

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](#)

版权声明: 本文为博主原创文章, 遵循 [CC 4.0 BY-SA](#) 版权协议, 转载请附上原文出处链接和本声明。

本文链接: <https://blog.csdn.net/whklhyyy/article/details/78917878>

版权



[CTF 专栏收录该内容](#)

163 篇文章 4 订阅

订阅专栏

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

401E0	0C F1 4D 00	00 00 00 00	13 00 00 00	00 00 00 00	.fm.....
401F0	00 00 00 00	00 00 00 00	06 00 00 00	00 00 00 00
40200	43 72 65 61	74 65 46 69	6C 65 57 00	00 00 00 00	CreateFile.....
40210	47 65 74 46	69 6C 65 54	79 70 65 00	00 00 00 00	GetFileType.....
40220	52 65 61 64	46 69 6C 65	00 31 00 00	00 00 00 00	ReadFile.1.....
40230	43 6C 6F 73	65 48 61 6E	64 6C 65 00	00 00 00 00	CloseHandle.....
40240	40 03 04 42	C0 00 00 00	10 00 00 00	00 00 00 00	@.BA.....
40250	40 D4 4D 00	00 00 00 00	0A 00 00 00	00 00 00 00	@OM.....
40260	50 61 73 73	77 6F 72 64	3A 20 00 00	00 00 00 00	Password:.....
40270	57 72 69 74	65 43 6F 6E	73 6F 6C 65	57 00 00 00	WriteConsoleW...
40280	A0 C2 54 00	00 00 00 00	E0 C2 54 00	00 00 00 00	AT.....AT.....
40290	47 65 74 41	43 50 00 00	31 00 00 00	31 00 00 00	GetACP.1..1...
402A0	31 32 33 34	35 36 37 38	32 00 32 00	32 00 00 00	123456782.2.2...
402B0	33 00 33 00	33 00 00 00	34 00 34 00	34 00 00 00	3.3.3...4.4.4...
402C0	35 00 35 00	35 00 00 00	36 00 36 00	36 00 00 00	5.5.5...6.6.6...
402D0	37 00 37 00	37 00 00 00	38 00 38 00	38 00 00 00	7.7.7...8.8.8...

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

00000000004015F6	48 89 AC 24 88 00 00	mov qword ptr ss:[rsp+88],rbp	
00000000004015FE	48 8D AC 24 88 00 00	lea rbp,qword ptr ss:[rsp+88]	
0000000000401606	48 8B 84 24 98 00 00	mov rax,qword ptr ss:[rsp+98]	[rsp+98]:&"1234567890123456"
000000000040160E	48 8B 48 08	mov rcx,qword ptr ds:[rax+8]	取长度
0000000000401612	48 8B 10	mov rdx,qword ptr ds:[rax]	[rax]:"1234567890123456"
0000000000401615	48 83 F9 10	cmp rcx,10	判断是否为16
0000000000401619	0F 8C 02 03 00 00	jl shell.401921	
000000000040161F	48 8B 9C 24 A8 00 00	mov rbx,qword ptr ss:[rsp+A8]	
0000000000401627	48 8B B4 24 B0 00 00	mov rsi,qword ptr ss:[rsp+B0]	
000000000040162F	31 FF	xor edi,edi	
0000000000401631	48 83 FF 10	cmp rdi,10	
0000000000401635	7D 3C	jge shell.401673	
0000000000401637	48 39 CF	cmp rdi,rcx	
000000000040163A	0F 83 DA 02 00 00	jae shell.40191A	

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

```
31 ((void (__fastcall *)(__int64, __int64, __int64, __int64))qword_453290[136])(v10, a2, a3, a4);
32 }
33 len = v29[1];
34 v12 = *v29;
35 if ( len < 16 )
36 {
37     v27 = "Access Denied!CertCloseStoreCreateProcessWCryptGenRandomFindFirstFileWFormatMessageWGC assist waitGC worker initGetConsole
38     v28 = 14i64;
39     result = sub_404290();
40 }
41 else
42 {
43     i = 0i64;
44     do
45     {
46         if ( i >= len || i >= v31 )
47             sub_427BA0();
48         if ( (*(unsigned __int8 *) (v12 + i) ^ (unsigned __int64) * (unsigned __int8 *) (v30 + i)) != qword_541C40[i] ) // 异或比较
49         {
50             v25 = "Access Denied!CertCloseStoreCreateProcessWCryptGenRandomFindFirstFileWFormatMessageWGC assist waitGC worker initGetCor
51             v26 = 14i64;
52             return sub_404290();
53         }
54         ++i;
55     }
56     while ( (signed __int64) i < 16 );
57     sub_43BF00((__int64) &unk_48BF40, v12);
58     v14 = 0i64;
```

<http://blog.csdn.net/whklhxxx>

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这传参机制真迷，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

r10=E8 'è'
r8=31 '1'
UPX0:000000000040165E shell.exe:\$165E #0

地址	十六进制	ASCII
000000C0420ECBE0	E8 7E B7 6D A5 95 98 1A FD EE 7D F8 34 62 9A 69	e--m¥...y1}o4b.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	yuAbGV>..EA..¥Ae
000000C0420ECC10	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

<http://blog.csdn.net/whklhxxx>

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、(_) 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:

The screenshot shows a debugger window with assembly code on the left and a memory dump on the right. The assembly code includes instructions like 'jae Shell.4018C7', 'movzx r9d,byte ptr ds:[r10+r8]', and 'lea rsi,qword ptr ds:[4CD2C0]'. The memory dump shows hex and ASCII values, including 'http://blog.csdn.net/whklh'.

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 "Hello_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

01F0h:	00 00 00 00	00 00 00 00	00 00 00 00	40 00 00 C0@..À
0200h:	33 2E 39 31	00 55 50 58	21 0D 24 08	07 18 09 E0	3.91.UPX!.\$....à
0210h:	1B C4 CF 28	AA AA 34 17	00 0A 05 07	00 00 0C 15	.Äï(==4.....
0220h:	00 49 1C 00	6B FE FF FF	FF FF 20 48	58 42 5F 52	.I..kpyyy HXB_R
0230h:	65 76 65 72	73 65 3A 20	22 30 31 30	37 66 65 38	reverse: "0107fe8
0240h:	30 65 34 66	30 62 35 39	6F FF AD DD	08 35 30 32	0e4f0b59oÿ-Ý.502
0250h:	0A 63 24 64	30 36 65 39	61 65 36 32	28 64 62 34	.c\$d06e9ae62 (db4
0260h:	37 6E 33 FF	FB 22 0A 20	FF CC 00 65	48 8B 0C 25	7n3ÿù". yÿ.eH<.%
0270h:	28 00 0E 89	00 0D EF DD	FF FF 8D 44	24 88 48 3B	(.%.iÿÿ.DS^H;
0280h:	41 10 0F 86	1C 00 02 C9	48 81 EC F8	2B 89 AC 24	A..†...ÉH.iø+%-S
0290h:	F0 3B 7F 6F	ED 67 0F 0F	57 C0 0F 11	50 40 1E 05	8;.oig..WA..Pè..

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