

看雪2017CTF第二题解法

转载

weixin_30947043 于 2018-06-30 13:09:00 发布 112 收藏
原文链接: <http://www.cnblogs.com/galano/p/9246766.html>
版权

• 1.前言

去年看了一下,两个Check函数验证RegCode,但是这两个校验是矛盾的。算了一下发现算不出,就放到一边没动过了。今天听朋友讲到溢出攻击,顿时想起了这个CrackMe,拿来练练手想来也是极好的。

• 2.试探

```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main proc near ; CODE XREF: start+AF1p
.text:00401000     push    offset aCrackmeForCtf2 ; "\n Crackme for CTF2017 @PediY.\n"
.text:00401005     call    sub_413D42
.text:0040100A     add     esp, 4
.text:0040100D     mov     dword_41B034, 2
.text:00401017     call    sub_401050
.text:0040101C     call    sub_401090
.text:00401021     call    sub_4010E0
.text:00401026     mov     eax, dword_41B034
.text:0040102B     test    eax, eax
.text:0040102D     jnz    short loc_40103F
.text:0040102F     push    offset aYouGetIt ; "You get it!\n"
.text:00401034     call    sub_413D42
.text:00401039     add     esp, 4
.text:0040103C     xor     eax, eax
.text:0040103E     retn
.text:0040103F
```

Check函数

这里标注了一个全局验证变量,初始值为2,后面跟着两个Check函数,每通过一次校验,则把验证变量递减1,最后判断验证变量值是否为0,为0则通过两层验证,不为0则失败。但是前面已经说过了,这两个Check是互斥的。感兴趣的同学可以进去研究一下这两个call,其实就是一个多元方程组。

第一个call sub_401050。这个call是输入RegCode的。进入看看代码。

```
.text:00401050
.text:00401050 sub esp, 0Ch
.text:00401053     push    offset aCodedByFpc_ ; " Coded by Fpc.\n\n"
.text:00401058     call    sub_413D42
.text:0040105D     add     esp, 4
.text:00401060     push    offset aPleaseInputYou ; " Please input your code: "
.text:00401065     call    sub_413D42
.text:0040106A     add     esp, 4
.text:0040106D     lea    eax, [esp+0Ch+var_C]
.text:00401071     push    eax
.text:00401072     push    offset aS ; "%s"
.text:00401077     call    _scanf
.text:0040107C     lea    eax, [esp+14h+var_C]
.text:00401080     add     esp, 14h
.text:00401083     retn
.text:00401083 sub_401050 endp
.text:00401083
```

很明显溢出点就在这里。程序从堆栈里面分配了12个字节作为输入缓冲区,而且调用的是不安全的scanf,且scanf的format参数里面只指定了一个%s。很明显,可以直接在这里进行溢出攻击。

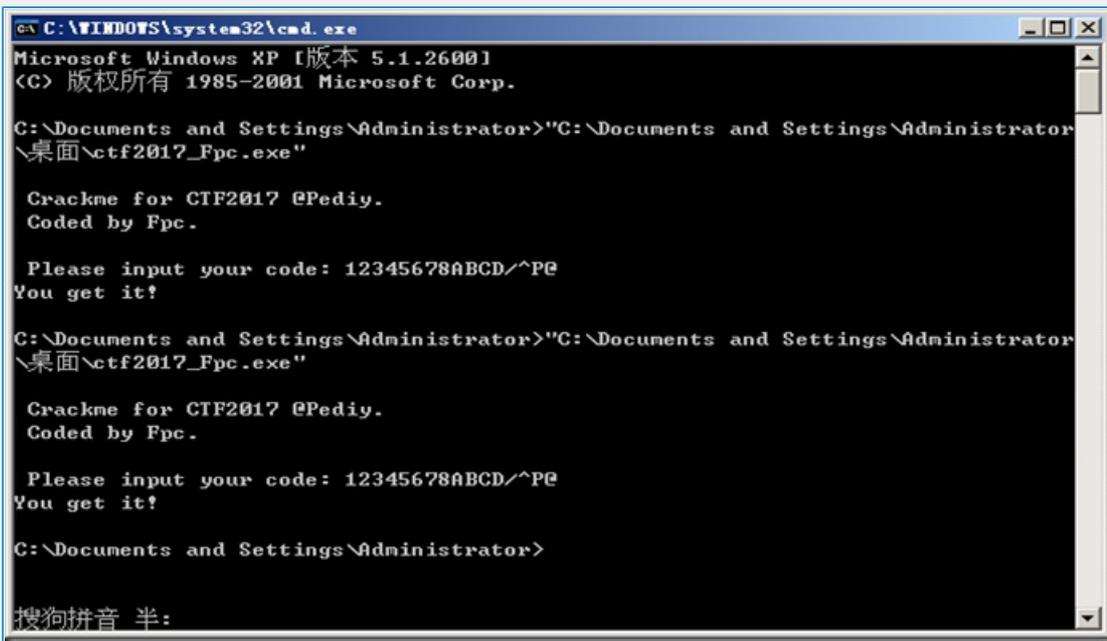
• 3.第一次溢出

```

.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main      proc near          ; CODE XREF: start+AF.jp
.text:00401000          push   offset aCrackmeForCtf2 ; "\n Crackme for CTF2017 @
.text:00401005          call   sub_413D42
.text:0040100A          add    esp, 4
.text:0040100D          mov    dword_41B034, 2
.text:00401017          call   sub_401050
.text:0040101C          call   sub_401090
.text:00401021          call   sub_4010E0
.text:00401026          mov    eax, dword_41B034
.text:0040102B          test   eax, eax
.text:0040102D          jnz    short loc_40103F
.text:0040102F          push   offset aYouGetIt ; "You get it!\n"
.text:00401034          call   sub_413D42
.text:00401039          add    esp, 4
.text:0040103C          xor    eax, eax
.text:0040103E          retn
.text:0040103F ; -----
.text:0040103F

```

我们只要能溢出攻击，让程序跳转到0x40102F就成功了。



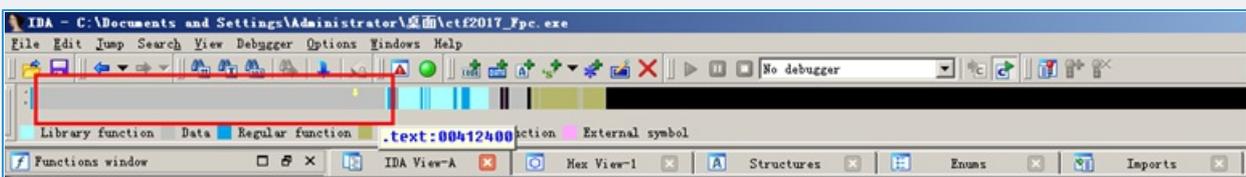
40102F -> /@

实际上0x10这个ASCII码对应的字符是没办法输入的，我们只能用复制粘贴的形式。

第一次溢出攻击成功。可问题是，如果这样算答案的话，那么答案是不会只有一组解的。

• 4.第2次溢出

再找找看有没有其它攻击点。



首先IDA提示，这里有一大段未知数据，感觉很可疑，就进去看了下。

```

.text:00413131          db  83h, 0C4h, 0F0h
.text:00413134          dd  20712A70h, 0F1C75F2h, 28741C71h, 2E0671DDh, 870F574h
.text:00413138          dd  76E47440h, 00C46700h, 0E074C00h, 00024107h, 0E574E77h

```

```

.text:00413134 dd 6983740Fh, 0EB75EB70h, 0DF7069h, 22712C70h, 0B8261F7Dh
.text:00413134 dd 2B741E71h, 3E067169h, 870F57Ch, 7CF17169h, 00C197002h
.text:00413134 dd 41B034A3h, 75E77400h, 0E571DC12h, 7C0CF271h, 0E9706903h
.text:00413134 dd 6965E97Dh, 70B8DC70h, 3E1D7127h, 710F1971h, 0DD257019h
.text:00413134 dd 0F6700571h, 71DD0870h, 700270F2h, 70580F14h, 0F1171ECh
.text:00413134 dd 0F671EA71h, 0DD03700Fh, 0ED71ED70h, 0FE1700Dh, 7F36217Eh
.text:00413134 dd 671A7D27h, 1D2A74B8h, 65690D7Eh, 67C067Fh, 1D361C7Eh
.text:00413134 dd 8BDC0E7Fh, 75EA74C8h, 7E69DC14h, 0C1F47FEh, 0F97CFB7Fh
.text:00413134 dd 0EA7DE27Fh, 0D87E6965h, 77207688h, 2E1A7F27h, 0DD29788Bh
.text:00413134 dd 778D0076h, 67EF207h, 0DD261876h, 58B80E77h, 1479EB78h
.text:00413134 dd 768DB865h, 0FF477EFh, 0F97EFB77h, 0EA7FE177h, 0B8D9768Dh
.text:00413134 dd 73F22372h, 1C756729h, 0DD2C740Fh, 66690E72h, 6740673h
.text:00413134 dd 0DD361E72h, 0DD261073h, 0E974D888h, 12751575h, 73ED72DCh
.text:00413134 dd 0FB730FF3h, 0E073F974h, 6966E875h, 740FD672h, 2E1D7527h
.text:00413134 dd 75DC1973h, 0DD267C19h, 742E0475h, 0F3751D08h, 16740272h
.text:00413134 dd 0ED7C58C1h, 0C1F3137Dh, 0F575EA75h, 1D03720Fh, 0EC73EC74h
.text:00413134 dd 0DF741D66h, 0F23EBDC, 0EB227585h, 85261DFAh, 74D08B29h
.text:00413134 dd 0E9F65E918h, 75D08B55h, 22F2E85Ch, 0E0754025h, 62F6F258h

```

这里会不会是可执行的代码。打开LordPE看看。

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	00018598	00001000	00019000	E0000020
.rdata	0001A000	000008D2	0001A000	00001000	40000040
.data	0001B000	00003EA8	0001B000	00003000	C0000040

这段数据全部在.text区块里面，意思是可执行喽。

而且这段数据的始地址就是0x413131，转换成字符就是11A。可输入字符，又一次加深了我的怀疑。这次尝试溢出到这个地方。

```

C:\WINDOWS\system32\cmd.exe
Coded by Ppc.

Please input your code: 12345678ABCD/^PE
You get it!

C:\Documents and Settings\Administrator>"C:\Documents and Settings\Administrator\桌面\ctf2017_Fpc.exe"

Crackme for CTF2017 @Pedy.
Coded by Ppc.

Please input your code: 12345678ABCD/^PE
You get it!

C:\Documents and Settings\Administrator>"C:\Documents and Settings\Administrator\桌面\ctf2017_Fpc.exe"

Crackme for CTF2017 @Pedy.
Coded by Ppc.

Please input your code: 12345678ABCD11A
Bad register-code, keep trying.

C:\Documents and Settings\Administrator>
搜狗拼音 半:

```

很神奇，居然执行了校验代码。看来真正的验证在这里，前面的两个Check就是金蝉脱壳之计。

而前面输入的12个字节的RegCode，应该也是有实际意义的。

打开IDA的Trace功能，记录一下执行代码。

```

1 00000C04
2 00000C04 .text:00413131 add esp, 0FFFFFF0h ESP=0012FF74 CF=1 PF=1 AF=0
3 00000C04 .text:00413134 je short near ptr dword_413134+2Ch
4 00000C04 .text:dword_413134+2 jno short near ptr dword_413134+24h
5 00000C04 .text:dword_413134+24 jno short near ptr dword_413134+0Dh
6 00000C04 .text:dword_413134+D jno short near ptr dword_413134+15h
7 00000C04 .text:dword_413134+15 jno short near ptr dword_413134+8h
8 00000C04 .text:dword_413134+8 jno short near ptr dword_413134+26h
9 00000C04 .text:dword_413134+26 jno short near ptr dword_413134+1Ch
10 00000C04 .text:dword_413134+1C mov eax, eax EAX=00000000 CF=0 ZF=1
11 00000C04 .text:dword_413134+1E je short near ptr dword_413134+0Ah
12 00000C04 .text:dword_413134+A jz short near ptr dword_413134+34h

```

```

13 00000C04 .text:dword_413134+34 jno short near ptr dword_413134+62h
14 00000C04 .text:dword_413134+36 jno short near ptr dword_413134+5Ah
15 00000C04 .text:dword_413134+5A jno short near ptr dword_413134+41h
16 00000C04 .text:dword_413134+41 jno short near ptr dword_413134+49h
17 00000C04 .text:dword_413134+49 jno short near ptr dword_413134+3Ch
18 00000C04 .text:dword_413134+3C jno short near ptr dword_413134+5Ch
19 00000C04 .text:dword_413134+5C jno short near ptr dword_413134+50h
20 00000C04 .text:dword_413134+50 mov dword_418034, eax
21 00000C04 .text:dword_413134+55 jz short near ptr dword_413134+3Eh
22 00000C04 .text:dword_413134+3E jz short near ptr dword_413134+68h
23 00000C04 .text:dword_413134+68 jse short near ptr dword_413134+94h
24 00000C04 .text:dword_413134+6D jno short near ptr dword_413134+8Ch
25 00000C04 .text:dword_413134+8C jno short near ptr dword_413134+78h
26 00000C04 .text:dword_413134+78 jno short near ptr dword_413134+7Fh
27 00000C04 .text:dword_413134+7F jno short near ptr dword_413134+73h
28 00000C04 .text:dword_413134+73 jno short near ptr dword_413134+8Eh
29 00000C04 .text:dword_413134+8E jno short near ptr dword_413134+86h
30 00000C04 .text:dword_413134+86 pop eax EAX=34333231 ESP=0012FF78
31 00000C04 .text:dword_413134+87 jno short near ptr dword_413134+75h
32 00000C04 .text:dword_413134+88 jno short near ptr dword_413134+8Bh

```

很明显，这是一段经过混淆的代码。代码的空间局部性被打乱了。不过这种程度的混淆，也没有必要特意写个去混淆的脚本。因为Trace记录天生的优势就在于能清楚代码的控制流，这种乱序完全可以无视的。

下面是我摘录的经过处理的重点代码。

```

00000C04 .text:dword_413134+86 pop eax EAX=34333231 ESP=0012FF78

00000C04 .text:dword_413134+B7 mov ecx, eax ECX=34333231

00000C04 .text:dword_413134+EB pop eax EAX=38373635 ESP=0012FF7C

00000C04 .text:dword_413134+120 mov ebx, eax EBX=38373635

00000C04 .text:dword_413134+155 pop eax EAX=44434241 ESP=0012FF80

00000C04 .text:dword_413134+181 mov edx, eax EDX=44434241

00000C04 .text:dword_413134+179 mov edx, eax

00000C04 .text:dword_413134+1AE mov eax, ecx EAX=34333231

00000C04 .text:dword_413134+1E2 sub eax, ebx EAX=FBFBFBFC CF=1 AF=1
ZF=0 SF=1

00000C04 .text:dword_413134+215 shl eax, 2 EAX=EFEFEFF0

00000C04 .text:dword_413134+24C add eax, ecx EAX=24232221 AF=0 SF=0

00000C04 .text:dword_413134+281 add eax, edx EAX=68666462 CF=0 PF=0

00000C04 .text:dword_413134+2B5 sub eax, 0EAF917E2h EAX=7D6D4C80 CF=1

00000C04 .text:dword_413934+1EA pop eax EAX=00413131 ESP=0012FF84

00000C04 .text:dword_413934+21A xor eax, 1210Eh EAX=0040103F CF=0 PF=1

```

从这里我们可以得到第一个方程：

$$(K1 - K2) * 4 + K1 + K3 = 0EAF917E2h$$

K1: 1-4

K2: 5-8

K3: 9-12

我们把RegCode分成3段，用K1、K2、K3表示。后同。

很明显，这里得到的还是一个多解方程。因为有3个未知数，所以方程应该有3组。而第一个恒等式没有成立，应该就是导致了后面的没有执行。

经过不断调整，我得到了第二个方程：

```
00413455  03C1      add  eax,ecx
00413489  2BC3      sub  eax,ebx
004134BF  8BD8      mov  ebx,eax
004134F3  D1E0      shl  eax,1
00413525  . 03C3     add  eax,ebx
00413559  03C1      add  eax,ecx
0041358F  > 8BC8     mov  ecx,eax
004135C3  . 03C2     add  eax,edx
004135F7  > 2D C808F5E8  sub  eax,0xE8F508C8

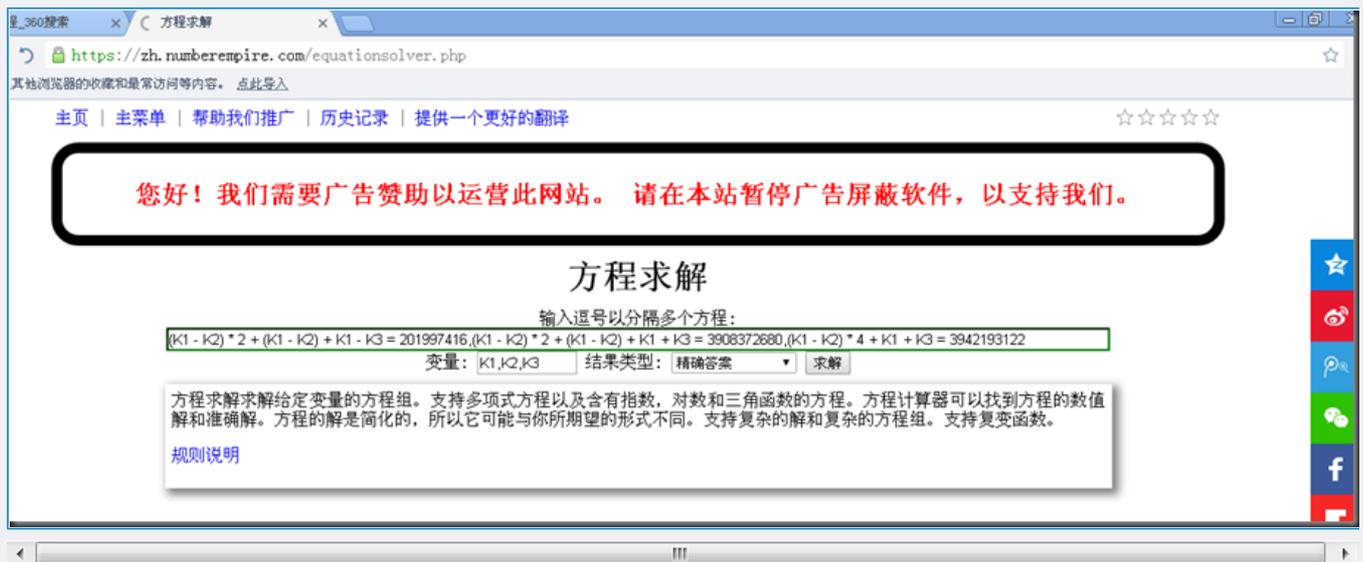
(K1 - K2) * 2 + (K1 - K2) + K1 + K3 = 0E8F508C8h
```

第三个方程：

```
00413665  > /8BC1    mov  eax,ecx
004136A7  . 2BC2     sub  eax,edx
004136D8  2D 683C0A0C  sub  eax,0xC0A3C68

(K1 - K2) * 2 + (K1 - K2) + K1 - K3 = 0C0A3C68h
```

得出方程组后，理论这些方程是有唯一解的。我在网上找了个在线解方程的网站。



K1=1953723722,K2=1919903280,K3=1853187632

K1 = 07473754Ah

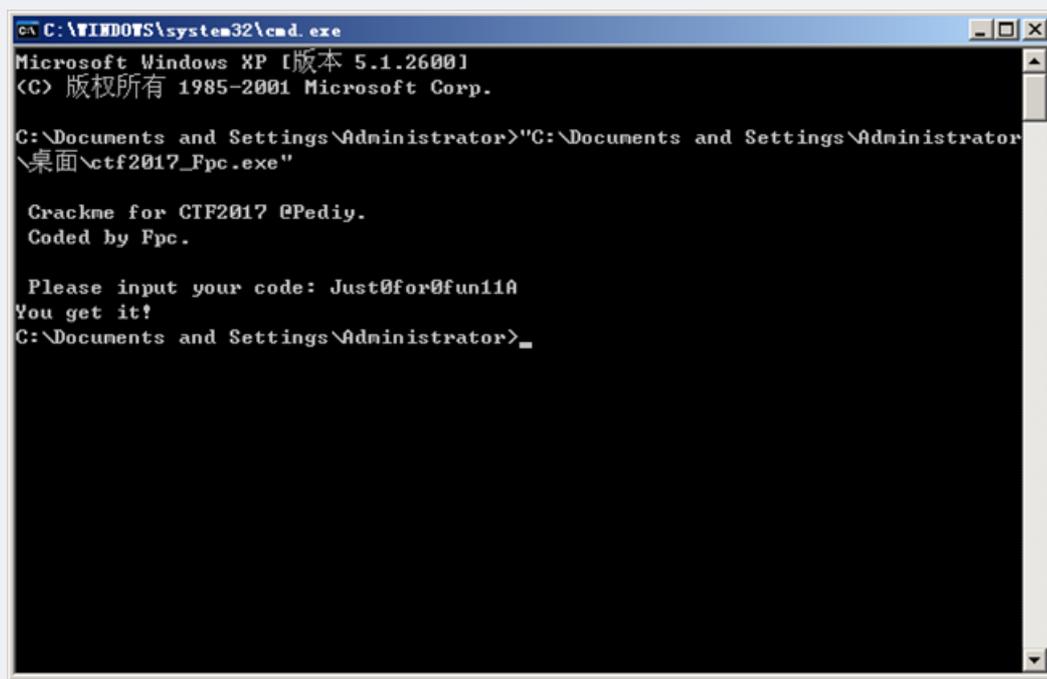
K2 = 0726F6630h

K3 = 06E756630h

K1 + K2 + K3 -> Just0for0fun

所以，答案应该是：

Just0for0fun11A



- 5.CrackMe下载

转载于:<https://www.cnblogs.com/galano/p/9246766.html>