内容写上1234567890。

关于去花指令，上面的视频有讲到，不过这里的花指令实在是多，只要把main函数里的花指令去完就可以了。（这里纯手动patch，去花脚本网上有很多，不过好像不管用）

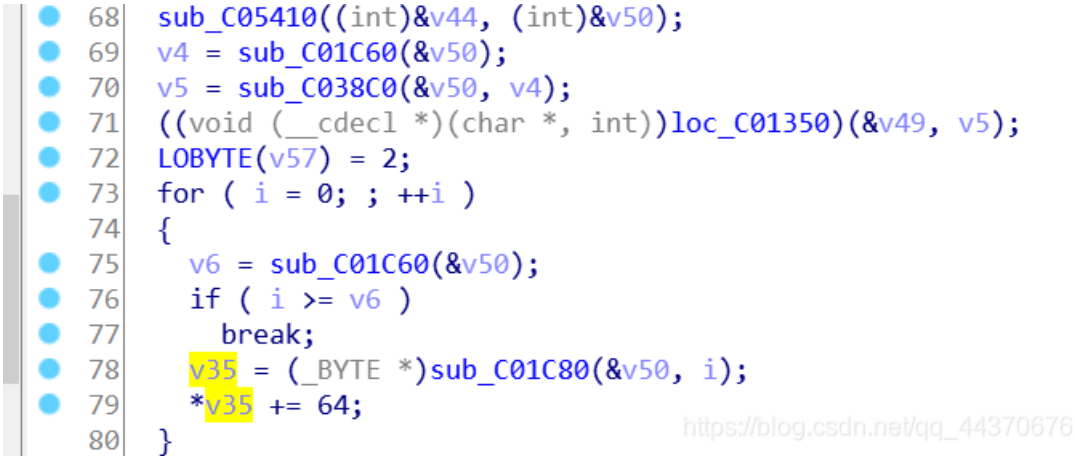
花指令去完后，在main函数开头右键->create function，就可以反编译main函数了。



动态调试过程

创建4个关键字符串

如图所示，接下来就是漫长的动态调试过程，具体过程不再详述，我们来看看关键部分。



这里的v50即输入的字符串，73行的这个for循环主要功能就是把输入字符串的ascii码 + 64赋值给v35。然后是下一个for循环

```
84  LOBYTE(v57) = 3;
85  for ( j = 0; ; ++j )
86  {
87    v9 = sub_C01C60(&v50);
88    if ( j >= v9 )
89      break;
90    v34 = (_BYTE *)sub_C01C80(&v50, j);
91    *v34 <<= 7;
92  }
```


这里，将v35字符串每个字符左移7位赋值给v34(别被v50迷惑了，这也是动态调试才能看出来)。然后是下一个for循环

```
LOBYTE(v57) = 4;
for ( k = 0; ; ++k )
{
  v12 = sub_C01C60(&v50);
  if ( k >= v12 )
    break;
  v13 = *(char *)sub_C01C80(&v50, k) - 158;
  *(_BYTE *)sub_C01C80(&v50, k) = v13;
}
```


这里，将v34每个字符-158赋值给v13

至此，程序以及创建了4个字符串， v50(输入,1234567890)， v35(v50 + 64, qrstuvwxyp), v34 (v34 <<7),v13(v34 - 158)。

对4个关键字符串进行base64编码，并将编码后的字符串重新组合

接下来就是如下的函数片段：

```
Src = 0;
memset(&Dst, 0, 0x3FFu);
for ( l = 0; ; ++l )
{
  v16 = sub_C01C60(&v49);
  if ( l >= v16 )
    break;
  if ( *(_BYTE *)sub_C01C80(&v49, l) != 61 )
  {
    v17 = (char *)sub_C01C80(&v49, l);
    *(&Src + 4 * l) = *v17;
  }
  v18 = *(_BYTE *)sub_C01C80(&v45, l);
  *(&Dst + 4 * l) = v18;
  v19 = (char *)sub_C01C80(&v46, l);
  v56[4 * l] = *v19;
  v20 = (char *)sub_C01C80(&v47, l);
  *(&v55 + 4 * l) = *v20;
}
sub_C01D90(&v28, &Src);
LOBYTE(v57) = 6;
```

动态调试的时候会发现：
v49的值为MTlzNDU2Nzg5MA== (v50 base64编码)
v45为cXJzdHV2d3h5cA== （v35 base64)
v46为gACAAIAAgACAAA== （v34base64）
v47为4mLiYuJi4mLiYg== (v13 base64)

src为 Mc4gTXmA… (*v49*v45*v47*v46*(v49 + 1)…)
dst指针指向src后一位，没太大用。

对重新组合过的字符串的变换

接着往下看：

```
134  memset(&v52, 0, 0x3FFu);
135  for ( m = 0; ; ++m )
136  {
137    sub_C01D90(&v27, &Src);
138    LOBYTE(v57) = 9;
139    v21 = (void *)sub_C017D0(&v26, &v27);
140    v32 = v21;
141    v22 = sub_C01C60(v21);
142    v33 = m < v22;
143    v42 = m < v22;
144    std::basic_string<char,std::char_traits<char>,std::alloca
145    LOBYTE(v57) = 8;
146    std::basic_string<char,std::char_traits<char>,std::alloca
147    if ( !v42 )
148      break;
149    v23 = (unsigned __int8 *)sub_C01C80(&v48, m);
150    sub_C05BA0((int)&v51, 1024, (int)"%s%.2hhx", &v51, *v23);
151  }
152  for ( n = 0; ; ++n )
153  {
154    v40 = &v51;
155    v31 = &v52;
156    v40 += strlen(v40);
157    v30 = ++v40 - &v52;
158    if ( n >= v40 - &v52 )
159      break;
160    *(&v51 + n) += 16;
161  }
162  v24 = sub_C050D0(std::cout, &v51);
```


这里135行的for循环中v23为对src进行base64编码后的字符串，v51为将v23转为16进制后的字符串(比如字符串’10’，ascii码分别为0x31,0x30，变成’3130’)

下一个for循环将v51每个字符 + 16,结果保存在flag.enc中，至此程序流程结束。

解密脚本

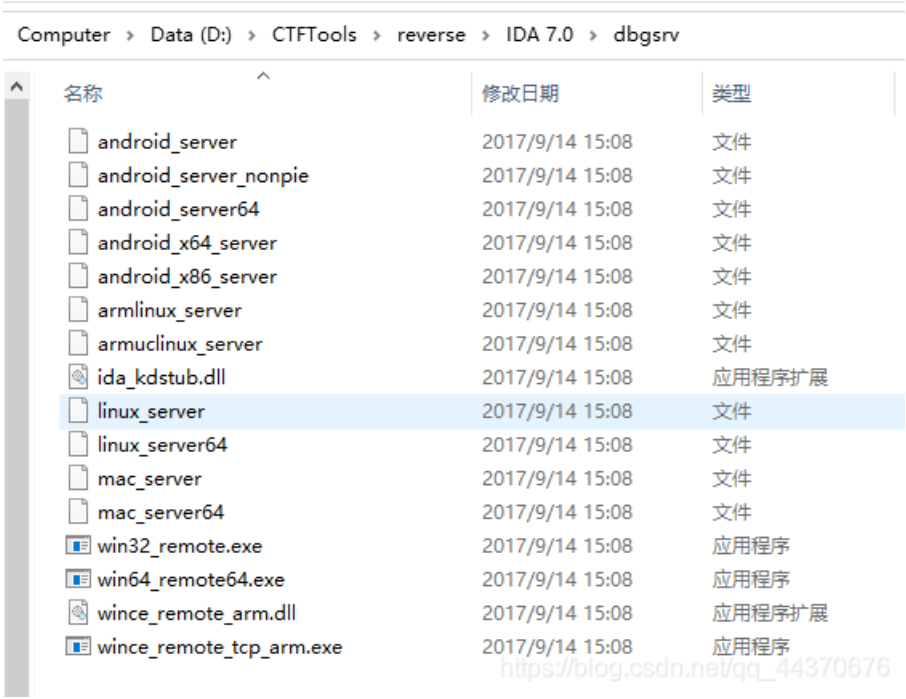
```python
import base64

message = open("../datas/flag.enc").read()[:-1]
v51 = ''.join([chr(ord(c) - 16) for c in message])
v23 = ""
v51_list = [c for c in v51]

for i in range(int(len(v51) / 2)):
    v23 += chr(int(''.join(v51_list[i * 2: i * 2 + 2]), 16))

src = base64.b64encode(v23.encode("latin-1")).decode('utf-8')
print(src)

input_b64 = src.replace("\n","")[::4]
print(input_b64)
print(len(input_b64))
input = base64.b64decode((input_b64 + '==').encode('utf-8'))
print(input)
```

解密完成后得到flag{juck_code_cannot_stop_you_reversing}

这里需要注意，其他wp大部分都是用python2写的，我这是python3写的。这里的v23包括一些不可见字符，python2和python3的base64在对不可见字符编码时，结果会不同，待会会讲一下。

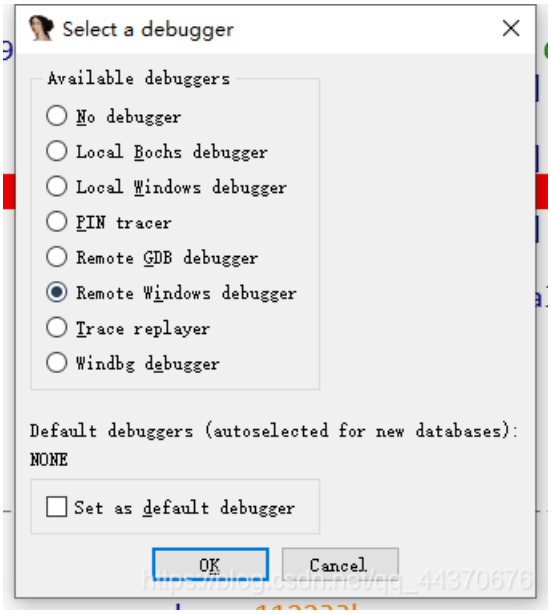
二.ida动态调试

我用的是IDA7.0, IDA7.x默认集成了F5插件，idapython等还包括一些动态调试器。在%IDAPATH%/dbgsrv文件夹下可以看到



这个程序是windows 32位的程序，所以在本机上运行win32_remote.exe.

之后在ida debugger中选择remote windows debugger



在参数选择界面中把ip地址填成127.0.0.1就行。然后就可以开始动态调试了。不得不说IDA7还是很牛逼的。

三.python2和python3的base64

这道题还有一个问题就是在解密过程中会涉及到对不可见字符的base64编码。
相关讨论可以参考：Python2/3 的 base64 对不可见字符编码结果不同
在python2中,字符串base64编码过程如下：

```
s3 = base64.b64encode(s2)
```

其中, s2,s3都是str类型的变量。

而在python3中，应该按下面方式写：

```
s3 = base64.b64encode(s2.encode("utf-8")).decode('utf-8')
```

base64.b64encode传入参数必须位bytes类型，返回值也是bytes类型。
而python3中 str.encode可以将str变成bytes。 bytes.decode可以将bytes变成str。

在python3中，我们可以做以下试验，看看0x00-0xFF中，非ascii码范围内(0x80-0xFF)的字符和ascii码范围内的字符utf-8编码的结果：

```
print('\xef'.encode('utf-8'))
print('\x30'.encode('utf-8'))
```

执行结果：

```
b'\xc3\xaf'
b'0'
```

可以看出，ascii码范围内的字符utf-8编码后字符值并未改变，而非ascii范围内的变了。这里就需要引入latin-1(ISO-8859-1)编码。

```
print('\xef'.encode('latin-1'))
print('\x30'.encode('latin-1'))
```

执行结果：

```
b'\xef'
b'0'
```

可见，这道题在对v23 base64编码前，应该用latin-1编码v23，base64过后依旧可以用utf-8解码。

```
src = base64.b64encode(v23.encode("latin-1")).decode('utf-8')
```