

2017看雪秋季赛 第二题

原创

Flying Fatty 于 2017-10-29 19:07:54 发布 290 收藏

分类专栏: CTF之旅 reverse

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/kevin66654/article/details/78387101>

版权



CTF之旅 同时被 2 个专栏收录

84 篇文章 2 订阅

订阅专栏



reverse

24 篇文章 0 订阅

订阅专栏

[先贴一发官方题解](#)

这个题很有意思: 以漏洞利用的方式来做reverse。

整体思路是这样的: 查找字符串, 看到了“有用”的, 单步跟踪。main中有三个函数分别是: 50, 90和E0。50是处理输入的, 90是check1, E0是check2。check函数是4个方程, 得出的是无解, 所以必须找其他地方。在IDA查找, 发现了413131的一堆乱码, 50里面有缓冲区溢出的漏洞, 可以控制EIP跳转到413131。里面是一堆看不懂的花指令, 去花的思想很简单: 只要找到对我们有用的好, 无用的跳转直接F8下一句就好了。

下面来解释细节。

```
; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near
push    offset aCrackmeForCtf2 ; "\n Crackme For CTF2017 @Pediy.\n"
call    printf
add    esp, 4
mov    dword_41B034, 2
call    sub_401050
call    sub_401090
call    sub_4010E0
mov    eax, dword_41B034
test   eax, eax
jnz    short loc_40103F
```

http://blog.csdn.net/kevin66654

```
push    offset aYouGetIt ; "You get it!\n"
call    printf
add    esp, 4
xor    eax, eax
ret
```

```
loc_40103F:           ; "Bad register-code, keep trying.\n"
push    offset aBadRegisterCod
call    printf
add    esp, 4
xor    eax, eax
ret
_main endp
```

看到main的几个call, 401050是输入, 401090和4010E0是check。单步调试发现算法如下:

```

def get(s):
    s = (s + 0x100000000) % 0x100000000
    return s

#if get(EAX * 5 + ECX) == 0x8F503A42
#if get(EAX * 0xD + EDX) == 0xEF503A42
#if get(EAX * 0x11 + ECX) == 0xF3A94883
#endif get(EAX * 0x7 + EDX) == 0x33A94883

```

想当然的，觉得解方程组完毕就出结果。以为是个大暴力水题，但是用数学方法算算：第三个式子 - 第一个式子：左边是EAX * 12 + 0x100000000 * k == 一个奇数。这个方程组是无解的。自己做的时候，做到了这里，就开始乱试了。

因为做题的时候，默认了是8位。

只有7位时，前面是00

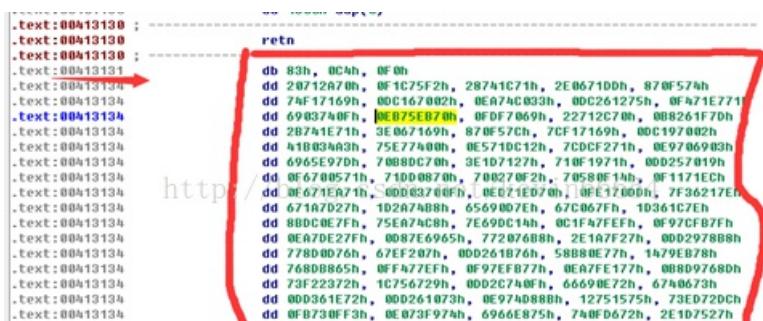
只有6位时，前面是7F00

只有5位时，前面是7FFD00

只有4位时，前面是7FFDE000

8-12位的时候，后面是忽略的。12位之后会报错，显然是有缓冲区溢出。

之后的，就是学到的东西。查看IDA，拖一拖汇编代码，会看到这一堆东西。



看401050的栈！

00401050 | \$ 83EC 0C | sub esp, 0xC

这几天复习了 esp 和 ebp，以及堆栈的使用套路。+)

这里 esp 减去了 12，栈里的空间就是 12 个。所以超过 12 个了，覆盖的就是返回地址+)

+)

0018FF2C	0041B08C	ASCII "%s"
0018FF30	0018FF34	
0018FF34	00413D8A	ctf2017_.<ModuleEntryPoint>
0018FF38	7FFDE000	
0018FF3C	00401000	返回到 ctf2017_.00401000 来自 http://www.csdn.net/kevin88654

注意 scanf 的参数规则，scanf ("%s", string) +)

这里，我们“正常的数据范围”就是从 0018FF30 – 0018FF3C。12 个字符长度+)

+)

0018FF2C	0041B08C	ASCII "%s"
0018FF30	0018FF34	ASCII "abcdefghijkl11A"
0018FF34	64636261	
0018FF38	68676665	
0018FF3C	6C6B6A69	
0018FF40	00413131	ctf2017_.00413131

00413131，都是可以输入的字符啊。11A的三个字符。那么我们可以控制EIP到这儿来。前面12个任意字符，11A为13-15个字符，单步调试到这儿。一脸懵逼。为什么会有这种东西。

00413131	83C4 F0	add esp, -0x10
00413134	v 70 2A	jo short ctf2017_-00413160
00413136	v 71 20	jno short ctf2017_-00413158
00413138	f2:75 1c	repne jne short 00413157
0041313B	0F	db 0F
0041313C	v 71 1C	jno short ctf2017_-0041315A
0041313E	v 74 28	je short ctf2017_-00413168
00413140	DD71 06	fsave (108-byte) ptr ds:[ecx+0x6]
00413143	2e:74 F5	bhnt je short 0041313b
00413146	v 70 08	jo short ctf2017_-00413150
00413148	6971 F1 740270	imul esi, dword ptr ds:[ecx-0xF], 0x16700274
0041314F	DC33	fdiv qword ptr ds:[ebx]
00413151	C074EA 75 12	sal byte ptr ds:[edx+ebp*8+0x75], 0x12
00413156	26:DC71 E7	fdiv qword ptr es:[ecx-0x19]
0041315A	^ 71 F4	jno short ctf2017_-00413150

想到在IDA的那一堆乱码，估计是花指令。忽略它，直接单步，可以在EAX, EBX, ECX, EDX里面看到我们的输入。这里有个小技巧：输入的时候有点规律，比如abcdefgijkl11A。好处是，每个字符不一样，而且是有顺序的，在寄存器里很明显可以看得出来。

得找到有用的判断，然后计算就好↓

004131BA | 58 | pop eax ↓

EAX 64636261 ↓

这里的 pop eax，取的是前 4 个字符↓

EAX 68676665
ECX 64636261 ↓

看到了一个 mov ecx, eax，然后又 pop eax↓

↓

00413316 | 11 2BC3 b1og_csd| sub eax, ebx 66654

到了这里开始有计算了↓

0x64636261 - 0x68676665↓

EAX FBFBFBFC
ECX 64636261
EDX 6C6B6A69
EBX 68676665 ↓

00413349 | C1E0 02 | shl eax, 0x2 ↓

00413380 | 03C1 | add eax, ecx ↓

把这些有用的记下来，跟着运算。在判断的时候，在右侧的EAX, EBX的寄存器把值改成需要的值，继续单步（不然就跳转到错误了）

得到了三个等式，三元一次方程组。

```
...
(A - B) * 4 + A + C = EAF917E2
(A - B) * 3 + A + C = E8F508C8
(A - B) * 3 + A - C = 0C0A3C68
...
def hextonumber(x):
#1234567890abcdef
#1234567890ABCDEF
if x>='0' and x<='9':
    return int(x)
elif x>='A' and x<='F':
    return ord(x)-55
else:
    return ord(x)-87

def hextoflag(s):
#word='666c61677b7769656e65725f61747461636b5f61747461636b5f796f757d'
#flag{wiener_attack_attack_you}
flag = ''
i = 0
```

```

while (i<len(s)):
    flag += chr(hextonumber(s[i])*16+hextonumber(s[i+1]))
    i += 2
return flag
#return s[2:].decode('hex')

C = (0xE8F508C8 - 0x0C0A3C68) / 2
A = -3 * (0xEAF917E2 - C) + 2 * (0xE8F508C8 + 0x0C0A3C68)
B = A - (0x0C0A3C68 + C - A) / 3
print hex(A),hex(B),hex(C)
s1 = str(hex(A))[2:-1]
s2 = str(hex(B))[2:-1]
s3 = str(hex(C))[2:-1]
print s1,s2,s3
s = hextoflag(s1)[::-1] + hextoflag(s2)[::-1] + hextoflag(s3)[::-1]
print s
...
A = 64636261
B = 68676665

FBFBFBFC = 64636261 - 68676665
EFEFEFF0 = FBFBFBFC * 4
54535251 = EFEFEFF0 + 64636261
C0BEBCBA = 54535251 + 6C6B6A69

004133E9 sub eax, 0xEAF917E2

(A - B) * 4 + A + C = EAF917E2
(A - B) * 3 + A + C = E8F508C8
(A - B) * 3 + A - C = 0C0A3C68

EAX D5C5A4D8
3A290739 = EAX + 64636261
D1C1A0D4 = EAX - 68676665
A38341A8 = D1C1A0D4 * 2
7544E27C = D1C1A0D4 + A38341A8
D9A844DD = 7544E27C + 64636261
4613AF46 = D9A844DD + 6C6B6A69

sub eax, 0xE8F508C8
...

```

于是，得到了flag。

从题解里学习到了一个没见过的高级姿势。

sympy

一个数学工具，方便解方程组。其他的分析文章也有很多值得学习的地方。

贴个样例代码

```
from sympy import *
x = Symbol('x')
y = Symbol('y')
z = Symbol('z')
print (solve([x+y+z-3,x+y-z-1,x-y-z+1],[x,y,z]))
```

解多元方程组的tools