171227 逆向-JarvisOJ（Shell）

原创

奈沙夜影 于 2017-12-28 02:32:35 发布 784 收藏

分类专栏： CTF

本文链接：https://blog.csdn.net/whklhhhh/article/details/78917878

1625-5 王子昂 总结《2017年12月27日》【连续第453天总结】

A. JarvisOJ-Shell

B.

这个64位的upx+golang真是够折腾人的..

首先查壳，发现PEiD直接罢工了，我还纳闷儿，明明都能运行了咋还不是有效的PE文件捏

然后ExeInfoPE才告诉我这货是64位文件

拖入IDA发现什么都没识别出来

但是通过区段名能看出来是UPX壳

然而掏UPX程序出来却报checksum error

还好UPX手脱难度不大

按照ESP定律轻松找到OEP

接下来就只能一个函数一个函数日了

找到输入函数sub_4643F0，但是一次只能接受8个字符，好像还有一些缓冲区什么的

来回断点调试，发现封装函数sub_464D00，它出来以后内存中就有完整的输入字符串了

地址	十六进制																ASCII
000000C0420402A0	31	32	33	34	35	36	37	38	32	00	32	00	32	00	00	00	123456782.2.2...
000000C0420402B0	33	00	33	00	33	00	00	00	34	00	34	00	34	00	00	00	3.3.3...4.4.4...
000000C0420402C0	35	00	35	00	35	00	00	00	36	00	36	00	36	00	00	00	5.5.5...6.6.6...
000000C0420402D0	37	00	37	00	37	00	00	00	38	00	38	00	38	00	00	00	7.7.7...8.8.8...
000000C0420402E0	39	00	39	00	39	00	00	00	30	00	30	00	30	00	00	00	9.9.9...0.0.0...
000000C0420402F0	31	32	33	34	35	36	37	38	39	30	31	32	33	34	35	36	1234567890123456
000000C042040300	31	00	31	00	31	00	00	00	32	00	32	00	32	00	00	00	1.1.1...2.2.2...
000000C042040310	33	00	33	00	33	00	00	00	34	00	34	00	34	00	00	00	3.3.3...4.4.4...
000000C042040320	35	00	35	00	35	00	00	00	36	00	36	00	36	00	00	00	5.5.5...6.6.6...
000000C042040330	0D	00	0D	00	0D	00	00	00	00	00	00	00	00	00	00	00	....
000000C042040340	31	32	33	34	35	36	37	38	39	30	31	32	33	34	35	36	1234567890123456
000000C042040350	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000C042040360	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

对首字符下内存断点，跑起来~

然后就结束了(:3」∠)

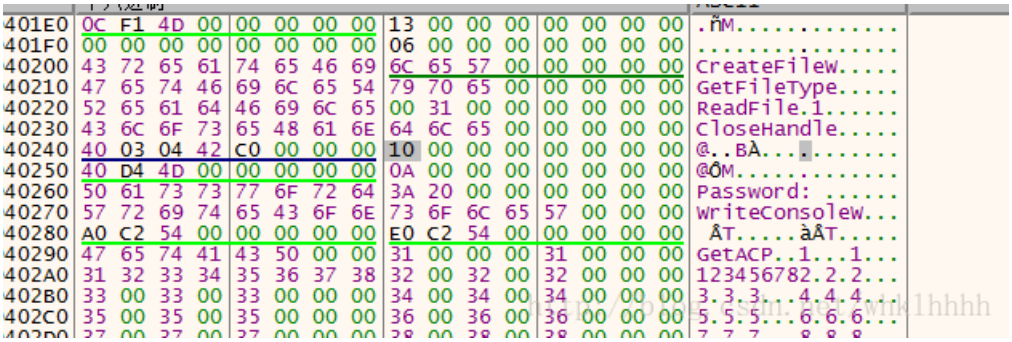
难道是之间听说过的Ring0级的内核调用API，所以Ring3的调试器断不到嘛？

不能这么轻易放弃，再思考一下

除了直接校验input内容以外，还有可能进行先行其他校验
比如首尾字符、哈希、长度校验等等

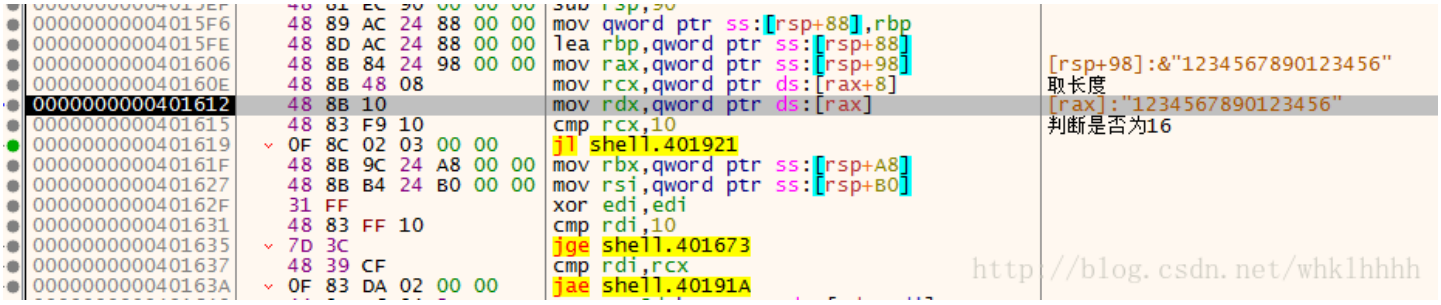
考虑到读取断点没有断到，猜测是长度校验
于是在函数外单步运行调试，观察寄存器中出现的跟测试字符串长度相等的值，注意它被保存到内存的哪里去了

反复改变长度调试以后，最终确定了这里



上面的CreateFileW、GetFileType和ReadFile有点让人在意呢~
WriteConsoleW很明显就是在向控制台输出内容的API了
所以说没事翻翻内存的上下文也挺有好处的233毕竟堆都在一起

言归正传，找到长度以后，下读取断点再跑，果然断到一处



这个地方将长度与16进行比较，大于等于才可继续

估计这里就是核心check函数了，直接到IDA中定位0x401612

按C令IDA将数据作为指令Code来分析，再向上翻到函数开头0x4015D0按P来CreateFunction，最后F5反编译完成

```
  ((void (__fastcall *)(__int64, __int64, __int64, __int64))qword_453290[136])(v10, a2, a3, a4);
}
len = v29[1];
v12 = *v29;
if ( len < 16 )
{
  v27 = "Access Denied!CertCloseStoreCreateProcessWCryptGenRandomFindFirstFileWFormatMessageWGC assist waitGC worker initGetConsole
  v28 = 14i64;
  result = sub_404290();
}
else
{
  i = 0i64;
  do
  {
    if ( i >= len || i >= v31 )
      sub_427BA0();
    if ( (*(unsigned __int8 *)(v12 + i) ^ (unsigned __int64)*(unsigned __int8 *)(v30 + i)) != qword_541C40[i] )// 异或比较
    {
      v25 = "Access Denied!CertCloseStoreCreateProcessWCryptGenRandomFindFirstFileWFormatMessageWGC assist waitGC worker initGetCor
      v26 = 14i64;
      return sub_404290();
    }
    ++i;
  }
  while ( (signed __int64)i < 16 );
  sub_43BF00((__int64)&unk_4BBF40, v12);
  v14 = 0i64;
```


这儿就比较明显了Access Denied就是输入错误的提示

在else以后进行了常规的异或变换和比较

根据len来自v29[1]，猜测v12比较可能是input内容

那么问题来了，v30是啥玩意儿？

向上索引也啥都没有

PS：Golang这传参机制真谜，怎么把整个堆栈一股脑全塞进去了OTZ亲切的栈帧去哪儿啦

没办法~只好动态调试来看咯

```
0000000000401640   44 0F B6 04 3A        movzx r8d,byte ptr ds:[rdx+rdi]      rdx+rdi*1:
0000000000401645   48 39 F7              cmp rdi,rsi
0000000000401648   0F 83 CC 02 00 00     jae shell.40191A
000000000040164E   4C 8D 0D EB 05 14 00  lea r9,qword ptr ds:[541C40]
0000000000401655   4D 8B 0C F9           mov r9,qword ptr ds:[r9+rdi*8]
0000000000401659   44 0F B6 14 3B        movzx r10d,byte ptr ds:[rbx+rdi]
RIP 000000000040165E   4D 31 C2          xor r10,r8
0000000000401661   4D 39 CA              cmp r10,r9
0000000000401664   0F 85 64 02 00 00     jne shell.4018CE
000000000040166A   48 FF C7              inc rdi
000000000040166D   48 83 FF 10           cmp rdi,10
0000000000401671   7C C4                 jl shell.401637
0000000000401673   48 8D 0D C6 A8 0B 00  lea rcx,qword ptr ds:[4BBF40]
000000000040167A   48 89 0C 24           mov qword ptr ss:[rsp],rcx
000000000040167E   48 C7 44 24 08 10 00  mov qword ptr ss:[rsp+8],10           [rsp+8]:&"
0000000000401687   48 C7 44 24 10 10 00  mov qword ptr ss:[rsp+10],10
0000000000401690   E8 6B A8 03 00        call shell.43BF00
0000000000401695   48 8B 44 24 18        mov rax,qword ptr ss:[rsp+18]
000000000040169A   48 89 44 24 48        mov qword ptr ss:[rsp+48],rax
000000000040169F   48 8B 4C 24 20        mov rcx,qword ptr ss:[rsp+20]
00000000004016A4   48 89 4C 24 38        mov qword ptr ss:[rsp+38],rcx
00000000004016A9   48 8B 54 24 28        mov rdx,qword ptr ss:[rsp+28]
00000000004016AE   48 89 54 24 40        mov qword ptr ss:[rsp+40],rdx
00000000004016B3   48 8B 9C 24 98 00 00  mov rbx,qword ptr ss:[rsp+98]       [rsp+98]:&
```

```
r10=E8 'è'
r8=31 '1'

UPX0:000000000040165E shell.exe:$165E #0
```

内存 1 | 内存 2 | 内存 3 | 内存 4 | 内存 5 | 监视 1 | [x=] 局部变量 | 结构体

地址	十六进制																ASCII
000000C0420ECBE0	E8	7E	B7	6D	A5	95	98	1A	FD	EE	7D	F8	34	62	9A	69	è~·m¥...ýî}ø4b.i
000000C0420ECBF0	2F	0A	7D	50	01	38	28	67	43	45	0C	00	47	43	6B	12	/.}P.8(gCE..GCk.
000000C0420ECC00	79	F9	C6	62	47	56	3E	0A	8F	C6	C5	7F	90	A5	C2	E8	yùÆbGV>..ÆÅ..¥Âè
000000C0420ECC10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

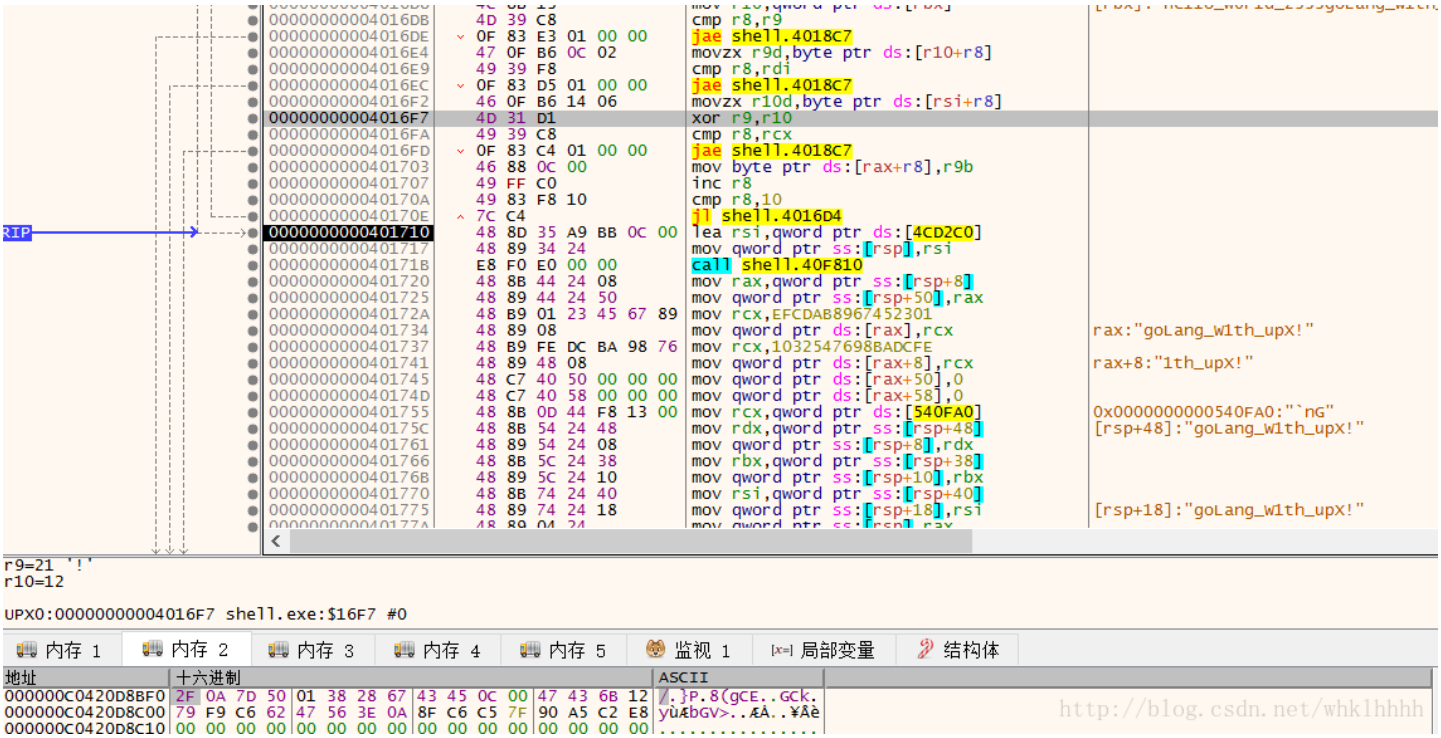
这里就是对应的汇编了

根据rbx，可以看到是内存中的那堆乱七八糟不明所以的东西

管他呢ヽ(ﾟ _ ゝ)ノdump下来跟硬编码数组异或就是了
得到password

输入程序即可得到flag

再往下跟就是生成flag的过程：



是将Password和下一行值异或得到的

```
n = [160, 27, 134, 92, 202, 202, 239, 42, 143, 223, 25, 167, 6, 81, 169, 90]
x = [0xE8, 0x7E, 0xB7, 0x6D, 0xA5, 0x95, 0x98, 0x1A, 0xFD, 0xEE, 0x7D, 0xF8, 0x34, 0x62, 0x9A, 0x69]
print("Password: ", end='')
for i in range(16):
    print(chr(n[i] ^ x[i]), end='')

print()

n = [ord(x) for x in "He11o_w0r1d_2333"]
x = [0x2F, 0x0A, 0x7D, 0x50, 0x01, 0x38, 0x28, 0x67, 0x43, 0x45, 0x0C, 0x00, 0x47, 0x43, 0x6B, 0x12]
print("Flag: flag{", end='')
for i in range(16):
    print(chr(n[i] ^ x[i]), end='')
print("}")
```

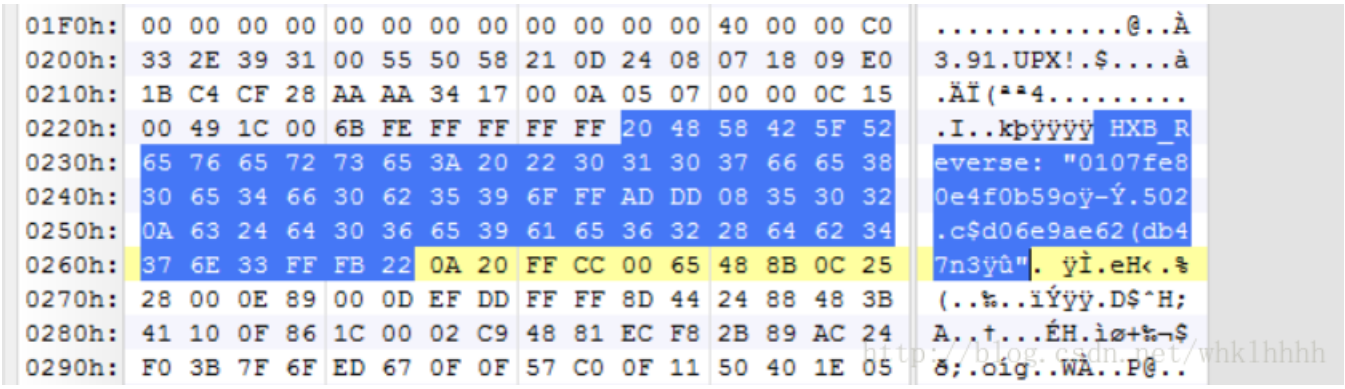
后记

- 那堆乱七八糟的数据是哪来的？

其实是文件的结尾，倒数0x30的字节
中间看到的CreateFileW、GetFileTypeA和ReadFile都是提示，在读取自身的内容
因此如果脱下来壳却没有Patch的话会导致输入什么都不对（我就对着Password和0异或要等于一个不可见字符懵逼了许久），也算一种自校验吧

- 明明是UPX为什么用程序拖不掉呢？

用编辑器打开程序，在UPX0区段的开头可以看到



HXB_Reverse这个字符串很明显是后加的，导致校验和错误从而无法脱壳
将其改为Go build ID即可正常脱壳

同时，Go build ID也正是Golang语言的标识
Golang语言作为编译型语言，生成的exe也是由汇编对应的机器码组成，因此IDA同样可以反编译

- 那么为什么IDA没有识别出来函数呢？

因为Golang语言在所有函数之间加入了0xCC（INT 3）来填充间隔，IDA只认识乖乖用00来分隔的C语言，哪见过这种阵仗啊，于是就起到了花指令的作用，全盘懵逼了
因此需要手工自己按C来帮助IDA识别间隔0xCC和真正的代码
以后学了IDA的脚本也可以自动创建函数

参考：https://www.40huo.cn/blog/huxiangbei-writeup.html

C. 明日计划
JarvisOJ-ima