xnuca-Strange Interpreter-浅谈逆向小技巧

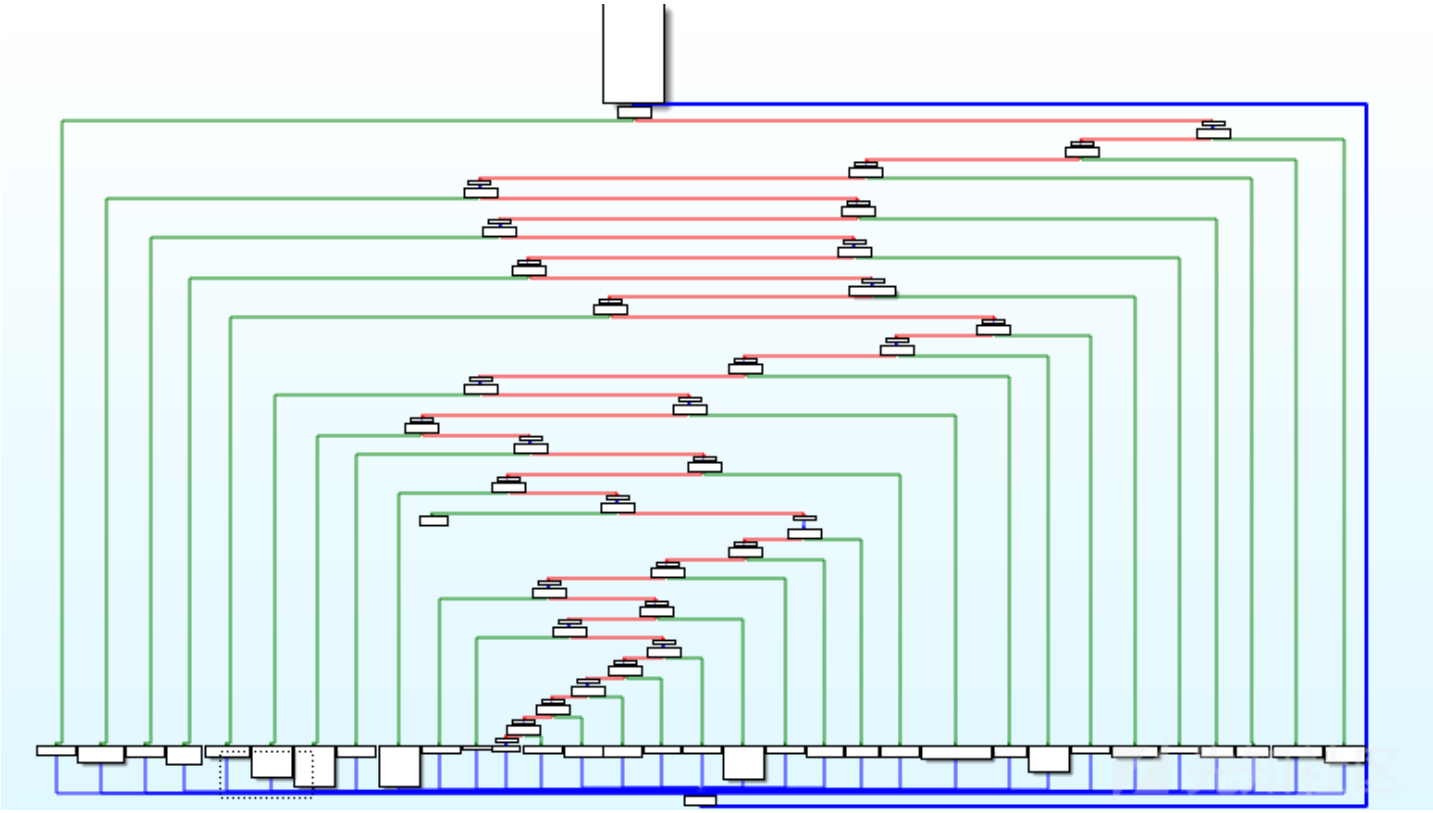
Badrer / 2018-12-01 08:15:00 / 浏览数 2829 技术文章 技术文章 顶(1) 踩(0)

前言

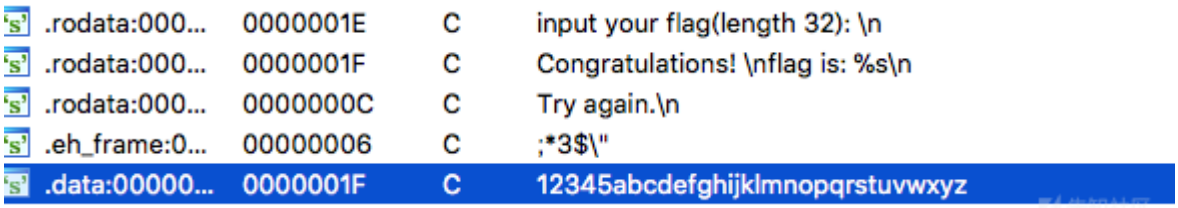
这题本来不想详细的说，因为我解此题用的比较笨的方法，但这也不失为一种方法，赛后反思总结了一下，觉得在逆向调试和分析的技巧上可以有所提高，并且最后也会提及

Strange Interpreter

通过`file`指令可知是一个64位，linux动态编译的程序，载入`IDA`可以看到程序混淆的十分严重。



不过和叹息之墙比起来，还差的远呢。
通过查看字符串，可以知道flag的长度为32，并且看到一串明文。



这段明文放在这里肯定会被使用到，交叉引用可以来到`loc_412164`基本块中

```
                        loc_412164:
B8 10 00 00 00          mov     eax, 10h
B9 00 02 00 00          mov     ecx, 200h
8B 14 25 E8 38 61+mov   edx, ds:index
83 EA 01                sub     edx, 1
89 14 25 F4 38 61+mov   ds:dword_6138F4, edx
8B 14 25 E8 38 61+mov   edx, ds:index
89 14 25 F8 38 61+mov   ds:dword_6138F8, edx
8B 14 25 F4 38 61+mov   edx, ds:dword_6138F4
89 D6                   mov     esi, edx
8B 14 B5 D0 30 61+mov   edx, ds:dword_6130D0[rsi*4]
2B 4D CC                sub     ecx, [rbp+var_34]
29 C8                   sub     eax, ecx
48 63 F0                movsxd  rsi, eax
03 14 B5 70 30 61+add   edx, dword_613070[rsi*4]
8B 04 25 F8 38 61+mov   eax, ds:dword_6138F8
89 C6                   mov     esi, eax
89 14 B5 D0 30 61+mov   ds:dword_6130D0[rsi*4], edx
8B 04 25 E8 38 61+mov   eax, ds:index
83 C0 01                add     eax, 1
89 04 25 E8 38 61+mov   ds:index, eax
C7 45 BC C1 9A 0B+mov   [rbp+var_44], 2B0B9AC1h
E9 7C 02 00 00          jmp     loc_412455
```

strings查看字符串。

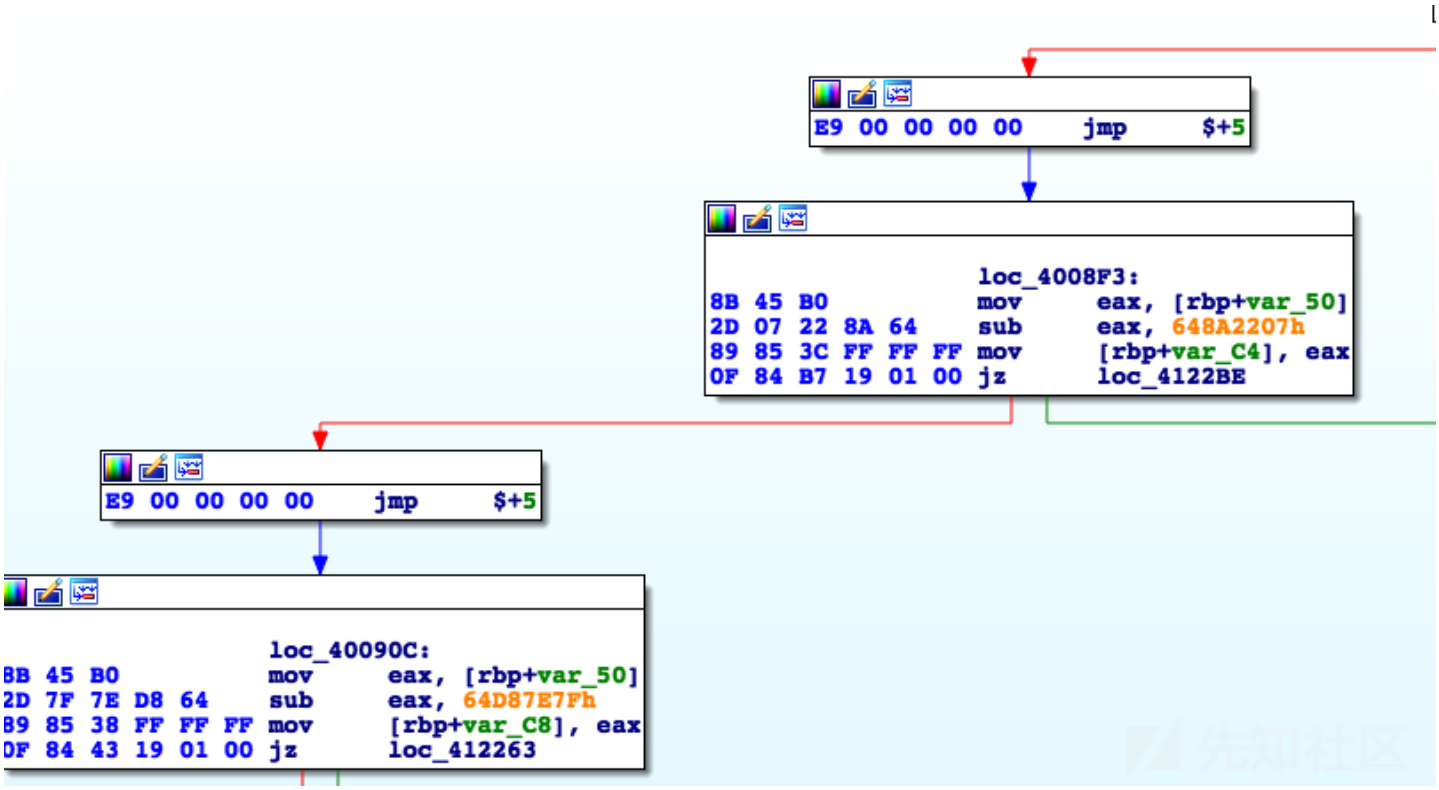
```
012345abcdefghijklmnopqrstuvwxyz
GCC: (Ubuntu 5.2.1-22ubuntu2) 5.2.1 20151010
Obfuscator-LLVM clang version 4.0.1  (based on Obfuscator-LLVM 4.0.1)
```

可以知道程序使用Obfuscator-LLVM 4.0.1编译的。

但是现在我们并不能看出什么东西，正常的逻辑是这样的，不过要是感觉敏锐一点，byte_613050是最后用来校验的数据，ds:dword_6130D0是经过加密变换后的数据，

开始

IDA是个神器，我们应该好好利用。IDA在分析完程序后，会显示CFG控制流程图，每个基本块之间的关系我们可以清楚的看到。
总揽一下代码可以看到这样那样的■■■以及使用了控制流平坦化的混淆方式，这也已经很常见了。



动态调试

假设大家都不知道llvm混淆的程序是怎样的，我们从loc_400655基本块开始看起。我们需要时刻盯着我们的输入。

```
007FFFEFD65FC0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa
007FFFEFD65FD0  61 61 61 61 61 61 61 61  61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa
007FFFEFD65FE0  00 60 D6 EF FF 7F 00 00  01 00 00 00 00 00 00 00  .`..............
```

不断的f8我们便可以来到loc_400979基本块中，其实可以发现全程只有jmp和jz两条跳转指令，从上到下的中间过程我们是无法修改的，也就是说在这个过程中，并没有

```
loc_400979:                ; read input
mov     eax, ds:index
mov     ecx, eax
movsx   eax, [rbp+rcx+var_30] ; read input
sub     eax, 20h
add     eax, 20h
mov     edx, ds:index
mov     ecx, edx
mov     ds:dword_6130D0[rcx*4], eax
mov     [rbp+var_44], 19142DA4h
jmp     loc_412455
```

可以看到它的功能是将输入的第一个字符取出，并存入到ds:dword_6130D0中，同时ecx作为下标，也就是ds:index(已重命名，最后我会附上idb文件)，可以想到后续必
F9 运行到下一个loc_4009A9基本块中,很明显是为了将ds:index++。

```
loc_4009A9:                ; index++
mov     eax, ds:index
add     eax, 1
mov     ds:index, eax
mov     [rbp+var_44], 0EC5EBE5Ch
jmp     loc_412455
```

同样的 F9 可以来到loc_40095C，比较ds:index是否大于0x20

```
loc_40095C:                ; CODE XREF: main+162↑j
mov     eax, 44FE6EDCh     ; cmp index <32
mov     ecx, 0BE6EC0D9h
cmp     ds:index, 20h
cmovb   eax, ecx
mov     [rbp+var_44], eax
jmp     loc_412455
```

其实这时已经比较明显了，但我们仍需要验证一下，再次F9再次来到loc_400979中，之后便是一个循环，此时我们可以取消先前设置的三个断点，再次F9，注意数据部分

```
loc_4009C6:                ; CODE XREF: main+2EF↑j
mov     eax, ds:index
sub     eax, 1
mov     ds:dword_6138F4, eax
mov     eax, ds:index
mov     ds:dword_6138F8, eax
mov     eax, ds:dword_6138F4
mov     ecx, eax
mov     eax, ds:dword_6130D0[rcx*4]
mov     edx, ds:dword_6138F8
mov     ecx, edx
add     eax, ds:dword_6130D0[rcx*4]
mov     ds:dword_6130D0[rcx*4], eax
mov     eax, ds:index
add     eax, 1
mov     ds:index, eax
mov     eax, ds:index
sub     eax, 1
mov     ds:dword_6138F4, eax
mov     eax, ds:index
```

此时的ds:dword_6130D0如下：

```
00006130C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00006130D0  61 00 00 00 61 00 00 00  61 00 00 00 61 00 00 00  a...a...a...a...
00006130E0  61 00 00 00 61 00 00 00  61 00 00 00 61 00 00 00  a...a...a...a...
00006130F0  61 00 00 00 61 00 00 00  61 00 00 00 61 00 00 00  a...a...a...a...
0000613100  61 00 00 00 61 00 00 00  61 00 00 00 61 00 00 00  a...a...a...a...
0000613110  61 00 00 00 61 00 00 00  61 00 00 00 61 00 00 00  a...a...a...a...
0000613120  61 00 00 00 61 00 00 00  61 00 00 00 61 00 00 00  a...a...a...a...
0000613130  61 00 00 00 61 00 00 00  61 00 00 00 61 00 00 00  a...a...a...a...
0000613140  61 00 00 00 61 00 00 00  61 00 00 00 61 00 00 00  a...a...a...a...
0000613150  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

loc_4009C6代码块非常的大，如果单步调试可能真的需要半天时间，因此我们需要找到重复的部分，我们紧盯着ds:dword_6130D0这部分数据，这是切入点，前面一大段
接下来的一大片代码都在增加index的值，我们可以快速的滑动滚轮略过，直到40F13D，出现大片奇怪的字符。

```
00040F13D mov     eax, ds:dword_6130D0[rcx*4]
00040F144 add     eax, 68h
00040F147 mov     edx, ds:dword_6138F8
00040F14E mov     ecx, edx
00040F150 mov     ds:dword_6130D0[rcx*4], eax
00040F157 mov     eax, ds:index
00040F15E add     eax, 1
00040F161 mov     ds:index, eax
00040F168 mov     eax, ds:index
00040F16F mov     ds:dword_6138F4, eax
00040F176 mov     eax, ds:index
00040F17D mov     ds:dword_6138F8, eax
00040F184 mov     eax, ds:dword_6138F4
00040F18B mov     ecx, eax
00040F18D mov     eax, ds:dword_6130D0[rcx*4]
00040F194 add     eax, 1Ch
00040F197 mov     edx, ds:dword_6138F8
00040F19E mov     ecx, edx
00040F1A0 mov     ds:dword_6130D0[rcx*4], eax
```

单步跟一会可以发现，程序将这些字符复制到了ds:dword_6130D0偏移index*4的位置

```
00613850  68 00 00 00 1C 00 00 00  00 00 00 00 00 00 00 00  h...............
00613860  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00613870  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00613880  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00613890  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
006138A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
006138B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
006138C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

同样略过重复代码，之后是大量的index--，也就是

```
040F618 mov     eax, ds:index
040F61F add     eax, 0FFFFFFFFh
040F622 mov     ds:index, eax
040F629 mov     eax, ds:index
040F630 add     eax, 0FFFFFFFFh
040F633 mov     ds:index, eax
040F63A mov     eax, ds:index
040F641 add     eax, 0FFFFFFFFh
040F644 mov     ds:index, eax
040F64B mov     eax, ds:index
040F652 add     eax, 0FFFFFFFFh
040F655 mov     ds:index, eax
040F65C mov     eax, ds:index
040F663 add     eax, 0FFFFFFFFh
040F666 mov     ds:index, eax
```

而后同样的又是将一块数据复制到ds:dword_6130D0偏移0x1d0*4处。最后来到0x411A09

```
0411A06 add     eax, 0FFFFFFFFh         ; init index
0411A09 mov     ds:index, eax
0411A10 mov     eax, ds:index
0411A17 mov     ds:dword_6138F8, eax
0411A1E mov     eax, ds:index
0411A25 add     eax, 1E0h
0411A2A mov     ds:dword_6138F4, eax
0411A31 mov     eax, ds:dword_6138F4
0411A38 mov     ecx, eax
0411A3A mov     eax, ds:dword_6130D0[rcx*4]
0411A41 mov     edx, ds:dword_6138F8
0411A48 mov     ecx, edx
0411A4A xor     eax, ds:dword_6130D0[rcx*4]
0411A51 mov     ds:dword_6130D0[rcx*4], eax
0411A58 mov     eax, ds:index
0411A5F add     eax, 1
0411A62 mov     ds:index, eax
0411A69 mov     eax, ds:index
0411A70 mov     ds:dword_6138F8, eax
0411A77 mov     eax, ds:index
0411A7E add     eax, 1E0h
```

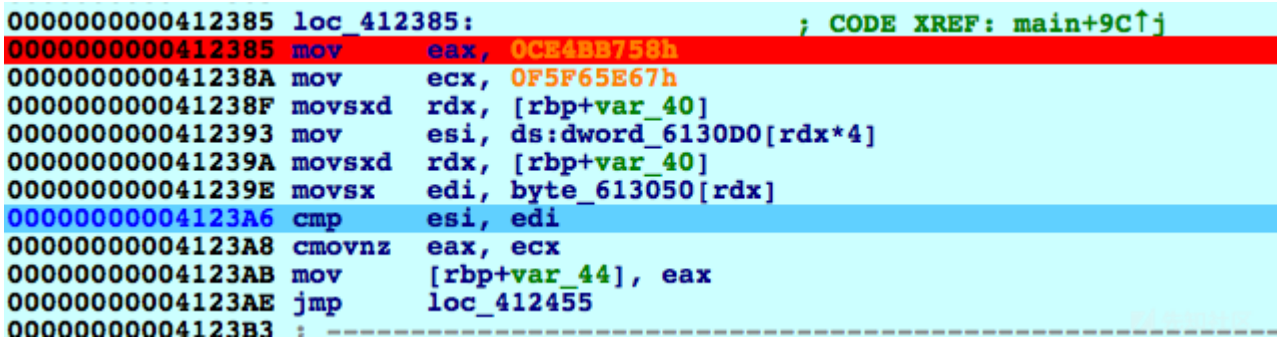
有一个很明显的xor操作，此时通过查看eax ■
rcx可知是将输入同刚刚初始化的数据进行xor，我们也可以进行一个反向验证，这很容易，因为xor是可逆操作。所以此部分代码可以略过，最后可以看到我们的输入变化如

只有前0x10字节发生了变化，因此我们便可以得到第一部分的解密脚本。

我想要讲的到这里已经差不多了，在带混淆调试时，我们不可能完全的单步调试，必须有选择的略过一些无用代码，这可以提高我们的效率，其实做逆向，看完题目后心

接下来的过程和第一部分差不多，至少方法上是一样的，因此我也不愿在这里做无用功，以上就是我的一点技巧，希望能给大家一些帮助，也希望同学们能动手调试一遍，感

最后的校验部分如图：



解密脚本如下：

```
dic = '012345abcdefghijklmnopqrstuvwxyz'
dic_list=list(dic)
xor1=[0x68,0x1C,0x7C,0x66,0x77,0x74,0x1A,0x57,0x06,0x53,0x52,0x53,0x02,0x5D,0x0C,0x5D]


xor2=[0x04,0x74,0x46,0x0E,0x49,0x06,0x3D,0x72,0x73,0x76,0x27,0x74,0x25,0x78,0x79,0x30]

xor3=[0x68,0x1C,0x7C,0x66,0x77,0x74,0x1A,0x57,0x06,0x53,0x52,0x53,0x02,0x5D,0x0C,0x5D]

print len(xor1)
flag1=""
for i in range(16):
   flag1+=chr(xor1[i]^ord(dic[i]))
print flag1
flag2=""
for i in range(16):
   j=i+16
   flag2+=chr(xor2[i]^ord(dic[j])^xor3[i]^ord(dic[i]))
print flag2
print flag1+flag2
```

使用llvm编译自己的程序

obfuscator-llvm

```
$ git clone -b llvm-4.0 https://github.com/obfuscator-llvm/obfuscator.git
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ../obfuscator/
$ make -j7
```

ubuntu下照着官方的流程来一遍就可以了。

使用如下方式进行编译：

```
$ path_to_the/build/bin/clang test.c -o test -mllvm -sub -mllvm -fla
```

这样我们便可以自己给自己出道llvm的题目了，是不是很刺激呢。

总结

其实解决这道题目的方法有很多，这里只是想分享一下自己的一些小思路，希望自己在以后遇到需要头铁分析的题目时能善于利用技巧，提高逆向效率。最后也欢迎大家一起

附件链接:https://pan.baidu.com/s/1hsf1fkIdwDiMiXBY6J76KQ 密码:kvs8

上一篇：APT28样本分析之宏病毒分析 下一篇：X-NUCA'2018 线上专题赛...

1. 0 条回复
   - 动动手指，沙发就是你的了！